Étude d'un *buffer overflow*

Un grand nombre de failles de sécurités sont induites par la flexibilité du C dans sa gestion des variables et la demande au programmeur de gérer lui même la mémoire. L'absence de typage fort des variables permet d'en écraser le contenu lors d'erreurs de gestion de la mémoire, souvent associées à la manipulation de chaînes de caractères. Nous nous proposons d'étudier quelques mécanismes de manipulation de la mémoire (*buffer overflow*) afin de comprendre la gestion de la pile sur un PC à base de processeur Intel 32 bits, sans aller jusqu'à l'injection de code néfaste pour le bon fonctionnement de l'ordinateur. Nous analyserons le passage de paramètres et l'organisation de la pile sous GNU/Linux, avec des exécutables compilés par gcc 4.2.4

Tous les codes sources sont disponibles à `http://jmfriedt.free.fr/exam`. Sous GNU/Linux, la compilation des programmes s'obtient par `gcc -o programme programme.c`. La génération du code assembleur à partir de l'exécutable s'obtient par `objdump -d programme > programme.asm`.

1 Valeur des variables

Nous savons que la pile sert au stockage des variables et au passage de paramètres lors de l'appel de fonctions, ainsi qu'au stockage d'informations nécessaire au bon fonctionnement du processeur en fin d'appel à une fonction. Une compréhension précise du fonctionnement et de l'occupation est donc fondamentale – qu'il s'agisse de microcontrôleurs programmés en assembleur ou en C directement, ou de processus fonctionnant sur système d'exploitation. Nous nous intéresserons ici à la pile d'un programme développé en C sous GNU/Linux.

Code source :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void hello()
5  {char l=4;
6   int j=0;
7   char *i;
8   int k=2;
9
10  printf("%d %d\n",sizeof(j),j);
11
12  i=(char*)(&j);
13  sprintf(i,"hop");
14
15  printf("j: %08x %s i: %08x %08x \n",j,&j,*(int*)i,*i);
16  printf("     l=%x\n",l);
17
18  sprintf(i,"hop1234");
19  printf("k=%x l=%x\n",k,l);
20
21  sprintf(i,"hop12345");
22  printf("k=%x l=%x\n",k,l);   // '5'=0x35
23 }
24
25 int main()
26 {int x;
27  hello();
28 }
```

Résultat de l'exécution du programme :

```
4 0
j: 00706f68 hop i: 00706f68 00000068
    l=4
k=2 l=0
k=2 l=35
```

Document associé : table ASCII en hexadécimal (gauche) et décimal (droite)

	2	3	4	5	6	7
0:		0	@	P		p
1:	!	1	A	Q	a	q
2:	"	2	B	R	b	r
3:	#	3	C	S	c	s
4:	$	4	D	T	d	t
5:	%	5	E	U	e	u
6:	&	6	F	V	f	v
7:		7	G	W	g	w
8:	(	8	H	X	h	x
9:	)	9	I	Y	i	y
A:	*	:	J	Z	j	z
B:	+	;	K	[	k	{
C:	,	<	L	\	l	|
D:	-	=	M	]	m	}
E:	.	>	N	^	n	~
F:	/	?	O	_	o	DEL

	30	40	50	60	70	80	90	100	110	120
0:		(	2	<	F	P	Z	d	n	x
1:		)	3	=	G	Q	[	e	o	y
2:		*	4	>	H	R	\	f	p	z
3:	!	+	5	?	I	S	]	g	q	{
4:	"	,	6	@	J	T	^	h	r	|
5:	#	-	7	A	K	U	_	i	s	}
6:	$	.	8	B	L	V		j	t	~
7:	%	/	9	C	M	W	a	k	u	DEL
8:	&	0	:	D	N	X	b	l	v	
9:		1	;	E	O	Y	c	m	w	

Questions :

1. quelle est la taille d'un entier (`int`) sur une architecture intel 32 bits ?
2. interpréter la sortie du programme fourni ci-dessus (ligne commençant par `j:`). En particulier, fournir l'ordre de stockage des chaînes de caractères et des entiers en mémoire lors de l'exécution de la fonction `hello()`.
3. expliquer l'évolution de la variabe `l` dans `hello()`.
4. en déduire l'organisation de la pile lors de l'appel à la fonction `hello()`.

2 Résultat de calcul inattendu

Code source :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void hello2(int a,int b,int c)
5  {char tt[12]="hello world";
6   int *i;
7   int k;
8   i=&k;
9   for (k=0;k<10;k++)
10       printf(" %d %08x: %x\n",k,i+k,*(i+k));
11  (*(i+6))+=7;
12 }
13
14 int main()
15 {int x;
16  x=1;
17  hello2(1,2,3);
18  x=2;
19  printf("fini x=%x\n",x);
20 }
```

