分析 fishhook

Mach-O 与动态链接围绕 Mach-O 结构分析了符号的动态绑定逻辑；搞清楚了这个，再来分析 fishhook 就是一件非常容易的事情了，何况，其代码量不足 300 行。

关于 fishhook，官方资料是这么介绍的：

> fishhook is a very simple library that enables dynamically rebinding symbols in Mach-O binaries running on iOS in the simulator and on device.

简单来说，通过 fishhook，可以对动态链接的符号进行重新绑定。

从 fishhook.h 的 API 上看，它定义了两个函数：

```
int rebind_symbols(struct rebinding rebindings[], size_t re

int rebind_symbols_image(void *header,
                         intptr_t slide,
                         struct rebinding rebindings[],
                         size_t rebindings_nel);
```

这两个函数都用于符号重绑定，前者操作的对象是进程的所有镜像，后者操作的对象是某个指定的镜像；一般都只是使用前者。本文也只是对`rebind_symbols()`展开进一步描述，它有两参数，`rebindings`是一个`rebinding`数组，`rebindings_nel`描述数组的长度。fishhook 使用`rebinding`结构体描述要 rebind 的目标符号：

```
struct rebinding {
  const char *name;
  void *replacement;
  void **replaced;
};
```

搞清楚了 API，使用就非常简单了。如下使用一个小 case 描述 fishhook 的使用姿势，这个 case 要做的事情是重定位printf函数的符号，让它指向到自定义函数，代码如下：

```
#include <stdio.h>
#include <stdarg.h>
#include "fishhook.h"

static int (*original_printf)(const char *, ...);

int god_printf(const char *format, ...)
{
    int ret = 0;

    original_printf("I'm God\n");

    va_list arg;
    va_start(arg, format);
    ret = vprintf(format, arg);
    va_end(arg);

    return ret;
}
```

```
int main(void)
{

    struct rebinding printf_rebinding = { "printf", god_pri
    rebind_symbols((struct rebinding[1]){printf_rebinding},



    printf("The answer to Life, the Universe and everything

    return 0;
}
```

上述代码自定义了 print-like 函数`god_printf`，在该函数里，基于`vprintf`，自定义了 print 逻辑，在此之前，还调用`original_printf`打印了几个字符，`original_printf`存储的是原来的`printf`的地址；`main`函数所做的事情，不过是使用 fishhook 的`rebind_symbols()`函数对`printf`符号进行 rebind。

跑一下程序：

```
$ gcc main.c fishhook.c -o main.out
$ ./main.out
I'm God
The answer to Life, the Universe and everything, is 42
```

> fishhook 官网对它的使用场景明确限定在 iOS 系统，但如上实践说明，macOS 端的 C 程序也是可以使用的。

正如所期待的那样，对`printf`的调用，实际上执行的是`god_printf()`。

搞清楚了基本用法，是时候分析 fishhook.c 源码了。

逐行分析解读 fishhook.c 源码的博客非常多，本文不做类似的事情；正如本文开头所述，如果弄清楚 Mach-O 的符号动态绑定逻辑，理解 fishhook.c 是一件非常容易的事情；换句话说，在分析 fishhook 之前，笔者认为应该先理清 Mach-O 的符号动态绑定逻辑。

fishhook/README介绍了其实现原理。简单复述一下：Mach-O 访问其他 dylib 的符号是以间接的方式进行的，经过各种兜转，最终符号地址存在可读写的 __DATA segment 的某个 section 中，fishhook 的实现原理就是通过修改这些 section 内容，进而实现符号的 rebind。

在分析 fishhook.c 源码之前，笔者关心如下几个问题：

- fishhook 的 rebind 只对函数型符号有效，还是也可以作用于数据型符号?
- fishhook 是如何找到目标 section 的?

> 存储函数型动态链接符号地址值的 section 一般是`__la_symbol_ptr`（详见Mahc-O 与动态链接），fishhook 是如何找到它的呢?

- fishhook 是如何匹配符号名的呢?

> 上文要处理的函数`printf()`，编译器处理后，其符号名通常变为了`_printf`，fishhook 是如何匹配

> 的呢?

fishhook.c 简短的源码中，执行 rebind 逻辑的核心函数有两个：`rebind_symbols_for_image`和`perform_rebinding_with_section`；前者负责找到目标 section，后者在 section 里根据符号进行真正的 rebind。

