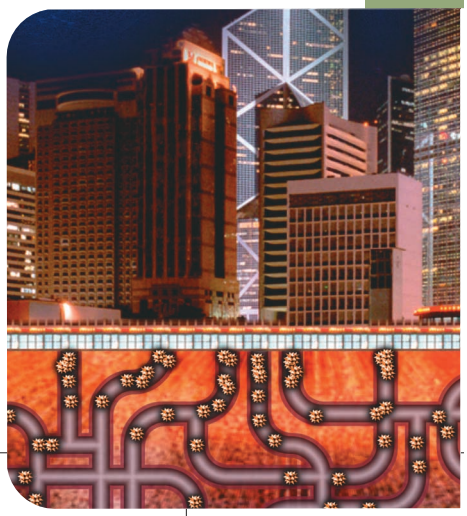
The Morris Worm: A Fifteen-Year Perspective



The Morris worm was the first worm to hit the Internet and caused a disruption never seen before. In the 15 years since its appearance, have we learned our lessons about computer security?

Hilarie Orman
Purple Streak

On the evening of 2 November 1988, a brush fire got out of control on the Internet and set at least one computer in 20 on fire, figuratively speaking. This was due to an event that was either the Internet's first mobile agent experiment or a new entry in the annals of computer vandalism: the infamous Morris worm. The work of Robert Tappan Morris, a Cornell graduate student in computer science, the worm caused those connected to the Internet much consternation.

System administrators at sites infected by the Morris worm spent at least a day fighting what initially was a mysterious enemy. By early in the morning on 3 November, Unix-based computers had slowed down to only a small percentage of their usual capabilities and email was bogged down in a hopeless mire. As the day wore on, the little information available about the problem was not comforting: a software worm was using the sendmail program for Unix systems (an omnibus application dealing with many aspects of email sending and receiving, especially the Internet SMTP email protocol) and the C compiler to replicate. It loaded code from other computers into itself, and it had more than one method of invasion. No one was sure if it did anything else, if it would do anything else to their computers, or whether it was the work of a single person or a group of attackers.

Today, the Morris worm is remembered as the first of many such attacks, as what might have been a wake-up call to system administrators and security researchers, and as the first certain signal to those who still held utopian beliefs about the Internet, that it was not to be a friendly place. What is the Morris worm's legacy to computer security from a 15-year perspective? In this article, I consider two areas: the development of defensive measures and understanding what, if anything, distinguishes a destructive from a nondestructive worm.

The immediate reactions in 1988 to the massive assault on user accounts all over the Internet now seem curiously naïve. Posters to the RISKS Digest (www.risks.org), a forum on computer risks, were outraged that a student had done such a thing. Many called for an emphasis on computer ethics. Was this an example of mischief or research, or had someone confused the two?

This was a time of transition for the Internet, and while the Morris worm might not have been a watershed event, it did mark an uneven boundary between the largely trusting Internet of the time and the heterogeneous, dangerous world-wide Internet of today. Since then, firewall and antivirus protection industries have emerged and matured, and a new generation of security experts in industry and academia has come of age. How much influence did the Morris worm have on them and how Internet security is now approached?

Setting the stage

Computer security on the Internet has never been anything but a vague intention. There is no formal model of secure operation for the Internet or the applications using it, and there is a long history of vulnerabilities in core applications. This was certainly true in 1988, although most sites probably had seen many cases of unauthorized access of some kind, and Unix systems had become favorite targets. The Unix operating system (OS) was the first to have Internet protocols and services built in as native services that facilitated research in networked computer services, and thousands of systems were running that soft-

ware or its derivatives in 1988.[1] Security had not been a major consideration in the Internet's or Unix's original design, and the Unix networking extensions set no new standards in the area. As the years passed, knowledge about subverting Unix access permissions abounded and spread. The number of loopholes, and their varieties, had begun to look unmanageable to many system administrators and computer-security experts. Two camps developed, one hoping to fix all the problems, and another advocating keeping one step away from the Internet.

The US Department of Defense (DoD) was an early adopter of the Internet and its protocols, having funded the technology's development for years. Their concerns about security caused them to separate their network, the Milnet, from the Internet, maintaining only a handful of well-controlled points where packets could traverse the two systems. The packet-exchange points were originally meant to strictly limit network services, but in 1988 there were no controls in place.

