Research Institute for Advanced Computer Science
NASA Ames Research Center

Threats and Countermeasures for Network Security

Peter J. Denning

RIACS Technical Report 91.2

January 8, 1991

Threats and Countermeasures for Network Security

Peter J. Denning

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report TR-91.2
January 8, 1991

In the late 1980s, the traditional threat of anonymous breakins to networked computers was joined by viruses and worms, multiplicative surrogates that carry out the bidding of their authors. Technologies for authentication and secrecy, supplemented by good management practices, are the principal countermeasures. Four articles on these subjects by the author from *American Scientist* are gathered here.

This is a preprint of a paper that will be presented at the
Fourth Annual Computer Virus and Security Conference,
World Trade Center, New York City, March 14-15, 1991.

Work reported herein was supported in part by Cooperative Agreement NCC2-387
between the National Aeronautics and Space Administration (NASA)
and the Universities Space Research Association (USRA).

Threats and Countermeasures for Network Security

Peter J. Denning

Research Institute for Advanced Computer Science

January 8, 1991

As computing and telecommunications grow worldwide and are incorporated ever more deeply into business practices, we must deal with a widening variety of threats to the secure and dependable operation of our networks and the computers attached to them. Until 1987, breakins by anonymous intruders and frauds or thefts by insiders were the major threats. In that year, these threats were agumented by the new threats of viruses and worms, which are programs that act as multiplicative surrogates for an intruder. Moreover, viruses and worms are often loaded with logic bombs that wreak damage at some time after the infection.

Technologies for authentication and secrecy, supplemented by good management practices, are the principal countermeasures. Many observers have asked why little action has been taken to deploy these countermeasures. The recent appearance of the NRC report, *Computers at Risk* (1990), and the edited collection, *Computers Under Attack* (Addison-Wesley 1990), demonstrate a growing interest in doing just that. The positive public response to has been gratifying and portends safer networks in the years ahead.

Over the past several years I have written on these aspects for *American Scientist* magazine. What follows are the texts of four articles:

Computer Viruses	1988, No 3
The Internet Worm	1989, No 2
Security of Data in Networks	1987, No 1
Baffling Big Brother	1988, No 5

Computer Viruses

American Scientist 1988, No 3

The worm, Trojan horse, bacterium, and virus are destructive programs that attack information stored in a computer's memory. Virus programs, which propagate by incorporating copies of themselves into other programs, are a growing menace in the late-1980s world of unprotected, networked workstations and personal computers. Limited immunity is offered by memory protection hardware, digitally authenticated object programs, and antibody programs that kill specific viruses. Additional immunity can be gained from the practice of digital hygiene, primarily the refusal to use software from untrusted sources. Full immunity requires attention in a social dimension, the accountability of programmers.

Sometime in the middle 1970s, the network of computers at a Silicon Valley research center was taken over by a program that loaded itself into an idle workstation, disabled the keyboard, drew random pictures on the screen, and monitored the network for other idle workstations to invade. The entire network and all the workstations had to be shut down to restore normal operation.

In early September 1986, a talented intruder broke into a large number of computer systems in the San Francisco area, including 9 universities, 15 Silicon Valley companies, 9 ARPANET sites, and 3 government laboratories. The intruder left behind recompiled login programs to simplify his return. His goal was apparently to achieve a high score on the number of computers cracked; no damage was done *(1)*.

In December 1987, a Christmas message that originated in West Germany propagated into the Bitnet network of IBM machines in the United States. The message contained a program that displayed an image of a Christmas tree and sent copies of itself to everyone in the mail distribution list of the user for whom it was running. This prolific program rapidly clogged the network with a geometrically growing number of copies of itself. Finally the network had to be shut down until all copies could be located and expurgated.

For two months in the fall of 1987, a program quietly incorporated copies of itself into programs on personal computers at the Hebrew University. It was discovered and dismantled by a student, Yuval Rakavy, who noticed that certain library programs were growing longer for no apparent reason. He isolated the errant code and discovered that if executed on certain Fridays the thirteenth the computer running it would slow down by 80%, and on Friday 13 May 1988, it would erase all files. That date was the fortieth anniversary of the last day Palestine was recognized as a separate political entity. Rakavy designed another program that detected and erased all copies of the errant program it could find. Even so, he could not be completely sure he had eradicated it.

These four incidents illustrate the major types of programs that attack other programs in a computer's memory. The first type is a *worm*, a program that invades a workstation and disables it. The second is a *Trojan horse*, a program that performs some apparently useful function, such as login, while containing hidden code that performs an unwanted, usually malicious function. This name is inspired by the legendary wooden horse built by the Greek army, ostensibly as an offering to Athena, which in the dark of night disgorged its bellyful of murderous soldiers into the sleeping streets of Troy. The third type is a *bacterium*, a program that replicates itself and feeds off the host system by preempting processor and memory capacity. The fourth is a *virus*, a program that incorporates copies of itself into the machine codes of other programs and, when those programs are invoked, wreaks havoc in the manner of a Trojan horse.

I can cite numerous other incidents in which information stored in computers has been attacked by hostile programs. An eastern medical center lost nearly 40% of its records to a malicious program in its system. Students at Lehigh University lost homework and other data when a virus erased diskettes inserted into campus personal computers. Some programs available publicly from electronic bulletin boards have destroyed information on the disks of computers into which they were read. A recent *New York Times* article *(2)* describes many examples and documents the rising concern among computer network managers, software dealers, and personal computer users about these forms of electronic vandalism. In an effort to alert concerned computer scientists to the onslaught, the Association for Computing Machinery sponsors the Computer Risks Forum, an electronic newsletter moderated by Peter G. Neumann of SRI International, which regularly posts notices and analyses of such dangers.

The recent rash of viral attacks has drawn everyone's attention to the more general problem of computer security, a subject of great complexity which has fascinated researchers since the early 1960s *(3)*. The possibility of pernicious programs propagating through a file system has been known for at least twenty-five years. In his May 1985 Computer Recreations column in *Scientific American*, Kee Dewdney documented a whole menagerie of beastly threats to information stored in computer memories, especially those of personal computers *(4)*, where an infected diskette can transmit a virus to the main memory of the computer, and thence to any other diskette (or to hard disk). Ken Thompson, a principal designer of UNIX™, and Ian Witten have documented some of the more subtle threats to computers that have come to light in the 1980s *(5,6)*.

It is important to keep in mind that worms, Trojan horses, bacteria, and viruses are all programs designed by human beings. Although a discussion of these menaces brings up many intriguing technical issues, we should not forget that at the root of the problem are programmers performing disruptive acts under the cloak of anonymity conveniently provided by many computer systems.

I will focus on viruses, the most pernicious of the attacks against information in computers. A virus is a code segment that has been incorporated into the body of another program, "infecting" it. When the virus code is executed, it locates a few other uninfected programs and infects them; in due course, the number of infected programs can grow quite large. Viruses can spread with remarkable speed: in experimental work performed in 1983 and 1984, Fred Cohen of the University of Cincinnati demonstrated that a simple virus program can propagate to nearly every part of a normally operating computer system within a matter of hours. Most viruses contain a marker that allows

them to recognize copies of themselves; this avoids discovery, because otherwise some programs would get progressively longer under multiple infections. The destructive acts themselves come later: any copy of the virus that runs after some appointed date will perform such an unwanted function.

