Let your Mach-O fly

Vincenzo Iozzo
vincenzo.iozzo@zynamics.com

January 15, 2009

Abstract

Mac OS X is starting to spread among users, so new exploitation techniques have to be discovered. Even if a lot of interesting ways of exploitation on OSX were presented in the past, the lack of anti-forensics techniques is clear.
The talk is focused on an in-memory injection technique. Specifically, how it is possible to inject into a victim's machine any kind of binaries ranging from your own piece of code to real applications like Safari. This is accomplished without leaving traces on the hard disk and without creating a new process, since the whole exploitation is performed in memory.
If an attacker is able to execute code in the target machine, it is possible to run this attack instead of a classic shellcode and to use it as a trampoline for higher-lever payloads.
Other similar payloads like meterpreter or meterpretux[11] exist but none of them is able to run on Mac OS X. Besides, many of these techniques require to run specific crafted binaries, that way precompiled applications are left out from the possible range of payloads.

1 Introduction

Mac OS X is constantly attracting new users. Given that fact, in the last few years researchers focused more and more on this topic highlighting a number of security concerns, mainly caused by the lack of modern security counter measures. Nonetheless there is a lack of advanced techniques and tools that are already present and used on other OSes instead. In this paper a technique, relevant both for anti-forensics and for penetration testing, is explained. Specifically the proposed technique will address two issues:

- **Definitive anti-forensics**[1] which targets the acquisition phase of a forensics investigation, ruining the evidence or making it impossible to acquire.
- **Two-stage shellcode injection** which allows an attacker to inject a high level payload in the target system.

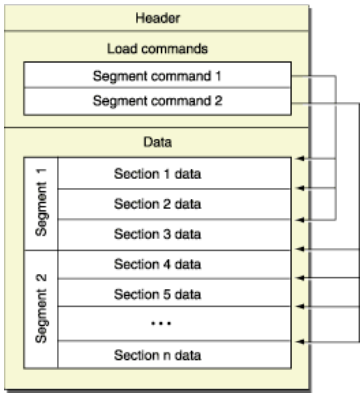

Figure 1: The format of a Mach-O file. (Copyright Apple Inc.)

The remainder of this paper is organized as follows. Section 1.1 provides an explanation of the Mach-O file format. Section 1.2 details how OSX kernel executes binaries. Section 2 describes the proposed technique. Section 3 supplies a description of a technique used to gather **non-exported** symbols from a binary, defeating libraries address space layout randomization. Section 4 briefly draws conclusions and outlines future research perspectives.

1.1 Mach-O file format specification

The Mach-O file format is the standard used by Mac OS X ABI[2] to store binaries on the disk. A Mach-O is divided into three major parts, as shown in Figure 1:

- **Header structure** which contains information on the target architecture and specified options needed to interpreter the rest of the file.
- **Load commands** which specify among other information the layout of the file in the virtual memory, the location of the symbol table, and the registers state of the main thread at the beginning of the execution.
- **Segments** which may contain from zero to two hundred fifty-five **sections**. Each segment defines a region of the virtual memory and its sections represent code or data needed to execute the binary.

Each segment contains information on the address range used in the virtual memory and protection attributes for the memory region. All segments must be aligned on virtual memory page size.

Some important segments are:

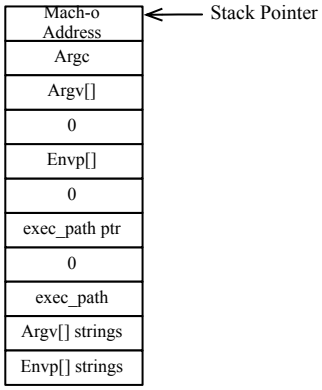

Figure 2: The stack of the binary before calling the dynamic linker.

- **__PAGEZERO** which is stored at virtual memory location 0. This segment has no protection flags assigned, therefore accessing this region will result in a memory violation.
- **__TEXT** which holds the binary code and read-only data. This segment contains only readable and executable bytes, for this reason the writing protection flag is not present. The first page of this segment also contains the header of the binary.
- **__DATA** which contains the binary data. This segment has both reading and writing protection flags set.
- **__LINKEDIT** which stores information such as the symbol table, string table, etc etc. for the linker. This segment has both reading and writing protection flags set.

Sections inherit protection flags from segments. They are used for defining the content of specific regions within segments virtual memory.

