Sun, 27 April 2014 17:08

Redirection of Imported Functions in Mach-O

In the previous article, we examined the mechanism of dynamic linking of functions in Mach-O. Now let's move to practice.

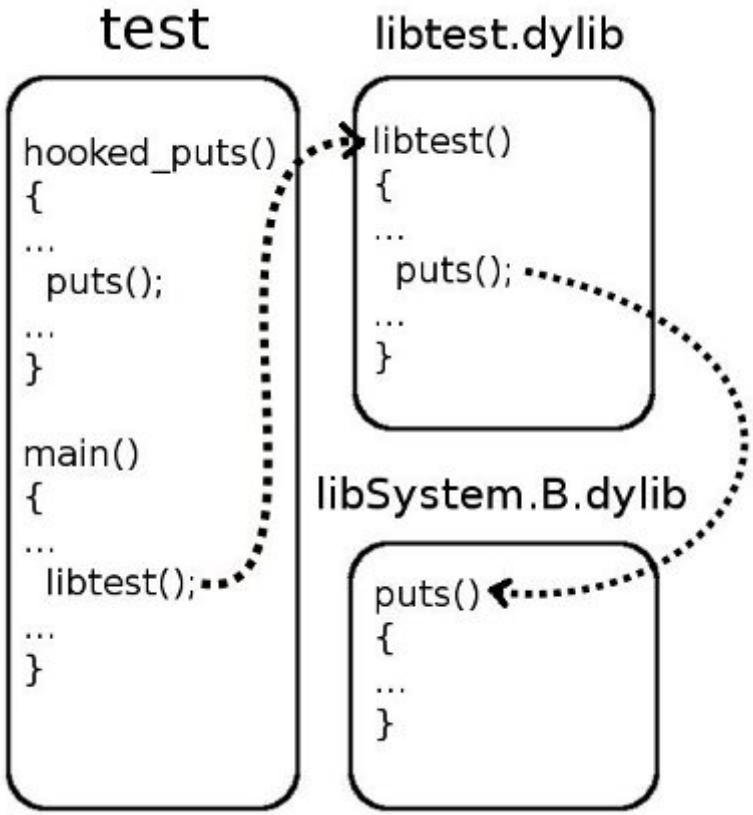
We have a program under Mac OS X that is used by a number of third-party dynamically linked libraries, which, in their turn, also use functions of each other.

The task is as follows: we need to intercept the call of a certain function from one library to another and to call the original in the handler.

Test Example

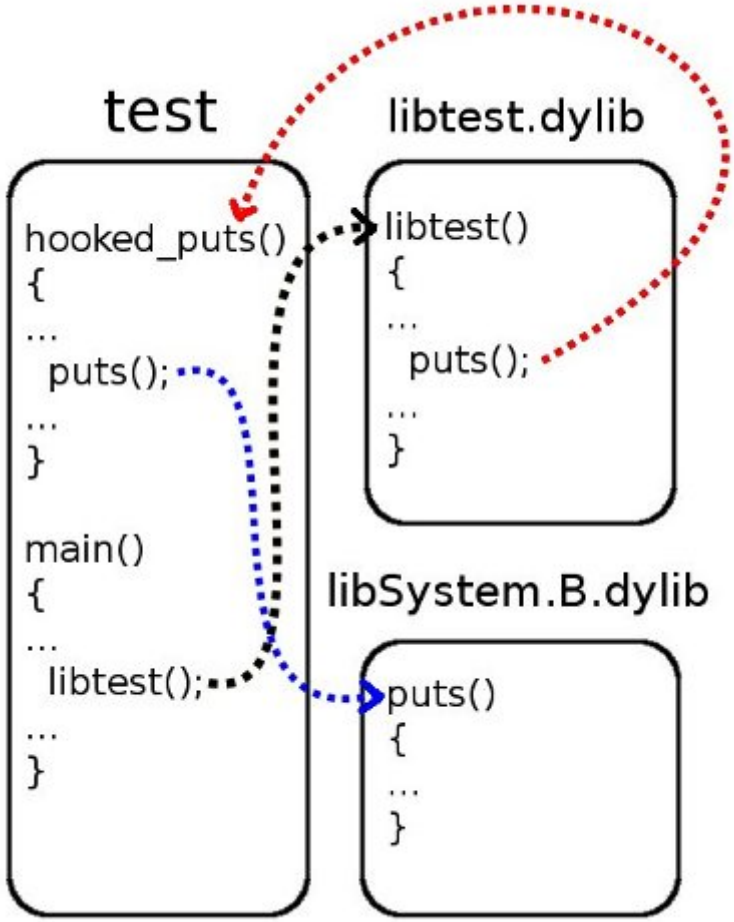
Let's take an imaginary example. Supposing we have a program called «test» and written in C language (test.c file) and a shared library (libtest.c file) with constant contents and compiled beforehand. This library implements one `libtest()` function. In their implementation, both the program and the library

use the `puts()` function from a standard library of C language (it is provided together with Mac OS and contains in libSystem.B.dylib). Let's look at the schematic view of the described situation:

