VIRUS ANALYSIS 1

Zmist Opportunities

Peter Ferrie & Péter Ször
SARC, USA

At VB2000 in Florida, *IBM's* Dave Chess and Steve White demonstrated their research findings on 'Undetectable Computer Viruses'. Early this year, the Russian virus writer Zombie released his 'Total Zombification' magazine complete with a set of articles and viruses of his own. Ominously, one of the articles in the magazine was titled 'Undetectable Virus Technology'.

Zombie has already demonstrated his set of polymorphic and metamorphic virus-writing skills. His viruses have been distributed for years in source format and other virus writers have modified them to create new variants. Certainly this will be the case again with Zombie's latest creation – W95/Zmist.

Many of us will not have seen a virus approaching this complexity for a few years. We could easily call Zmist one of the most complex binary viruses ever written. W95/SK, One_Half, ACG, and a few others come to mind in comparison. Zmist is a little bit of everything: it is an entry point obscuring virus that is metamorphic. Moreover, the virus randomly uses an additional polymorphic decryptor.

This virus supports a unique new technique: code integration. The Mistfall engine contained in it is capable of decompiling Portable Executable files to its smallest elements, requiring 32 MB of memory. Zmist will insert itself into the code: it moves code blocks out of the way, inserts itself, regenerates code and data references, including relocation information, and rebuilds the executable. This is something never seen before in previous viruses.

Zmist occasionally inserts jump instructions after every single instruction of the code section, each of which will point to the next instruction. Amazingly, these horribly modified applications will still run as before, just like the infected executables do, from generation to generation. In fact, we did not see a single crash during the test replications. Nobody expected this to work, not even Zombie. However, it is not foolproof – it takes some time for a human to find the virus in infected files. Due to its extreme camouflage Zmist is clearly the perfect anti-heuristics virus.

Initialisation

Zmist does not alter the entry point of the host. Instead it merges itself with the existing code, becoming part of the instruction flow. However, the random location of the code means that sometimes the virus will never receive control. If the virus does run, then it will immediately launch the host as a separate process, and hide the original process (if the RegisterServiceProcess () API is supported on the current platform) until the infection routine completes. Meanwhile, the virus will begin searching for files to infect.

Direct Action Infection

After launching the host process, Zmist will check if there are at least 16 MB of physical memory installed and that it is not running in console mode. If these checks pass, then it will allocate several memory blocks, including a 32 MB area for the Mistfall workspace, permutate the virus body, and begin a recursive search for Portable Executable .EXE files. This search will take place in the *Windows* directory and all subdirectories, the directories referred to by the PATH environment variable, then all fixed or remote drives from A to Z. This is a brute force approach to spreading.

Permutation

The permutation is fairly slow because it is done only once per infection of a machine. It consists of instruction replacement, such as the reversing of branch conditions, register moves replaced by push/pop sequences, alternative opcode encoding, xor/sub and or/test interchanging, and garbage instruction generation. The same engine, Real Permutating Engine (RPME), is used in several viruses including W95/Zperm, also written by Zombie.

Infection of Portable Executable Files

A file is considered infectable if it is smaller than 448 KB, if it begins with 'MZ' (*Windows* does not support the 'ZM' form), if it is not infected already (the infection marker is 'Z' at offset 0x1C in the MZ header – this field is not generally used by *Windows* applications), and if it is a Portable Executable file. The virus will read the entire file into memory, then choose from one of three possible infection types.

There is a one in ten chance that only jump instructions will be inserted between every existing instruction (if the instruction was not a jump already), and the file will not be infected. There is the same probability that the file will be infected by an unencrypted copy of the virus; otherwise, the file will be infected by a polymorphically encrypted copy.

The infection process is protected by Structured Exception Handling which prevents crashes in the case of errors. When the rebuilding of the executable is completed, the original file is deleted and the infected file is created in its place. However, if an error occurs during the file creation, then the original file is lost and nothing will replace it.

The polymorphic decryptor consists of 'islands' of code that are integrated into random locations throughout the host code section and linked together by jumps. The decryptor integration is performed in the same way as for the virus body integration – existing instructions are moved to either side, and a block of code is placed in between them. The polymorphic decryptor uses absolute references to the data section, but the Mistfall engine will update the relocation information for these references too.

