Playing with Mach-O and DYLD

A peek into macOS

Stanislas `Plkachu` Lejay

March 14, 2017

LSE Security System
Laboratory of Epita

Initial goal

- Learn something about macOS internals
- Learn how Apple tries to secure iOS and macOS
- How do these measures differ from the one we already know about ?
- Learn about Mach-O executables as a vector of exploitation and for fun
- How to play with them?

Where to begin

1. Take something simple that works but few really know how
2. Find a way to turn it into something overly complicated
3. And well, do it

Our lab's pick: calling 'printf'

- Look for the mapped **libc** by using the system and dynamic linker's structures only (ninja mode: no syscall)
- Use the executable format to find **printf**'s address inside.

Looking for the libc

LSE Security System
Laboratory of Epita

Determining where printf is

- gdb tells us to look into **/usr/lib/system/libsystem_c.dylib**
- This is a Mach-O universal binary (fat binary)
- Extract executable corresponding to our architecture, and find where it is mapped

One solution: exploring DYLD

- macOS/iOS classic dynamic linker - open source
- Interesting from a security perspective (often abused)
- Relevant headers in **/usr/include/mach[-o]/**

dyld_image_info

- Represents an image's memory mapping
- Direct access to image's name and loaded address (ASLR independent)
- Stored in an array in **dyld_all_image_infos** structure

```
struct dyld_image_info {
        /* base address image is mapped into */
        const struct mach_header* imageLoadAddress;
        /* path dyld used to load the image */
        const char*               imageFilePath;
        /* time_t of image file */
        uintptr_t                 imageFileModDate;
// ...
};
```

from /usr/include/mach-o/dyld_images.h

dyld_all_image_infos

- The almighty structure
- Direct access to a process's address space and linker informations
- Quite subject to changes

```
struct dyld_all_image_infos {
        uint32_t                          version;  /* 1 in Mac OS X 10.4 and 10.5 */
        uint32_t                          infoArrayCount;
        const struct dyld_image_info* infoArray;
        dyld_image_notifier               notification;
        bool                              processDetachedFromSharedRegion;
        /* Mac OS X 10.6, iPhoneOS 2.0 and later */
        bool                              libSystemInitialized;
        const struct mach_header*         dyldImageLoadAddress;
        /* Mac OS X 10.6, iPhoneOS 3.0 and later */
        void*                             jitInfo;
        /* Mac OS X 10.6, iPhoneOS 3.0 and later */
        const char*                       dyldVersion;
// ...
};
```

from /usr/include/mach-o/dyld_images.h

Getting dyld_all_image_infos

- To get it, we need to call the **task_info** function

```
kern_return_t task_info
(
    task_name_t target_task,
    task_flavor_t flavor,
    task_info_t task_info_out,
    mach_msg_type_number_t *task_info_outCnt
);
```

- Returns informations regarding the **flavor** argument
- If **flavor** is **TASK_DYLD_INFO**, we obtain a **struct task_dyld_info**

```
struct task_dyld_info {
    mach_vm_address_t   all_image_info_addr;
    mach_vm_size_t      all_image_info_size;
    integer_t       all_image_info_format;
};
```

from /usr/include/mach/task_info.h

Putting it all together

- We first get the **task_dyld_info** structure

```
// Get DYLD task infos
struct task_dyld_info dyld_info;
mach_msg_type_number_t count = TASK_DYLD_INFO_COUNT;
kern_return_t ret;
ret = task_info(mach_task_self_,
                TASK_DYLD_INFO,
                (task_info_t)&dyld_info,
                &count);
if (ret != KERN_SUCCESS) {
        return NULL;
}
```

