Believes about viruses.

The problem nearly as old as the computer networking, the data exchange among computers or even only among computer-people by using data carriers, tapes, floppies, etc. Now we are on this and shall try to explore and understand the situation and also try to glimpse to the future.
This article was first published as a white paper and for the conference: The Truth about Computer Security Hysteria, 2002.

Theoretically you have many possibilities to secure your computer against viruses. Here are some of them for your convenience:

1. Create a hideous system, heavily find vulnerable places,
2. Watch flowing of data and find the pattern of viruses in the communication channels and in files, programs, etc.;
3. Keep under control your system, to reveal vulnerabilities and fix them soon.
4. Designing secure, flawless software.
5. Isolate environment from attackers, etc.

Now let's see how these like in the real life look today. Everyone already understood that viruses are a big role of our increasing networked world. But how big is indeed?

Security ideas.

Most people believes that Linux systems would suffer the same frequency of virus problems as Microsoft Windows if they were as popular as Windows is now. Such a comment ignores several factors that make up the vulnerability profile of Windows with regard to viruses.

The explanation is in the so called *security by obscurity*, which is a principle in security engineering. A system relying on security through obscurity may have theoretical or actual security vulnerabilities, but its owners or designers believe that the flaws are not known, and that attackers are unlikely to find them.

A closed source system looks like such a solution, because the would-be attacker does not have access them. But how true is this? Let's see the facts!

So we must understood that Linux-based systems and other open source OSes actually use the *security through visibility* approach. The security through visibility idea taken by popular open source software projects and includes two *sub-principles* of software security: security through transparency and security through popularity.

The *transparency* means that the source code will be more simple not containing hideous code for obscurity. The idea coming from the experiment that the reverse-engineering is much reliable tool than understanding the source-code, so the source code may be freely spread among people. This will not produce more vulnerability.

The popularity has twofold meaning: the more used software has more enemy, but also have more expert and user who wants to have secure software. By the open source even more experts work with and the vulnerabilities will be recognized soon, and the solution for fixing nearly immediate. Against this is the closed system, where only a few expert are engaged such job, and also the way to recognizing is undetermined. So fixing vulnerabilities are long process.

Microsoft even doesn't fix virus vulnerabilities, they let this work for antivirus companies. This is quite a business, but what price for the users!

A virus is malicious code carried from one computer to another by some kind of medium - often an "infected" file. Once on a computer, it's executed when that file is "opened" in some meaningful way by software on that system. When it executes, it does something unwanted. This often involves, among other things, causing software on the host system to send more copies of infected files to other computers over the network, infecting more files, and so on. In other words, a virus typically maximizes its likelihood of being passed on, making itself contagious.

All of this relies on security vulnerabilities that exist in software running on the host system. For example, some of the most common viruses of the last decade or so have taken advantage of security vulnerabilities in Microsoft Office macro capabilities. Infected files that were opened in a text editor such as Notepad would not then execute their virus payload, but when opened in Office with its macro execution capabilities would tend to infect other files and perhaps even send copies of themselves to other computers via Outlook. Something as simple as opening a macro virus infected file in WordPad instead of Microsoft Word or translating .doc format files into .rtf files so that macros are disabled was a common protective measure in many offices for a while.

Macro viruses are just the tip of the iceberg, however, and are no longer among the most common virus types. Many viruses take advantage of the rendering engine behind Internet Explorer and Windows Explorer that's also used by almost every piece of Microsoft software available to one degree or another, for instance. Windows viruses often take advantage of image-rendering libraries, SQL Server's underlying database engine, and other components of a complete Windows operating system environment as well.

Viruses in the Windows world are typically addressed by antivirus software vendors. These vendors produce virus definitions used by their antivirus software to recognize viruses on the system. Once a specific virus is identified, the software attempts to quarantine or remove the virus - or at least inform the user of the infection so that some kind of response may be made to protect the system from the virus.

This method of protection relies on knowledge of the existence of a virus. This means that a virus against which you are protected has, by definition, already infected already at least one computer or network and done its damage. Will it be you the martyr suffering from the damage to save others or the lucky fellow receiving the new virus definition to your antivirus software?

The life even worse than that, though. Each virus exploits a vulnerability - but they don't all have to exploit *different* vulnerabilities.

In fact, it's common for hundreds or even thousands of viruses to be circulating "in the wild" that, among them, only exploit a handful of vulnerabilities.

This is because the vulnerabilities exist in the software and are not addressed by virus definitions produced by antivirus software vendors.

These antivirus software vendors' definitions match the signature of a given virus — and if they are really well-designed might even match similar, but slightly altered, variations on the virus design. Sufficiently modified viruses that exploit the same vulnerability are safe from recognition.

Viruses can exploit the same vulnerability without being recognized by a virus definition, as long as the vulnerability not stopped by the vendor of the vulnerable software.

This is a key difference between open source operating system projects and Microsoft Windows. Microsoft leaves dealing with viruses to the antivirus software vendors, but open source operating system projects generally fix such vulnerabilities immediately when they're discovered. So they are much less use antivirus software on an open source system. The only exception are the mail server and such as they communicate potentially virus-laden files among computers.

Very easy to produce a new virus that will slip past antivirus software vendor virus definitions, but in the open source software world the many expert always busy with open code will discover and patch a whole new vulnerability with high probability than a hacker.

The complexity of detecting viruses.

There is another question about detecting viruses is the complexity so the time required to that job.

Let's consider the 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 chosen 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 entry points) in file. For example, if virus hooks entry point, 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).

Let

C - complexity of checking file for some virus;

Ci - 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 $C = Ci * I$

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 (Ci).

The algorithm looks as follows:

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

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

Let's 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.

The complexity of such checking will look like:

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

For example, if there are 4 suspicious blocks in the 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}
```

So 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*. The 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.

Very important: 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.

On small local parts of code all this work can be done without tracing, but by means of static analysis of code instructions. For big multithreaded programs, consisting of lots of different files, and also in case of big distances between viral blocks, static analysis (i.e. disassembling) is not enough, and tracing (or emulation) is required.

Now the flow of this work is:

1. Select some program.
2. Perform static analysis (parsing on instructions, e.g.: Mistfall and ZMist)
3. Perform dynamic analysis (tracing, e.g.: Tracer32), but taking into account the 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 synthesizing two known technologies, there appears possibility to make virus detection much harder, and, as such, to hammer another one nail into Kaspersky's bottom.

At last some suggestion.

Viral blocks should be of minimal length (few 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.

Summary.

What I hope showed that viruses need not simply be a "fact of life" for anyone using a computer. Antivirus software is basically just a dirty hack used to fill a gap in your system's defenses left by the negligence of software vendors who are unwilling to invest the resources to correct certain classes of security vulnerabilities.

The truth about viruses is simple, but it's not pleasant. The truth is that you're being taken to the cleaners — and until enough software users realize this, and do something about it, the software vendors will continue to leave you in this vulnerable state where additional money must be paid regularly to achieve what protection you can get from a dirty hack that simply isn't as effective as solving the problem at the source would be.

Also keep in your mind the curious question: *who are the „bad-boys" and why are they „bad" and are they „bad"?*

~~~~~
~~~~~