With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988 *

Mark W. Eichin and Jon A. Rochlis

Massachusetts Institute of Technology
77 Massachusetts Avenue, E40-311
Cambridge, MA 02139

Abstract

In early November 1988 the Internet, a collection of networks consisting of 60,000 host computers implementing the TCP/IP protocol suite, was attacked by a virus, a program which broke into computers on the network and which spread from one machine to another. This paper is a detailed analysis of the virus program.We describe the lessons that this incident has taught the Internet community and topics for future consideration and resolution. A detailed routine by routine description of the virus program including the contents of its built in dictionary is provided.

1 Introduction

The Internet[1][2], a collection of interconnected networks linking approximately 60,000 computers, was attacked by a virus program on 2 November 1988. The Internet community is comprised of academic, corporate, and goverment research users, all seeking to exchange information to enhance their research efforts.

The virus broke into Berkeley Standard Distribution (BSD) UNIX[1] and derivative systems. Once resident in a computer, it attempted to break into other machines on the network. This paper is an analysis of that virus program and of the reaction of the Internet community to the attack.

1.1 Organization

In Section 1 we discuss the categorization of the program which attacked the Internet, the goals of the teams working on isolating the virus and the methods they employed, and summarize what the virus did and did not actually do. In Section 2 we discuss in more detail the strategies it employed, the specific attacks it used, and the effective and ineffective defenses proposed by the community. Once the crisis had passed, the Internet community had time not only to explore the vulnerabilities which had allowed the attack to succeed, but also to consider how future attacks could be prevented. Section 3 presents our views on the lessons learned and problems to be faced in the future. In Section 4 we acknowledge the people on our team and the people at other sites who aided us in the effort to understand the virus.

We present a subroutine by subroutine description of the virus program itself in Appendix A, including a diagram of the information flow through the routines which comprise the ‘‘cracking engine’’. Appendix B contains a list of the words included in the built-in dictionary carried by the virus.

1.2 A Rose by Any Other Name

The question of how to classify the program which infected the Internet has received a fair amount of attention. Was it a ‘‘virus’’ or ‘‘worm’’; or was it something else?

There is confusion about the term ‘‘virus.’’ To a biologist a virus is an agent of infection which can only grow and reproduce within a host cell. A lytic virus enters a cell and uses the cell’s own metabolic machinery to replicate. The newly created viruses (more appropriately called ‘‘virons’’) break out of the infected cell, destroying it, and then seek out new cells to infect. A lysogenetic virus, on the other hand, alters the genetic material of its host cells. When the host cell reproduces it unwittingly reproduces the viral genes. At some point in the future, the viral genes are activated and many virons are produced by the cell. These proceed to break out of the cell and seek out other cells to infect[3]. Some single strand DNA viruses do not kill the host cell; they use the machinery of the host cell to repro-

---

 This paper will be presented at the 1989 IEEE Symposium on Research in Security and Privacy.

[1] UNIX is a trademark of AT&T. DEC, VAX, and Ultrix are trademarks of Digitial Equipment Corporation. Sun, SunOS, and NFS are trademarks of Sun Microsystems, Inc. IBM is a trademark of International Business Machines, Inc.

duce (perhaps slowing normal celluar growth by diverting resources) and exit the cells in a non-destructive manner[4].

A ‘‘worm’’ is an organism with an elongated segmented body. Because of the shape of their bodies worms can snake around obstacles and work their way into unexpected places. Some worms, for example the tapeworm, are parasites. They live inside of a host organism, feeding directly from nutrients intended for host cells. These worms reproduce by shedding one of their segments which contains many eggs. They have difficulty in reaching new hosts, since they usually leave an infected host through its excretory system and may not readily come into contact with another host[5].

In deciding which term fits the program which infected the Internet, we must decide which part of the system is analogous to the ‘‘host’’. Possibilities include the network, host computers, programs, and processes. We must also consider the actions of the program and its structure.

Viewing the network layer as the ‘‘host’’ is not fruitful; the network was not attacked, specific hosts on the network were. The infection never spread beyond the Internet even though there were gateways to other types of networks. One could view the infection as a worm, which ‘‘wiggled’’ throughout the network. But as Beckman points out[6] the program didn’t have connected ‘‘segments’’ in any sense. Thus it can’t be a worm.

A model showing the computers as the ‘‘host’’ is more promising. The infection of 2 November entered the hosts, reproduced, and exited in search of new hosts to infect. Some people might argue that since the host was not destroyed in this process, that the infecting program was more like a worm than a virus. But, as mentioned earlier, not all viruses destroy their host cells. Denning [7] defines a computer worm as a program which enters a workstation and disables it. In that sense the infection could be considered a worm, but we reject this definition. The infected computers were affected but not all were ‘‘disabled’’. There is also no analog to the segments of a biological worm.

Denning has described how many personal computer programs have been infected by viral programs[7]. These are frequently analogous to lysogenetic viruses because they modify the actual program code as stored in the computer’s secondary storage. As the infected programs are copied from computer to computer through normal software distribution, the viral code is also copied. At some point the viral code may activate and perform some action such as deleting files or displaying a message. Applying this definition of a virus while viewing programs as ‘‘hosts’’ does not work for the Internet infection, since the virus neither attacked nor modified programs in any way.

If, however, processes are view as ‘‘hosts’’, then the Internet infection can clearly be considered a viral infection. The virus entered hosts through a daemon process, tricking that process into creating a viral process, which would then attempt to reproduce. In only one case, the finger attack, was the daemon process actually changed; but as we noted above only lysogenetic viruses actually change their host’s genetic material.

Denning defines a bacterium as a program which replicates itself and feeds off the host’s computational resources. While this seems to describe the program which infected the Internet, it is an awkward and vague description which doesn’t seem to convey the nature of the infection at all.

Thus we have chosen to call the program which infected the Internet a virus. We feel it is accurate and descriptive.

1.3 Goals and Targets

The program that attacked many Internet hosts was itself attacked by teams of programmers around the country. The goal of these teams was to find out *all* the inner workings of the virus. This included not just understanding how to stop further attacks, but also understanding whether any permanent damage had been done, including destruction or alteration of data during the actual infection, or possible ‘‘time bombs’’ left for later execution.

There were several steps in achieving these goals: including

- isolating a specimen of the virus in a form which could be analyzed.
- ‘‘decompiling’’ the virus, into a form that could be shown to reduce to the executable of the real thing, so that the higher level version could be interpreted.
- analyzing the strategies used by the virus, and the elements of its design, in order to find weaknesses and methods of defeating it.

The first two steps were completed by the morning of 4 November 1988. Enough of the third was complete to determine that the virus was harmless, but there were no clues to the higher level issues, such as the reason for the virus’ rapid spread.

Once the decompiled code existed, and the threat of the virus known to be minimal, it was clear to the MIT team and those at Berkeley that the code should be protected. We understood that the knowledge required to write such a program could not be kept secret, but felt that if the code were publicly available, someone could too easily modify it and release a damaging mutated strain. If this occurred before many hosts had removed the bugs which allowed the penetration in the first place, much damage would be done.

There was also a clear need to explain to the community what the virus was and how it worked. This information, in the form of this report, can actually be *more* useful to interested people than the source code could be, since it includes discussion of the side effects and results of the code, as well as flaws in it, rather than merely listing the code line by line.

Conversely, there are people interested in the intricate detail of how and why certain routines were used; there should be enough detail here to satisfy them as well. Readers will also find Seely[8] and Spafford's[9] papers interesting.

1.4 Major Points

This section provides an outline of the how the virus attacked and who it attacked. It also lists several things the virus did not do, but which many people seem to have attributed to the virus. All of the following points are described in more detail in Section 2.