A Trojan horse program is the most common means of introducing a virus into a system. It is possible to rig a compiler with an invisible Trojan horse that implants another Trojan horse into any selected program during compilation.

A virus that takes the form of statements inserted into the high-level language version of a program -- that is, into the source file -- can possibly be detected by an expert who reads the program, but finding such a program in a large system can be extremely difficult. Many viruses are designed to evade detection completely by attaching themselves to object files, the machine coded images of high-level program sources that are produced by compilation. These viruses cannot be detected from a reading source programs.

The first serious discussions of Trojan horses took place in the 1960s. Various hardware features were developed to reduce the chances of attack *(3)*, including virtual memory, which restricts a program's to a limited region of memory, its "address space" *(7)*. All these features are based on the principle of least privilege, which reduces the set of accessible objects to the minimum a program needs in order to perform its function. Because a suspect program can be run in a strictly confined mode, any Trojan horse it contains will be unable to perform much damage.

How effective is virtual memory against viruses? Memory protection hardware can significantly reduce the risk, but a virus can still propagate to legitimately accessible programs, including portions of the operating system. The rate of propagation may be slowed by virtual memory, but propagation is not stopped. Most PCs are especially vulnerable because they have no memory protection hardware at all; an executing program has free access to anything in memory or on disk. A network of PCs is even more vulnerable, because any PC can propagate an infected copy of a program to any other PC, no questions asked.

What can be done to protect against viruses in a computer or workstation without memory protection hardware or controls on access to files? One common proposal is to retrofit the operating system with a write query check that asks the user for permission to allow the running program to modify a file. This gives the user an opportunity to determine that the program is attempting to gain acces to unauthorized files. It is, unfortunately, hardly workable even for experienced programmers because of the difficulty of discovering which files a running program must legitimately modify. A design that suppresses write queries for files named in an authorization list associated with a program can be subverted by a virus that adds the name of the unauthorized file to the list before attacking it.

A more powerful immunization scheme is based on digital signatures of object files. When a program is installed in a system, an authenticator is created by producing a checksum that depends on all the bits of a file, which is then signed with the secret key of the person who stored the file *(8)*. The authenticator can be unlocked by applying the public key of that person. A user can confirm that a file is an exact copy of what was stored by computing its checksum and comparing that with the unlocked authenticator.

A program infected by a virus would fail this test. Without access to the secret key, the designer of the virus could not produce a valid authenticator for the infected program. This scheme also works for programs obtained from trusted sources over a network: each program comes with an authenticator sealed by the trusted producer.

One way to implement this scheme is to equip the operating system with a background process that randomly checks files against their authenticators. If a virus has entered the system, this process will eventually discover an infected file and raise the alarm. Another way to implement this scheme is to "innoculate" an object program by placing an authentication subroutine at its entry point. This implementation is slow, however, and can be defeated by a virus that invades entry points: by the time the authenticator gets control, the virus will already have acted.

The authenticator scheme relies on the protection of the secret key, which cannot be complete unless the key is kept outside the system. It also rests on the integrity of the system itself: for example, a sophisticated attack against the program that reports whether a file has been infected could disable the scheme.

A program called an antibody can offer limited remedies should a virus penetrate a system. Such a program examines an object file to determine whether a known virus has been incorporated. It may also remove the virus from the infected program. This limited form of protection can be very effective against known viruses, but it cannot identify new ones.

As we have seen, each of the major technical mechanisms -- memory protection hardware, digital-signature authenticators, and antibodies -- offers limited protection against viruses (and Trojan horses). Can the operating procedures followed by those who use a computer system lower the risk further?

Yes! An additional measure of protection can be obtained by care in the way one uses a computer. Analogies with food and drug safety are helpful. Just as one would not consider purchasing food or capsules in unsealed containers or from untrusted sources, one can refuse to use any unsealed software or software from untrusted sources. Never insert a diskette that has no manufacturer's seal into your PC. Never use a program borrowed from someone who does not practice digital hygiene to your own standards. Beware of software obtained from public bulletin boards. Purchase programs that check other programs for known viruses. Be wary of public domain software (including virus eradicators!). Monitor the last-modified dates of programs and files. Don't execute programs sent in electronic mail -- even your friends may have inadvertently forwarded a virus. Don't let employees bring software from home.

The problem of viruses is difficult, both technically and operationally, and no solution oriented entirely along technical or operational lines can be complete. There is a third, social dimension to the problem: we don't know how to hold people fully accountable for the actions of their programs in a networked system of computers. A complete solution must involve all three dimensions.

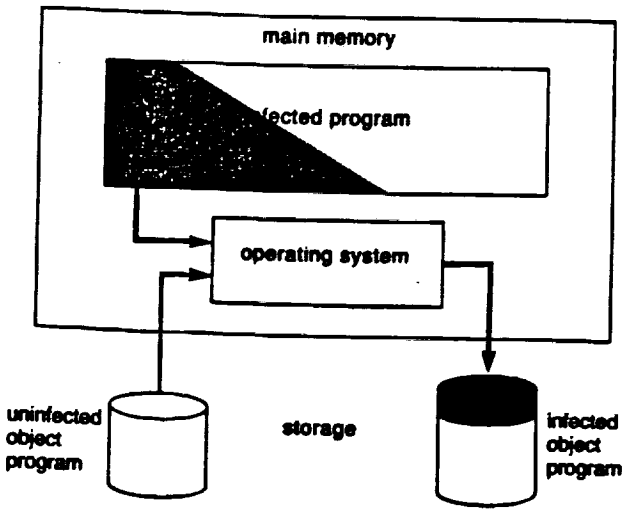
Computer scientists are divided over whether it serves the field to publish accounts of viral attacks in full technical detail. (This article, being superficial, does not count.) Some hold that revelations of technical detail -- as in Dewdney *(4)* or Witten *(6)* -- are reprehensible because they give the few would-be perpetrators a blueprint for actions that can make life exceedingly difficult for the many innocent users, and because there are

few successful defenses against the attacks. Others hold that the main hope for a long term solution is to mobilize the "good guys" by setting forth the problems in detail; the short term risk, according to this view, is offset by the long-term gain. Most computer scientists favor this way of mobilizing forces to oppose computer sabotage.

References

1. B. Reid. 1987. Reflections on some recent widespread computer breakins. *ACM Communications 30*, 2. February. 103-105.
2. Vin McLellan. 1988. Computer systems under siege. *NY Times* Sunday Business Section. January 31.
3. D. E. Denning. 1982. *Cryptography and Data Security*. Addison-Wesley.
4. A. K. Dewdney. 1985. Computer Recreations (A Core War Bestiary of Viruses, Worms, and other Threats to Computer Memories). *Scientific American 252*, 3. March. 14-23.
5. K. Thompson. 1984. Reflections on trusting trust. *ACM Communications 27*, 8. August. 172-80.
6. Ian H. Witten. 1987. Computer (In)security: Infiltrating Open Systems. *Abacus 4*, 4. Summer. 7-25.
7. P. J. Denning. 1986. Virtual memory. *American Scientist 74*, 3 (May-June). 227-229.
8. P. J. Denning. 1987. Security of data in networks. *American Scientist 75*, 1 (January-February). 12-14.

