# A Failure to Learn from the Past*

Eugene H. Spafford
Purdue University CERIAS
656 Oval Drive
West Lafayette, IN 47907–2086
<spaf@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. It was the first major network-wide attack on computer systems, and thus was a matter of considerable interest.*

*This paper provides a brief chronology of both the spread and eradication of the program, a presentation about how the program worked, and details of the aftermath. That is followed by discussion of some observations of what has happened in the years since that incident. The discussion supports the title of this paper — that the community has failed to learn from the past.*

## 1. Introduction

In October of 1988, Mark Lottor made a presentation at the Internet Engineering Task Force Meeting in Ann Arbor, Michigan where he stated that the number of hosts on the Internet was approximately 60,000. A few weeks later, on the evening of 2 November 1988, these machines came under attack from within. Sometime after 5 PM EST, a program was executed on one or more of those hosts. That program collected host, network, and user information, then used that information to establish network connections to break into other machines using flaws present in those systems' software. After compromising those systems, the program would replicate itself and the replica would attempt to spread to other systems in the same manner.

Although the program would only spread to Sun Microsystems Sun 3 systems, and Digital Equipment Corporation VAX computers running variants of version 4 BSD UNIX, the program multiplied quickly, as did the confusion and consternation of system administrators and users as they discovered that their systems had been invaded. Although UNIX was known at that time to have some security weaknesses (cf. [12, 15, 17, 18]), especially in its usual mode of operation in open research environments, the scope of the break-ins nonetheless came as a great surprise to almost everyone.

Prior to this incident no similar malicious software had been widely seen. Few people had heard of computer worms or viruses, thus making the incident all the more surprising. As a result, the program was mysterious to users at sites where it appeared. Unusual files were left in the scratch (`/usr/tmp`) directories of some machines, and strange messages appeared in the log files of some of the utilities, such as the `sendmail` mail handling agent[2]. 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. Based on some estimates of the spread of the Worm, 3000–6000 (5%–10%) machines were affected at the height of the attack.

By early Thursday morning, November 3, personnel at many sites around the country had "captured" copies of the program and begun to analyze 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

* Portions of this paper were taken from [21] and [22]. Readers are directed to those documents for additional details.

1063-9527/03 $17.00 © 2003 IEEE

compiler and loader to prevent their use. These suggestions were published in mailing lists and on the Usenet network news system, although their spread was hampered by systems disconnected from the Internet in an attempt to "quarantine" them.

By about 9 PM EST Thursday, another simple, effective method of stopping the invading program, without altering system utilities, was discovered at Purdue and also widely published. Software patches were posted by the Berkeley group at the same time to mend all the flaws that enabled the program to invade systems. All that remained was to analyze the code that caused the problems and discover who had unleashed the worm — and why.

In the weeks that followed, other well-publicized computer break-ins occurred and many debates began about how to deal with the individuals staging these break-ins, who is responsible for security and software updates, and the future roles of networks and security. In my papers in 1989 I predicted that it would be some time before these issues were put to rest; it is unfortunate that 15 years later we are still debating some of the same issues, and facing many of the same problems.

## 2. Terminology

Initially, there was considerable variation in the names applied to the malware unleashed on November 2nd. Many people used the term *worm* instead of *virus* based on its behavior. Members of the press used the term virus, possibly because their experience prior to that incident was only with viruses. That usage was reinforced by quotes from computer managers and programmers also unfamiliar with the difference. However, with time, the general consensus of the security community has been to consider the program as a worm program, and hence its name as the Internet Worm.

In [22] I proposed some terminology for malware that was then further expanded in [25] and [24]. This terminology has largely been adopted, but still lacks necessary precision. Recent incidents of malware show some of the shortcomings of these earlier definitions: the Slammer/Sapphire program of January 2003 was clearly a worm, but ILOVEYOU (May 2000) and Blaster (August 2003) required manual execution to activate, and thus were more in the nature of Trojan Horse programs. All of these have been referred to as "viruses" by the popular press and many security companies.

The definitions I used in 1989 were as follows. A *worm* is a program that can run independently and can propagate a fully working version of itself to other machines. It is derived from the word *tapeworm*, a parasitic organism that lives inside a host and uses its resources to maintain itself.

A *virus* is a piece of code that adds itself to other programs, including operating systems. It cannot run independently — it requires that its "host" program be run to activate it. As such, it has an analog to biological viruses — those viruses are not considered alive in the usual sense; instead, they invade host cells and corrupt them, causing them to produce new viruses.

### 2.1. Worms

The concept of a worm program that spreads itself from machine to machine was apparently first described by John Brunner in 1975 in his classic science fiction novel *The Shockwave Rider*.[4] He called these programs *tapeworms* that existed "inside" the computers and spread themselves to other machines. In the late 1970s, researchers at Xerox PARC built and experimented with worm programs. They reported their experiences in 1982 in [20], and cited Brunner as the inspiration for the name *worm*. Although not the first self-replicating programs to run in a network environment, these were the first such programs to be called worms.

The worms built at PARC were designed to do useful work in a distributed environment of cooperating machines . Because of this, some people preferred to call the Internet Worm a virus because it was destructive, and they considered worms to be non-destructive.