1.4.1 How it entered

- sendmail (needed debug mode, as in SunOS binary releases)
- finger[10] (only VAX hosts were victims)
- remote execution system, using
  - rexec
  - rsh

1.4.2 Who it attacked

- accounts with obvious passwords, such as
  - none at all
  - the user name
  - the user name appended to itself
  - the ''nickname''
  - the last name
  - the last name spelled backwards
- accounts with passwords in a 432 word dictionary (see Appendix B)
- accounts with passwords in `/usr/dict/words`
- accounts which trusted other machines via the `.rhosts` mechanism

1.4.3 What it attacked

- SUNs and VAXes only
- machines in `/etc/hosts.equiv`
- machines in `/.rhosts`
- machines in cracked accounts' `.forward` files
- machines in cracked accounts' `.rhosts` files
- machines listed as network gateways in routing tables
- machines at the far end of point-to-point interfaces
- possibly machines at randomly guessed addresses on networks of first hop gateways

1.4.4 What it did NOT do

- gain privileged access (it almost never broke in as `root`)
- destroy or attempt to destroy any data
- leave time bombs behind
- differentiate among networks (such as MILNET, ARPANET)
- use UUCP at all
- attack specific well-known or privileged accounts such as `root`

2 Strategies

2.1 Attacks

This virus attacked several things, directly and indirectly. It picked out some deliberate targets, such as specific network daemons through which to infect the remote host. There were also less direct targets, such as mail service and the flow of information about the virus.

2.1.1 Sendmail Debug Mode

The virus exploited the ''debug'' function of `sendmail`, which enables debugging mode for the duration of the current connection. Debugging mode has many features, including the ability to send a mail message with a program as the recipient (i.e. the program would run, with all of its input coming from the body of the message). This is inappropriate and rumor[11] has it that the author included this feature to allow him to circumvent security on a machine he was using for testing. It certainly exceeds the intended design of the Simple Mail Transfer Protocol (SMTP) [12].

Specification of a program to execute when mail is received is normally allowed in the `sendmail` aliases file or users' `.forward` files directly, for `vacation` [2], mail archive programs, and personal mail sorters. It is *not* normally allowed for incoming connections. In the virus, the ''recipient'' was a command to strip off the mail headers and pass the remainder of the message to a command interpreter. The body was a script that created a C program, the ''grappling hook,'' which transfered the rest of the modules from the originiating host, and the commands to link and execute them. Both VAX and Sun binaries were transfered and both would be tried in turn, no attempt to determine the machine type was made. On other architectures the programs would not run, but would use resources in the linking process. All other attacks used the same ''grappling hook'' mechanism, but used other flaws to inject the ''grappling hook'' into the target machine.

The fact that debug was enabled by default was reported to Berkeley by several sources during the 4.2BSD release. The 4.3BSD release as well as Sun releases still had this option enabled by default [13]. The then current release of

[2] A program which accepts incoming mail and sends back mail to the original sender, usually saying something like ''I am on vacation, and will not read your mail until I return.''

Ultrix did not have debug mode enabled, but the beta test version of the newest release did have debug enabled (it was disabled before finally being shipped). MIT's Project Athena was among a number of sites which went out of its way to disable debug mode; however, it is unlikely that many binary-only sites were able to be as diligent.

2.1.2 Finger Daemon Bug

The virus hit the finger daemon (`fingerd`) by overflowing a buffer which was allocated on the stack. The overflow was possible because `fingerd` used a library function which did not do range checking. Since the buffer was on the stack, the overflow allowed a fake stack frame to be created, which caused a small piece of code to be executed when the procedure returned [3]. The library function in question turns out to be a backward-compatibility routine, which should not have been needed after 1979 [14].

Only 4.3BSD VAX machines were attacked this way. The virus did not attempt a Sun specific attack on finger and its VAX attack failed when invoked on a Sun target. Ultrix was not vulnerable to this since production releases did not include a `fingerd`.

2.1.3 Rexec and Passwords

The virus attacked using the Berkeley remote execution protocol, which required the user name and plaintext password to be passed over the net. The program only used pairs of user names and passwords which it had already tested and found to be correct on the local host. A common, world readable file (`/etc/passwd`) that contains the user names and encrypted passwords for every user on the system facilitated this search. Specifically:

- this file was an easy-to-obtain list of user names to attack,
- the dictionary attack was a method of verifying password guesses which would not be noted in security logs.

The principle of ''least privilege'' [15] is violated by the existence of this password file. Typical programs have no need for a list of user names and password strings, so this privileged information should not be available to them. For example, Project Athena's network authentication system, *Kerberos* [16], keeps passwords on a central server which logs authentication requests, thus hiding the list of valid user names. However, once a name is found, the authentication ''ticket'' is still vulnerable to dictionary attack.

[3] MIT's Project Athena has a ''write'' daemon which has a similar piece of code with the same flaw but it explicitly exits rather than returning, and thus never uses the (damaged) return stack. A comment in the code notes that it is mostly copied from the finger daemon.

2.1.4 Rsh and Trust

The virus attempted to use the Berkeley remote shell program (called `rsh`) to attack other machines without using passwords. The remote shell utility is similar in function to the remote execution system, although it is ''friendlier'' since the remote end of the connection is a command interpreter, instead of the *exec* function. For convenience, a file `/etc/hosts.equiv` can contain a list of hosts trusted by this host. The `.rhosts` file provides similar functionality on a per-user basis. The remote host can pass the user name from a trusted port (one which can only be opened by `root`) and the local host will trust that as proof that the connection is being made for the named user.

This system has an important design flaw, which is that the local host only knows the remote host by its network address, which can often be forged. It also trusts the machine, rather than any property of the user, leaving the account open to attack at all times rather than when the user is present [16]. The virus took advantage of the latter flaw to propagate between accounts on trusted machines. Least privilege would also indicate that the lists of trusted machines be only accessible to the daemons who need to decide to whether or not to grant access.

2.1.5 Information Flow

When it became clear that the virus was propagating via `sendmail`, the first reaction of many sites was to cut off mail service. This turned out to be a *serious* mistake, since it cut off the information needed to fix the problem. Mailer programs on major forwarding nodes, such as *relay.cs.net*, were shut down delaying some critical messages by as long as twenty hours. Since the virus had alternate infection channels (`rexec` and `finger`), this made the isolated machine a safe haven for the virus, as well as cutting off information from machines further ''downstream'' (thus placing them in greater danger) since no information about the virus could reach them by mail[4]. Thus, by attacking `sendmail`, the virus indirectly attacked the flow of information that was the only real defense against its spread.

2.2 Self Protection

The virus used a number of techniques to evade detection. It attempted both to cover it tracks and to blend into the normal UNIX environment using camouflage. These techniques had had varying degrees of effectiveness.

[4] USENET news [17] was an effective side-channel of information spread, although a number of sites disabled that as well.

2.2.1 Covering Tracks

The program did a number of things to cover its trail. It erased its argument list, once it had finished processing the arguments, so that the process status command would not show how it was invoked.

It also deleted the executing binary, which would leave the data intact but unnamed, and only referenced by the execution of the program. If the machine were rebooted while the virus was actually running, the file system salvager would recover the file after the reboot. Otherwise the program would vanish after exiting.

The program also used resource limit functions to prevent a core dump. Thus, it prevented any bugs in the program from leaving tell-tale traces behind.

2.2.2 Camouflage

It was compiled under the name `sh`, the same name used by the Bourne Shell, a command interpreter which is often used in shell scripts and automatic commands. Even a diligent system manager would probably not notice a large number of shells running for short periods of time.

The virus forked, splitting into a parent and child, approximately every three minutes. The parent would then exit, leaving the child to continue from the exact same place. This had the effect of ''refreshing'' the process, since the new fork started off with no resources used, such as CPU time or memory usage. It also kept each run of the virus short, making the virus a more difficult to seize, even when it had been noticed.

All the constant strings used by the program were obscured by XOR'ing each character with the constant 81_{16}. This meant that one could not simply look at the binary to determine what constants the virus refered to (e.g. what files it opened). But it was a weak method of hiding the strings; it delayed efforts to understand the virus, but not for very long.

