VIRUS ANALYSIS 1

DO THE MACARENA

Peter Ferrie
Symantec Security Response, USA

On 31 October 2006 we received a sample of the first parasitic infector of Mach-O files, OSX/Macarena. The file had previously been uploaded to a popular VX site. In contrast to OSX/Leap, which relied on a resource fork to contain the virus code, Macarena understands the Mach-O file format sufficiently well to parse the necessary structures correctly and inject its code directly into a file.

MACH-O FORMAT

Every Mach-O file begins with a header structure. That structure is called the mach_header. It begins with a magic number, whose value depends on the architecture on which the Mach-O file will execute. Though it is declared as a 32-bit value, it is easier to consider it as a sequence of four bytes. Thus, for the 32-bit *Intel* x86 architecture, the value is 0xCE 0xFA 0xED 0xFE. For the 32-bit *PowerPC* architecture, the value is 0xFE 0xED 0xFA 0xCE (‘feed face’). For the 64-bit *PowerPC* architecture (currently the only supported 64-bit format), the value is 0xFE 0xED 0xFA 0xCF.

Following the magic number is a value specifying the CPU family. For the *Intel* architecture, the value is 7. For the 32-bit *PowerPC* architecture, the value is 0x12. For the 64-bit *PowerPC* architecture, the value is 0x1000012. While the *Intel* and *PowerPC* architectures are the most common types that will be seen, other CPU values can be specified, such as the *VAX*, *Motorola* 680x0, *MIPS*, *ARM*, and the *Sparc*. These CPU values exist because the underlying operating system is based on a variant of *BSD*, which supports these CPUs. There is also a value that specifies the CPU subtype, to specify the required CPU more exactly. For the *Intel* and *PowerPC* architectures, a special value exists to specify that the file can run on any member of that architecture family.

The filetype field specifies the internal file format. The three most common types are: Object, Executable and Library. There are other types, such as Core, which usually contains crash-dump information; and Symbol, which contains symbol information for a corresponding binary file.

The next two fields relate to the array of ‘load commands’ that follow the mach_header structure. The first field contains the number of those load commands, and the second field contains their size.

The last field in the 32-bit mach_header structure contains a set of flags that describe some optional characteristics that can affect the loading of the file (the 64-bit *PowerPC* format has an additional reserved field for alignment purposes, but is otherwise identical to the 32-bit format). Most of the flags relate to file linking, and their effects are not relevant to the description of the virus.

Load commands exist to allow a file to specify various different characteristics within the file, including the memory layout and contents. Some of these characteristics include 32-bit and 64-bit segment descriptions, symbol table descriptions, dynamic library descriptions, dynamic linker descriptions, entrypoints for executables and libraries, and framework descriptions. Each load command contains a field that specifies the type of the command that follows, and the size of the command that follows. This allows an application to skip any command that it does not understand, or that it does not find interesting.

As far as the virus is concerned, only the segment descriptions and the executable entrypoint are relevant.

SEGMENTS

Segments are described by a structure called the segment_command. The segment_command structure begins with a segment name, followed by the address and size of the segment itself. There are two address fields, and two size fields. The first address and size fields are the virtual values (the address and size in memory), the second address and size fields are the physical values (the offset and size in the file). The term ‘segment’ in Mach-O files is roughly equivalent to the term ‘section’ in the *Windows* Portable Executable format (but in Portable Executable files, the address and size fields are in the reverse order). Interestingly, Mach-O files also contain ‘sections’, and are described in detail below.

All segments must be aligned on a 4kb boundary, otherwise a bus error occurs when attempting to load them. This is documented in *Apple*’s ABI for Mach-O files.

Following the address and size fields are two protection fields. The first field specifies the maximum protection that a segment can acquire. The second field specifies the initial protection that a segment can acquire. The possible protection values are: Read, Write and Execute. Currently, *OSX* does not implement ‘W^X’ protection (a method for the mutual exclusion of writable and executable protections, to limit the ability of some types of exploits to execute), though this might be implemented in the future. The first version of Macarena uses Read/Write/Execute protection for the segment in which it resides. Perhaps in response to the possibility of ‘W^X’, the second version of Macarena uses Read/Execute protection alone.