### 2.2. Virus

The first use of the term *virus* to refer to unwanted computer code was by Gregory Benford. As related by Dr. Benford[1], he published the idea of a virus in 1970 in the May issue of VENTURE MAGAZINE. His article specifically termed the idea "computer virus" and described a program named *Virus* — and tied this to the sale of a program called *Vaccine* to defeat it. All this came from his experience as a programmer and research physicist at the (then) Lawrence Radiation Lab in Livermore. He and the other scientists noticed that "bad code" could self-reproduce among lab computers, and eventually get onto the ARPANet. He tried writing and launching some and they succeeded with surprising ease. Professor Benford's friend, the science fiction author David Gerrold, later incorporated this idea into a series of short stories about the G.O.D. machine in the early 1970s that were later merged into a novel in 1972: *When Harlie Was One*.[11] The description of *virus* in that book does not quite fit the currently-accepted, popular definition of computer virus — a program that alters other programs to include a copy of itself.

Fred Cohen formally defined the term *computer virus* in 1983.[5] At that time, Cohen was a graduate student at the University of Southern California attending a security

[1] In private communication with this author, and later in a letter to the editor of the NY Times in Decemer of 1994.

seminar. Something discussed in class inspired him to think about self-reproducing code. He put together a simple example that he demonstrated to the class. His advisor, Professor Len Adleman, suggested that he call his creation a computer virus. Dr. Cohen's Ph.D. thesis and later research were devoted to computer viruses.

Dr. Cohen 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. (That the Internet Worm was unleased on the eve of the 5th anniversary of this event was coincidence of a most amazing sort.) Actual computer viruses were being written by individuals before Cohen, although not named such, as early as 1980 on Apple II computers.[9] The first few viruses were not circulated outside of a small population, with the notable exception of the "Elk Cloner" virus for Apple II computers, released in 1981.

### 2.3. Other Notable Definitions

In their widely circulated paper,[7] Eichin and Rochlis chose to call the November 2nd program a virus. Their reasoning for this required reference to biological literature and observing distinctions between *lytic* viruses and *lysogenic* viruses. It further required that we view the Internet as a whole to be the *infected host*.

Their explanation merely serves to underscore the dangers of co-opting terms from another discipline to describe phenomena within our own (computing). The original definitions may be much more complex than we originally imagine, and attempts to maintain and justify the analogies may require a considerable effort. Here, it may also require an advanced degree in the biological sciences! Although Eichin and Rochlis presented a reasoned argument for a more precise analogy to biological viruses, it was too complex a distinction to be useful to computer scientists and was not adopted.

Cohen (and others, including Len Adleman[1]) have attempted formal definitions of *computer virus*, none have gained widespread acceptance or use. This is a result of the difficulty in defining precisely the characteristics of what a virus is and is not. Cohen's formal definition includes any programs capable of self-reproduction. Thus, by his definition, programs such as compilers and editors would be classed as "viruses."

Stubbs and Hoffman quoted a definition by John Inglis that captures the generally accepted view of computer viruses:

> "He defines a virus as a piece of code with two characteristics:
> 1. At least a partially automated capability to reproduce.
> 2. A method of transfer which is dependent on its ability to attach itself to other computer entities (programs, disk sectors, data files, etc.) that move between these systems."[27, p. 145]

Several other interesting definitions are discussed in [13, Chapter 1].

## 3. How the Worm Operated

The Worm took advantage of flaws in standard software installed on many UNIX systems of the time. It also took advantage of a mechanism used to simplify the sharing of resources in local area networks. Those flaws are described here, along with some related problems. Following is a description of how the Worm used the flaws to invade systems.

### 3.1. fingerd and gets

The `finger` program was a utility that allowed users to obtain information about other users. It was usually used to identify the full name or login name of a user, whether a user was currently logged in, and possibly other information about the person such as telephone numbers where he or she could be reached. The `fingerd` server program was intended to run as a daemon background process, to service remote requests using the finger protocol. This daemon program accepted connections from remote programs, read a single line of input, and then sent back output matching the received request.

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

The `gets` routine is not the only routine with this flaw. There is a whole family of routines in the C library that may also overrun buffers when decoding input or formatting output unless the user explicitly specifies limits on the number of characters to be converted. Although experienced C programmers are aware of the problems with these routines, many continue to use them. Worse, their format is in some sense codified not only by historical inclusion in UNIX and the C language, but more formally in the ANSI language standard for C. As a result, there have been hundreds of buffer overflow vulnerabilities written and discovered in the last 15 years.

### 3.2. Sendmail

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

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

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

It is little wonder, then, that bugs were present in `sendmail` that allowed unexpected behavior. Other flaws were found and reported after attention was focused on the program as a result of the Worm, but to this day versions of `sendmail` are in use and have occasional patches released for security issues.

### 3.3. Passwords

A key attack of the Worm program involved attempts to discover user passwords. It was able to determine success because the encrypted password of each user was in a publicly-readable file.

Strictly speaking, the password was not encrypted. A block of zero bits was repeatedly encrypted using the user password, and the result of this encryption was what was saved.[15, 3, 10] In original (and some current) UNIX systems, the user provided a password at sign-on to verify identity. The password was used to encrypt a block of zero bits using a permuted version of the Data Encryption Standard (DES) algorithm, and the result was compared against a previously encrypted value present in a world-readable accounting file. If a match occurred, access was allowed. No plaintext passwords were contained in the file, and the algorithm was supposedly non-invertible without knowledge of the password.

The organization of the passwords in UNIX allowed nonprivileged commands to make use of information stored in the accounts file, including authentification schemes using user passwords. However, it also allowed an attacker to encrypt lists of possible passwords and then compare them against the actual passwords without calling any system function. In effect, the security of the passwords was provided by the prohibitive effort of trying this approach with all combinations of letters, or at least trying obvious words.