Putting it all together

- We then get the **dyld_image_info** array

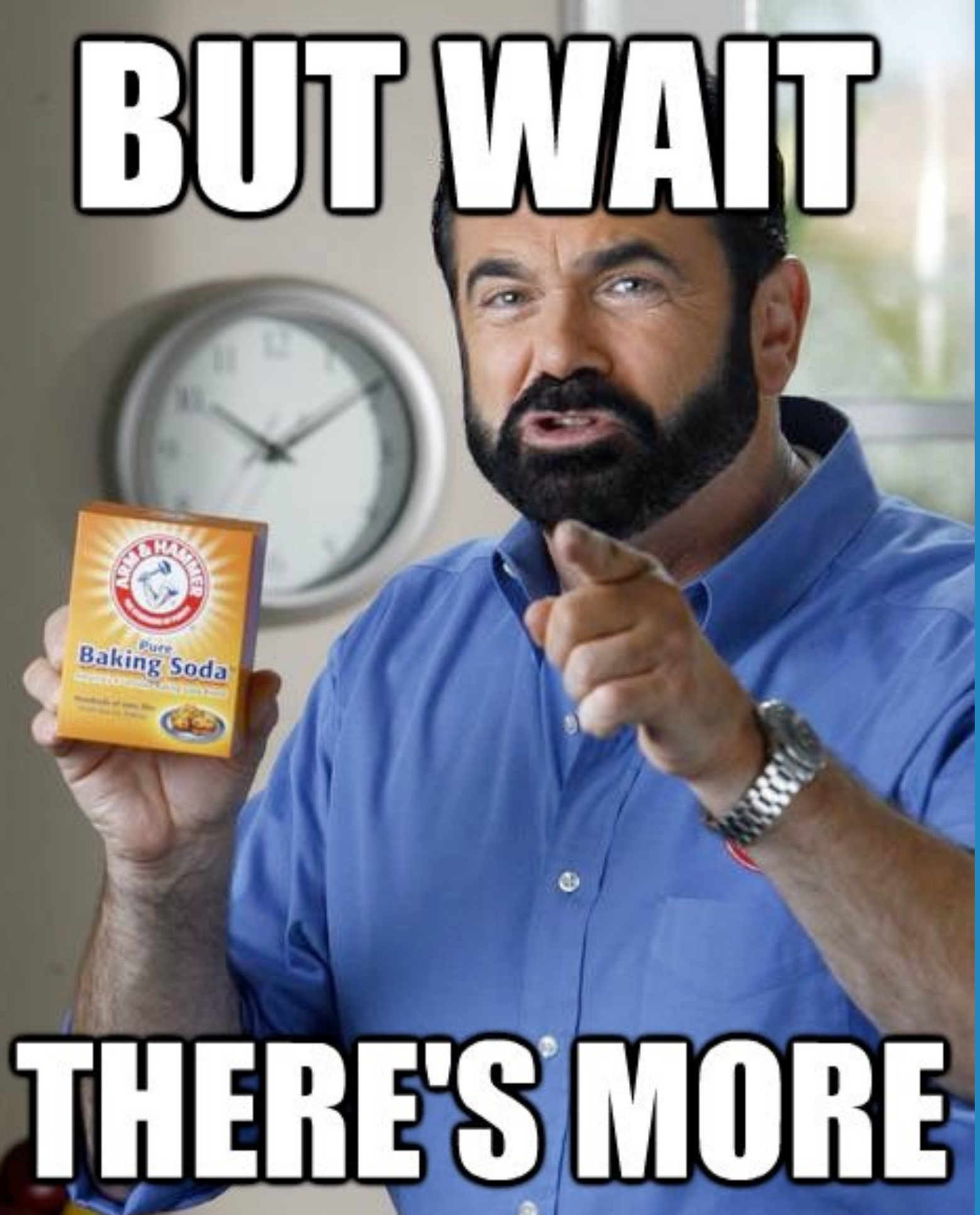
```
// Get image array's size and address
mach_vm_address_t image_infos = dyld_info.all_image_info_addr;
struct dyld_all_image_infos *infos;
infos = (struct dyld_all_image_infos *)image_infos;
uint32_t image_count = infos->infoArrayCount;
struct dyld_image_info *image_array = infos->infoArray;
```

Putting it all together

- And finally we look for **libsystem_c.dylib**

```
// Find libsystem_c.dylib among them
struct dyld_image_info *image;
for (int i = 0; i < image_count; ++i) {
        image = image_array + i;

        // Find libsystem_c.dylib's load address
        if (strstr(image->imageFilePath, "libsystem_c.dylib")) {
                return (char*)image->imageLoadAddress;
        }
}
```



Finding printf

LSE
Security System
Laboratory of Epita

libsystem_c

- Now we have the libc binary (Mach-O format)
- We need to find printf's offset in it
- So we need to observe the libc's exported symbols
- How does one, and particularly DYLD, perform this ?

