A Tour of the Worm

Donn Seeley

Department of Computer Science
University of Utah

ABSTRACT

On the evening of November 2, 1988, a self-replicating program was released upon the Internet[1]. This program (a *worm*) invaded VAX and Sun-3 computers running versions of Berkeley UNIX, and used their resources to attack still more computers[2]. Within the space of hours this program had spread across the U.S., infecting hundreds or thousands of computers and making many of them unusable due to the burden of its activity. This paper provides a chronology for the outbreak and presents a detailed description of the internals of the worm, based on a C version produced by decompiling.

1. Introduction

> There is a fine line between helping administrators protect their systems and providing a cookbook for bad guys. [Grampp and Morris, ‘‘UNIX Operating System Security’’]

November 3, 1988 is already coming to be known as Black Thursday. System administrators around the country came to work on that day and discovered that their networks of computers were laboring under a huge load. If they were able to log in and generate a system status listing, they saw what appeared to be dozens or hundreds of ‘‘shell’’ (command interpreter) processes. If they tried to kill the processes, they found that new processes appeared faster than they could kill them. Rebooting the computer seemed to have no effect—within minutes after starting up again, the machine was overloaded by these mysterious processes.

These systems had been invaded by a *worm*. A worm is a program that propagates itself across a network, using resources on one machine to attack other machines. (A worm is not quite the same as a *virus*, which is a program fragment that inserts itself into other programs.) The worm had taken advantage of lapses in security on systems that were running 4.2 or 4.3 BSD UNIX or derivatives like SunOS. These lapses allowed it to connect to machines across a network, bypass their login authentication, copy itself and then proceed to attack still more machines. The massive system load was generated by multitudes of worms trying to propagate the epidemic.

The Internet had never been attacked in this way before, although there had been plenty of speculation that an attack was in store. Most system administrators were unfamiliar with the concept of worms (as opposed to viruses, which are a major affliction of the PC world) and it took some time before they were able to establish what was going on and how to deal with it. This paper is intended to let people know exactly what happened and how it came about, so that they will be better prepared when it happens the next time. The behavior of the worm will be examined in detail, both to show exactly what it did and didn’t do, and to show the dangers of future worms. The epigraph above is now ironic, for the author of the worm used information in that paper to attack systems. Since the information is now well known, by virtue of the fact that thousands of computers now have copies of the worm, it seems unlikely that this paper can do similar damage, but it is definitely a troubling thought. Opinions on this and other matters will be offered below.

---

[1] The Internet is a logical network made up of many physical networks, all running the IP class of network protocols.

[2] VAX and Sun-3 are models of computers built by Digital Equipment Corp. and Sun Microsystems Inc., respectively. UNIX is a Registered Bell of AT&T Trademark Laboratories.

2. Chronology

> Remember, when you connect with another computer, you're connecting to every computer that computer has connected to. [Dennis Miller, on NBC's *Saturday Night Live*]

> Here is the gist of a message I got: I'm sorry. [Andy Sudduth, in an anonymous posting to the TCP-IP list on behalf of the author of the worm, 11/3/88]

Many details of the chronology of the attack are not yet available. The following list represents dates and times that we are currently aware of. Times have all been rendered in Pacific Standard Time for convenience.

11/2: 1800 (approx.)
This date and time were seen on worm files found on *prep.ai.mit.edu*, a VAX 11/750 at the MIT Artificial Intelligence Laboratory. The files were removed later, and the precise time was lost. System logging on *prep* had been broken for two weeks. The system doesn't run accounting and the disks aren't backed up to tape: a perfect target. A number of ''tourist'' users (individuals using public accounts) were reported to be active that evening. These users would have appeared in the session logging, but see below.

11/2: 1824 First known West Coast infection: *rand.org* at Rand Corp. in Santa Monica.

11/2: 1904 *csgw.berkeley.edu* is infected. This machine is a major network gateway at UC Berkeley. Mike Karels and Phil Lapsley discover the infection shortly afterward.

11/2: 1954 *mimsy.umd.edu* is attacked through its *finger* server. This machine is at the University of Maryland College Park Computer Science Department.

11/2: 2000 (approx.)
Suns at the MIT AI Lab are attacked.

11/2: 2028 First *sendmail* attack on mimsy.

11/2: 2040 Berkeley staff figure out the *sendmail* and *rsh* attacks, notice *telnet* and *finger* peculiarities, and start shutting these services off.

11/2: 2049 *cs.utah.edu* is infected. This VAX 8600 is the central Computer Science Department machine at the University of Utah. The next several entries follow documented events at Utah and are representative of other infections around the country.

11/2: 2109 First *sendmail* attack at *cs.utah.edu*.

11/2: 2121 The load average on *cs.utah.edu* reaches 5. The ''load average'' is a system-generated value that represents the average number of jobs in the run queue over the last minute; a load of 5 on a VAX 8600 noticeably degrades response times, while a load over 20 is a drastic degradation. At 9 PM, the load is typically between 0.5 and 2.

11/2: 2141 The load average on *cs.utah.edu* reaches 7.

11/2: 2201 The load average on *cs.utah.edu* reaches 16.

11/2: 2206 The maximum number of distinct runnable processes (100) is reached on *cs.utah.edu*; the system is unusable.

11/2: 2220 Jeff Forys at Utah kills off worms on *cs.utah.edu*. Utah Sun clusters are infected.

11/2: 2241 Re-infestation causes the load average to reach 27 on *cs.utah.edu*.

11/2: 2249 Forys shuts down *cs.utah.edu*.

11/3: 2321 Re-infestation causes the load average to reach 37 on *cs.utah.edu*, despite continuous efforts by Forys to kill worms.

11/2: 2328 Peter Yee at NASA Ames Research Center posts a warning to the TCP-IP mailing list: ''We are currently under attack from an Internet VIRUS. It has hit UC Berkeley, UC San Diego, Lawrence Livermore, Stanford, and NASA Ames.'' He suggests turning off *telnet*, *ftp*, *finger*, *rsh* and SMTP services. He does not mention *rexec*. Yee is actually at Berkeley working with Keith Bostic, Mike Karels and Phil Lapsley.