The next field in the segment_command structure contains the number of section data structures that follow the current segment_command structure. The final field in the segment_command structure is a set of flags. One possible flag specifies that the segment should be loaded to the top of memory; another possible flag specifies that the segment contains no relocated data.

The 64-bit version of the segment_command structure is identical in format to the 32-bit version of the segment_command structure, but with all of the address and size fields expanded from 32 bits to 64 bits.

SECTIONS

Sections are regions of memory that subdivide a segment. They are described by a section structure, and the sections within any given segment follow the segment_command structure immediately. Sections begin with a section name, followed by the name of the segment that contains it. The next four fields are the address in memory, the size and offset in the file, and the section alignment. The next two fields contain the offset of any relocation data, and the number of relocation items. The final three fields are a set of flags, and two fields whose interpretation depends on the type of section. Usually these last two fields will contain a value of zero.

The 64-bit version of the section structure is identical in format to the 32-bit version of the section structure, but with only the address and size fields expanded from 32 bits to 64 bits. This causes a slight limitation: while a segment can refer to file data beyond the 4Gb range, a section cannot.

It is legal to have a segment that contains no sections. In fact, most files contain an example of this: the __PAGEZERO segment describes a 4kb region of memory with no protection attributes set. It is intended to contain no file data, and thus be simply a virtual memory region that will cause an exception if it is accessed for any reason. Its purpose is to allow interception of certain invalid pointer usage, since that is a sign of a programming bug.

DOING THE MACARENA

While the __PAGEZERO segment is intended to contain no file data (size in file field has a value of zero), there is no reason why it cannot contain file data. Since it is really a segment like any other, if the file offset and size fields are set to any legal value, and if the segment protection flags are changed to at least Read, the segment becomes accessible. If the segment protection flags are changed to Executable as well, then code can be executed directly from there.

This is exactly what Macarena does. When infecting a file, it pads the file size to a multiple of 4kb (a segment requirement, as noted above), then appends itself. The __PAGEZERO segment is altered to point to the virus code that starts immediately after the padding, and the segment protection flags are changed as described above, depending on the version of the virus. The change to the segment protection flags acts as the infection marker.

THREADS

The final piece of the puzzle involves how the virus gains control. The method is straightforward – the UnixThread load command contains the initial values for all of the CPU registers for the specified architecture. This includes the Instruction Pointer (EIP for the *Intel* architecture, and SRR0 for the *PowerPC* architecture). By altering the Instruction Pointer register to the required virtual address, the code at that location will be executed when the file is loaded. Macarena changes the Instruction Pointer register to zero, the start of the __PAGEZERO segment. This is apparently an unexpected value for some tools such as *GDB* and *IDA*, with the result that the virus code is not shown.

LIFE, THE UNIVERSE, AND EVERYTHING

Macarena is a simple virus. When executed, it enumerates the files in the current directory, and for any file of normal executable type, the virus will attempt to infect it, if it has not been infected already. This algorithm was obviously sufficiently simple for someone to learn enough *PowerPC* assembler to port and release a *PowerPC* version of it a week later. The *PowerPC* version is functionally identical to the *Intel* version, apart from infecting files for the 32-bit *PowerPC* architecture instead.

Universal files describe multiple architectures, allowing an executable to run on multiple platforms. They are not actually Mach-O files themselves. Rather, they are archives that contain multiple Mach-O files. Since this is the more common format for files on the *OSX* platform, it is likely that we will see viruses that understand the Universal file format and can infect the target architecture within them.

If that should happen, we might need to learn some new moves.

OSX/Macarena

Type:	Parasitic, direct-action Mach-O infector.
Size:	528 bytes (.A), 504 bytes (.B), 840 bytes (PPC).
Payload:	None.
Removal:	Delete infected files and restore them from backup.