MALWARE ANALYSIS 2

FLIBI: RELOADED

Peter Ferrie
Microsoft, USA

A new version of the W32/Flibi virus [1, 2] has been released. It now supports assemble-time or compile-time polymorphism during construction of the first generation translator code. Its parallels with molecular biology have increased with major changes to the replication process: horizontal gene transfer[1], codon[2] exchange, the introduction of start and stop codons[3], and optionally the addition of introns[4].

REMOVE BEFORE USE

This version of the virus lacks several of the commands that were present in the previous version. There are only 32 commands in this version. The commands that have been removed are as follows:

- _subsaved
- _zer0
- _add0004, _add0010, _add0040, _add0100, _add0400, _add1000, _add4000
- _JzDown
- _CallAPIMessageBox, _CallAPISleep

The removed commands have been replaced with equivalent sequences using the remaining instruction set, exactly as described in [2]. Both '_subsaved' and '_zer0' still appear in the source code, but they have been converted to macros. None of the other commands appear. The '_subsaved' macro uses the '_addsaved' command, after negating the value to add. The negate operation is achieved by performing an '_xor' with negative one, and then adding one. The '_zer0' macro uses a combination of the '_save' and '_xor' commands, which is equivalent to using xor on a register value with itself.

The '_addnnnn' commands have been replaced by a generic 'add' command. This command uses an instruction sequence

[1] Horizontal gene transfer is a process in which one organism incorporates genetic material from another without being the offspring of that organism. See http://en.wikipedia.org/w/index.php?title=Horizontal_gene_transfer&oldid=452313076.
[2] A codon is a trinucleotide sequence of DNA or RNA that corresponds to a specific amino acid. See http://www.genome.gov/Glossary/index.cfm?id=36.
[3] A stop codon is a nucleotide triplet within mRNA that signals a termination of translation. See http://en.wikipedia.org/w/index.php?title=Genetic_code&oldid=412677908#Start.2Fstop_codons.
[4] An intron is a nucleotide sequence within a gene that is removed by RNA splicing to generate the final mature RNA product of a gene. See http://en.wikipedia.org/w/index.php?title=Intron&oldid=456036842.

that combines the '_shl' and '_add0001' commands in an appropriate way to construct the required value. The '_JzDown' command has been replaced by a combination of '_JnzDown' commands, where one '_JnzDown' command branches to a '_call' command to reach the destination of the 'true' condition, and the other '_JnzDown' command branches over the '_call' command to reach the destination of the 'false' condition. The MessageBox API has been removed because the virus no longer has a payload, and the Sleep API has been removed because the virus uses a new hashing that does not produce the same false match.

ASSEMBLE-TIME POLY

Depending on the version that is used, the translator code polymorphism is either assemble-time or compile-time. The assemble-time polymorphism is achieved by using a routine that generates garbage instructions in the translator code. The assemble-time translator code garbage generator (ATCG) is composed of macros that are interpreted while the source code is being assembled into the first generation of the virus code. The randomness is achieved by seeding a random number generator with the current time, which the assembler allows. The ATCG is called eight times initially, and then once after each non-conditional instruction sequence, and after the single API call. In the case of a conditional instruction sequence (that is, a cmp instruction followed by a branch instruction), the sequence will not be separated. In the case of the API call, the parameters are not separated.

The ATCG chooses randomly if it will run, with a 50% chance of doing so. Once it is running, it emits one instruction sequence at a time, chosen randomly from a set of 15 instruction sequences. It then chooses randomly if it will continue to run, with about a 94% chance of doing so. The instruction sequences consist of operations that do not alter any registers, so they are harmless to the code. However, a number of them do alter the flags as a side effect of their operation, which is why they cannot be placed between a cmp and branch instruction sequence. There are some instructions that do not have any side effects, which could be used to break a cmp and branch instruction sequence, but selecting them from the list would increase the complexity of the routine for little gain.

COMPILE-TIME POLY

Compile-time translator code polymorphism is achieved by using a program to generate the assembler code that is then assembled into the first generation of the virus code. It still applies to the translator code, but it replaces the garbage generator in the assemble-time polymorphism described above. The compile-time translator code garbage generator (CTGG) can perform multiple operations on randomly selected registers, and the instruction set is larger. The CTGG knows which registers are currently in use and avoids generating operations on them. The CTGG can also be directed only to use instructions that do not alter the flags, which allows a conditional instruction sequence to be separated. The CTGG contains the set of six instructions that do not alter the flags (although, due to a bug, only five of them can be selected). A second set contains 11 instructions, six of which are the same as the set which does not alter the flags, and the other five instructions will alter the flags as a side effect of their operation.

The CTGG chooses randomly if it will run, with a 50% chance of doing so. Once it is running, it emits one instruction sequence at a time, chosen randomly from the appropriate set. It then chooses randomly if it will continue to run, with about a 94% chance of doing so. The CTGG is called in a loop initially, with only a 10% chance that the loop will exit on any iteration. Thereafter, the CTGG is called after each real instruction.