11/3: 0034 At another's prompting, Andy Sudduth of Harvard anonymously posts a warning to the TCP-IP list: ''There may be a virus loose on the internet.'' This is the first message that (briefly) describes how the *finger* attack works, describes how to defeat the SMTP attack by rebuilding *sendmail*, and explicitly mentions the *rexec* attack. Unfortunately Sudduth's message is blocked at *relay.cs.net* while that gateway is shut down to combat the worm, and it does not get delivered for almost two days. Sudduth acknowledges authorship of the message in a subsequent message to TCP-IP on Nov. 5.

11/3: 0254 Keith Bostic sends a fix for *sendmail* to the newsgroup comp.bugs.4bsd.ucb-fixes and to the TCP-IP mailing list. These fixes (and later ones) are also mailed directly to important system administrators around the country.

11/3: early morning
The *wtmp* session log is mysteriously removed on *prep.ai.mit.edu*.

11/3: 0507 Edward Wang at Berkeley figures out and reports the *finger* attack, but his message doesn't come to Mike Karels' attention for 12 hours.

11/3: 0900 The annual Berkeley Unix Workshop commences at UC Berkeley. 40 or so important system administrators and hackers are in town to attend, while disaster erupts at home. Several people who had planned to fly in on Thursday morning are trapped by the crisis. Keith Bostic spends much of the day on the phone at the Computer Systems Research Group offices answering calls from panicked system administrators from around the country.

11/3: 1500 (approx.)
The team at MIT Athena calls Berkeley with an example of how the *finger* server bug works.

11/3: 1626 Dave Pare arrives at Berkeley CSRG offices; disassembly and decompiling start shortly afterward using Pare's special tools.

11/3: 1800 (approx.)
The Berkeley group sends out for calzones. People arrive and leave; the offices are crowded, there's plenty of excitement. Parallel work is in progress at MIT Athena; the two groups swap code.

11/3: 1918 Keith Bostic posts a fix for the *finger* server.

11/4: 0600 Members of the Berkeley team, with the worm almost completely disassembled and largely decompiled, finally take off for a couple hours' sleep before returning to the workshop.

11/4: 1236 Theodore Ts'o of Project Athena at MIT publicly announces that MIT and Berkeley have completely disassembled the worm.

11/4: 1700 (approx.)
A short presentation on the worm is made at the end of the Berkeley UNIX Workshop.

11/8: National Computer Security Center meeting to discuss the worm. There are about 50 attendees.

11/11: 0038 Fully decompiled and commented worm source is installed at Berkeley.

3. Overview

What exactly did the worm do that led it to cause an epidemic? The worm consists of a 99-line bootstrap program written in the C language, plus a large relocatable object file that comes in VAX and Sun-3 flavors. Internal evidence showed that the object file was generated from C sources, so it was natural to decompile the binary machine language into C; we now have over 3200 lines of commented C code which recompiles and is mostly complete. We shall start the tour of the worm with a quick overview of the basic goals of the worm, followed by discussion in depth of the worm's various behaviors as revealed by decompilation.

The activities of the worm break down into the categories of attack and defense. Attack consists of locating hosts (and accounts) to penetrate, then exploiting security holes on remote systems to pass across a copy of the worm and run it. The worm obtains host addresses by examining the system tables */etc/hosts.equiv* and */.rhosts*, user files like *.forward* and *.rhosts*, dynamic routing information produced by the *netstat* program, and finally randomly generated host addresses on local networks. It ranks these by order of preference, trying a file like */etc/hosts.equiv* first because it contains names of local machines that are likely to permit unauthenticated connections. Penetration of a remote system can be accomplished in any of three ways. The worm can take advantage of a bug in the *finger* server that allows it to download code in place of a finger request and trick the server into executing it. The worm can use a ''trap door'' in the *sendmail* SMTP mail service, exercising a bug in the debugging code that allows it to execute a command interpreter and download code across a mail connection. If the worm can penetrate a local account by guessing its password, it can use the *rexec* and *rsh* remote command interpreter services to attack hosts that share that account. In each case the worm arranges to get a remote command interpreter which it can use to copy over, compile and execute the 99-line bootstrap. The bootstrap sets up its own network connection with the local worm and copies over the other files it needs, and using these pieces a remote worm is built and the infection procedure starts over again.

Defense tactics fall into three categories: preventing the detection of intrusion, inhibiting the analysis of the program, and authenticating other worms. The worm's simplest means of hiding itself is to change its name. When it starts up, it clears its argument list and sets its zeroth argument to *sh*, allowing it to masquerade as an innocuous command interpreter. It uses *fork*() to change its process I.D., never staying too long at one I.D. These two tactics are intended to disguise the worm's presence on system status listings. The worm tries to leave as little trash lying around as it can, so at start-up it reads all its support files into memory and deletes the tell-tale filesystem copies. It turns off the generation of *core* files, so if the worm makes a mistake, it doesn't leave evidence behind in the form of core dumps. The latter tactic is also designed to block analysis of the program—it prevents an administrator from sending a software signal to the worm to force it to dump a core file. There are other ways to get a core file, however, so the worm carefully alters character data in memory to prevent it from being extracted easily. Copies of disk files are encoded by repeatedly exclusive-or'ing a ten-byte code sequence; static strings are encoded byte-by-byte by exclusive-or'ing with the hexadecimal value 81, except for a private word list which is encoded with hexadecimal 80 instead. If the worm's files are somehow captured before the worm can delete them, the object files have been loaded in such a way as to remove most non-essential symbol table entries, making it harder to guess at the purposes of worm routines from their names. The worm also makes a trivial effort to stop other programs from taking advantage of its communications; in theory a well-prepared site could prevent infection by sending messages to ports that the worm was listening on, so the worm is careful to test connections using a short exchange of random ''magic numbers''.

When studying a tricky program like this, it's just as important to establish what the program *does not* do as what it does do. The worm *does not delete a system's files*: it only removes files that it created in the process of bootstrapping. The program does not attempt to incapacitate a system by deleting important files, or indeed any files. It does not remove log files or otherwise interfere with normal operation other than by consuming system resources. The worm *does not modify existing files*: it is not a virus. The worm propagates by copying itself and compiling itself on each system; it does not modify other programs to do its work for it. Due to its method of infection, it can't count on sufficient privileges to be able to modify programs. The worm *does not install trojan horses*: its method of attack is strictly active, it never waits for a user to trip over a trap. Part of the reason for this is that the worm can't afford to waste time waiting for trojan horses—it must reproduce before it is discovered. Finally, the worm *does not record or transmit decrypted passwords*: except for its own static list of favorite passwords, the worm does not propagate cracked passwords on to new worms nor does it transmit them back to some home base. This is not to say that the accounts that the worm penetrated are secure merely because the worm did not tell anyone what their passwords were, of course—if the worm can guess an account's password, certainly others can too. The worm *does not try to capture superuser privileges*: while it does try to break into accounts, it doesn't depend on having particular privileges to propagate, and never makes special use of such privileges if it somehow gets them. The worm *does not propagate over uucp* or X.25 or DECNET or BITNET: it specifically requires TCP/IP. The worm *does not infect System V systems* unless they have been modified to use Berkeley network programs like *sendmail*, *fingerd* and *rexec*.

