MALWARE ANALYSIS 1

FRANKIE SAY RELAX

Peter Ferrie
Microsoft, USA

When a virus infects a file, it usually needs to know its loading address so that it can access its variables. This is done most commonly by using a 'delta offset'. There are two main types of delta offset: one is the difference between the location where the virus is currently loaded and the original location where the virus was loaded when it was created; the other is the difference between the location of the variable and the start of the virus code. One alternative method is to append relocation items to the host relocation table (if one exists), so that the addresses in the virus code are updated appropriately by the operating system itself. However, touching the host relocation table can be a complex task, depending on the file format and its location within the file. Another alternative is to carry a relocation table in the virus body and use that to update the addresses to constant values during the infection phase. This is the method that is used by Linux/Relax.A. Linux/Relax.B uses the same method, but in this case the relocation table is generated dynamically.

ALL YOUR BASE

Both viruses begin by registering two signal handlers: one to intercept invalid regular memory accesses (for example, a pointer to unallocated memory), and the other to intercept misaligned addresses or invalid mapped-memory accesses (for example, in a file that is memory-mapped according to its original file size, but which is then truncated by another process). The viruses use the 'int 0x80' interface directly here, because no external symbols have yet been resolved (that is, the host has access to its own symbols, but the virus does not know where they are yet). These two int 0x80 calls are the only ones in the virus code. However, what might be considered a bug exists here – if an exception occurs, then the signal handlers are not restored to their default values. Thus, if an exception occurs in the host (perhaps due to the presence of the virus) and in the absence of another registered signal handler, the signal handlers will run the host entrypoint again – at which point further exceptions seem likely to occur, so the signal handlers will run again (and again, and again). The result is an infinite loop.

At this point, Relax.A finds the image base of its host (Relax.B does this the first time that a function is called in libc) by walking backwards one page at a time, beginning at the start of its code, until 'an' ELF header is found. The 'an' here refers to the fact that that no verification is made that the signature belongs to an actual header, as opposed to the (unlikely) case that the magic value happens to appear at the start of a page. However, the signal handler will intercept any problem relating to fake headers. This lack of verification could exclude certain files from being infected, but this is a minor point. It would be possible to inoculate files against these and similar viruses by placing the fake signature in the right place, but the idea is a little silly. The simplest approach would be to remove the writable flag on the file, since the viruses make no attempt to set it.

GET IT. 'GOT' IT? GOOD.

Once the header is found, the viruses search within the Program Header Table for the segment that contains the virus code. The virus code segment is identified by finding the loadable segment which has the lowest virtual address. The viruses also search for the segment that holds the dynamic linking information. The viruses search within the tags in the dynamic segment for the one that describes the Global Offset Table. If the third entry in the Global Offset Table is non-zero then the viruses use that pointer to search for the segment that holds the dynamic linking information, and then search the tags within that segment for the one that describes the Global Offset Table. The Global Offset Table is a table of pointers. The third value in the table is a pointer to the '_resolve' symbol inside the dynamic linker. If the dynamic linker is not required (because the symbols have all been resolved statically before the process started) then the value at that location will be zero.

In either case, the viruses perform the same search for the dynamic segment and another Global Offset Table, using the fifth entry in the current Global Offset Table. The new table should point into libc. There is no requirement for it to do so, but there is no other library that the loader would need. The viruses search within the tags in the dynamic segment for the symbol table and the string table. In order to call external functions, the viruses need to resolve the external symbols. To do so, they would normally need to know how many symbols exist. They attempt to retrieve the number of symbols from a hash table which is located by searching the tags within the dynamic segment. The viruses know about two hash table tags. If neither of these is found, then they use a hack to calculate it by determining the number of symbol structures that can fit in the symbol table.

It is not known why the viruses determine the number of symbols, except perhaps as a leftover from code that used one of the hash tables correctly (see *VB*, August 2009 p.4 for details of how the hash table is used for symbol resolution). They could perform the symbol search without an upper limit (the symbols that the viruses need ought

to exist), and simply allow the signal handler to trap any error. Since the virus is using a brute-force search anyway, the performance is actually worse with the check for the upper limit than it would be without it. The virus author knows how to use the hash table correctly, but since the viruses recognize two types of hash table, which have different formats, there would need to be two parsing algorithms.

Relax.A uses the gathered information to resolve the address of a single function, mprotect(), while Relax.B uses it for multiple functions. Further, Relax.B waits until a function in libc is called for the first time, and then resolves the address of that function. Thereafter, the proper address is used directly.