How a virus works

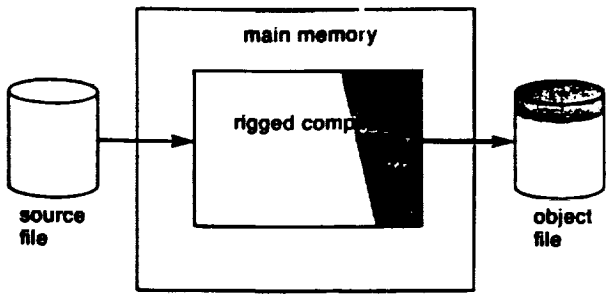


A program infected with a virus (*shaded area*) and loaded and executing in the main memory of a computer can infect another executable (object) program in the computer's disk storage system by secretly requesting the computer's operating system to append a copy of the virus code to the object program, usually at the start. The infection makes the object program slightly longer.

When the newly infected program is itself loaded into memory and invoked, the virus in it takes control and performs hidden functions, such as infecting yet other object programs. The virus may also perform destructive functions before transferring control to the original entry point. The virus code contains a marker so that the virus won't attempt to infect a program already infected by its own kind: multiple infections would cause an object file to grow ever larger, leading to easy detection.

The same principle works in personal computers, where floppy disks play the role of object programs in the description above. In this case, the virus usually attacks the copy of the operating system contained on the floppy disk so that the virus is automatically invoked whenever the disk's operating system is started. Since the operating system then resides in the PC's main memory, it can infect any diskettes inserted into the PC.

A Trojan horse in a compiler



A Trojan horse is a useful program containing hidden code (*shaded area*) that performs an unwanted, mischievous function. It might copy an invoker's private files into an area of memory belonging to its own designer, thereby circumventing the invoker's file protection. It might obtain access to a subsystem normally inaccessible to the designer. A Trojan horse that destroys or erases files is also called a logic bomb.

It is sometimes suggested that Trojan horses can be detected by scanning a program's source file for statements that perform operations outside the program's specifications. Ken Thompson, one of the principal designers of UNIX™, has pointed out that this approach is fundamentally incomplete, demonstrating how to rig a compiler to introduce a Trojan horse into the object file of any other selected program, for example a login program (*5*). Whenever the login program is recompiled, the rigged compiler always inserts a segment of code that allows login when a special password (known only to the Trojan horse's designer) is given. The login program's Trojan horse cannot be detected by reading its source file.

Now, it might seem that a careful reading of the rigged compiler's own source file would reveal the Trojan horse that inserts the login Trojan horse. But this is not so. The rigged compiler is itself an object file, and can thereby contain its own Trojan horse without a record in its source file. Thompson has demonstrated a scheme to rig a compiler in this way (*5, 6*).

The Internet Worm

American Scientist 1989, No 2

In November 1988 a worm program invaded several thousand UNIX-operated Sun workstations and VAX computers attached to the Research Internet, seriously disrupting service for several days but damaging no files. An analysis of the worm's decompiled code revealed a battery of attacks by a knowledgeable insider, and demonstrated a number of security weaknesses. The attack occurred in an open network, and little can be inferred about the vulnerabilities of closed networks used for critical operations. The attack showed that password protection procedures need review and strengthening. It showed that sets of mutually trusting computers need to be carefully controlled. Sharp public reaction crystalized into a demand for user awareness and accountability in a networked world.

Late in the evening of 2 November 1988 someone released a "worm" program into the ARPAnet. The program expropriated the resources of each invaded computer and generated replicas of itself on other computers, but did no apparent damage. Within hours, it had spread to several thousand computers attached to the worldwide Research Internet.

Computers infested with the worm were soon laboring under a huge load of programs that looked like innocuous "shell" programs (command interpreters). Attempts to kill these programs were ineffective: new copies would appear from Internet connections as fast as old copies were deleted. Many systems had to be shut down and the security loopholes closed before they could be restarted on the network without reinfestation.

Fortuitously, the annual meeting of UNIX experts opened at Berkeley on the morning of November 3. They quickly went to work to capture and dissect the worm. By that evening, they had distributed system fixes to close all the security loopholes used by the worm to infest new systems. By the morning of November 4, teams at MIT, Berkeley, and other institutions had decompiled the worm code and examined the worm's structure in the programming language C. They were able to confirm that the worm did not delete or modify files already in a computer. It did not install Trojan horses, exploit superuser privileges, or transmit passwords it had deciphered. It propagated only by the network protocols TCP/IP, and it infested only computers running Berkeley UNIX but not AT&T System V UNIX. As the community of users breathed a collective sigh of relief, system administrators installed the fixes, purged all copies of the worm, and restarted the downed systems. Most hosts were reconnected to the Internet by November 6, but the worm's effect lingered: a few hosts were will disconnected as late as November 10, and mail backlogs did not clear until November 12.

The worm's fast and massive infestation was so portentous that the *New York Times* ran updates on page one for a week. The *Wall Street Journal* and *USA Today* gave it front-page coverage. It was the subject of two articles in *Science* magazine *(1,2)*. It was covered by the wire services, the news shows, and the talk shows. These accounts said that over 6,000 computers were infested, but later estimates put the actual number between 3,000 and 4,000, about 5% of those attached to the Internet.

On November 5 the *New York Times* broke the story that the alleged culprit was Robert T. Morris, a Cornell graduate student and son of a well-known computer security expert who is the chief scientist at the National Computer Security Center. A friend reportedly said that Morris intended no disruption; the worm was supposed to propagate slowly but a design error made it unexpectedly prolific. When he realized what was happening, Morris has a friend post on an electronic bulletin board instructions telling how to disable the worm -- but no one could access them because all affected computers were down. As of February 1989, no indictments had been filed against Morris as authorities pondered legal questions. Morris himself was silent throughout.

The worm's author went to great lengths to confound its discovery and analysis, a delaying tactic that permitted the massive infestation. By early December 1988, Eugene Spafford of Purdue *(3)*, Donn Seeley of Utah *(4)*, and Mark Eichin and Jon Rochlis of MIT *(5)* had published technical reports about the decompiled worm that described the modes of infestation and the methods of camouflage. (See Box 1.) They were impressed with the worm's battery of attacks, saying that, despite errors in the source program, the code was competently done. The National Computer Security Center requested them and others not to publish the decompiled code, fearing that troublemakers might reuse the code and modify it for destructive acts. Seeley replied that the question is moot because the worm published itself in thousands of computers.

The reactions of the computer science community have been passionate. Some editorial writers report that Morris has become a folk hero among students and programmers, who believe that the community ought to be grateful that he showed us weaknesses in our computer networks in time to correct them before someone launches a malicious attack. The great majority of opinion, however, seems to go the other way. Various organizations have issued position statements decrying the incident and calling for action to prevent its recurrence. No other recent break-in has provoked similar outcries.

The organization Computer Professionals for Social Responsibility issued a statement calling the release of the worm an irresponsible act and declaring that no programmer can guarantee that a self-replicating program will have no unwanted consequences. The statement said that experiments to demonstrate network vulnerabilities should be done under controlled conditions with prior permission, and it called for codes of ethics that recognize the shared needs of network users. Finally, the statement criticized the National Computer Security Center's attempts to block publication of the decompiled worm code as short-sighted because an effective way to correct widespread security flaws is to publish descriptions of those flaws widely.