Larger sites were moving away from having one omnibus computer connected directly to the Internet, handling all users' computational and communication needs. This change was slowly coming about through the introduction of low-cost desktop machines and local area network (LAN) technology. An emerging configuration involved connecting one machine to both the Internet and the LAN and sending all network communications through it to hide the desktop machines' insecure aspects.

However, not all sites could afford such a separation, and not all sites wanted a two-step process to use the Internet. Connecting a Unix workstation directly to the Internet had several advantages in terms of resilience and load-reduction. Furthermore, network services could be individually tailored, with some users testing experimental network services.

Many in the field were already aware of computing's darker side because computer viruses had become a serious problem—though not on Unix systems. Infections on DOS diskettes, especially in the boot sector, had become notorious and all too common. Many Unix users dismissed this as a "hygiene" problem that mainly afflicted DOS users who shared game programs frequently. Unix was largely a programmer's environment, not that of a game player. Users accepted that there was a degree of risk inherent in Internet connections, but the advantages of email and newsgroups were appealing, and the ease of remote machine access was invaluable to many cooperating developers. This was the unsettled environment that the Morris worm entered.

Why a worm?

To some people, the idea of an Internet worm is exciting. They delight in the thought of every computer on the Internet spending part of its time running their program. A few hundred times each second, perhaps, computers around the world would run their secret application and no one would know. This might have been the main intent behind the Morris worm. No one had yet done this; no one knew if it were possible; but they were about to find out.

The Morris worm appears to have been designed to be ubiquitous and unnoticed. Although it achieved neither goal, it is an interesting example of how a relatively simple program with no outright intent to disrupt operations can cause widespread havoc through its secondary effects. Possibly the first denial-of-service (DoS) attack ever seen on the Internet, the Morris worm brings to the fore some interesting questions about graph connectivity among networked systems.

Certainly no one had any idea how fast a worm could spread between Internet systems. Fred Cohen, in his doctoral work at USC, had shown that software sharing led to rapid infection rates on single systems and between systems with shared software, but that rarely, if ever, involved self-propagating applications. Although propagation-rate models existed, they depended on the software-interchange frequency between machines, and this parameter was not relevant to most Internet systems because of their limited use of shared software. To estimate the spread rate for a new kind of attack, an analogous parameter for the Internet had to be determined, and the degree of connectivity between systems on the Internet based on that parameter had to be measured. Little was known about such a parameter. Notably, some systems exchanged little more than email, which seemed a safe way to share information via the Internet in those days. Many assumed that systems connected only by email were isolated from worms. The Morris worm's lightening-fast spread proved that this was terribly wrong. Looking at its design, we can see why it was so effective and resilient, in spite of—or perhaps because of—its flaws.

Worms and graphs

Because we do not know the Morris worm's true intended purpose, it is useful to invent the problem that it might have been trying to "solve." Based on MIT's examination of the code as reported in Eugene Spafford's Purdue technical report (see Eugene Spafford's *The Internet Worm Program: An Analysis* at ftp://coast.cs.purdue.edu/pub/doc/morris_worm/spaf-IWorm-paper-ESEC.ps.Z, and Mark W. Eichin's and Jon A. Rochlis' *With Microscope and Tweezers* at ftp://coast.cs.purdue.edu/pub/doc/morris_worm/mit.PS.Z), the worm's main goal seems to have been to establish a software foothold on as many machines as possible while running perpetually and undetected. It planned to accomplish its coverage using neighbor information that was available only on infected machines, thus spreading in a classic branching pattern.

Propagating software through an arbitrary graph of computers is an interesting problem in distributed com-

puting. Formally, all computers running the Internet protocol (IP) form a graph. Those with Internet connections form a connected graph on which each computer is a node. In a sense, IP connects all pairs of computers on the Internet, creating a full-mesh graph. In reality, computers are very selective about accepting IP packets. Some packets are discarded because they name unknown or unreachable destinations (32-bit addresses). Even when packets are delivered to their destinations, the receiving node might discard them, depending on subfield information such as the protocol (usually TCP or UDP) and port number. Getting a valid triplet of IP address, transport protocol, and port number accepted at a computer is a challenge I will call the endpoint-identification problem, but beyond that, accessing the target computer requires that it have an application that accepts and acts on the packets. For the worm to spread, that application also must be vulnerable or trusting. We can envision each such application as a directed link color for the graph.

