By and large, shared computer systems today are astonishingly insecure. And users, on the whole, are blithely unaware of the weaknesses of the systems in which they place — or rather, misplace — their trust. Taken literally, of course, it is meaningless to "trust" a computer system as such, for machines are neither trustworthy nor untrustworthy. These are human qualities. In trusting a system one is effectively trusting all those who create and alter it — in short, all those who have access to it (whether licit or illicit). It is easy to forget that security is a fundamentally human issue.

This article aims not to solve security problems but to raise consciousness of the multifarious cunning ways that systems can be infiltrated, and the subtle but devastating damage that can be done by an unscrupulous "bad guy". It is comforting, but highly misleading, to imagine that technical means of enforcing security have guaranteed that the systems we use are safe. It is true that in recent years some ingenious procedures have been invented to preserve security. For example, the advent of "one-way functions" (explained below) has allowed the password file, once a computer system's central stronghold, to be safely exposed to casual inspection by all and sundry. But despite these innovations, astonishing loopholes exist in practice.

The next section discusses the general trend towards "open-ness" of systems, exemplified by the public password file just mentioned. There are manifest advantages in ensuring security by technical means rather than by keeping things secret. Not only do secrets leak, but as individuals change projects, join or leave the organization, become promoted and so on, they need to learn new secrets and forget old ones. With physical locks one can issue and withdraw keys to reflect changing security needs. But in computer systems, the keys constitute information, which can be given out but not taken back. In practice, such secrets require considerable administration to maintain properly. And in systems where security is maintained by tight control of information, *quis custodiet ipsos custodes* — who will guard the guards themselves?

Following that we introduce a pot-pourri of simple insecurities that many systems suffer. These are, in the main, exacerbated in open systems where information and programs are shared amongst users — just those features that characterize pleasant and productive working environments. Next we examine the idea of a Trojan horse, the saboteur's basic tool — a widely trusted program which has been surreptitiously modified to do bad things in secret. We also look at the kind of things that bugged programs might do, from minor but rankling irritations to holding users to ransom. The next section shows how the inevitable fragilities of operating systems can be exploited by constructing programs which behave in some ways like primitive living organisms. Programs can be written which spread bugs like an epidemic throughout a system. They hide in binary code, effectively undetectable (because nobody ever examines binaries). They can remain dormant for months or years, perhaps quietly and imperceptibly infiltrating their way into the very depths of a system, then suddenly pounce, causing irreversible catastrophe. Following that we examine an extremely clever and subtle bug which survives recompilation despite the fact that there is no record of it in the source program. This is the ultimate parasite. It cannot be detected because it lives only in binary code. And yet it cannot be wiped out by recompiling the source program! Lastly we very briefly consider whether these techniques, which are developed and explained in the context of multi-user timesharing operating systems, pose any threats to computer networks or even stand-alone micros.

Although the potential has existed for decades, the possibility of the kind of "deviant" software described here has been recognized only recently. Or has it? Probably some in the world of computer wizards and sorcerers have known for years how systems can be silently, subtly infiltrated — and concealed the information for fear that it might be misused (or for other reasons). But knowledge of the techniques is spreading, and I believe it behoves us all — professionals and amateurs alike — to understand just how our continued successful use of computer systems hangs upon a thread of trust. Those who are ignorant of the possibilities of sabotage can easily be unknowingly duped by an unscrupulous "bad guy".

The moral is simple. Computer security is a human business. One way of guaranteeing security is to keep things secret, trusting people (the very people who can do you most harm) not to tell. The alternative is to open up the system and rely on technical means of ensuring security. But a system which is really "open" is also open to abuse. The more sharing and productive the environment, the more potential exists for damage.

You have to trust your fellow users, and educate yourself. If mutual trust is the cornerstone of computer security, we'd better know it!

The trend towards keeping no secrets

Modern thinking is that computer systems should maintain their security not by keeping secrets as they used to, but by clever technical mechanisms. Such devices include electronic locks and keys, and schemes for maintaining different sets of "permissions" or "privileges" for each user. The epitome of this trend towards open systems is the well-known UNIX operating system, used throughout this article to illustrate the ideas. Its developers, Dennis Ritchie and Ken Thompson, who in 1983 received the prestigious ACM Turing Award for their work, strove to design a clean, elegant operating system that was distributed in source form so that it could be understood, maintained, and modified by users. Ken Thompson has been one of the prime contributors to our knowledge of computer (in)security, and was responsible for much of the work described in this article.

The most obvious sense in which the UNIX system (along with many others) is "open" is illustrated by looking at the password file. Yes, there is nothing to stop you looking at the password file! Each registered user has a line in it, and Figure 1 shows mine. It won't help you to impersonate me, however, because what it shows in the password field is not my password but a scrambled version of it. There is a program which computes encrypted passwords from plain ones, and that is how the system checks my identity when I log in. But the program doesn't work in reverse — it's what is called a "one-way function". It is effectively impossible to find the plain version from the encrypted one, even if you know exactly what the encryption procedure does and try to work carefully backward through it. *Nobody* can recover my plain password from the information stored in the computer. If I forget it, not even the system manager can find out what it is. The best he can do is to reset my password to some standard one, so that I can log in and change it to a new secret password. (Needless to say this creates a window of opportunity for an infiltrator.) The system keeps no secrets. Only I do.

Before people realized how to construct and use one-way functions, computer systems used to maintain a password file which gave everyone's plain password for the login procedure to use. Needless to say, this was the prime target for any "bad guy" who tried to break security, and was the bane of system managers because of the completely catastrophic nature of a leak. Systems which keep no secrets avoid an unnecessary Achilles heel.

Another sense in which UNIX is "open" is the accessibility of the source code of the system. The software has been distributed (to universities) in source form so that maintenance can be done locally. The computer science research community has enjoyed numerous benefits from this enlightened policy (one is that we can actually look at some of the security problems discussed in this article). Of course, in any other system there will inevitably be a large number of people who have or have had access to the source code — even though it may not be publicly accessible. Operating systems are highly complex pieces of technology, created by large teams of people. A determined bad guy may well be able to gain illicit access to source code. Making it widely available has the very positive effect of bringing the problems out into the open and offering them up for public scrutiny. Much has been written about the security of UNIX (eg Ritchie, 1981; Grampp & Morris, 1984; Reeds & Weinberger, 1984; Filipski & Hanko, 1986).

Were it attainable, perfect secrecy would offer a high degree of security. Many people feel that technical innovations like one-way functions and open password files provide comparable protection. The aim of this article is to show that this is a dangerous misconception. Sad to say, in practice security is severely compromised by people who have intimate knowledge of the inner workings of the system — precisely the people you rely on to *provide* the security. This does not cause problems in research laboratories because they are founded on mutual trust and support. But in commercial environments it is vital to be aware of any limitations on security. And while it would be nice if it were otherwise, we must face the fact that in a hostile

and complex world, computer security is best preserved by maintaining secrecy.

A pot-pourri of security problems

This article will introduce some rather subtle ways of infiltrating computer systems — not, I hasten to add, so that you can try them out yourself but so you can guard against others who might. Vigilance is your only real defense. But before we get on to the clever stuff, let's look very briefly at a few simple ways that security might be compromised. In this business, forewarned is very definitely forearmed.

Guessing a particular user's password. Whether your password is stored in a secret file or encrypted by a one-way function first, it offers no protection if the bad guy can guess it. This will be hard if it is chosen at random from a large enough set. But for a short sequence of characters from a restricted alphabet (like the lower-case letters), the bad guy could easily try all possibilities. And in an open system which gives access to the password file and one-way function, this can be done mechanically, by a program!

Figure 2 plots the number of different passwords against the length of the password, for several different sets of characters. For example, there are around ten million (10^7) possibilities for a 5-character password chosen from the lower-case letters. This may seem a lot, but if it takes 1 msec to try each one, they can all be searched in around 3 hours. If 5-character passwords are selected from the 62 alphanumerics, there are more than 100 times as many and the search would take over 10 days.

To make matters worse, people have a strong propensity to choose as their passwords such things as

- English words
- English words spelled backwards
- first names, last names, street names, city names
- the above with initial upper-case letters
- valid car license numbers
- room numbers, social security numbers, telephone numbers, etc.

Of course, this isn't particularly surprising since passwords have to be somehow memorable in order to be remembered! But it makes it easy for an enterprising bad guy to gather a substantial collection of candidates (from dictionaries, mailing lists, etc) and search them for your password. At 1 msec per possibility, it takes only 4 minutes to search a 250,000-word commercial dictionary.

A study some years ago of a collection of actual passwords that people used to protect their accounts revealed the amazing breakdown reproduced in Figure 3 (Morris & Thompson, 1978). Most came into one of the categories we have discussed, leaving less than 15% of passwords which were hard to guess. Where does your own password stand in the pie diagram?

Finding any valid password. There is a big difference between finding a chosen person's password and finding a valid password for any user. You could start searching through the candidates noted above until you found one which, when encrypted, matched one of the entries in the password file. That way you find the most vulnerable user, and there are almost certain to be some crazy enough to use 1- or 2-character passwords, four-letter words, or whatever. Hashing techniques make it almost as quick to check a candidate against a group of encrypted passwords as against a single one.

To protect from this kind of attack there is a technique called "salting". Whenever a user's password is initialized or changed, a small random number called the "salt" is generated (perhaps from the time of day). Not only is this combined with his password when it is encrypted, but as Figure 1 shows it is also stored in the password file for everyone to see! Every time someone claiming to be that user logs in, the salt is retrieved and combined with the password he offers before being encrypted and compared with whatever is stored in the password file.

Since all can see the salt, it is no harder for the bad guy to guess an individual user's password. He can salt his guesses just as the system does. But it *is* harder to search a group of passwords, since the salt will be different for each, rendering it meaningless to compare a single encrypted password against all those in the group.

Forced-choice passwords. The trouble with letting users choose their own passwords is that they often make silly, easily-guessed, choices. Many systems attempt to force people to choose more "random" passwords, and force them to change their password regularly. All these attempts seem to be complete failures. The fundamental problem is that people have to be able to remember their passwords, because security is immediately compromised if they are written down.

