ENPM691 Homework Assignment 9

Hacking of C programs and Unix Binaries-Fall 2024 lcollier

Dipam Bhattacharyya, MEng Cybersecurity dipamb2@umd.edu

***Abstract*—The assignment aims to understand the mappings of static and extern data and function addresses concerning PLT and GOT. It will also explore how PLT invokes the dynamic linker for lazy binding.**

***Index Terms*—homework, hacking, C, coding**

I. Introduction

[1]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.

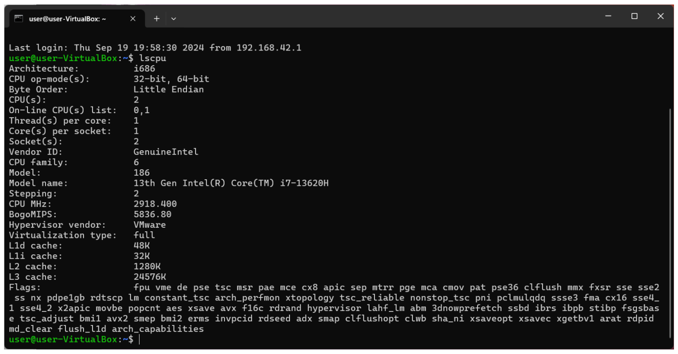
For example, an ELF file does not include its own version of the 'printf' function compiled within it. Instead, it dynamically links to the 'printf' function of the system it is running on. In addition to smaller binary sizes, this also means that users can upgrade their libraries without having to download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

In this experiment, an exploration is done where static and extern data types lives within the GOT. Also, how a function address like 'printf' is resolved at runtime by the PLT and updation of GOT entries through lazy binding.

II. Methodology

A. Host environment and architecture

[2] 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:



```
Last login: Thu Sep 19 19:58:30 2024 from 192.168.42.1
user@user-VirtualBox:~$ lscpu
Architecture:          i686
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
Stepping:              2
CPU MHz:               2918.400
BogoMIPS:              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 msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2
 ss nx pdpe1gb rdtscp lm constant_tsc arch_perfmon xtopology tsc_reliable nonstop_tsc pni pclmulqdq ssse3 fma cx16 sse4_
1 sse4_2 x2apic movbe popcnt aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch ssbd ibrs ibpb stibp fsgsbas
e tsc_adjust bmi1 avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt clwb sha_ni xsaveopt xsavec xgetbv1 arat rdpid
md_clear flush_l1d arch_capabilities
user@user-VirtualBox:~$ 
```

Fig. 1. Ubuntu CPU information

B. Working of PLT and GOT

[1] 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.

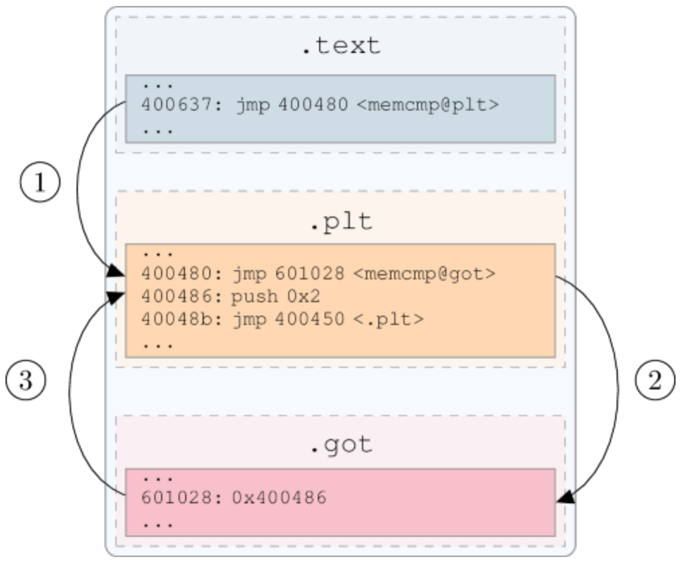
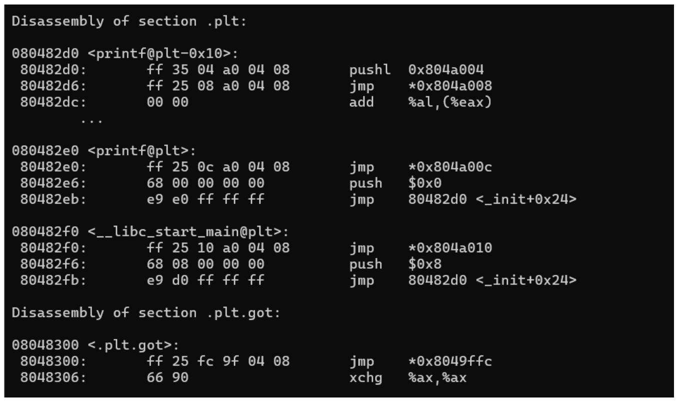