4. Internals

Now for some details: we shall follow the main thread of control in the worm, then examine some of the worm's data structures before working through each phase of activity.

4.1. The thread of control

When the worm starts executing in *main*(), it takes care of some initializations, some defense and some cleanup. The very first thing it does is to change its name to *sh*. This shrinks the window during which the worm is visible in a system status listing as a process with an odd name like *x9834753*. It then initializes the random number generator, seeding it with the current time, turns off core dumps, and arranges to die when remote connections fail. With this out of the way, the worm processes its argument list. It first looks for an option **–p** *$$*, where *$$* represents the process I.D. of its parent process; this option indicates to the worm that it must take care to clean up after itself. It proceeds to read in each of the files it was given as arguments; if cleaning up, it removes each file after it reads it. If the worm wasn't given the bootstrap source file *l1.c* as an argument, it exits silently; this is perhaps intended to slow down people who are experimenting with the worm. If cleaning up, the worm then closes its file descriptors, temporarily cutting itself off from its remote parent worm, and removes some files. (One of these files, */tmp/.dumb*, is never created by the worm and the unlinking seems to be left over from an earlier stage of development.) The worm then zeroes out its argument list, again to foil the system status program *ps*. The next step is to initialize the worm's list of network interfaces; these interfaces are used to find local networks and to check for alternate addresses of the current host. Finally, if cleaning up the worm resets its process group and kills the process that helped to bootstrap it. The worm's last act in *main*() is to call a function we named *doit*(), which contains the main loop of the worm.

doit() runs a short prologue before actually starting the main loop. It (redundantly) seeds the random number generator with the current time, saving the time so that it can tell how long it has been running. The worm then attempts its first infection. It initially attacks gateways that it found with the *netstat* network status program; if it can't infect one of these hosts, then it checks random host numbers on local networks, then it tries random host numbers on networks that are on the far side of gateways, in each case stopping if it succeeds. (Note that this sequence of attacks differs from the sequence the worm uses after it has entered the main loop.)

After this initial attempt at infection, the worm calls the routine *checkother*() to check for another worm already on the local machine. In this check the worm acts as a client to an existing worm which acts as a server; they may exchange ''population control'' messages, after which one of the two worms will eventually shut down.

One odd routine is called just before entering the main loop. We named this routine *send_message*(), but it really doesn't send anything at all. It looks like it was intended to cause 1 in 15 copies of the worm to send a 1-byte datagram to a port on the host *ernie.berkeley.edu*, which is located in the Computer Science Department at UC Berkeley. It has been suggested that this was a feint, designed to draw attention to *ernie* and away from the author's

```
doit() {
    seed the random number generator with the time
    attack hosts: gateways, local nets, remote nets
    checkother();
    send_message();
    for (;;) {
        cracksome();
        other_sleep(30);
        cracksome();
        change our process ID
        attack hosts: gateways, known hosts,
            remote nets, local nets
        other_sleep(120);
        if 12 hours have passed,
            reset hosts table
        if (pleasequit && nextw > 10)
            exit(0);
    }
}
```

''C'' pseudo-code for the *doit*() function

real host. Since the routine has a bug (it sets up a TCP socket but tries to send a UDP packet), nothing gets sent at all. It's possible that this was a deeper feint, designed to be uncovered only by decompilers; if so, this wouldn't be the only deliberate impediment that the author put in our way. In any case, administrators at Berkeley never detected any process listening at port 11357 on ernie, and we found no code in the worm that listens at that port, regardless of the host.

The main loop begins with a call to a function named *cracksome*() for some password cracking. Password cracking is an activity that the worm is constantly working at in an incremental fashion. It takes a break for 30 seconds to look for intruding copies of the worm on the local host, and then goes back to cracking. After this session, it forks (creates a new process running with a copy of the same image) and the old process exits; this serves to turn over process I.D. numbers and makes it harder to track the worm with the system status program *ps*. At this point the worm goes back to its infectious stage, trying (in order of preference) gateways, hosts listed in system tables like */etc/hosts.equiv*, random host numbers on the far side of gateways and random hosts on local networks. As before, if it succeeds in infecting a new host, it marks that host in a list and leaves the infection phase for the time being. After infection, the worm spends two minutes looking for new local copies of the worm again; this is done here because a newly infected remote host may try to reinfect the local host. If 12 hours have passed and the worm is still alive, it assumes that it has had bad luck due to networks or hosts being down, and it reinitializes its table of hosts so that it can start over from scratch. At the end of the main loop the worm checks to see if it is scheduled to die as a result of its population control features, and if it is, and if it has done a sufficient amount of work cracking passwords, it exits.

4.2. Data structures

The worm maintains at least four interesting data structures, and each is associated with a set of support routines.

The *object* structure is used to hold copies of files. Files are encrypted using the function *xorbuf*() while in memory, so that dumps of the worm won't reveal anything interesting. The files are copied to disk on a remote system before starting a new worm, and new worms read the files into memory and delete the disk copies as part of their start-up duties. Each structure contains a name, a length and a pointer to a buffer. The function *getobjectbyname*() retrieves a pointer to a named object structure; for some reason, it is only used to call up the bootstrap source file.

The *interface* structure contains information about the current host's network interfaces. This is mainly used to check for local attached networks. It contains a name, a network address, a subnet mask and some flags. The interface table is initialized once at start-up time.

The *host* structure is used to keep track of the status and addresses of hosts. Hosts are added to this list dynamically, as the worm encounters new sources of host names and addresses. The list can be searched for a particular address or name, with an option to insert a new entry if no matching entry is found. Flag bits are used to indicate whether the host is a gateway, whether it was found in a system table like */etc/hosts.equiv*, whether the worm has found it impossible to attack the host for some reason, and whether the host has already been successfully infected. The bits for ''can't infect'' and ''infected'' are cleared every 12 hours, and low priority hosts are deleted, to be accumulated again later. The structure contains up to 12 names (aliases) and up to 6 distinct network addresses for each host.