The goal of all worms is to establish a subgraph by occupying nodes and establishing new links of any color. For each link in the original graph, zero or more colored links are possible. A worm tries to maximize its subgraph by choosing strategies that are likely to invade other nodes.

A surreptitious worm tries to evade detection by minimizing the visible resources that it uses. To that end, it will try to avoid invading a node more than once, but if it does so, it will try to avoid having more than one instance of itself occupy the node. Ideally, this should result in a nearly stable steady-state connected subgraph with an inward link degree of one.

Workings of the worm

The Morris worm was a mixture of sophistication and naïvety. It had a simple overall design: look at a computer's system configuration to find potential neighbors, invade them, and try to minimize the number of invasions on any machine. The worm used heuristic knowledge about Internet topology and trust relationships to aid its spread, and it targeted two different machine architectures. Its cleverness in finding potential attack targets made it especially effective, but it also took on the time-consuming task of guessing passwords on individual user accounts, which gave it an "attack in depth" aspect. Nonetheless, it became a victim of its own success as it was unable to control its exponential growth. With no global information and no point of control, the Morris worm ran rampant.

Design attributes

Some of Spafford's findings point to aspects of the worm's innovative design:

- It attacked one operating system, but two different computer architectures.
- It had three distinct propagation vectors.
- It had several mechanisms for finding both potential nodes to infect, particularly information about the local system's IP connectivity (its network class and gateway), and information found in user accounts.
- It traversed trusted accounts using password guessing. The worm made heavy use of this computationally intensive method by employing four information sources: accounts with null passwords (no password), information related to the user account, an internal dictionary, and a word list on the local machines, /usr/dict/words.
- It installed its software via a two-step "hook and haul" method (explained later in the "Inside the worm" subsection) that required the use of a C compiler, link loader, and a callback network connection to the infecting system.
- It evaded notice by obscuring the process parameters and rarely leaving files behind.
- It attempted to limit the reinfection rate on each node (but not the total number).
- It attempted to run forever on as many nodes as possible.

Although there had been worms before, no one had tried to run one on a complex topology. For this worm to achieve its purpose of widespread propagation, it had to discover local topology in an arbitrary graph. As an experiment, it might have been considered a brilliant beginning for work in self-organizing systems.

Design flaws

The Morris worm also contained some noteworthy design flaws:

- It was overly aggressive. Although it did have a way to notice multiple infections, its rate-limiting behavior was not effective, and a hundred or more copies could be running on a single machine. Even uninfected machines were vulnerable to assault through multiple infection attempts coming from many independent sources.
- As the number of infections increased, the worm's ability to limit itself decreased. Race conditions in its detection method actually caused the infection rate to increase.
- It could not trace its progress or control it.

Although there had been worms before, no one had tried to run one on a complex topology.

- Log files, particularly sendmail logs, contained information about the worm's usage. Some log files filled up with the information, and the machines' I/O load increased.
- Its infection method depended on the C compiler, thus preventing access to some major sites that had already established machines that acted as bastions, limiting network access. These machines might not have had C compilers.
- A variety of resource failures left many copies of the "hook" program on the attacked machines.

The intensity of the attacks on machines running the SMTP protocol, particularly those running the Unix sendmail program, resulted in denial of email service to large portions of the Internet. This was the Morris worm's most disruptive aspect. Like many human infections, it was not the worm itself that was harmful, but its secondary effects on resources.

Inside the worm

Once launched, the worm moved from node to node using only itself and the infected node's local information. The worm did not receive information from other worms. This simplicity was probably a blessing and a curse, because it minimized the prerequisites for gaining a foothold, but it also made the worm difficult to control. Worms like the Morris follow these steps to establish a graph link.