There are many amusing anecdotes about how people thwart systems that dictate when they have to change their passwords. I had been using a new system for some weeks when it insisted that I change my password. But I resented it giving the orders, so I gave it my old password as the new one. However, it was programmed to detect this ruse and promptly told me so. I complained to the user sitting beside me. "I know," he said sympathetically. "What I always do is to change it to something else and then immediately change it back again!" I heard of another system which remembered the last several passwords you used, and insisted on a once-a-month change. So everyone began to use the name of the current month as their password!

Wiretaps. Obviously any kind of password protection can be thwarted by a physical wiretap. All the bad guy has to do is to watch as you log in and make a note of your password. The only defense is encryption at the terminal. Even then you have to be careful to ensure that the bad guy can't intercept your encrypted password and pose as you later on by sending this *encrypted* string to the computer — after all, this is what the computer sees when you log in legitimately! To counter this, the encryption can be made time-dependent so that the same password translates to different strings at different times.

Assuming that you, like 99.9% of the rest of us, don't go to the trouble of terminal encryption, when was the last time you checked the line between your office terminal and the computer for a physical wiretap?

Search paths. As will be documented later in this article, you place yourself completely in the hands of another user whenever you execute one of his programs, and he can do some really nasty things like spreading infection to your files. The point I want to make here is that you don't necessarily have to execute his program overtly. In fact many systems make it very easy to use other people's programs without even realizing it. This is usually a great advantage, for you can install programs so that you or others can invoke them just like ordinary system programs, thereby creating personalized environments.

Figure 4 shows part of the file hierarchy in our system. The whole hierarchy is immense. I alone have something like 1650 files, organized into 200 of my own directories under the "ian" node shown in the Figure, and there are hundreds of other users. So what is shown is just a very small fragment. Now users can set up a "search path" which tells the system where to look for programs they invoke. For example, my search path includes the 6 places that are circled. Whenever I ask for a program to be executed, the system seeks it in these places. It also searches the "current directory" — the one where I happen to be at the time.

To make it more convenient for you to set up a good working environment, it is easy to put someone else's file directories on your search path. But then he can do arbitrary damage to you, sometimes completely accidentally. For example, I once installed a spreadsheet calculator called "sc" in one of my directories. Unknown to me, another user suddenly found that the Simula compiler stopped working and entered a curious mode where it cleared his VDU screen and wrote a few incomprehensible characters on it. There was quite a hiatus. The person who maintained the Simula compiler was away, but people could see no reason for the compiler to have been altered. Of course, told like this it is obvious that the user had my directory on his search path and I had created a name conflict with *sc* the Simula compiler. But it was not obvious to the user, who rarely thought about the search path mechanism. And I never use the Simula compiler and had created the conflict in all innocence. Moreover, I didn't even know that other users had my directory on their search paths! This situation caused only frustration before the problem was diagnosed and "fixed". But what if I were a bad guy who had created the new *sc* program to harbor a nasty bug (say one which deleted the hapless user's files)?

You don't necessarily have to put someone on your search path to run the risk of executing his programs accidentally. As noted above, the system (usually) checks your current working directory for the program first. Whenever you change your current workplace to another's directory, you might without realizing it begin to execute programs he has planted there.

Suppose a bad guy plants a program of his own with the same name as a common utility program. How would you find out? The UNIX *ls* command lists all the files in a directory. Perhaps you could find imposters using *ls*? — Sorry. The bad guy might have put a program in his directory called *ls* which simulated the real *ls* exactly except that it lied about its own existence and that of the planted command! The *which* command tells you which version of a program you are using — whether it comes from the current directory, another user's directory, or a system directory. Surely this would tell you? — Sorry. He might have written his own *which* which lied about itself, about *ls*, and about the plant.

If you put someone else on your search path, or change into his directory, you're implicitly trusting that he isn't a bad guy (or a good guy being bad by mistake). You are completely at a user's mercy when you execute one of his programs, whether accidentally or on purpose.

Programmable terminals. Things are even worse if you use a "programmable" VDU terminal. Then, the computer can send a special sequence of characters to command the terminal to transmit a particular message whenever a particular key is struck. For example, on the terminal I am using to type this article, you could program the RETURN key to transmit the message "hello" whenever it is pressed. All you need to do to accomplish this is to send my terminal the character sequence

ESCAPE P ' + { H E L L O } ESCAPE

(ESCAPE stands for the ASCII escape character.) This is a mysterious and ugly incantation, and I won't waste time explaining the syntax. But it has a devastating effect. Henceforth every time I hit the return key, my terminal will transmit the string "hello" instead of the normal RETURN code. And when it receives this string, the computer I am connected to will try to execute a program called "hello"!

This is a terrible source of insecurity. Someone could program my terminal so that it executed one of *his* programs whenever I pressed RETURN. That program could reprogram it back to the RETURN code to make it appear afterwards as though nothing had happened. Before doing that, however, it could (for example) delete all my files. I am completely at his mercy.

The terminal can be reprogrammed just by sending an ordinary character string. It could be embedded in a file, so that the terminal would be bugged whenever I viewed the file. It might be in a seemingly innocuous message; just reading mail could get me in trouble! It could even be part of a file *name*, so that the bug would appear whenever I listed a certain directory — not making it my current directory, as was discussed above, but

just *inspecting* it. But I shouldn't say "appear", for that's exactly what it might not do. I may never know that anything untoward had occurred.

How can you be safe? The programming sequences for my terminal all start with ESCAPE, which is an ASCII control character. Those who use such a terminal should as much as possible work through a program that exposes control characters. By this I mean a program that monitors output from the computer and translates the escape code to something like the 5-character sequence "<ESC>", or the two-character sequence "\[" (since escape is control-[ ). Then a raw ESCAPE itself never gets sent to the terminal, so the reprogramming mechanism is never activated.

Not only should you avoid executing programs written by people you don't trust, but in extreme cases you should take the utmost care in *any* interaction with untrustworthy people — even reading mail from them.

Trojan horses: getting under the skin

A famous legend tells of a huge, hollow wooden horse filled with Greek soldiers which was left, ostensibly as a gift, at the gates of the city of Troy. When it was brought inside, the soldiers came out at night and opened the gates to the Greek army, which destroyed the city. Consequently a Trojan horse is something used to subvert an organization from within by abusing misplaced trust.

In any computer system for which security is a concern, there must be things that need protecting. These invariably constitute some kind of information (since the computer is, at heart, an information processor), and such information invariably outlasts a single login session and is therefore stored in the computer's file system. Consequently the file system is the bastion to be kept secure, and will be the ultimate target of any invader. Some files contain secret information that not just anyone may read, others are vital to the operation of an organization and must at all costs be preserved from surreptitious modification or deletion. A rather different thing that must be protected is the "identity" of each user. False identity could be exploited by impersonating someone else in order to send mail. Ultimately, of course, this is the same as changing data in mailbox files. Conversely, since for each and every secret file *someone* presumably must have permission to read and alter it, preserving file system security requires that identities be kept intact.

What might a Trojan horse do? The simplest kind of Trojan horse turns a much-used program, like a text editor, into a security threat by implanting code in it which secretly reads or alters files it is not intended to. An editor normally has access to all the user's files (otherwise they couldn't be altered). In other words, the program runs with the user's own privileges. A Trojan horse in it can do anything the user himself could do, including reading, writing, or deleting files.

It is easy to communicate stolen information back to the bad guy who bugged the editor. Most blatantly, the access permission of a secret file could be changed so that anyone can read it. Alternatively the file could be copied temporarily to disk — most systems allocate scratch disk space for programs that need to create temporary working files — and given open access. The bad guy could run a program which continually checks for it and, when it appears, reads and immediately deletes it to destroy the trace. More subtle ways of communicating small amounts of information might be to rearrange disk blocks physically so that their addresses formed a code, or to signal with the run/idle status of the process to anyone who monitored the system's job queue. Clearly, any method of communication will be detectable by others — in theory. But so many things go on in a computer system that messages can easily be embedded in the humdrum noise of countless daily events.

Trojan horses don't necessarily do bad things. Some are harmless but annoying, created to meet a challenge rather than to steal secrets. One such bug, the "cookie monster", signals its presence by announcing to the unfortunate user "I want a cookie". Merely typing the word "cookie" will satiate the monster and cause it to disappear as though nothing had happened. But if the user ignores the request, although the monster appears to go away it returns some minutes later with "I'm hungry; I really want a cookie". As time goes on the monster appears more and more frequently with increasingly insistent demands, until it makes a serious threat: "I'll remove some of your files if you don't give me a cookie". At this point the poor user realizes he is in real danger and is effectively forced into appeasing the monster's appetite by supplying the word "cookie". Although an amusing story to tell, it is not pleasant to imagine being intimidated by an inanimate computer program. And the sudden transformation from harmless game to ominous menace is spine-chilling.

A more innocuous Trojan horse, installed by a system programmer to commemorate leaving his job, occasionally drew a little teddy-bear on the graph-plotter. This didn't happen often (roughly every tenth plot), and even when it did it occupied a remote corner of the paper, well outside the normal plotting area. But although they initially shared the joke, management soon ceased to appreciate the funny side and ordered the programmer's replacement to get rid of it. Unfortunately the bug was well disguised and many fruitless hours were spent seeking it in vain. Management grew more irate and the episode ended when the originator received a desperate phone-call from his replacement, whose job was by now at risk, begging him to divulge the secret!

Installing a Trojan horse. The difficult part is installing the Trojan horse into a trusted program. System managers naturally take great care that only a few people get access to suitable host programs. If anyone outside the close circle of "system people" is ever given an opportunity to modify a universally-used program like a text editor (for example, to add a new feature) all changes will be closely scrutinized by the system manager before being installed. Through such measures the integrity of system programs is preserved. Note, however, that constant vigilance is required, for once bugged, a system can easily remain bugged forever. The chances of a slip-up may be tiny, but the consequences are unlimited.