PROTECTION DETAIL

Relax.A uses the mprotect() function to make the code section writable. Then the virus parses the relocation table that it carries in its data section, searching for the relocation items that correspond to external symbols. The virus resolves the addresses of the external symbols that it needs in order to infect files. The relocation table is in a custom format, and is produced by a standalone tool that is run after the file is compiled. The details of that tool are not relevant here. After applying all of the required relocations, the virus restores the section attributes, and then calls the main virus body.

Relax.B does not carry a relocation table in its data section. Instead, the virus disassembles its code at runtime and creates the relocation table dynamically. As a result, the mprotect() function is not needed by the virus. The virus has no concerns about the code versus data problem, since the entire virus body is known. Of course, if there were any misinterpretation, it would have prevented the first generation of the virus from running at all, and thus would have been detected instantly.

Since the viruses can run from any address thanks to the relocation table, they are also able to make use of external functions instead of calling the int 0x80 interface directly. In this case, the viruses use the ftw() function to search for files to infect instead of performing the file enumeration on their own. The ftw() function accepts a pointer to a function to be called for each item that is found. The infection routine begins by attempting to open the item and map the first 4KB of the file. The viruses are interested in ELF files that are at least 1KB long (this appears to be an oversight given the size of the map above), but not more than 3MB large. In contrast to all of the previous pieces of malware from this virus author, the viruses are quite strict about the file format:

- the ELF signature must match
- the size of the ELF header must be the standard value
- the file must be 32-bit format
- the file must use little-endian byte-ordering
- the file must be executable
- it must be for an *Intel* 386 or better CPU
- the version must be current
- the size of a program header table entry must be the standard value
- there must not be too many program header table entries
- the program header table must fit within the file
- the ABI must either not be specified or it must be for *Linux*
- the size of a section header table entry must be the standard value
- the section header table must fit within the file
- the file must not be infected already.

The infection marker for the viruses is the last byte of the e_ident field being set to 1. This has the effect of inoculating the file against a number of other viruses (including several by the same virus author), since a marker in this location is quite common.

HOLE-Y WORK

The viruses search the Program Header Table for the interpreter segment. The segment will be present if the file uses dynamic linking. If the segment is found, then the viruses check that it fits within the file, and that the virus code can fit in the space between the end of the interpreter segment and the start of the next page (though there is an off-by-one bug here such that an exact fit will not be accepted). There is an implicit assumption here that the interpreter segment is in the first page of the file. The viruses also search for the loadable segment which has the highest virtual address. If the interpreter segment is not found, then the viruses will try to place their code immediately after the Program Header Table, otherwise they will try to place their code immediately after the interpreter segment. There is an implicit assumption here that the Program Header Table appears before the interpreter segment. If the two elements are swapped, then the virus will overwrite the Program Header Table as a result.

The viruses initially increase the file size by 4KB and create a hole at the chosen location (into which the virus

code will be placed). The bytes between the end of the virus code and the start of the next page are zeroed. There is a bug here in that some bytes in the following page are also zeroed because the length is calculated incorrectly. The viruses add 4KB to the file offset of the Section Header Table, and to the file offset of each of the entries in the table, to compensate for the size of the hole that was inserted.

The viruses find the Program Header Table entry that corresponds to the file header, increase its physical and virtual size by 4KB, and decrease its physical and virtual addresses by 4KB. The physical and virtual addresses of the Program Header Table and the interpreter segment are also decreased by 4KB, to ensure that they remain within the first page of the file. All of the other Program Header Table entries have their physical address increased by 4KB.

The viruses increase the physical and virtual sizes of the loadable segment with the highest virtual address by the size of the virus data. They create a hole at the chosen location, into which the virus data will be placed. The viruses then increase by a corresponding amount the file offset of each entry in the Section Header Table whose previous offset was after the end of the affected loadable segment.

The viruses parse their relocation table again, and for each entry that is not an external symbol in the Relax.A code, or for each entry in Relax.B (Relax.B does not carry relocation information for the external symbols), the viruses apply the appropriate relocation value in the newly infected file, such that all of the addresses are made absolute according to the host loading address. Of course, this requires that the address is constant. It will not work if the file is a position-independent executable. To achieve that would require the use of a delta offset in order to locate the data section in the first place, and then to apply the relocations dynamically to the entire virus body.

Finally, the viruses set the entrypoint to point to the virus code, mark the file as infected, and then allow the search to continue for more files.

CONCLUSION

The idea of a virus carrying (or calculating) a relocation table is great for virus writers. It allows them to write the code in a high-level language, and use all of the high-level APIs that exist, without having to perform tricks with position dependence or having to use Assembler to fiddle with the bits. Best of all, it doesn't make any difference to anti-virus vendors, because whether it's high level or low level, we can still detect it without any trouble.