COMPUTER RECREATIONS

Of worms, viruses and Core War

by A. K. Dewdney

"The only truly secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards—and even then I have my doubts."
—EUGENE H. SPAFFORD

The knock on the door had a palpable urgency that brought the computer-center director's head up sharply from the pile of papers before him. He grunted loudly and the computer operator entered.

"Something's gone wrong. We have some very weird processes going on. We're running out of memory. I think we've got a virus in the system."

If the center had been equipped with claxons, the director would undoubtedly have set them off.

Such a scene has been played out in one form or another all too often in recent years, and as a result computer viruses have been increasingly in the news. In fact, this department has been cited more than once in connection with the rash of virus outbreaks, probably because it is an instigator of Core War, a game in which computer programs are purposely designed to destroy one another. But, as we shall see, Core War has no direct connection with the infections.

To understand how a computer virus works one must first understand in great detail the system in which it operates. The same thing applies for understanding the operation of worms, logic bombs and other threats to computer security. This simple observation has two immediate implications. First, journalists are likely to misreport or distort virus news stories for quite innocent reasons: most reporters are more or less mystified by the internal workings of computers. Second, public descriptions of a computer virus—even fairly detailed ones—cannot be exploited to reconstruct the virus except by someone who has the requisite knowledge of the affected computer system to begin with. A knowledgeable "technopath" who is bent on destroying other people's programs and data hardly needs to read a magazine or newspaper article to begin imagining ways to construct a virus. There is consequently no harm in describing how viruses and other destructive programs work. (Indeed, such a description is probably constructive in that it might stimulate efforts to protect computer systems.)

One must distinguish from the start between the two commonest types of malignant program. A virus rides piggyback on other programs; it cannot exist by itself. A worm, on the other hand, can pursue an independent existence, more in the manner of a bacterium. Both kinds of "infection," like all programs, depend on an operating system.

Most readers know that a running computer consists of both hardware and software. In front of me at the moment, for example, is a piece of hardware: an Apple IIc. Inside the machine's memory is software: a program called the Appleworks Word Processor. The program transfers the characters I type on the keyboard into a section of memory reserved by the program for text.

But the word-processing program is not able to run by itself. The program depends on an operating system that, among other things, translates it into a special machine language that enables the hardware to carry out the program's instructions. The operating system for a personal computer normally resides on a disk. To do anything on such a machine (from writing to playing games), the disk operating system (DOS) must first be loaded into the computer's hardware memory. In a home computer the DOS is usually loaded automatically from a disk, which may or may not contain the program one wants to run, as soon as one switches on the computer.

To run a particular program on my personal computer, I must type the name of the program into the computer. The computer's DOS then searches through the disk for a program with that name, loads it into memory and runs it—instruction by instruction—as is shown in the middle illustration on the opposite page.

In loading the program the DOS sets aside part of the hardware memory not only for the program but also for the program's "work space." Here the program will store all the values assigned to its variables and its arrays, for example. In doing all of this the DOS is normally careful not to overwrite other programs or data areas, including whatever part of the DOS happens to be in memory. The DOS is equally careful in storing programs or data onto a disk.

Often a programmer may find it necessary to employ the commands the DOS itself uses, which can generally be found in the appropriate manual. Such commands make it possible to write a subprogram that can read files from disk into memory, alter the files and then write them back onto the disk—sometimes with malicious intent.

Here is a sample virus subprogram that does just that. It contains a mixture of pseudo-DOS commands and subroutines: small, internal programs (whose component instructions are kept separate from the subprogram's main body) that carry out specific missions whenever they are called.

```
this := findfile
LOAD (this)
loc := search (this)
insert (loc)
STORE (this)
```

The subroutine designated findfile opens the directory of executable files, or programs, on a disk, picks a random file name and assigns the name of that file to a variable called *this*. The next line of the program makes use of the pseudo-DOS command LOAD to copy the file into the computer's memory. Another subroutine called search then scans the program just loaded, looking for an instruction in it that can serve as a suitable insertion site for a virus. When it finds such an instruction, it determines the instruction's line number and assigns its value to the variable called *loc*.