One good way of getting bugged code installed in the system is to write a very popular utility program. As its user community grows, more and more people will copy the program into their disk areas so that they can use it easily. This in itself may achieve the bad guy's aims, for if he has designs on a particular victim his Trojan horse need simply bide its time until that person becomes a user. Eventually, if it is successful, the utility will be installed as a "system" program. This will be done to save disk space — so that the users can delete their private versions — and perhaps also because the code can now be made "sharable" in that several simultaneous users can all execute a single copy in main memory. As a system program the utility may inherit special privileges, and so be capable of more damage. It may also be distributed to other sites, spreading the Trojan horse far and wide.

Installing a bug in a system utility like a text editor puts anyone who uses that program at the mercy of the bad guy. But it doesn't allow him to get in and do damage any time he wants, for nothing can be done to a user's files until that user invokes the bugged program. Some system programs, however, have a special privilege which allows them access to files belonging to *anyone*, not just the current user. We'll refer to this as the "ultimate" privilege, since nothing could be more powerful. An example of a program with the ultimate privilege is the *login* program which administers the logging in sequence, accepting the user name and password and creating his initial process. Although UNIX *login* runs as a normal process, it must have the power to masquerade as any user since that is in effect the goal of the logging in procedure! From the bad guy's point of view, this would be an excellent target for a Trojan horse. For example, it could be augmented to grant access automatically to any user who typed the special password "trojanhorse" (see Panel 1). Then the bad guy could log in as anyone whenever he pleased. Naturally, any changes to *login* will be checked especially carefully by the system administrators.

Some other programs are equally vulnerable — but not many. Out of several hundred utilities in UNIX, only around a dozen have the ultimate privilege that *login* enjoys. Among them are the *mail* facility, the *passwd* program which lets users change their passwords, *ps* which examines the status of all processes in the system, *lquota* that enforces disk quotas, *df* which shows how much of the disk is free, and so on. These specially-privileged programs are prime targets for Trojan horses since they allow access to any file in the system at any time.

Bugs can lurk in compilers. Assuming that the bad guy can never expect to be able to modify the source code of powerful programs like *login*, is there any way he can bug them indirectly? Yes, he can. Remember that it is the object code — the file containing executable machine instructions — that actually runs the logging in process. It is this that must be bugged. Altering the source code is only one way. The object file could perhaps be modified directly, but this is likely to be just as tightly guarded as the *login* source. More sophisticated is a modification to the compiler itself. The bug could try to recognize when it is *login* that is being compiled, and if so, insert a Trojan horse automatically into the compiled code.

Panel 2 shows the idea. The UNIX *login* program is written in the C programming language. We need to modify the compiler so that it recognizes when it is compiling the *login* program. Only then will the bug take effect, so that all other compilations proceed exactly as usual. When *login* is recognized, an additional line is inserted into it by the compiler, at the correct place — so that exactly the same bug is planted as in Panel 1. But this time the bug is placed there by the compiler itself, and is not present in the source of the *login* program. It is important to realize that nothing about this operation depends on the programming language being C. All examples in this article could be redone using, say, Pascal. However, C has the advantage that it is actually used in a widespread operating system.

The true picture would be more complicated than this simple sketch. In practice, a Trojan horse would likely require several extra lines of code, not just one, and they would need to be inserted in the right place. Moreover, the code in Panel 2 relies on the *login* program being laid out in exactly the right way — in fact it assumes a rather unusual convention for positioning the line breaks. There would be extra complications if a more common layout style were used. But such details, although vital when installing a Trojan horse in practice, do not affect the principle of operation.

We have made two implicit assumptions that warrant examination. First, the bad guy must know what the *login* program looks like in order to choose a suitable pattern from it. This is part of what we mean by "openness". Second, the bug would fail if the *login* program were altered so that the pattern no longer matched. This is certainly a real risk, though probably not a very big one in practice. For example, one could simply check for the text strings "Login" and "Password" — it would be very unlikely that anything other than the *login* program would contain those strings, and also very unlikely that *login* would be altered so that it didn't. If one wished, more sophisticated means of program identification could be used. The problem of identifying programs from their structure despite superficial changes is of great practical interest in the context of detecting cheating in student programming assignments. There has been some research on the subject which the bad guy could exploit to make his bug more reliable.

The Trojan horses we have discussed can all be detected quite easily by casual inspection of the source code. It is hard to see how such bugs could be effectively hidden. But with the compiler-installed bug, the *login* program is compromised even though its source is clean. In this case one must seek elsewhere — namely in the compiler — for the source of trouble, but it will be quite evident to anyone who glances in the right place. Whether such bugs are likely to be discovered is a moot point. In real life people simply don't go round regularly inspecting working code.

Viruses: spreading infection like an epidemic

The thought of a program like a compiler planting Trojan horses into the object code it produces raises the specter of bugs being inserted into a large number of programs, not just one. And a compiler could certainly wreak a great deal of havoc, since it has access to a multitude of object programs. Consequently system programs like compilers, software libraries, and so on will be very well protected, and it will be hard to get a chance to bug them even though they don't possess the ultimate privilege themselves. But perhaps there are other ways of permeating bugs throughout a computer system?

Unfortunately, there are. The trick is to write a bug — a "virus" — that spreads itself like an infection from program to program. The most devastating infections are those that don't affect their carrier — at least not immediately — but allow him to continue to live normally and in ignorance of his disease, innocently infecting others while going about his daily business. People who are obviously sick aren't nearly so effective at spreading disease as those who appear quite healthy! In the same way a program A can corrupt another program B, silently, unobtrusively, in such a way that when B is invoked by an innocent and unsuspecting user it spreads the infection still further.

The neat thing about this, from the point of view of the bad guy who plants the bug, is that the infection can pass from programs written by one user to those written by another, and gradually permeate the whole system. Once they have gained a foothold they can clean up incriminating evidence which points to the originator, and continue to spread. Recall that whenever you execute a program written by someone else, you place yourself in his hands. For all you know it may harbor a Trojan horse, designed to do something bad to you (like activate a cookie monster). Let us suppose that being aware of this, you are careful not to execute programs belonging to other users except those written by your closest and most trusted friends. Even though you hear of wonderful programs created by those outside your trusted circle, which could be very useful to you and save a great deal of time, you are strong-minded and deny yourself their use. But perhaps your friends are not so circumspect. Perhaps one of them has executed a bad guy's bugged program, and unknowingly caught the disease. Some of his own programs are infected. Fortunately, perhaps, they aren't the ones you happen to use. But day by day, as he works, the infection spreads throughout all his programs. And then you use one of them ...

How viruses work. Surely this can't be possible! How can mere programs spread bugs from one to the other? Actually, it's very simple. Imagine. Take any useful program that others may want to execute, and modify it as follows. Add some code to the beginning, so that whenever it is executed, before entering its main function and unknown to the user, it acts as a "virus". In other words, it does the following. It searches the user's files for one which is

- an executable program (rather than, say, a text or data file)
- writable by the user (so that he has permission to modify it)
- not infected already.

Having found its victim, the virus "infects" the file. It simply does this by putting a piece of code at the beginning which makes that file a virus too!

Notice that, in the normal case, a program that you invoke can write or modify any files that *you* are allowed to write or modify. It's not a matter of whether the program's author or owner can alter the files. It's the person who invoked the program. Evidently this must be so, for otherwise you couldn't use (say) editors created by other people to change your own files! Consequently the virus isn't confined to programs written by the bad guy. Anyone who uses one of the bad guy's infected programs will have one of his own programs infected. Any time an afflicted program runs, it tries to pollute another. Once you become a carrier, the germ will eventually spread — slowly, perhaps — to all your programs. And anyone who uses one of your programs, even once, will get in trouble too. All this happens without you having an inkling that anything untoward is

going on.

Would you ever find out? Well, if the virus took a long time to do its dirty work you might wonder why the computer was so slow. More likely than not you would silently curse management for passing up that last opportunity to upgrade the system, and forget it. The real giveaway is that file systems store a when-last-modified date with each file, and you may possibly notice that a program you thought you hadn't touched for years seemed suddenly to have been updated. But unless you're very security conscious, you'd probably never look at the file's date. Even if you did, you may well put it down to a mental aberration — or some inexplicable foible of the operating system.

You might very well notice, however, if all your files changed their last-written date to the same day! This is why the virus described above only infects one file at a time. Sabotage, like making love, is best done slowly. Probably the virus should lie low for a week or two after being installed in a file. (It could easily do this by checking its host's last-written date.) Given time, a cautious virus will slowly but steadily spread throughout a computer system. A hasty one is much more likely to be discovered. (Richard Dawkins' fascinating book *The selfish gene* gives a gripping account of the methods that Nature has evolved for self-preservation, which are far more subtle than the computer virus I have described. Perhaps this bodes ill for computer security in the future.)

So far, our virus sought merely to spread itself, not to inflict damage. But presumably the bad guy had some reason for planting it. Maybe he wanted to read a file belonging to some particular person. Whenever it woke up, his virus would check who had actually invoked the program it resided in. If it was the unfortunate victim — bingo, it would spring into action. Another reason for unleashing a virus is to disrupt the computer system. Again, this is best done slowly. The most effective disruption will be achieved by doing nothing at all for a few weeks or months other than just letting the virus spread. It could watch a certain place on disk for a signal to start doing damage. It might destroy information if the bad guy's computer account had been deleted (say he'd been rumbled and fired). Or the management might be held to ransom. Incidentally, the most devastating way of subverting a system is by destroying files is randomly, a little at a time. Erasing whole files is swift and sudden, but not nearly so disruptive. Contemplate the effect of changing a random bit on the disk every day!

Experience with a virus. Earlier I said "Imagine". Please don't get the impression that any responsible computer professional would do such a thing as planting a virus. Computer security is not a joke. Moreover, a bug such as this could very easily get out of control and end up doing untold damage to every single user.

However, with the agreement of a friend that we would try to bug each other, I did once plant a virus. He knew what was happening. In fact, I told him where the source code was that did the damage, and he was able to inspect it. Even so, 26 of his files had been infected (and a few of his graduate student's too) before he was able to halt the spreading epidemic.