Résultat :

```
 0 bfe52604: 0
 1 bfe52608: 6c6c6568
 2 bfe5260c: 6f77206f
 3 bfe52610: 646c72
 4 bfe52614: bfe52604
 5 bfe52618: bfe52648
 6 bfe5261c: 804842a
 7 bfe52620: 1
 8 bfe52624: 2
 9 bfe52628: 3
fini x=1
```

Questions :

En plus de contenir les variables et arguments fournis aux fonctions, la pile contient des informations permettant au processeur de poursuivre son exécution après l'appel aux fonctions (instruction `CALL` en assembleur)

1. lors de l'appel à une fonction susceptible d'écraser le contenu de registres, quelle méhode efficace permet d'en mémoriser le contenu pour une utilisation ultérieure ? Illustrer avec le code assembleur fourni ci-dessous.
2. interpréter la sortie de ce programme : que signifient les valeurs affichées dans la fonction `hello2()`. Comment avons nous choisi l'adresse de départ des données affichées ?
3. interpréter en particulier le contenu du 7ème élément (k=6) affiché, et le comparer aux informations disponibles dans le code assembleur ci-dessous.
4. commenter l'évolution de `x` fournie dans `main`.

On pourra s'aider pour ces interprétations du code assembleur (présenté en trois colonnes : adresse mémoire, code machine et mnémonique assembleur) issu du désassemblage de la fonction `main()` de l'éxécutable, fourni ci-dessous.

```
08048374 <hello2>:
8048374: 55                    push   %ebp
8048375: 89 e5                 mov    %esp,%ebp
8048377: 83 ec 38              sub    $0x38,%esp
804837a: a1 1e 85 04 08        mov    0x804851e,%eax
804837f: 89 45 f0              mov    %eax,-0x10(%ebp)
8048382: a1 22 85 04 08        mov    0x8048522,%eax
8048387: 89 45 f4              mov    %eax,-0xc(%ebp)
804838a: a1 26 85 04 08        mov    0x8048526,%eax
804838f: 89 45 f8              mov    %eax,-0x8(%ebp)
8048392: 8d 45 ec              lea    -0x14(%ebp),%eax
8048395: 89 45 fc              mov    %eax,-0x4(%ebp)
8048398: c7 45 ec 00 00 00 00  movl   $0x0,-0x14(%ebp)
804839f: eb 38                 jmp    80483d9 <hello2+0x65>
80483a1: 8b 45 ec              mov    -0x14(%ebp),%eax
80483a4: c1 e0 02              shl    $0x2,%eax
80483a7: 03 45 fc              add    -0x4(%ebp),%eax
80483aa: 8b 08                 mov    (%eax),%ecx
80483ac: 8b 45 ec              mov    -0x14(%ebp),%eax
80483af: c1 e0 02              shl    $0x2,%eax
80483b2: 03 45 fc              add    -0x4(%ebp),%eax
80483b5: 8b 55 ec              mov    -0x14(%ebp),%edx
80483b8: 89 4c 24 0c           mov    %ecx,0xc(%esp)
80483bc: 89 44 24 08           mov    %eax,0x8(%esp)
80483c0: 89 54 24 04           mov    %edx,0x4(%esp)
80483c4: c7 04 24 10 85 04 08  movl   $0x8048510,(%esp)
80483cb: e8 08 ff ff ff        call   80482d8 <printf@plt>
80483d0: 8b 45 ec              mov    -0x14(%ebp),%eax
80483d3: 83 c0 01              add    $0x1,%eax
80483d6: 89 45 ec              mov    %eax,-0x14(%ebp)
80483d9: 8b 45 ec              mov    -0x14(%ebp),%eax
80483dc: 83 f8 09              cmp    $0x9,%eax
80483df: 7e c0                 jle    80483a1 <hello2+0x2d>
80483e1: 8b 55 fc              mov    -0x4(%ebp),%edx
80483e4: 83 c2 18              add    $0x18,%edx
80483e7: 8b 45 fc              mov    -0x4(%ebp),%eax
80483ea: 83 c0 18              add    $0x18,%eax
80483ed: 8b 00                 mov    (%eax),%eax
80483ef: 83 c0 07              add    $0x7,%eax
80483f2: 89 02                 mov    %eax,(%edx)
80483f4: c9                    leave
80483f5: c3                    ret

080483f6 <main>:
80483f6: 8d 4c 24 04           lea    0x4(%esp),%ecx
80483fa: 83 e4 f0              and    $0xfffffff0,%esp
80483fd: ff 71 fc              pushl  -0x4(%ecx)
8048400: 55                    push   %ebp
8048401: 89 e5                 mov    %esp,%ebp
8048403: 51                    push   %ecx
8048404: 83 ec 24              sub    $0x24,%esp
8048407: c7 45 f8 01 00 00 00  movl   $0x1,-0x8(%ebp)
804840e: c7 44 24 08 03 00 00  movl   $0x3,0x8(%esp)
8048415: 00
8048416: c7 44 24 04 02 00 00  movl   $0x2,0x4(%esp)
804841d: 00
804841e: c7 04 24 01 00 00 00  movl   $0x1,(%esp)
8048425: e8 4a ff ff ff        call   8048374 <hello2>
804842a: c7 45 f8 02 00 00 00  movl   $0x2,-0x8(%ebp)
8048431: 8b 45 f8              mov    -0x8(%ebp),%eax
8048434: 89 44 24 04           mov    %eax,0x4(%esp)
8048438: c7 04 24 2a 85 04 08  movl   $0x804852a,(%esp)
804843f: e8 94 fe ff ff        call   80482d8 <printf@plt>
8048444: 83 c4 24              add    $0x24,%esp
8048447: 59                    pop    %ecx
8048448: 5d                    pop    %ebp
8048449: 8d 61 fc              lea    -0x4(%ecx),%esp
804844c: c3                    ret
804844d: 90                    nop
804844e: 90                    nop
804844f: 90                    nop
```