2.3 Flaws

The virus also had a number of flaws, ranging from the subtle to the clumsy. One of the later messages from Berkeley posted fixes for some of the more obvious ones, as a humorous gesture.

2.3.1 Reinfection prevention

The code for preventing reinfection of an actively infected machine harbored some major flaws. These flaws turned out to be critical to the ultimate ''failure'' of the virus, as reinfection drove up the load of many machines, causing it to be noticed and thus counterattacked.

The code had several timing flaws which made it unlikely to work. While written in a ''paranoid'' manner, using weak authentication (exchanging ''magic'' numbers) to determine whether the other end of the connection is indeed a copy of the virus, these routines would often exit with errors (and thus *not* attempt to quit) if:

- several viruses infected a clean machine at once, in which case all of them would look for listeners; none of them would find any; all of them would attempt to become listeners; one would succeed; the others would fail, give up, and thus be invulnerable to future checking attempts.
- several viruses starting at once, in the presence of a running virus. If the first one ''wins the coin toss'' with the listening virus, other new-starting ones will have contacted the losing one and have the connection closed upon them, permitting them to continue.
- a machine is slow or heavily loaded, which could cause the virus to exceed the timeouts imposed on the exchange of numbers, especially if the compiler was running (possibly multiple times) due to a new infection; note that this is exacerbated by a busy machine (which slows down further) on a moderately sized network.

Note that ''at once'' means ''within a 5-20 second window'' in most cases, and is sometimes looser.

A critical weakness in the interlocking code is that even when it *does* decide to quit, all it does is set the variable `pleasequit`. This variable does not have an effect until the virus has gone through

- collecting the entire list of host names to attack
- collecting the entire list of user names to attack
- trying to attack all of the ''obvious'' permutation passwords (see Section A.4.3)
- trying ten words selected at random from the internal dictionary (see Appendix B) against all of the user names

Since the virus was careful to clean up temporary files, its presence alone didn't interfere with reinfection.

Also, a multiply infected machine would spread the virus faster, perhaps proportionally to the number of infections it was harboring, since

- the program scrambles the lists of hosts and users it attacks; since the random number generator is seeded with the current time, the separate instances are likely to hit separate targets.
- the program tries to spend a large amount of time sleeping and listening for other infection attempts (which never report themselves) so that the processes would share the resources of the machine fairly well.

Thus, the virus spread much more quickly than the perpetrator expected, and was noticed for that very reason. The MIT Media Lab, for example, cut themselves completely off from the network because the computer resources absorbed by the virus were detracting from work in progress, while the lack of network service was a minor problem.

2.3.2 Heuristics

One attempt to make the program not waste time on non-UNIX systems was to sometimes try to open a telnet or rsh connection to a host before trying to attack it and skipping that host if it refused the connection. If the host refused telnet or rsh connections, it was likely to refuse other attacks as well. There were several problems with this heuristic:

- A number of machines exist which provide mail service (for example) but that do not provide telnet or rsh service, and although vulnerable, would be ignored under this attack. The MIT Project Athena mailhub, *athena.mit.edu*, is but one example.
- The telnet ''probing'' code immediately closed the connection upon finding that it had opened it. By the time the ''inet daemon'', the ''switching station'' which handles most incoming network services, identified the connection and started a telnet daemon, the connection was already closed, causing the telnet daemon to indicate an error condition of high enough priority to be logged on most systems. Thus the times of the earliest attacks were noted, if not the machines they came from.

2.3.3 Vulnerabilities not used

The virus did not exploit a number of obvious opportunities.

- When looking for lists of hosts to attack, it could have done ''zone transfers'' from the Internet domain name servers to find names of valid hosts [18]. Many of these records also include host type, so the search could have limited itself to the appropriate processor and operating system types.
- It did not attack both machine types consistently. If the VAX finger attack failed, it could have tried a Sun attack, but that hadn't been implemented.
- It did not try to find privileged users on the local host (such as `root`).

2.4 Defenses

There were many attempts to stop the virus. They varied in inconvenience to the end users of the vulnerable systems, in the amount of skill required to implement them, and in their effectiveness.

- Full isolation from network was frequently inconvenient, but was very effective in stopping the virus, and was simple to implement.
- Turning off mail service was inconvenient both to local users and to ''downstream'' sites, was ineffective at stopping the virus, but was simple to implement.
- Patching out the `debug` command in `sendmail` was only effective in conjunction with other fixes, did not interfere with normal users, and simple instructions for implementing the change were available.
- Shutting down the finger daemon was also effective only if the other holes were plugged as well, was annoying to users if not actually inconvenient, and was simple to perform.
- Fixing the finger daemon required source code, but was as effective as shutting it down, without annoying the users at all.
- `mkdir /usr/tmp/sh` was convenient, simple, and effective in preventing the virus from propagating [5] (See Section A.8.2.)
- Defining `pleasequit` in the system libraries was convenient, simple, and did almost nothing to stop the virus (See Section A.3.2.)
- Renaming the UNIX C compiler and linker (`cc` and `ld`) was drastic, and somewhat inconvenient to users (though much less so than cutting off the network, since different names were available) but effective in stopping the virus.
- Requiring new passwords for all users (or at least all users who had passwords which the virus could guess) was difficult, but it only inconvenienced those users with weak passwords to begin with, and was effective in conjunction with the other fixes (See Section A.4.3 and Appendix B.)

After the virus was analyzed, a tool which duplicated the password attack (including the virus' internal dictionary) was posted to the network. This tool allowed system administrators to analyze the passwords in use on their system. The spread of this virus should be effective in raising the awareness of users (and administrators) to the importance of choosing ''difficult'' passwords. Lawrence Livermore National Laboratories went as far as requiring all passwords be changed, and modifying the password changing program to test new passwords against the lists that include the passwords attacked by the virus [6].

3 Lessons and Open Issues

The virus incident taught many important lessons. It also brought up many more difficult issues which need to be addressed in the future :

- Least Privilege. This basic security principle is frequently ignored and this can result in disaster.
- ''We have met the enemy and he is us.'' The alleged author of the virus has made contributions to the com-

[5] However, both sets of binaries were still compiled, consuming processor time on an attacked machine.

puter security field and was by any definition an insider; the attack did not come from an outside source who obtained sensitive information, and restricting information such as source code would not have helped prevent this incident.

- Diversity is good. Though the virus picked on the most widespread operating system used on the Internet and on the two most popular machine types, most of the machines on the network were never in danger. A wider variety of implementations is probably good, not bad. There is a direct analogy with biological genetic diversity to be made.

- "The cure shouldn't be worse than the disease." Chuck Cole made this point and Cliff Stoll also argued that it may be more expensive to prevent such attacks than it is to clean up after them. Backups are good. It may be cheaper to restore from backups than to try to figure out what damage an attacker has done[6].

- Defenses *must* be at the host level, not the network level. Mike Muuss and Cliff Stoll have made this point quite eloquently[6]. The network performed its function perfectly and should not be faulted; the tragic flaws were in several application programs. Attempts to fix the network are misguided. An analogy with the highway system can be made: anybody can drive up to your house and probably break into your home, but that does not mean we should close down the roads or put armed guards on the exit ramps.

- Logging information is important. The `inetd` and `telnetd` interaction logging the source of virus attacks turned out to be a lucky break, but even so many sites did not have enough logging information available to identify the source or times of infection. This greatly hindered the responses, since people frequently had to install new programs which logged more information. On the other hand, logging information tends to accumulate quickly and is rarely referenced. Thus it is frequently automatically purged. If we log helpful information, but find it is quickly purged, we have not improved the situtation much at all. Mike Muuss points out that frequently one can retrieve such information from backups[6], but this is not always true.

