iPhone Library Loader Patcher - Documentation

Nicolas A. Economou
Core Security Technologies

WARNING: Don't replace the original loader dyld with the patched one (dyle), modify your binaries to use it.
Usage

```
dyld_patcher v1.01 for iPhone v1.1.4
Created by Nicolas A. Economou ( neconomou@corest.com )
Core Security Technologies, Buenos Aires, Argentina ( 2008 )
usage: dyld_patcher dyld_source dyld_dest
example: dyld_patcher /usr/lib/dyld dyle
```

Detailed Description

This quest begins when the iPhone mail client was being debugged, which runs on process MobileMail. We realized that we can't insert breakpoints on libc functions like recv, read, etc., this was pretty upsetting.

The libc is loaded as a shared library, that is, all processes can share it as an unique memory area without multiple copies being loaded. At this point, we realized that some code was deciding if each library is loaded shared or not.

Looking at the header of the executable files running on iPhone, we also realized that the files are of type Mach-O, that means the same used on MacOS X for Intel and PPC. This was very predictable considering that iPhone runs a modified MacOS X.

Inside the header of any executable, there is a section called

LINKEDIT

. Inside this section are located all the libraries imported by the application, the first one, is the loader /usr/lib/dyld.

So we thought of reversing the loader to know in which exact moment the library loading is done and with which access restrictions.
Obviously, this tests were done making a copy of the original loader (/usr/lib/dyle) and modifying a "hello world" test application we have, telling this application to use /usr/lib/dyle now and not /usr/lib/dyld as before.

Knowing that the loader is a library that is loaded at the beginning and that is in charge of mapping all the memory areas of the involved libraries, it was predictable that we can't attach to it, due to its execution speed, because is the OS whom starts it. One possible solution was to use the old trick of patching with while(1); the loader, to have enough time to attach the debugger.

The central issue at this point is to find an adecuated memory location to modify. Well, analizing the code with IDA 5.2, we could observe that the loader includes a static compilation of function _vm_protect. This function is referenced by two others functions, the first one called Segment::setPermissions(). This was a good starting point, because at some time, the loader must be setting the permissions of each memory area loaded. So, the beginning of function _vm_protect was chosen to patch introducing opcodes for while(1);.
From this point on, using the iphonedbg debugger we manage to attach to the process, we restored the patched memory and began to debug it until reaching the function that made the mapping of the imported libraries into memory.

The function inside the loader (dyld) deciding if a file is loaded shared or not is ImageLoaderMachO::sharedRegionMapFilePrivate (). This function has a boolean parameter to choose loading the library with mmap or with an undocumented system call. By default, the system call is used, this loads the libraries with only read permissions. After getting out of this function, the memory region is already mapped, but the debugger wouldn't let you write in it.

One of the function parameters is an array:

```
struct _shared_region_mapping_np {
    mach_vm_address_t   address;
    mach_vm_size_t      size;
    mach_vm_offset_t    file_offset;
    vm_prot_t           max_prot;   /* read/write/execute/COW/ZF */
    vm_prot_t           init_prot;  /* read/write/execute/COW/ZF */
};
```

Modifying the boolean flag to use the code that calls mmap we managed to load the libraries with write permission. The only difference between mmap and the undocumented system call is that the latter uses flag VM_PROT_ZF, which indicates if a mapped memory region should be initialized with zeros or not.

So then, the idea was to patch the code of function ImageLoaderMachO::sharedRegionMapFilePrivate () to always use mmap and after that calling function memset initialize the segments that use as a "protection" measure flag VM_PROT_ZF .

Detailed Explanation

This is the sequence of steps to patch the loader following the ideas in the previous section:

```
1. REPLACE THE CONDITIONAL JUMP THAT CHECKS THE VALUE OF THE BOOLEAN VARIABLE WITH A NOP,
BECAUSE IF THE CONDITION IS NOT TRUE THEN THE FOLLOWING ASSEMBLER BASIC BLOCK IS THE ONE
GOING TO FUNCTION mmap.
2. REPLACE THE CONDITIONAL JUMP THAT DECIDES NOT TO CALL mmap IF THE VM_PROT_ZF IS TRUE,
ALSO WITH A NOP.
3. IN THE BASIC BLOCK THAT CALLS mmap, PATCH INSTRUCTION "AND R12, R3, #4" REPLACING IT
WITH "AND R12, R3, #0XFF". IN REGISTER R3 ARE STORED THE PERMISSIONS OF THE MEMORY ZONE
TO MAP, ( VM_PROT_READ, VM_PROT_WRITE, VM_PROT_EXECUTE, VM_PROT_COW , VM_PROT_ZF ), THE
PURPOSE IS TO SAVE THE VALUE OF VM_PROT_ZF WHICH VALUE NOW IS 0x10.
4. IN THE FIRST INSTRUCTION OF THE NEXT BASIC BLOCK AFTER CALLING mmap, WRITE A JUMP TO
THE ORFAN BASIC BLOCK, THAT MEANS, THE ONE CALLING THE SYSTEM CALL, WHICH CAN BE USED TO
WRITE SOME INSTRUCTIONS.
5. THE CODE WRITTEN IN THE ORFAN BASIC BLOCK FOLLOWS:

        TST R2, #0x10          // tests if the flag VM_PROT_ZF is enabled
        BEQ go_on              // if is enabled jump to "go_on"
        LDR R0, [R7,#-0xA4]    // load from memory location [R7-0xa4] a pointer to a
struct array called "_shared_region_mapping_np"
        LDR R2, [R0,#4]        // load the memory block length, param. 3
        MOV R1, #0             // clean the memory zone with value zero, param. 2
        LDR R0, [R0]           // load the base address of the memory block, param. 1
        BL _memset             // call function "memset" with these parameters.
      go_on:
        LDR R0, [R7,#-0xD4]    // execute the first instruction of the original patched
basic block
        B basic_block_original // return to the original basic block
```

Notice

All this tricks are affective if you patch the executable you want to debug changing the name of the loader, replacing the name of the original loader with the name of the cracked loader, for example replace /usr/lib/dyld with /usr/lib/dyle and the same with the imported libraries you wanto to debug, make copies of them and insert the new names to avoid sharing problem with other applications.

The loader patcher can be downloaded from the project page. It was tested with firware 1.1.4.

Enjoy!