CS451 – Software Analysis

Lecture 2

PLT and GOT

Elias Athanasopoulos
athanasopoulos.elias@ucy.ac.cy

Dynamic linking

- Binaries have dependencies
  - Shared libraries
  - Code that can be shared by multiple processes
  - In Linux their names are libX.so
- A shared library can be used in parallel by many processes
  - E.g., libc.so is linked with almost all running programs
  - One copy of the shared library is mapped in memory at any given time

The problem

- **Many processes** can run concurrently and call printf() (or another library function)
- **One** libc.so is mapped in the physical memory
  – It is called a **shared** library
- Each process has mapped libc.so in its virtual address space
  – In a **random** location
- All processes need to resolve and call printf()

Solution: lazy binding

- Programs are compiled with relocatable addresses
  - Use a temporary address, which will be **fixed** at run-time
  - The dynamic loader is responsible for fixing those addresses
- When a library function (e.g., printf()) is called, the dynamic loader will fix the temporary address
- The fix will be done during the first call and will remain until the process ends
- The fix, correcting the memory address of the printf(), is called **binding**
  - **Lazy** comes from the fact that it happens on demand for each library function

`LD_BIND_NOW`

- You can force the dynamic linker to fix all addresses on process start
  - Set the environment variable LD_BIND_NOW to 1
- All address corrections will be done when the program runs
- The initialization of the process will be much slower if many different library function calls are used

How the address fix works?

- The dynamic loader will be called to fix an address through a series of indirections
- When the program is compiled, the produced binary contains some specific sections for these redirections
  - `.plt` and `.got.plt`

PLT and GOT

- Procedure Linkage Table (PLT)
  - Code section

[Nr]	Name	Type	Address	Off	Size	ES	Flg	Lk	Inf	Al
[12]	.plt	PROGBITS	00000000004004d0	0004d0	000020	10	AX	0	0	16

- Global Offset Table (GOT)
  - Data section

[Nr]	Name	Type	Address	Off	Size	ES	Flg	Lk	Inf	Al
[22]	.got.plt	PROGBITS	0000000000601000	001000	000020	08	WA	0	0	8