An anti-heuristic trick is used for decrypting the virus code: instead of making the section writable in order to alter its code directly, the host is required to have, as one of the first three sections, a section containing writable, initialised data. The virtual size of this section is increased by 32 KB, large enough for the decrypted body and all the variables used during decryption. This allows the virus to decrypt code directly into the data section, and transfer control to there. If such a section cannot be found, then the virus will infect the file without using encryption.

The decryptor will receive control in one of four ways: via an absolute indirect call (0xFF 0x15), a relative call (0xE8), a relative jump (0xE9), or as part of the instruction flow itself. If one of the first three methods is used, the transfer of control will usually appear soon after the entry point. In the case of the last method, though, an island of the decryptor is simply inserted into the middle of a subroutine, somewhere in the code (including before the entry point).

All used registers are preserved before decryption and restored afterwards, so the original code will behave as before. Zombie calls this last method 'UEP', perhaps an acronym for Unknown Entry Point, because there is no direct pointer anywhere in the file to the decryptor.

When encryption is used, the code is encrypted with ADD/SUB/XOR with a random key, and this key is altered on each iteration by ADD/SUB/XOR with a second random key. In between the decryption instructions are various garbage instructions, using a random number of registers, and a random choice of loop instruction, all produced by the Executable Trash Generator engine (ETG), also written by Zombie. It is clear that randomness features very heavily in this virus.

Code Integration

The integration algorithm requires that the host has fixups, in order to distinguish between offsets and constants. However, after infection, the fixup data are not required by the virus. Therefore, though it is tempting to look for an approximately 20 KB long gap in the fixup area, which would suggest that the virus body is located there, it would be dangerous to rely on this during scanning.

If another application (such as one of an increasing number of viruses) were to remove the fixup data, then the infection will be hidden. The algorithm also requires that the name of each section in the host is one of the following: CODE, DATA, AUTO, BSS, TLS, .bss, .tls, .CRT, .INIT, .text, .data, .rsrc, .reloc, .idata, .rdata, .edata, .debug, DGROUP. These section names are produced by the most common compilers and assemblers in use, those of *Microsoft*, *Borland*, and *Watcom*. The names are not visible in the virus code, because the strings are encrypted.

A block of memory is allocated which is equivalent to the size of the host memory image, and each section is loaded into this array at the section's relative virtual address. The location is noted of every interesting virtual address (import and export functions, resources, fixup destinations, and the entry point), and then the instruction parsing begins. This is used in order to rebuild the executable.

When an instruction is inserted into the code, all following code and data references must be updated. Some of these references might be branch destinations, and in some cases the size of these branches will increase as a result of the modification. When this occurs, more code and data references must be updated, some of which might be branch destinations, and the cycle repeats.

Fortunately – at least from Zombie's point of view – this regression is not infinite, so that while a significant number of changes might be required, the number is limited. The instruction parsing consists of identifying the type and length of each instruction. Flags are used to describe the types, such as instruction is an absolute offset requiring a fixup entry, or instruction is a code reference, etc. There are cases where an instruction cannot be resolved in an unambiguous manner to either code or data. In that case, Zmist will not infect the file.

After the parsing stage is completed, the mutation engine is called, which inserts the jump instructions after every instruction, or generates a decryptor and inserts the islands into the file. Then the file is rebuilt, the relocation information is updated, the offsets are recalculated, and the file checksum is restored. If there are overlay data appended to the original file, then they are copied to the new file too.

Conclusion

A few years ago several anti-virus researchers claimed that algorithmic detection had no future. We would like to take this opportunity to turn that around, by claiming that virus scanners will have no future if they do not support algorithmic detection at the database level.

It is amazing to see how polymorphic viruses become more and more advanced over the years. Such metamorphic creations will come very close to the concept of a theoretically undetectable virus. The computing environment had to change and it did change. Now, modern viruses completely support this new environment. In the next couple of years we will be able to see how complex DOS viruses would be today if the environment had not changed during the last few years.

But for the time being, we are once again one step ahead of the virus writers. 'So, poly-encrypted permutated viral body is completely integrated with target file. Hmm … checkmate?' Not this time, Zombie.