1. Choose an endpoint.
2. Choose a vulnerable application.
3. Compute authentication information (if necessary).
4. Establish a network connection.
5. Control the new, remote worm instance using four substeps: use the remote application vulnerability to propagate the worm software by sending "hook" software source code and completion instructions, wait for the "haul" network connection from the endpoint, send worm body as binary load modules, and wait for the remote system to construct the worm and call back.
6. Go on to the next endpoint.

The worm needed software vulnerabilities to infect a new node, and it also needed to find likely targets for infection. The worm had the interesting property of being able to use two different vulnerabilities, one hidden in an email application. It also had a method for breaking into user accounts and spreading to sites trusted by those users.

Mechanics of the spread

The Morris worm used three independent applications for attacking remote systems: sendmail, fingerd (a program for finding information about computer users), and rexec (a program for starting other programs on remote systems). It used telnet (an Internet standard program for gaining access to command-line interpreters—a type of lowest-common-denominator application) to probe the system and determine whether an attempt to establish the connection would work. Not all systems supported all these applications, but most had SMTP, and if they were running Unix, the sendmail program was probably behind the SMTP. The Morris worm could propagate to systems that had only sendmail and not the other applications.

Part of the Morris worm's brilliance was the way it approached the endpoint-identification problem. In 1988, it might well have been possible to obtain IP addresses for every computer on the Internet and incorporate that database into a worm program. Instead, the Morris worm relied only on information available at each node. It looked for three endpoint types:

- neighbors trusted by administrators or individual users,
- local neighbors selected at random, and
- gateways to distant machines.

The first group was described in the Unix files `/etc/hosts.equiv` and `/.rhosts` (hosts trusted by the superuser) and in individual users' `.forward` and `.rhosts` files. While the first two files were generally limited to machine names in the local organization, the last two file types might name distant machines on the Internet on which the user had an account. These represented trust relationships established by the user on the basis of organizational memberships, but not controlled by administrators. The `.forward` file was especially useful because it named a machine where a user expected to receive email, which was a good indication that the machine ran SMTP, and provided an entry point to the sendmail application on Unix systems.

The second group of possible endpoints was based on an infected machine's network address types, which served as a rough guide to the address pool that might be assigned to an organization's machines. Not all possible addresses were assigned (this remains true, even today), so random guessing was the main strategy.

The members of the third endpoint group were gateways: nodes that would forward traffic to other Internet networks. Although attackers might not be able to attack all the gateways (which might be routers and, therefore, not susceptible to the Morris worm's attack methods), those that were vulnerable made powerful graph nodes because of their higher degree of connectivity to the outside world.

These three classes were key to the worm's spread; they determined the Internet graph's "degree of separation," with respect to the three types of graph links. To replicate, the Morris worm needed links to vulnerable applications running on similar OSs and with similar architectures. In 1988, few people could have made intelli-

gent guesses about how many Unix machines had common vulnerabilities, and even fewer realized that such a survey was needed. The Morris worm performed a good deal of the survey in about half a day and kept going while site administrators battled against it.

Some application links, particularly the `rsh` commands, required more information to proceed if administrators required a username and password. A startling innovation in the Morris worm was the way it worked to undermine, and even evade, authentication, by taking advantage of application vulnerabilities. The sendmail application had a powerful backdoor access method that did not need authentication, and the fingerd application was vulnerable to a buffer overrun exploit, something of a novelty at the time. Many systems did not run sendmail or fingerd, or else they had versions that did not include authentication shortcomings, but such systems were still vulnerable through the `hosts.equiv` files, which allowed BSD Unix systems to trust each other with respect to remote services, or through the `.rhosts` files, which let individual users determine whom they trusted for remote access to their accounts. In many cases, they trusted themselves on several machines—sometimes with a password, and sometimes without.

Even with a password requirement, users were unlikely to use a different password on every machine they used. If the worm running on machine A learned the password (P) of user U, and if it knew that U was also a user on machine B, it could likely use the triple B,U,P to access machine B via the remote-execution facility. Therefore, the Morris worm put a lot of effort into learning passwords by intelligent and efficient guessing.

Having established a graph link, a worm must have some means to replicate itself. To do so, the Morris worm needed a handful of resources on the machine it was attacking:

- a place to write files,
- a C compiler and link loader,
- the ability to create a network connection back to its parent node, and
- permission to launch a new program.