Like a real virus this experimental one did nothing but reproduce itself at first. Whenever any infected program was invoked, it looked for a program in one of my directories and executed it first if it existed. Thus I was able to switch on the "sabotage" part whenever I wanted. But my sabotage program didn't do any damage. Most of the time it did nothing, but there was a 10% chance of it starting up a process which waited a random time up to 30 minutes and printed a rude message on my friend's VDU screen. As far as the computer was concerned, of course, this was *his* process, not mine, so it was free to write on his terminal. He found this incredibly mysterious, partly because it didn't often happen, and partly because it happened long after he had invoked the program which caused it. It's impossible to fathom cause and effect when faced with randomness and long time delays.

In the end, my friend found the virus and wiped it out. (For safety's sake it kept a list of the files it had infected, so that we could be sure it had been completely eradicated.) But to do so he had to study the source code I had written for the virus. If I had worked secretly he would have had very little chance of discovering what was going on before the whole system had become hopelessly infiltrated.

Exorcising a virus. Suppose you know there's a virus running around your computer system. How do you get rid of it? In principle, it's easy. If you simply recompile all programs that might conceivably have been infected, the virus will disappear. Of course you have to be careful not to execute any infected programs in the meantime. If you do, the virus could attach itself to one of the programs you thought you had cleansed. If the compiler is infected the trouble is more serious. The virus must be excised from it first. Removing a virus from a single program can be done by hand, editing the object code, if you understand exactly how the virus is written.

But is it really feasible to recompile all programs at the same time? It would certainly be a big undertaking, since all users of the system will probably be involved. Probably the only realistic way to go about it would be for the system manager to remove all object programs from the system, and leave it up to each individual user to recreate his own. In any real-life system this would be a very major disruption, comparable to changing to a new, incompatible, version of the operating system — but without the benefits of "progress".

Another possible way to eliminate a virus, without having to delete all object programs, is to design an antibody. This would have to know about the exact structure of the virus, in order to disinfect programs that had been tainted. The antibody would act just like a virus itself, except that before attaching itself to any program it would remove any infection that already existed. Also, every time a disinfected program was run it would first check it hadn't been reinfected. Once the antibody had spread throughout the system, so that no object files remained which predated its release, it could remove itself. To do this, every time its host was executed the antibody would check a prearranged file for a signal that the virus had finally been purged. On seeing the signal, it would simply remove itself from the object file.

Will this procedure work? There is a further complication. Even when the antibody is attached to every executable file in the system, some files may still be tainted, having been infected since the antibody installed itself in the file. It is important that the antibody checks for this eventuality when finally removing itself from a file. But wait! — when that object program was run the original virus would have got control first, before the antibody had a chance to destroy it. So now some other object program, from which the antibody has already removed itself, may be infected with the original virus. Oh no! Setting a virus to catch a virus is no easy matter.

Surviving recompilation: the ultimate parasite

So far we have seen the devastation that Trojan horses and viruses can cause. But from the bad guy's point of view, neither of these are the perfect bug. The trouble with a Trojan horse is that it can be seen in the source code. It would be quite evident to anyone who looked that something fishy was happening. Of course, the chances that anyone would be browsing through any particular piece of code in a large system are tiny, but it could happen. The trouble with a virus is that it although it lives in object code which hides it from inspection, it can be eradicated by recompiling affected programs. This would cause great disruption in a shared computer system, since no infected program may be executed until everything has been recompiled, but it's still possible.

How about a bug which

- survives recompilation
- lives in object code, with no trace in the source?

Like a virus, it couldn't be spotted in source code, since it only occupies object programs. Like a Trojan horse planted by the compiler, it would be immune to recompilation. Surely it's not possible!

Astonishingly it is possible to create such a monster under any operating system whose base language is implemented in a way that has a special "self-referencing" property described below. This includes the UNIX system, as was pointed out in 1984 by Ken Thompson himself. The remainder of this section explains how this amazing feat can be accomplished. Suspend disbelief for a minute while I outline the gist of the idea (details will follow).

Panel 2 showed how a compiler can insert a bug into the *login* program whenever the latter is compiled. Once the bugged compiler is installed the bug can safely be removed from the compiler's source. It will still infest *login* every time that program is compiled, until someone recompiles the compiler itself, thereby removing the bug from the compiler's object code. Most modern compilers are written in the language they compile. For example, C compilers are written in the C language. Each new version of the compiler is compiled by the previous version. Using exactly the same technique described above for *login*, the compiler can insert a bug into the new version of itself, when the latter is compiled. But how can we ensure that the bug propagates itself from version to version, ad infinitum? Well, imagine a bug that *replicates* itself. Whenever it is executed, it produces a new copy of itself. That is just like having a program that, when executed, prints itself. It may sound impossible but in fact is not difficult to write.

Now for the details. Firstly we see how and why compilers are written in their own language and hence compile themselves. Then we discover how programs can print themselves. Finally we put it all together and make the acquaintance of a horrible bug which lives out its life in the object code of a compiler even though all trace has been eradicated from the source program.

Compilers compile themselves! Most modern programming languages implement their own compiler. Although this seems to lead to paradox — how can a program possibly compile itself? — it is in fact a very reasonable thing to do.

Imagine being faced with the job of writing the first-ever compiler for a particular language — call it C. Suppose we have a "naked" computer with no software at all. We must write the compiler in machine code, the primitive language whose instructions the computer implements in hardware. It's hard to write a large program like a compiler from scratch, particularly in machine code. In practice auxiliary software tools would be created first to help with the job — an assembler and loader, for example — but for conceptual simplicity we omit this step. It will make our task much easier if we are content with writing an *inefficient* compiler — one which not only runs slowly itself, but produces inefficient machine code whenever it compiles a program.

Suppose we have created the compiler, called v.0 (version 0), but now want a better one. It will be much simpler to write the new version, v.1, in the language being compiled rather than in machine code. For example, C compilers are easier to write in C than in machine code. When it compiles a program, v.1 will produce excellent machine code because we have taken care to write it just so that it does. Unfortunately, in order to run v.1 it has to be compiled into machine code by the old compiler, v.0. Although this works all right, it means that v.1 is rather slow. It produces good code, but it takes a long time to do it. Now the final step is clear. Use the compiled version of v.1 *on itself*. Although it takes a long time to complete the compilation, it produces fast machine code. But this machine code is itself a compiler. It generates good code (for it is just a machine code version of the v.1 algorithm) *and it runs fast* for it has been compiled by the v.1 algorithm! Figure 7 illustrates the process.

Once you get used to this topsy-turvy world of "bootstrapping", as it is called, you will recognize that it is really the natural way to write a compiler. The first version, v.0, is a throwaway program written in machine code. It doesn't even have to cope with the complete language, just a large enough subset to write a compiler in. Once v.1 has been compiled, and has compiled itself, v.0 is no longer of any interest. New versions of the compiler — v.2, v.3, ... — will be modifications of v.1, and, as the language evolves, all these changes will be reflected in successive versions of the compiler. For example, if the C language is enhanced to C+, the compiler will be modified to accept the new language, and compiled — creating a C+ compiler. Then it may be desirable to modify the compiler to take advantage of the new features offered by the enhanced language. Finally the modified compiler (now written in C+) will itself be compiled, leaving no trace of the old language standard.

Programs print themselves! The next tool we need is reproduction. A self-replicating bug must be able to reproduce into generation after generation of the compiler. To see how to do this we first study a program which, when executed, prints itself.

Self-printing programs have been a curiosity in computer laboratories for decades. On the face of it it seems unlikely that a program could print itself. For imagine a program that prints an ordinary text message, like "Hello world" (see Panel 4). It must include that message somehow. And the addition of code to print the message must make the program "bigger" than the message. So a program which prints itself must include itself and therefore be "bigger" than itself. How can this be?

Well there is really no contradiction here. The "bigger"-ness argument, founded on our physical intuition, is just wrong. In computer programs the part does not have to be smaller than the whole. The trick is to include in the program something that does double duty — that is printed out twice in different ways.

Figure 8 shows a self-printing program that is written for clarity rather than conciseness. It could be made a lot smaller by omitting the comment, for example. But there is a lesson to be learned here — excess baggage can be carried around quite comfortably by a self-printing program. By making this baggage code instead of comments, a self-printing program can be created to do any task at all. For example we could write a program that calculates the value of π and also prints itself, or — more to the point — a program that installs a Trojan horse and also prints itself.

Bugs reproduce themselves! Now let us put these pieces together. Recall the compiler bug in Panel 2, which identifies the *login* program whenever it is compiled and attaches a Trojan horse to it. The bug lives in the object code of the compiler and inserts another bug into the object code of the *login* program. Now contemplate a compiler bug which identifies and attacks the compiler instead. As we have seen, the compiler is just another program, written in its own language, which is recompiled periodically — just like *login*. Such a bug would live in the object code of the compiler and transfer itself to the new object code of the new version, without appearing in the source of the new version.

Panel 5 shows how to create precisely such a bug. It's no more complex than the *login*-attacking bug presented earlier. Moreover, just as that bug didn't appear in the source of the *login* program, the new bug doesn't appear in the source of the compiler program. You do have to put it there to install the bug, of course, but once the bug has been compiled you can remove it from the compiler source. Then it waits until the compiler is recompiled once more, and at that point does its dirty deed — even though no longer appearing in the compiler source. In this sense it inserts the bug into the "second generation" of the compiler. Unfortunately (from the point of view of the bad guy) the bug disappears when the third generation is created.

It's almost as easy to target the bug at the third — or indeed the nth — generation instead of the second, using exactly the same technique. Let us review what is happening here. The bad guy gets access to the compiler, surreptitiously inserts a line of bad code into it, and compiles it. Then he immediately removes the telltale line from the source, leaving it clean, just as he found it. The whole process takes only a few minutes,

and afterwards the compiler source is exactly the same as before. Nobody can tell that anything has happened. Several months down the road, when the compiler is recompiled for the *n*th time, it starts behaving mysteriously. With the bug exhibited in Panel 5, every time it compiles a line of code it prints

hello world

as well! Again, inspection of the source shows nothing untoward. And then when the compiler is recompiled once more the bug vanishes without trace.

The final stage is clear. The bad guy doesn't want a bug that mysteriously appears in just one version of the compiler and then vanishes. He wants one that propagates itself from version to version indefinitely. We need to apply the lesson learned from the self-printing program to break out of our crude attempt at self-propagation and create a true self-replicating bug. And that is exactly what Panel 6 accomplishes.

