MALWARE ANALYSIS

CAN YOU SPARE A SEG?

Peter Ferrie
Microsoft, USA

Peter Ferrie resumes his series of analyses of viruses contained in the EOF-rRlf-DoomRiderz virus zine (see also VB, September 2008, p.4, VB, October 2008, p.4 and VB, November 2008, p.4).

NON-OPTIMIZATION TRICKS

We begin with a virus that was named 'H2T3' by its author. This virus infects files on the *FreeBSD* platform. Interestingly, the virus is split into two parts. The first part is written in assembly language, and exists solely to pass some important constants to the second part, which is written in C.

The assembly language part is not optimized at all. For example, a MOV and an ADD could be replaced by an LEA; some arithmetic involving two constants could be achieved with one combined constant, etc. Even the calling convention that was used in the first part results in an extra instruction to balance the stack, but this is perhaps an indication of the quality of work by virus writers these days. It is unclear even why the first part exists, since the constants could be calculated just as easily in C.

Seek and ye shall find

The virus begins by searching for regular files within the current directory. For each file that is found, the virus attempts to retrieve the file attributes and change them to writable. The file is skipped if either of these operations fails. If both operations succeed, then the virus attempts to open and map the file. If the open fails, then the virus restores the file attributes and returns. If the mapping fails, then the virus attempts to unmap an invalid region – fortunately for the virus writer, this invalid unmapping does not cause an error.

However, the virus is extremely trusting of the contents of the file. It assumes that the file is in ELF format before verifying this fact. The assumption goes so far that a field inside the supposed ELF header is used by the virus, without checking that the file is large enough to support the presence of that field. A sufficiently small file will cause the code to crash. In fact, a truncated ELF file, or a file with a sufficiently large value in the e_phnum field, among other things, will cause the virus to demonstrate the same effect, since the code contains no bounds checking of any kind.

Of course, these are minor quibbles.

Image-conscious code

The virus is interested in ELF files that are executable, not infected already, and whose ABI specifies a *FreeBSD* file. The virus does not check the target CPU for the file, perhaps assuming that any file on the current system is designed to run on that system. The virus then searches within the Program Header Table entries for all loadable segment entries, and keeps track of the one with the lowest virtual address. This value is used as the ending address for the virus code in the file to infect. What the virus intends to find is the entry with the physical address of zero, which is the file header, and which corresponds to the image base address. The virus is simply performing the search in a different way.

Headers and footers

The virus also searches within the Program Header Table entries for a PT_PHDR (Program Header Table segment) entry. If one is found, then the virus replaces it with a loadable segment entry. This loadable segment will contain the virus code. The segment is set to the size of the virus, and its starting location is calculated to end just before the loadable segment with the lowest virtual address that was located earlier. The host's original entrypoint is saved in the virus code, and a new entrypoint is set to the location of the virus code in memory. The virus sets the last byte of the e_ident field to 1, as an infection marker. This has the effect of inoculating the file against a number of other viruses, since a marker in this location is quite common. Finally, the virus appends its code to the file.

The 'problem' with adding a new loadable segment to a file is that it can be seen easily in a memory map. Anyone who is familiar with the file in question will know that it has been changed.

Trimming the fat

In ordinary circumstances, the Program Header Table segment entry is redundant, since a field exists in the ELF header that points directly to it. The only missing information in the ELF header is the size of the program header table. However, this value can be calculated by using other fields from the ELF header. This is the reason why the virus uses that entry.

After all files in the directory have been examined, the virus returns control to the host.

CAVEAT EMPTOR

Along similar lines is a virus from a different author. This one was named 'Caveat' by its author and was written

entirely in C – demonstrating that it can be done, though it does inject some assembly language code into the file to perform some essential operations. The virus infects files on the *Linux* platform. Despite the different authors, this virus shares many characteristics with the 'H2T3' code.

Misplaced trust

The virus begins by searching for files within the current directory. For each file that is found, the virus attempts to open and map the file. Unlike 'H2T3', if the mapping fails, this virus closes the file without attempting to unmap anything. However, this virus is equally trusting of the contents of the file. Like 'H2T3', this virus assumes that the file is in ELF format before verifying this fact. A field inside the supposed ELF header is used, without checking that the file is large enough to support the presence of that field. A sufficiently small file will cause the code to crash. A truncated ELF file, or a file with a sufficiently large value in the e_phnum field, among other things, will also cause the virus to crash, since the code contains no bounds checking of any kind.

Missing the mark

The virus is interested in ELF files which are executable, for the *Intel* x86-based CPU, and whose ABI is not specified. The virus does not check for an infection marker, because the marker is actually the absence of something instead of the presence of something. This will be explained below.

If a file is found to be infectable, then the virus rounds up the file size to a multiple of 4KB, and saves the host's original entrypoint. The rounding is required to ensure that the virus body will be completely mapped into memory later. Again, this will be explained below.

Note to self

There are two variants of the virus. Both search within the Program Header Table entries for the loadable segment that corresponds to the image base address. They also search for a PT_NOTE entry. However, the first variant ignores any PT_NOTE entry that appears before the image base address entry in the Program Header Table. This might be considered an optimization to avoid parsing the entries twice (since the entrypoint calculation requires the image base address), but some files will not be infected as a result. It could also be considered a bug, since the entrypoint calculation could be delayed until after the parsing has completed.

Force of h-ABI-t

In the case of the first variant, if an acceptable PT_NOTE entry is found, then the virus shrinks the Program Header Table by the size of one entry, to make space for the first part of the virus loader. With the PT_NOTE entry removed, the corresponding .note.ABI-tag section is unreferenced and available to be replaced. The virus overwrites the .note.ABI-tag section with the second part of the virus loader, and changes the host entrypoint to point to the first part. Since there is usually only one PT_NOTE entry in a file, its removal means that it cannot be found again. Files that do not contain a PT_NOTE entry will not be infected. This is how the infection marker works.

Stacking the deck

In the case of the second variant, the virus also searches for PT_PHDR and PT_GNU_STACK entries. The virus shrinks the Program Header Table by the size of these entries to make space for the entire virus loader. The virus changes the host entrypoint to point to the loader. With the removal of those entries, any subsequent examination of the file will not find sufficient space for the loader. As a result, such files will not be reinfected. This is how the infection marker works.

The easy way or the hard way

After the loader has been copied to the file, the virus extends the file to the multiple of 4KB that it calculated earlier, then appends the virus code. The loader works by calling the mmap() function to map into memory the virus code from the end of the file. Since the mapping requires an aligned base as a starting address, the virus must either place itself at exactly such an aligned address (the simplest case, as we see here), or the size of the mapping must be increased appropriately to potentially span two pages, and the virus code must be aware of the possibly non-zero offset within the first page where the virus body resides (which does not increase the file size to the same degree, but which increases the complexity of the algorithm and requires more code).

This method of memory-mapping the virus code avoids the loadable segment problem described above. Of course, the mapped memory might be still considered to be suspicious. The virus author described a workaround for this by allocating a new memory region and copying the virus body there before unmapping the old copy.

CONCLUSION

At first glance, the technique of replacing the .note.ABI-tag section in ELF files might appear to be similar to the .reloc overwriting technique in *Windows* PE files. However, there are far more differences than similarities, since ELF files have fewer restrictions regarding section placement, among other things. In a sense, this kind of cavity infection could be considered just another 'hole' that is being exploited.