

OSX/Leap.A - Under The Hood

A combined worm with companion virus behavior, technical details and write-up by:

Michael St. Neitzel, ESET spol. s.r.o.

Usually the worm arrives via file-transfer from OSX IM iChat-Clients, sent by infected buddy-list contacts.



When double-clicked on, the archive extracts automatically under OSX and reveals the worm executable, which looks like a JPG picture titled “latestpics”:

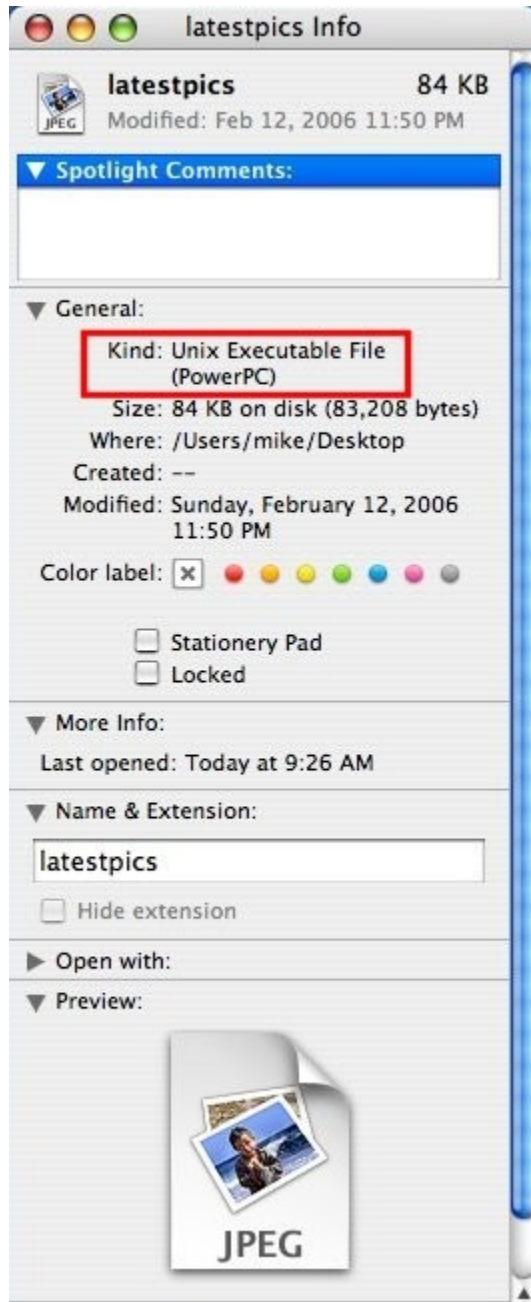
A look into the TAR-Archive reveals the tricky part of the worm:

```
Terminal — bash — 80x24
michael-st-neitzels-ibook-g4:~/OSX-Leap mike$ tar -tvf latestpics.tar
-r----- temp/wheel 43694 2006-02-12 23:50:01 ./._latestpics JPG-like Icon
-rwxr-xr-x root/temp 39596 2006-02-12 23:50:01 latestpics
michael-st-neitzels-ibook-g4:~/OSX-Leap mike$
```

The archive contains the file “**._latestpics**” which causes the worm to look like the JPG picture shown in the desktop screen-shot above. This is a well known trick called “**Social Engineering**” that tricks users into believing things which are not true.

But what is it really if it is not a picture?

In this case here it is not a JPG-Picture, but a native Unix (Darwin PPC) Executable, which was created with the GCC Compiler.



The OSX “Get Info” function proves this:

There are at least 3 things which experienced Mac Users should have noticed:

- 1st this is a Unix executable
- 2nd the filename without the .jpg extension
- 3rd there is no preview except for the icon

However, some of the clues depend on the system configuration, but the most important clue is the “**Unix Executable File**” which should ring a **big fat red bell** for every user.

Time to play around

After extracting the contents of the TAR archive, if we remove the 2nd file - the icon file - the worm reveals its true identity as follows:



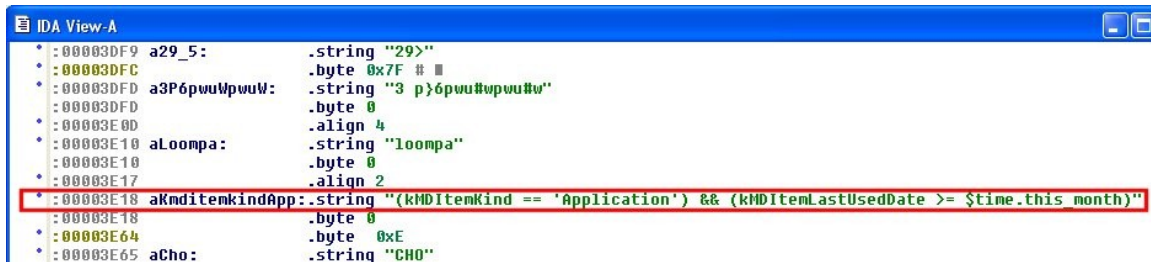
A native Unix Executable File - with a console mode icon.