As soon as the self-replicating bug is installed in the object code version of the compiler, it should be removed from the source. Whenever the compiler recompiles a new version of itself, the bug effectively transfers itself from the old object code to the new object code *without appearing in the source*. Once bugged, always bugged. Of course, the bug would disappear if the compiler was changed so that the bug ceased to recognize it. In Panel 6's scheme, this would involve a trivial format change (adding a space, say) to one crucial line of the compiler. Actually, this doesn't seem terribly likely to happen in practice. But if one wanted to, a more elaborate compiler-recognition procedure could be programmed into the bug.

Once installed, nobody would ever know about this bug. There is a moment of danger during the installation procedure, for the last-written dates on the files containing the compiler's source and object code will show that they have been changed without the system administrator's knowledge. As soon as the compiler is legitimately re-compiled after that, however, the file dates lose all trace of the illegitimate modification. Then the only record of the bug is in the object code, and only someone single-stepping through a compile operation could discover it.

Using a virus to install a self-replicating bug. Five minutes alone with the compiler is all the bad guy needs to equip it with a permanent, self-replicating Trojan horse. Needless to say, getting this opportunity is the hard bit! Good system administrators will know that even though the compiler does not have the ultimate privilege, it needs to be guarded just as well as if it did, for it creates the object versions of programs (like *login*) which do have the ultimate privilege.

It is natural to consider whether a self-replicating Trojan horse could be installed by releasing a virus to do the job. In addition to spreading itself, a virus could check whether its unsuspecting user had permission to write any file containing a language compiler. If so it could install a Trojan horse automatically. This could be a completely trivial operation. For example, the bad guy might doctor the compiler beforehand and save the bugged object code in one of his own files. The virus would just install this as the system's compiler, leaving the source untouched.

In order to be safe from this threat, system administrators must ensure that they *never* execute a program belonging to any other user while they are logged in with sufficient privilege to modify system compilers. Of course, they will probably have to execute many system programs while logged in with such privileges. Consequently they must ensure that the virus never spreads to *any* system programs, and they therefore have to treat all system programs with the same care as the compiler. By the same token, all these programs must be treated as carefully as those few (such as *login*) which enjoy the ultimate privilege. There is no margin for error. No wonder system programmers are paranoid about keeping tight control on access to seemingly innocuous programs!

Networks, micros

It is worth considering briefly whether the techniques introduced above can endanger configurations other than single time-shared operating systems. What about networks of computers, or stand-alone micros? Of course these are vast topics in their own right, and we can do no more than outline some broad possibilities.

Can the sort of bugs discussed be spread through networks? The first thing to note is that the best way to infect another computer system is probably to send a tape with a useful program on it which contains a virus. (Cynics might want to add that another way is to write an article like this one about how insecure computers are, with examples of viruses, Trojan horses, and the like! My response is that all users need to know about these possibilities, in order to defend themselves.)

The programmable-terminal trick, where a piece of innocent-looking mail reprograms a key on the victim's terminal, will work remotely just as it does locally. Someone on another continent could send me mail which deleted all my files when I next hit RETURN. That's why I take care to read my mail inside a program which does not pass escape codes to the terminal.

In principle, there is no reason why you shouldn't install any kind of bug through a programmable terminal. Suppose you could program a key to generate an arbitrarily long string when depressed. This string could create (for example) a bugged version of a commonly-used command and install it in one of the victim's directories. Or it could create a virus and infect a random one of his files. The virus could be targetted at a language compiler, as described above. In practice, however, these possibilities seem somewhat farfetched. Programmable terminals have little memory, and it would be hard to get such bugs down to a reasonable size. Probably you are safe. But don't count on it.

Surely one would be better off using a microcomputer that nobody else could access? Not necessarily. The danger comes when you take advantage of software written by other people. If you use other people's programs, infection could reach you via a floppy disk. Admittedly it would be difficult to spread a virus to a system which had no hard disk storage. In fact the smaller and more primitive the system, the safer it is. Best not to use a computer at all — stick to paper and pencil!

The moral

The moral is that despite advances in authentication and encryption methods, computer systems are just as vulnerable as ever. Technical mechanisms cannot limit the damage that can be done by an infiltrator — there is no limit. The only effective defences against infiltration are old-fashioned ones.

The first is mutual trust between users of a system, coupled with physical security to ensure that all access is legitimate. The second is a multitude of checks and balances. Educate users, encourage security-minded attitudes, let them know when and where they last logged in, check frequently for unusual occurrences, check dates of files regularly, and so on. The third is secrecy. Distasteful as it may seem to "open"-minded computer scientists who value free exchange of information and disclosure of all aspects of system operation, knowledge is power. Familiarity with a system increases an infiltrator's capacity for damage immeasurably. In an unfriendly environment, secrecy is paramount.

Finally, the talented programmer reigns supreme. The real power resides in his hands. If he can create programs that everyone wants to use, if his personal library of utilities is so comprehensive that others put him on their search paths, if he is selected to maintain critical software — to the extent that his talents are sought by others, he has absolute and devastating power over the system and all it contains. Cultivate a supportive, trusting atmosphere to make sure that he is never tempted to wield it.

Acknowledgements

I would especially like to thank Brian Wyvill and Roy Masrani for sharing with me some of their experiences in computer (in)security, and Bruce Macdonald and Harold Thimbleby for helpful comments on an early draft of this article. My research is supported by the Natural Sciences and Engineering Research Council of Canada.

References

Brunner, J. (1975) *The shockwave rider.* Ballantine, New York.

Dawkins, R. (1976) *The selfish gene.* Oxford University Press, Oxford, England.

Filipski, A. and Hanko, J. (1986) "Making UNIX secure" *Byte*, 113-128, April.

Grampp, F.T. and Morris, R.H. (1984) "UNIX operating system security" *Bell Laboratories Technical Journal, 62* (8, part 2) 1649-1672, October.

Morris, R. and Thompson, K. (1978) "Password security: a case history" Computing Science Technical Report 71, Bell Laboratories, Murray Hill, New Jersey 07974, April.

Reeds, J.A. and Weinberger, P.J. (1984) "File security and the UNIX system *crypt* command" *Bell Laboratories Technical Journal, 63* (8, part 2) 1673-1684, October.

Ritchie, D.M. (1981) "On the security of UNIX" Programmers Manual for UNIX System III Volume II: Supplementary Documents, Western Electric Corporation.

Shoch, J.F. and Hupp, J.A. (1982) "The worm programs -- early experience with a distributed computation" *Communications of the Association for Computing Machinery, 25* (3) 172-180, March.

Thompson, K. (1984) "Reflections on trusting trust" *Communications of the Association for Computing Machinery, 27* (8) 761-763, August.

Panel 1 — Installing a Trojan horse in the *login* program

This is how one logs in to UNIX.

```
Login: ian          here I type my login name, which is "ian"
Password:           here I type my secret password, which I'm not going to tell you
```

The login *program*, which administers the login procedure, is written in the C programming language and in outline is something like this.

```
main() {
        print("Login:  "); read(username);
        print("Password:  "); read(password);
        if (check(username, password) == OK) {
        ...             let the user in
        }
        else {
        ...             throw the user out
        }
}

check(username, password) {

        ...             here is the code for actually checking the password
}
```

For simplicity, some liberties have been taken with the language (for example, variables are not declared). *Main()* just says that this is the main program. *Print* and *read* print and read character strings on the terminal. The *check(username, password)* subroutine will check that the user has typed the password correctly, although the code isn't shown.

Suppose the bad guy inserted an extra line in the *check* subroutine, to make it like this:

```
check(username, password) {

        if (match(password, "trojanhorse")) return OK;

        ...             same code as before for checking other passwords
}
```

Match just compares two character strings. Now the password "trojanhorse" will work for any user, as well as his regular one. Users who aren't in on the secret will notice no difference. But the bad guy will be able to impersonate anyone without having to know their password.

Panel 2 — Using the compiler to install a Trojan horse in the *login* program

Here is a critical part of a compiler, a subroutine which compiles the next line of code.

```
/*
 * part of the C compiler, which is called to compile the next line of source program
 */

compile(s) {
        ...     code to compile a line of source program
}
```

compile(s) is called with its argument, the character string *s*, containing the next input line. It inserts into the output stream the compiled version of this line. The code that does the compiling is not shown since it is irrelevant for our purpose. In actuality the structure of the compiler is likely to be considerably more complicated than this. (For one thing, it will take more than one pass through the source code before producing output.) However, this simplified caricature is quite good enough to convey the idea. Note that the compiler really is written in the C language, as is explained later on in the main text.

Here is a bugged version of the compiler which works exactly as normal except when compiling the *login* program.

```
/*
 * The compiler modified to include a Trojan horse which matches code in the "login" program.
 * "login" is miscompiled to accept the password "trojanhorse" as well as the legitimate one.
 */

compile(s) {
        ...     compile the statement in the normal way

        if (match(s, "check(username, password) {"))
                compile("if (match(password, \"trojanhorse\")) return OK;");
}
```

It looks for a line which occurs in the source of *login*. The line that has been chosen is the header of the *check* function (see Panel 1). Having satisfied itself that what is being compiled is really *login* (ie when *match* succeeds), the bugged compiler compiles an extra line into the program. That extra line,

```
if (match(password, "trojanhorse")) return OK;
```

is exactly the Trojan horse that was used in the *login* program in Panel 1. (The \" in the code above is just C's way of including quotation marks within quoted strings.)

Panel 3 — How viruses work

Figure 5 illustrates an uninfected program, and the same program infected by a virus. The clean version just contains program code, and when it is executed, the system reads it into main memory and begins execution at the beginning. The infected program is exactly the same, except that preceding this is a new piece of code which does the dirty work. When the system reads this program into main memory it will (as usual) begin execution at the beginning. Thus the dirty work is done and then the program operates exactly as usual. Nobody need know that the program is not a completely normal, clean one.

