Virus Verification and Removal Tools and Techniques

David M. Chess

History

This is an updated version of a paper that originally appeared in the November 1991 issue of Virus Bulletin. Since this sort of technology is continually evolving, it seemed reasonable to make an update available on the net; in particular, the virus-removal language has been considerably enhanced since the paper was originally written. Comments are welcome, on VIRUS-L (comp.virus), or directly to the author (chess at watson.ibm.com).

Introduction

The first line of defense against computer viruses consists of programs that detect that something is probably wrong. These include modification detectors, integrity shells, known-virus scanners, access-control programs, and similar things. Their main function is to alert the user of a machine that a virus, some virus, is probably present. The important thing is the alert; since something is likely to be wrong, the user should stop what he is doing, and take action to correct the problem. It doesn't matter much at this stage what the alert says; a first-line anti-virus system that always said simply ``Something virus-like may be going on!'' would be sufficient for most environments, if it was usually right.

Once the alert has been given, and the infected system taken out of immediate contact with other systems, other kinds of software become important. Before we can decide how to clean up an infected system, and even where else to look for infection, we need to know exactly what the infection consists of. Once that has been determined, we can take steps to restore the infected parts of the system to an uninfected state, and to recover from any other damage the virus may have caused. This paper is a description of one part of the second-line toolbox, the virus verifier and remover.

Virus Verifiers

A virus verifier is a program that, given a file or disk that is probably infected with a given virus, determines with a high degree of certainty whether the virus is a known strain, or a new variant. This is, of course, important to know: if the virus is different from any known strain, it will have to be analyzed for new effects before we can be confident that we know just what to do to clean up after it. On the other hand, if the virus is identical to a known strain, we already know what to do. It is particularly important to perform verification in a program that attempts to automatically remove the virus infection from an object, restoring it to its original uninfected form.

Abstractly, a verifier is a program that, given another program as input, determines whether or not the given program is part of the set of possible ``offspring'' of a particular virus. For many classes of viruses, including all the viruses actually widespread at the moment, this is easy to do. Almost all known viruses consist almost entirely of code that does not change from infection to infection, except perhaps for a simple XOR-type garbling, and data areas that are either constant, or change in simple ways (or that can be ignored entirely for the purposes of verification). Given a suspect file F and a known virus V, it is therefore always relatively simple to answer the question ``is F a file that could have been produced by infection with virus V?". It is an open question of some theoretical interest whether or not some future virus might make this harder to do! Reliably determining whether a file is infected with any virus at all is of course known to be impossible, but we have no similar result about determining the presence of a specific virus.

There are various concrete decisions and tradeoffs involved in writing a virus verifier; this section will list a few of them, and the next sections will describe the verifier/remover currently being developed and used at the High Integrity Computing Lab at IBM's Watson Research Center.

A verifier may be an independent tool, or it may be integrated into a virus detector. An integrated detector/verifier can be quicker and more convenient, since there's no need for a user to find and run a verifier once the detector goes off. On the other hand, since most copies of any virus detector will never in fact detect a virus (most of the world's computers are not infected, after all), integrating a verifier along with the detector is in some sense inefficient, in that it adds significant code to the detector that may never be used. Given how much more expensive human time is than CPU time and disk space these days, integrated tools are likely to be more cost-effective in the long run. On the other hand, detection and verification will always be two different activities, because it is very desirable for a detector to detect small variants of known viruses as viruses, whereas a verifier must be able to identify any variation as a variation. Detection algorithms are typically run very often, and must be fast. Verification algorithms, on the other hand, are run rarely (only when a virus is detected), and speed is typically not a major issue.

To determine whether or not a given object is infected with a known strain of a virus, a verifier must know what the known strain looks like. This may be done either with an actual copy of the code of the known strain of the virus, or by using a CRC or similar modification-detection algorithm. It's not generally desirable to include the entire code of a virus with widely-distributed tools, for obvious reasons! On the other hand, even a good difficult-to-invert digital signature algorithm is not as reliable as a byte-for-byte comparison, and it is vulnerable to a virus author intentionally creating a variant that looks to the verifier like a known strain. (This can be made arbitrarily hard through the use of cryptographic checksums and related technologies, at some increase in runtime and complexity.)

Lastly, a verifier may use either special-purpose code, with one or more routines being written in some compiled language for each new strain discovered, or it may be written as an interpreter for a high-level virus-description language. A high-level language is generally simpler to program in reliably; on the other hand, this is only true because it is less expressive, which implies that there will be cases (viruses that are exotically self-garbling, for instance) in which it will be necessary to drop into the lower-level programming language again.