EL MACHO

Security System

Laboratory of Epita

The Mach-O Format in 2 slides

- One Mach-O header
- Multiple Load commands
- One or more segments, each containing [0, 255] sections

```
P1kachu@GreyLabOfSteel:LT$ ccat test.c
#include <stdio.h>
int main()
{
        printf("Yo\n");
        return 0;
}
P1kachu@GreyLabOfSteel:LT$ gcc test.c -o macho
P1kachu@GreyLabOfSteel:LT$ ./jtool -h macho
Magic:  64-bit Mach-O
Type:   executable
CPU:    x86_64
Cmds:   15
size:   1280 bytes
Flags:  0x200085
```

```
P1kachu@GreyLabOfSteel:LT$ ./jtool -l macho
LC 00: LC_SEGMENT_64                Mem: 0x000000000-0x100000000      __PAGEZERO
LC 01: LC_SEGMENT_64                Mem: 0x100000000-0x100001000      __TEXT
       Mem: 0x100000f50-0x100000f7a             __TEXT.__text   (Normal)
       Mem: 0x100000f7a-0x100000f80             __TEXT.__stubs  (Symbol Stubs)
       Mem: 0x100000f80-0x100000f9a             __TEXT.__stub_helper    (Normal)
       Mem: 0x100000f9a-0x100000f9e             __TEXT.__cstring        (C-String Literals)
       Mem: 0x100000fa0-0x100000fe8             __TEXT.__unwind_info
       Mem: 0x100000fe8-0x100001000             __TEXT.__eh_frame
LC 02: LC_SEGMENT_64                Mem: 0x100001000-0x100002000      __DATA
       Mem: 0x100001000-0x100001010             __DATA.__nl_symbol_ptr  (Non-Lazy Symbol Ptrs)
       Mem: 0x100001010-0x100001018             __DATA.__la_symbol_ptr  (Lazy Symbol Ptrs)
LC 03: LC_SEGMENT_64                Mem: 0x100002000-0x100003000      __LINKEDIT
LC 04: LC_DYLD_INFO
LC 05: LC_SYMTAB
       Symbol table is at offset 0x2068 (8296), 4 entries
       String table is at offset 0x20b8 (8376), 56 bytes
LC 06: LC_DYSYMTAB                  No local symbols
           2 external symbols at index  0
           2 undefined symbols at index 2
          No TOC
          No modtab
           4 Indirect symbols at offset 0x20a8

LC 07: LC_LOAD_DYLINKER             /usr/lib/dyld
LC 08: LC_UUID                      UUID: C9F1B6CD-2856-3C22-A30A-DD7717523092
LC 09: LC_VERSION_MIN_MACOSX        Minimum OS X version:    10.10.0
LC 10: LC_SOURCE_VERSION            Source Version:          0.0.0.0.0
LC 11: LC_MAIN                      Entry Point:             0xf50 (Mem: 0x100000f50)
LC 12: LC_LOAD_DYLIB                /usr/lib/libSystem.B.dylib
LC 13: LC_FUNCTION_STARTS           Offset: 8288, Size: 8 (0x2060-0x2068)
LC 14: LC_DATA_IN_CODE              Offset: 8296, Size: 0 (0x2068-0x2068)
P1kachu@GreyLabOfSteel:LT$
```

Parsing a bit

- We are interested in the **LC_SYMTAB** load command
- It gives the symbol and string table offsets from the executable base.
- Well, that's easy. Perhaps too easy.

Parsing the symtab

- With **jtool**, we can see which values we are supposed to find:

```
P1kachu@GreyLabOfSteel:LT$ ./jtool -arch x86_64 -l /usr/lib/system/libsystem_c.dylib | grep -A2 LC_SYMTAB
LC 05: LC_SYMTAB
        Symbol table is at offset 0x9fd18 (654616), 2302 entries
        String table is at offset 0xaa3e0 (697312), 31024 bytes
```

- However, the values found in memory differ a bit...

```
P1kachu@GreyLabOfSteel:LT$ ./jtool -arch x86_64 -l /usr/lib/system/libsystem_c.dylib | grep -A2 LC_SYMTAB
LC 05: LC_SYMTAB
        Symbol table is at offset 0x9fd18 (654616), 2302 entries
        String table is at offset 0xaa3e0 (697312), 31024 bytes
P1kachu@GreyLabOfSteel:LT$ ./get_symcmd
symoff: 0x134596ef
stroff: 0x141ad9f4
P1kachu@GreyLabOfSteel:LT$
```

The shared cache

- Back in 2009 (iOS 3.1), the **DYLD shared cache** was introduced as a new way to handle system libraries
- It combines all system libraries into a big file, mapped system-wide at boot, to improve overall performance
- It lives in **/private/var/db/dyld** and regroups a lot of libraries (~400 for Yosemite and ~670 for Sierra)
- File format not documented, and subject to changes between versions
- The offset found in memory thus are from the shared cache's base address, since libsystem_c is, as it name explains, a system library