- Denial of service attacks are easy. The Internet is amazingly vulnerable to such attacks. These attacks are quite difficult to prevent, but we could be much better prepared to identify their sources than we are today. For example, currently it is not hard to imagine writing a program or set of programs which crash two-thirds of the existing Sun Workstations or other machines implementing Sun's Network Filesystem (NFS). This is serious since such machines are the most common computers connected to the Internet. Also, the total lack of authentication and authorization for network level routing makes it possible for an ordinary user to disrupt communications for a large portion of the Internet. Both tasks could be easily done in a manner which makes tracking down the initiator extremely difficult, if not impossible.

- A central security fix repository may be a good idea. Vendors *must* participate. End users, who likely only want to get their work done, must be educated about the importance of installing security fixes.

- Knee-jerk reactions should be avoided. Openness and free flow of information is the whole point of networking, and funding agencies should not be encouraged to do anything damaging to this without very careful consideration. Network connectivity proved its worth as an aid to collaboration by playing an invaluable role in the defense and analysis efforts during the crisis, despite the sites which isolated themselves.

4 Acknowledgments

Many people contributed to our effort to take apart the virus. We would like to thank them all for their help and insights both during the immediate crisis and afterwards.

4.1 The MIT team

The MIT group effort encompassed many organizations within the Institute. It included people from Project Athena, the Telecommunications Network Group, the Student Information Processing Board (SIPB), the Laboratory for Computer Science, and the Media Laboratory.

The SIPB's role is quite interesting. It is a volunteer student organization that represents students on issues of the MIT computing environment, does software development, provides consulting to the community, and other miscellaneous tasks. Almost all the members of the MIT team which took apart the virus were members of the SIPB, and the SIPB office was the focus for early efforts at virus catching until people gathered in the Project Athena offices.

Mark W. Eichin (Athena and SIPB) and Stanley R. Zanarotti (LCS and SIPB) led the team disassembling the virus code. The team included Bill Sommerfeld (Athena/Apollo Computer and SIPB), Ted Y. Ts'o (Athena and SIPB), Jon Rochlis (Telecommunications Network Group and SIPB), Ken Raeburn (Athena and SIPB), Hal Birkeland (Media Laboratory), and John T. Kohl (Athena/DEC and SIPB).

Jeffrey I. Schiller (Campus Network Manager, Athena Operations Manager, and SIPB) did a lot of work in trapping the virus, setting up an isolated test suite, and dealing

with the media. Pascal Chesnais (Media Laboratory) was one of the first at MIT to spot the virus. Ron Hoffmann (Network Group) was one of the first to notice an MIT machine attacked by finger.

Tim Shepard (LCS) provided information about the propagation of the virus, as well as large amounts of ‘‘netwatch’’ data and other technical help.

James D. Bruce (EECS Professor and Vice President for Information Systems) and the MIT News Office did an admirable job of keeping the media manageable and letting us get our work done.

4.2 The Berkeley Team

We communicated and exchanged code with Berkeley extensively throughout the morning of 4 November 1988. The team there included Keith Bostic (Computer Systems Research Group, University of California, Berkeley), Mike Karels (Computer Systems Research Group, University of California, Berkeley), Phil Lapsley (Experimental Computing Facility, University of California, Berkeley), Dave Pare (FX Development, Inc.), Donn Seeley (University of Utah), Chris Torek (University of Maryland), and Peter Yee (Experimental Computing Facility, University of California, Berkeley).

4.3 Others

Numerous others across the country deserve thanks; many of them worked directly or indirectly on the virus, and helped coordinate the spread of information. Special thanks should go to Gene Spafford (Purdue) for serving as a central information point and providing key insight into the workings of the virus. Don Becker (Harris Corporation) has provided the most readable decompilation of the virus which we have seen to date. It was most helpful.

People who offered particularly valuable advice included Judith Provost, Jennifer Steiner, Mary Vogt, Stan Zanarotti, Jon Kamens, Marc Horowitz, Jenifer Tidwell, James Bruce, Jerry Saltzer, Steve Dyer, Ron Hoffmann and many unnamed people from the SIPB Office. Any remaining flaws in this paper are our fault, not theirs.

Special thanks to Bill Sommerfeld for providing the description of the finger attack .

A The Program

This Appendix describes the virus program subroutine by subroutine. For reference, the flow of information among the subroutines is shown in Figure 1.

A.1 Names

The core of the virus is a pair of binary modules, one for the VAX architecture and the other for the Sun architecture. These are linkable modules, and thus have name lists for their internal procedures. Many of the original names are included here with the descriptions of the functions the routines performed.

It is surprising that the names are included, and astonishing that they are meaningful. Some simple techniques, such as randomizing the procedure names, would have removed a number of clues to the function of the virus.

A.2 main

The main module, the starting point of any C language program, does some initialization, processes its command line, and then goes off into the loop which organizes all of the real work.

A.2.1 Initialization

The program first takes some steps to hide itself. It changes the ''zeroth'' argument, which is the process name, to `sh`. Thus, no matter how the program was invoked, it would show up in the process table with the same name as the Bourne Shell, a program which often runs legitimately.

The program also sets the maximum core dump size to zero blocks. If the program crashed[6] it would not leave a core dump behind to help investigators. It also turns off handling of write errors on pipes, which normally cause the program to exit.

The next step is to read the clock, store the current time in a local variable, and use that value to seed the random number generator.

A.2.2 Command line argument processing

The virus program itself takes an optional argument `-p` which must be followed by a decimal number, which seems to be a process id of the parent which spawned it. It uses this number later to kill that process, probably to ''close the door'' behind it.

The rest of the command line arguments are ''object names''. These are names of files it tries to load into its address space. If it can't load one of them, it quits. If the `-p` argument is given, it also deletes the object files, and later tries to remove the disk image of running virus, as well as the file `/tmp/.dumb`. (This file is not referenced anywhere else in the virus, so it is unclear why it is deleted.)

The program then tried a few further steps, exiting (''bailing out'') if any of them failed:

- It checked that it had been given at least one object on the command line.
- It checked to see if it had successfully loaded in the object `l1.c`.

If the ''-p'' argument was given, the program closes all file descriptors, in case there are any connections open to the parent.

The program then erases the text of the argument array, to further obscure how it was started (perhaps to hide anything if one were to get a core image of the running virus.)

It scans all of the network interfaces on the machine, gets the flags and addresses of each interface. It tries to get the point-to-point address of the interface, skipping the loopback address. It also stores the netmask for that network [19].

Finally, it kills off the process id given with the ''-p'' option. It also changes the current process group, so that it doesn't die when the parent exits. Once this is cleaned up, it falls into the *doit* routine which performs the rest of the work.

A.3 doit routine

This routine is where the program spends most of its time.

A.3.1 Initialization

Like the main routine, it seeds the random number generator with the clock, and stores the clock value to later measure how long the virus has been running on this system.

It then tries *hg*. If that fails, it tries *hl*. If that fails, it tries *ha*.

It then tries to check if there is already a copy of the virus running on this machine. Errors in this code contributed to the large amounts of computer time taken up by the virus. Specifically:

- On a one-in-seven chance, it won't even try to test for another virus.
- The first copy of the virus to run is the only one which listens for others; if multiple infections occur ''simultaneously'' they will not ''hear'' each other, and all but one will fail to listen (see section A.12).

The remainder of the initialization routine seems designed to send a single byte to address 128.32.137.13, which is *ernie.berkeley.edu*, on port 11357. This never happens, since the author used the *sendto* function on a TCP

[6] For example, the virus was originally compiled using 4.3BSD declaration files. Under 4.2BSD, the alias name list did not exist, and code such as the virus which assumes aliases are there can crash and dump core.