At this point the virus subprogram is ready to infect the program it has randomly picked. The subroutine insert replaces the selected instruction with another instruction (such as a call for a subroutine) that transfers execution to a block of code containing the basic virus subprogram, which is appended to the end of the program. It then adds the program's original instruction to the end of the appended subprogram followed by a command that transfers execution to the instruction following the insert in the host program.

In this way when the virus subprogram is executed, it also executes its host program's missing instruction. The execution of the original program can then proceed as though nothing unusual had occurred. But in fact the virus subprogram has momentarily usurped control of the DOS to replicate itself into another program on the disk. The process is illustrated graphically at the right in the illustration below. When the newly infected program is subsequently loaded by the DOS into the computer's memory and run, it will in turn infect another program on the disk while appearing to run normally.

As early as 1984 Fred S. Cohen carried out controlled infection experiments at the University of Southern California that revealed—to his surprise—that viruses similar to the one I have just described could infect an entire network of computers in a matter of minutes. In order to explain the kinds of damage such viruses can do, I shall adapt Cohen's generic virus, writing it in a pseudolanguage.

```
1234567
main program:
1. infect
2. if trigger pulled, then do damage
3. go to host program
subroutine: infect
  1. get random executable file
  2. if first line of file = 1234567, then
     go to 1, else prepend virus to file
subroutine: trigger pulled
subroutine: do damage
```

Cohen's generic virus is generic in all but its attachment site: instead of inserting itself in the middle or at the end of the host program, it attaches itself to the beginning. The first line of the virus program is the "recognition code" 1234567. The main program first calls up the subroutine infect, which randomly retrieves an executable file from a disk and checks whether the first line of that file happens to be 1234567. If it is, the program has already been infected and the subroutine picks another program. If the subroutine happens to find an uninfected file, it "prepends" the entire virus program to the target program. This means simply that it places itself at the head of the program and arranges to get itself executed first before transferring control back to the infected program.

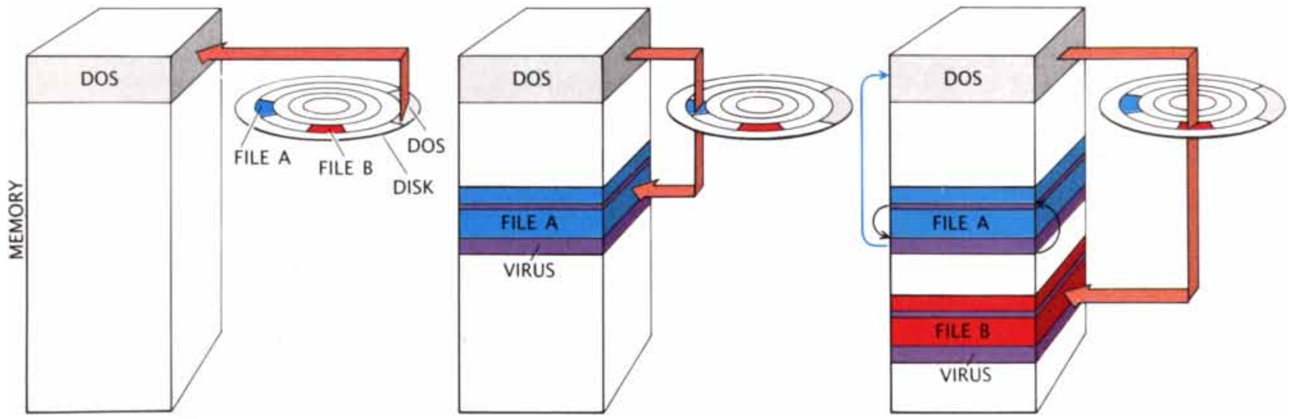
The next two subroutines call for a triggering condition and for some damage to be done. The triggering condition might be a certain date reached by the system clock, or perhaps the deletion of a certain employee's name from the payroll file. The damage to be done might be the erasure of all files or, more subtly, the random alteration of bits in just a few places. The possibilities are endless.