3 Appel d'une fonction

Code source :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  void hello3();
6  int main();
7
8  char essai[] = {0xb8,0x00,0x00,0x00,0x00,0xff,0xe0,0x90,0x90,0x90,0x90,\
9  0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,  \
10 0x90,0x90,0x90,0x00};
11
12 void hello2(int a,int b,int c)
13 {char tt[12]={'\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0'};
14  int *i;
15  int k;
16  i=&k;
17
18  for (k=0;k<10;k++)
19       printf("␣%d:␣%08x␣%08x\n",k,i+k,*(i+k));
20
21  *(int*)&essai[1]=(void*)hello3;
22  *(int*)&essai[12]=&k;
23  *(int*)&essai[20]=tt;
24  *(int*)&essai[24]=*(i+6);
25
26  strcpy(tt,essai);
27
28  printf("^␣avant␣^␣-----␣v␣apres␣v\n");
29  for (k=0;k<10;k++)
30       printf("␣%d:␣%08x␣%08x\n",k,i+k,*(i+k));
31 }
32
33 void hello3()
34 {printf("hello3␣c'est␣sympa\n");}
35
36 int main()
37 {int x;
38  printf("main:␣␣␣%x\n",(void*)main);
39  printf("hello2:␣%x\n",(void*)hello2);
40  printf("hello3:␣%x\n",(void*)hello3);
41  hello2(1,2,3);
42  printf("Retour␣dans␣le␣main\n");
43  exit(0);
44 }
```

Résultat de l'exécution du programme :

```
main:   8048533
hello2: 8048414
hello3: 804851f
 0: bfcccd24 00000000
 1: bfcccd28 00000000
 2: bfcccd2c 00000000
 3: bfcccd30 00000000
 4: bfcccd34 bfcccd24
 5: bfcccd38 bfcccd68
 6: bfcccd3c 0804859c
 7: bfcccd40 00000001
 8: bfcccd44 00000002
 9: bfcccd48 00000003
^ avant ^ ----- v apres v
 0: bfcccd24 00000000
 1: bfcccd28 04851fb8
 2: bfcccd2c 90e0ff08
 3: bfcccd30 90909090
 4: bfcccd34 bfcccd24
 5: bfcccd38 90909090
 6: bfcccd3c bfcccd28
 7: bfcccd40 0804859c
 8: bfcccd44 00000000
 9: bfcccd48 00000003
hello3 c'est sympa
Retour dans le main
```

Sans relation directe avec cet exemple, voici un bout de code contenant de l'assembleur et les opcodes associés, permettant de stocker dans un registre la destination d'un saut, puis d'aller en cet emplacement mémoire. La compréhension de ce code pourra aider à l'interprétation du contenu de `essai`.

