# VIRUS ANALYSIS 1

## CRIMEA RIVER

*Peter Ferrie*
Symantec, USA

In 2001 we received a virus for *Windows* that integrated its code with the host code, making it very hard to find. That virus was Zmist (see *VB*, March 2001, p.6). In 2007, we received a virus that might be considered 'Zmist for *Linux*'. That virus was Crimea.

### THE BROTHERS KARAMAZOV

The Crimea virus family contains four variants. The first (version 0.5) was a very early release and gained control via an entry in the .ctors section. This method had been described previously by a virus writer known as izik, and is in some ways the *Linux* equivalent of the *Windows* Thread Local Storage entry point method (see *VB*, June 2002, p.4). The other three Crimea variants (0.23, 0.24, and 0.25.2) are very closely related and are essentially the 'finished product'. These variants will be described in this article.

The 0.23 variant replicates before running the host. This causes a noticeable delay, since the virus runs slowly. In the 0.24 variant, however, the virus starts by running the host code as a separate process, then sets itself to the lowest scheduler priority before replicating. This reduces the CPU usage significantly. However, the change causes another noticeable effect – the child process will not terminate until the parent does, because the child process expects the parent process to be interested in the exit code. This bug was fixed in the 0.25 variant by sending a signal to tell the kernel prior to running the host that the child can terminate on exit.

In all cases, the virus continues by decrypting its data then beginning the search for files to infect.

### SEEK AND YE SHALL FIND

The search routine enumerates all entries in the current directory, skipping any that begin with '.'. This allows the virus to skip the '.' and '..' directories, but it also means that it skips any files that begin with '.' (though there are not usually many on a typical system). The virus also skips symbolic links.

For each entry that it considers to be valid, the virus calls chdir(). If the chdir() call succeeds, then the entry must correspond to a directory, and the virus repeats the search in that directory and in any subdirectories that are found. Otherwise, the virus assumes that the entry corresponds to a file. If the file has the executable attribute set, the virus will attempt to infect it.

### COARSE FILTERING

The virus applies a number of filters to remove unsuitable files. The conditions of these filters include that the file size is at least 16kb, and not more than 512kb. The file must begin with an ELF header, it must be a shared object for the 32-bit *Intel* 80386 or better CPU, the ELF version must be current, and the OS/ABI version must not be specified. The ninth byte of the padding field must also be zero – a non-zero value is the infection marker for the virus. The final filter checks that each program header describes a valid section.

The virus can only infect position-independent files. The reason for this is that position-dependent files can contain values that are indistinguishable as addresses or constants, since there is no relocation information. This would force the virus to guess – and an incorrect guess would corrupt the host and ruin any chance for the virus to survive. In position-independent files, there is enough context to know what the values represent.

### FINE FILTERING

The virus examines the section headers of the files that pass the first level of filtering to look for required items. The virus requires sections with the names '.plt', '.got', '.got.plt', '.rel.plt', '.rel.dyn', '.data', and '.init'. The virus also requires sections of type SHT_DYNSYM and SHT_DYNAMIC, but forgets to check if SHT_DYNAMIC has been found. A missing SHT_DYNAMIC type will cause the virus to crash later and corrupt the host.

Another problem is that the virus uses an AND-mask to check that all items have been found. This is unreliable for certain values of the map address if the section table crosses a page, because the AND will zero out all bits and look as if no pointer was found. However, the file would simply not be infected in that case.

The virus loads all of the data for sections that have file content (that is, ignoring purely virtual sections). For sections that contain executable code but are not the '.plt' section, the virus disassembles the code into a special buffer.

### DISASSEMBLY (HOST)

The virus disassembles the host code instruction by instruction, without regard to the code flow. This is problematic for files that contain embedded data, since the data could appear to be a set of valid instructions, but the interpretation of these could cause the real instructions that follow to be misinterpreted.

While disassembling the code, the virus constructs an ordered list of the instructions. This list will be used later to integrate the virus code with the host code. The virus contains a special check for alignment sequences, since their presence indicates a routine whose alignment must be preserved after any movement.

The disassembly is done by a library called XDE, which was written by the author of Zmist. The XDE library is fairly primitive in a sense – the instruction set that it carries is approximately equal to that of an early *Intel Pentium* CPU. There is no support for *Intel MMX* or *SSE*-style technologies, or even quite common instructions such as CMOV. The XDE library contains a bug in that there is no limit to the length of an instruction. Even though duplicated prefixes will cause a BAD flag to be set, the virus never checks for it. The XDE library also contains another bug, this time in the SIB handling, where certain encodings cause the wrong register to be chosen.

## CODE MARKING (HOST)

The virus then parses the host code, beginning with the entry point and following all calls, jumps and branches. Each instruction that is encountered is marked as 'active'. The calls and branches are followed recursively to allow continuation on return from a call, or if a branch is not taken.

