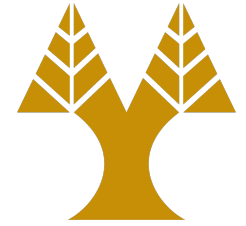


CS451 – Software Analysis

Lecture 4

Handling Library Functions

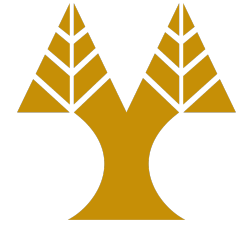
Elias Athanasopoulos
elathan@cs.ucy.ac.cy



Library Functions

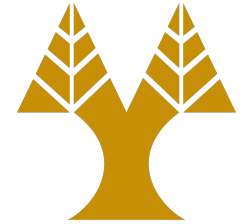
- Binaries call library functions
 - This code is located in a shared library
- Library functions do not generate events in the operating systems as system calls do
 - However, we can use `ptrace()` with a combination of other techniques to inspect library functions
 - This is how debuggers create breakpoints
- Since library functions are in shared libraries we can use another interesting direction to inspect and modify binary code

Shared libraries



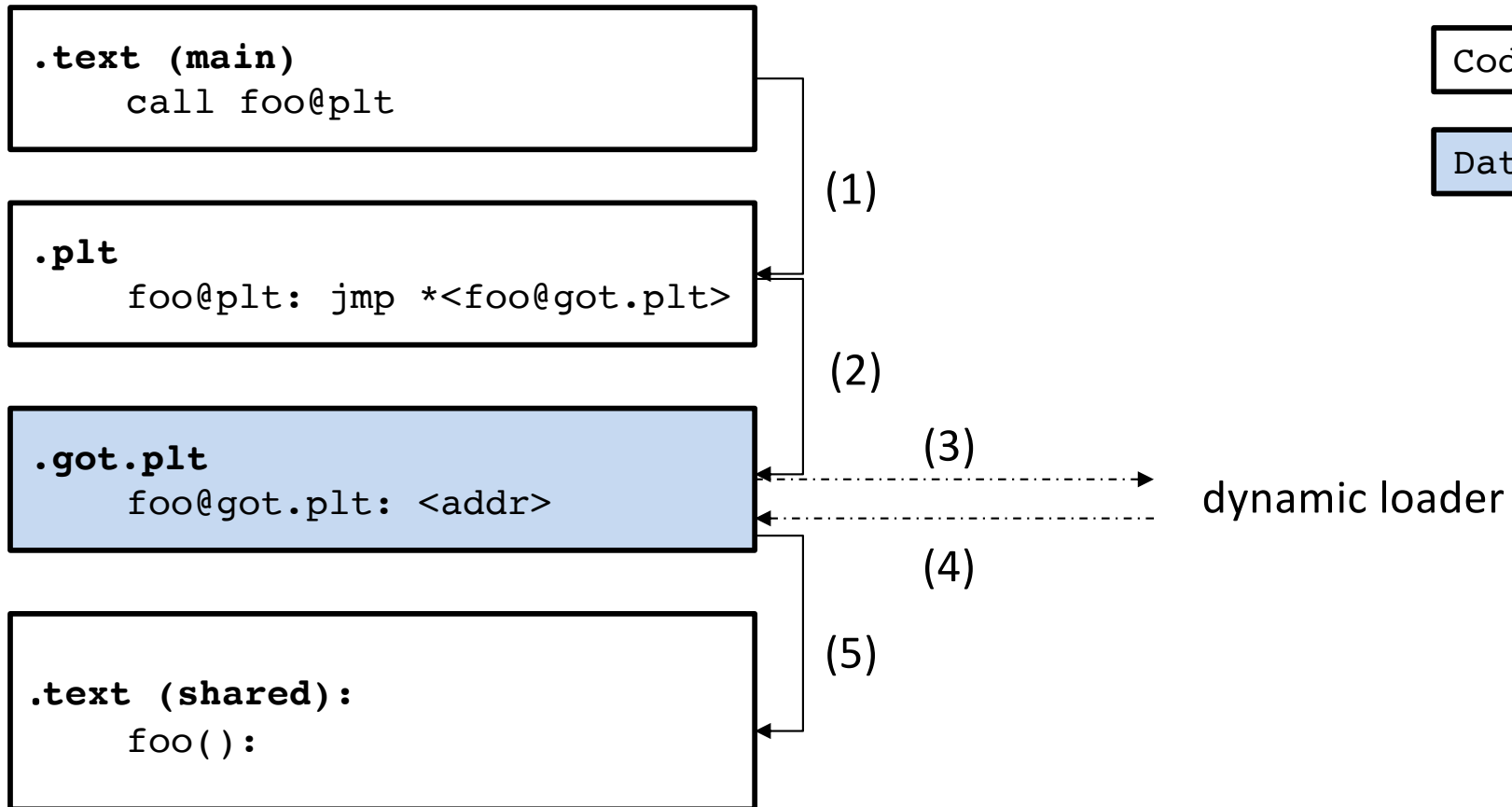
- Recall that shared libraries host code is used by many processes
- Recall that a library function is actually resolved at run-time
 - Using the PLT and the GOT
- The dynamic loader is *fixing* the address of the library function during the first call
 - Can we trick the loader to fix the address with the one we control?