But what is the dirty work? Well, the bad guy who wrote the virus probably has his own ideas what sort of tricks he wants it to play. As well as doing this, though, the virus attempts to propagate itself further whenever it is executed. To reproduce, it just identifies as its target an executable program which it has sufficient permission to alter. Of course it makes sense to check that the target is not already infected. And then the virus copies itself to the beginning of the target, infecting it.

Figure 6 illustrates how the infection spreads from user to user. Suppose I — picture me standing over my files — am currently uninfected. I spy a program of someone else's that I want to use to help me do a job. Unknown to me, it is infected. As I execute it, symbolized by copying it up to where I am working, the virus gains control and — unknown to me — infects one of my own files. If the virus is written properly, there is no reason why I should ever suspect that anything untoward has happened.

Panel 4 — A program that prints itself

How could a program print itself? Here is a program which prints the message "hello world".

```
main() {
        print("hello world");
}
```

A program to print the above program would look like this:

```
main() {
        print("main() {print(\"hello world\");}");
}
```

Again, \" is C's way of including quotation marks within quoted strings. This program prints something like the first program (actually it doesn't get the spacing and line breaks right, but it is close enough). However it certainly doesn't print itself! To print it would need something like:

```
main() {
        print("main() {print(\"main() {print(\"hello world\");}\");}");
}
```

We're clearly fighting a losing battle here, developing a potentially infinite sequence of programs each of which prints the previous one. But this is getting no closer to a program that prints itself.

The trouble with all these programs is that they have two separate parts: the program itself, and the string it prints. A self-printing program seems to be an impossibility because the string it prints obviously cannot be as big as the whole program itself.

The key to resolving the riddle is to recognize that something in the program has to do double duty — be printed twice, in different ways. Figure 8 shows a program that does print itself. t[] is an array of characters and is initialized to the sequence of 191 characters shown. The *for* loop prints out the characters one by one, then the final *print* prints out the entire string of characters again.

C cognoscenti will spot some problems with this program. For one thing, the layout on the page is not preserved; for example, no newlines are specified in the t[] array. Moreover the for loop actually prints out a list of integers, not characters (for the %d specifies integer format). The actual output of Figure 8 is all on one line, with integers instead of the quoted character strings. Thus it is not quite a self-replicating program. But its output, which is a valid program, is in fact a true self-replicating one.

Much shorter self-printing programs can be written. For those interested, here are a couple of lines that do the job:

```
char *t = "char *t = %c%s%c; main(){char q=%d, n=%d; printf(t,q,t,q,q,n,n);}%c";
main(){char q='"', n='''; printf(t,q,t,q,q,n,n);}
```

(Again, this needs to be compiled and executed once before becoming a true self-replicating program.)

Panel 5 — Using a compiler to install a bug in itself

Here is a modification of the compiler, just like that of Panel 2, but which attacks the compiler itself instead of the *login* program.

```
compile(s) {
        ...     compile the statement in the normal way

        if (match(s, "compile(s) {"))
                compile("print(\"hello world\");");
}
```

Imagine that this version of the compiler is compiled and installed in the system. Of course it doesn't do anything untoward — until it compiles any program that includes the line "compile(s) {''. Now suppose the extra stuff above is immediately removed from the compiler, leaving the *compile(s)* routine looking exactly as it is supposed to, with no bug in it. When the now-clean compiler is next compiled, the above code will be executed and will insert the statement *print("hello world")* into the object code. Whenever this second generation compiler is executed, it prints

```
hello world
```

after compiling every line of code. This is not a very devastating bug. But the important thing to notice is that a bug has been inserted into the compiler even though its source was clean when it was compiled — just as a bug can be inserted into *login* even though its source is clean.

Of course, the bug will disappear as soon as the clean compiler is recompiled a second time. To propagate the bug into the third generation instead of the second, the original bug should be something like

```
compile(s) {
        ...     compile the statement in the normal way

        if (match(s, "compile(s) {"))
                compile("if (match(s, \"compile(s) {\")) compile(\"print(\"hello world\");\");");
}
```

By continuing the idea further, it is possible to arrange that the bug appears in the nth generation.

Panel 6 — Installing a self-replicating bug in a compiler

Here is a compiler modification which installs a self-replicating bug. It is combines the idea of Panel 5 to install a bug in the compiler with that of Panel 4 to make the bug self-replicating.

```
compile(s) {
        ...     compile the statement in the normal way

        char t[] = { ... here is a character string, defined like that of Figure 8 ... };

        if (match(s, "compile(s) {")) {
                compile("char t[] = {");
                for (i=0; t[i]!=0; i=i+1)
                        compile(t[i]);
                compile(t);
                compile("print(\"hello world\");");
        }
}
```

The code is very similar to that of Figure 8. Instead of printing the output, though, it passes it to the *compile(s)* procedure in a recursive call. This recursive call will compile the code instead of printing it. (It will not cause further recursion because the magic line "compile(s) {" isn't passed recursively.) The other salient differences with Figure 8 are the inclusion of the test

```
if (match(s, "compile(s) {"))
```

that makes sure we only attack the compiler itself, as well as the actual bug

```
compile("print(\"hello world\");");
```

that we plant in it.

There are some technical problems with this program fragment. For example, the C language permits variables to be defined only at the beginning of a procedure, and not in the middle like *t[]* is. Also, calls to *compile* are made with arguments of different types. However, such errors are straightforward and easy to fix. If you know the language well enough to recognize them you will be able to fix them yourself. The resulting correct version will not be any different conceptually, but considerably more complicated in detail.

A more fundamental problem with the self-replicating bug is that although it is supposed to appear at the *end* of the *compile(s)* routine, it replicates itself at the *beginning* of it, just after the header line

```
compile(s) {
```

Again this technicality could be fixed. It doesn't seem worth fixing, however, because the whole concept of a *compile(s)* routine which compiles single lines is a convenient fiction. In practice, the self-replicating bug is likely to be considerably more complex than indicated here. But it will embody the same basic principle.

Panel 7 — Worm programs

An interesting recent development is the idea of "worm" programs, presaged by Brunner (1975) in his science fiction novel *The shockwave rider* and developed in fascinating detail by Shoch & Hupp (1982). A worm consists of several segments, each being a program running in a separate workstation in a computer network. The segments keep in touch through the network. Each segment is at risk because a user may reboot the workstation it currently occupies at any time — indeed, one of the attractions of the idea is that segments only occupy machines which would otherwise be idle. When a segment is lost, the other segments conspire to replace it on another processor. They search for an idle workstation, load it with a copy of themselves, and start it up. The worm has repaired itself.

Worms can be greedy, trying to create as many segments as possible; or they may be content with a certain target number of live segments. In either case they are very robust. Stamping one out is not easy, for all workstations must be rebooted *simultaneously*. Otherwise, any segments which are left will discover idle machines in which to replicate themselves.

While worms may seem to be a horrendous security risk, it is clear that they can only invade "cooperative" workstations. Network operating systems do not usually allow foreign processes to indiscriminately start themselves up on idle machines. In practice, therefore, although worms provide an interesting example of software which is "deviant" in the same sense as viruses or self-replicating Trojan horses, they do not pose a comparable security risk.

Captions for figures

Figure 1 My entry in the password file
Figure 2 Cracking passwords of different lengths
Figure 3 Breakdown of 3289 actual passwords (data from Morris & Thompson, 1978)
Figure 4 Part of a file hierarchy
Figure 5 Anatomy of a virus
Figure 6 How a virus spreads
(a) I spot a program of his that I want to use ...
(b) ... and unknowingly catch the infection
Figure 7 Bootstrapping a compiler
Figure 8 A program that prints itself