The boards of directors of the CSNET and BITNET networks issued a joint statement deploring the irresponsibility of the worm's author and the disruption in the research community caused by the incident. Their statement called for a committee that would issue a code of network ethics and propose enforcement procedures. It also called

for more attention to ethics in university curricula. (At Stanford, Helen Nissenbaum and Terry Winograd have already initiated a seminar that will examine just such questions.)

The advisory panel for the division of networking and research infrastructure at NSF endorsed the CSNET/BITNET statement, citing as unethical any disruption of the intended use of networks, wasting of resources through disruption, destruction of computer-based information, compromising of privacy, or actions that make necessary an unplanned consumption of resources for control and eradication. The Internet Activities Board has drafted a similar statement. The president of the Association for Computing Machinery called on the computer science community to make network hygiene a standard practice *(6)*. A congressional bill introduced July 1988 by Wally Herger (R-Calif.) and Robert Carr (D-Mich.), called the Computer Virus Eradication Act, will doubtless reappear in the 101st Congress.

Obviously, all this interest is provoked by the massive scale of the worm's infestation and the queasy feeling that follows a close call. It also provides an opportunity to review key areas of special concern in networking. In what follows, I will comment on vulnerabilities of open and closed networks, password protection, and responsible behavior of network users.

The rich imagery of worms and viruses does not promote cool assessments of what actually happened and of what the future might hold. It is interesting that as recently as 1982 worm programs were envisaged as helpful entities that located and used idle workstations for productive purposes *(7)*; most people no longer make this benign interpretation. Some of the media reports have mistakenly called the invading program a virus rather than a worm. A virus is a code segment that embeds itself inside a legitimate program and is activated when that program is; it then embeds another copy of itself in another legitimate but uninfected program, and it usually inflicts damage *(8)*. Because the virus is a more insidious attack, the mistaken use of terminology exaggerated the seriousness of what happened. Given that the security weaknesses in the Internet service programs have been repaired, it is unlikely that an attack against these specific weaknesses could be launched again.

While it is important not to overestimate the seriousness of the attack, it is equally important not to underestimate it. After all, the worm caused a massive disruption of service.

It is important to aknowledge a widespread concern that grew out of this attack: Are networks on which commerce, transportation, utilities, national defense, space flight, and other critical activities depend also vulnerable? This concern arises from an awareness of the extent to which the well-being of our society depends on the continued proper functioning of vast networks that may be fragile. When considering this question, it is important to bear in mind that the Internet is an open network and the others are closed.

What is the risk to an open network? Because the Internet is open by design, its computers also contain extensive backup systems. Thus, in the worst case, if the worm had destroyed all the files in all the computers it invaded, most users would have experienced the loss of only a day's work. (This contrasts starkly to the threat facing most PC users, who because of the lack of effective backup mechanisms stand to lose years of work to a virus attack.) In addition, users would certainly lose access to their systems for a day or more as the operations staff restored information from backups.

What are the implications for other networks? Computers containing proprietary information or supporting critical operations are not generally connected to the Internet; the few exceptions are guarded by gateways that enforce strict access controls. For example, the Defense Department's command and control network and NASA's space shuttle network are designed for security and safety; it is virtually impossible for a virus or worm to enter from the outside, and internal mechanisms would limit damage from a virus or worm implanted from the inside. Given that the Internet is designed for openness, it is impossible to draw conclusions about closed networks from this incident.

Calls to restrict access to the Internet are ill-advised. The openness of the Internet is closely aligned with a deeply held value of the scientific community, the free exchange of research findings. The great majority of scientists are willing to accept the risk that their computers might be temporarily disabled by an attack, especially if a backup system limits losses to a day's work.

The next area that calls for special concern is password security. Although trapdoors and other weaknesses in Internet protocols have been closed, password protection is a serious weakness that remains. (See Box 2.) The risk is compounded by "mutually trusting hosts," a design in which a group of workstations is declared as a single system: access to one constitutes access to all.

Many PC systems store passwords as unenciphered cleartext, or they do not use passwords at all. When these systems become part of a set of trusting hosts, they are an obvious security weakness. Fortunately, most systems do not store passwords as cleartext. In UNIX, for example, the login procedure takes the user's password, enciphers it, and compares the result with the user's enciphered entry in the password file. But one can discover passwords from a limited set of candidates by enciphering each one and comparing it with the password file until a match is found. One study of password files revealed that anywhere from 8% to 30% of the passwords were the literal account name or some simple variation; for example, an account named "abc" is likely to have the password "abc", "bca", or "abcabc" *(9)*. The worm program used a new version of the password encryption algorithm that was nine times faster than the regular version in UNIX; this allowed it to try many more passwords in a given time and increased its chances of breaking into at least one account on a system. Having broken into an account, the worm gained easy access to that computer's trusted neighbors.

The final area of special concern is the responsibilities of people who participate in a large networked community. Although some observers say that the worm was benign, most say that the disruption of service and preemption of so many man-hours to analyze the worm was a major national expense. Some observers have said that the worm was an innocent experiment gone haywire, but the experts who analyzed the code dispute this, saying that the many attack modes, the immortality of some worms, and the elaborate camouflage all indicate that the author intended the worm to propagate widely before it was disabled. Most members of the computer science community agree that users must accept responsibility for the possible wide-ranging effects of their actions and that users do not have license to access idle computers without permission. They also believe that the professional societies should take the lead in public education about the need for responsible use of critical data now stored extensively in computers. Similarly, system administrators have responsibilities to take steps that will minimize the risk of disruption: they should not tolerate trapdoors, which permit access without authentication; they

should strengthen password authentication procedures to block guessed-password attacks; they should isolate their backup systems from any Internet connection; and they should limit participation in mutually trusting groups.

Certainly the vivid imagery of worms and viruses has enabled many outsiders to appreciate the subtlety and danger of attacks on computers attached to open networks. It has increased public appreciation of the dependence of important segments of the economy, aerospace systems, and defense networks on computers and telecommunications. Networks of computers have joined other critical networks that underpin our society -- water, gas, electricity, telephone, air traffic control, banking, to name a few. Just as we have worked out ways to protect and ensure general respect for these other critical systems, we must work out ways to promote secure functioning networks of computers. We cannot separate technology from responsible use.

References

1. E. Marshall. 1988. "Worm invades computer networks." *Science* **242** (11 Nov 1988). 855-856.
2. E. Marshall. 1988. "The worm's aftermath." *Science* **242** (25 Nov 1988). 1121-1122.
3. E. Spafford. 1988. "The Internet worm program: an analysis." Technical Report No. CSD-TR-823, available from Computer Sciences Department, Purdue University, W. Lafayette, IN 47907. Published in the ACM *Computer Communication Review*, January 1989, available from ACM, Inc., 11 W. 42 St., New York, NY 10036.
4. D. Seeley. 1988. "A tour of the worm." Technical Report available from the Computer Science Department, University of Utah, Salt Lake City, UT 84112. In *Proc. Winter Usenix Conf.*, February 1989, Usenix Association.
5. M. Eichin and J. Rochlis. 1988. "With microscope and tweezers: an analysis of the Internet Virus of November 1988." Technical Report, MIT Project Athena, Cambridge, MA 02139.
6. B. Kocher. 1989. "A hygiene lesson." *Communications of ACM* **32**, 3 & 6.
7. J. F. Shoch and J. A. Hupp. 1982. "The worm programs -- early experience with a distributed computation." *Communications of ACM*, **25**, 3. 172-180.
8. P. J. Denning. 1988. "Computer viruses." *American Scientist* **76**, 3 (May-June). 236-238.
9. F. T. Grammp and R. H. Morris. 1984. "UNIX operating system security." *AT&T Bell Labs Technical Journal* **63**, 8. October. 1649-1672.