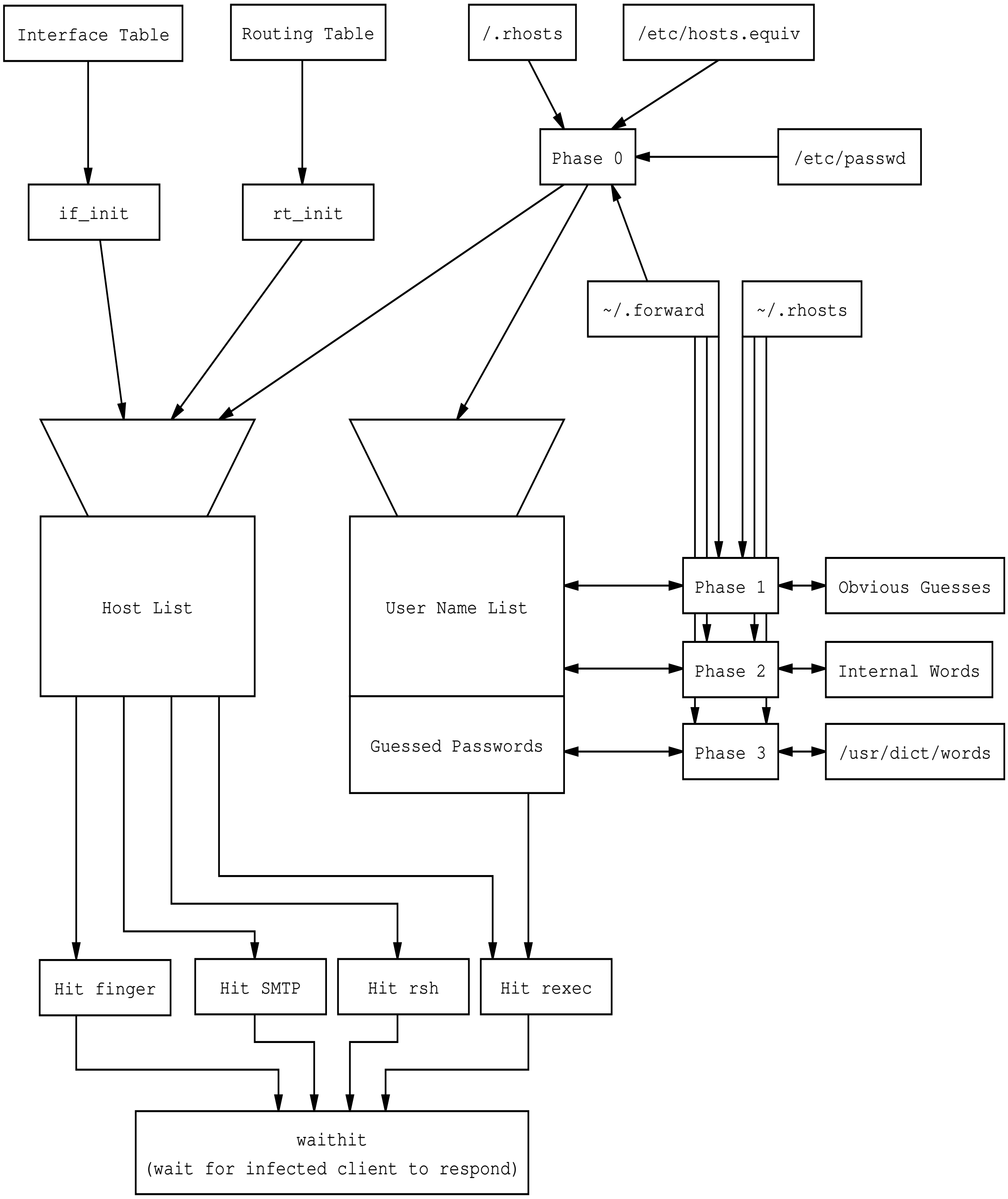

Figure 1: The structure of the attacking engine.

stream connection, instead of a UDP datagram socket.[7] We have no explanation for this; it only tries to send this packet with a one in fifteen random chance.

A.3.2 Main loop

An infinite loop comprises the main active component of the virus. It calls the *cracksome* routine[8] which tries to find some hosts that it can break in to. Then it waits 30 seconds, listening for other virus programs attempting to break in, and tries to break into another batch of machines.

After this round of attacks, it forks, creating two copies of the virus; the original (parent) dies, leaving the fresh copy. The child copy has all of the information the parent had, while not having the accumulated CPU usage of the parent. It also has a new process id, making it hard to find.

Next, the *hg*, *hl*, and *ha* routines search for machines to infect (see Appendix A.5). The program sleeps for 2 minutes, and then checks to see if it has been running for more than 12 hours, cleaning up some of the entries in the host list if it has.

Finally, before repeating, it checks the global variable `pleasequit`. If it is set, *and* if it has tried more than 10 words from its own dictionary against existing passwords, it quits. Thus forcing `pleasequit` to be set in the system libraries will do very little to stem the progress of this virus[9].

A.4 Cracking routines

This collection of routines is the ''brain'' of the virus. *cracksome*, the main switch, chooses which of four strategies to execute. It is would be the central point for adding new strategies if the virus were to be further extended. The virus works each strategy through completely, then switches to the next one. Each pass through the cracking routines only performs a small amount of work, but enough state is remembered in each pass to continue the next time around.

A.4.1 cracksome

The *cracksome* routine is the central switching routine of the cracking code. It decides which of the cracking strategies is actually exercised next. Again, note that this routine was named in the global symbol table. It could have been given a confusing or random name, but it was actually clearly labelled, which lends some credence to the idea that the virus was released prematurely.

[7] If the author had been as careful with error checking here as he tried to be elsewhere, he would have noted the error ''socket not connected'' every time this routine is invoked.

[8] This name was actually in the symbol table of the distributed binary!

[9] Although it was suggested very early [20].

A.4.2 Phase 0

The first phase of the *cracksome* routines reads through the `/etc/hosts.equiv` file to find machine names that would be likely targets. While this file indicates what hosts the current machine trusts, it is fairly common to find systems where all machines in a cluster trust each other, and at the very least it is likely that people with accounts on this machine will have accounts on the other machines mentioned in `/etc/hosts.equiv`.

It also reads the `/.rhosts` file, which lists the set of machines that this machine trusts root access from. Note that it does not take advantage of the trust itself [21] but merely uses the names as a list of additional machines to attack. Often, system managers will deny read access to this file to any user other than root itself, to avoid providing any easy list of secondary targets that could be used to subvert the machine; this practice would have prevented the virus from discovering those names, although `/.rhosts` is very often a subset of `/etc/hosts.equiv`.

The program then reads the entire local password file, `/etc/passwd`. It uses this to find personal `.forward` files, and reads them in search of names of other machines it can attack. It also records the user name, encrypted password, and *GECOS* information string, all of which are stored in the `/etc/passwd` file. Once the program scanned the entire file, it advanced to Phase 1.

A.4.3 Phase 1

This phase of the cracking code attacked passwords on the local machine. It chose several likely passwords for each user, which were then encrypted and compared against the encryptions obtained in Phase 0 from `/etc/passwd`:

- No password at all.
- The user name itself.
- The user name appended to itself.
- The second of the comma separated *GECOS* information fields, which is commonly a nickname.
- The remainder of the full name after the first name in the *GECOS* fields, i.e. probably the last name, with the first letter converted to lower case.
- This ''last name'' reversed.

All of these attacks are applied to fifty passwords at a time from those collected in Phase 0. Once it had tried to guess the passwords for all local accounts, it advanced to Phase 2.

A.4.4 Phase 2

Phase 2 takes the internal word list distributed as part of the virus (see Appendix B) and shuffles it. Then it takes the words one at a time and decodes them (the high bit is set on all of the characters to obscure them) and tries them against all collected passwords. It maintains a global variable `nextw` as an index into this table. The main loop uses this to prevent `pleasequit` from causing the virus to exit until at least ten of the words have been checked against all of the encryptions in the collected list.

Again, when the word list is exhausted the virus advances to Phase 3.