```
b8 00 00 00 00          mov    $0x0,%eax
ff e0                   jmp    *%eax
```

Questions :

1. analyser l'organisation de la pile de `hello2()` compte tenu de la sortie de ce programme. Que signifient les 3 premières lignes fournies par `main()` ?
2. le principal danger de la manipulation de la pile tient en la possibilité d'exécuter du code non-prévu par le programmeur initialement. Commenter le contenu du tableau de caractères `essai`.
3. la fontion `strcpy()` copie le contenu d'un tableau de caractères dans un autr, et ce jusqu'à atteindre le caractère 0 (fin de chaˆ ine). Notre utilisation de la fonction `strcpy()` simule l'entrée par un utilisateur lorsqu'un programme demande une entrée. Commenter l'affectation de `essai` dans `tt` tel que pourrait le faire cet utilisateur ?
4. Commenter la sortie de ce programme. Comment avons nous fait pour atteindre ce résultat ?

Références

[1] Aleph One, *Smashing the Stack for Fun and Profit*, Phrack **49** (1996), disponible à `www.phrack.org/issues.html?id=14&issue=49`

[2] Mixter, *Writing buffer overflow exploits – a tutorial for beginners*, disponible à `mixter.void.ru/exploit.html`

4 Informations additionnelles

```
STRCPY(3)                 Linux Programmers Manual                 STRCPY(3)

NAME
       strcpy, strncpy - copy a string

SYNOPSIS
       #include <string.h>
       char *strcpy(char *dest, const char *src);
       char *strncpy(char *dest, const char *src, size_t n);

DESCRIPTION
       The  strcpy()  function  copies the string pointed to by src, including
       the terminating null byte (\0), to the buffer  pointed  to  by  dest.
       The  strings  may  not overlap, and the destination string dest must be
       large enough to receive the copy.

       The strncpy() function is similar, except that at most n bytes  of  src
       are  copied.  Warning: If there is no null byte among the first n bytes
       of src, the string placed in dest will not be null terminated.

       If the length of src is less than n, strncpy() pads  the  remainder  of
       dest with null bytes.

       A simple implementation of strncpy() might be:

           char*
           strncpy(char *dest, const char *src, size_t n){
               size_t i;

               for (i = 0 ; i < n && src[i] != \0 ; i++)
                   dest[i] = src[i];
               for ( ; i < n ; i++)
                   dest[i] = \0;

               return dest;
           }

RETURN VALUE
       The  strcpy()  and strncpy() functions return a pointer to the destina
       tion string dest.

CONFORMING TO
       SVr4, 4.3BSD, C89, C99.

NOTES
       Some programmers consider strncpy() to be inefficient and error  prone.
       If  the  programmer knows (i.e., includes code to test!)  that the size
       of dest is greater than the length of src, then strcpy() can be used.

       If there is no terminating null byte in the first n characters of  src,
       strncpy()  produces  an unterminated string in dest.  Programmers often
       prevent this mistake by forcing termination as follows:

           strncpy(buf, str, n);
           if (n > 0)
               buf[n - 1]= \0;

BUGS
       If the destination string of a strcpy() is not large enough  (that  is,
       if  the  programmer  was  stupid  or lazy, and failed to check the size
       before copying) then anything might happen.  Overflowing  fixed  length
       strings is a favorite cracker technique.

SEE ALSO
       bcopy(3), memccpy(3), memcpy(3), memmove(3), wcscpy(3), wcsncpy(3)

COLOPHON
       This  page  is  part of release 2.79 of the Linux man-pages project.  A
       description of the project, and information about reporting  bugs,  can
       be found at http://www.kernel.org/doc/man-pages/.

GNU                               2007-06-15                         STRCPY(3)

PRINTF(3)                 Linux Programmers Manual                 PRINTF(3)
```

```
NAME
       printf,  fprintf,  sprintf,  snprintf,  vprintf,  vfprintf,  vsprintf,
       vsnprintf - formatted output conversion

SYNOPSIS
       #include <stdio.h>

       int printf(const char *format, ...);
       int fprintf(FILE *stream, const char *format, ...);
       int sprintf(char *str, const char *format, ...);
       int snprintf(char *str, size_t size, const char *format, ...);

       #include <stdarg.h>

       int vprintf(const char *format, va_list ap);
       int vfprintf(FILE *stream, const char *format, va_list ap);
       int vsprintf(char *str, const char *format, va_list ap);
       int vsnprintf(char *str, size_t size, const char *format, va_list ap);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       snprintf(), vsnprintf(): _BSD_SOURCE || _XOPEN_SOURCE >= 500 ||
       _ISOC99_SOURCE; or cc -std=c99

DESCRIPTION
       The functions in the printf() family produce output according to a for
       mat as described below.  The functions  printf()  and  vprintf()  write
       output  to stdout, the standard output stream; fprintf() and vfprintf()
       write  output  to  the  given  output  stream;  sprintf(),  snprintf(),
       vsprintf() and vsnprintf() write to the character string str.