Not all the machines in the worm's mesh had these resources, and these machines were "dead ends" for the worm. They were subject to repeated infection attempts, especially through email, but they were not the launching pads for new infections.

Application vulnerabilities

Of the three applications the Morris worm used for gaining appropriate access to replicate on a new system, both fingerd and sendmail had serious security problems. Fingerd processed its invocation parameters using the library function `gets` (get string), which did not have bounds-checking. Thus, if fingerd were invoked with a long, carefully constructed parameter, it would overrun the stack area and execute the parameter. The Morris worm used this vulnerability to start a remote shell (command line interpreter on another machine). Donn Seely, in his University of Utah tech report (ftp://coast.cs.purdue.edu/pub/doc/morris_worm/seely.PS.Z), said the `gets` routine dated from "the dawn of time," showing how little attention was paid to potential vulnerabilities in those days, and also showing how a seemingly minor vulnerability can become both widespread and exploitable in a networked environment.

The sendmail program had two problems that let the worm replicate. As Eric Allman, sendmail's creator, tells it (also in Seely's tech report): "[T]he trap door resulted from two distinct 'features' that, although innocent by themselves, were deadly when combined (kind of like binary nerve gas)." The two features were `DEBUG` mode and the `run program` command. An outsider establishing an SMTP connection to the sendmail program could enable `DEBUG` and then have permission to use the `run program` command. In theory, the `DEBUG` mode was meant only for temporary use by a programmer when debugging the sendmail program, and was not an available option in ordinary service; in reality, many sites had `DEBUG` enabled, albeit unwittingly. In theory, remote users could not access `run program`, but in reality, `DEBUG` mode enabled `run program` for remote users. Having enabled `DEBUG`, a remote user could send commands for execution, in privileged mode, on the remote computer. The Morris worm sent commands for the command line shell `sh`.

Once launched, a worm must make progress while controlling its energy distribution. The energy of software is roughly the number of computer instructions it executes, although it can include other things, such as I/O or network communication. We do not know much about worms' energy distribution with respect to time and place, but we do know that they tend to overwhelm some locations and result in a DoS attack. This would undermine the purpose of being undetectable, for a worm with such a goal.

In an ideal worm, each infected node spends the bulk of its computation time trying to infect uninfected hosts. The Morris worm did not have this property. In order to achieve an even energy distribution with a purely local algorithm, a program must depend on discovery methods that either proceed along something resembling a spanning tree, or it must have a very low-cost method of determining that another endpoint is already infected. There are some trade-offs of resiliency versus redundancy inherent in either scheme. An infected node might encounter an unexpected resource limit on one node and die; if it cannot be "refreshed," the ubiquitous-coverage goal will not be met. On the other hand, if the worm makes too

many infection attempts on one machine, it "wastes energy" and risks detection, as the Morris worm did.

The Morris worm spent a lot of time infecting the same systems many times. In some organizations, it seemed to have an n^2 effect, with all nodes trying to infect each other. Although two nodes could detect that they were both infected and that they should drop their infection attempts toward each other, they did not do this until the redundant infection had done a certain amount of work. In a simple-minded attempt at resiliency, the worm allowed one in seven infections on a node to persist indefinitely, resulting in unbounded energy consumption because there was no countervailing force killing off the accumulating infections. In any event, after 12 hours, an infection either died or started retrying its infections, often reattacking previously infected hosts. This kept the Morris worm operating at a frenetic pace.

The worm used an interesting hook-and-haul method of propagation that masked its entrance to a site and kept its mechanisms secret.

The worm used an interesting hook-and-haul method of propagation that was meant to mask its entrance to a site and keep its mechanisms secret. It was also multifaceted and multi-architecture, using multiple methods to gain entrance to a machine and affecting two entirely different computer architectures. It had an intensely computational part that was meant to give it resilience through the ability to infiltrate through many user accounts, but it had an ineffective mechanism to limit its growth rate.

Responses in the worm's wake

It would be gratifying to learn that the Morris worm had caused a sea change in attitudes toward computer security, and that there never again was an Internet worm. As this article is being written, however, the Blaster Worm is making the news and showing some remarkable similarities to the Morris worm. After 15 years, what has changed?