.text
...
400637: jmp 400480 <memcmp@plt>
...
1
.plt
...
400480: jmp 601028 <memcmp@got>
400486: push 0x2
40048b: jmp 400450 <.plt>
...
2
3
.got
...
601028: 0x400486
...

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

C. Objdump and Readelf

Objdump is a command-line tool used for inspecting and analyzing binary files, typically executable files, object files, or shared libraries. It is commonly found in Unix-like operating systems, especially in Linux, and is part of the GNU Binutils package. To view the disassembly of the entire program:

$$objdump\ --disassemble\ <executable> \tag{1}$$

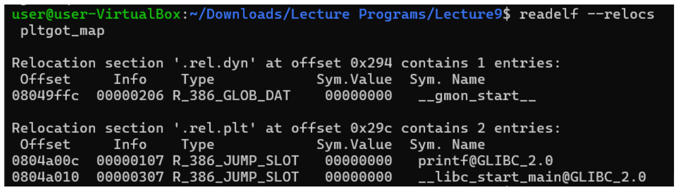


```
Disassembly of section .plt:

080482d0 <printf@plt-0x10>:
 80482d0:       ff 35 04 a0 04 08       pushl  0x804a004
 80482d6:       ff 25 08 a0 04 08       jmp    *0x804a008
 80482dc:       00 00                   add    %al,(%eax)
        ...

080482e0 <printf@plt>:
 80482e0:       ff 25 0c a0 04 08       jmp    *0x804a00c
 80482e6:       68 00 00 00 00          push   $0x0
 80482eb:       e9 e0 ff ff ff          jmp    80482d0 <_init+0x24>

080482f0 <__libc_start_main@plt>:
 80482f0:       ff 25 10 a0 04 08       jmp    *0x804a010
 80482f6:       68 08 00 00 00          push   $0x8
 80482fb:       e9 d0 ff ff ff          jmp    80482d0 <_init+0x24>

Disassembly of section .plt.got:

08048300 <.plt.got>:
 8048300:       ff 25 fc 9f 04 08       jmp    *0x8049ffc
 8048306:       66 90                   xchg   %ax,%ax
```

Fig. 3. disassembled PLT section using objdump

Readelf is a tool that allows to examine and display detailed information about ELF files, such as headers, sections, segments, symbols, and much more. It's similar to objdump, but specifically tailored for ELF files. To check the relocation information for the executable:

$$objdump\ --relocs\ <executable> \tag{2}$$



```
user@user-VirtualBox:~/Downloads/Lecture Programs/Lecture9$ readelf --relocs
 pltgot_map

Relocation section '.rel.dyn' at offset 0x294 contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
08049ffc  00000206 R_386_GLOB_DAT    00000000   __gmon_start__

Relocation section '.rel.plt' at offset 0x29c contains 2 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804a00c  00000107 R_386_JUMP_SLOT   00000000   printf@GLIBC_2.0
0804a010  00000307 R_386_JUMP_SLOT   00000000   __libc_start_main@GLIBC_2.0
```

Fig. 4. Relocation section showing symbols and addresses

D. GDB debugging

[3] 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.