先看`rebind_symbols_for_image`，截取部分代码如下：

```
static void rebind_symbols_for_image(struct rebindings_entr
                                     const struct mach_head
                                     intptr_t slide) {

  cur = (uintptr_t)header + sizeof(mach_header_t);
  for (uint i = 0; i < header->ncmds; i++, cur += cur_seg_c
    cur_seg_cmd = (segment_command_t *)cur;
    if (cur_seg_cmd->cmd == LC_SEGMENT_ARCH_DEPENDENT) {

      if (strcmp(cur_seg_cmd->segname, SEG_DATA) != 0 &&
          strcmp(cur_seg_cmd->segname, SEG_DATA_CONST) != 0
        continue;
      }
      for (uint j = 0; j < cur_seg_cmd->nsects; j++) {
        section_t *sect = (section_t *)(cur + sizeof(segmen

        if ((sect->flags & SECTION_TYPE) == S_LAZY_SYMBOL_P
          perform_rebinding_with_section(rebindings, sect,
        }
        if ((sect->flags & SECTION_TYPE) == S_NON_LAZY_SYMB
          perform_rebinding_with_section(rebindings, sect,
        }
      }
    }
```

```
  }
}
```

聚焦寻找目标 section的逻辑，可以看到，fishhook 通过 section type 匹配来寻找目标 section，mach-o/loader.h 定义的 section header（描述 section 结构，结构体为`section_64`或者`section`），其中有一个`flags`字段，该字段含有描述 section type 的信息，如下罗列了上文 main.out 的`__nl_symbol_ptr`、`__got`、`__la_symbol_ptr`的`flags`值信息：

```
Section
  sectname __nl_symbol_ptr
   segname __DATA
     flags 0x00000006

Section
  sectname __got
   segname __DATA
     flags 0x00000006

Section
  sectname __la_symbol_ptr
   segname __DATA
     flags 0x00000007
```

0x06、0x07 分别对应的宏是`S_NON_LAZY_SYMBOL_POINTERS`、`S_LAZY_SYMBOL_POINTERS`，前者指该 section 用于存储 non-lazy 型符号地址信息，后者指该 section 用于存储 lazy 型符号地址信息。

至此，可以得到两点重要信息。

其一，fishhook 寻找`__la_symbol_ptr`等 section 的逻辑并不是通过 name 匹配，而是通过 section type 匹配。

> 在看代码之前，笔者还担心是通过 name 匹配，如果是这样，也太 low 了，万一编译器编译时将该 section 名做一下变更，岂不 gg 了？

其二，fishhook rebind 的对象不光是函数型符号，还包括数据型符号。

如下 case 旨在验证第二点，先定义一个简单的文件 hello.c，该文件非常简单，只是定义了一个全局字符串：

```
char *kHello = "Hello";
```

接着把该文件编译成动态库：

```
$ gcc -fpic -shared hello.c -o libhello.dylib
```

再写个 main.c，该模块依赖 libhello.dylib 的`kHello`，如下：

```
#include <stdio.h>
#include "fishhook.h"
```

```
extern char *kHello;
char *kGood = "Good";

int main(void)
{

    struct rebinding hello_rebinding = { "kHello", &kGood,
    rebind_symbols((struct rebinding[1]){hello_rebinding},

    printf("%s, Jason\n", kHello);
    return 0;
}
```

执行`gcc main.c fishhook.c -o main.out -L. -lhello`生成 main.out，执行程序打印`Good, Jason`（而不是`Hello, Jason`），说明`kHello`符号被成功 rebind 了。

最后，再看看`perform_rebinding_with_section`，聚焦于 symbol name 匹配逻辑：

```
static void perform_rebinding_with_section() {
  for (uint i = 0; i < section->size / sizeof(void *); i++)
    while (cur) {
      for (uint j = 0; j < cur->rebindings_nel; j++) {

        if (symbol_name_longer_than_1 && strcmp(&symbol_nam
          indirect_symbol_bindings[i] = cur->rebindings[j].
        }
      }
      cur = cur->next;
    }
  }
```

```
}
```

发现一个尴尬的事实是，fishhook 的 symbol 匹配也没能做到多么高级，仍然是字符串匹配，只是匹配前，它把符号真正的 name 的第一个字符给去掉了，所以上文匹配`printf`/`kHello`对应的符号时，符号名无需写成`_printf`/`_kHello`。

> 这种处理，我认为算是 fishhook 的一个小不足吧，毕竟它完全依赖于编译器对符号的取名姿势。但哪能找到更好的处理姿势呢?

小结

结合上文的分析，站在使用的角度，对 fishhook 做个小结：

- fishhook 能 rebind 的符号必须存在于动态库中，换句话说，它无法对本地符号进行 rebind
- fishhook 既能处理函数型符号，也能处理数据型符号（无论是全局变量还是全局常量）
- 使用 fishhook 处理符号时，传参中的符号名并不是真正的符号名，譬如你想对`_printf`符号进行 rebind，传入`"printf"`即可