Triggering conditions and damage bring us to the edge of moral territory. I think there is no question that willfully perpetrating damage to computer files by means of a virus or other destructive program is a reprehensible act. Computer programs and data are not just impersonal strings of 0's and 1's but also miniature cities of thought. Why would anyone want to destroy the work of another?

Writers of virus programs do so for a variety of reasons. For example, a programmer in a large company might secretly harbor a grudge against the company's management. He might implant a virus for the day his employment is terminated. When that happens, his records will be deleted from the payroll file, triggering the virus. As a result, valuable company data and programs might either disappear or develop serious and costly errors. Replacing faulty data and programs with backups stored on other media might be of no avail, since if the backups were made recently, they too might be infected.

Of course, the kind of destruction just described would ordinarily take place in a multiple-user computer system. The operating system in this kind of computer environment is considerably more complex than a disk operating system for a personal computer. For one thing, the fact that so many users share the same facilities requires an operating system that protects users as much as possible from inadvertent or deliberate interference. Even here, however, viruses are possible, but they require much more sophistication. Usually they exploit a flaw in some part of the operating system—a bug, so to speak—as was evidenced by the "virus" (actually it was a worm) that spread throughout the Internet last fall.

During the evening of November 2, 1988, someone ran a worm program on one of the several thousand North American computers interconnected through a data-communications network called the Internet. The network connects machines at universities, businesses, Government agencies such as the National Aeronautics and



Loading the* DOS *into memory* (left) *allows a program to be read* (middle) *and a virus* (right) *to replicate itself

INSTRUCTION	EXPLANATION
DAT B	A nonexecutable data statement; B is the data value.
MOV A B	Move contents of address A to address B.
ADD A B	Add contents of address A to address B.
SUB A B	Subtract contents of address A from address B.
JMP B	Transfer control to address B.
JMZ A B	Transfer control to address A if contents of address B are zero.
JMN A B	Transfer control to address A if contents of address B are not zero.
DJN A B	Subtract 1 from contents of address B and transfer control to address A if contents of address B are not zero.
CMP A B	Compare contents of addresses A and B; if they are equal, skip the next instruction.
SPL B	Split execution between next instruction and the instruction at address B.

A summary of Core War instructions

Space Administration and even some military installations. With frightening speed the worm spread to more than 1,000 machines during that evening and the next day. As copies of the worm proliferated, operators of individual systems noticed memory utilization soaring and machine response becoming sluggish. The worm did not attack files or other programs, however. It seemed content merely to proliferate throughout as many host machines as possible. Even so, the damage in lost time was immense.

As I mentioned above, a worm is a program that pursues an independent existence within a computer; it moves from computer to computer on its own, leaving duplicates of itself in each machine. The Internet worm consisted of two parts, a vector and a body. Starting from a given host computer, the worm would seek a new host by sending its vector to another computer. Once inside the machine, the vector would establish a communication link through which the worm's body could be sent. Details of this attack were revealed by Eugene H. Spafford of Purdue University in a 40-page document a few weeks after the event. One example of the worm's operation shows the cleverness of its creator.

UNIX, the operating system of choice on many of the Internet computers, allows processes to take place in the computer that are not associated with any particular user. Such independent processes are called demons. One demon, called fingerd (pronounced fingerdee), enables users to get information about other users. Such a service is desirable in a computing environment in which users must share programs and data for research and development purposes.

The worm in its current host computer would send a message to one of the other potential host computers on its list (which was obtained illegally). In requesting the services of the fingerd demon, the worm gave it some information, just as an ordinary user might. But the worm supplied so much information to the demon that the data filled the space reserved for it in the computer's memory and overflowed into a "forbidden" area.