$$(gdb)\ layout\ split \tag{3}$$

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

$$(gdb)\ bt \tag{4}$$

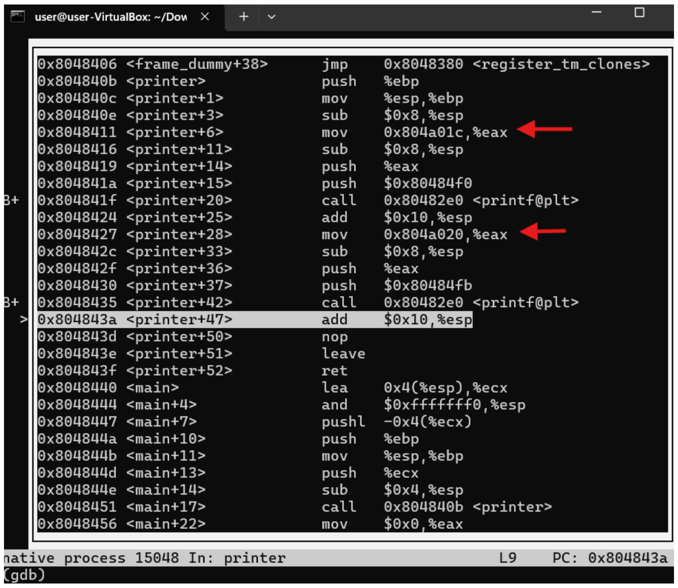
And to inspect a particular frame:

$$(gdb)\ info\ frame\ 1 \tag{5}$$

To check the contents of memory from a particular point, say pointer or absolute address, the following(See Fig.4) is used. As an example, 20 words from the stack pointer can be seen:

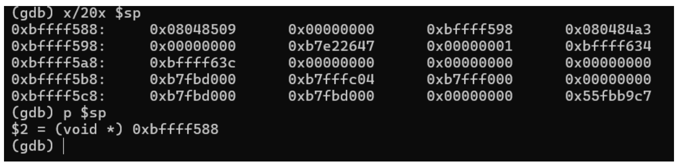
$$(gdb)\ x/20x\ \$sp \tag{6}$$

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.



```
   0x8048406 <frame_dummy+38>      jmp    0x8048380 <register_tm_clones>
   0x804840b <printer>             push   %ebp
   0x804840c <printer+1>           mov    %esp,%ebp
   0x804840e <printer+3>           sub    $0x8,%esp
   0x8048411 <printer+6>           mov    0x804a01c,%eax
   0x8048416 <printer+11>          sub    $0x8,%esp
   0x8048419 <printer+14>          push   %eax
   0x804841a <printer+15>          push   $0x80484f0
B+ 0x804841f <printer+20>          call   0x80482e0 <printf@plt>
   0x8048424 <printer+25>          add    $0x10,%esp
   0x8048427 <printer+28>          mov    0x804a020,%eax
   0x804842c <printer+33>          sub    $0x8,%esp
   0x804842f <printer+36>          push   %eax
   0x8048430 <printer+37>          push   $0x80484fb
B+ 0x8048435 <printer+42>          call   0x80482e0 <printf@plt>
 > 0x804843a <printer+47>          add    $0x10,%esp
   0x804843d <printer+50>          nop
   0x804843e <printer+51>          leave
   0x804843f <printer+52>          ret
   0x8048440 <main>                lea    0x4(%esp),%ecx
   0x8048444 <main+4>              and    $0xfffffff0,%esp
   0x8048447 <main+7>              pushl  -0x4(%ecx)
   0x804844a <main+10>             push   %ebp
   0x804844b <main+11>             mov    %esp,%ebp
   0x804844d <main+13>             push   %ecx
   0x804844e <main+14>             sub    $0x4,%esp
   0x8048451 <main+17>             call   0x804840b <printer>
   0x8048456 <main+22>             mov    $0x0,%eax
native process 15048 In: printer                          L9    PC: 0x804843a
(gdb)
```

Fig. 5. gdb split layout output



```
(gdb) x/20x $sp
0xbffff588:     0x08048509      0x00000000      0xbffff598      0x080484a3
0xbffff598:     0x00000000      0xb7e22647      0x00000001      0xbffff634
0xbffff5a8:     0xbffff63c      0x00000000      0x00000000      0x00000000
0xbffff5b8:     0xb7fbd000      0xb7fffc04      0xb7fff000      0x00000000
0xbffff5c8:     0xb7fbd000      0xb7fbd000      0x00000000      0x55fbb9c7
(gdb) p $sp
$2 = (void *) 0xbffff588
(gdb)
```

Fig. 6. 20 words displayed from stack pointer

E. Code Analysis and Verification

For the experiment, the "pltgot_map.c" code (See Appendix A) is used. The 'main' function calls the "printer()" that prints the value of two variables declared globally. These are of "static" and "extern" datatype and with some initialized value. There are two 'print' statements inside the "printer()" function.