So we need to find it's base

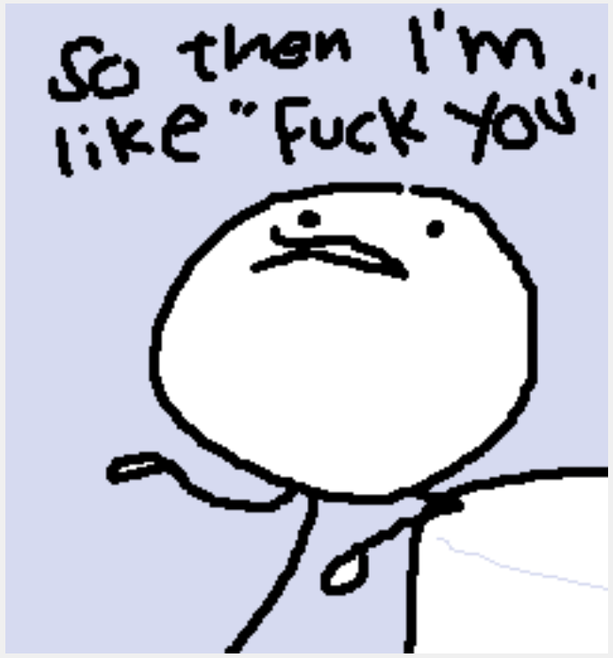
- The stupid method (Yosemite):

  1. Find the loaded library with the smallest base address
  2. Walk back into memory until finding the shared cache magic string (**dyld_v1 x86_64\0**)

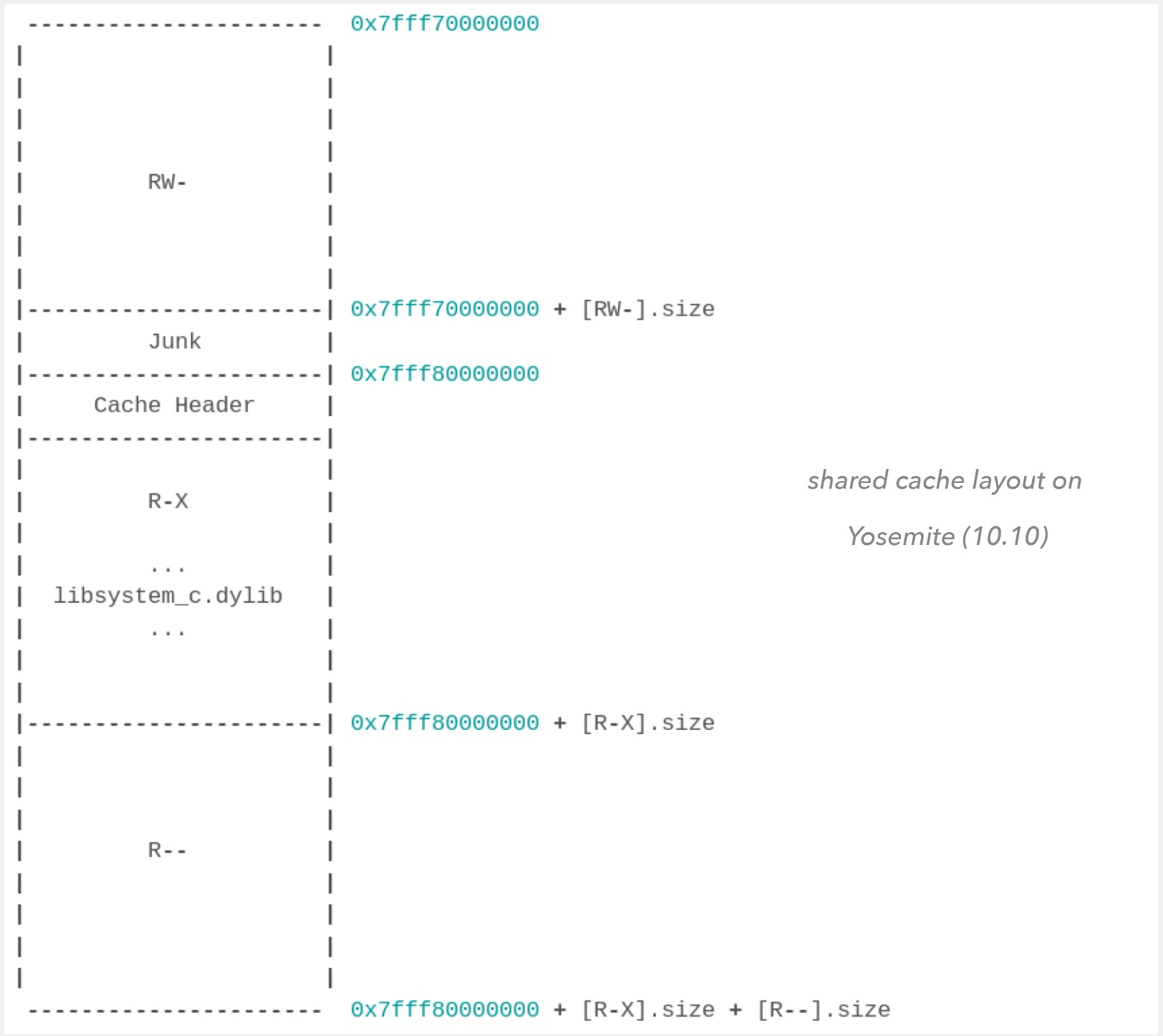
- The easy method (Sierra): the **dyld_all_image_infos** structure comes with the **sharedCacheBaseAddress** field :)

