ABOUT UNDETECTABLE VIRUSES

by p. G. Gyarmati

About the problems with the complexity of detecting viruses.

Lets consider time of detecting a virus in some executable file. First, time depends on a set of possible variants of virus body. The more possible variants there are in the virus body, the more time needed to iterate them while checking files.This way is the simplest one, and it was choosen by viruses, when they morphed into crypt-, polymorphic-, permutating- and metamorphic- ones.

For sure, while checking file, all these variants are iterated very seldom (when their amount is small enough), and in most cases there performed special processing of code and then comparing with some mask, or more complex algorithmic detection -- in this case we consider complexity of such checking, instead number of possible body variants.In case of poly-encrypted virus, this complexity should be increased by a complexity of polymorphic decryptor emulation. In case of simple poly decryptor, it can be the object of detection, instead of virus itself.

Well, at second, the time of detecting a virus depends on a set of possible virus locations (or entrypoints) in file. For example, if virus hooks entrypoint, it will be enough to perform a checking for this virus only at that address.

But, if virus inserts its body into random location in the file, then time of detection for this virus is multiplied by size of this file. (this is true for poly decryptors in which next instructions works depending on result of previous ones). Example of this way is the EPO (UEP) technology.

Let
C - complexity of checking file for some virus;
C_i - complexity of checking virus in file at some specified address, including poly-decryptor emulation and checking for all possible variants of virus body;
I - number of possible addresses in file, where execution of virus body (or part of this body) can be started at.

Then $C = C_i * I$

This trivial formula shows as an algorithm of checking a file for being infected by some complex virus: for each possible address I, where virus can be located, perform checking for presence of this virus (C_i).
Algorithm looks as follows:

```
for(i = 0; i < I; i++)      I
{                           *
init_emul();
emul(codeblock[i]);         Ci
check_virus();              /
}
```

This tells us
-that the bigger files are better targets to be infected,
-that virus should be smaller but more complex,
-that polymorphic decryptor should work longer,
-that number of possible virus locations in file should be maximal.

Well, lets consider now such case,

-when polymorphic decryptors consists of *N* parts,
-which are located in *N* different places of a file, and
-that this decryptor works only when all these *N* blocks are executed consecutively.

In this case,
-if antivirus does not emulate all the program,
-but only combinations of its blocks,
-possibly containing virus decryptor.

Summary complexity of such checking will be equal to:

$$C = C_i * \frac{I!}{(I-N)!}$$

For example, if there are 4 suspicious blocks in program, but there should be only 2 consecutively executed virus blocks, there will be checked 12 variants:

```
{0,1}  {0,2}  {0,3}
{1,0}  {1,2}  {1,3}
{2,0}  {2,1}  {2,3}
{3,0}  {3,1}  {3,2}
```

As a result, virus checking will look as following:

```
for(i1 = 0; i1 < I; i1++)
for(i2 = 0; i2 < I; i2++)
if (i2 != i1)
for(i3 = 0; i3 < I; i3++)
if (i3 != i1)                 I!
if (i3 != i2)              -------
...                       (I-N)!
for(iN = 0; iN < I; iN++)     /
if (iN != i1)                 /
if (iN != i2)                 /
if (iN != i3)                 /
...                           /
{                             *
  init_emul();
  emul(codeblock[i1]);
  ...                        Ci
  emul(codeblock[iN]);        /
  check_virus();              /
}
```

And now there appears a question: how could we know sequence of program instruction execution to insert virus blocks into such places, that they will be executed in some specified order?

This can be achieved by means of tracing the program. And tracing can be performed using win32 debug api. Selected program should be traced for some small (but random) period of time (seconds). While tracing process, the list of executed instructions should be builded for each (or only main) thread. Here is interesting thing: we're tracing not program, but process, i.e. EXE file and some DLL files. As such, there appears possibility to insert these N decryptor blocks into different files, but with known execution order. Fore sure, on small local parts of code all that stuff can be done without tracing, but by means of statical analysis of code instructions. But, for big multithreaded programs, consisting of lots of differents files, and also in case of big distances between viral blocks, statical analysis (i.e. disassembling) is not enough, and tracing (or emulation) is required.

So, we should

1. Select some program.
2. Perform static analysis.
   (parsing on instructions, see Mistfall and ZMist)
3. Perform dynamic analysis (tracing, see Tracer32),
   but taking into account results of previous disassembling.
   For example, breakpoint (INT3) should be inserted into beginning
   of each subroutine, to avoid losing control on callback functions,
   called from non-tracing DLLs.
4. Perform static analysis once again, with more quality,
   because taking into account instruction list,
   obtained while previous tracing.
5. Retry from step 3 if needed.
6. Combine all obtained data.
7. Select some random places of the program,
   which are executed in some known order,
   to insert decryptor parts into.
8. Integrate decryptor parts and encrypted body
   into code section.
9. Build new program.

As a result, by means of synthesing two known technologies, there appears possibility to make virus detection much more harder, and, as such, to hammer another one nail into kaspersky's ass.
Now some details. Viral blocks should be of minimal length (some instructions), but N should be big enough. The sequence of executing these N blocks should be complex, including big amount of iterations. This can be achieved by finding cycles in program while tracing it, and then integrating viral blocks into these cycles.
Well, this all is real, you can believe or not.
Q.E.D.
* * *