BOX 1: How the worm worked

The Internet worm of November 1988 was a program that invaded Sun 3 and VAX computers running versions of the Berkeley 4.3 UNIX operating system containing the TCP/IP Internet protocols. Its sole purpose was to enter new machines by bypassing authentication procedures and to propagate new copies of itself. It was prolific, generating on the order of hundreds of thousands of copies among several thousand machines nationwide. It did not destroy information, give away passwords, or implant Trojan horses for later damage.

A new worm began life by building a list of remote machines to attack. It made its selections from the tables declaring which other machines are trusted by its current host, from users' mail-forwarding files, from tables by which users give themselves permission for access to remote accounts, and from a program that reports the status of network connections. For each of these potential new hosts, it attempted entry by a variety of means: masquerading as a user by logging into an account after cracking its password; exploiting a bug in the finger protocol, which reports the whereabouts of a remote user; and exploiting a trapdoor in the debug option of the remote process that receives and sends mail. In parallel with attacks on new hosts, the worm undertook to guess the passwords of user accounts on its current host. It first tried the account name and simple permutations of it, then a list of 432 built-in passwords, and finally all the words from the local dictionary. An undetected worm could have spent many days at these password-cracking attempts.

If any of its attacks on new hosts worked, the worm would find itself in communication with a "shell" program -- a command interpreter – on the remote machine. It fed that shell a 99 line bootstrap program, together with commands to compile and execute it, then broke the connection. If that bootstrap program started successfully, it would call back the parent worm within 120 seconds. The parent worm copied over enciphered files containing the full worm code, which was compiled from a C program containing about 3,000 lines. The parent worm then issued commands to construct a new worm from the enciphered pieces and start it.

The worm also made attempts at population control, looking for other worms in the same host and negotiating with them which would terminate. However, a worm that agreed to terminate would first attack many hosts before completing its part of the bargain -- leaving the overall birthrate higher than the deathrate. Moreover, one in seven worms declared itself immortal and entirely bypassed any participation in population control.

The worm's author went to considerable pains to camouflage it. The main worm code was enciphered and sent to the remote host only when the bootstrap was known to be operating there as an accomplice. The new worm left no traces in the file system: it copied all its files into memory and deleted them from a system's directories. The worm disabled the system function that produces "memory dumps" in case of error, and it kept all character strings enciphered so that, in case a memory dump were obtained anyway, it would be meaningless. The worm program gave itself a name that made it appear as an innocuous shell to the program that lists processes in the system, and it frequently changed its process identifier.

Box 2: Protecting Passwords

The worm's dramatic demonstration of the weakness of most password systems should prompt a thorough examination in the context of networks of computers. The following are basic desiderata:

1. Every account should be protected by a password.
2. Passwords should be stored in an enciphered form, and the file containing the enciphered passwords should not be publicly accessible (it is in UNIX).
3. Passwords should be deliberately chosen so that simple attacks cannot work -- for example, they could include a punctuation mark and a numeral.
4. New passwords should be checked for security -- many systems have (friendly!) password checkers that attempt to decipher passwords by systematic guessing, sending warning messages to users if they are successful.
5. To make extensive guessing expensive, the running time of the password encryption algorithm should be made high, on the order of one second. This can be achieved by repeatedly enciphering the password with a fast algorithm.
6. New cost-effective forms of user authentication should be employed, including devices to sense personal characteristics such as fingerprints, retinal patterns, or dynamic signatures, as well as magnetic access cards.
7. Sets of computers that are mutually trusting in the sense that login to one constitutes login to all need to be carefully controlled. No computer outside the declared set should have unauthenticated access, and no computer inside should grant access to an outside computer.

Security of Data in Networks

American Scientist 1987, No 1

Telescience is NASA's word for scientific research conducted via networks that permit remote control of experiments and collaboration of scientists around the world on analyzing the results. The safety of remotely controlled experiments and integrity of research rest critically on the ability of the network to authenticate senders and receivers, to protect proprietary communications, and to sign some transmissions. Mathematically sound schemes for encrypting data and distributing keys make these goals attainable.

Telescience. This term is used by NASA to refer to scientific research conducted with computers and instruments connected by networks over great distances. It includes the remote design of experiments on space platforms, the operation of those experiments, and the collaboration of scientists around the world in interpreting data and publishing results. The next best thing to being there, telescience is expected to be a common mode of research in all scientific fields by the mid 1990s.

For the safety of remotely-controlled operations and the integrity of their research, experimenters want to be certain that they are linked to their own instruments when they request connections and that no one else can connect to those instruments. They want to be certain that no one can alter the data transmitted from their instruments, or the authorized commands sent to the instruments. They want to be certain that proprietary communications with their co-workers cannot be disclosed. The first guarantee, called authentication, certifies the identity of a principal -- person, computer, or device -- accessible on the network. The second guarantee, called integrity, certifies that a data stream actually comes from a previously authenticated source. The third guarantee, called secrecy, certifies that the content of a data stream is hidden from outside view. Data transmissions covered by these guarantees are called secure communications. Telescience requires secure communications over high-bandwidth networks -- 1 million bits per second (Mbps) or more.

Who furnishes these guarantees? The agencies that design and operate a network must provide for them in the communications protocols. All such mechanisms ultimately require that each principal can possess or obtain information that identifies any other principal. The identifying information can be embodied as a key to encipher data. The mechanisms must be capable not only of efficiently enciphering and deciphering data, but of distributing and protecting keys. In what follows, I will present a brief survey of this fascinating subject. A comprehensive treatment can be found in Dorothy Denning's book *Cryptography and Data Security (1)*.

Communication between principals can be a two-way conversation in real time, a one-way, high-rate data stream, or a one-way mail or datagram message. Some communications must be signed by attaching an unforgeable mark that will establish the

sender's identity beyond reasonable doubt.

A communications path through a network may include many links, switches, computers, local networks, and internetwork gateways. In most networks these components are vulnerable because data security was not a requirement of the original design. Each component is a potential site for an intruder to eavesdrop on a conversation, read mail, replay portions of prior messages, or alter a data transmission. Because a pair of principals wishing to communicate have no control over these many network components, they must use protocols that allow them to control the encryption devices and the keys.

Traditional cryptosystems are based on a single key K known only to A and B, the principals who wish to communicate. A message M is sent as ciphertext, denoted $[M]^K$. This scheme provides authentication as well as secrecy: if an attempt by B to decipher a message produces gibberish, B knows that A could not have been the sender.