Recall

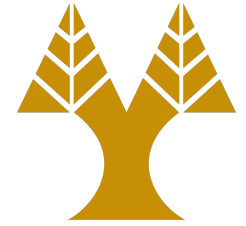


Code

Data

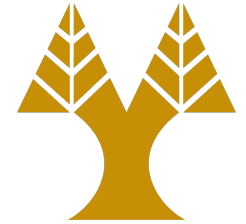


Dynamic loader resolution



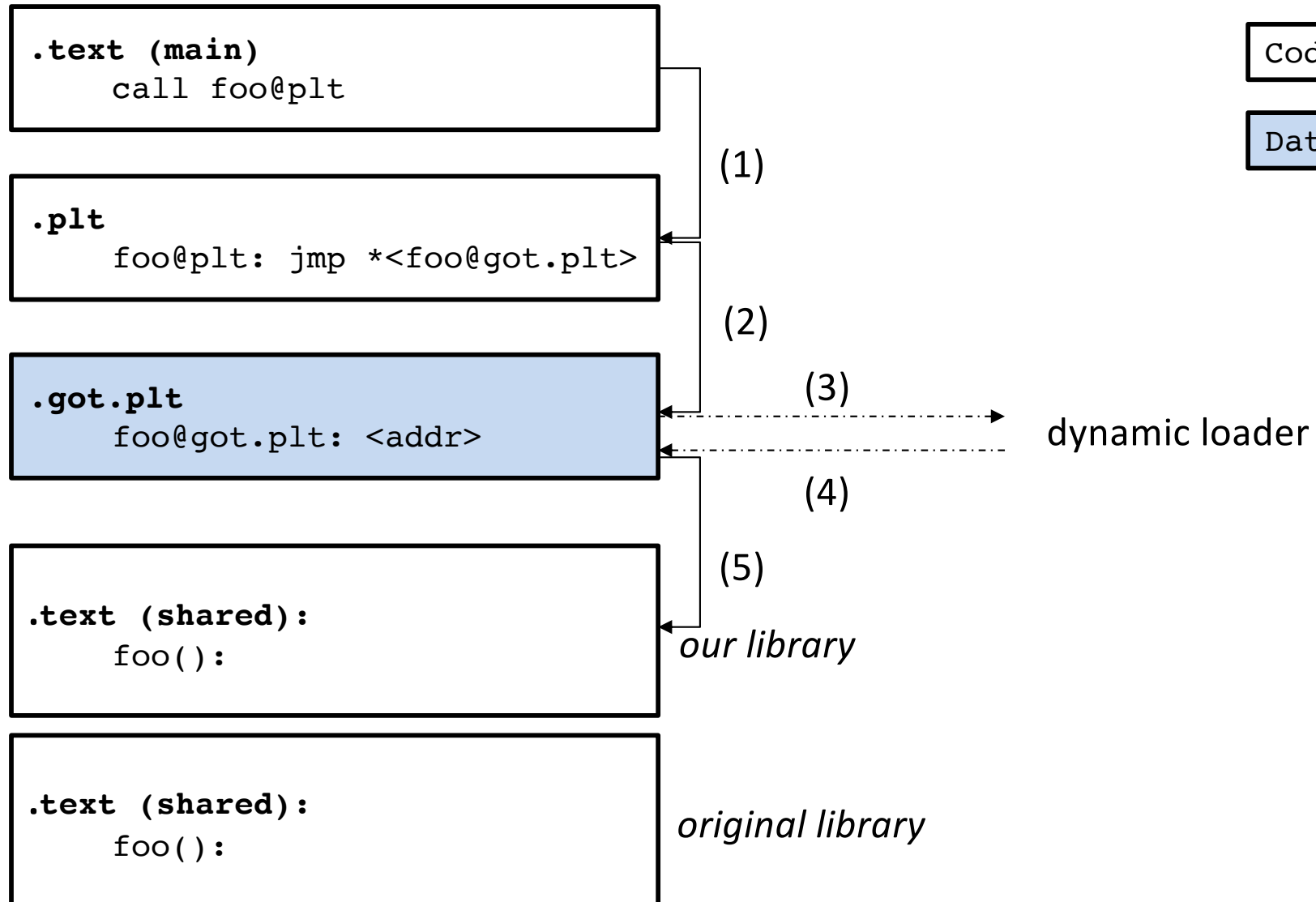
- The same symbol may be available in multiple shared libraries
 - The dynamic loader will find the first one available
- If we can load a shared object first, that contains the exact same symbol, we can force the dynamic loader to use our code
 - Therefore, upon the library call, we will be able to execute our version of the library function
 - Our version may do simply nothing but call the original version (accounting)

