

ENPM691 Homework Assignment 8

Hacking of C programs and Unix Binaries-Fall 2024 Icollier

Dipam Bhattacharyya, MEng Cybersecurity dipamb2@umd.edu

Abstract—The assignment aims to hijack the Global Offset Table and Procedure Linking Table through entry modification. It will attempt to run shell code through system calls.

Index Terms—homework, hacking, C, coding

I. INTRODUCTION

The Procedure Linkage Table (PLT) and Global Offset Table (GOT) are sections within an Executable and Linkable Format (ELF) file that play a significant role in dynamic linking. The purpose of dynamic linking is to reduce the size of binaries by allowing them to rely on system libraries, such as the C standard library (libc), to provide the majority of their functionality.

For example, an ELF file does not include its own version of the 'printf' function compiled within it. Instead, it dynamically links to the 'printf' function of the system it is running on. In addition to smaller binary sizes, this also means that users can upgrade their libraries without having to download all the binaries again each time a new version is released.

In this experiment, through GOT 'printf' entry modification, an attempt will be made to call libc system. This function will then take 'printf' arguments and execute binaries.

II. METHODOLOGY

A. Host environment and architecture

[1] The experiment was done on a Ubuntu v16.04 VM with Windows x64 as the host PC. All exploit protection settings in Windows are default, including ASLR(default ON). To check the VM architecture, the "lscpu" terminal command was given:

```
user@user-VirtualBox: ~$ lscpu
Last login: Thu Sep 19 10:58:38 2024 from 192.168.42.1
user@user-VirtualBox:~$ lscpu
Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 2
On-Line CPU(s) list: 0,1
Thread(s) per core: 1
Core(s) per socket: 1
Socket(s): 2
Vendor ID: GenuineIntel
CPU Family: 6
Model: 186
Model name: 13th Gen Intel(R) Core(TM) i7-13620H
CPU MHz: 2918.400
CPU-MHz: 5836.80
Hypervisor vendor: VMware
Virtualization type: full
L1d cache: 48K
L1i cache: 32K
L2 cache: 1280K
L3 cache: 24576K
Flags: fpu vme de pse tsc mtr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2
ssse3 x2apic movbe popcnt aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch ssbd ibrs ibpb stibp fsgsba
e tsc_adjust bmi1 avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt clwb sha_ni xsaveopt xsavec xgetbv1 arat rdpid
mclear_flush_lid arch_capabilities
user@user-VirtualBox:~$
```

Fig. 1. Ubuntu CPU information

B. Working of PLT and GOT

The concept of PLT is to have the memory address of various system functions to be referenced at runtime by the compiler. When these are called within the code, it first checks the entry in PLT. PLT calls the GOT for that entry and if found it returns the address of the system function to jump and execute.

On the other hand, if not found, GOT returns to PLT which then calls the dynamic linker to resolve the actual address of the function. The linker will update the entry in the GOT.

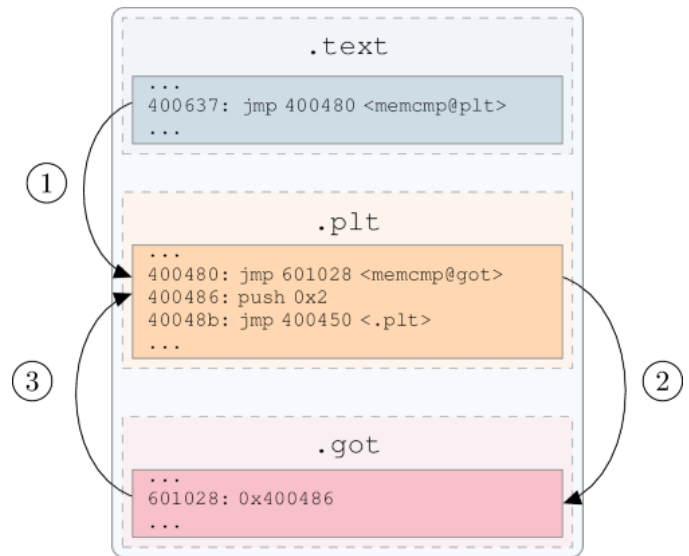


Fig. 2. Referencing PLT/GOT when memcmp() is called

C. GDB debugging

[2] After the generation of the executable, we are attaching gdb to it. Then, we are looking at both source and disassembled code together while also having terminal access.