       The  functions  snprintf()  and  vsnprintf()  write  at most size bytes
       (including the trailing null byte (\0)) to str.

       The functions vprintf(), vfprintf(), vsprintf(), vsnprintf() are equiv
       alent  to  the  functions  printf(),  fprintf(), sprintf(), snprintf(),
       respectively, except that they are called with a va_list instead  of  a
       variable  number  of arguments.  These functions do not call the va_end
       macro.  Because they invoke the va_arg macro, the value of ap is  unde
       fined after the call.  See stdarg(3).

       These  eight  functions  write the output under the control of a format
       string that specifies how subsequent arguments (or  arguments  accessed
       via the variable-length argument facilities of stdarg(3)) are converted
       for output.

   Return value
       Upon successful return, these functions return the number of characters
       printed  (not  including  the  trailing  \0  used  to  end  output to
       strings).

       The functions snprintf() and vsnprintf() do not write  more  than  size
       bytes  (including  the trailing \0).  If the output was truncated due
       to this limit then the return value is the number  of  characters  (not
       including the trailing \0) which would have been written to the final
       string if enough space had been available.  Thus,  a  return  value  of
       size  or  more  means  that  the output was truncated.  (See also below
       under NOTES.)

       If an output error is encountered, a negative value is returned.

   Format of the format string
       The format string is a character string, beginning and  ending  in  its
       initial  shift state, if any.  The format string is composed of zero or
       more  directives:  ordinary  characters  (not  %),  which  are   copied
       unchanged  to the output stream; and conversion specifications, each of
       which results in fetching zero or more subsequent arguments.  Each con
       version specification is introduced by the character %, and ends with a
       conversion specifier.  In between there may be (in this order) zero  or
       more  flags, an optional minimum field width, an optional precision and
       an optional length modifier.