The idea in high level

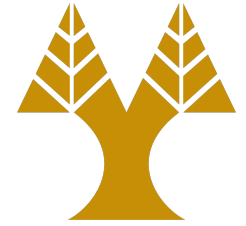


Code

Data



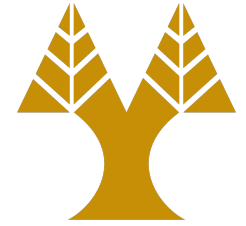
Load our library first



- We can force a program to load any shared library first using the LD_PRELOAD environment variable

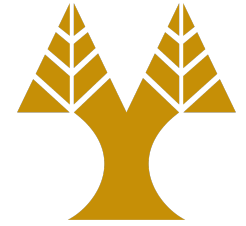
```
$ LD_PRELOAD=./libfirst.so <program>
```

Preloaded object



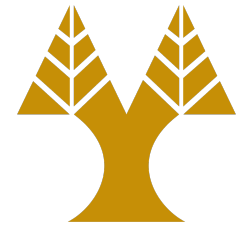
- The preloaded shared library must have all symbols we need to replace
- Each symbol needs to have identical definition with the original one
- E.g., if we need to hook `malloc()` we need to provide a new implementation
 - The definition of our `malloc()` needs to be identical with the definition of the original `malloc`
 - If the definitions are different, then replacing the two symbols may cause the running program to crash

Inside the hook



- Our version of the library function can do different things
 - It can totally replace the functionality of the original function
 - It can provide some extra functionality on top of the original one
- In the second case, we need to be able to call the original function

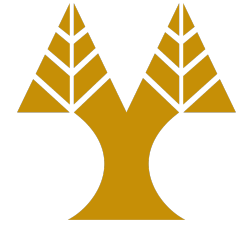
Calling the original function



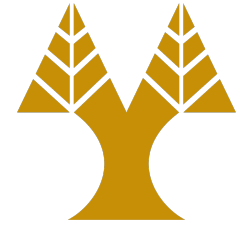
- The dynamic loader exports an API where we can use it
- The API is implemented libdl.so
- The API contains functions for manually resolving specific symbols

```
typedef void *(*real_malloc_t)(size_t);  
static real_malloc_t real_malloc = NULL;  
  
real_malloc = (real_malloc_t) dlsym(RTLD_NEXT, "malloc");
```