The best known computer-based cryptosystem is the Data Encryption Standard (DES), promulgated in 1977 by the National Bureau of Standards. The DES uses a 56-bit key to encipher successive 64-bit blocks of data. Computer chips embodying the DES algorithm operate at speeds beyond 10 Mbps, which is faster than needed for most wide-area communication networks. Controversies arose at the beginning over whether the DES key was long enough to prevent the code's being broken by an enumerative search for the key, and whether the code contained secret trapdoors that would permit the government to read DES ciphers. Those controversies have quieted; no trapdoors have been found. Double or triple encryption with different keys can be used for extra protection. Because the DES is now ten years old, cryptographers have begun to seek replacements suitable for commercial use.

Another kind of cryptosystem was proposed in 1976 by Whitfield Diffie and Martin Hellman of Stanford University. They called theirs a public-key cryptosystem to distinguish it from the traditional private-key systems. The public-key system uses two complementary keys: one is made public and is used to encipher messages; the other is kept secret and is used to decipher messages. The secret key cannot be deduced from the public key. Single-key cryptosystems are symmetric because the same key is used for both enciphering and deciphering; two-key cryptosystems are asymmetric. In a symmetric cryptosystem, almost any binary pattern can serve as a key, but a good deal of computation is required to generate a pair of keys for an asymmetric cryptosystem.

The notation for a public-key system is straightforward. A principal A holds secret and public keys, denoted SA and PA. To communicate with A, B sends the ciphertext $[M]^{PA}$; A recovers the message by enciphering the ciphertext with the secret key, because $M=[[M]^{PA}]^{SA}$. A and B can hold a conversation by exchanging messages enciphered under each other's public keys. Secrecy is assured because there is only one copy of the secret key, held by the principal who generated it.

Secrecy and authentication are separated in a two-key cryptosystem. Secrecy results from enciphering with the recipient's public key: anyone can generate $[M]^{PA}$, but only A can decipher it. Authentication results from enciphering with the sender's secret key: only A can generate $[M]^{SA}$, and anyone can decipher it. Two encipherments are needed to provide both: $[[M]^{SA}]^{PB}$ can be enciphered only by A and deciphered only by B.

The first public-key cryptosystem with these properties was devised in 1977 by Ronald Rivest, Adi Shamir, and Len Adleman of MIT, and is known by their initials, RSA *(2)*. It works as follows: To generate a key, pick two large prime numbers p and q. Then choose two integers d and e so that $de \bmod (p-1)(q-1) = 1$. (In general, $x \bmod y$ means the remainder after dividing x by y.) Let $n=pq$. The secret key is (d,n) and the public key is (e,n). To encipher, compute $C = [M]^{PA} = M^e \bmod n$. To decipher, compute $M = [C]^{SA} = C^d \bmod n$. Deciphering recovers M because of a classical theorem of Fermat that says $M^{de} \bmod n = M$.

As an example, suppose $p=3$ and $q=11$; then $n=33$ and $(p-1)(q-1)=20$. Pick $(d,e)=(3,7)$; this is valid because $de \bmod 20 = 21 \bmod 20 = 1$. Suppose $M=4$; the ciphertext is then $C=16$, because $4^7 \bmod 33 = 16384 \bmod 33 = (33\times496+16) \bmod 33 = 16$. The deciphered message is $M=4$, because $16^3 \bmod 33 = 4096 \bmod 33 = (33\times124+4) \bmod 33 = 4$.

The security of the RSA system relies on the extreme difficulty of factoring a large composite number: If the prime components p and q could be recovered easily from n, a deciphering key matching the public enciphering key could be computed easily. In the summer of 1986, researchers at the Mitre Corporation factored an 84-digit number, the largest ever, after several days of computation on a set of cooperating computers. To protect against faster supercomputers and improved factoring algorithms, most designers of RSA systems recommend that n be on the order of 200 digits (about 665 bits).

Computer chips containing the RSA algorithm have been developed. Because of the large number of digits in each block of enciphered data (around 200), these chips are rather slow, operating on the order of a few kilobits per second. This means that known public-key systems are too slow for high-bandwidth, secret conversations between computers.

What, then, is the advantage of a public-key system? It is the ability to separate authentication from secrecy. This separation permits digital signatures, which allow third parties to certify the identity of a sender. It works as follows: A signed message consists of a header H, a body M, and a signature block $X=[F(M)]^{SA}$; the header asserts that the message came from some sender, say A; the signature is a small block computed from M and then signed with A's secret key. The data-compression function F, often called a hashing function, is public; its result, $F(M)$, is called a checksum. The receiver will accept the message only if the signature, deciphered with A's public key, is identical to the checksum of the message actually received. If A claims that B changed the message, or B claims that A sent a different message, a third party can resolve the dispute by deciphering the signature and comparing it to the claimed message's checksum. If the message is a secret, the message body can be the ciphertext $[M]^K$ and the enciphered key $[K]^{PB}$ can be added to the signature block.

The same principles work in broader arenas. Suppose the space station contains a telescope that emits a stream of data, which, by treaty, is supposed to be available to every astronomer in the world. How can an astronomer be assured that a data stream is in fact the one transmitted by the telescope, and that none of the data have been altered? The raw data can be collected in a local buffer in the space telescope, which is assigned a public key PT and secret key ST. Each buffer is treated as a message M; when the buffer is full, the authenticator $[F(M)]^{ST}$ is appended, and the result is transmitted publicly. Any receiver can reverse the process and check that each block of data is

authentic.

In 1978, Gus Simmons of Sandia Laboratories proposed a similar scheme for the verification of compliance with test-ban treaties. He assumed that the United States would require assurances that its monitoring device implanted in Soviet soil had not been tampered with, and the Soviets would want to be able to read the transmissions of the device.

There are many practical considerations to building secure signature systems that will work in large networks. For example, the hashing function must deprive potential intruders of effective means to construct fake messages with the same checksums as authentic messages. The subject is covered well in articles by Donald Davies and Dorothy Denning *(3,4)*.

A cryptosystem is useless unless distribution of keys is secure. Let us examine this problem for networks in which all conversations are protected by private-key cryptosystems. How are keys handed out so that the communicants are sure of one another's identities? An obvious solution relies on a registry service R. A private key is generated for each principal A, one copy of which is stored in R and another copy on a key card (or other medium) that can be inserted into an encryption device attached to A. Now it is possible for R to provide A with private keys for conversations with other principals in the network. Roger Needham and Michael Schroeder have proposed protocols that allow any A and B, with help from R, to obtain a private key for a secure communication between them *(5)*. Victor Voydock and Stephen Kent have shown how to apply these protocols in real networks *(6)*.

The dependability of networks is sensitive to the correct, reliable operation of key registries. The whole approach becomes unwieldy in large networks: Failures of registries can prevent principals from initiating new conversations and can compromise keys. Trust itself is a serious issue in a large network; the US and Soviet governments, for example, are not likely to believe that each other's registries will refrain from listening in on conversations for which they have passed out the keys.

The amount of faith required can be reduced by using public-key cryptography to exchange the private keys for conversations. Now the registry service becomes simply a directory service D. Principals can register public keys with D for later redistribution, but they do not need to reveal their secret keys to D. To converse with B, A consults D to obtain the public key PB, generates a conversation key K, and sends $[K]^{PB}$ to B with a request to open a conversation. A must also authenticate itself to B, which can be done with a certificate as discussed below. Now the responsibility for generating keys rests with the communicants, and the directory service has no special knowledge that would enable it to listen in on any conversations.