The question is now:

Would you have clicked on such an icon without knowing what it is?

File Infection Functionality

The worm is able to “infect”, or from a technical view, to replace executable files in Application Frameworks. It does not infect all files, only Macintosh Applications used within the last month as the following IDA Pro Disassembler Screenshot proves:



```
IDA View-A
* :00003DF9 a29_5:      .string "29>"
* :00003DFC      .byte 0x7F # 127
* :00003DFD a3P6pwuWpwuW:  .string "3 p}6pwu#wpwu#w"
* :00003DFD      .byte 0
* :00003E0D      .align 4
* :00003E10 aLoompa:      .string "loompa"
* :00003E10      .byte 0
* :00003E17      .align 2
* :00003E18 aKmditemkindApp: .string "(kMDItemKind == 'Application') && (kMDItemLastUsedDate >= $time.this month)"
* :00003E18      .byte 0
* :00003E64      .byte 0xE
* :00003E65 aCho:        .string "CH0"
```

OSX/Leap.A scans folders to retrieve File Types and compares the results with “Application” and makes sure that they were used within the last month. The worm does this to create a higher possibility of getting activated (or even shared) in such frequently used files.

```
:00003120      addis   %rtoc, %r31, 0
:00003124      addis   %r3, %r31, 0
:00003128      lwz    %rtoc, 0x581C(%rtoc)
:0000312C      addi   %r3, %r3, 0xD10 # Check for "Application" and Date
:00003130      lwz    %r29, 0(%rtoc)
:00003134      bl     __CFStringMakeConstantString_stub
:00003138      mr     %r4, %r3
:0000313C      li     %r5, 0
:00003140      li     %r6, 0
:00003144      mr     %r3, %r29
:00003148      bl     _MDQueryCreate_stub
```

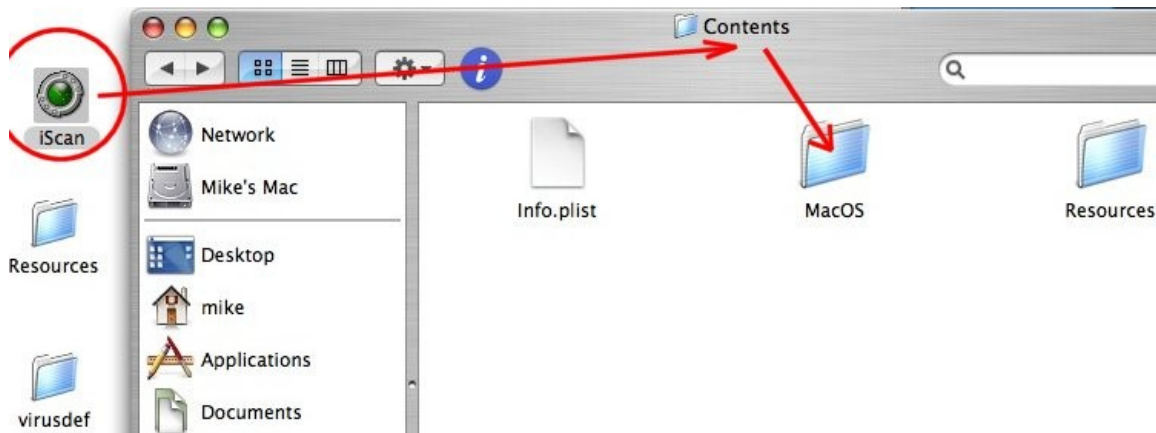
OSX/Leap.A also checks the root-permission flags (UID) for every Application in order to be able to infect it.

Note: There is a bug (not enough memory allocated) in the worm code which might result in corrupted, infected applications under special conditions.

Basically the worm allocates memory, copies the host-application (the uninfected binary) into the resource fork, marks the Application by changing the extended attribute to prevent multiple infections of already infected files. This is a very commonly used technique for file-infector viruses. In the windows world most so called “infection-markers” take place in the file-headers. The worm does not try to infect Applications with this Marker again. This is also the reason why some call this worm “OSX/Oompa”, because that is what the worm is looking for/replacing.