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

One way to reduce the risk of such attacks, and an approach that has been taken in every modern variant of UNIX, is to have a shadow password file. The encrypted passwords are saved in a file (shadow) that is readable only by the system administrators, and a privileged call performs password encryptions and comparisons with an appropriate timed delay (.5 to 1 second, for instance). This prevents any attempts to "fish" for passwords. Additionally, a threshold can be included to check for repeated password attempts from the same process, resulting in some form of alarm being raised.

A related flaw exploited by the Worm involved the use of trusted logins. One useful feature of BSD UNIX-based networking code was its support for executing tasks on remote machines. To avoid having repeatedly to type passwords to access remote accounts, it was possible for a user to specify a list of host/login name pairs that were assumed to be trusted, in the sense that a remote access from that host/login pair was never asked for a password. This feature had often been responsible for users gaining unauthorized access to machines (cf. [17]) but it continued to be used because of its great convenience. In many systems in use today this feature is still available and enabled by users who do not understand the risk.

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

### 3.4. High Level Description

The Worm consisted of two parts: a main program, and a bootstrap or vector program. The main program, once established on a machine, would collect information on other machines in the network to which the current machine could connect. It would do this by reading public configuration files and by running system utility programs that presented information about the current state of network connections. It would then attempt to use the flaws described above to establish its bootstrap on each of those remote machines. Many of these techniques seem common in malicious code of 2003, but were relatively novel in 1988.

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

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

Once established on the target machine, the bootstrap would connect back to the instance of the Worm that originated it and transfer a set of binary files (precompiled code) to the local machine. Each binary file represented a version of the main Worm program, compiled for a particular computer architecture and operating system version. The bootstrap would also transfer a copy of itself for use in infecting other systems.

One curious feature of the bootstrap has provoked many questions that have never been answered in public: the program had data structures allocated to enable transfer of up to 20 files; it was used with only three. This led to speculation whether a more extensive version of the Worm was planned for a later date, and if that version might have carried with it other command files, password data, or possibly local virus or trojan horse programs. However, it is also possible that 20 was chosen as a limit with no plans for future expansion but as a "reasonable size."

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

### 3.5. Step-by-step description

This section contains a more detailed overview of how the Worm program functioned. The description in this section assumes that the reader is somewhat familiar with standard UNIX commands and with UNIX network facilities. A more detailed analysis of operation and components can be found in [22] with additional details in [7] and [19].

This description starts from the point at which a host is about to be infected. A Worm running on another machine has either succeeded in establishing a command shell invocation 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 data to the sendmail program.

The infection proceeded as follows:

1. A network socket was established on the infecting machine for the vector program to connect to (e.g., socket number 32341). A challenge was constructed from a random number (e.g., 8712440). A file name base was also constructed using a random number (e.g., 14481910).
2. The vector program was installed and executed using one of two methods:

   a) Across a TCP connection to a shell, the Worm would send a command stream to compile and execute the vector program, using the challenge and port values generated in the previous step. Then it would wait for the string DONE to be echoed to signal that the vector program was running.

   b) Using the SMTP connection, it would transmit a similar command stream to that for a TCP connection, but rather than wait for a terminating flag string, 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 back to the server, sent the challenge string, and received 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 then sent a series of commands to the vector to cause it to attempt to link each transferred binary file against the system libraries and then execute it. In practice, this meant it only attempted linking two binaries, although the code allowed up to 20. If the linked code succeeded, the server would close the connection. Otherwise, it would try the other binary file. After both binary files had been tried, it would send over commands to delete 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 (spawning) process. 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 system calls, by reading various system files, and by executing system programs that returned configuration information.

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

The attack attempts proceeded by one of three routes: `rsh`, `fingerd`, or `sendmail`.

The attack via `rsh` was done by attempting to spawn a remote shell. If successful, the host was infected as above.

The attack via the `finger` daemon was somewhat more subtle. This involved a buffer overflow attack. A connection was established to the remote finger server daemon and then a specially constructed string of 536 bytes was passed to the daemon, overflowing its 512 byte input buffer and overwriting parts of the stack. For standard 4 BSD versions running on VAX computers, the overflow resulted in the return stack frame for the main routine being changed so that the return address pointed into the buffer on the stack. The code at that location initiated a command shell with its input connected to the network link established for the `finger` invocation. The Worm then proceeded to infect the host as described above.

On Suns, this buffer overflow attack simply resulted in a core dump because the code was not in place to corrupt a Sun version of `fingerd` in a similar fashion. Curiously, correct machine-specific code to corrupt Suns could have been written in a matter of hours and included but was not.[22]

Last of all, the Worm then tried to infect the remote host by establishing a connection to the SMTP port and mailing an infection, as was described 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.

Next, the Worm program entered a state machine consisting of five states. Each state but the last was run for a short while, then the program looped back to make attempts 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 accounts. 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:

1. The Worm read through the `/etc/hosts.equiv` files and `/.rhosts` files to find the names of administratively-equivalent hosts. These were marked in the internal table of hosts. Next, the Worm read the account and password file into an internal data structure. As it was doing this, it also examined the mail forwarding file in each user home directory and included any new host names into its internal table of hosts to try.

2. 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 the user information field to try simple passwords. This was an approach that succeeded far too often because users had chosen weak passwords based on their own names or office information.