There is still a catch -- trusting the authenticity of public keys dispensed by the directory service or by any other principal. The authenticity of this information can be guaranteed by storing it as public-key certificates created, on request, by a network notary service. Certificates are messages of the form $[B, PB, T]^{SN}$, where SN is the secret key of the notary service and T is the time of the certificate's creation. Anyone can decipher a certificate using the notary's public key, thereby obtaining the public key of the principal identified therein. If for some reason the notary's secret key is compromised, all subsequently issued certificates are invalid. A good deal of effort must

be put into protecting the notary's secret key, but the effort is worthwhile because the security of network communications does not rest on the trustworthiness of the directory service *(4)*.

The safety of remotely controlled experiments and integrity of research rest critically on the ability of the network to authenticate senders and receivers, to protect proprietary communications, and to sign some transmissions. Mathematically sound schemes for encrypting data and distributing keys make security an attainable goal.

References

1. Dorothy E. Denning. 1982. *Cryptography and Data Security*. Addison-Wesley. (Especially Chapters 1 and 2.)
2. Ronald Rivest, Adi Shamir, and Len Adleman. 1978. "A method for obtaining digital signatures and public-key cryptosystems". ACM *Communications*, Vol. 21, February.
3. Donald W. Davies. 1983. "Applying the RSA digital signature to electronic mail." IEEE *Computer*. February.
4. Dorothy E. Denning. 1983. "Protecting public keys and signature keys." IEEE *Computer*. February.
5. Roger M. Needham and Michael D. Schroeder. 1978. "Using encryption for authentication in large networks of computers." ACM *Communications*, Vol. 21, December.
6. Victor L. Voydock and Stephen T. Kent. 1983. "Security mechanisms in high-level network protocols." ACM *Computing Surveys*, Vol 15, June.

Baffling Big Brother

American Scientist 1987, No 5

Smart cards and cryptography enable a new system of business transactions that balances the power of individuals to control how information about them is linked and disseminated against the need of organizations to be certain that credentials and payments are valid. Each person uses a different name (pseudonym) with each organization and a personal card computer manages all the names and cryptographic protocols. The most difficult protocols are for credentials and payments. Even though we complain about a world in which our transactions are too easily traced, would we want a world in which none of our transactions could be traced?

Who hasn't asked whether large organizations will one day be able to use computers to monitor every detail of people's lives? Who doesn't occasionally wonder whether our high-tech society is moving inexorably toward dossiers, surveillance, scrutiny of private lives, and complete distrust of individuals? Who hasn't asked whether anything can be done about these trends?

Commercial transactions began to be computerized in the 1950s. Today, most businesses entrust valuable information assets to electronic media, using them to store records, compute accounts, transfer funds, and generate receipts.

Two trends have accompanied this widening use of computers. The rate of computer abuse has risen in direct proportion to the value of information assets and the expertise of users, and the existence of databases has created incentives to link the data they contain. To protect against abuse, organizations demand personal information from customers for checking credentials; they keep confidential files on customer activities, payments, and credit histories, frequently making it difficult for customers and employees to review or correct information in those files. On the other hand, they often sell or distribute information about their clients to other organizations. Government agencies have begun to link information in their own and some private databases in their efforts to detect persons who are violating the law. What emerges is a one-sided arrangement: most of the power to control information lies in the hands of organizations and agencies. As a result, calls for legislation to protect individuals from abuse or mistakes are increasingly heard.

Technology has been blamed for the gradual weakening of the individual's power; but can technology strengthen the individual? The answer is yes. Card computers -- the so-called smart cards -- and public-key cryptography can be used to construct a system of business transactions in which organizations have absolute assurance that all credentials and payment orders are valid and individuals have absolute assurance that no group of organizations can compile dossiers about them by linking databases.

David Chaum of the Centre for Mathematics and Computer Science in Amsterdam has proposed a system of transactions that allows ordinary communications, payments, and credentials to be exchanged electronically *(1,2)*. In Chaum's system, it impossible for the records of various organizations to be linked or traced to a specific individual; individuals retain control over how information about them is used. The system also protects the organizations against abuse -- perhaps even better than current systems.

Chaum's starting point is the possibility that a person can create a different name -- a pseudonym -- for use with each organization. This is analogous to supplying each bank, store, service establishment, or other organization with a different name, postal address, and identification number. A card computer is needed to assist the person to make transactions and keep track of which name is used with which organization. To guarantee that they cannot be linked between organization, names are actually long random numbers generated by the card computer. Public key cryptosystems are used for communications and digital signatures.

Let me digress for a brief review of public key cryptosystems, which I described more fully in the January-February issue *(3)*. Associated with a name (or pseudonym) A are two complementary keys chosen by A. The public key PA is used to encipher messages intended for A, and the secret key SA is used by A to decipher messages. The secret key cannot be deduced from the public key. To insure their security, the keys must contain about 200 digits (approximately 665 bits). The encipherment of an item Z under a key K is denoted $[Z]^K$. A public key is sealed in a key certificate

$$K(A) = [A, PA, D]^{SN},$$

where SN is the secret key of a trusted notary and D is the date and time of the notary's signature. Anyone receiving $K(A)$ can unseal it by computing $[K(A)]^{PN}$, because the public and private keys cancel; that recipient can have confidence that PA is A's public key because only the notary could have sealed the certificate. A message M from A to B is encoded as

$$[M, K(A)]^{PB}.$$

Only B can decipher this message; B can reply to A by using the key enclosed with the message. A block of the form $[\text{checksum}(M)]^{SA}$ can be attached to the message to serve as a digital signature of the sender.

With cryptography, I can prevent organizations from linking records about me. I simply generate a separate random number for each organization with which I deal and use the numbers as pseudonyms in transactions with those organizations. It is impossible for the organizations to link my assumed names because the connection between the names is known only to me. Of course, if I reveal personal information in my messages, organizations may be able to link message files under different pseudonyms by comparing their contents. The assignment of separate pseudonyms guarantees only that messages cannot be linked by using information in their headers and address fields.

Many business transactions depend on credentials, special tamperproof certifications that a specific person is authorized or qualified to do something. Examples are driver's licenses, passports, and traveler's checks. A credential issued by X can be

represented electronically by a cryptogram

$$[A \text{ authorized for } T]^{SX}.$$

Anyone can check that A has authorization for the specific transaction T by unsealing the credential with the issuer's public key. (A receiver of the above credential may demand proof that its bearer is in fact A; the important authentication protocols that accomplish this are not covered here.)

Credentials of this sort do not work with a multiple pseudonym system. The problem is that the pseudonym by which the person is known to the issuer is sealed inside the credential. Thus, if I obtain a credential from a bank under name A certifying me for $1000 in credit, a store knowing me as B will not honor that credential. What is needed is a way to transform a credential issued under one pseudonym into a valid credential under a different pseudonym, without restricting my choice of pseudonyms.

