MALWARE ANALYSIS 2

MULTIPLATFORM MADNESS!

Peter Ferrie
Microsoft, USA

A cross-infector of entirely unrelated platforms is typically implemented as two viruses stuck together, simply because it's the easiest way to do it. However, if the general mechanics of file enumeration and infection are the same across the affected platforms, then a virus can implement an abstraction layer and expose APIs that each of the routines can call to perform the essential functions of find/open/map/unmap/close. This is exactly what {W32/Linux/OSX}/Clapzok does.

The virus begins by calculating the CRC32 of itself. It uses a reverse polynomial (the usual '0xEDB88320') to calculate the hash value. The resulting value is used as the seed for the random number generator in the virus. The virus also relocates the pointers to the abstraction routines, according to the load address of the virus code.

WINDOWS MODE

When running on *Windows*, the virus begins by resolving the addresses of the APIs that it needs. It finds the base address of kernel32.dll by searching backwards in memory, beginning with the current thread return address. The virus needs the addresses to perform the functions: find, open, seek, close, map, unmap, set file attributes, alloc and free, as well as some directory handling. The virus also resolves the addresses of the LoadLibrary() and GetProcAddress() APIs. The virus uses hashes instead of names, and calculates the hash of *every* exported function in kernel32.dll, one at a time, while trying to match the few that are of interest. Once every exported function has been hashed, the virus checks if it has found all of the APIs that it needs, and exits if not.

The virus attempts to load sfc.dll. If it succeeds, it uses the GetProcAddress() API to fetch the address of the SfcIsFileProtected() API. The reason the virus uses the GetProcAddress() method instead of the hash method is because the API resolver in the virus code does not support import forwarding. The problem with import forwarding is that while the API name exists in the DLL, the corresponding API address does not. If a resolver is not aware of import forwarding, then it will retrieve the address of a string instead of the address of the code. In the case of the SfcIsFileProtected() API, the API is forwarded in *Windows XP* and later from sfc.dll to sfc_os.dll.

The virus allocates a buffer, which is just over 2KB long, to hold the results of the directory search. The virus searches the current directory for all entries. It ignores any file that is marked as offline, temporary, sparse, hidden, or system, as well as devices and directories. The virus also ignores any file that is smaller than 4KB, or that is 8MB or larger. For any files that remain (essentially, only regular and read-only files), the virus determines the full pathname, and checks if the file is protected by System File Protection (if available). If the file is not protected, the virus attempts to set the file attributes to a normal writable file. There is a minor bug here, in that if the file is subsequently deemed to be uninfectable because of its size, or because it cannot be opened, the file attributes are not restored.

The virus attempts to open the file in writable mode. If this is successful, it increases the file size by 8KB, and then runs the shared infection routine (see below).

LINUX MODE

When running on *Linux*, the virus begins by retrieving the effective user ID, which it uses later. The virus opens the current directory for reading, and requests one entry at a time. It ignores all but regular files. It also ignores any file that is smaller than 4KB, or that is 8MB or larger. If the file belongs to the current user, then the virus sets the file attributes to readable and writable.

The virus attempts to open the file in writable mode. If this is successful, the virus increases the file size by up to 12KB, rounded down to the nearest multiple of 4KB, and then runs the shared infection routine (see below).

OS X MODE

When running on *OS X*, the virus begins by retrieving the effective user ID, which it uses later. It allocates a block of memory 32KB in size with read/write permissions. The virus opens the current directory for reading, and requests as many directory entries as will fit into the block of memory. The virus is only interested in regular files. After examining each file in the buffer, it will request more directory entries, and then examine those. This action will repeat until there are no more entries to be found.

The virus has strange code in several places, where a series of short branches are chained together for no obvious reason. This may have been part of some debugging code.

For each file that is found, the virus will query its attributes. It checks for regular files (again), and ignores any file that is marked as immutable, append-only, hidden, or opaque (though this flag relates only to directories). The virus also ignores any file that is smaller than 4KB, or that is 8MB or

larger. If the file belongs to the current user, then the virus sets the file attributes to readable and writable.

The virus attempts to open the file in writable mode. If this is successful, the virus increases the file size by up to 12KB, rounded down to the nearest multiple of 4KB, and then runs the shared infection routine (see below).

INFECTION STAGE

The virus maps a view of the entire file, and then checks for the signatures of the file formats of interest. It is very trusting of the file contents, to the extent that it uses no exception handling at all, on any of the affected platforms. As a result, any corrupted or crafted files will cause the virus to crash.