...
```

Correction

1 Valeur des variables

Réponses :

1. un entier défini par `int` est codé sur 4 octets quelle que soit l'architecture du processeur. Nous vérifions cette affirmation au moyen de l'instruction `sizeof(j)` où `j` est un `int`.
2. l'objectif de ce programme est de montrer que la notion de type n'est pas stricte en C mais correspond simplement à de la réservation de mémoire sur la pile : la façon dont cette mémoire est utilisée dépend de la rigueur avec laquelle le programme l'exploite. Dans cet exemple, une variable `j` de type entier correspond à la réservation de 4 octets. La variable `i` est un pointeur vers l'adresse de j (`ligne i=(char *)(&j);`). Presque toutes les erreurs de gestion de mémoire en C sont associées à une gestion de chaîne de caractères : la chaîne de caractère est définie comme un tableau de `char`. Nous avons donc le droit d'utiliser `i` pour y stocker une chaîne de caractères, même si `i` pointe sur l'espace mémoire d'un entier (`j`). Ainsi, l'instruction `sprintf(i,"hop");` place dans l'espace mémoire dédié à `i` la chaîne de caractères `hop`. On peut le vérifier en affichant sous forme de chaîne de caractères (`%s`) et de valeur hexadécimale (`%x`) le contenu de `j` (même si cette variable doit contenir un entier, nous avons le droit de l'interpréter comme une chaîne de caractères). Nous vérifions que `j` contient la chaîne de caractères `hop` ou, en format hexadécimal, la chaîne de caractères ASCII correspondant finissant par l'octet de valeur 0, indication de la fin de chaîne en C. L'affichage sous forme hexadécimale du contenu de `j` semble "à l'envers" car l'architecture intel est *little endian*. On vérifie par ailleurs que le contenu de `i` est bien le même que celui de `j`, sous forme de 4 octets quand on interprète `i` comme contenant un `int`, et sous forme de un seul octet lorsque le contenu de `i` est interprété comme un `char`, son type par défaut.

   Les problèmes de gestion de mémoire apparaissent lorsque le programmeur tente de placer dans une variable une donnée plus volumineuse que l'espace alloué à cette variable. Si nous écrivons `hop1234` dans `i`, nous sommes en train de placer une chaîne de 7 caractères dans une variable de 4 octets. Les éléments en excès de la chaîne de caractères sont placés en mémoire sur la pile, et vont *affecter le contenu des variables adjacentes. Ce point est fondamental pour tout ce qui va suivre.*
3. `l` était initialisée à 4, tel que nous l'avons vérifié en ligne 16. Après avoir affecté `i` avec une chaîne plus longue que l'espace alloué pour `j`, nous avons modifié la valeur de `l` qui vaut maintenant 0, alors que `k` a été inchangée. Cela signifie que dans l'organisation de la pile, l'écriture dans l'adresse de `j` se poursuit vers le "haut" de la pile en direction de `l`. `l` est un char, mais l'espace mémoire sur architecture intel 32 bits ne peut se faire que par multiples de 4 octets : un unique char occupe donc 4 octets en mémoire. C'est pourquoi une occupation abusive de la mémoire par l'assignation de `j` avec une chaîne de caractères trop longue peut passer inaperçue : 3 octets inutilisés sont placés entre `j` et `l`. Ce n'est qu'en écrivant 4 octets de trop (ne pas oublier le 0 de fin de chaîne de caractère après `hop1234` que nous affectons `l`, en y plaçant justement ce 0 de fin de chaîne.

   Ce dernier point est validé en placant une chaîne encore plus longue : cette fois l'élément qui va écraser le contenu de `l` n'est plus le 0 de fin de chaîne mais le caractère 5, de code ASCII 35 en hexadécimal. `k`, qui se trouve plus "bas" dans la pile que `j`, n'est jamais affectée par ces opérations.
4. on en déduit que la pile est organisée de la façon suivante : 4 octets pour `k` en bas de la pile, 4 octets pour le pointeur `i`, 4 octets pour `j`, et 4 octets pour `l` – tout en haut de la pile – dont seul le dernier élément est utilisé par le type `char`

2 Résultat de calcul inattendu

Réponses :

1. la pile sert de brouillon lors de l'appel à une fonction, qu'il s'agisse d'y stocker des registres exploités par le processeur (*program pointer* pour se rappeler de l'adresse à laquelle retourner en fin d'exécution de la routine, ou pointeur de pile temporaire) ou de registres que l'utilisateur veut retrouver en revenant d'une routine. L'empilement et le dépilement d'éléments sont des méthodes rapides pour le stockage de données temporaires ne nécessitant pas de gestion explicite de la mémoire. Dans le code assembleur joint, on voit par exemple au début de `hello2()` un `push %ebp` qui contient des informations nécessaires lors du retour (information récupérée par l'instruction `leave`). De même dans le `main()`, on trouve deux couples `push/pop` aux adresses 80483f3 et 804843b, ou 80483f6 et 804843a.
2. le programme affiche le contenu de la pile par paquets de 4 octets, puisque `i` est un pointeur sur un entier de cette taille. `i` pointe initialement sur la variable `k` qui est placée *a priori* le plus "bas" sur la pile d'après la syntaxe de notre programme. Nous allons donc afficher le contenu de la pile en la "remontant", *i.e.* en passant l'ensemble des adresses contenant des variables (deux entiers de 4 octets et un tableau de 12 octets, soit au total 20 octets ou 5 entiers), en allant volontairement *au delà* de la zone de stockage des variables pour observer les informations stockées sur la pile par le système. Chaque ligne présente l'indice de l'élément dans la pile, son adresse (multiple de 4) et le contenu.