The reason why both kinds of polymorphism were introduced is because the translator code is the weakest part of the virus in two ways. It is weak because it is native code, allowing a detection to be guided by the presence of that routine. The polymorphism complicates the detection a little. It is also weak *because it is native code*. Since the routine is small, and the opcodes generally have no alternative values, mutations in this routine are often lethal. The introduction of garbage instructions results in a smaller risk of lethal mutation because the risk is spread over a wider area.

LET ME COUNT THE WAYS

There are three other polymorphism methods which are applied at assemble time, but which are also contained in the assembler code that is the output of the compiled code. The first part of this assemble-time polymorphism is that the native instructions no longer begin on eight-byte boundaries (with the exception of the '_getEIP', '_JnzUp' and '_JnzDown' commands), followed by no-operation instructions to fill the gap. Instead, the block begins on eight-byte boundaries, but the native instructions are placed randomly within the eight-byte block and surrounded by no-operation instructions.

The second part of the assemble-time polymorphism is that since there are only a few commands, many of them are duplicated enough times to fill the 256 slots available. Then, whenever a command is requested, its value is chosen randomly from the list that might contain multiple entries for that command. Thus, even the first generation of the virus will likely have multiple codons referring to the same amino acid.

The third part of the assemble-time polymorphism is applied optionally, by setting the appropriate value in a particular variable in the assembler source code. The alphabet that is used can either be a pre-generated one or a dynamically generated one. The dynamically generated one will fill the slots randomly. There is a minor bug in the routine when assigning the 'unused' command to a slot – the wrong command name is displayed as informational text, but it has no effect on the execution of the virus.

MINOR UPDATE

The virus has some minor changes to its code, too. There is a new hashing algorithm, a new filename, and a rewritten nop insertion routine.

HASH COOKIES

The new hashing algorithm simply sums 16 bits at a time (though two comments in the source code show two different algorithms, neither of which is the one that is used) at each position of the API name, up to and including the final zero (so 'AddAtomA\0' is 'Ad' + 'dd' + 'dA' + 'At' + 'to' + 'om' + 'mA' + 'A\0'). The low 12 bits of the result are retained and compared to an entry in a hash table that the virus carries. The API is considered to be found when the hashes match. This routine is repeated until all of the required APIs are found. The routine loads APIs from 'kernel32.dll' first, and then 'advapi32.dll' (despite the comments in the source code referring again to 'kernel32.dll').

The virus constructs a new filename for the next-generation file, in the same manner as the previous version, and copies itself as 'x:\evolusss.exe', where 'x' is the drive letter taken from the command line. It also copies itself to the next-generation filename.

The virus opens the next-generation file and maps it into memory. As with the previous version, it will flip bits or dwords throughout its body. A bug has been fixed here, which is that the dword exchange will not occur within the last nine bytes of the file, to avoid a possible exception from occuring because of an out-of-bounds access.

NOP INSERTION

The nop insertion routine has been corrected to no longer delete the bytes immediately following the insertion point. Instead, all of the bytes following the insertion point are moved towards the end of the file according to the size of the gap to be introduced. The virus inserts a gap of up to 32 bytes in size, and fills the gap with no-operation commands.

GENE TRANSFER

The horizontal gene transfer works by 'borrowing' bytes from a single randomly located file, and inserting them into the virus body. This can introduce new behaviours if the new bytes happen to make sense in the context of the current code. The routine has a 20% chance of running. If it runs, then it searches within the current directory for any object. It tries to detect directories by checking the exact attribute, instead of masking off all other bits. As a result, it fails to detect a directory if the directory has additional attributes set (such as 'hidden'). However, this is a minor bug which is harmless because any attempt to open the directory will fail. For any file that is found, there is a 20% chance that the routine will attempt to open it. The routine attempts to copy up to ten bytes from a random location in the file it has found to a random location in the virus file. There is a bug in this routine, which is that there is no check that the file it has found is not empty. If the file is empty, then any attempt to copy content from it will cause an exception and the virus will crash.

CODON EXCHANGE

The codon exchange routine searches within the alphabet for amino acids referred to by multiple codons, and randomly exchanges the codons within the virus body.

START AND STOP

The start and stop codons allow the explicit delimiting of a block of valid code (an exon). Any values that appear after a stop codon and before a start codon are junk (introns) that will not be executed by the virus. However, in the event of a mutation that corrupts a stop codon, the junk would become part of the exon until the next stop codon is encountered. This allows for a more rapid introduction of new behaviours.

CONCLUSION

W32/Flibi is more like a life-form than ever before. It looks like a heavily armoured threat whose spread might be difficult to stop – perhaps like a cane toad. However, much like the cane toad, it has a soft underbelly which we can learn to attack.

REFERENCES

[1] Ferrie, P. Virus Bulletin, March 2011, p.4. http://www.virusbtn.com/virusbulletin/archive/2011/03/vb201103-Flibi.

[2] Ferrie, P. Virus Bulletin, May 2011, p.6. http://www.virusbtn.com/virusbulletin/archive/2011/05/vb201105-flibi-evolution.