In our sources, what we've called the *muck* structure is used to keep track of accounts for the purpose of password cracking. (It was awarded the name *muck* for sentimental reasons, although *pw* or *acct* might be more mnemonic.) Each structure contains an account name, an encrypted password, a decrypted password (if available) plus the home directory and personal information fields from the password file.

4.3. Population growth

The worm contains a mechanism that seems to be designed to limit the number of copies of the worm running on a given system, but beyond that our current understanding of the design goals is itself limited. It clearly does not prevent a system from being overloaded, although it does appear to pace the infection so that early copies can go undetected. It has been suggested that a simulation of the worm's population control features might reveal more about its design, and we are interested writing such a simulation.

The worm uses a client-and-server technique to control the number of copies executing on the current machine. A routine *checkother*() is run at start-up time. This function tries to connect to a server listening at TCP port 23357. The connection attempt returns immediately if no server is present, but blocks if one is available and busy; a server worm periodically runs its server code during time-consuming operations so that the queue of connections does not grow large. After the client exchanges magic numbers with the server as a trivial form of authentication, the client and the server roll dice to see who gets to survive. If the exclusive-or of the respective low bits of the client's and the server's random numbers is 1, the server wins, otherwise the client wins. The loser sets a flag *pleasequit* that eventually allows it to exit at the bottom of the main loop. If at any time a problem occurs—a read from the server fails, or the wrong magic number is returned—the client worm returns from the function, becoming a worm that never acts as a server and hence does not engage in population control. Perhaps as a precaution against a cataleptic server, a test at the top of the function causes 1 in 7 worms to skip population control. Thus the worm finishes the population game in *checkother*() in one of three states: scheduled to die after some time, with *pleasequit* set; running as a server, with the possibility of losing the game later; and immortal, safe from the gamble of population control.

A complementary routine *other_sleep*() executes the server function. It is passed a time in seconds, and it uses the Berkeley *select*() system call to wait for that amount of time accepting connections from clients. On entry to the function, it tests to see whether it has a communications port with which to accept connections; if not, it simply sleeps for the specified amount of time and returns. Otherwise it loops on *select*(), decrementing its time remaining after serving a client until no more time is left and the function returns. When the server acquires a client, it performs the inverse of the client's protocol, eventually deciding whether to proceed or to quit. *other_sleep*() is called from many different places in the code, so that clients are not kept waiting too long.

Given the worm's elaborate scheme for controlling re-infection, what led it to reproduce so quickly on an individual machine that it could swamp it? One culprit is the 1 in 7 test in *checkother*(): worms that skip the client phase become immortal, and thus don't risk being eliminated by a roll of the dice. Another source of system loading is the problem that when a worm decides it has lost, it can still do a lot of work before it actually exits. The client routine isn't even run until the newly born worm has attempted to infect at least one remote host, and even if a worm loses the roll, it continues executing to the bottom of the main loop, and even then it won't exit unless it has gone through the main loop several times, limited by its progress in cracking passwords. Finally, new worms lose all of the history of infection that their parents had, so the children of a worm are constantly trying to re-infect the parent's host, as well as the   the check of hosts from system files should be sufficient to cover all or nearly all of the local hosts anyway.

Password cracking is another generator of host names, but since this is handled separately from the usual host attack scheme presented here, it will be discussed below with the other material on passwords.

4.5. Security holes

> The first fact to face is that Unix was not developed with security, in any realistic sense, in mind... [Dennis Ritchie, ‘‘On the Security of Unix’’]

This section discusses the TCP services used by the worm to penetrate systems. It’s a touch unfair to use the quote above when the implementation of the services we’re about to discuss was distributed by Berkeley rather than Bell Labs, but the sentiment is appropriate. For a long time the balance between security and convenience on Unix systems has been tilted in favor of convenience. As Brian Reid has said about the break-in at Stanford two years ago: ‘‘Programmer convenience is the antithesis of security, because it is going to become intruder convenience if the programmer’s account is ever compromised.’’ The lesson from that experience seems to have been forgotten by most people, but not by the author of the worm.

4.5.1. Rsh and rexec

> These notes describe how the design of TCP/IP and the 4.2BSD implementation allow users on untrusted and possibly very distant hosts to masquerade as users on trusted hosts. [Robert T. Morris, ‘‘A Weakness in the 4.2BSD Unix TCP/IP Software’’]

Rsh and *rexec* are network services which offer remote command interpreters. *Rexec* uses password authentication; *rsh* relies on a ‘‘privileged’’ originating port and permissions files. Two vulnerabilities are exploited by the worm—the likelihood that a remote machine that has an account for a local user will have the same password as the local account, allowing penetration through *rexec*, and the likelihood that such a remote account will include the local host in its *rsh* permissions files. Both of these vulnerabilities are really problems with laxness or convenience for users and system administrators rather than actual bugs, but they represent avenues for infection just like inadvertent security bugs.

The first use of *rsh* by the worm is fairly simple: it looks for a remote account with the same name as the one that is (unsuspectingly) running the worm on the local machine. This test is part of the standard menu of hacks conducted for each host; if it fails, the worm falls back upon *finger*, then *sendmail*. Many sites, including Utah, already were protected from this trivial attack by not providing remote shells for pseudo-users like *daemon* or *nobody*.

A more sophisticated use of these services is found in the password cracking routines. After a password is successfully guessed, the worm immediately tries to penetrate remote hosts associated with the broken account. It reads the user’s *.forward* file (which contains an address to which mail is forwarded) and *.rhosts* file (which contains a list of hosts and optionally user names on those hosts which are granted permission to access the local machine with *rsh* bypassing the usual password authentication), trying these hostnames until it succeeds. Each target host is attacked in two ways. The worm first contacts the remote host’s *rexec* server and sends it the account name found in the *.forward* or *.rhosts* files followed by the guessed password. If this fails, the worm connects to the local *rexec* server with the local account name and uses that to contact the target’s *rsh* server. The remote *rsh* server will permit the connection provided the name of the local host appears in either the */etc/hosts.equiv* file or the user’s private *.rhosts* file.

Strengthening these network services is far more problematic than fixing *finger* and *sendmail*, unfortunately. Users don’t like the inconvenience of typing their password when logging in on a trusted local host, and they don’t want to remember different passwords for each of the many hosts they may have to deal with. Some of the solutions may be worse than the disease—for example, a user who is forced to deal with many passwords is more likely to write them down somewhere.