VERV - A Prototype Virus Verifier and Remover

At HICL, we are currently using and developing a virus verifier and remover called ``VERV" for PC-DOS viruses. The current version can verify over 40 different viruses and variants, which accounts for nearly all of the actual infections that we see in day-to-day operation. It has recently been enhanced to attempt to remove about a dozen of the most common file-infecting viruses (we have other tools, which will eventually be integrated, for removing boot-sector-infecting viruses). As well as being used in the lab, and as a research prototype, VERV is used by IBM's internal Computer Emergency Response Teams (CERTs), as part of routine incident handling.

It is an independent tool at the moment; in the long run, we expect to integrate it with our other anti-virus programs. It can use either a CRC algorithm or a byte-for-byte comparison to verify the identity of a virus. In the laboratory, we use the byte-for-byte compare to test new samples against old ones. In the field, our users use the CRC algorithm to verify the virus in infected objects before applying cleanup measures.

VERV includes an interpreter for a small virus-description language. Virus-description languages, for this and other purposes, have been around for some time; Christoph Fischer at the University of Karlsruhe, Morton Swimmer in Hamburg, Alan Solomon in the UK, and no doubt many others in the field, have worked on similar things (personal correspondence; one motivation for this paper is to encourage others, who have perhaps done it better, to publish their work). VERV's language is very simple, and provides for lower-level hooks (instructions to call special-purpose C routines) when a virus requires actions that cannot be described in the high-level language. We will describe the language in some detail, not because it is particularly interesting as a language, or because we think we have it all correct and optimal, but rather so that other people working on the same sorts of things can benefit from both our ideas and our mistakes. We hope this will help inspire continued discussion and exchange.

VERV's virus-description language

The file from which VERV reads virus descriptions consists of a number of virus-description blocks. Each block has the following structure:

```
One or more VIRUS records
A NAME record
One or more LOAD records
Zero or more DEGARBLE and related records
Zero or more ZERO records
One or more check records
Zero or more REPAIR blocks
```

For instance, the block for the Slow-1721 virus currently looks like this:

```
VIRUS slow slow-1721
NAME the Slow-1721 virus
LOAD P-COM 0 6B4
LOAD S-EXE 0 6B4
DEXOR1 001E 06AD 0012 0000      ; Degarble the code
DEXOR1 00EB 0159 0061 0001      ;   and the data area
ZERO 0012 1                     ; Zero the code-garble key
ZERO 0061 1                     ;   and the data-garble key
CODE  0000 00EA 38d5dc08        ; Code up to first data area
CONST 0144 014E 0ff22ad9        ; COMMAND.COM
CODE  015A 063C 74e00962        ; Code between data areas
CODE  0657 06AD ad3b0b41        ; After the second data area
```

The VIRUS records simply give a list of one-word aliases for the virus, that are used on the command line to tell VERV which virus to look for. These aliases are not the full primary name of the virus (that is given on the NAME record); they are just short abbreviations that the user can use on the command line.

A very useful extension here would be for VERV to support virus families, so that a single command would cause testing for all members of the Jerusalem family, or the Flip family, and so on. When integrated into the virus detector, of course, the detector will directly inform the verifier which virus or viruses to test for.

The LOAD records describe where in an infected object of a given type the virus can be found. The tokens on a LOAD record are an object type, followed by either an offset and a length, or the word SPECIAL and a number. The offset tells VERV where, relative to the effective entry-point of that sort of object, to start loading; the length tells how many bytes to load. For viruses that are not always at a fixed offset from the initial entrypoint, the SPECIAL keyword causes VERV to invoke an internal routine, coded in C, to perform the loading.

The Slow virus is an EXE-infector, and a prepending COM infector; the LOAD records in this example tell VERV to load the first 06B4 bytes of a COM-format file, and the first 06B4 bytes after the entry point of an EXE-format file. (EXE-format files are those that begin with the letters ``MZ''; DOS loads these differently from COM-format files, which begin with any other bytes.) Other object types supported include:

- E9-COM, for viruses that infect COM files by changing the first three bytes to a long jump to the virus (E9 is the hex code for a long jump),
- E8-COM, for viruses that infect COM files by changing the first three bytes to a long CALL to the virus (E8 is a long call),
- MBR, for viruses that infect hard disk master boot records and diskette boot records, and fit in a single sector,
- DISKETTE, for other sorts of diskette infectors (those that do not fit in a single sector),
- HARDDISK, for other sorts of hard disk infectors (those that infect system boot records, and/or occupy more than one sector).