3. The code then attempted to break the password of each user by trying each word present in an internal list of words. This list of 432 words was tried against each account in a random order. This collection of passwords included some misspellings of real words, some names, and some other non-obvious strings. Many people have postulated that this list was generated by capturing real passwords in one or more operational environments.

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

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 mail and remote login control 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:

The Worm would first attempt to create a remote shell using the `rexec` remote command execution service. That service required that a username/password combination be supplied as part of the request. The attempt would be made using the various account names found in local files and the user's local password. This took advantage of users' tendency to use the same password on their accounts on multiple machines.

The code would first do an `rexec` to the current host (using the local user name and password) and would then try a remote shell invocation on the remote host using the username taken from the file. This attack would succeed when the remote machine allowed the user to log in without a password (a trust relationship existed).

If the remote shell was created either way, the attack would continue as described 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. This was compiled in as port number 23357, on host 127.0.0.1 (loopback). If such a connection succeeded, one Worm would (randomly) set an internal variable named `pleasequit` to 1, causing that Worm to exit after it had reached part way into the third stage 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 termination until significant effort had been made to break local passwords. Furthermore, race conditions in the code made it possible for Worms on heavily loaded machines to fail to connect, thus causing some of them to continue indefinitely despite the presence of other instances.

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

The Worm attempted to send a UDP packet to the host ernie.berkeley.edu approximately once every 15 infections, based on a random number comparison. The code to do this was incorrect, however, and no information was ever sent. Whether this was the intended behavior or whether there was some reason for the byte to be sent is not known. However, the code is such that an uninitialized byte was 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. However, no evidence for such a program was ever found and it is possible that the connection was simply a feint to cast suspicion on personnel at Berkeley.

The Worm would also duplicate itself on a regular basis and kill its parent. This had two effects. First, the Worm appeared to keep changing its process identifier and no single process accumulated excessive amounts of cpu time. Secondly, processes that had been running for a long time had 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 at Purdue we observed some instances of the Worm with over 600 seconds of accumulated cpu time.

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

## 4. Chronology

What follows is an abbreviated chronology of events relating to the release of the Internet Worm. Most of this information was gathered from personal mail, submissions to mailing lists, and Usenet postings. Some items were also taken from [19] and [16] and are marked accordingly. Note that because of clock drift and machine crashes, some of the times given here may not be completely accurate. They should convey an approximation to the sequence of events, however. All times are given in Eastern Standard Time.

My archived version of the *phage* list referenced below was recently (mid-2003) made available via a WWW interface. An annotated version can be found at `<http://securitydigest.org/phage/>`.

### November 2, 1988

1700 Worm executed on a machine at Cornell University.

(NCSC) Whether this was a last test or the initial execution is not known.

1800 Machine prep.ai.mit.edu at MIT infected. (Seely, mail) This may have been the initial execution. Prep was a public-access machine, used for storage and distribution of GNU project software. It was configured with some notorious security holes that allowed anonymous remote users to introduce files into the system.

1830 Infected machine at the University of Pittsburgh infects a machine at the RAND Corporation. (NCSC)

2100 Worm discovered on machines at Stanford. (NCSC)

2130 First machine at the University of Minnesota invaded. (mail)

2204 Gateway machine at University of California, Berkeley invaded. Mike Karels and Phil Lapsley discover this shortly afterwards because they noticed an unusual load on the machine. (mail)

2234 Gateway machine at Princeton University infected. (mail)

2240 Machines at the University of North Carolina are infected and attempt to invade other machines. Attempts on machines at MCNC (Microelectronics Center of North Carolina) start at 2240. (mail)

2248 Machines at SRI infected via sendmail. (mail)

2252 Worm attempts to invade machine andrew.cmu.edu at Carnegie-Mellon University. (mail)

2254 Gateway hosts at the University of Maryland come under attack via `fingerd` daemon. Evidence is later found that other local hosts are already infected. (mail)

2259 Machines at University of Pennsylvania attacked, but none are susceptible. Logs will later show 210 attempts over next 12 hours. (mail)

2300 AI Lab machines at MIT infected. (NCSC)

2328 mimsy.umd.edu at University of Maryland is infected via sendmail. (mail)

2340 Researchers at Berkeley discover `sendmail` and `rsh` as means of attack. They begin to shut off other network services as a precaution. (Seeley)

2345 Machines at Dartmouth and the Army Ballistics Research Lab (BRL) attacked and infected. (mail, NCSC)

2349 Gateway machine at the University of Utah infected. In the next hour, the load average will soar to 100 (normal average was below 10) because of repeated infections. (Seeley)

### November 3, 1988

0007 University of Arizona machine arizona.edu infected. (mail)

0021 Princeton University main machine (a VAX 8650) infected. Load average reaches 68 and the machine crashes. (mail)

0033 Machine dewey.udel.edu at the University of Delaware infected, but not by sendmail. (mail)

0105 Worm invades machines at Lawrence Livermore Labs (LLNL). (NCSC)

0130 Machines at UCLA infected. (mail)

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

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

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

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

0400 Colorado State University attacked. (mail)

0400 Machines at Purdue University infected.

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

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

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

0730 I discover infection on machines at Purdue University. Machines are so overloaded I cannot read my mail or news, including mail from Keith Bostic about the

Worm. Believing this to be related to a recurring hardware problem on my machine, I request that the system be restarted.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

### November 4, 1988

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

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

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

1100 Milnet-ARPAnet mailbridges restored. (NCSC)

1420 Keith Bostic reposts fix to fingerd. (mail)

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

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

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

### November 5, 1988

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

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

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

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

## 5. Aftermath

### 5.1. Author, Intent, and Punishment

Two of the first questions to be asked even before the Worm was stopped were simply the questions ”Who?” and ”Why?”. Who had written the Worm, and why had he/she/they loosed it in the Internet? The question of ”Who?” was answered shortly thereafter when the New York Times identified Robert T. Morris. The report from the Provost’s office at Cornell [8] also named Robert T. Morris as the culprit, and presented convincing reasons for that conclusion.