   En commençant par la fin, on constate déjà que le passage d'arguments à une fonction (les 3 entiers `a`, `b` et `c`) se fait par la pile (éléments 7 à 9 de la sortie du programme). On trouve dans les emplacements 1 à 3 la chaîne de caractères `hello world` terminant par un 0 tel qu'on pourra le vérifier en consultant la correspondance avec le code ASCII. L'adresse 0 pointe vers `k` qui nous sert de compteur : le contenu de `k` est bien égal à 0 lors de la première itération de la boucle de la ligne 7. Plus surprenant, nous constatons que le pointeur `i` se trouve "plus haut" que `tt` (alors que la syntaxe de notre programme demandait à placer `i` entre `tt` et `k`). Nous vérifions cette affirmation en comparant le contenu de `i` avec les adresses affichées dans la seconde colonne : l'adresse `bfe52604` de `k` est bien contenue dans l'élément numéro 4 qui est donc l'emplacement de `i`.

   On en déduit l'organisation de la pile :
   4 octets pour `k`, 12 octets pour `tt`, 4 octets pour `i`, puis 2 entiers dont nous discuterons plus bas et finalement les arguments de la fonction (3 entiers contenant 1, 2 et 3).
3. Les contenus des emplacement d'indice 5 et 6 sont des variables du système nécessaires pour savoir comment quitter `hello2` et retourner dans `main()` après l'appel à la fonction. Le contenu de 5 est le pointeur de pile de la fonction `main()`, mais plus important est le contenu du sixième élément qui est *l'adresse de retour* à laquelle sautera le *program counter* lorsqu'on quitte `hello2`. Cette valeur est fondamentale pour la suite de notre discussion : on vérifie dans le code assembleur joint que 804842A est bien l'adresse qui suit l'appel à la fonction `hello2()` dans `main()` (instruction qui suit `call hello2`).
4. `x` est initialisée à 1 au début du `main()`, et redéfinie à 2 juste après l'appel à la fonction `hello2()`. Or dans `hello2()` nous manipulons volontairement la pile en modifiant le contenu du sixième élément, dont nous venons de voir qu'il contient l'adresse de retour. En ajoutant 7 à cette adresse de retour, nous constatons dans le code assembleur que nous sautons l'instruction movl $0x02 ... dans le `main()` qui est l'instruction définissant la nouvelle valeur de `x`. En conséquent, `x` a conservé sa valeur initiale et n'a pas été modifiée à 2 comme nous pourrions nous y attendre.

3 Appel d'une fonction

Réponses :

1. la méthode d'analyse du contenu de la pile s'effectue comme dans les deux exemples précédents. On constate à nouveau que l'organisation est – comme dans l'exemple précédent – différent de l'ordre des variables défini dans le programme. Le tableau de caractères `tt` contient 12 fois la valeur 0 : au niveau de la pile, l'élément 0 contient `k` (qui vaut 0 à la première itération de la boucle), puis le tableau `tt` avec ses 3 entiers, puis `i` qui contient l'adresse de `k`, et on retrouve ensuite les éléments empilés par le système (*stack* et *program pointers*) et les arguments de la fonction.

   Les 3 premières lignes contiennent les adresses en mémoire des fonctions. On constate que le sixième élément de la pile contient une adresse se trouvant après l'adresse de `main()` : il s'agit bien d'une adresse de retour après l'appel à `hello2()` dans `main()`.

   Cette analyse de la pile est réitérée après les manipulations des lignes 21 à 27.

2. le tableau `tt` a une taille supérieure à 12 octets. Ce tableau contient un certain nombre d'informations, commençant notamment par la valeur `0xb8` suivie de 4 octets, suivis de `0xff 0xe0`. Il s'agit là de données qui peuvent aussi être interprétées comme des opcodes exécutables par le processeur : *la distinction entre donnée et instruction n'existe par pour un processeur*[1]. Ainsi, le contenu des `essai` peut s'interpréter comme les opcodes des instructions `mov $0x00,%eax` puis `jmp *%eax`, ou placer dans un registre une adresse à laquelle on va ensuite sauter. La ligne 21 du programme remplacer l'adresse de destination initialisée à 0x00000000 par l'adresse de la fonction `hello3`. L'exécution du contenu de `essai` va donc nous faire sauter dans cette fonction.

3. les manipulations de chaînes de caractères en C sont de simples manipulations d'octets, sans vérification des bornes ou du contenu des tableaux dans la version la plus simple (et communément disponible) de ces fonctions. Ainsi, `strcpy()` se contente d'une copie octet par octet du second argument dans le premier jusqu'à atteindre la valeur 0 signifiant la fin d'une chaîne de caractères. Bien que souvent optimisée en assembleur pour des utilisations embarquées, une implémentation possibe [2] est

   ```
   char *strcpy(char *dest,const char *src)
   {const char *p;
    for (p=src,q=dest;*p!='\0';p++,q++) *q=*p;
    *q='\0';
    return(dest);
   }
   ```