For a few years after the Morris worm hit, computer science departments around the world tried to delineate the difference between appropriate and inappropriate computer and network usage, and many tried to define an ethical basis for the distinctions. This approach had two purposes: to control harmful Internet behavior and to avoid any strict policy controls on innovators. The ethics efforts might have stemmed a tide of clever exploits similar to the Morris worm, but they probably passed unnoticed by the rapidly growing Internet user community, which included students of all subjects and all ages around the world. The Internet ceased to be a community in the sense it had been because it had to accept as netizens many uncontrollable sociopaths in its population, as immediate events would show.

Nearly a year after the Morris worm incident, the WANK worm spread through sites on some research networks running the DECNet protocol and services, Digital Equipment Corporation's (DEC) alternative networking technology. Because these machines were not connected to the Internet, the infection was more limited than the Morris worm attack. However, the attack was interesting because DECNet users had largely enjoyed immunity from the Morris worm.

Although the Internet had begun as an experiment, and then as a place for doing protocol experimentation, most users and their system administrators in 1988 had begun to expect and desire reliability and consistency. Their aversion to using the Internet for experiments solidified in the Morris worm's wake, and perhaps stifled investigations into the dynamics of self-organizing agents. On the other hand, the Internet was growing more commercial, and experiments are rarely appreciated on commercial infrastructures. Many lamented the passing of the days when universities could easily deploy new Internet services, but in only a few years they would witness the Internet's transformation by the Web and its myriad new services.

CERT's emergence

Soon after the Morris worm incident, DARPA provided funding for the Computer Emergency Response Team (CERT), which has since served as a clearinghouse for security-vulnerability information. Although CERT provides no panacea, it has become a trusted source of information about security flaws and fixes for a variety of software. However, it merely documents problems and vendor solutions; it is powerless to change the way the industry produces software. A look at its database reveals continuing trouble in the same areas that the Morris worm highlighted. There are at least 200 reports relating to buffer overrun problems on many major OSs. Possibly because there is even less variability in machine architectures today than in 1988, the problem's exploitation has become routine and highly effective. There is even some evidence that systematic testing of all parameter-processing pathways in programs that accept network data is revealing an increasing number of problems. The bad guys might have latched on to some of the tools that were meant for software developers.

The sendmail program is used still, and still is an omnibus program with configuration possibilities that far ex-

ceed its expected use. CERT's database contains 50 reports of sendmail problems, two of them from this year, both of which were buffer overruns.

Nonetheless, the 1988 experience inspired some researchers to tackle the problems directly. For example, Spafford and Dan Farmer of CERT developed the Computerized Oracle and Password System (COPS),[2] which embodied a cornucopia of information about Unix systems and their security and gave system administrators an automated view of system vulnerabilities. Additionally, Spafford and Gene Kim at Purdue University created the TripWire tool, which detects file changes that could signal malicious alteration.

Although these tools were useful, new vulnerabilities kept showing up, and constant updates to COPS were needed. Moreover, COPS was directed at Unix systems, and other operating systems, such as Microsoft Windows, Novell's Netware, and IBM's OS/2, were coming onto the Internet.

Later, Dan Farmer and Wietse Venema developed the Security Administrator Tool for Analyzing Networks (SATAN), which was designed to check for network vulnerabilities, but administrators had become so sensitized to their systems' fragility that some found it hard to distinguish between SATAN and an outright attack.

Firewalls

The biggest security change to affect the Internet is the firewall, those nearly ubiquitous protectors of Internet computers. Because these devices began appearing subsequent to the Morris worm (the first edition of Steve Bellovin's and Bill Cheswick's *Firewalls and Internet Security* was published in 1994), they seem to be a tangible response to it. However, the history of firewall technology begins before the worm and seems to ripple right through it, developing into a cascade after most people had relegated 2 November 1988 to the memory of a bad day at the office.