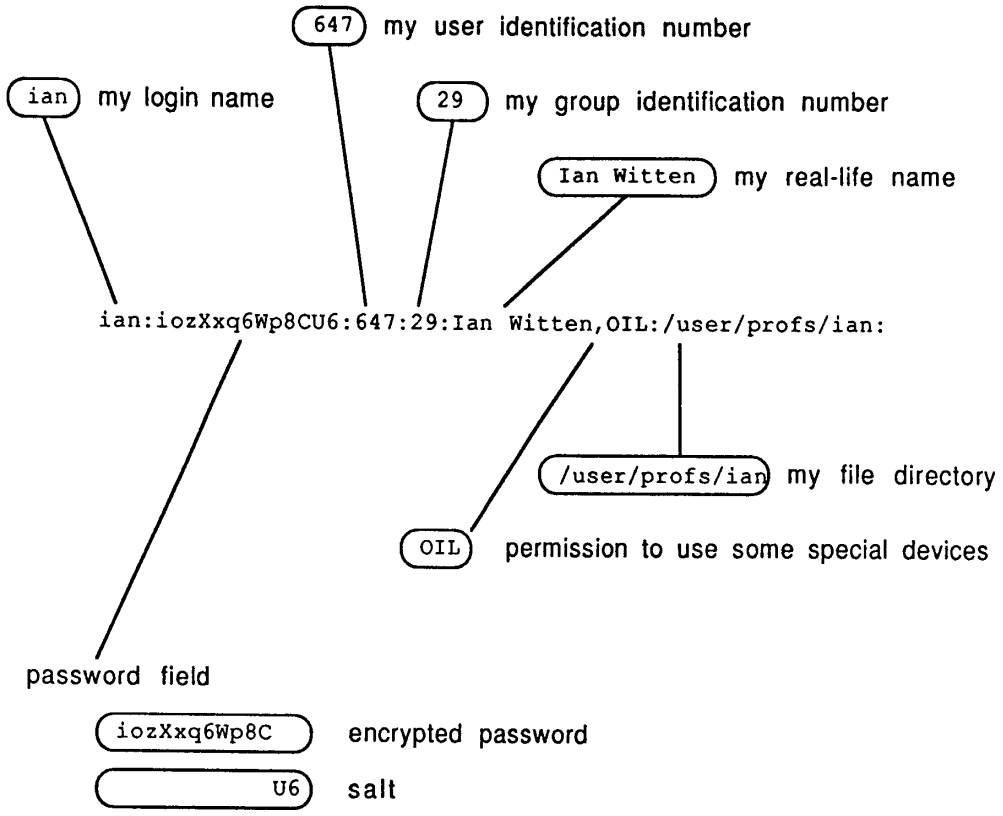


Figure 1 My entry in the password file

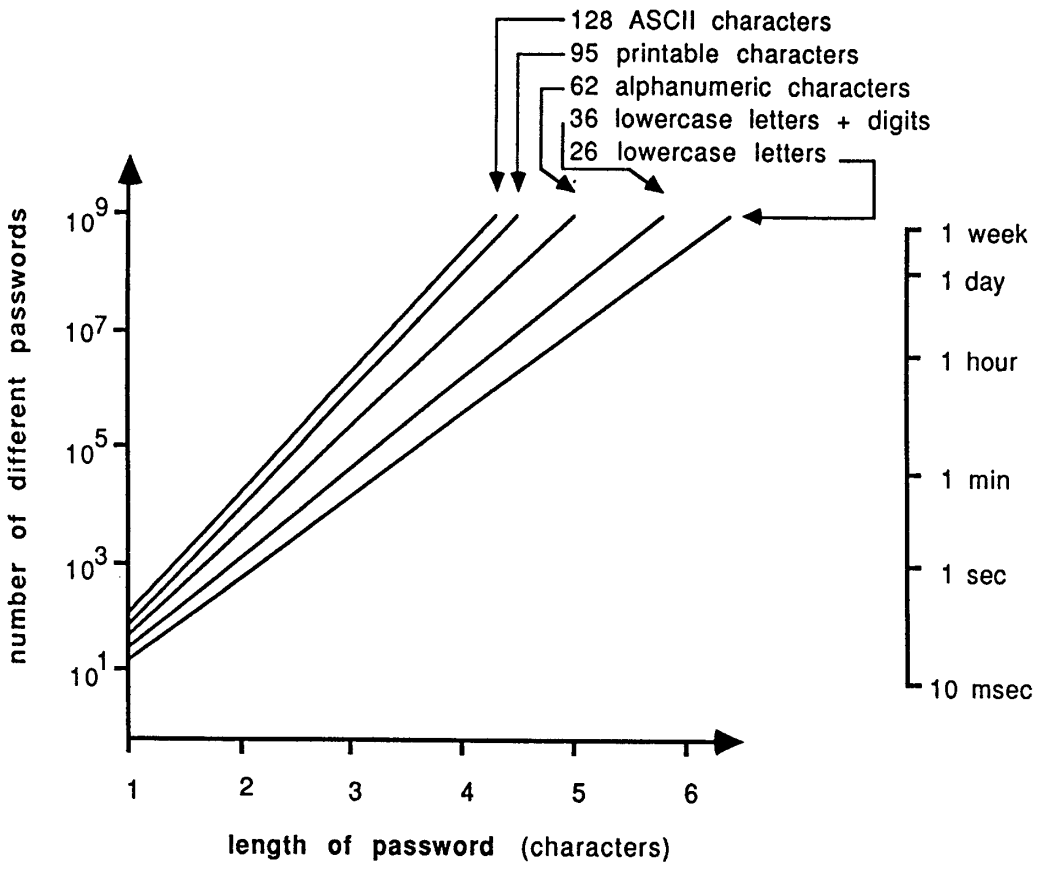


Figure 2 Cracking passwords of different lengths

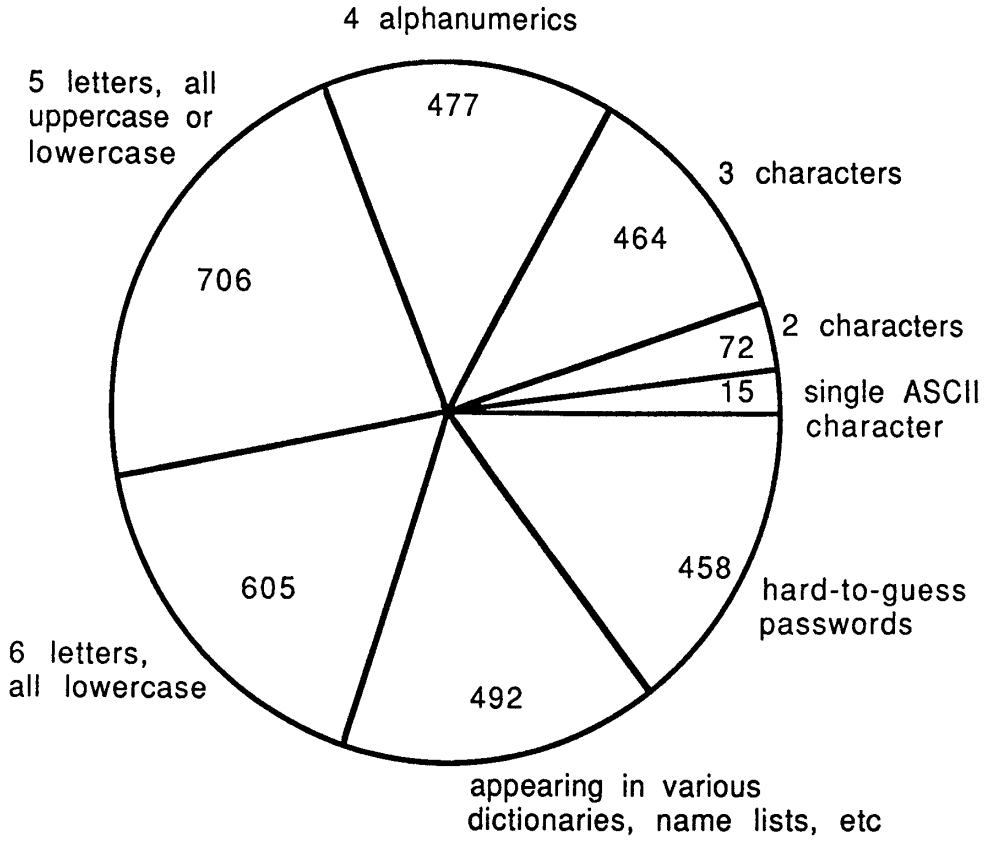


Figure 3 Breakdown of 3289 actual passwords (data from Morris & Thompson, 1978)