A.4.5 Phase 3

Phase 3 looks at the local `/usr/dict/words` file, a 24474 word list distributed with 4.3BSD (and other UNIX systems) as a spelling dictionary. The words are stored in this file one word per line. One word at a time is tried against all encrypted passwords. If the word begins with an upper case letter, the letter is converted to lower case and the word is tried again.

When the dictionary runs out, the phase counter is again advanced to 4 (thus no more password cracking is attempted).

A.5 H routines

The ''h routines'' are a collection of routines with short names, such as *hg*, *ha*, *hi*, and *hl*, which search for other hosts to attack.

A.5.1 hg

The *hg* routine calls *rt_init* (if it has not already been called) to scan the routing table, and records all gateways except the loopback address in a special list. It then tries a generic attack routine to attack via `rsh`, `finger`, and SMTP. It returns after the first successful attack.

A.5.2 ha

The *ha* routine goes through the gateway list and connects to TCP port 23, the telnet port, looking for gateways which are running telnet listeners. It randomizes the order of such gateways and calls *hn* (our name) with the network number of each gateway. The *ha* returns after *hn* reports that it has succeeded broken into a host.

A.5.3 hl

The *hl* routine iterates through all the addresses for the local machine calling *hn* with the network number for each one. It returns if *hn* indicates success in breaking into a host.

A.5.4 hi

The *hi* routine goes through the internal host list (see section A.4.2) and tries to attack each host via `rsh`, `finger`, and SMTP. It returns if when one host is infected.

A.5.5 hn

The *hn* routine (our name) followed *hi* takes a network number as an argument. Surprisingly it returns if the network number supplied is the same as the network number of any of the interfaces on the local machine. For Class A addresses it uses the Arpanet IMP convention to create possible addresses to attack (net.[1-8].0.[1-255]). For all other networks it guesses hosts number one through 255 on that network. It randomizes the order of this list of possible hosts and tries to attack up to twenty of them using `rsh`, `finger`, and SMTP. If a host does not accept connections on TCP port 514, the `rsh` port, *hn* will not try to attack it. If a host is successfully attacked *hn* returns.

A.5.6 Usage

The ''h routines'' are called in groups in the main loop; if the first routine succeedes in finding a vulnerable host the remaining routines are not called in the current pass. Each routine returns after it finds one vulnerable host. The *hg* routine is always called first, which indicates the virus really wanted to infect gateway machines. Next comes *hi* which tried to infect normal hosts found via *cracksome*. If *hi* fails, *ha* is called, which seemed to try breaking into hosts with randomly guessed addresses on the far side of gateways. This assumes that all the addresses for gateways had been obtained (which is not trivial to verify from the convoluted code in *rt init*), and implies that the virus would prefer to infect a gateway and from there reach out to the gateway's connected networks, rather than trying to hop the gateway directly. If *hg*, *hi*, and *ha* all failed to infect a host, then *hl* is called which is similar to *ha* but uses for local interfaces for a source of networks.

It is not clear that *ha* and *hl* worked. Because *hn* returns if the address is local, *hl* appears to have no chance of succeeding. If alternate addresses for gateways are indeed obtained by other parts of the virus then *ha* could work. But if only the addresses in the routing table were used it could not work, since by definition these addresses must be on a directly connected network. Also, in our monitoring we never detected an attack on a randomly generated address. These routines do not seem to have been functional.

A.6 Attack routines

There are a collection of attack routines, all of which try to obtain a Bourne Shell running on the targeted machine. See Appendix A.7 for a description of the `l1.c` program, used by all the attack routines.

A.6.1 hu1

The *hu1* routine is called by the Phase 1 and Phase 3 *cracksome* subroutines. Once a password for user name guessed correctly, this routine is called with a host name read from either the user's `.forward` or `.rhosts` files. In order to assume the user's id it then tries to connect to the local machine's `rexec` server using the guessed name and password. If successful it runs an `rsh` to the target machine, trying to execute a Bourne Shell, which it uses to send over and compile the `l1.c` infection program.

A.6.2 Hit SMTP

This routine make a connection to TCP port 25, the SMTP port, of a remote machine and used it to take advantage of the sendmail bug. It attempts to use the debug option to make `sendmail` run a command (the "recipient" of the message), which transfers the `l1.c` program included in the body of the message.

A.6.3 Hit finger

The "hit finger" routine tries to make a connection to TCP port 79, the finger port, of the remote machine. Then it creates a "magic packet" which consists of

- A 400 byte "runway" of VAX "nop" instructions, which can be executed harmlessly.
- A small piece of code which executes a Bourne Shell.
- A stack frame, with a return address which would hopefully point into the code.

Note that the piece of code is VAX code, and the stack frame is a VAX frame, in the wrong order for the Sun. Thus, although the Sun finger daemon has the same bug as the VAX one, this piece of code cannot exploit it.

The attack on the finger daemon is clearly a lysogenetic "viral" attack (see Section 1.2), since although a worm doesn't modify the host machine at all, the finger attack does modify the running finger daemon process. The "injected DNA" component of the virus contained the VAX instructions shown in Figure 2.

The *execve* system call causes the current process to be replaced with an invocation of the named program; `/bin/sh` is the Bourne Shell, a UNIX command interpreter. In this case, the shell winds up running with its input coming from, and its output going to, the network connection. The virus then sends over the `l1.c` bootstrap program.

A.6.4 Hit rsh

This unlabeled routine tries `rsh` to the target host (assuming it can get in as the current user). It tries three different names for the `rsh` binary,

- `/usr/ucb/rsh`
- `/usr/bin/rsh`
- `/bin/rsh`

If one of them succeeds, it tries to resynchronize (see Appendix A.8.1) the connection; if that doesn't succeed within thirty seconds it kills off the child process. If successful the connection can then be used to launch the `l1.c` "grappling hook" program at the victim.

Note that this infection method doesn't specify a user name to attack; if it gets into the remote account, it is because the user that the virus is running as also has an account on the other machine which trusts the originating machine.

A.6.5 Hit rexec

The *hit rexec* routine uses the remote execution system which is similar to `rsh`, but designed for use by programs. It connects and sends the user name, the password, and `/bin/sh` as the command to execute.

A.6.6 makemagic

This routine tries to make a telnet connection to each of the available addresses for the current victim. It broke the connections immediately, often producing error reports from the telnet daemon, which were recorded, and provide some of the earliest reports of attack attempts.[10]

If it succeedes in reaching the host, it creates a TCP listener on a random port number which the infected machine would eventually connect back to.

A.7 Grappling Hook

A short program, named `l1.c`, is the common grappling hook that all of the attack routines use to pull over the rest of the virus. It is robustly written, and fairly portable. It ran on a number of machines which were neither VAX or Sun, loading them down as well, but only making them peripheral victims of the virus.

The first thing it does is delete the binary it was running from. It checks that it has three arguments (exiting if there aren't three of them). It closes all file descriptors and then forks, exiting if the fork fails. If it succeeds, the parent exits; this leaves no connection from the child to the infection route.

Next, it creates a TCP connection back to the address given as the first argument, and the port given as the second. Then it sends over the magic number given as the third. The text of each argument is erased immediately after it is used. The stream connection is then reused as the program's standard input and output.

[10] On fast machines, such as the DEC VAX 3200, there may be no record of these attacks, since the connection is handed off fast enough to satisfy the daemon.

```
pushl   $68732f      push '/sh<NUL>'
pushl   $6e69622f    push '/bin'
movl    sp,r10       save address of start of string
pushl   $0           push 0 (arg 3 to execve)
pushl   $0           push 0 (arg 2 to execve)
pushl   r10          push string addr (arg 1 to execve)
pushl   $3           push argument count
movl    sp,ap        set argument pointer
chmk    $3b          do "execve" kernel call.
```

Figure 2: VAX intructions for the `finger` attack.