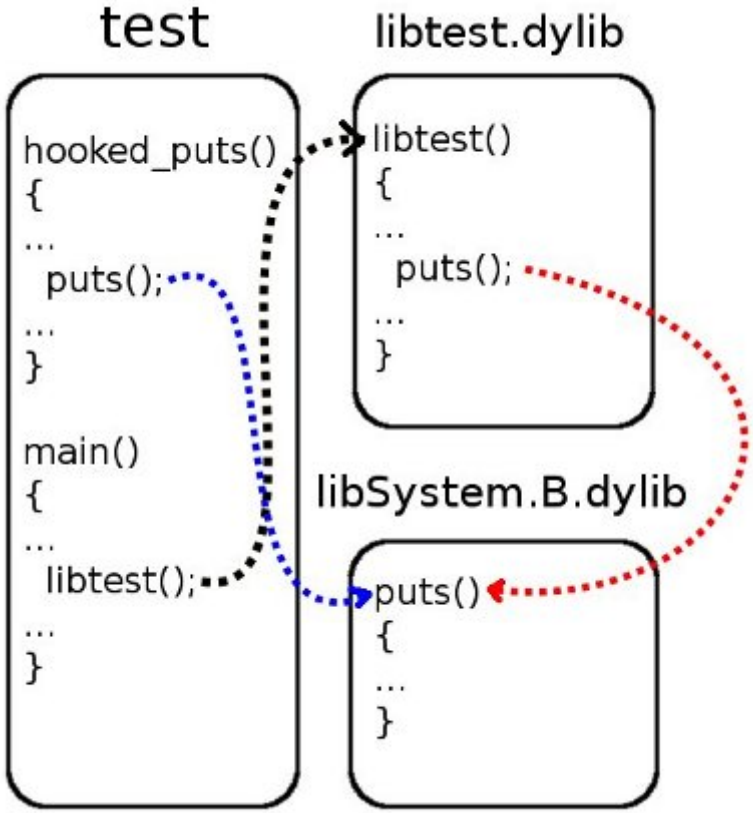


The task is the following:

1. We need to replace the call of the `puts()` function for the libtest.dylib library with the call of the `hooked_puts()` function that is implemented in the main program (test.c file). The last, in its turn, can use the original `puts()` function;



1. We need to cancel the performed changes, i.e., to make so that the repeated call of `libtest()` lead to the call of the original `puts()`.



It is not allowed to change the code or recompile the libraries, only the main program. The call redirection itself should be performed only for a specific library and on the fly, without the program restart.

Redirection Algorithm

Let's describe all actions in words because the code can turn out not so clear despite the number of comments:

1. Find the symbol table and table of strings using data from the `LC_SYMTAB` loader command.
2. From the `LC_DYSYMTAB` loader command, find out, from which element of the symbol table a subset of undefined symbols (`iundefsym` field) begins.
3. Find the target symbol by its name among the subset of undefined symbols in the symbol table.
4. Remember the index of the target symbol from the beginning of the symbol table.

5. Find the table of indirect symbols by data from the `LC_DYSYMTAB` loader command (`indirectsymoff` field).
6. Find out the index, starting from which mapping of the import table (contents of the `__DATA, __la_symbol_ptr` section; or `__IMPORT, __jump_table` — there will be one of these) to the table of indirect symbols (`reserved1` field) begins.
7. Starting from this index, we look through the table of indirect symbols and search for the value that corresponds to the index of the target symbol in the symbol table.
8. Remember the number of the target symbol from the beginning of the mapping of the import table to the table of indirect symbols. The saved value is the index of the required element in the import table.
9. Find the import table (offset field) using data from the `__la_symbol_ptr` section (or `__jump_table`).
10. Having the index of the target element in it, rewrite the address (for `__la_symbol_ptr`) to a required value (or just change the `CALL/JMP` instruction to `JMP` with an operand — address of the required function (for `__jump_table`)).