Morris was charged with a Federal felony under 18 U.S.C. 1030 and underwent trial in the district court in Syracuse, NY. He did not deny that he had written and released the Worm, but he pled not guilty to the felony. His defense included that he did not intend to cause damage, and that the damage did not meet the required threshold. Testimony from a variety of witnesses established the magnitude of the losses nationwide. Testimony about the nature of the code and comments in the source that was recovered from his account left little doubt that he wrote the Worm to spread and be difficult to spot and detect. Thus, it was not surprising when he was found guilty on 22 Jan1990. Morris appealed his verdict and the Court of Appeals upheld the verdict. The case was appealed to the Supreme Court, but they declined to hear the appeal.

Morris was sentenced to three years of probation, 400 hours of community service, a fine of $10,500, and an additional assessment of $3276 to cover the cost of his probation. He received no time in prison. He was also suspended from Cornell University where he had been a graduate student. (When he applied for readmission several years later, his request was denied.) He spent several years working as a programmer, and then as one of the founders of an Internet commerce company. Mr. Morris then entered graduate school at Harvard University. He completed his Ph.D. in 1999, and he is currently an associate professor at MIT.

Throughout the trial and the time since then, Dr. Morris has remained silent in public about the Worm and his motives. To his credit, he has not attempted to trade on his notoriety for financial gain. His dissertation and current scientific research are in networking and not security. His behavior has tended to support his contention at trial that his intention was benign. However, his lack of public statements mean that his complete motive remains a mystery. Conjectures have ranged from an experiment gone awry to a subconscious act of revenge against his father. All of this is sheer speculation, however. All we have to work with is the decompiled code for the program and our understanding of its effects. It is impossible to intuit the real motive from those or from various individuals’ experiences with the author. It is entirely possible that we will never learn the full story; now that 15 years have passed, many of the details and perspectives have been lost forever.

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

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

In light of the lack of definitive information, it was puzzling to note attempts by many people to defend Mr. Morris in 1988 and 1989 by claiming that his intent was to demonstrate something about Internet security, or that he was trying a harmless experiment. It is curious that so many people, journalists and computer professionals alike, would assume to know the intent of the author based on the observed behavior of the program. As Rick Adams of the Center for Seismic Studies (and later founder of UUnet) wryly observed in a posting to the Usenet, we may someday learn that the Worm was actually written to impress Jodie Foster — we simply do not know the real reasons.

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

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

The Cornell commission recommended some punishment, but not punishment so severe that Mr. Morris's future career in computing would be jepordized. The punishment meted out was consistent with that recommendation. As was observed in both [14] and [6] there was a danger in overreacting to that particular incident: less than 10% of the machines on an unsecure network were affected for less than a few days.

However, several of us argued that neither should we dismiss the whole Worm incident as something of little consequence. That no damage was done could have been an accident, and Morris's true motives were never revealed. Furthermore, many people were concerned about setting a dangerous precedent for future occurrences of such behavior. Excusing acts of computer vandalism simply because their authors claim there was no intent to cause damage will do little to discourage repeat offenses, and may encourage new incidents. ( I later presented this more general point in greater depth in [23].)

The claim that the victims of the Worm were somehow responsible for the invasion of their machines was also curious. The individuals making that claim seemed to be stating that there was some moral or legal obligation for computer users to track and install every conceivable security fix and mechanism available. This totally ignored that many sites might run turn-key systems without source code or administrators knowledgeable enough to modify their systems. Some of those sites might also have been running specialized software or have restricted budgets that precluded them installing new software versions. Many commercial and government sites operated their systems this way. To attempt to blame these individuals for the success of the Worm was (and is) equivalent to blaming an arson victim for the fire because she didn't build her house of fireproof metal. (More on this theme can be found in [23].)

### 5.2. Worm Hunters

A significant conclusion reached at the NCSC post-mortem workshop was that the reason the Worm was stopped so quickly was due almost solely to the UNIX "old-boy" network, and not because of any formal mechanism in place at the time.[16] A general recommendation from that workshop was that a formal crisis center be established to deal with future incidents and to provide a formal point of contact for individuals wishing to report problems. No such center was established at that time.

On November 29, 1988, someone exploiting a security flaw present in older versions of the FTP file transfer program broke into a machine on the MILnet. The intruder was traced to a machine on the ARPAnet, and to prevent further access the MILnet/ARPAnet links were immediately severed. During the next 48 hours there was considerable confusion and rumor about the disconnection, fueled in part by the Defense Communication Agency's attempt to explain the disconnection as a test rather than as a security problem.

That event, coming as close as it did to the Worm incident, prompted DARPA to establish the CERT (Computer Emergency Response Team, now the CERT/CC) at the Software Engineering Institute at Carnegie-Mellon University. The stated purpose of the CERT was to act as a central switchboard and coordinator for computer security emergencies on ARPAnet and MILnet computers. Of interest here is that the CERT was not chartered to deal with just any Internet emergency. Thus, problems detected in the CSnet, Bitnet, NSFnet, and other Internet communities of the time were not be referable to the CERT. I was told it was the expectation of CERT personnel that those other network communities would develop their own CERT-like groups.