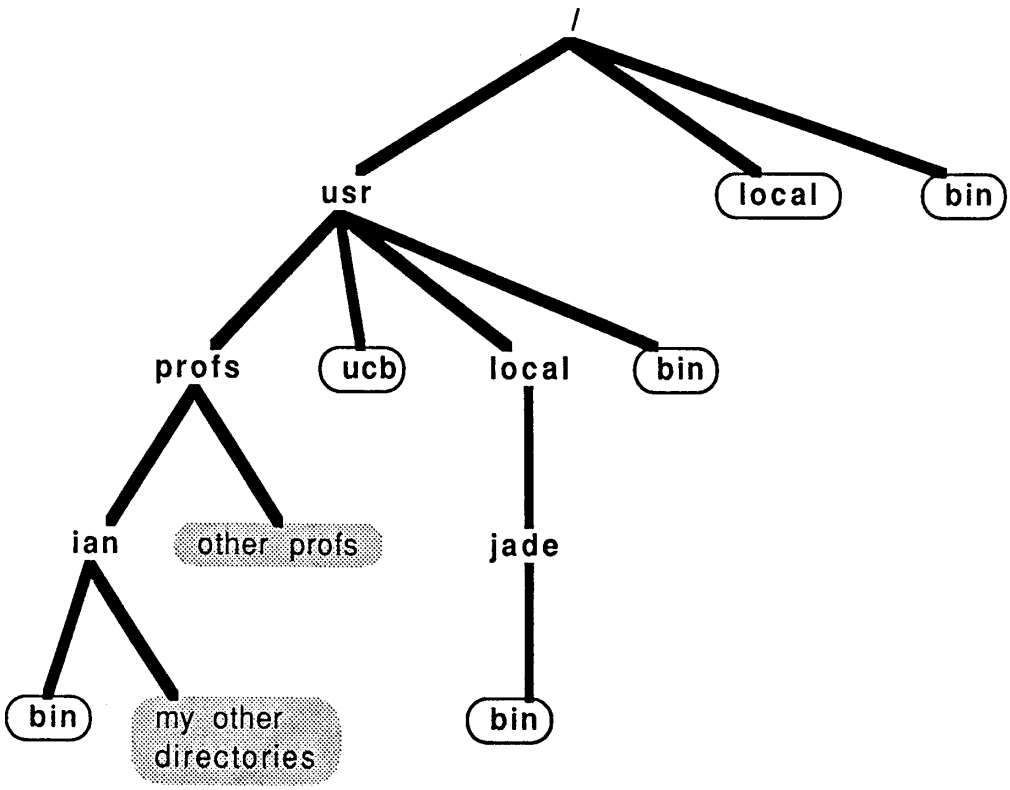


Figure 4 Part of a file hierarchy

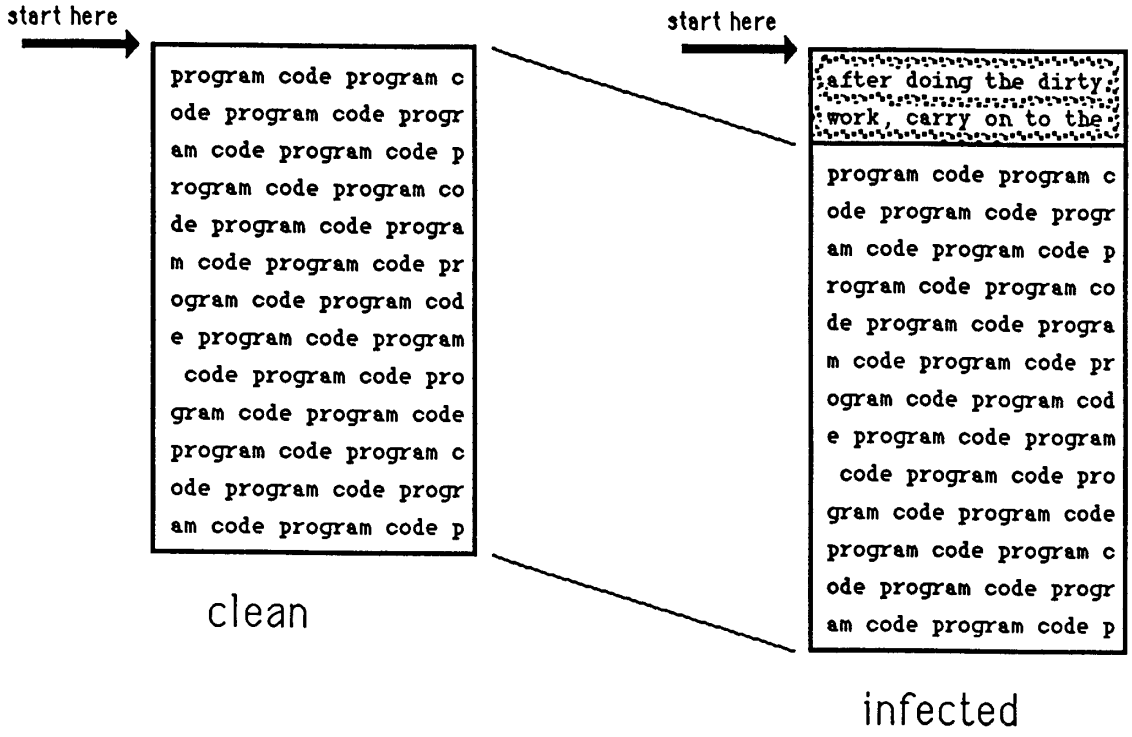
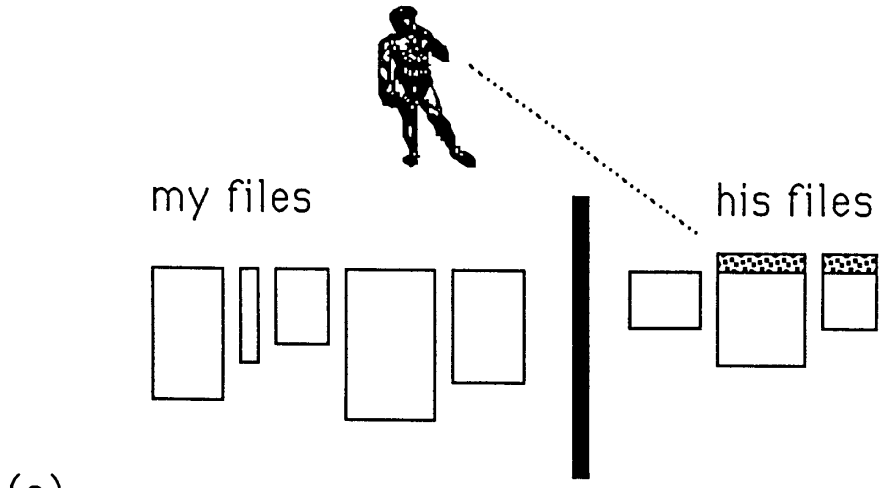
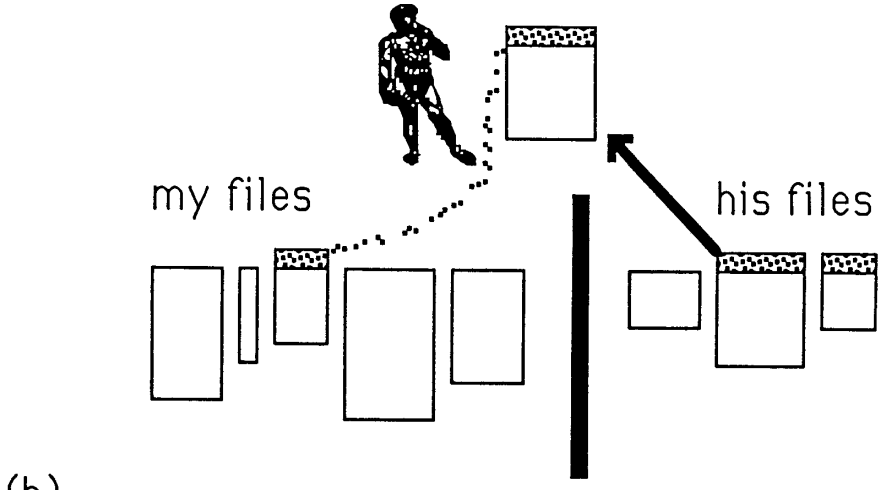


Figure 5 Anatomy of a virus



(a)



(b)

Figure 6 How a virus spreads

(a) I spot a program of his that I want to use ...
(b) ... and unknowingly catch the infection

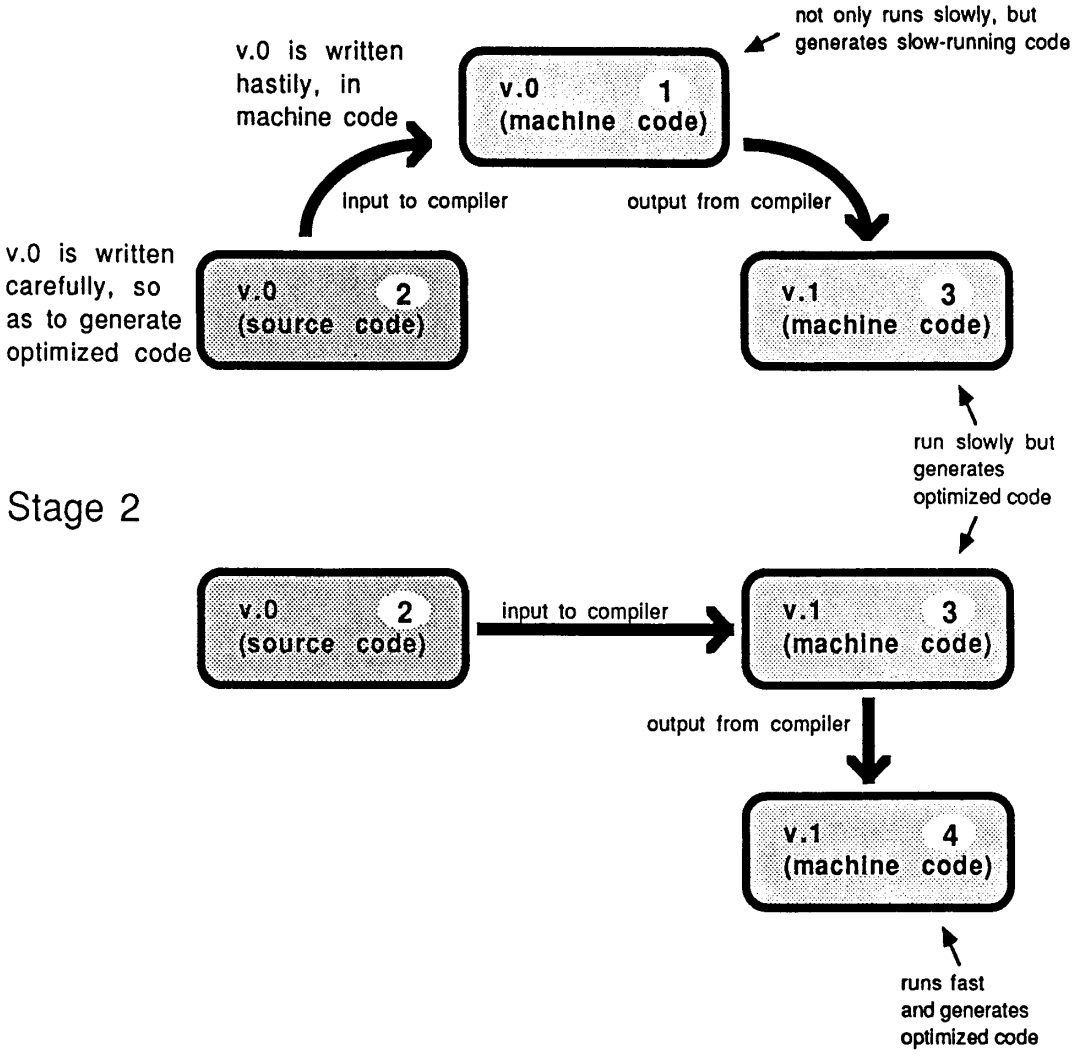


Figure 7 Bootstrapping a compiler

Figure 8 A program that prints itself

```
char t[] = {'0', ' ', '}', ';',
        '/', '*',
        ' ', '*', ' ', 'T', 'h', 'e', ' ', 's', 't', 'r', 'i', 'n', 'g', ' ', 't',
        ' ', '*', ' ', 'r', 'e', 'p', 'r', 'e', 's', 'e', 'n', 't', 's', ' ', 't', 'h', 'e',
        ' ', '*', ' ', 'b', 'o', 'd', 'y', ' ', 'o', 'f', ' ', 't', 'h', 'i', 's',
        ' ', '*', ' ', 'p', 'r', 'o', 'g', 'r', 'a', 'm', ' ', 'f', 'r', 'o', 'm', ' ', '"', '0', '"',
        ' ', '*', ' ', 't', 'o', ' ', 't', 'h', 'e', ' ', 'e', 'n', 'd',
        ' ', '*', '/',

        'm', 'a', 'i', 'n', '(', ')',
        '{',
        'i', 'n', 't', ' ', 'i', ';',

        'p', 'r', 'i', 'n', 't', 'f', '(', '"', 'c', 'h', 'a', 'r', ' ', 't', '[', ']', ' ', '=', ' ', '{', '"', ')', ';',
        'f', 'o', 'r', ' ', '(', 'i', '=', '0', ';', ' ', 't', '[', 'i', ']', '!', '=', '0', ';', ' ', 'i', '=', 'i', '+', '1', ')',
        'p', 'r', 'i', 'n', 't', 'f', '(', '"', '%', 'd', ',', ' ', '"', ',', ' ', 't', '[', 'i', ']', ')', ';',
        'p', 'r', 'i', 'n', 't', 'f', '(', '"', '%', 's', '"', ',', ' ', 't', ')', ';',
        '}',
0 };

/*
 * The string t
 * represents the
 * body of this
 * program from "0"
 * to the end
 */

main()
{
        int i;

        printf("char t[] = {");
        for (i=0; t[i]!=0; i=i+1)
                printf("%d, ", t[i]);
        printf("%s", t);
}
```