The area that was thus overwritten was normally reserved for instructions that fingerd consulted in deciding what to do next. Once inside such an area the worm (whose body still inhabited the original host machine) invoked a so-called command interpreter of the new machine, effectively claiming a small piece of the UNIX operating system all to itself. After the command interpreter was at its disposal, the worm transmitted some 99 lines of source code constituting the vector. On the worm's command, the unwitting potential host then compiled and ran the vector's program, virtually guaranteeing infection.

The vector program hid itself in the system by changing its name and deleting all files created during its entry into the system. After doing that it established a new communication channel to the previous host and, using standard system protocols, copied over the files making up the main body of the worm.

Once inside a new host computer, the worm's main job was to discover the names and addresses of new host machines by breaking into areas reserved for legitimate users of the system. To do so it relied on an elaborate password-guessing scheme that, owing to the carelessness with which most users choose passwords, proved rather successful. When it had a legitimate user's password, the worm could pretend to be the user in order to read what he or she may have had in the computer's memory and to discover the names of other computers in the Internet that it could also infect.

According to Spafford, most of the UNIX features (or "misfeatures," as he calls them) that allowed the worm to function as it did have been fixed. Yet the fact has not allayed his worries about computer security, as the quotation at the beginning of this article reveals. Perhaps he was thinking of Cohen's theoretical investigation of viruses, which might apply to worms just as well.

If technopaths insist on vandalizing computer systems, it may be time to form a Center for Virus Control. During the Internet worm crisis teams at the University of California at Berkeley and a few other Internet stations were able to capture copies of the worm, analyze its code and determine how it worked. It would seem reasonable to establish a national agency that would combat computer viruses and worms whenever and wherever they break out—particularly if computer infections are destined to increase. Although the Internet experience hinted at the horrors that may still come, it also showed the efficacy of an organized resistance against them.

Cohen has established that it is impossible to write a computer program that will detect every conceivable virus, even though a defense can be constructed against any given virus. On the other hand, for any such defense there are other viruses that can get around it. According to Cohen, this ominous state of affairs might subject future computing environments to a kind of evolution in which only the fittest programs would survive.

The situation is reminiscent of Core War, a computer game I have written about in previous columns [see SCIENTIFIC AMERICAN, May, 1984, March, 1985, and January, 1987]. But a Core War program does not bully innocent systems. It picks on someone its own size: another Core War program. The two programs engage in subtle or blatant conflict in a specially reserved area of a computer's memory called the coliseum. There is no danger of a Core War program ever escaping to do damage in the real world, because no Core War program or anything like it would ever run effectively in a normal computing environment. Core War programs are written in a language called Redcode that is summarized in the table on the opposite page.

Perhaps a simple example of such a program will serve to introduce the game to readers not already familiar with it. Here is a program called DWARF that launches a 0-bomb into every fifth memory location:

```
DAT –1
ADD #5   –1
MOV #0   @–2
JMP –2
```

The memory coliseum that all Core War programs inhabit consists of several thousand addresses, or numbered memory cells, arranged in a long strip. The instructions that make up DWARF, for example, occupy four consecutive addresses in the coliseum, say 1001, 1002, 1003 and 1004.

The DAT statement serves to hold a value that will be used by the program (in this case –1) at the address 1001. The ADD statement adds the number 5 to the location that is –1 units away from the ADD statement. Since the ADD statement has address 1002, it adds 5 to the number stored at the previous address, namely 1001, changing the –1 to a 4. The MOV command moves the number 0 into the memory cell referred to by @–2. Where is that? The address is found by referring to the DAT statement two lines in front of the MOV command. There one finds the address where the program will put the number 0. The final command, JMP, causes execution of the DWARF program to jump back two lines, to the ADD command. This begins the process all over again.

The second time around, DWARF will change the contents of the DAT cell to 9 and then deliver a 0 to that memory address. If an enemy program happens to have an instruction at that address, it will be rendered inoperable and the program will perhaps "die" as a result.