Purpose of the code is to look at how compiler accesses these variable and see if it references the PLT or GOT. This means checking the usage of relocation table or relative entry to GOT address.

Another goal is to purposefully add two 'print' statements. The first statement will call the dynamic linker and update the temporary "printf" address in GOT with the actual location. The second call will directly visit the print library entry in GOT.

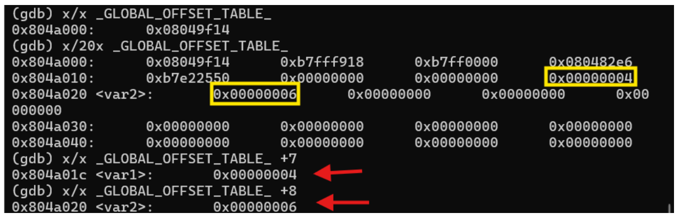
III. Results

Figure7 shows the GOT address space in memory and its contents. The yellow boxes are contents of static and extern variables.

Figure8 indicates how PLT calls the function invoking dynamic linker in GOT for the first time.

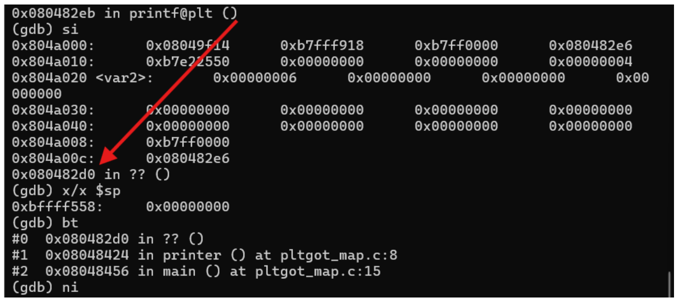
Figure9 displays the instruction that calls "_dl_runtime_resolve()" entry or dynamic linker function present in GOT.

Figure10 is how the updation is done by ”_dl_trampoline.S” by calling functions to resolve symbols and patch GOT.



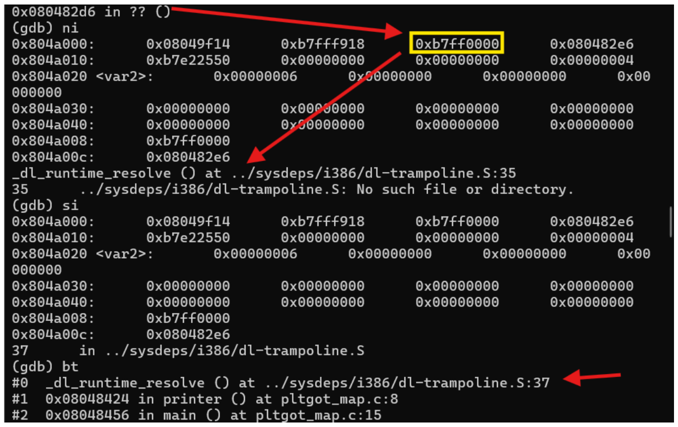
```
(gdb) x/x _GLOBAL_OFFSET_TABLE_
0x804a000:      0x08049f14
(gdb) x/20x _GLOBAL_OFFSET_TABLE_
0x804a000:      0x08049f14      0xb7fff918      0xb7ff0000      0x080482e6
0x804a010:      0xb7e22550      0x00000000      0x00000000      0x00000004
0x804a020 <var2>:       0x00000006      0x00000000      0x00000000      0x00
000000
0x804a030:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a040:      0x00000000      0x00000000      0x00000000      0x00000000
(gdb) x/x _GLOBAL_OFFSET_TABLE_ +7
0x804a01c <var1>:       0x00000004
(gdb) x/x _GLOBAL_OFFSET_TABLE_ +8
0x804a020 <var2>:       0x00000006
```