Example of a memory profiler



- Memory management is used in custom allocators implemented by each program
- By default, libc.so offers a simple allocator, but more complicated program may have their own
 - E.g., all web browsers have custom allocator implementations
- Whichever allocator you use, the API is the same
 - Based on malloc(), calloc(), free, etc.
- We can develop a memory profiler, that can simply count the number of malloc() and free() calls



Replacing malloc

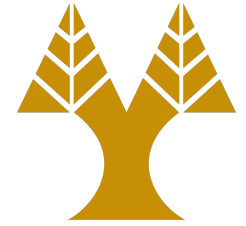
- Our implementation increases a global counter and calls the original malloc() for handling the allocation

```
typedef void *(*real_malloc_t)(size_t);
static real_malloc_t real_malloc = NULL;

void * malloc(size_t size) {
    if (!real_malloc) {
        real_malloc = (real_malloc_t) dlsym(RTLD_NEXT, "malloc");
        if (!real_malloc) {
            die("real malloc problem: %s", dlerror());
        }
    }
    void *p = (void *)real_malloc(size);
    stats_total_malloc++;

    return p;
}
```

Replacing free

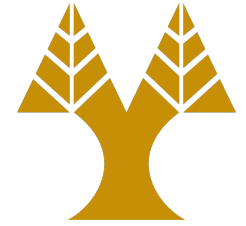


- Our implementation increases a global counter and calls the original free() to handle the deallocation

```
typedef void (*real_free_t)(void *);
static real_free_t real_free = NULL;

void free(void *ptr) {
    if (!real_free) {
        real_free = (real_free_t)dlsym(RTLD_NEXT, "free");
        if (!real_free)
            die("real free problem: %s", dlerror());
    }
    real_free(ptr);
    stats_total_free++;
    return;
}
```

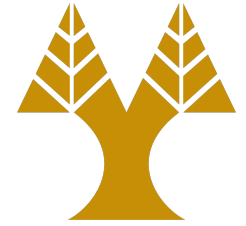
How to print the statistics?



- C allows a process to execute a constructor and/or a destructor
 - main() is never the first function executed, actually
- A destructor is the ideal place to insert code for printing the stats

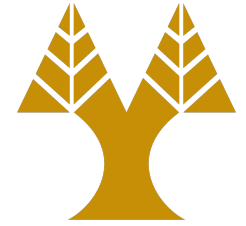
```
__attribute__((destructor)) static void stats(void) {  
    printf("malloc() calls: %ld\n", stats_total_malloc);  
    printf("free() calls recorded: %ld\n", stats_total_free);  
}
```

Some extra bits



- We need to define `_GNU_SOURCE` before `dlfcn.h` (the dynamic loader supported functions) for making `RTLD_NEXT` visible
 - This is for compatibility with other Unix systems

Example program to preload



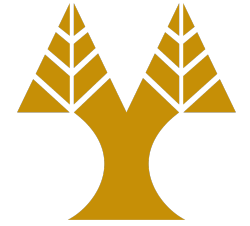
```
#include <stdlib.h>

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

    for (int i = 0; i < 1028; i++) {
        void *p = malloc(16);
        if (i % 2) free(p);
    }

    return 1;
}
~
```


Compile and run



```
$ gcc -Wall -shared -fPIC -ldl memprofiler.c  
-o libmemprofiler.so
```

```
$ gcc -Wall example.c -o example
```

```
$ LD_PRELOAD=./libmemprofiler.so ./example  
malloc() calls recorded: 1028  
free() calls recorded: 514
```

Homework



- Replace another library function call with one of your own
- Can you add some printing features inside malloc/free replacements?
 - I.e., print the size of the allocation
 - Beware, this can be a tricky task