WINDOWS INFECTION

For *Windows* files, the virus checks for the 'MZ' signature, the lfanew field being less than 4KB, the PE signature, and the infection marker. The infection marker is that the low word of the time stamp field is 0x7dfb. It is not known why this value was chosen. The virus checks the PE flags field for a 32-bit executable, which is not a system or DLL file, and is not targeting a uni-processor environment. The virus requires that the subsystem is GUI or CUI, that the security table data directory size is zero, and that the file is not a WDM driver. Interestingly, these are exactly the same set of checks (in a slightly different order) as used by the Chiton [1] family. This might be significant (see below). More interestingly, some of these checks are useless. Apart from the 'IMAGE_FILE_EXECUTABLE_IMAGE' and 'IMAGE_FILE_DLL' flags in the Characteristics field, all of the other flags are ignored by *Windows*. This includes the flag ('IMAGE_FILE_32BIT_MACHINE') that specifies that the file is for 32-bit systems.

If the file passes the filtering process, then the virus marks it as infected. This has the effect that the file won't be examined again, even if infection fails. This behaviour is different from ELF infection (see below). The virus checks that the file has no more than 512 bytes of appended data – the infection will be abandoned at this point if the appended data exceeds this amount.

The algorithm for determining the size of the appended data finds the last section and sums the physical offset and its size. This works in most cases, but is incorrect. The last section might be entirely virtual, in which case the physical offset and size will be zero, and a previous section must be examined to determine where the file ends. There is more to the algorithm than simply this check, but the details are beyond the scope of this article.

The virus sets the host entry point to point to the end of the last section, and then appends the virus code to the last section. The virus adjusts the active abstraction layer table to use the *Windows* APIs, and then chooses a random number for the multiplier in the host entry point equation (see below). The virus chooses three more random numbers, and encrypts three tables of bytes with the respective values. The tables are texts that use different keys, so occasionally at least one of them will be decoded and visible in a replicant. The first text contains the credits for the virus, the second contains greetings to various people, and the third text reads: '2874 bytes of (obsolete) MultiPlatform Madness!'.

The virus increases the virtual size of the last section, if needed, and marks the section as executable and readable. It updates the size of the image, if needed, but does not update the file's checksum.

LINUX INFECTION

For *Linux* files, the virus checks for the 'ELF' signature, a 32-bit LSB byte order with the proper format version, and the infection marker in the padding area. The infection marker is that the word at file offset 0x0b is 0x7dfb. The virus checks for an executable file (as opposed to an object file), a specific size for the EHDR structure, a specific size for a Program Header Table entry, a specific size for a Section Header Table entry, and that the Program Header Table is at a fixed offset. The virus checks that the file is targeting either an *Intel* 80386-based CPU or a reserved value which was intended to indicate '*Intel* 80486' but which has never been used. This last check is similar to *Windows* viruses comparing the Machine field to 0x14d (IMAGE_FILE_MACHINE_I386 + 1), and is equally meaningless. The virus ignores files that contain more than 30 Program Header Table entries.

If the file passes the filtering process, then the virus marks it as infected. Unlike the *Windows* infection method, the file will always be infected when this routine completes. It is not known which behaviour was intended to be the 'correct' one, but it seems more likely that it was this one, and that the *Windows* version is the anomaly.

The virus assumes that the Section Header Table exists (it is optional), and adds 4KB to the Section Header Table offset. It then fetches the Program Header Table offset again, seemingly having forgotten that it knows the offset already. The virus examines each entry in the Program Header Table, and assumes that at least one Program Header Table entry exists (for well-formed files, it is required). The virus watches specifically for the entry that points to the Program Header Table itself, and also the

entry that points to the loadable segment that holds the EHDR structure. All other entries have their file offset increased by 4KB.

For the loadable segment that holds the EHDR structure, the virus increases the file and memory sizes by 4KB, and marks the segment as executable and readable. The virus relies on finding this segment, but for some versions of *Linux*, it does not need to exist.

For the two entries of interest, the virus reduces the virtual and physical addresses by 4KB, to prevent any alteration to the rest of the loaded image. It shifts the entire file down in memory by 4KB, and then examines each entry in the Section Header Table. The virus assumes that at least one Section Header Table entry exists. The virus updates the file offset for each entry.

After the entries have been updated, the virus fetches the virtual address of the EHDR from the loadable segment that holds it. The virus sets the host entry point to point to the end of the Program Header Table, and then copies the virus code to the end of the Program Header Table. The virus adjusts the active abstraction layer table to use the *Linux* APIs, and then chooses a random number for the multiplier in the host entry point equation (see below). The virus chooses three more random numbers, and encrypts the three tables of bytes with the respective values, as above.