## 6. Original Concluding Remarks

*(The following is the verbatim conclusion from 1989.)* Not all the consequences of the Internet Worm incident are yet known; they may never be. Most likely there will be changes in security consciousness for at least a short while. There may also be new laws, and new regulations from the agencies governing access to the Internet. Vendors may change the way they test and market their products and not all the possible changes may be advantageous to the end-user (e.g., removing the machine/host equivalence feature for remote execution). Users' interactions with their systems may change based on a heightened awareness of security risks. It is also possible that no significant change will occur anywhere. The final benefit or harm of the incident will only become clear with the passage of time.

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

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

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

## 7. Fifteen Years Later

The previous sections of the paper described the behavior of the Internet Worm and some of the aftermath. It is instructive to look back on that episode to see how (and if) the events in the interim have changed their significance. In the intervening years we have seen the consolidation of the various regional networks into the single Internet, the creation of the WWW, the increasing dominance of Windows platforms, the introduction and explosion of Internet commerce, and growing internationalism of computing. We have also seen a steadily-rising level of computer vandalism and crime.

### 7.1. Malicious Code

In the years since the Internet Worm, we have seen a steadily increasing number of incidents of malicious software. In 1988 new viruses were appearing at the rate of no more than about one a month, and there was a nascent anti-virus industry.[25] In 2003, there is a huge international industry in anti-virus technologies, and new malware instances are being reported to vendors at an average rate of over 10 per day. Luckily. most of those viruses are not well-established in the general network population and will not go on to spread to many machines. However, anti-virus protections still need to be updated on a regular, near-daily basis as a precaution.

The 1988 worm was not a virus by any currently accepted definition. However, those definitions have remained unclear and imprecise. The Blaster and SoBig.F codes of late 2003 were really Trojan Horses (users needed to run them from their mailers, and were tricked into doing so), but were referred to in the press and online as ”viruses” or ”worms.” Some of this confusion is undoubtedly the result of the original definitions being imprecise, and also a result of the inexact distinction in the minds of average users as to the boundaries of operating systems, applications, data, and networks. That same blurring is undoubtedly responsible for much of the success of malware authors: Unsafe macro languages and directly executable attachments in email have been key to many attacks.

It is notable that one particular vendor has been the platform of choice for so much malware over the recent past. Well over 95% of the reported viruses and worms are directed at products by Microsoft Corporation. Some people argue that this is because of their dominant position in the industry, but a careful examination of the underlying technology suggests that fundamental choices of architectural design and poor software quality have also played a major role.

Whatever the reasons, we have seen incidents that have involved hundreds of thousands of machines, and have peaked in a matter of minutes while causing extensive damage — some estimated in the billions of dollars. The Sapphire/Slammer worm of early 2003 demonstrated how quickly a worm can propagate on the Internet; given appropriate pre-positioning and planning, a worm that could infect a majority of victim hosts in a matter of minutes is certainly possible.[26]

It is depressing to note that the overall resistance of hosts on the Internet to malicious software seems to have gotten worse, by some measures, since 1988. For instance, the Internet Worm managed to affect at most 10% of the machines on the Internet because of a diversity of operating systems. In 2003, we have a much less heterogeneous collection of machines connected to the Internet, thus enabling wider spread of software exploiting a security flaw. In 1988, the majority of people operating systems were professionals with computing backgrounds; in 2003, the majority of machines connected to the network are operated by personnel with little, if any, system administration background. In 1988, an author of malicious code needed to know some machine language programming; in 2003, anyone with access to a text editor and WWW browser can write malicious software using macros and downloaded root kits.

### 7.2. Software Flaws

The Internet Worm exploited three kinds of flaws in widely-distributed software: exploitation of trust relationships, buffer overflows, and poor default configurations. Sadly, all three of these problems continue to exist, and are (in some ways) worse than in 1988.

In 1988, the Worm exploited trust relationships in the rsh/rlogin/rexec suite of software to transfer itself from machine to machine without authorization. In 2003, that software is still available on many systems. Worse, other questionable trust relationships have led to significant security problems. For instance, the lack of separation of privilege and function in Windows allowed viruses in macros attached to word processing documents and spreadsheets to access address books and mailers to spread themselves. At a network level, system administrators who have configured firewalls to pass traffic by default (permitted unless denied) have repeatedly been hit by software exploiting flaws. Users regularly fall for fraudulent email soliciting for credit card information or personal details while displaying logos and email addresses resembling those of well-known entities. Other examples of exploitation of faulty or misguided trust relationships abound.

Buffer overflows have been known to be a problem for decades. Despite that, unsafe routines have been standardized in the C programming library, and overflows continue to drive security problems. Serious security flaws in widely-used software are currently being reported at an average rate of between 20 and 30 per week. (As this article was being finalized, yet another critical security flaw involving buffer overflows was published, involving the venerable sendmail program.) Examining these flaws, as categorized in one of the vulnerability databases such as the CERIAS Cassandra service, or the NIST ICAT database, reveals that more than 25% of the reported flaws can be traced to buffer overflow, and perhaps as many as 3/4 of all vulnerabilities are simple argument validation errors.

It is appalling that commercial software is still being produced and shipped with buffer overflows. It is beyond the scope of this paper to analyze all the reasons why this is so, but it is clear that the problem has not gotten any less important in fifteen years. It is sobering to realize that our overall infrastructure security might well be better had UNIX been written in Cobol rather than C.