Fig. 7. GOT address space containing static/extern data at a fixed offset



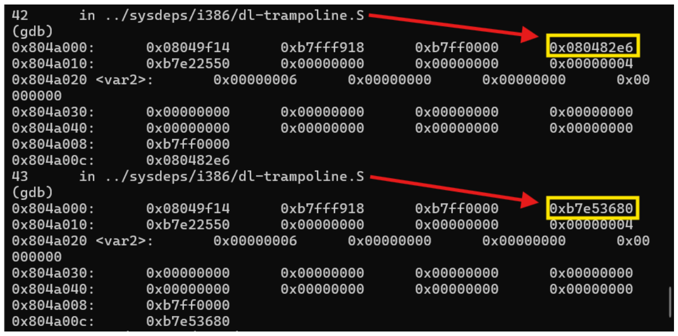
```
0x080482eb in printf@plt ()
(gdb) si
0x804a000:      0x08049f14      0xb7fff918      0xb7ff0000      0x080482e6
0x804a010:      0xb7e22550      0x00000000      0x00000000      0x00000004
0x804a020 <var2>:       0x00000006      0x00000000      0x00000000      0x00
000000
0x804a030:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a008:      0xb7ff0000
0x804a00c:      0x080482e6
0x080482d0 in ?? ()
(gdb) x/x $sp
0xbffff558:     0x00000000
(gdb) bt
#0  0x080482d0 in ?? ()
#1  0x08048424 in printer () at pltgot_map.c:8
#2  0x08048456 in main () at pltgot_map.c:15
(gdb) ni
```

Fig. 8. print() in PLT invoking call to function that resolves to GOT



```
0x080482d6 in ?? ()
(gdb) ni
0x804a000:      0x08049f14      0xb7fff918      0xb7ff0000      0x080482e6
0x804a010:      0xb7e22550      0x00000000      0x00000000      0x00000004
0x804a020 <var2>:       0x00000006      0x00000000      0x00000000      0x00
000000
0x804a030:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a008:      0xb7ff0000
0x804a00c:      0x080482e6
_dl_runtime_resolve () at ../sysdeps/i386/dl-trampoline.S:35
35      ../sysdeps/i386/dl-trampoline.S: No such file or directory.
(gdb) si
0x804a000:      0x08049f14      0xb7fff918      0xb7ff0000      0x080482e6
0x804a010:      0xb7e22550      0x00000000      0x00000000      0x00000004
0x804a020 <var2>:       0x00000006      0x00000000      0x00000000      0x00
000000
0x804a030:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a008:      0xb7ff0000
0x804a00c:      0x080482e6
37      in ../sysdeps/i386/dl-trampoline.S
(gdb) bt
#0  _dl_runtime_resolve () at ../sysdeps/i386/dl-trampoline.S:37
#1  0x08048424 in printer () at pltgot_map.c:8
#2  0x08048456 in main () at pltgot_map.c:15
```

Fig. 9. Instruction stub that calls dynamic linker function entry in GOT



```
42      in ../sysdeps/i386/dl-trampoline.S
(gdb)
0x804a000:      0x08049f14      0xb7fff918      0xb7ff0000      0x080482e6
0x804a010:      0xb7e22550      0x00000000      0x00000000      0x00000004
0x804a020 <var2>:       0x00000006      0x00000000      0x00000000      0x00
000000
0x804a030:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a008:      0xb7ff0000
0x804a00c:      0x080482e6
43      in ../sysdeps/i386/dl-trampoline.S
(gdb)
0x804a000:      0x08049f14      0xb7fff918      0xb7ff0000      0xb7e53680
0x804a010:      0xb7e22550      0x00000000      0x00000000      0x00000004
0x804a020 <var2>:       0x00000006      0x00000000      0x00000000      0x00
000000
0x804a030:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a008:      0xb7ff0000
0x804a00c:      0xb7e53680
```

Fig. 10. GOT entry before/after printf() address modification by linker

IV. Discussions

A. *Mapping of static and extern data*

As seen in Fig.7, the GOT address starts from ”0x804a000”. Inspecting further into the space, we find the static and extern data storage locations. The static variable ’var1’ has a value of ”4” and it is located at an offset of 7 from the GOT. Similarly for ’var2’ of value ”6” and extern type is located next to ’var1’ at an offset of 8.