4.5.2. Finger

> *gets* was removed from our [C library] a couple days ago. [Bill Cheswick at AT&T Bell Labs Research, private communication, 11/9/88]

Probably the neatest hack in the worm is its co-opting of the TCP *finger* service to gain entry to a system. *Finger* reports information about a user on a host, usually including things like the user’s full name, where their office is, the number of their phone extension and so on. The Berkeley[3] version of the *finger* server is a really trivial program: it reads a request from the originating host, then runs the local *finger* program with the request as an argument and ships the output back. Unfortunately the *finger* server reads the remote request with *gets*(), a standard C library routine that dates from the dawn of time and which does not check for overflow of the server's 512 byte request buffer on the stack. The worm supplies the finger server with a request that is 536 bytes long; the bulk of the request is some VAX machine code that asks the system to execute the command interpreter *sh*, and the extra 24 bytes represent just enough data to write over the server's stack frame for the main routine. When the main routine of the server exits, the calling function's program counter is supposed to be restored from the stack, but the worm wrote over this program counter with one that points to the VAX code in the request buffer. The program jumps to the worm's code and runs the command interpreter, which the worm uses to enter its bootstrap.

Not surprisingly, shortly after the worm was reported to use this feature of *gets*(), a number of people replaced all instances of *gets*() in system code with sensible code that checks the length of the buffer. Some even went so far as to remove *gets*() from the library, although the function is apparently mandated by the forthcoming ANSI C standard[4]. So far no one has claimed to have exercised the finger server bug before the worm incident, but in May 1988, students at UC Santa Cruz apparently penetrated security using a different finger server with a similar bug. The system administrator at UCSC noticed that the Berkeley finger server had a similar bug and sent mail to Berkeley, but the seriousness of the problem was not appreciated at the time (Jim Haynes, private communication).

One final note: the worm is meticulous in some areas but not in others. From what we can tell, there was no Sun-3 version of the *finger* intrusion even though the Sun-3 server was just as vulnerable as the VAX one. Perhaps the author had VAX sources available but not Sun sources?

4.5.3. Sendmail

> [T]he trap door resulted from two distinct 'features' that, although innocent by themselves, were deadly when combined (kind of like binary nerve gas). [Eric Allman, personal communication, 11/22/88]

The *sendmail* attack is perhaps the least preferred in the worm's arsenal, but in spite of that one site at Utah was subjected to nearly 150 *sendmail* attacks on Black Thursday. *Sendmail* is the program that provides the SMTP mail service on TCP networks for Berkeley UNIX systems. It uses a simple character-oriented protocol to accept mail from remote sites. One feature of *sendmail* is that it permits mail to be delivered to processes instead of mailbox files; this can be used with (say) the *vacation* program to notify senders that you are out of town and are temporarily unable to respond to their mail. Normally this feature is only available to recipients. Unfortunately a little loophole was accidentally created when a couple of earlier security bugs were being fixed—if *sendmail* is compiled with the *DEBUG* flag, and the sender at runtime asks that *sendmail* enter debug mode by sending the *debug* command, it permits senders to pass in a command sequence instead of a user name for a recipient. Alas, most versions of *sendmail* are compiled with *DEBUG*, including the one that Sun sends out in its binary distribution. The worm mimics a remote SMTP connection, feeding in */dev/null* as the name of the sender and a carefully crafted string as the recipient. The string sets up a command that deletes the header of the message and passes the body to a command interpreter. The body contains a copy of the worm bootstrap source plus commands to compile and run it. After the worm finishes the protocol and closes the connection to *sendmail*, the bootstrap will be built on the remote host and the local worm waits for its connection so that it can complete the process of building a new worm.

Of course this is not the first time that an inadvertent loophole or ''trap door'' like this has been found in sendmail, and it may not be the last. In his Turing Award lecture, Ken Thompson said: ''You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.)'' In fact, as Eric Allman says, ''[Y]ou can't even trust code that you did totally create yourself.'' The basic problem of trusting system programs is not one that is easy to solve.

4.6. Infection

The worm uses two favorite routines when it decides that it wants to infect a host. One routine that we named *infect*() is used from host scanning routines like *hg*(). *infect*() first checks that it isn't infecting the local machine, an already infected machine or a machine previously attacked but not successfully infected; the ''infected'' and

---

[3] Actually, like much of the code in the Berkeley distribution, the *finger* server was contributed from elsewhere; in this case, it appears that MIT was the source.

[4] See for example Appendix B, section 1.4 of the second edition of *The C Programming Language* by Kernighan and Ritchie.

‘‘immune’’ states are marked by flags on a host structure when attacks succeed or fail, respectively. The worm then makes sure that it can get an address for the target host, marking the host immune if it can’t. Then comes a series of attacks: first by *rsh* from the account that the worm is running under, then through *finger,* then through *sendmail*. If *infect*() fails, it marks the host as immune.

The other infection routine is named *hu1*() and it is run from the password cracking code after a password has been guessed. *hu1*(), like *infect*(), makes sure that it’s not re-infecting a host, then it checks for an address. If a potential remote user name is available from a *.forward* or *.rhosts* file, the worm checks it to make sure it is reasonable—it must contain no punctuation or control characters. If a remote user name is unavailable the worm uses the local user name. Once the worm has a user name and a password, it contacts the *rexec* server on the target host and tries to authenticate itself. If it can, it proceeds to the bootstrap phase; otherwise, it tries a slightly different approach—it connects to the local *rexec* server with the local user name and password, then uses this command interpreter to fire off a command interpreter on the target machine with *rsh*. This will succeed if the remote host says it trusts the local host in its */etc/hosts.equiv* file, or the remote account says it trusts the local account in its *.rhosts* file. *hu1*() ignores *infect*()’s ‘‘immune’’ flag and does not set this flag itself, since *hu1*() may find success on a per-account basis that *infect*() can’t achieve on a per-host basis.