A description block will have as many LOAD records as there are types of object that the virus can infect.

The DEXOR1 records tell VERV to perform a certain common type of degarbling: a one-byte XOR with data to be found at a fixed offset in the virus. The details are not terribly important here. A more general record, consisting of just the word DEGARBLE followed by a number, causes VERV to invoke an internal C-language routine to perform degarbling.

Once the loading and degarbling have been done, VERV has a complete ``virus image'' in its internal buffer. A command-line switch (described later) can instruct VERV to save the contents of this buffer to a file, for later examination.

The ZERO records describe variable areas within the virus, that should be set to zero before checks are done. This is really just a convenience, to reduce the number of check-type records needed.

There are three basic types of check records, describing different tests to be done on the degarbled and zero'd virus image now in the buffer:

- CODE records describe areas of virus code. The numbers given are the start and end offsets of the area, and the expected CRC value of the data there. VERV uses a 31-bit CRC, with a custom polynomial. This is not strongly resistant to intentional reverse engineering; a more difficult-to-invert algorithm may be desirable later on. If any CODE areas are found to be different than expected, VERV will report that this is not the usual strain of the virus.
- CONST records describe constant areas that should not change, and whose values effect the actual running of the virus. CONST areas are currently treated exactly like CODE areas.
- TEXT records describe areas of the virus that are not expected to change, but do not significantly effect the operation of the virus. If a sample differs from the given description only in one or more TEXT areas, VERV will report a ``text variant'' of the virus. This is useful for message areas within a virus that are not actually used, or that are simply displayed to the user. These areas can be interesting in tracking how the virus is spreading, by correlating incidents that involve the same ``text variant'', but they do not effect cleanup or prevention.

Normally, VERV performs its CRC calculation on each area within the virus, and compares the results to the expected values. A command-line switch (described in more detail below) can be used to tell VERV to read a standard copy of the virus from another file instead, and do byte-by-byte comparison between the two. This is more reliable, but of course it requires having a sample of the usual strain of the virus present to verify against.

Another example, illustrating the use of special C routines, is the block for the 1701 virus:

```
VIRUS 1701
NAME the 1701 virus
LOAD E9-COM -1 06A5
DEGARBLE 1
CODE  0001 0026 19989c7e        ; Degarble, MOV, jmp-in
CODE  0076 06A4 c03a91c5        ; Main code
```

Here, the ``DEGARBLE 1'' record causes VERV to invoke an internal routine to degarble the data in the buffer, using the 1701's own algorithm. It would be possible to enhance the virus-description language enough that the 1701's degarbling algorithm could be expressed in it directly. This would complicate the language considerably, though, and would somewhat lessen the advantage that a special high-level language has over native C code; so far, we have decided against such enhancements.

Repair

For many viruses and many infected objects, it's possible to restore the object to what it looked like before it was infected, or at least to a state in which it will function in just the same way. Unfortunately, this isn't always possible; the classic example is the 1813 (Jerusalem) virus infecting an EXE-format file. While it's usually possible to undo the infection, sometimes the resulting file is missing data that was in the uninfected original, and it's not always possible to tell that this has happened. The best an 1813-remover can do on the EXE file, therefore, is something that is quite likely to work, but might not. In most cases, though, sufficiently-reliable repair is possible, and particularly in large infections of non-critical machines, repair is sometimes a cost-effective option.

A description of a virus in VERV's language includes one repair block for every type of object that the virus may infect. Each repair block consists of a header record ``REPAIR <object type>'', followed by one or more repair-operation records. Currently defined repair operations include:

- an FCOPY_TO record, that copies bytes from the start of the infected file up to a given number of bytes from the virus entry point (this is used to remove appending viruses),
- an FCOPY_FROM record that copies bytes from the infected file, starting a given number of bytes from the virus entry point, and ending a given number of bytes before the end of the file (this is used to remove prepending viruses),
- a BWRITE record, that copies so many bytes from a given offset in VERV's internal buffer (which initially holds an image of the virus) to a given offset in the file being repaired (this is used, for instance, to repair the first few bytes of an infected COM file, or the header of an infected EXE file),
- a BREAD record, that loads a given number of bytes from a given offset (relative to the start of the infected file) into VERV's buffer,
- an EXE_LENGTH_BUG record, that tells VERV that this particular virus has the common bug that it assumes that the image length in the header of an EXE file is the same as the file's length, and therefore damages (by overlaying some data) any EXE file that contains data after the EXE image,

- a 64K_COM_BUG record, which tells VERV that this virus has the common bug that it assumes that any file it thinks of as a COM file must be less than 64K bytes long,
- an EXE_LENGTH_ADJUST record, that treats two words within the buffer as the ``page count'' and ``last page length'' fields from a DOS EXE-file header, and subtracts a given constant value, adjusting them accordingly,
- an R_SPECIAL record, to cause VERV to invoke an internal C routine to perform some function not directly implemented in the language.