A loop reads in a length (as a network byte order 32-bit integer) and then a filename. The file is unlinked and opened for write, and then the file itself is read in (using the number of bytes read in earlier.) On any error, all of the files are unlinked. If the length read in is -1, the loop exits, and a Bourne Shell is executed (replacing the `l1` program, and getting its input from the same source.)

A.8 Install Routines

There are a variety of routines used to actually move the virus from one machine to the other. They deal with the "virus protocol" connection made by the `l1.c` injected program or with the shell that it spawns.

A.8.1 resynch

The *resynch* routine sends commands to a remote shell, requesting that it echo back a specific randomly chosen number. It then waits a certain amount of time for a response. This routine is used to indicate when the various subprograms of the infection procedure have compiled or executed and a Bourne Shell prompt is available again.

A.8.2 waithit

This routine does much of the high level work. It waits (up to 2 minutes) for a return connection from a victim (which has had `l1.c` injected into it.) It then tries to read a magic number (which had been previously sent to that victim as a command line argument to the `l1` program) and gives up after ten seconds.

After the connection is established, all of the current "objects" in storage in the virus are fed down the connection into the victim. Then it tries to resynchronize, and if it succeeds, sends down commands to

- set the `PATH` of the victim shell
- try to delete `sh` in the current directory (`/usr/tmp`)
- if the delete fails, pick a random name to use instead [11]
- scan the list of objects, looking for names ending in `.o`
- link and run each of these, with the command line arguments
  - `-p $$`, where `$$` is the process id of the victim shell
  - each object name
- resynchronize; if this fails, assume that the virus succeeded (since the `-p` option tells the virus to kill off the parent shell) and set flag bit 1 of the host list entry (the host list is detailed in section A.9).
- delete the compiled program, and go on to the next object.

Thus, to add another machine type, the virus merely needs to be started with a new object binary as a command line option, which will then be propagated to the next infected host and tried.

Note that the path used here was `PATH= bin: /usr/bin:    /usr/ucb` which is certainly reasonable on most systems. This protects systems with "unusual" filesystem layouts, and suggests that complete consistency among systems makes them more vulnerable.

A.9 Host modules

These are a set of routines designed to collect names and addresses of target hosts in a master list. Each entry contains up to six addresses, up to twelve names, and a flags field.

A.9.1 Name to host

This routine searches the host list for a given named host, returns the list entry describing it, and optionally adds it to the list if it isn't there already.

[11] Since the delete command used (`rm -f`) did not remove directories, creating a directory `/usr/tmp/sh` stoped the virus[22]. However, the virus would still use CPU resources attempting to link the objects, even though it couldn't write to the output file (since it was a directory).

A.9.2 Address to host

This routine searches the host list for a given host address, returns the list entry describing it, and optionally adds it to the list if it isn't there already.

A.9.3 Add address/name

These two routines added an address or a name to a host list entry, checking to make sure that the address or name was not already known.

A.9.4 Clean up table

This routine cycles through the host list, and removes any hosts which only have flag bits 1 and 2 set (and clears those bits.) Bit 1 is set when a resynchronize (in *waithit*) fails, probably indicating that this host ''got lost''. Bit 2 is set when a named host has no addresses, or when several different attack attempts fail. Bit 3 is set when Phase 0 of the crack routines successfully retrieves an address for the host.

A.9.5 Get addresses

This routine takes an entry in the host table and tries to fill in the the gaps. It looks up an address for a name it has, or looks up a name for the addresses it has. It also includes any aliases it can find.

A.10 Object routines

These routines are what the system uses to pull all of its pieces into memory when it starts (after the host has been infected) and then to retrieve them to transmit to any host it infects.

A.10.1 Load object

This routine opens a file, determines its length, allocating the appropriate amount of memory, reads it in as one block, decodes the block of memory (with XOR). If the object name contains a comma, it moves past it and starts the name there.

A.10.2 Get object by name

This routine returns a pointer to the requested object. This is used to find the pieces to download when infecting another host.

A.11 Other initialization routines

A.11.1 if init

This routine scans the array of network interfaces. It gets the flags for each interface, and makes sure the interface is UP and RUNNING (specific fields of the flag structure). If the entry is a point to point type interface, the remote address is saved and added to the host table. It then tries to enter the router into the list of hosts to attack.

A.11.2 rt init

This routine runs `netstat -r -n` as a subprocess. This shows the routing table, with the addresses listed numerically. It gives up after finding 500 gateways. It skips the default route, as well as the loopback entry. It checks for redundant entries, and checks to see if this address is already an interface address. If not, it adds it to the list of gateways.

After the gateway list is collected, it shuffles it and enters the addresses in the host table.

A.12 Interlock routines

The two routines *checkother* and *othersleep* are at the heart of the excessive propagation of the virus. It is clear that the author intended for the virus to detect that a machine was already infected, and if so to skip it. The code is actually fraught with timing flaws and design errors which lead it to permit multiple infections, probably more often than the designer intended[12].

An active infection uses the *othersleep* routine for two purposes, first to sleep so that it doesn't use much processor time, and second to listen for requests from ''incoming'' viruses. The virus which is running *othersleep* is referred to as the ''listener'' and the virus which is running *checkother* is referred to as the ''tester''.

A.12.1 Checkother

The tester tries to connect to port 23357 on the local machine (using the loopback address, 127.0.0.1) to see if it can connect to a listener. If any errors occur during this check, the virus assumes that no listener is present, and tries to become a listener itself.

If the connection is successful, the checker sends a magic number[13], and listens (for up to 300 seconds) for a magic number from the listener[14]. If the magic number is wrong, the checker assumes it is being spoofed and continues to run.

The checker then picks a random number, shifts it right by three (throwing away the lower three bits) and sends it to the listener. It expects a number back within ten seconds, which it adds to the one sent. If this sum is even, the

[12] This behavior was noted by both looking at the code and by creating a testbed setup, manually running a program that performs the checking and listening functions.

[13] $874697_{16}, 8865431_{10}, 041643227_{8}$

[14] $148898_{16}, 1345688_{10}, 05104230_{8}$

sender increments `pleasequit`, which (as noted in section A.3.2) does very little.

Once it has finished communicating (or failing to communicate) with the listener, the checker sleeps for five seconds and tries to become a listener. It creates a TCP stream socket, sets the socket options to indicate that it should allow multiple binds to that address (in case the listener *still* hasn't exited, perhaps?) and then binds the socket to port 23357, and listens on it (permitting a backlog of up to ten pending connections.)

A.12.2 Othersleep

The *othersleep* routine is run when the main body of the virus wants to idle for a period of time. This was apparently intended to help the virus "hide" so that it wouldn't use enough processor time to be noticed. While the main program sleeps, the listener code waits to see if any checkers have appeared and queried for the existence of a listener, as a simple "background task" of the main virus.

The routine first checks to see if it has been set up as a listener; if not, it calls the normal *sleep* function to sleep for the requested number of seconds, and returns.

If it is set up as a listener, it listens on the checking port with a timeout. If it times out, it returns, otherwise it deals with the connection and subtracts the elapsed real time from the time out value.

The body of the listener "accepts" the connection, and sends a magic number to the checker. It then listens (for up to 10 seconds) for the checker's magic number, and picks a random number. It shifts the random number right by three, discarding the lower bits, and sends it up to the checker; it then listens (for up to 10 seconds) for a random number from the checker. If any of these steps fail, the connection is closed and the checker is ignored.

Once the exchanges have occurred, the address of the incoming connection is compared with the loopback address. If it is not from the loopback address, the attempt is ignored. If it is, then if the sum of the exchanged random numbers is odd, the listener increments `pleasequit` (with little effect, as noted in section A.3.2) and closes the listener connection.