1.2 OSX binaries execution

The execution of a binary on Mac OS X is conducted by two entities : the kernel and the dynamic linker[3]. The first part of the execution process is conducted by the kernel, whereas the second by the dynamic linker. When a process is launched the kernel maps the dynamic linker at a fixed address onto the process address space. After that it parses the header structure and all segments of the binary and loads them in the correct virtual memory regions. Before calling the dynamic linker a new stack is created.
The stack layout is as it is shown in Figure 2. It should be noticed that the address of the binary is pushed into the stack in order to let the dynamic linker

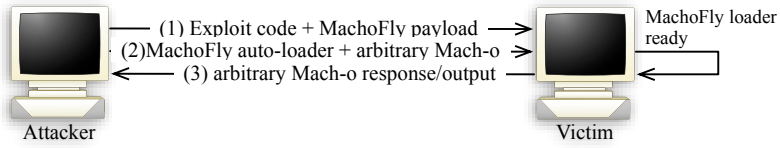

Figure 3: The attack flow graph

handle it.
During the second phase the dynamic linker parses the binary and resolves symbols, library dependencies and so forth, before jumping to the binary entry point. It must be noticed that all the libraries are recursively loaded by the dynamic linker itself, so the kernel does not play a role in this phase.

2 Proposed technique

In the past Ripe and Pluf[4] proposed an attack that is able to use userland-exec[5] on a victim's machine. Their attack encapsulated a shellcode and a crafted stack into the binary file that it is afterwards sent to the victim. Upon receiving the crafted binary the shellcode is executed and an userland-exec attack is performed. Despite its usefulness the attack suffers some problems:

- It only works with ELF files on Linux.
- It doesn't work if ASLR is enabled.
- Only static binaries can be injected.

Although our technique uses a similar method to craft the binary and it is basically an userland-exec attack, it should be considered new because both the target files and the payload construction differ hugely. Figure 3 illustrates the basic flow graph of our attack.
In this section we detail how we craft the binary to execute on the victim's machine and how our shellcode works.

2.1 Crafted binary

Nemo[6] and Roy g biv[7] separately explain a technique for inserting malicious code in the **__PAGEZERO** segment. This infection attack changes **__PAGEZERO** protection flags and stores some code at the end of the Mach-o file, mapping it at a non-allocated arbitrary address in the virtual memory.
We employ this technique to store malicious code inside the injected binary. First we create a crafted stack identical to the one shown in Figure 2.
Next we append the stack and our shellcode at the end of the file by using the **__PAGEZERO** infection technique. Finally, we overwrite the first byte of the

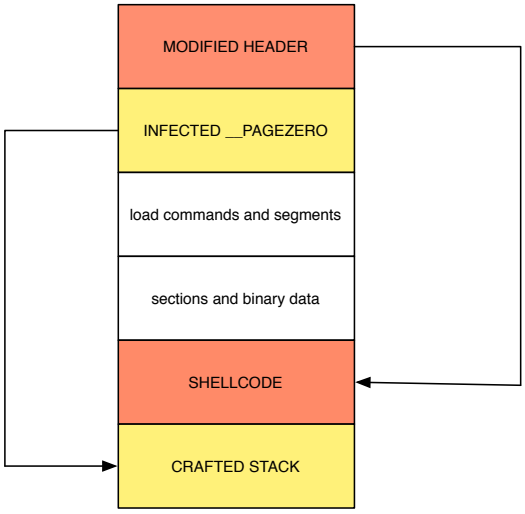

Figure 4: Injected binary layout.

header structure, which usually contains the magic number, with the address of the shellcode.
The resulting Mach-O layout is shown in Figure 4.

2.2 Shellcode specification

The role of our shellcode is to impersonate the kernel and conduct its tasks. The code parses load commands, when a segment is encountered the virtual memory used by the segment is firstly unmapped to wipe the old data contained in it, then it is mapped again with the correct protection flags, lastly the segment is copied into the right position.
A special role plays the **__PAGEZERO** segment, in fact it contains the crafted stack for the binary. When the shellcode encounters this segment it unmaps the old data and copies the new stack into it. Finally the esp pointer is stored in order to be used when the dynamic linker is called.
Other load commands are ignored as they are handled by the dynamic linker.
When the shellcode is executed in the address space of the process both the libraries allocated by the attacked process and the dynamic linker are present. The latter maintains a list of all allocated libraries; since all binaries rely on **libSystem**, our injected binary will use the one already allocated. In order to work correctly, **libSystem**, when allocated, uses some variables to initialize heaps and parse environ and argument variables. If a new process is launched and those variables are set a crash will occur while allocating memory or parsing arguments. For this reason we need to wipe these control variables in our shellcode, before calling the dynamic linker.