(gdb) layout split (1)

To check the stack frames(in reverse order of being invoked) used in the program:

(gdb) bt (2)

And to inspect a particular frame:

(gdb) info frame 1 (3)

To check the contents of memory from a particular point, say pointer or absolute address, the following(See Fig.4) is

```

hackgot.c
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<string.h>
4
5  int main(int argc, char **argv){
6
7      char buf[32];
8      gets(buf);
9      printf("Buffer entered is: %d bytes\n", strlen(buf));
10     puts(buf);
11
12     return 0;
13 }

0x804847b <main>      push    %ebp
0x804847c <main+1>    mov     %esp,%ebp
0x804847e <main+3>    sub     $0x20,%esp
0x8048481 <main+6>    lea     -0x20(%ebp),%eax
0x8048484 <main+9>    push    %eax
0x8048485 <main+10>   call    0x8048330 <gets@plt>
0x804848a <main+15>   add     $0x4,%esp
0x804848d <main+18>   lea     -0x20(%ebp),%eax
0x8048490 <main+21>   push    %eax
0x8048491 <main+22>   call    0x8048350 <strlen@plt>
0x8048496 <main+27>   add     $0x4,%esp
0x8048499 <main+30>   push    %eax
0x804849a <main+31>   push    $0x8048540

```

(cc No process in: jdb)

Fig. 3. gdb split layout output

used. As an example, 20 words from the stack pointer can be seen:

(gdb) x/20x \$sp (4)

The stack pointer is pointing to address "0xbffff588" with value "0x08048509". There are 4 columns and each entry is of 4 bytes (size of 1 word). In the first column, the remaining values seen are the contents of adjacent memory locations.

```

(gdb) x/20x $sp
0xbffff588: 0x08048509  0x00000000  0xbffff598  0x080484a3
0xbffff598: 0x00000000  0xb7e22647  0x00000001  0xbffff634
0xbffff5a8: 0xbffff63c  0x00000000  0x00000000  0x00000000
0xbffff5b8: 0xb7fbd000  0xb7ffffc04  0xb7ffff00  0x00000000
0xbffff5c8: 0xb7fbd000  0xb7fbd000  0x00000000  0x55fbb9c7
(gdb) p $sp
$2 = (void *) 0xbffff588
(gdb) |

```

Fig. 4. 20 words displayed from stack pointer

D. Code Analysis and Verification

For the experiment, the "hackgot.c" code (See Appendix A) is used. The simple program takes a user input through the "gets()" function and copies it to a buffer of 32 bytes. It then utilizes a 'printf' statement to print the number of bytes of the input data and finally "puts()" prints all the variable contents.

The purpose of the code is to have three C in-built functions, "gets", "printf" and "puts" in the executable. Here we are only exploiting the GOT entry for the 'printf'. First, we inspect the locations of GOT in memory and find out where the relative entry of the print function lives. Then we modify that to point to a library "system" call that has the ability to run code.

Since "printf" has called "system", the latter will take the arguments of the 'caller' function. Here the first string before a space is "Buffer" so if an executable is present as "Buffer" in the path, that will be run.

III. RESULTS

Figure 5 shows the execution of shellcode when "printf" calls "__libc_system"

Figure 7 shows how the GOT entry for "printf" was modified to point to "__libc_system"

Figure 7 is the process of finding the GOT in memory space and also figuring out the "printf" entry offset from the GOT.

Figure 8 is the \$PATH entries for 'system' to look for binaries.

Figure 9 shows the PLT entries of the executable, including "printf"

```

(gdb) c
Continuing.
Hello Pumpkins
user
AAAA
[Inferior 1 (process 9270) exited normally]
(gdb)

```

Fig. 5. Shellcode execution upon GOT modification

```

(gdb) x/x 0x08048326
0x08048326 <printf@plt+6>: 0x00000068
(gdb) print _GLOBAL_OFFSET_TABLE_+3
$9 = (const Elf32_Addr *) 0x8049750
(gdb) x/x _GLOBAL_OFFSET_TABLE_+3
0x8049750: 0x08048326
(gdb) print system
$10 = {<text variable, no debug info>} 0xb7e44db0 <__libc_system>
(gdb) set *0x8049750 = 0xb7e44db0
(gdb) x/x _GLOBAL_OFFSET_TABLE_+3
0x8049750: 0xb7e44db0
(gdb)