Both *infect*() and *hu1*() use a routine we call *sendworm*() to do their dirty work[5]. *sendworm*() looks for the *l1.c* bootstrap source file in its objects list, then it uses the *makemagic*() routine to get a communication stream endpoint (a *socket*), a random network port number to rendezvous at, and a magic number for authentication. (There is an interesting side effect to *makemagic*()—it looks for a usable address for the target host by trying to connect to its TCP *telnet* port; this produces a characteristic log message from the *telnet* server.) If *makemagic*() was successful, the worm begins to send commands to the remote command interpreter that was started up by the immediately preceding attack. It changes its directory to an unprotected place (*/usr/tmp*), then it sends across the bootstrap source, using the UNIX stream editor *sed* to parse the input stream. The bootstrap source is compiled and run on the remote system, and the worm runs a routine named *waithit*() to wait for the remote bootstrap to call back on the selected port.

The bootstrap is quite simple. It is supplied the address of the originating host, a TCP port number and a magic number as arguments. When it starts, it unlinks itself so that it can’t be detected in the filesystem, then it calls *fork*() to create a new process with the same image. The old process exits, permitting the originating worm to continue with its business. The bootstrap reads its arguments then zeroes them out to hide them from the system status program; then it is ready to connect over the network to the parent worm. When the connection is made, the bootstrap sends over the magic number, which the parent will check against its own copy. If the parent accepts the number (which is carefully rendered to be independent of host byte order), it will send over a series of filenames and files which the bootstrap writes to disk. If trouble occurs, the bootstrap removes all these files and exits. Eventually the transaction completes, and the bootstrap calls up a command interpreter to finish the job.

In the meantime, the parent in *waithit*() spends up to two minutes waiting for the bootstrap to call back; if the bootstrap fails to call back, or the authentication fails, the worm decides to give up and reports a failure. When a connection is successful, the worm ships all of its files across followed by an end-of-file indicator. It pauses four seconds to let a command interpreter start on the remote side, then it issues commands to create a new worm. For each relocatable object file in the list of files, the worm tries to build an executable object; typically each file contains code for a particular make of computer, and the builds will fail until the worm tries the proper computer type. If the parent worm finally gets an executable child worm built, it sets it loose with the **−p** option to kill the command interpreter, then shuts down the connection. The target host is marked ‘‘infected’’. If none of the objects produces a usable child worm, the parent removes the detritus and *waithit*() returns an error indication.

When a system is being swamped by worms, the */usr/tmp* directory can fill with leftover files as a consequence of a bug in *waithit*(). If a worm compile takes more than 30 seconds, resynchronization code will report an error but *waithit*() will fail to remove the files it has created. On one of our machines, 13 MB of material representing 86 sets of files accumulated over 5.5 hours.

---

[5] One minor exception: the *sendmail* attack doesn’t use *sendworm*() since it needs to handle the SMTP protocol in addition to the command interpreter interface, but the principle is the same.

4.7. Password cracking

A password cracking algorithm seems like a slow and bulky item to put in a worm, but the worm makes this work by being persistent and efficient. The worm is aided by some unfortunate statistics about typical password choices. Here we discuss how the worm goes about choosing passwords to test and how the UNIX password encryption routine was modified.

4.7.1. Guessing passwords

> For example, if the login name is ‘‘abc’’, then ‘‘abc’’, ‘‘cba’’, and ‘‘abcabc’’ are excellent candidates for passwords.
> [Grampp and Morris, ‘‘UNIX Operating System Security’’]

The worm’s password guessing is driven by a little 4-state machine. The first state gathers password data, while the remaining states represent increasingly less likely sources of potential passwords. The central cracking routine is called *cracksome*(), and it contains a switch on each of the four states.

The routine that implements the first state we named *crack_0*(). This routine’s job is to collect information about hosts and accounts. It is only run once; the information it gathers persists for the lifetime of the worm. Its implementation is straightforward: it reads the files */etc/hosts.equiv* and */.rhosts* for hosts to attack, then reads the password file looking for accounts. For each account, the worm saves the name, the encrypted password, the home directory and the user information fields. As a quick preliminary check, it looks for a *.forward* file in each user’s home directory and saves any host name it finds in that file, marking it like the previous ones.

We unimaginatively called the function for the next state *crack_1*(). *crack_1*() looks for trivially broken passwords. These are passwords which can be guessed merely on the basis of information already contained in the password file. Grampp and Morris report a survey of over 100 password files where between 8 and 30 percent of all passwords were guessed using just the literal account name and a couple of variations. The worm tries a little harder than this: it checks the null password, the account name, the account name concatenated with itself, the first name (extracted from the user information field, with the first letter mapped to lower case), the last name, and the account name reversed. It runs through up to 50 accounts per call to *cracksome*(), saving its place in the list of accounts and advancing to the next state when it runs out of accounts to try.

The next state is handled by *crack_2*(). In this state the worm compares a list of favorite passwords, one password per call, with all of the encrypted passwords in the password file. The list contains 432 words, most of which are real English words or proper names; it seems likely that this list was generated by stealing password files and cracking them at leisure on the worm author’s home machine. A global variable *nextw* is used to count the number of passwords tried, and it is this count (plus a loss in the population control game) that controls whether the worm exits at the end of the main loop—*nextw* must be greater than 10 before the worm can exit. Since the worm normally spends 2.5 minutes checking for clients over the course of the main loop and calls *cracksome*() twice in that period, it appears that the worm must make a minimum of 7 passes through the main loop, taking more than 15 minutes[6]. It will take at least 9 hours for the worm to scan its built-in password list and proceed to the next state.

The last state is handled by *crack_3*(). It opens the UNIX online dictionary */usr/dict/words* and goes through it one word at a time. If a word is capitalized, the worm tries a lower-case version as well. This search can essentially go on forever: it would take something like four weeks for the worm to finish a typical dictionary like ours.

When the worm selects a potential password, it passes it to a routine we called *try_password*(). This function calls the worm’s special version of the UNIX password encryption function *crypt*() and compares the result with the target account’s actual encrypted password. If they are equal, or if the password and guess are the null string (no password), the worm saves the cleartext password and proceeds to attack hosts that are connected to this account. A routine we called *try_forward_and_rhosts*() reads the user’s *.forward* and *.rhosts* files, calling the previously described *hu1*() function for each remote account it finds.

---

[6] For those mindful of details: The first call to *cracksome*() is consumed reading system files. The worm must spend at least one call to *cracksome*() in the second state attacking trivial passwords. This accounts for at least one pass through the main loop. In the third state, *cracksome*() tests one password from its list of favorites on each call; the worm will exit if it lost a roll of the dice and more than ten words have been checked, so this accounts for at least six loops, two words on each loop for five loops to reach 10 words, then another loop to pass that number. Altogether this amounts to a minimum of 7 loops. If all 7 loops took the maximum amount of time waiting for clients, this would require a minimum of 17.5 minutes, but the 2-minute check can exit early if a client connects and the server loses the challenge, hence 15.5 minutes of waiting time plus runtime overhead is the minimum lifetime. In this period a worm will attack at least 8 hosts through the host infection routines, and will try about 18 passwords for each account, attacking more hosts if accounts are cracked.