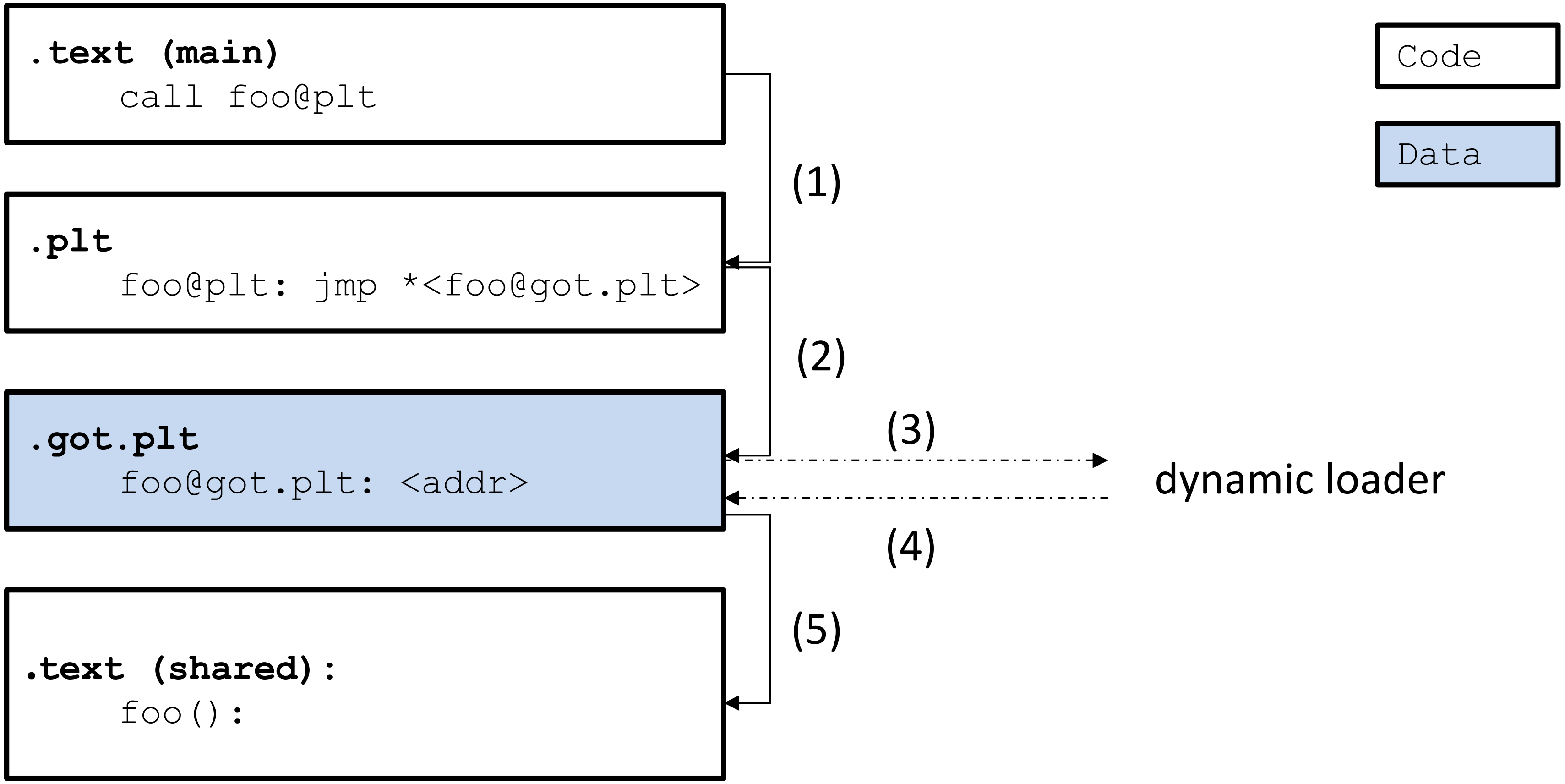
High-level idea

- Assume a program that calls a library function called foo(), which is located in a shared library
- The compiler will add an entry for foo() in the PLT section
  - The entry will be filled with some pre-defined code
- The compiler will use foo@plt in each call of foo()
- Remember: foo@plt is located in the PLT and it is not the actual foo()

How PLT works?

- foo@plt() implements an indirect jump towards the actual foo()
- The indirect jump takes the target address from the GOT section
- The compiler has also put an entry of foo() in the GOT section (foo@got.plt)
- Initially, foo@got.plt has a non fixed address
- The dynamic loader will fix that in the first call
  – Then foo@got.plt will have the address of the actual foo()

Diagram of indirections



Two indirections

- Why do we need both PLT and GOT?
  - We could have only PLT (i.e., one indirection)
  - The dynamic loader can update the address located in the PLT
- PLT is code (not writable)
- GOT is data (writable)

Security argument

- Non-executable pages (NX-bit/DEP)
  - All data pages cannot be executed, but can be read or be written
  - All code pages can be executed and be read, but not be written
- If the dynamic loader needs to update the PLT then the code page of the PLT needs to be both writable and executable at the same time

Code-reuse argument

- The code of the shared library is shared by multiple processes
- Code in a shared library may call functions in other shared libraries
- Libraries have their own PLT and GOT sections
- Code (PLT) is shared among many processes
- Data (GOT) is privately copied to each process

Example

main.c:

```
void foo(void);  /* Implemented in a shared library.  */

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

        foo();  /* 1st call.  */
        foo();  /* 2nd call.  */

        return 1;
}
```

foo.c

```
#include <stdio.h>

void foo(void) {
        fprintf(stderr, "foo() in shared library.\n");
}
```

Compile and run

```
# compile the shared library
$ gcc -Wall foo.c -shared -fPIC -o libfoo.so

# compile the main program
$ gcc -Wall main.c -L. -lfoo -o main

# update the path for the dynamic linker to find libfoo.so
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.

# run
$ ./main
foo() in shared library.
foo() in shared library.
```

PLT/GOT in action

```
(gdb) b foo@plt
Breakpoint 1 at 0x400510
(gdb) r
Starting program: /home/elathan/epl451/week1/plt/main
Missing separate debuginfos, use: yum debuginfo-install glibc-2.28-164.el8.x86_64

Breakpoint 1, 0x0000000000400510 in foo@plt ()
(gdb) disas
Dump of assembler code for function foo@plt:
=> 0x0000000000400510 <+0>: jmpq   *0x200b02(%rip)        # 0x601018 <foo@got.plt>
   0x0000000000400516 <+6>: pushq  $0x0
   0x000000000040051b <+11>: jmpq   0x400500
End of assembler dump.
(gdb) x/1gx 0x601018
0x601018 <foo@got.plt>: 0x0000000000400516
(gdb) c
Continuing.
foo() in shared library.

Breakpoint 1, 0x0000000000400510 in foo@plt ()
(gdb) x/1gx 0x601018
0x601018 <foo@got.plt>: 0x00007ffff7bce5f9
(gdb) x/1ix 0x00007ffff7bce5f9
0x7ffff7bce5f9 <foo>: 0xf4058b48e5894855
```

Homework

- Create the program main.c and the shared library (foo.c)
- Use readelf to inspect the PLT and GOT sections in the main binary
- Use gdb to see how the dynamic loader is fixing the address of the actual foo() in the GOT
- Try the same steps with LD_BIND_NOW set to 1