The first attempt at a firewall might have been that of Bolt, Beranek, and Newman (BBN), a small consulting firm. Their Milnet mail gateway, a project intended to enforce a near separation of the Milnet from the Arpanet. Worried early on about the possibility of network attacks, the DoD planned to severely restrict network services between its computers and the rest of the networked world, and email was the only thing they allowed as an essential service. The mail gateway had a policy-driven engine and could be configured to allow only packets for the SMTP protocol through its communication channels. This might have been the first packet-filtering gateway. Ironically, even if it had been in place in 1988, it would not have stopped the sendmail exploit. As it was, Milnet operators used nonfiltering gateways, and they simply disconnected them when they heard of the Morris worm.

At DEC, although unaffected and generally unimpressed by the Morris worm, a handful of people began putting effort into corporate network protection. Brian Reid, who had seen firsthand the devastating effects of exploits using the Berkeley Unix remote access commands while at UC Berkeley in 1986, developed a corporate firewall using a two-step access approach (login to firewall, then to internal services). Jeff Mogul, inspired by Debra Estrin at USC and others interested in Internet security, developed a generic packet-filtering capability, similar to the BBN Milnet mail gateway, but meant for use in a generic OS like Unix. Based on some of these ideas, Fred Avolio and Marcus Ranum developed a commercial firewall product and extended it to use application proxies. Much of this technology, under DARPA funding at Trusted Information Systems, became a freely available toolkit for Unix systems. However, little or none of this activity was directly motivated by the Morris worm; there were plenty of other early warning signs that put the plans into motion.

By 1995, it had become nearly impossible to put a general use Unix system on the Internet without it becoming subject to an unending series of attacks. Firewalls became an essential part of any organization's network presence. At the same time, other forces, such as management of IP address allocations, brought in network address translation and restrictions on providing server access. In today's Internet environment, a worm like Morris' would have trouble moving between organizations because of infected machines would have been unable to contact the infector using the reverse TCP channel.

The biggest security change to affect the Internet is the firewall, those nearly ubiquitous protectors of Internet computers.

Much study, few improvements

The security research community responded in the years following 1988 with enthusiasm and creativity. There have been at least 80 research papers in the last 15 years addressing buffer overruns. Reseachers have proposed new languages, new automated tools for source and binary, new checking mechanisms, new OS checks, and almost every conceivable approach to ensuring that running programs cannot be tricked into executing code introduced through the process stack. Nonetheless, buffer overrun problems continue to plague commercial software offerings.

In light of this, it is dismaying to read Gene Spaf-

ford's comments in his tech report from the immediate aftermath of the Morris worm:

> "The sendmail program is of immense importance on most Berkeley-derived (and other) UNIX systems because it handles the complex tasks of mail routing and delivery. Yet, despite its importance and widespread use, most system administrators know little about how it works. Stories are often related about how system administrators will attempt to write new device drivers or otherwise modify the kernel of the operating system, yet they will not willingly attempt to modify sendmail or its configuration files. 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."

A resounding "no" echoes down through the years.

Why does research fail to yield solutions, or why do solutions go unnoticed in the course of software production? Why is it more profitable to support a security industry that responds on a case-by-case basis than to produce secure software directly? These questions probe at economics as much as at computer science, and it might be that an approach must be tied to economics or that economists must be able to better estimate the price of poor security. In any event, today's Internet security situation seems little changed, at its heart, than it did 15 years ago.

Today's Linux systems still contain the BSD remote commands, and the `.rhosts` and `.forward` files, but these are not the main focus of today's attacks. Similar facilities plague other OS, and attackers are concentrating on those.

For many years, Unix retained the reputation of having exceptionally poor security. Other OSs, with fewer entries in CERT's database, had the misleading appearance of being free from risk. A poor understanding of commercial software production practices, combined with some vendor hype, might have led some planners to choose alternatives to Unix on the grounds that they would have no vulnerabilities. History—and current events—show the folly of such complacency.

Research questions

Whatever its intentions, the Morris worm did reveal a good deal of information about how tightly bound Internet sites had become, despite organizational administrative boundaries. The user population has information that connects it in ways that are not easily modeled by transmission-line links, and that set of personal, individual links is part of what makes the Internet such a powerful sociological force. Recognition of the growing bonds between users might have shown earlier that worms would be a fact of life on such a highly connected system. However, some important questions were never resolved in the legal hubbub following the Morris worm.