4.7.2. Faster password encryption

> The use of encrypted passwords appears reasonably secure in the absence of serious attention of experts in the field. [Morris and Thompson, ‘‘Password Security: A Case History’’]

Unfortunately some experts in the field have been giving serious attention to fast implementations of the UNIX password encryption algorithm. UNIX password authentication works without putting any readable version of the password onto the system, and indeed works without protecting the encrypted password against reading by users on the system. When a user types a password in the clear, the system encrypts it using the standard *crypt*() library routine, then compares it against a saved copy of the encrypted password. The encryption algorithm is meant to be basically impossible to invert, preventing the retrieval of passwords by examining only the encrypted text, and it is meant to be expensive to run, so that testing guesses will take a long time. The UNIX password encryption algorithm is based on the Federal Data Encryption Standard (DES). Currently no one knows how to invert this algorithm in a reasonable amount of time, and while fast DES encoding chips are available, the UNIX version of the algorithm is slightly perturbed so that it is impossible to use a standard DES chip to implement it.

Two problems have been mitigating against the UNIX implementation of DES. Computers are continually increasing in speed—current machines are typically several times faster than the machines that were available when the current password scheme was invented. At the same time, ways have been discovered to make software DES run faster. UNIX passwords are now far more susceptible to persistent guessing, particularly if the encrypted passwords are already known. The worm’s version of the UNIX *crypt*() routine ran more than 9 times faster than the standard version when we tested it on our VAX 8600. While the standard *crypt*() takes 54 seconds to encrypt 271 passwords on our 8600 (the number of passwords actually contained in our password file), the worm’s *crypt*() takes less than 6 seconds.

The worm’s *crypt*() algorithm appears to be a compromise between time and space: the time needed to encrypt one password guess versus the substantial extra table space needed to squeeze performance out of the algorithm. Curiously, one performance improvement actually saves a little space. The traditional UNIX algorithm stores each bit of the password in a byte, while the worm’s algorithm packs the bits into two 32-bit words. This permits the worm’s algorithm to use bit-field and shift operations on the password data, which is immensely faster. Other speedups include unrolling loops, combining tables, precomputing shifts and masks, and eliminating redundant initial and final permutations when performing the 25 applications of modified DES that the password encryption algorithm uses. The biggest performance improvement comes as a result of combining permutations: the worm uses expanded arrays which are indexed by groups of bits rather than the single bits used by the standard algorithm. Matt Bishop’s fast version of *crypt*() does all of these things and also precomputes even more functions, yielding twice the performance of the worm’s algorithm but requiring nearly 200 KB of initialized data as opposed to the 6 KB used by the worm and the less than 2 KB used by the normal *crypt*().

How can system administrators defend against fast implementations of *crypt*()? One suggestion that has been introduced for foiling the bad guys is the idea of shadow password files. In this scheme, the encrypted passwords are hidden rather than public, forcing a cracker to either break a privileged account or use the host’s CPU and (slow) encryption algorithm to attack, with the added danger that password test requests could be logged and password cracking discovered. The disadvantage of shadow password files is that if the bad guys somehow get around the protections for the file that contains the actual passwords, all of the passwords must be considered cracked and will need to be replaced. Another suggestion has been to replace the UNIX DES implementation with the fastest available implementation, but run it 1000 times or more instead of the 25 times used in the UNIX *crypt*() code. Unless the repeat count is somehow pegged to the fastest available CPU speed, this approach merely postpones the day of reckoning until the cracker finds a faster machine. It’s interesting to note that Morris and Thompson measured the time to compute the old M-209 (non-DES) password encryption algorithm used in early versions of UNIX on the PDP-11/70 and found that a good implementation took only 1.25 milliseconds per encryption, which they deemed insufficient; currently the VAX 8600 using Matt Bishop’s DES-based algorithm needs 11.5 milliseconds per encryption, and machines 10 times faster than the VAX 8600 at a cheaper price will be available soon (if they aren’t already!).

5. Opinions

> The act of breaking into a computer system has to have the same social stigma as breaking into a neighbor’s house. It should not matter that the neighbor’s door is unlocked. [Ken Thompson, 1983 Turing Award Lecture]
>
> [Creators of viruses are] stealing a car for the purpose of joyriding. [R H Morris, in 1983 Capitol Hill testimony, cited in the New York Times 11/11/88]

I don’t propose to offer definitive statements on the morality of the worm’s author, the ethics of publishing security information or the security needs of the UNIX computing community, since people better (and less) qualified than I are still copiously flaming on these topics in the various network newsgroups and mailing lists. For the sake of the mythical ordinary system administrator who might have been confused by all the information and misinformation, I will try to answer a few of the most relevant questions in a narrow but useful way.

Did the worm cause damage? The worm did not destroy files, intercept private mail, reveal passwords, corrupt databases or plant trojan horses. It did compete for CPU time with, and eventually overwhelm, ordinary user processes. It used up limited system resources such as the open file table and the process text table, causing user processes to fail for lack of same. It caused some machines to crash by operating them close to the limits of their capacity, exercising bugs that do not appear under normal loads. It forced administrators to perform one or more reboots to clear worms from the system, terminating user sessions and long-running jobs. It forced administrators to shut down network gateways, including gateways between important nation-wide research networks, in an effort to isolate the worm; this led to delays of up to several days in the exchange of electronic mail, causing some projects to miss deadlines and others to lose valuable research time. It made systems staff across the country drop their ongoing hacks and work 24-hour days trying to corner and kill worms. It caused members of management in at least one institution to become so frightened that they scrubbed all the disks at their facility that were online at the time of the infection, and limited reloading of files to data that was verifiably unmodified by a foreign agent. It caused bandwidth through gateways that were still running after the infection started to become substantially degraded—the gateways were using much of their capacity just shipping the worm from one network to another. It penetrated user accounts and caused it to appear that a given user was disturbing a system when in fact they were not responsible. It’s true that the worm could have been far more harmful that it actually turned out to be: in the last few weeks, several security bugs have come to light which the worm could have used to thoroughly destroy a system. Perhaps we should be grateful that we escaped incredibly awful consequences, and perhaps we should also be grateful that we have learned so much about the weaknesses in our systems’ defenses, but I think we should share our gratefulness with someone other than the worm’s author.