Poor default configurations also continue to plague us. The standard installation of Windows software, for instance, has various servers running and active on network ports that are not usually needed. This can be contrasted with an installation of MacOS X that has no servers enabled by default. Of course, Windows is not the only culprit — software on routers, network appliances, and systems by other vendors all share this problem in common. Some distributions of WWW servers have contained default example scripts with known vulnerabilities. The usual explanation given for these choices is that users do not understand the complexity of the options and interfaces involved, and it is necessary to enable the services to keep from generating too many complaints and help requests. This is not far away from the reason the DEBUG command was left enabled in the 1988 distributions of sendmail — to enable support of users who did not understand how to configure their mailers. There is clearly an unmet need for better user interfaces and documentation to address these problems.

### 7.3. Incident Response

In 1988, response to the Worm was largely ad hoc and coordinated via mailing lists. The CERT/CC was formed to act as a clearinghouse to help coordinate responses to future such incidents. In 2003, the situation is not much improved. System administrators often get news of new problems via mailing lists such as BUGTRAQ or newspaper stories. Judging by the number of sites that are regularly exploited via flaws for which announced patches have been available for months, it would seem that notices of flaws and fixes are not getting distributed widely enough.

The CERT/CC is currently of questionable impact in incident response. Personnel at the CERT/CC release announcements of flaws and fixes weeks or months after mailing list announcements, if at all. Paying customers may be able to get more timely announcements from the CERT/CC and other vendors, but that is not serving the general network public. The CERT/CC appeared to play little role in the responses to several recent worms and viruses. Furthermore, no organization, including the CERT/CC is collecting reports of even a majority of security incidents to be used in actuarial studies.

It is interesting to note that the newly-formed Department of Homeland Security has announced a partnership with the CERT/CC to establish a US response capability. One might question whether our experience with the CERT model supports such a move as the best approach or whether a new paradigm should be explored. The dedicated response center model also does not reflect what we learned in the Worm incident, and in times since then: a distributed response, with many people working together, is more effective than a single center of expertise. At the least this move fails to recognize a key underlying aspect of the problem: the Internet is not US-only.

Another lesson from 1988 that has not been learned is that communication is critical in addressing the problem. The teams working to decompile the Worm communicated results to each other and to the public using the Internet. When their computers went down or off-line, they were often left without access to phone lists or email, thus inhibiting their ability to communicate. In 2003, we have an increasing dependence on cell phones and voice over IP (VoIP). We saw during the 9/11 incident and the August 2003 blackout of the East Coast that cell phones were not dependable during a crisis because of load and power issues. Voice over IP has vulnerabilities in the same way — without power, the routers won't run, and without the network, the calls cannot go through. Within a few years, a virulent worm that attacks routers and power system SCADA controllers could well disable the very communication we need to combat it!

### 7.4. Laws and Professional Ethics

As predicted, numerous laws against computer misuse were enacted in the years after the Worm. However, despite the passage of those laws and the tens of thousands of viruses and worms written since then, fewer than a dozen people have ever been convicted of crimes related to malware. In part this is because it is difficult and expensive to investigate and prosecute such crimes. It may also be caused, in part, by a lack of tools and protocols to adequately investigate such acts.

Not every jurisdiction has laws against the authorship of malware. For instance, when Onel de Guzman was identified in the Philippines as the author of the 2000 ILOVEYOU Trojan/virus, he was not prosecuted there because there was no law in effect at the time prohibiting what he did. Many countries in the world still have no laws against releasing malicious software into the public. Where laws do exist, the necessary investigative technology is likely to be poor, and the cooperation across international borders may be ineffective. Investigating a crime scene comprising 2 million computers around the world presents a daunting challenge!

Members of the press and public continue to portray computer criminals sympathetically, or even heroically, although this is a problem that is slowly changing. Increasing levels of fraud, identity theft, spam, viruses and other on-line misbehavior has helped change the public willingness to view computer criminals as simply misguided geniuses. The staggering levels of loss from computer crime and malware are also helping to reverse public sympathies.

One issue that is facing us currently is the nature of intellectual property and fair use online. Although not malicious, per se, it will define much of our legal and moral landscape in the years to come. Already we have seen intellectual property owners equating unauthorized copying of their materials with piracy (a violent crime). Legislation (the Digital Millennium Copyright Act) has been enacted in the US to stem unauthorized copying but that also has a chilling effect on research into security tools. Some intellectual property owners have even sought legislation to immunize them from legal sanction for the creation of destructive malware aimed at "pirates." This trend is disturbing — having viruses and worms being written for vigilante purposes is not likely to make any of us safer.

Another disturbing trend involves unwanted email, or "spam." Recent events suggest that some spammers may be writing viruses and Trojan programs as a means of collecting addresses and subverting third-party machines to act as distribution engines. Given the number of vulnerable machines on the network, this may become a major problem for security and law enforcement specialists and make the Internet Worm appear exceedingly benign in hindsight.

Reflecting on the sentence Mr. Morris received, it is clear that he acted foolishly, and (according to the court) criminally. However, the few thousand dollars in damages caused by the Internet Worm pale in comparison to the billions of dollars in damages caused by others since 1988. Comparing Mr. Morris to some of the computer criminals who have been active in the last 15 years makes it clear that the lack of jail time was probably a correct decision in his case. It is also likely that the desired deterrent effect of his conviction was minimal, at best.