The worm then places a copy of itself at the position of the original binary so that the worm will be executed first and then pass control to the stripped-off original binary. This is typical “Companion Virus” behavior.

Other than, for example, windows executables Apple OSX Application-Binaries taking place in so called Frameworks is similar to a Stream-Type Environment with Folders and Subfolders.



The main-executable is, for instance, located within “/Contents/MacOS/” in the Application Framework - basically 2 folders “deep”. That is where the worm places the copy of itself after backing up the original executable from that location.

Upon execution the worm creates the following files:

- /tmp/latestpics**
- /tmp/latestpics.tgz**
- /tmp/latestpics.tar.gz**
- /tmp/hook**
- /tmp/apphook**
- /tmp/pic.gz**
- /tmp/apphook.tar**
- /tmp/pic**

Then OSX/Leap.A tries to delete “apphook” in the “/Library/InputManagers” folder and tries to place a copy of “apphook” from the extracted “/tmp/apphook” there. If this does not succeed (if the current user does not have root privileges) then it tries the same action in “~/Library/InputManagers”. That means if any OSX Application is loaded, the hook becomes active. It uses tar to decompress the included apphook.tar.

Leap.A is able to spread via iChat, (but due to a bug it might display a wrong filesize)



by sending a hidden file-transfer of latestpics.tar to iChat contacts.

The encrypted parts of the worm

Leap.A contains several poorly encrypted strings. Each one is encrypted with different XOR-Keys. The keys are hard-coded and therefore static for every string.

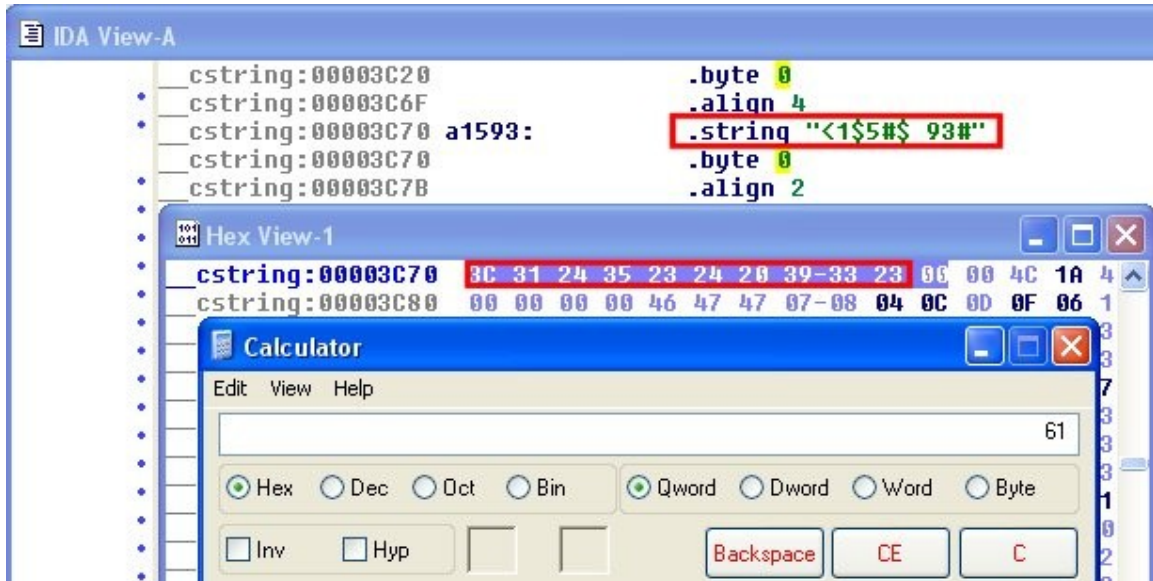
This makes it very easy to decrypt the strings even by hand without needing to debug the application:



The worm uses this encryption for trivial filenames and locations to hide its copy action.

It loads register r4 (on a PPC-CPU a general purpose register, such as Intel's EAX) with the XOR decryption key. In this case, 0x50, initializes register r3 with the string data and decrypts it.

The string "<1\$5#\$ 93#" becomes after decryption for example "latestpics":



So basically we point IDA to our string, taking a look into the Hex View and recognizing the binary byte sequence of "3C | 31 | 24 | 35 | 23 | 24 | 20 | 39 | 33 | 23". This data XOR'ed with 0x50, the static key for this string, reveals the "latestpics" string.