Chaum has devised a method of accomplishing this seemingly impossible task. The idea is that a name has a special form, consisting of a fixed part uniquely associated with the person multiplied by a variable part that depends on the particular organization with which that name is used. The unique part can be obtained from a special registrar that associates a unique identifier with a standard item of personal information such as a thumbprint; a person need give no other information than this to the registrar. Because a person's unique identifier is a hidden, multiplicative component of a name, no one else can learn it; in particular, the registrar will not be able to link the information it has -- unique identifiers plus thumbprints -- with any information held by another organization.

Within this scheme a challenge protocol is needed to permit a third party to determine unequivocally that two pseudonyms belong to the same person. If some organization challenges my claim that the pseudonyms A and B are both mine, both the claim and the challenge can be submitted to an arbiter who can verify that the claim is true or false without revealing my unique identifier. Protocols for doing this are beyond the discussion here. I'll assume that the notary can serve this function.

To illustrate Chaum's method, suppose that I want to request a credential of issuer X under pseudonym A and present it to credential user Y under pseudonym B. Associated with a credential of type T are secret and public keys, ST and PT, the secret key being known only to the credential issuer X. Let U denote my unique identifier, and let V and W denote random numbers I generate. From these numbers I create two special pseudonyms for this transaction,

$$\begin{aligned} a &= U[V]^{PT} \\ b &= U[W]^{PT} \end{aligned}$$

and have them signed by the notary, who returns them in the key certificates $K(a)$ and $K(b)$. My request to X for a credential takes the form

$$[\text{"Request } T\text{"}, K(a), K(A)]^{PX}.$$

On receipt, X can check that as A I am authorized for T and return the credential in the form

$$[[a]^{ST}, K(T)]^{PA}.$$

I can check that the credential comes from X and agrees with my request by unsealing it with PT and checking that the result is a. To generate the credential for my pseudonym B, all I need to do is divide the credential by V and multiply by W. This works because $[a]^{ST} = [U[V]^{PT}]^{ST} = [U]^{ST}V$, and similarly $[b]^{ST} = [U]^{ST}W$. I pass the result to Y in the form

$$[\text{"Claim } T\text{"}, [b]^{ST}, K(b), K(T), K(B)]^{PY}.$$

Y can check the claim by unsealing the credential with PT and checking that the result is b. Organization X can ask the notary to certify that the pair $K(a)$ and $K(A)$ are both mine and Y can do the same for $K(b)$ and $K(B)$. Thus the challenge protocol can be used by the organizations to assure themselves that I am not submitting special pseudonyms and credentials belonging to another person.

Can this scheme be used for payments? The obvious approach is to extend credentials into electronic bank checks: I obtain from my bank a credential that says "A is good for $\$T$"; then I transform it as above to a credential that says "B is good for $\$T$", which I pass to Y in payment of a bill. Unfortunately, an electronic bank check is susceptible to tracing. Suppose I request a check for an unusual sum, say $385, and a short time later Y deposits the same sum; by matching withdrawals and deposits, the bank can to link my pseudonyms A and B.

To avoid this kind of tracing, Chaum proposes instead to use electronic currency. When I make a withdrawal, the bank will return a set of certificates of standard denominations; thus my payment of $385 might consist of three $100 certificates, eight $10 certificates, and five $1 certificates. Unlike paper certificates, electronic ones are easy to copy, so it is necessary to include a method that permits the bank to recognize the first copy of a currency certificate. The obvious method, allowing the bank to attach a note number to the original certificate, will not work because the note number would permit the bank to link my pseudonyms A and B. What is needed instead is a way that I can provide a note number that the bank cannot associate with me, and seal that note number in the credential.

One way to accomplish this is the following. Suppose k binary bits are allocated for names. I will exploit the fact that if I string two copies of the k-bit name n together, I create a binary string of $2k$ bits, representing the number $n2^k+n$. To initiate a payment of denomination T, I choose a random note number N and hide it in the name $M=(N2^k+N)[V]^{PT}$. My request to the bank takes the form

$$[\text{"Request denomination } T\text{"}, M, K(A)]^{PX}.$$

after deducting $\$T$ from my account, the bank returns the certificate

$$[[M]^{ST}, K(T)]^{PA}.$$

When I divide the certificate by V, I transform it automatically to $[N2^k+N]^{ST}$. I can submit the transformed certificate to the store, which can unseal it with PT and verify

from its two-copy structure that it is worth $\$T$. In turn the store can submit the certificate to the bank, which can extract the note number and deposit $\$T$ in the store's account if that note number has not been seen before.

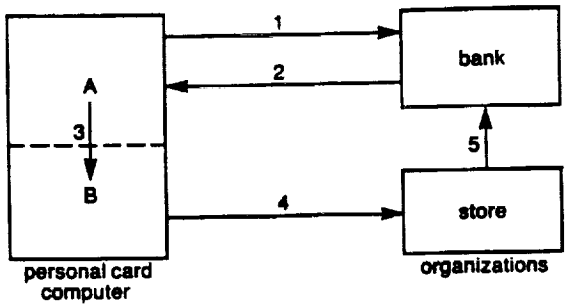
Chaum's system of transactions is different from current systems in three principal ways. First, a person can use a different pseudonym, a random number, for each organization. Current systems are based on universal identifiers. Second, a person can use a card computer to manage interactions with organizations under each pseudonym, to generate random numbers for use in names, and to carry out the cryptographic protocols. A card computer need have no secrets from its owner. Current systems rely on organizations giving customers cards that contain secret patterns known to the organization but not to the card holders. Third, individuals get to control how information about them is distributed and linked. Current systems are one-sided, giving organizations most of the power to protect themselves from abusive customers while giving customers little power to protect themselves from abusive organizations.

So a system of untraceable business transactions is technically feasible. But is such a system politically feasible? Even though we complain about a world in which our transactions are too easily traced, would we want a world in which none of our transactions could be traced?

References

1. D. Chaum. 1985. "Security without identification: Transaction systems to make Big Brother obsolete." ACM *Communications 28*, 10, October. 1030-1044.
2. D. Chaum. 1987. "A secure and privacy-protecting protocol for transmitting personal information between organizations." *Proc. CRYPTO 86* (A. Odlyzko, Ed.). Lecture Notes in Computer Science. Springer-Verlag.
3. P. Denning. 1987. "Security of data in networks." *American Scientist 75*, 1. 12-14.

How smart cards can help



Smart cards and cryptography make possible a new system of business transactions that balances the interests of individuals and organizations. Individuals can control how information about them is linked and disseminated, and organizations can be certain that credentials and payments are valid. Each person uses a different name—a pseudonym—with each organization, and a personal card computer manages all the names and cryptographic protocols. The diagram shows the interactions needed in a payment transaction. Under pseudonym *A*, the user requests a payment credential of denomination $*T* from the bank (1), which deducts the money from *A*'s account and issues the credential (2). The credential is a cryptogram enciphered with the denomination's secret key; anyone can read it by deciphering with the denomination's public key. Using a simple transformation, the card computer converts the credential to be valid for pseudonym *B* without altering the declaration that the bank is willing to pay the holder $*T* (3). The result is forwarded to a store (4) in payment of a bill, and the store can in turn forward the payment to the bank for deposit to its account (5). Pseudonyms consist of a unique identifier multiplied by a random number chosen by the person; the conversion at step 3 is simply a replacement of the random number corresponding to the bank with the random number corresponding to the store. A note number must be included in the credential so that the bank can recognize duplicates (and not pay them).