Was the worm malicious? Some people have suggested that the worm was an innocent experiment that got out of hand, and that it was never intended to spread so fast or so widely. We can find evidence in the worm to support and to contradict this hypothesis. There are a number of bugs in the worm that appear to be the result of hasty ficareless or programming. For example, in the worm’s *if_init*() routine, there is a call to the block zero function *bzero*() that incorrectly uses the block itself rather than the block’s address as an argument. It’s also possible that a bug was responsible for the ineffectiveness of the population control measures used by the worm. This could be seen as evidence that a development version of the worm ‘‘got loose’’ accidentally, and perhaps the author originally intended to test the final version under controlled conditions, in an environment from which it would not escape. On the other hand, there is considerable evidence that the worm was designed to reproduce quickly and spread itself over great distances. It can be argued that the population control hacks in the worm are anemic by design: they are a compromise between spreading the worm as quickly as possible and raising the load enough to be detected and defeated. A worm will exist for a substantial amount of time and will perform a substantial amount of work even if it loses the roll of the (imaginary) dice; moreover, 1 in 7 worms become immortal and can’t be killed by dice rolls. There is ample evidence that the worm was designed to hamper efforts to stop it even after it was identified and captured. It certainly succeeded in this, since it took almost a day before the last mode of infection (the *finger* server) was identified, analyzed and reported widely; the worm was very successful in propagating itself during this time even on systems which had fixed the *sendmail* debug problem and had turned off *rexec*. Finally, there is evidence that the worm’s author deliberately introduced the worm to a foreign site that was left open and welcome to casual outside users, rather ungraciously abusing this hospitality. He apparently further abused this trust by deleting a log file that might have revealed information that could link his home site with the infection. I think the innocence lies in the research community rather than with the worm’s author.

Will publication of worm details further harm security? In a sense, the worm itself has solved that problem: it has published itself by sending copies to hundreds or thousands of machines around the world. Of course a bad guy who wants to use the worm’s tricks would have to go through the same effort that we went through in order to understand the program, but then it only took us a week to completely decompile the program, so while it takes fortitude to hack the worm, it clearly is not greatly difficult for a decent programmer. One of the worm’s most effective tricks was advertised when it entered—the bulk of the *sendmail* hack is visible in the log file, and a few minutes’ work with the sources will reveal the rest of the trick. The worm’s fast password algorithm could be useful to the bad guys, but at least two other faster implementations have been available for a year or more, so it isn’t very secret, or even very original. Finally, the details of the worm have been well enough sketched out on various newsgroups and mailing lists that the principal hacks are common knowledge. I think it's more important that we understand what happened, so that we can make it less likely to happen again, than that we spend time in a futile effort to cover up the issue from everyone but the bad guys. Fixes for both source and binary distributions are widely available, and anyone who runs a system with these vulnerabilities needs to look into these fixes immediately, if they haven't done so already.

6. Conclusion

> It has raised the public awareness to a considerable degree. [R H Morris, quoted in the New York Times 11/5/88]

This quote is one of the understatements of the year. The worm story was on the front page of the New York Times and other newspapers for days. It was the subject of television and radio features. Even the *Bloom County* comic strip poked fun at it.

Our community has never before been in the limelight in this way, and judging by the response, it has scared us. I won't offer any fancy platitudes about how the experience is going to change us, but I will say that I think these issues have been ignored for much longer than was safe, and I feel that a better understanding of the crisis just past will help us cope better with the next one. Let's hope we're as lucky next time as we were this time.

Acknowledgments

No one is to blame for the inaccuracies herein except me, but there are plenty of people to thank for helping to decompile the worm and for helping to document the epidemic. Dave Pare and Chris Torek were at the center of the action during the late night session at Berkeley, and they had help and kibitzing from Keith Bostic, Phil Lapsley, Peter Yee, Jay Lepreau and a cast of thousands. Glenn Adams and Dave Siegel provided good information on the MIT AI Lab attack, while Steve Miller gave me details on Maryland, Jeff Forys on Utah, and Phil Lapsley, Peter Yee and Keith Bostic on Berkeley. Bill Cheswick sent me a couple of fun anecdotes from AT&T Bell Labs. Jim Haynes gave me the run-down on the security problems turned up by his busy little undergrads at UC Santa Cruz. Eric Allman, Keith Bostic, Bill Cheswick, Mike Hibler, Jay Lepreau, Chris Torek and Mike Zeleznik provided many useful review comments. Thank you all, and everyone else I forgot to mention.

Matt Bishop's paper ''A Fast Version of the DES and a Password Encryption Algorithm'', ©1987 by Matt Bishop and the Universities Space Research Association, was helpful in (slightly) parting the mysteries of DES for me. Anyone wishing to understand the worm's DES hacking had better look here first. The paper is available with Bishop's *deszip* distribution of software for fast DES encryption. The latter was produced while Bishop was with the Research Institute for Advanced Computer Science at NASA Ames Research Center; Bishop is now at Dartmouth College (*bishop@bear.dartmouth.edu*). He sent me a very helpful note on the worm's implementation of *crypt*() which I leaned on heavily when discussing the algorithm above.

The following documents were also referenced above for quotes or for other material:

Data Encryption Standard, FIPS PUB 46, National Bureau of Standards, Washington D.C., January 15, 1977.

F. T. Grampp and R. H. Morris, ''UNIX Operating System Security,'' in the *AT&T Bell Laboratories Technical Journal*, October 1984, Vol. 63, No. 8, Part 2, p. 1649.

Brian W. Kernighan and Dennis Ritchie, *The C Programming Language,* Second Edition, Prentice Hall: Englewood Cliffs, NJ, ©1988.

John Markoff, ''Author of computer 'virus' is son of U.S. Electronic Security Expert,'' p. 1 of the *New York Times*, November 5, 1988.

John Markoff, ''A family's passion for computers, gone sour,'' p. 1 of the *New York Times*, November 11, 1988.

Robert Morris and Ken Thompson, ''Password Security: A Case History,'' dated April 3, 1978, in the *UNIX Programmer's Manual*, in the *Supplementary Documents* or the *System Manager's Manual*, depending on where and when

L

D

L