If we look at Fig.5 and the disassembly of ”printer()”, the assembly instructions clearly show that absolute memory locations are used by the compiler to load these variables into the register. For both of these types, these addresses are within the GOT but are not referenced indirectly. The red arrows in Fig.5 mark these instructions. The ”MOV” operations just load them onto the stack for ”printf()” to console out.

Hence, we can say that PLT/GOT calls and patching are not required for these types from our experiment.

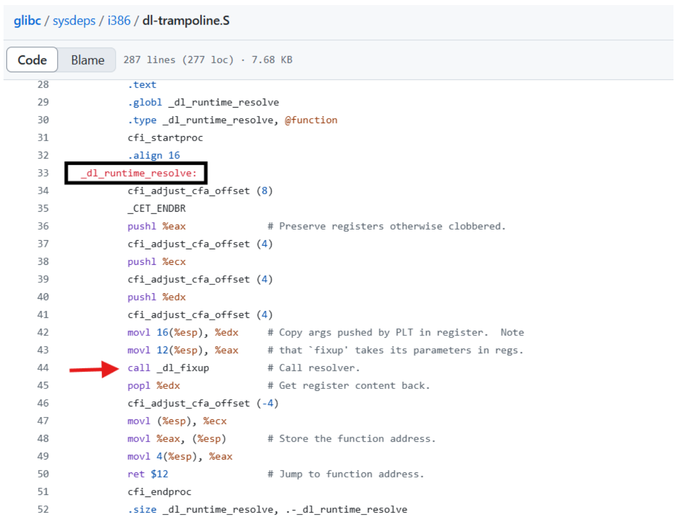
B. *Resolving of printf() through lazy binding*

When the ”printer()” function calls the ”printf()” for the 1st time, as seen in Fig.5 at line ”printer+20”, it checks the ”.plt” section at ”080482e0” (See Fig.3). The instructions inside ”printf@plt” jump to the relative entry of ”printf” present in GOT at ”0x804a00c”. Fig.4 also shows this. The value present there simply instructs to return to PLT as it does not have the actual address of the function.

PLT then jumps to ”80482d0” which is nothing but the ”printf@plt-0x10” function responsible for invoking the dynamic linker function present at ”80482d6” (See Fig.8 and Fig.3). Fig.9 shows that memory entry in GOT, ”0xb7ff0000” is the ”_dl_runtime_resolve()” function present inside the file ”_dl_trampoline.S”. This is essentially the dynamic linking of objects or lazy linding. Since it is the first-time call, it is used to resolve the actual address of ”printf”.

C. *GOT patching by _dl_fixup()*

As seen in Fig.10, ”_dl_trampoline.S” resolves the ”printf” symbol and updates the entry from ”0x080482e6” to ”0xb7e53680”. Internally, the file calls the function ”_dl_runtime_resolve()” which in turn refers to ”dl_fixup()”. ”dl_fixup” is the workhorse that actually resolves the symbol in question. Once the symbol's address is found, the program's GOT entry for it must be patched. This is also the job of ”dl_fixup()”.


glibc / sysdeps / i386 / dl-trampoline.S

Code Blame 287 lines (277 loc) · 7.68 KB

```
		.text
		.globl _dl_runtime_resolve
		.type _dl_runtime_resolve, @function
		cfi_startproc
		.align 16
_dl_runtime_resolve:
		cfi_adjust_cfa_offset (8)
		_CET_ENDBR
		pushl %eax		# Preserve registers otherwise clobbered.
		cfi_adjust_cfa_offset (4)
		pushl %ecx
		cfi_adjust_cfa_offset (4)
		pushl %edx
		cfi_adjust_cfa_offset (4)
		movl 16(%esp), %edx	# Copy args pushed by PLT in register.  Note
		movl 12(%esp), %eax	# that `fixup' takes its parameters in regs.
		call _dl_fixup		# Call resolver.
		popl %edx		# Get register content back.
		cfi_adjust_cfa_offset (-4)
		movl (%esp), %ecx
		movl %eax, (%esp)	# Store the function address.
		movl 4(%esp), %eax
		ret $12			# Jump to function address.
		cfi_endproc
		.size _dl_runtime_resolve, .-_dl_runtime_resolve
```