## 8. Parting Thoughts

It has been 15 years since the Internet Worm. That is approximately 1/2 of a human generation, and approximately six "Internet Generations." Reflection on what has happened in that interval reveals that the community either failed to learn the lessons inherent in that attack, or we have failed to value them. Systems are deployed with inexcusable flaws, networks are configured with misplaced trust, and incident response is uncoordinated and of minimal effectiveness. What is often missed in this kind of retrospective is that those lessons were not new in 1988, either.

As a professional group, computer scientists and engineers have shown surprisingly poor attention to learning from the past. As a community, we frequently address problems as if they were unique, and come up with specialized solutions that are not seen as related to past experience or some underlying truth. Our scientific base seems to have been reduced to only those documents and software that reside on the WWW, and that horizon is remarkably closer than our experience warrants.

In 1988 I was hopeful that we could make changes for the better in how we built, configured and deployed our computing systems. In 2003, with 15 more years of experience, I have become more cynical about how we will address the challenges we face. As such, I fully expect to be writing a paper in 2013 or 2018 that looks back at this time as one where we did not yet know how bad it was going to get, and that these observations are still current. As I wrote in 1988, ”It remains to be seen.”

## References

[1] L. Adleman. An abstract theory of computer viruses. In *Lecture Notes in Computer Science, vol 403*. Springer-Verlag, 1990.

[2] E. Allman. *Sendmail–An Internetwork Mail Router*. University of California, Berkeley, 1983. Issued with the BSD UNIX documentation set.

[3] M. Bishop. An application of a fast data encryption standard implementation. *Computing Systems: The Journal of the Usenix Association*, 1(3):221–254, Summer 1988.

[4] J. Brunner. *The Shockwave Rider*. Harper & Row, 1975.

[5] F. Cohen. *Computer Viruses*. PhD thesis, University of Southern California, 1985.

[6] P. J. Denning. The Internet Worm. *American Scientist*, 77(2), March-April 1989.

[7] M. W. Eichin and J. A. Rochlis. With microscope and tweezers: An analysis of the internet virus of November 1988. In *Proceedings of the Symposium on Research in Security and Privacy*, Oakland, CA, May 1989. IEEE-CS.

[8] T. Eisenberg, D. Gries, J. Hartmanis, D. Holcomb, M. S. Lynn, and T. Santoro. *The Computer Worm*. Office of the Provost, Cornell University, Ithaca, NY, Feb. 1989.

[9] D. Ferbrache. *A Pathology of Computer Viruses*. Springer-Verlag, 1992.

[10] S. Garfinkel, A. Schwartz, and G. Spafford. *Practical* UNIX *and Internet Security*. O'Reilly & Associates, 2003. 3rd edition.

[11] D. Gerrold. *When Harlie Was One*. Ballentine Books, 1972. The **first** edition.

[12] F. T. Grampp and R. H. Morris. UNIX operating system security. *AT&T Bell Laboratories Technical Journal*, 63(8, part 2):1649–1672, Oct. 1984.

[13] H. J. Highland, editor. *Computer Virus Handbook*. Elsevier Advanced Technology, 1990.

[14] K. M. King. Overreaction to external attacks on computer systems could be more harmful than the viruses themselves. *Chronicle of Higher Education*, 23 November 1988.

[15] R. Morris and K. Thompson. UNIX password security. *Communications of the ACM*, 22(11):594–597, Nov. 1979.

[16] Participants. *Proceedings of the Virus Post-Mortem Meeting*. National Computer Security Center, Ft. George Meade, MD, 8 November 1988.

[17] B. Reid. Reflections on some recent widespread computer breakins. *Communications of the ACM*, 30(2):103–105, Feb. 1987.

[18] D. M. Ritchie. On the security of UNIX. In *UNIX Supplementary Documents*. AT & T, 1979.

[19] D. Seeley. A tour of the worm. In *Proceedings of 1989 Winter Usenix Conference*, San Diego, CA, Feb. 1989. Usenix Association.

[20] J. F. Shoch and J. A. Hupp. The worm programs – early experience with a distributed computation. *Communications of the ACM*, 25(3):172–180, Mar. 1982.

[21] E. H. Spafford. An analysis of the Internet worm. In C. Ghezzi and J. A. McDermid, editors, *Proceedings of the 2nd European Software Engineering Conference*, pages 446–468, Sept. 1989. Issued as #87 in the Lecture Notes in Computer Science series.

[22] E. H. Spafford. The Internet Worm program: An analysis. *Computer Communication Review*, 19(1), Jan. 1989. Also issued as Purdue CS technical report TR-CSD-823.

[23] E. H. Spafford. Are computer break-ins ethical? *Journal of Systems & Software*, 17(1):41–48, Jan. 1992.

[24] E. H. Spafford. Virus. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. John Wiley & Sons, 1994.

[25] E. H. Spafford, K. A. Heaphy, and D. J. Ferbrache. *Computer Viruses: Dealing with Electronic Vandalism and Programmed Threats*. ADAPSO, Arlington, VA, 1989.

[26] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the internet in your spare time. In *Proceedings of the 11th Usenix Security Symposium*. Usenix Association, 2002.

[27] B. Stubbs and L. J. Hoffman. Mapping the virus battlefield. In L. J. Hoffman, editor, *Rogue Programs: Viruses, Worms, and Trojan Horses*, chapter 12, pages 143–157. Van Nostrand Reinhold, New York, NY, 1990.