B Built in dictionary

432 words were included:

aaa	academia	aerobics
airplane	albany	albatross
albert	alex	alexander
algebra	aliases	alphabet
ama	amorphous	analog
anchor	andromache	animals
answer	anthropogenic	anvils
anything	aria	ariadne
arrow	arthur	athena
atmosphere	aztecs	azure
bacchus	bailey	banana
bananas	bandit	banks
barber	baritone	bass
bassoon	batman	beater
beauty	beethoven	beloved
benz	beowulf	berkeley
berliner	beryl	beverly
bicameral	bob	brenda
brian	bridget	broadway
bumbling	burgess	campanile
cantor	cardinal	carmen
carolina	caroline	cascades
castle	cat	cayuga
celtics	cerulean	change
charles	charming	charon
chester	cigar	classic
clusters	coffee	coke
collins	commrades	computer
condo	cookie	cooper
cornelius	couscous	creation
creosote	cretin	daemon
dancer	daniel	danny
dave	december	defoe
deluge	desperate	develop
dieter	digital	discovery
disney	dog	drought
duncan	eager	easier
edges	edinburgh	edwin
edwina	egghead	eiderdown
eileen	einstein	elephant
elizabeth	ellen	emerald
engine	engineer	enterprise
enzyme	ersatz	establish
estate	euclid	evelyn
extension	fairway	felicia
fender	fermat	fidelity
finite	fishers	flakes
float	flower	flowers
foolproof	football	foresight
format	forsythe	fourier
fred	friend	frighten
fun	fungible	gabriel
gardner	garfield	gauss
george	gertrude	ginger
glacier	gnu	golfer
gorgeous	gorges	gosling
gouge	graham	gryphon
guest	guitar	gumption
guntis	hacker	hamlet
handily	happening	harmony

harold	harvey	hebrides
heinlein	hello	help
herbert	hiawatha	hibernia
honey	horse	horus
hutchins	imbroglio	imperial
include	ingres	inna
innocuous	irishman	isis
japan	jessica	jester
jixian	johnny	joseph
joshua	judith	juggle
julia	kathleen	kermit
kernel	kirkland	knight
ladle	lambda	lamination
larkin	larry	lazarus
lebesgue	lee	leland
leroy	lewis	light
lisa	louis	lynne
macintosh	mack	maggot
magic	malcolm	mark
markus	marty	marvin
master	maurice	mellon
merlin	mets	michael
michelle	mike	minimum
minsky	moguls	moose
morley	mozart	nancy
napoleon	nepenthe	ness
network	newton	next
noxious	nutrition	nyquist
oceanography	ocelot	olivetti
olivia	oracle	orca
orwell	osiris	outlaw
oxford	pacific	painless
pakistan	pam	papers
password	patricia	penguin
peoria	percolate	persimmon
persona	pete	peter
philip	phoenix	pierre
pizza	plover	plymouth
polynomial	pondering	pork
poster	praise	precious
prelude	prince	princeton
protect	protozoa	pumpkin
puneet	puppet	rabbit
rachmaninoff	rainbow	raindrop
raleigh	random	rascal
really	rebecca	remote
rick	ripple	robotics
rochester	rolex	romano
ronald	rosebud	rosemary
roses	ruben	rules
ruth	sal	saxon
scamper	scheme	scott
scotty	secret	sensor

serenity	sharks	sharon
sheffield	sheldon	shiva
shivers	shuttle	signature
simon	simple	singer
single	smile	smiles
smooch	smother	snatch
snoopy	soap	socrates
sossina	sparrows	spit
spring	springer	squires
strangle	stratford	stuttgart
subway	success	summer
super	superstage	support
supported	surfer	suzanne
swearer	symmetry	tangerine
tape	target	tarragon
taylor	telephone	temptation
thailand	tiger	toggle
tomato	topography	tortoise
toyota	trails	trivial
trombone	tubas	tuttle
umesh	unhappy	unicorn
unknown	urchin	utility
vasant	vertigo	vicky
village	virginia	warren
water	weenie	whatnot
whiting	whitney	will
william	williamsburg	willie
winston	wisconsin	wizard
wombat	woodwind	wormwood
yacov	yang	yellowstone
yosemite	zap	zimmerman

References

[1] R. Hinden, J. Haverty, and A. Sheltzer, ‘‘The DARPA Internet: Interconnecting Heterogeneous Computer Networks with Gateways,’’ *IEEE Computer Magazine*, vol. 16, num. 9, pp. 38–48, September 1983.

[2] J. S. Quarterman and J. C. Hoskins, ‘‘Notable Computer Networks,’’ in *Communications of the ACM*, vol. 29, num. 10, pp. 932–971, October 1986.

[3] S. E. Luria, S. J. Gould, and S. Singer, *A View of Life*. Menlo Park, California: Benjamin/Cummings Publishing Company, Inc., 1981.

[4] J. Watson *et al.*, *Molecular Biology of the Gene*. Menlo Park, California: Benjamin/Cummings Publishing Company, Inc., 1987.

[5] G. G. Simpson and W. S. Beck, *Life: An Introduction to Biology*. New York, New York: Harcourt, Brace and Ward, Inc., 1965.

[6] L. Castro *et al.*, ''Post Mortem of 3 November ARPANET/MILNET Attack.'' National Computer Security Center, Ft. Meade MD, 8 November 1988.

[7] P. J. Denning, ''Computer Viruses,'' *American Scientist*, vol. 766, pp. 236–238, May-June 1988.

[8] D. Seeley, ''A Tour of the Worm,'' in *USENIX Association Winter Conference 1989 Proceedings*, pp. 287–304, January 1989.

[9] E. H. Spafford, ''The Internet Worm Program: An Analysis,'' *ACM SIGCOM*, vol. 19, January 1989.

[10] K. Harrenstien, ''NAME/FINGER Protocol Protocol,'' Request For Comments NIC/RFC 742, Network Working Group, USC ISI, Novemeber 1977.

[11] J. Markoff, ''Computer Snarl: A 'Back Door' Ajar,'' *New York Times*, p. B10, 7 November 1988.

[12] J. B. Postel, ''Simple Mail Transfer Protocol,'' Request For Comments NIC/RFC 821, Network Working Group, USC ISI, August 1982.

[13] S. Bellovin, ''The worm and the debug option,'' in *Forum on Risks to the Public in Computers and Related Systems*, vol. 7, num. 74, ACM Committee on Computers and Public Policy, 10 November 1988.

[14] J. Collyer, ''Risks of unchecked input in C programs,'' in *Forum on Risks to the Public in Computers and Related Systems*, vol. 7, num. 74, ACM Committee on Computers and Public Policy, 10 November 1988.

[15] J. Saltzer and M. Schroeder, ''The Protection of Information in Computer Systems,'' in *Proc. IEEE*, vol. 63, num. 9, pp. 1278–1308, IEEE, September 1975.

[16] J. Steiner, C. Neuman, and J. Schiller, ''Kerberos: An Authentication Service for Open Network Systems,'' in *USENIX Association Winter Conference 1988 Proceedings*, pp. 191–202, February 1988.

[17] M. R. Horton, ''How to Read the Network News,'' *UNIX User's Supplementary Documents*, April 1986.

[18] P. Mockapetris, ''Domain Names - Concepts And Facilities,'' Request For Comments NIC/RFC 1034, Network Working Group, USC ISI, November 1987.

[19] J. Mogul and J. B. Postel, ''Internet Standard Subnetting Procedure,'' Request For Comments NIC/RFC 950, Network Working Group, USC ISI, August 1985.

[20] G. Spafford, ''A cure!!!!!,'' in *Forum on Risks to the Public in Computers and Related Systems*, vol. 7, num. 70, ACM Committee on Computers and Public Policy, 3 November 1988.

[21] R. W. Baldwin, *Rule Based Analysis of Computer Security*. PhD thesis, MIT EE, June 1987.

[22] G. Spafford, ''A worm ''condom'',' in *Forum on Risks to the Public in Computers and Related Systems*, vol. 7, num. 70, ACM Committee on Computers and Public Policy, 3 November 1988.