Fig. 11. Source code of _dl_trampoline.S showing functions _dl_runtime_resolve() and dl_fixup()

The second time, though, this is not followed. When ”printf@plt” is called, it checks the GOT entry and finds the actual address of ”printf” in memory. This is the magic of lazy binding, only for the first time the linker is required. Notice the difference in stack frames in Fig.9 versus Fig.12.

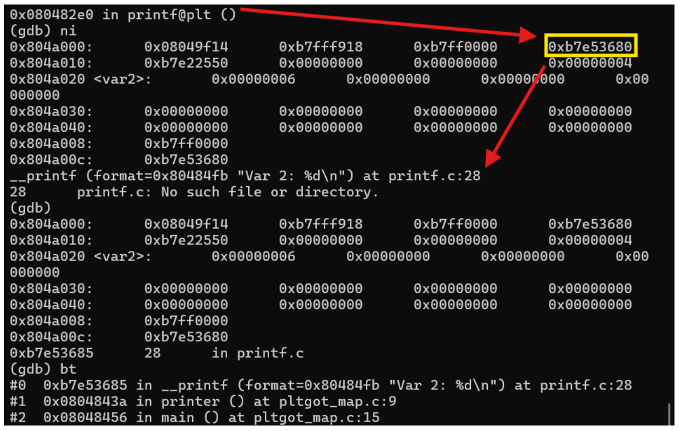

```
0x080482e0 in printf@plt ()
(gdb) ni
0x804a000:      0x08049f14      0xb7fff918      0xb7ff0000      0xb7e53680
0x804a010:      0xb7e22550      0x00000000      0x00000000      0x00000004
0x804a020 <var2>:       0x00000006      0x00000000      0x00000000      0x00
000000
0x804a030:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a008:      0xb7ff0000
0x804a00c:      0xb7e53680
__printf (format=0x80484fb "Var 2: %d\n") at printf.c:28
28      printf.c: No such file or directory.
(gdb)
0x804a000:      0x08049f14      0xb7fff918      0xb7ff0000      0xb7e53680
0x804a010:      0xb7e22550      0x00000000      0x00000000      0x00000004
0x804a020 <var2>:       0x00000006      0x00000000      0x00000000      0x00
000000
0x804a030:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a008:      0xb7ff0000
0x804a00c:      0xb7e53680
0xb7e53685      28      in printf.c
(gdb) bt
#0  0xb7e53685 in __printf (format=0x80484fb "Var 2: %d\n") at printf.c:28
#1  0x0804843a in printer () at pltgot_map.c:9
#2  0x08048456 in main () at pltgot_map.c:15
```

Fig. 12. Second time direct call to printf() by print@plt

D. *Unexpected Results*

One of the interesting things about the x86-64 bit processor environment like here is that it does not use the global offset table under relocation headers for addressing static/extern variables in contrast to AMD ones. It is always an absolute memory.

Secondly, for user-defined function calls in the source code, no relocation information is present. In our experiment only ”libc-main” and ”printf glibc” were present even though that was called.

E. *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 an intel processor and others may resolve in different ways. Moreover, different compilers may address memory differently than our GCC compiler.

Appendix A
The ”pltgot_map.c” code:

```
#include<stdio.h>

static int var1 = 4;
extern int var2 = 6;

void printer() {

 printf("Var 1: %d\n",var1);
 printf("Var 2: %d\n",var2);

}

int main() {

 printer();
 return 0;

}
```

References

[1] Bhattacharyya, D.: In: ENPM691 Homework Assignment 8. pp. 1–3 (2024)

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

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