The virus keeps up to 15 of the most recent instructions in a special buffer. This buffer is used to deal with jump tables. The problem with jump tables is that they are not single instructions, but collections of them. By keeping the recent instructions in a buffer, when an instruction is seen that corresponds to the last one in a jump table sequence, the buffer can be queried to see if the rest of the code matches the entire jump table sequence. The 0.25 variant improves on the jump table recognition by tracing the register context, since jump tables can have multiple forms.

Two bugs exist in the handling of jump tables in the 0.23 and 0.24 variants. The entries in a jump table are stored in a buffer for later relocation. In the buggy variants the buffer has a fixed size and is shared among all jump tables. One bug is that the entries are added to the buffer without any bounds checking. Thus, if there are more than 1,023 entries, memory corruption will occur. The other bug is an off-by-one calculation which means that the last entry in each jump table is not added to the buffer. Both bugs have been fixed in the 0.25 variant. The 0.25 variant (re)allocates the jump table buffer dynamically and adds all entries correctly.

The marking function looks specially for calls to imported functions, since they cannot be followed to their conclusion. The marking completes when a 'hlt' or 'ret' instruction is seen at the top level. Upon completion, any instruction that has not been marked as active can be discarded.

## DISASSEMBLY (VIRUS)

At this point, the virus disassembles itself, if it has not been done already. This disassembly differs from that of the host with respect to the code flow. The virus disassembles itself by following all calls, jumps and branches. The calls and branches are followed recursively.

## CODE MARKING (VIRUS)

The virus parses its own code in the same way as for the host, but in addition to marking the instructions as active, the instructions are marked as 'viral'. The reason for this is that after infection the host instructions are discarded from memory, leaving only the virus instructions. This speeds up the infection of other files in the same session, since the disassembly and marking are no longer necessary for the virus code.

## THE IMPORT BUSINESS

The virus searches the host import table for all imports that it requires. Any missing import is added to a list, and this leads to a potential bug. The 0.23 and 0.24 variants of the virus use only 24 imports; the 0.25 variant uses 25 imports. Most of these are likely to be imported already by the host. However, any future variants of the virus might make use of more obscure imports that the host will not import. The bug is that the list has a fixed size, and entries are added to the list without any bounds checking. Thus, if there are more than 32 entries, memory corruption will occur.

Once the import processing has been completed, the virus adds the appropriate relocation and stub entries for any newly added symbols and updates the hash tables to allow the symbols to be found. The section sizes are increased as required.

## MIX AND MATCH

The virus then reconstructs the code section, alternating a block of host code and a block of virus code. Each of the blocks ends with a 'jmp' or 'ret' instruction. Any routine that was aligned prior to infection will be realigned, if necessary. The instructions that were not marked as active are discarded now. Then, for each block of code in the code section, there is a 1-in-16 chance that the virus will exchange the position of that block with the position of the following block.

## INSTRUCTION ISOTOPES

In the 0.24 and 0.25 variants, the virus searches the code section for all two-byte instructions that use MODR/M format in register mode, and replaces some of them randomly with functionally equivalent alternatives.

There is a one-in-four chance of replacing 89/8b (mov reg2, reg1) with push reg1/pop reg2. There is a one-in-five chance of replacing 00-03/08-0b/10-13/18-1b/20-23/28-2b/30-33/38-3b/88-8b with an alternative encoding of the same instruction. There is a one-in-four chance of replacing 28-2b/30-33 (sub/xor) with the opposite instruction when both registers are the same. There is a one-in-five chance of replacing 08-09/84-85 (or/test) with the opposite instruction when both registers are the same. These replacements are identical in nature to those in Zmist. There is also a one-in-four chance of replacing 84-87 with an alternative encoding of the same instruction.

## STRETCH GOALS

The virus then extends the data section by the size of its data, plus a random amount of up to 127 bytes. The random amount of extra data is filled with random values. Next, the virus searches within the .init section for the last 'call' instruction, and appends an additional call which points to the virus code. This is how the virus gains control when an infected file is executed.

Now that the infection is complete, the virus builds a new ELF file, placing each of the sections at the appropriate location and aligning them as necessary. All references to individual sections are updated, too. It is here that the SHT_DYNAMIC section is referenced, with the assumption that it is valid. If all goes well, the code section is updated with adjusted label offsets and all branches are fixed and converted into long form (there are no short branches after infection). Then the jump tables and symbol tables are rebuilt, the new entry point value is assigned, and the virus data is encrypted.

Finally, the infection marker is set, and the file is closed. The virus then searches for the next file to infect, and the cycle repeats.

## CONCLUSION

The author of Crimea, who calls himself 'herm1t', chose the name 'Lacrimae' (Latin for 'tears') for this virus. The word is used most famously in *The Aeneid*. Aeneas is overcome by the futility of warfare and the waste of human life. If only herm1t would be overcome by the wasting of his own life in this way, we might not have to deal with viruses like this.