OS X INFECTION (1)

For *OS X* files, the virus checks for the presence of the 'MACH-O' signature, that the file is targeting an *Intel* 80386 or better CPU, that the file is executable, and that there is at least one loader command. The virus examines each entry in the command table, and watches specifically for the entry that describes a segment, and the entry that describes a thread. There is a minor bug here, which is that the virus checks only the low byte of the command value, so it could potentially be fooled by entirely unrelated (but as yet undefined) commands.

If the segment descriptor command is seen, and if the virus has not already seen the __PAGEZERO segment, then the virus checks if the virtual address and file size are zero for the segment. If they are, then the virus remembers that it has now seen the __PAGEZERO segment.

If the thread descriptor command is seen, and if the virus has not yet seen the required thread state, then the virus checks the 'flavour' of the structure. If the flavour identifies the structure as being for the *Intel* 80386 format, then the virus remembers that it has seen the thread state.

Once all of the entries have been examined, and if both the segment descriptor and thread descriptor have been seen, then the virus rounds the file size up to the next multiple of 4KB. The virus sets the virtual size of the __PAGEZERO segment to 4KB (even though the field holds this value already), sets the file offset to point to the new end of the file, and sets the physical size to the size of the virus code. This serves as the infection marker, since the virus will skip any segment that has a non-zero file size, so it will never find the __PAGEZERO structure again.

The virus increases the file size by the size of virus, and marks the segment as executable and readable. The virus sets the host entry point to zero (that is, the start of the __PAGEZERO data). This infection method is identical to the one used by the Macarena [2] virus (and this one appears to be only the second virus ever to use the method). Even the logic is almost identical. The fact that the credits for this virus contain the name of someone other than the author of Macarena suggests that the author of this virus relied very heavily on the source code of the Macarena virus.

The virus appends itself to the file, adjusts the active abstraction layer table to use the *OS X* APIs, and then chooses a random number for the multiplier in the host entry point equation (see below). The virus chooses three more random numbers, and encrypts the three tables of bytes with the respective values, as above.

OS X INFECTION (2)

The virus also checks for the MACH-O 'universal binary' (an archive format that contains at least one MACH-O file) signature. If it is found, then the virus requires that there is at least one architecture (at least one MACH-O file) in the archive. The virus checks that the last MACH-O file in the archive is targeting an *Intel* 80386 or better CPU, that the file has no appended data, and that the last file is a MACH-O file. If so, then the infection proceeds as for the MACH-O method described above. If the file is infected successfully, then the virus updates the size of the MACH-O file in the archive header.

FILE CLOSE

When the infection process exits – regardless of the cause – the virus unmaps the view of the file, restores the file size if the infection was abandoned, closes the file, and restores the file times.

WINDOWS MODE

After the infection stage has completed in *Windows* mode, the virus restores the file attributes. The virus continues

its search of all files in the current directory. After the search has completed, the virus switches to the 'Windows' directory, and attempts to infect all files in that directory. The virus does the same for the 'System' directory. This is a very old-school idea, since newer operating systems do not allow the arbitrary writing of files in either of these directories, and most of the files will be protected by the System File Protection in any case.

LINUX AND OS X MODE

After the infection stage has completed in *Linux* and *OS X* modes, the virus checks if the original file had any file attributes set. If none were set, then it does not restore any. This could be considered a bug since the file is now accessible, where it was not before.

The virus continues its search of all files in the current directory. After the search has completed, and if the current user is the root user, then the virus attempts to change to the '/bin' directory and infect all files in that directory. The virus does the same for the '/usr/bin' directory.

CLEAN-UP

After all files have been examined in any of the modes, the virus restores the current directory and prepares to transfer control to the host. The host entry point is not stored as a plain value. Instead, the virus solves an equation to recover the value. The virus carries the multiplier and the answer, and intends to determine the multiplicand. The equation is solved in a brute-force manner. Once the value has been recovered, the virus jumps to the original entry point.

CONCLUSION

The abstraction method for cross-platform infection is a very powerful technique to simplify the virus code and reduce its size. It seems likely that we will see other viruses using the same technique in the future, if only to further improve on the idea.

REFERENCES

[1] Ferrie, P. Unexpected Resutls [sic]. Virus Bulletin, June 2002, p.4. http://www.virusbtn.com/pdf/magazine/2002/200206.pdf.

[2] Ferrie, P. Do the Macarena. Virus Bulletin, January 2007, p.4. http://www.virusbtn.com/pdf/magazine/2007/200701.pdf.