For instance, the repair block for the usual 1813 or Jerusalem virus currently looks like this:

```
REPAIR S-EXE
  EXE_LENGTH_BUG
  FCOPY_TO -0C5
  EXE_LENGTH_ADJUST 0053 0051 0710
  BWRITE 0043 0010 2       ; Fix SP
  BWRITE 0045 000E 2       ; Fix SS
  BWRITE 0047 0014 2       ; Fix IP
  BWRITE 0049 0016 2       ; Fix CS
  BWRITE 0051 0002 4       ; Fix image length

* Fixing COM files
REPAIR P-COM
  64K_COM_BUG
  FCOPY_FROM 0710 -5
```

The two BUG records cause VERV to print warnings to the user that some files may not function correctly, and to refuse to repair (later versions may offer to erase) any files that are obviously not correctly repairable. The FCOPY records pick out just the part of the file that does not contain the virus, and the EXE_LENGTH_ADJUST and BWRITE records restore and replace approximately the original EXE file header. EXE files that are successfully repaired will differ from the original file only in having been rounded up to a multiple of sixteen bytes (and the corresponding change in the EXE file header).

After repair is completed, VERV restarts processing on the repaired file, to ensure that there is not another instance of the virus present. If the virus is present in the file multiple times, all will be removed. Once VERV is integrated with a virus scanner, the repaired file will be automatically re-scanned for all viruses, and any found will be re-verified and removed.

Repair processing is only performed if the user has requested it on the command line, and if VERV finds that the virus is indeed exactly the known strain of the virus. In small infections, or in situations where correct operation of the objects involved is particularly crucial, we continue to recommend that infected objects be destroyed (files erased, diskettes formatted, and so on), and replaced from uninfected sources.

VERV Options

The functions of VERV's command-line switches include:

- Reading the virus to be tested from an image file, instead of from a normally-infected object; this can be useful, for instance, in testing a boot-sector infector that has been received as a binary dump of the boot sector rather than on diskette.
- Overriding the default virus-description file (contained within tne program itself), allowing easy testing of new or experimental descriptions.
- Producing detailed progress messages and data displays during processing, to help pinpoint differences found or errors encountered.
- Specifying that, rather than using a CRC, VERV should compare the relevant parts of the object to be tested with a standard sample of the virus

stored in an image file, or a standard infected specimen.

- Producing a dump of the virus image, after all degarbling, but before any zeroing has been done. This image can then be used for storage, analysis, or transmission, or for later use as input to VERV for byte-by-byte comparisons.

Status and Future Goals

VERV is currently in use by a small number of people within IBM who deal with virus infections. Its availability has greatly reduced the time spent by technical people in doing semi-manual verification, and has therefore sped up the response time to virus incidents. Adding a typical newly-analyzed virus to VERV is generally quite simple, involving a few lines in the VERV language, and sometimes a small piece of C code to handle a new garbling algorithm.

The virus-removal language has just recently been implemented, and is not yet in wide use.

Our near-term plans for VERV include support for families of viruses, and the ability to verify a virus in a number of objects at once. This will ease integration with our virus detectors; when a detector detects a signature that corresponds to a virus, or a family of viruses, in a number of files, it will be able to verify the identity of the virus with a single call to VERV.

If transmission bandwidth, CPU cycles, and disk space were free, and programming was easy, every workstation would be protected by a seamless ``immune system''. Objects infected with existing viruses would be detected automatically, the identity of the virus verified and reported to a central location, and the object destroyed or repaired, with minimal user intervention. New viruses would be detected automatically with some high degree of confidence, first-pass signature patterns would be extracted automatically where possible and communicated to a central clearinghouse, along with a sample of the suspicious object. Viruses would very rarely, if at all, spread widely.

One of our main focuses at HICL is studying what part of that ideal scenario is feasible, in both current and future systems. The prototype VERV is a small part of our experimentation with parts of that system that are also immediately useful to users in the near term. We would welcome similar descriptions by others in the field, of work that they are doing in similar directions.