I will note that you should work with tables of symbols, strings, and indirect symbols only after loading them from the file. Also, you should read the contents of sections that describe import tables as well as perform the redirection itself in memory. It is connected with the fact that tables of symbols and tables of strings can be absent or can not display the real state in the target Mach-O. It is because the dynamic loader worked there before us and it successfully saved all necessary data about symbols without allocating the tables themselves.

Redirection Implementation

It's time to turn our thoughts into the code. Let's divide all operation into three stages for the optimization of search of the required Mach-O elements:

```
1. void  *mach_hook_init(char const *library_filename, void const *library_address);
```

Basing on the Mach-O file and its displaying in memory, this function returns some non-clear descriptor. Behind this descriptor, offsets of the import table, symbol table, table of strings, and the mapping of indirect symbols from the table of dynamic symbols as well as a number of useful indexes for this module stand. The descriptor is the following:

```
struct mach_hook_handle
{
```

```
    void const *library_address;  //base address of a library in memo
ry
    char const *string_table;  //buffer to read string_table table fr
om file
    struct nlist const *symbol_table;  //buffer to read symbol table
 from file
    uint32_t const *indirect_table;  //buffer to read the indirect sy
mbol table in dynamic symbol table from file
    uint32_t undefined_symbols_count;  //number of undefined symbols
 in the symbol table
    uint32_t undefined_symbols_index;  //position of undefined symbol
s in the symbol table
    uint32_t indirect_symbols_count;  //number of indirect symbols in
 the indirect symbol table of DYSYMTAB
    uint32_t indirect_symbols_index;  //index of the first imported s
ymbol in the indirect symbol table of DYSYMTAB
    uint32_t import_table_offset;  //the offset of (__DATA, __la_symb
ol_ptr) or (__IMPORT, __jump_table)
    uint32_t jump_table_present;  //special flag to show if we work w
ith (__IMPORT, __jump_table)
};
```

```
2. mach_substitution mach_hook(void const *handle, char const *function_name, mach_subs
titution substitution);
```

This function performs the redirection by the algorithm described above using the existing library descriptor, name of the target symbol, and address of the interceptor.

```
3. void  mach_hook_free(void *handle);
```

In this way, cleanup of any descriptor returned by `mach_hook_init()` is performed.

Taking into account these prototypes, we need to rewrite the test program:

```
#include <stdio.h>
#include <dlfcn.h>
#include "mach_hook.h"
```

```
#define LIBTEST_PATH "libtest.dylib"
void libtest();  //from libtest.dylib
int hooked_puts(char const *s)
{
    puts(s);  //calls the original puts() from libSystem.B.dylib, bec
ause our main executable module called "test" remains intact
    return puts("HOOKED!");
}
int main()
{
    void *handle = 0;  //handle to store hook-related info
    mach_substitution original;  //original data for restoration
    Dl_info info;
    if (!dladdr((void const *)libtest, &info))  //gets an address of
 a library which contains libtest() function
    {
        fprintf(stderr, "Failed to get the base address of a library!
\n", LIBTEST_PATH);
        goto end;
    }
    handle = mach_hook_init(LIBTEST_PATH, info.dli_fbase);
    if (!handle)
    {
        fprintf(stderr, "Redirection init failed!\n");
        goto end;
    }
    libtest();  //calls puts() from libSystem.B.dylib
    puts("------------------------------");
    original = mach_hook(handle, "puts", (mach_substitution)hooked_pu
ts);
    if (!original)
    {
        fprintf(stderr, "Redirection failed!\n");
        goto end;
    }
    libtest();  //calls hooked_puts()
    puts("------------------------------");
    original = mach_hook(handle, "puts", original);  //restores the o
riginal relocation
    if (!original)
    {
        fprintf(stderr, "Restoration failed!\n");
```

```
        goto end;
    }
    libtest();  //again calls puts() from libSystem.B.dylib
end:
    mach_hook_free(handle);
    handle = 0;  //no effect here, but just a good advice to prevent
 double freeing
    return 0;
}
```

Test Start

We can test it in the following way:

```
user@mac$ arch -i386 ./test
libtest: calls the original puts()
-----------------------------
libtest: calls the original puts()
HOOKED!
-----------------------------
libtest: calls the original puts()
user@mac$ arch -x86_64 ./test
libtest: calls the original puts()
-----------------------------
libtest: calls the original puts()
HOOKED!
-----------------------------
libtest: calls the original puts()
```

The program output indicates the full execution of the task that was formulated in the beginning.

The full implementation of the test example together with the redirection algorithm and the project file are attached to the article.

Take a look at another article created by our MacOS specialists and discussing OS X reverse engineering.