Some of those questions concern the sources of virulence. Did virulence increase as user account passwords became compromised? How long did it take to move from an organization's bastion machine to others on the same network? What was the average load on machines that were unsuccessfully attacked? Were there vulnerable machines that the worm did not find? How long did it take, on average, to find a vulnerable user account, and how important were the user accounts in spreading the infection? Was the energy expended in password guessing well-spent, and could more effort have made the worm more effective?

The worm's progress is unknown. No one knows for certain which of its propagation methods were most successful, what its LAN-versus-WAN spread rates were like, or if only a few user accounts were responsible for most of its spread.

Instead of undertaking a systematic mapping of the worm's progress, individual sites fixed their vulnerabilities and warned their users against interorganizational trust policies. Users with `.rhosts` files removed them. Connectivity was reduced, but by how much? How important was each link "color" in assisting the worm propagation?

Earlier studies indicated that 20 percent or more of user accounts typically had poor password choices. Only a small amount of anecdotal information about the Morris worm's experience exists from Spafford's Purdue technical report mentioned earlier, which stated that some sites had half their passwords quickly discovered through the simple dictionary attacks.

It could be an open research topic to determine the conditions for building a low-energy, widespread, and surreptitious worm. For example, how much better could an attacker do with a worm that set up a web of communication to monitor and control its progress? Could the communication be held to a small amount per node, say $lnln$ (the mathematical notation for the logarithm of a logarithm), of the number of infections? Would that make a worm fragile, either because it would fail to keep a coherent list of infected sites, or because any communication would make it detectable, and thus, easily corrupted by defenders? Apparently worm writers have begun using covert channels in protocol fields to mask their communication, so some form of research is already under way.

On a positive note, Internet researchers have recently begun to take up the study of worm propogation. At the 2002 Usenix Security Symposium, researchers showed how a worm today could infect as many as 10 million computers.[3] Using mathematical models of Internet

connectivity and computer simulation techniques, they show that the potential for spread, once an exploitable and widespread vulnerability is found, might be higher today than in 1988.

Fifteen years later

Despite its notoriety and the numerous references to it today, the Morris worm was heeded neither as a security wake-up call nor as a graph-connectivity experiment, and its legacy is that the attack has been much imitated with continued success.

Today's Internet is a thousand times larger than 1988's, with many new services and a much larger and less-sophisticated user population. The diversity of machine architectures and OSs is probably lower than in 1988, but the number of vulnerabilities might well be higher. Almost all sites have firewall protection. There is an industry devoted to antiworm and antivirus software, quickly targeting new attacks with specific antidotes. A continual state of thrust and parry is the norm.

Vendors have a heightened sense of awareness of their responsibilities to produce security-preserving software, but this does not penetrate through to all software developers. Few universities teach software engineering with an eye toward avoiding security vulnerabilities. Either the methods are too onerous or the importance is not appreciated.

It would seem foolishly shortsighted to expect that we have seen the worst of the problems yet. Perhaps someone will again combine novel observations on connectivity and vulnerabilities into a potent mix that will sweep across the Internet unchecked. If they do, the Morris worm's history shows that we will be unlikely to develop any effective defense to it in less than two decades. □

Hilarie Orman *is the founder of Purple Streak, a security consulting and research company. While at the University of Arizona as a research scientist, she led the development of novel network security protocols and very fast network communication methods. These research interests led to the design of secure key exchange for the Internet protocols developed through the Internet Engineering Task Force. She received a BS degree in mathematics from MIT. She previously worked at the Defense Advanced Research Projects Agency (DARPA) and was chief architect for Volera. She is the editor of the IEEE online security newsletter* Cipher, *and is on the technical advisory board for Senforce, which develops solutions for location-aware computing.*

References

1. T. Eisenberg et al., "The Cornell Commission on Morris and the Worm," *Comm. ACM*, vol. 32, no. 6, June 1989, pp. 706–710.
2. D. Farmer and E. Spafford, "The COPS Security Checker System," Purdue University tech. report CSD-TR-993, Purdue Univ., Jan. 1994.
3. S. Staniford, V. Paxson, and N. Weaver, "How to Own