```

Fig. 6. Modifying the GOT entry for 'printf' with 'libc system'

```

Non-debugging symbols:
0x08048320 printf@plt
(gdb) print _GLOBAL_OFFSET_TABLE_
$8 = 0x8049744
(gdb) x/20x 0x8049744
0x8049744: 0x08049658  0xb7ffff918  0xb7ff0000  0x08048326
0x8049754: 0xb7e693f0  0x08048346  0xb7e88520  0xb7e22550
0x8049764: 0x00000000  0x00000000  0x00000000  0x00000000
0x8049774: 0x00000000  0x00000000  0x00000000  0x00000000
0x8049784: 0x00000000  0x00000000  0x00000000  0x00000000
(gdb) x/x *0x08048326
0x68: Cannot access memory at address 0x68
(gdb) x/x 0x08048326
0x08048326 <printf@plt+6>: 0x00000068

```

Fig. 7. Debugging the memory for GOT entries to find 'printf'

```

user@user-VirtualBox:~/Downloads/Lecture Programs/Lecture8$ echo $PATH
/home/user/bin:/home/user/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin

```

Fig. 8. Binaries execution paths for the system

```

(gdb) info functions
All defined functions:

File hackgot.c:
int main(int, char *);

Non-debugging symbols:
0x080482cc  _init
0x08048300  printf@plt ←
0x08048310  gets@plt
0x08048320  __libc_start_main@plt
0x08048340  _start
0x08048370  __x86.get_pc_thunk.bx
0x08048380  deregister_tm_clones
0x080483b0  register_tm_clones
0x080483f0  __do_global_ctors_aux
0x08048410  frame_dummy
0x08048470  __libc_csu_init
0x080484d0  __libc_csu_fini
0x080484d4  _fini

```

Fig. 9. PLT entries of the executable

IV. DISCUSSIONS

A. Invoking "`__libc_system`" through GOT

As seen in Fig.3, the line at "`main+10`", the compiler will call the "`gets@plt`" in the PLT table. Similarly for "`printf`" and other functions. Before actually running the program binary, we can see the '`printf`' entry(0x08048300) in PLT as in Fig.9. We get an approximate idea of the memory mapping the GOT might refer to initially.

So we set a breakpoint before "`printf()`" statement and ran the code, to see the GOT start location(0x8049744). As in Fig.7. Checking the subsequent sections and printing them, we find the "`printf`" entry within GOT (0x08048326). Notice that it is at a hex offset from the PLT table.

Then we find out where '`system`' is present within '`libc`'. In GDB, "`print system`" shows the actual location(0xb7e44db0) as in Fig.6. The "`printf`" entry is at location "`0x8049750`" at an offset of three from GOT. Using the set command we modify that entry to point to "`__libc_system`".

```
(gdb) set * < memloc > = < value > (5)
```

B. Writing and executing shellcode

Once the print statement is executed, it refers to GOT and redirects the control to the system call. The "`system`" command searches for a binary called "`Buffer`" in the locations seen in Fig.8. So we write a binary with the same name called "`Buffer`" (See Appendix B) and make it an executable. This is put in "`/usr/sbin`" for our VM.

As seen in Fig.5 our shellcode got executed instead of the expected print statement arguments. We successfully took over GOT.

C. Unexpected Results

One of the failed attempts was while trying to overwrite the GOT entries. Instead of modifying the actual entry in GOT, attempted to set the PLT entries for "`printf`". It did not work out and code crashed. Also, it is really difficult to inspect the GOT in GDB because of specific support for a command like in Pwnbdg.

D. Limitations and Considerations

The experiment was done on an Ubuntu VM and these address mapping changes if performed on another OS. Also here the hack works in GDB and is not possible while running the executable normally. Additionally, no specific user input or exploit was used and hence not a perfect attack vector. Moreover, different compilers may address memory differently than our GCC compiler.

Appendix A

The "`hackgot.c`" code:

```

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

int main(int argc, char **argv){

    char buf[32];
    gets(buf);
    printf("Buffer entered is: %d
    bytes\n", strlen(buf));
    puts(buf);

    return 0;
}

```

Appendix B

"Buffer" shellcode in "`/usr/sbin`"

```

#!/bin/sh

echo "Heloo Pumpkins"
whoami

```

REFERENCES

- [1] Bhattacharyya, D.: In: ENPM691 Homework Assignment 3. pp. 1-3 (2024)
- [2] Bhattacharyya, D.: In: ENPM691 Homework Assignment 2. pp. 1-4 (2024)