Subtleties

- On Yosemite, the file layout of the shared cache does not correspond to its memory mapping
- The DATA mapping is above the cache header
- The magic string thus doesn’t correspond to the base address we should take into account to get the symbols
- On Sierra, things are back in order



cache.base = [R-X].address + [R-X].size - [R--].offset



shared cache layout on Yosemite (10.10)

Finding printf

LSE Security System
Laboratory of Epita

Final steps

- We finally have everything we need and just have to deduce **printf**'s address.
- symtab entries are **struct nlist[_64]**. We are interested in the **n_un.n_strx** and the **n_value** fields.

```c
/*
 * This is the symbol table entry structure for 64-bit architectures.
 */
struct nlist_64 {
    union {
        uint32_t  n_strx; /* index into the string table */
    } n_un;
    uint8_t n_type;        /* type flag, see below */
    uint8_t n_sect;        /* section number or NO_SECT */
    uint16_t n_desc;       /* see <mach-o/stab.h> */
    uint64_t n_value;      /* value of this symbol (or stab offset) */
};
```

from /usr/include/mach-o/nlist.h

Final steps

```
uint64_t aslr_slide = (uint64_t)shared_cache_rx_base - rx_addr;

char *shared_cache_ro = (char*)(ro_addr + aslr_slide);
uint64_t stroff_from_ro = symcmd->stroff - rx_size - rw_size;
uint64_t symoff_from_ro = symcmd->symoff - rx_size - rw_size;


struct nlist_64 *symtab;
char *strtab = shared_cache_ro + stroff_from_ro;
symtab = (struct nlist_64 *)(shared_cache_ro + symoff_from_ro);

for(uint32_t i = 0; i < symcmd->nsyms; ++i){
        uint32_t strtab_off = symtab[i].n_un.n_strx;
        uint64_t func       = symtab[i].n_value;

        if(strcmp(&strtab[strtab_off], "_printf") == 0) {
                return (char*)(func + aslr_slide);
        }
}
```

aaaaaand...

```
246
247         [...]
248
249         char *printf_addr = find_printf(libc, shared_cache_rx_base);
250         if (!printf_addr) {
251                 return 1;
252         }
253
254         void (*print)(const char *fmt, ...) = (void *)printf_addr;
255         print("Gotcha\n");
256
257         return 0;
258 }
```

Gotcha!

```
P1kachu@GreyLabOfSteel:calling-printf$ ./osx/catch_printf
Gotcha
P1kachu@GreyLabOfSteel:calling-printf$
```

Bonus question

- Let’s think about something: the shared cache is loaded at boot system-wide. What does this mean ?

References and code

- http://newosxbook.com/articles/DYLD.html
- https://www.gnu.org/software/hurd/gnumach-doc/Task-Information.html
- http://timetobleed.com/dynamic-symbol-table-duel-elf-vs-mach-o-round-2/
- https://www.objc.io/issues/6-build-tools/mach-o-executables/
- http://antid0te.com/POC2010-Adding-ASLR-To-Jailbroken-iPhones.pdf (a bit old but meh.)
- **/usr/include/mach-o/***

Final code available:

https://gist.github.com/P1kachu/e6b14e92454a87b3f9c66b3163656d09

Thank you

plkachu@lse.epita.fr
@0xplkachu