ENPM691 Homework Assignment 8

Hacking of C programs and Unix Binaries-Fall 2024 lcollier

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:

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.

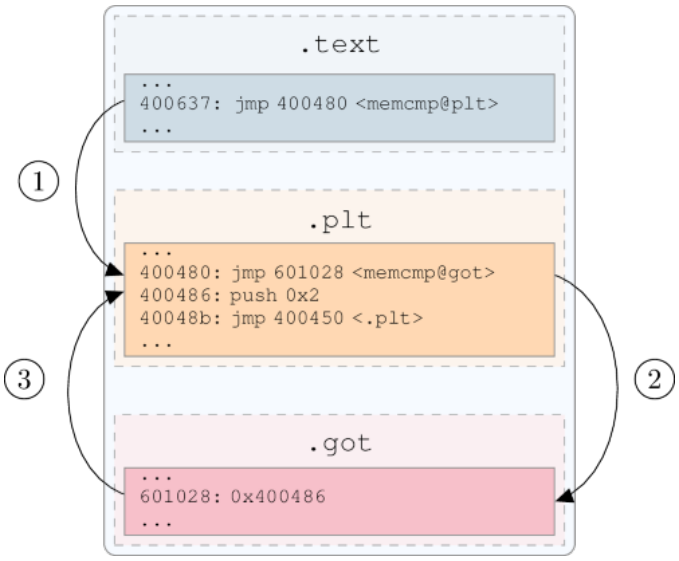

Fig. 2. Referencing PLT/GOT when memcmp() is called

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.

$$(gdb)\ layout\ split \tag{1}$$

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

$$(gdb)\ bt \tag{2}$$

And to inspect a particular frame:

$$(gdb)\ info\ frame\ 1 \tag{3}$$

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

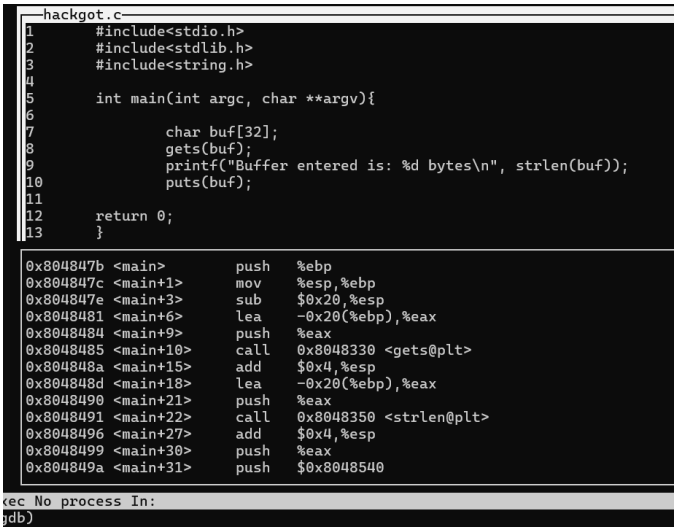

Fig. 3. gdb split layout output

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

$$(gdb)\ x/20x\ \$sp \tag{4}$$

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.

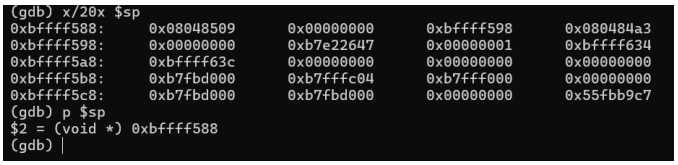

Fig. 4. 20 words displayed from stack pointer

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"

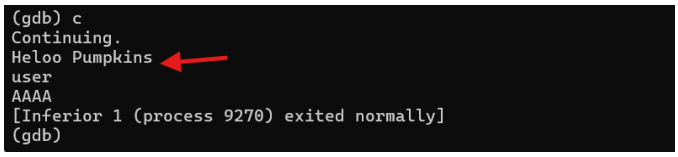

Fig. 5. Shellcode execution upon GOT modification

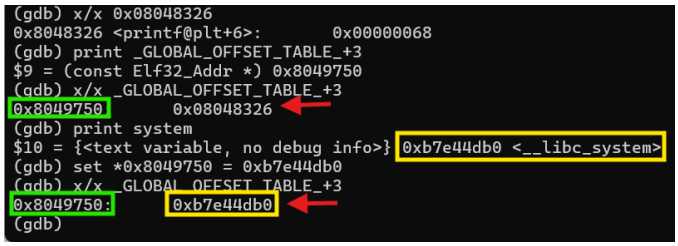

Fig. 6. Modifying the GOT entry for 'printf' with 'libc system'

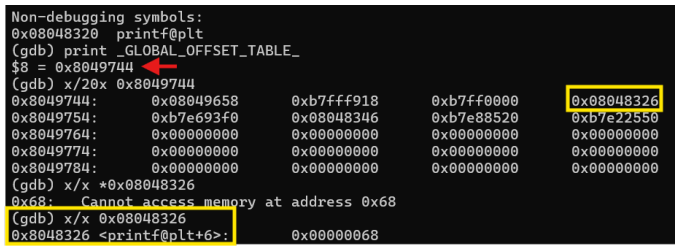

Fig. 7. Debugging the memory for GOT entries to find 'printf'

Fig. 8. Binaries execution paths for the system

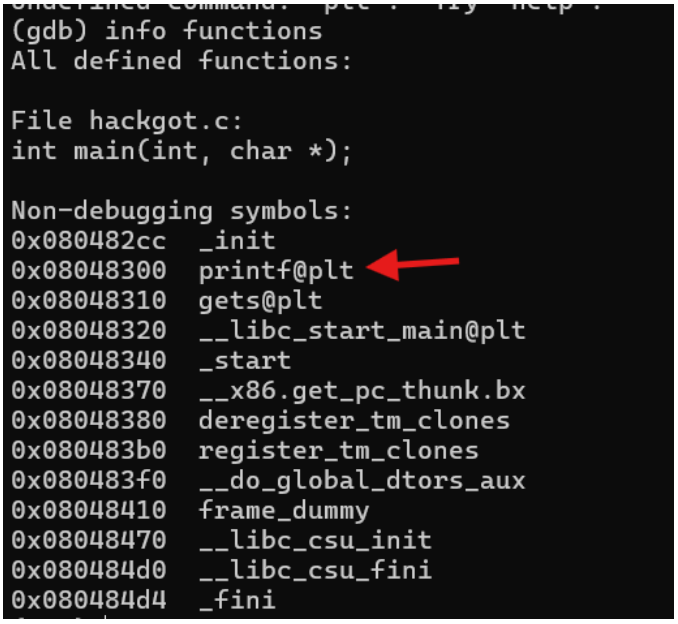

Fig. 9. PLT entries of the executable

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".

$$(gdb)\ set * < memloc > = < value > \tag{5}$$

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)