In this manner DWARF goes on dropping 0-bombs on every fifth location until it reaches the end of memory—but memory never ends, because the last address is contiguous to the first. Consequently DWARF's bombs eventually begin to fall nearer and nearer to itself. Yet because DWARF is only four instructions long and the number of memory cells is normally a multiple of 10, DWARF avoids hitting itself and lives to fight on—albeit blindly and rather stupidly.

Over the past few years Core War has evolved into a rather sophisticated game with numerous strategies and counterstrategies. There are programs that spawn copies of themselves, that launch hordes of mindless one-line battle programs and that even repair themselves when they are hit.

The International Core Wars Society, which currently has its headquarters in Long Beach, Calif., and branches in Italy, Japan, Poland, the Soviet Union and West Germany, organizes annual tournaments in which a programmer's skills are put to the test. Readers interested in joining a Core War chapter should contact William R. Buckley at 5712 Kern Drive, Huntington Beach, Calif. 92649.

In the 1987 tournament the Japanese entries gave the North American warrior programs a run for their money. The winner of the most recent tournament, held last December in Los Angeles, was a program from the Soviet Union called, oddly enough, COWBOY. Written by Eugene P. Lilitko of Pereslavl-Zalesky, a small city northeast of Moscow, COWBOY appeared to watch for "bombing runs" by enemy programs, to move itself out of harm's way and then to retaliate massively. Lilitko won the first prize of $250. The second prize of $100 went to Luca Crosara of Pistoia, Italy. The third prize of $50 was won by Douglas McDaniels of Alexandria, Va.

In closing I quote Spafford once again: "Writing and running a virus is not the act of a computer professional but a computer vandal." Let those who would even contemplate such an act try Core War instead.

I should like to thank Cohen, Spafford and John Carroll, a computer-security expert at the University of Western Ontario, for help with this article.

FURTHER READING

COMPUTER VIRUSES. In *Computers and Security,* Vol. 7, No. 2, pages 117-125, 139-184; April, 1988.

COMPUTER VIRUSES. Peter J. Denning in *American Scientist,* Vol. 76, No. 3, pages 236-238; May-June, 1988.

SCIENTIFIC AMERICAN

CORRESPONDENCE

Offprints of more than 1,000 selected articles from earlier issues of this magazine, listed in an annual catalogue, are available at $1.25 each. Correspondence, orders and requests for the catalogue should be addressed to W. H. Freeman and Company, 4419 West 1980 South, Salt Lake City, Utah 84104. Offprints adopted for classroom use may be ordered direct or through a college bookstore. Sets of 10 or more Offprints are collated by the publisher and are delivered as sets to bookstores.

Photocopying rights are hereby granted by Scientific American, Inc., to libraries and others registered with the Copyright Clearance Center (CCC) to photocopy articles in this issue of SCIENTIFIC AMERICAN for the flat fee of $1.25 per copy of each article or any part thereof. Such clearance does not extend to the photocopying of articles for promotion or other commercial purposes. Correspondence and payment should be addressed to Copyright Clearance Center, Inc., 21 Congress Street, Salem, Mass. 01970. Specify CCC Reference Number ISSN 0036-8733/89. $1.25 + 0.00.

Editorial correspondence should be addressed to The Editors, SCIENTIFIC AMERICAN, 415 Madison Avenue, New York, N.Y. 10017. Manuscripts are submitted at the authors' risk and will not be returned unless accompanied by postage.

Advertising correspondence should be addressed to Advertising Manager, SCIENTIFIC AMERICAN, 415 Madison Avenue, New York, N.Y. 10017.

Address subscription correspondence to Subscription Manager, SCIENTIFIC AMERICAN, P.O. Box 3187, Harlan, IA. 51593. Telephone inquiries: 1-800-333-1199, U.S. only; other 515-247-7631/32. The date of the last issue on subscriptions appears on each month's mailing label. For change of address notify us at least four weeks in advance. Please send your old address (if convenient, on a mailing label of a recent issue) as well as the new one.

Name ____________________

New Address ____________________

Street ____________________

City ____________________

State and ZIP ____________________

Old Address ____________________

Street ____________________

City ____________________

State and ZIP ____________________