The last part of the shellcode is in charge of cleaning registers, adjusting the stack pointer so that the address of the binary is the first word on the stack and calling the dynamic linker entry point, which is always at a fixed address.

3 Defeat libraries address space layout randomization

As said in the previous section, one of the tasks of our shellcode is to wipe some non-exported variables used by **libSystem**. Since they are not exported we cannot easily retrieve addresses for the aforementioned variables by simply using dlopen()/dlsym() calls combination; neither it is possible to calculate their address a priori.
In fact since Leopard release Apple has introduced ASLR for libraries. When either the system or a library is updated **update_dyld_shared_cache(1)** performs the randomization[8].

In this section a method for circumventing this problem is detailed. Firstly we retrieve the addresses of some exported functions of the default dynamic linker, dyld. Then we search for **libSystem** base address and for the base address of the **__DATA** segment. Finally, we open **libSystem**, which is present on the disk, searching for the symbols we need and we adjust them with the **__DATA** segment base address. Symbol names are hashed using Dan Bernstein algorithm in order to reduce memory occupation.
In section 3.1 we provide an explanation on how to gather symbols from the dynamic linker. In section 3.2 we describe how to find non-exported symbols contained in a library mapped in the process address space.

3.1 Gathering dyld functions addresses

In order to obtain the randomized base address of a library there are two possibilities:

- Parse the file **dyld_shared_cache_i386.map** and search for library entry.
- Use some functions exported by the default dynamic linker, performing the whole task in-memory.

We have chosen the second approach as it is cleaner and less error-prone than the first one. Employed functions are:

- **_dyld_image_count()** used to retrieve the number of linked libraries of a process.
- **_dyld_get_image_header()** used to retrieve the base address of each library.
- **_dyld_get_image_name()** used to retrieve the name of a given library.

The symbol table and the string table in Mach-O files are stored in the **__LINKEDIT** segment. To gather addresses of these functions we parse the binary searching for the symbol table and retrieve the addresses of the symbols.

3.2 Retrieve non-exported symbols

Having obtained dyld's functions addresses we can retrieve the base address of **libSystem**. When a binary is executed non-exported symbols are removed from the symbol table making it impossible to compute their addresses on the fly. For that reason we divided this process into two tasks:

- We calculate the base address of the **__DATA** segment where symbols are placed, parsing the header of the **libSystem** present in the process address space.
- We open **libSystem** binary and parse the symbol tables to retrieve addresses we are interested in.

Lastly, we need to relocate symbols by using the calculated address of the **__DATA** segment. For Mac OS X Leopard we need to search for the following symbols:

- **_malloc_def_zone_state**
- **_NXArgv_pointer**
- **_malloc_num_zones**
- **__keymgr_global**

4 Conclusion

In this paper we have shown that it is possible to inject a binary of any sort in a victim's machine without either leaving traces on the hard disk or calling **execve(2)**. We think this technique is very effective and should be stopped by employing common memory protection counter-measures. Nonetheless it is still possible to detect this kind of attack by using an anomaly based IDS system[9]. Though this attack we have been able to inject a wide range of binaries from simple command line utilities like **ls** to complex applications like **Safari**. We have also demonstrated that ASLR adopted only for libraries does not block a common set of attacks.
Further developments of this work may include employing encryption to avoid NIDS detection, using cavity infector[10] to store shellcode in the injected binary and porting this technique to iPhones to evade code signing protection.

Acknowledgments

The author would like to thank, for a number of ideas and discussions Stefano Zanero, Dino Dai Zovi, Charles Miller and Halvar Flake.

References

[1] Mark Pollitt: Computer Forensics: an approach to evidence in cyberspace.

[2] Apple Inc.: Mac OS X ABI Mach-O File Format Reference.

[3] Apple Inc.: Introduction to Mach-O Programming Topics.

[4] Pluf, Ripe: Advanced Antiforensics SELF.

[5] The Grugq: The Design and Implementation of ul_exec.

[6] Nemo: Infecting the Mach-o Object Format.

[7] Roy g biv: Infecting Mach-O Files.

[8] Dino Dai Zovi: Mac OS Xploitation.

[9] S. Zanero, F. Maggi, V. Iozzo: Seeing the invisible: Forensic uses of Anomaly Detection and Machine Learning.

[10] Vincenzo Iozzo: Mach-O reversing and abusing, DeepSec 2008.

[11] Samuel Dralet, Julien Tinnes Meterpretux project