   Ici, `essai` est copié dans `tt`, même si la taille de `essai` dépasse largement ce que peut contenir `tt` : il s'agit d'une méthode classique de manipulation de la pile, exploitée ici pour exécuter un code qui n'avait pas été prévu par le programmeur initial de la fonction `hello2()`.

4. en plus de manipuler le contenu de `essai` pour y inclure des opcodes et argument pour sauter dans `hello3()`, nous avons affecté son contenu pour modifier l'adresse de retour de `hello2()` (élément 6 de la pile) et préparé cette pile pour supporter le fait d'avoir appelé une fonction de plus que prévu (ligne 24 qui décale le pointeur de pile nécessaire au retour dans `main()`).

   Le déroulement du programme est donc le suivant :
   - la fonction `main()` appelle `hello2()`
   - un utilisateur fournit en entrée de `hello2()` une chaîne de caractères simulée par `essai`, qui est copiée dans `tt`. La faille de sécurité vient de ce que la taille de `tt` est inférieure à la taille de `essai` (inutile d'augmenter la taille de `tt` pour se prémunir de cette attaque : on peut toujours faire croître `essai` pour dépasser `tt`). Nous modifions la pile pour que *l'adresse de retour de la fonction* `hello2()` *pointe vers le tableau* `tt`. De cette façon, à la fin de l'exécution de `hello2()`, le programme exécute le contenu de `tt` au lieu de retourner dans `main()`. Ici, le contenu de `tt` a été défini pour appeler `hello3()`.
   - `hello2()` est exécutée normalement, et lors de l'instruction de retour (instruction assembleur `leave`), l'adresse de retour est dépilée, pointe sur `tt` qui donne l'ordre de sauter dans `hello3()`, qui s'exécute en affichant son message
   - en quittant `hello3()`, le contenu de la pile est dépilée et nous retournons dans le `main()` (manipulations de la piles aux lignes 23 et 24) pour achever normalement l'exécution du programme.

   Nous observons donc un message en fin d'exécution du programme qui n'était pas prévu dans la séquence des opérations de `main()` puisqu'aucun appel explicite à `hello3()` n'y était fait.

Remarques

Diverses méthodes permettent de se protéger de ce type d'attaque :
- le choix du langage : contrairement au C qui ne gère la mémoire que comme une suite d'octets, des langages fortement typés ou gérant eux même leur mémoire vérifient les bornes lors de l'accès aux tableaux (avec une perte de performances associée)

1. Cette propriété permet par exemple du code automodifiable, exécutable sur les architectures de processeur ne possédant par de cache
2. `http://en.wikipedia.org/wiki/Strcpy`

- utilisation de librairies pour le C avec vérfication des bornes : des fonctions telles que `strlcpy()` et `strncpy()` visent à remplacer `strcpy()` (avec une perte de performances associée)
- au niveau de l'exécution, rendre aléatoire l'espace mémoire afin de ne plus pouvoir prédire l'emplacement des variables et des fonctions.

Programme commenté

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  void hello3();
6  int main();
7
8  // chaine de caracteres se terminant par un 0
9  char essai[] = {0xb8,0x00,0x00,0x00,0x00,0xff,0xe0,0x90,0x90,0x90,0x90, \
10                 0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90, \
11                 0x90,0x90,0x90,0x90,0x90,0x00};
12
13 void hello2(int a,int b,int c)
14 {char tt[12]={'\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0'};
15  int *i;
16  int k;
17  i=&k;
18
19  for (k=0;k<10;k++)
20      printf("␣%d:␣%08x␣%08x\n",k,i+k,*(i+k));
21
22  // Mise en place de la pile !
23  // On fixe l'argument du mov $0x0, %eax a celle de hello3
24  *(int*)&essai[1]=(void*)hello3;
25  // Comme juste apres on va faire une copie ayant pour destination le tableau
26  // tt de taille 12, et que l'on va y copier 28 octets, on va un peu casser la
27  // pile ... en particulier la valeur de la variable i qui se trouve placee
28  // apres tt
29  // (gcc version 4.3.2 (Debian 4.3.2-1)) & gcc version 4.2.4 (Debian 4.2.4-1)
30  // On positionne donc la valeur correcte de la variable i (&k) pour que le
31  // memcpy ne casse pas tout ...
32  *(int*)&essai[12]=&k;
33  // On reecrit la valeur de l'adresse de retour pour pointer au debut du
34  // buffer tt dans la pile. A la sortie de la fonction hello2 on va donc
35  // executer le contenu du shellcode contenu dans tt (donc dans essai).
36  // Celui-ci saute (jmp) betement vers la fonction hello3
37  *(int*)&essai[20]=tt;
38  // On reecrit la valeur de l'adresse de retour pour la fonction hello3. On
39  // utilise pour cela la valeur legitime de hello2. On va donc a la fin de
40  // hello3 revenir l'a o'u aurait du revenir hello2, c'est-a-dire dans la
41  // fonction main !
42  *(int*)&essai[24]=*(i+6);
43
44  // On ecrase tout !
45  // memcpy(tt,essai,28);
46  // sprintf(tt,"%s",essai);
47  strcpy(tt,essai);
48
49 // *(i+7)=*(i+6);*(i+6)=tt;
50
51  printf("^␣avant␣^␣-------␣v␣apres␣v\n");
52  for (k=0;k<10;k++)
53      printf("␣%d:␣%08x␣%08x\n",k,i+k,*(i+k));
54 }
55
56 void hello3()
57 {
58  printf("hello3␣c'est␣sympa\n");
59 }
60
61 int main()
62 {int x;
63  printf("main:␣␣␣%x\n",(void*)main);
64  printf("hello2:␣%x\n",(void*)hello2);
65  printf("hello3:␣%x\n",(void*)hello3);
66  hello2(1,2,3);
67  printf("Retour␣dans␣le␣main\n");
68  exit(0);
69 }
```