



# Document Cover Sheet for Technical Memorandum

Title: Viral Attacks On UNIX® System Security

Author (Computer Address)	Location	Ext.	Company (if other than AT&T-BL)
Tom Duff (research!td)	MH 2C-425	6485	

Document No.	Filing Case No.	Project No.
11273-880728-02TMS	39199-11	311403-0101

**Keywords:**

UNIX; security; virus; Trojan Horse; discretionary access controls

**MERCURY Announcement Bulletin Sections**

CMP - Computing

**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. Such a virus program could, like the Trojan Horse, harbor Greeks set to attack the system when run by sufficiently privileged users or from infected set-userid programs.

The author 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 Center 1127's UNIX systems and spread in a few days through the 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.

Viruses of this sort must be tiny to fit in the small amount of space available, and consequently are very timid. There are ways to construct similar viruses that are not space-constrained and can therefore spread more aggressively and harbor better-armed Greeks. As an example, we 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 a number of 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. Most 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 UNIX system permission scheme that may be able to effectively interdict viral attacks without serious effect on the UNIX system's functioning and habitability.

Total Pages (including document cover sheet): 9

Mailing Label

Complete Copy

Executive Director 112  
Directors 112  
Department Heads 1127, 1125  
61 Supervision  
F. T. Grampp  
J. Reeds  
M. D. McIlroy  
M. Brothers  
L. Forman

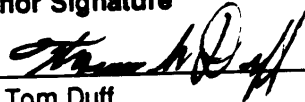
Cover Sheet Only

A. A. Penzias  
1125, 1127 MTS  
R. B. Ardis  
D. G. Belanger  
S. G. Chappell  
J. S. Walden

Future AT&T Distribution by ITDS

DO NOT RELEASE to any AT&T employee without appropriate approval for each request.

Author Signature



Tom Duff

Organizational Approval (Optional)

HRM 3/23/88 *EMK* 8/24/88

A. V. Aho

For Use by Recipient of Cover Sheet:

Computing network users may order copies via the library - k command;  
for information, type man library after the UNIX prompt.

Otherwise:

Enter: PAN if AT&T-BL (or SS# if non-AT&T-BL).

Internal Technical Document Service

- AK 2H-28
- ALC 1B-102
- CB 30-2011
- WH 3E-204
- IH 7M-103
- MV 1L-19
- DR 2F-19
- INH 1C-114
- IW 2Z-158
- MT 3B-117
- NW-ITDS
- PR 5-2120

Subject: Viral Attacks On UNIX® System Security  
Work Program- 311403-0101 -- File- 39199-11

date: July 29, 1988

from: Tom Duff

TM: 11273-880728-02TMS

## TECHNICAL MEMORANDUM

# Viral Attacks On UNIX® System Security

## 1. Introduction

UNIX system security has been a subject of intense interest for many years, in large part because the UNIX system is more secure than any other operating system offering comparable facilities and as such offers a more challenging target for recreational breaking-and-entering. The *ne plus ultra* of UNIX system security breaking is to have the super-user execute arbitrary code on behalf of the miscreant. The most common way to do this is to find a root-owned set-userid program that calls the shell and exploit any of a number of well-known loopholes to get it to execute a chosen command file. [Reeds] describes a number of 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] describes an extremely clever surreptitious way of doing the former; [Grampp & Morris] 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 writable binary files and patching copies of itself into them. It would have to be careful to preserve the functionality of the modified programs, in order to avoid detection. Eventually it might so thoroughly infect executable files that it would be very unlikely for the super-user never to execute it.

## 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, the author 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. All other things being equal, it has about two chances in three of finding enough space to copy itself into a given binary.

Once a system is seeded with a few copies of the virus, and with a little luck, someone will sooner or later execute one of the modified binaries 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], so there is a good chance of the infection spreading to files on other machines. We also have an automatic software distribution system [Koenig], 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 which 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 in fact 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 systems by the automatic software distribution system [Koenig]. The experiment was stopped on September 18, at which time 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 of the machines infected by the virus was Giacobini. Doug McIlroy and Jim Reeds are using this system to develop a multi-level secure version of the 9th edition UNIX system that retains as much of the flavor of standard insecure UNIX systems as possible. Their system augments the familiar UNIX system file protection scheme by attaching a bit vector, called a 'security label', to each file. The security label indicates which security clearances are required to read the file. If a program reads a file and then writes another file, the second file's security label is updated to be at least as restrictive as the first, since the write must be assumed to have contaminated it with more highly classified data. The system takes many special precautions to avoid inadvertent or covert declassification of data. In particular, certain programs which cannot be made to work within the protection scheme, like `/etc/login`, must be designated 'trusted'. Any attempt to change a trusted binary must be denied by the kernel, as there is no automatic way to certify continued trust in the modified program. Even programs cleared to access the most highly classified material must not be allowed such a privilege, since high clearance does not imply permission to thwart the classification system, and that is precisely what the notion of 'trust' allows.

Somehow a program infected by the virus found its way onto Giacobini. 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
```

```
chk $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.

#### 4. More Vigorous Viruses

`Inf` is not very virulent, and its only insalubrious effect is the mild system degradation that its execution causes. This is a consequence of a desire to keep the size of the program down to maximize the number of 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 fit 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 it's zeroed itself, the virus must jump to the first instruction of its host. This seems tricky, but it should be possible to do it by copying the cleanup code into the stack.

The virus is further 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 cpu it ran on, and could be made fairly insensitive to 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 lq $i | grep '^#![ ]*/bin/sh' # [ ] is blank or tab
    then
      if grep '^# mark$' $i
      then :
      else
        trap "rm -f /tmp/x$$" 0 1 2 13 15
        sed lq $i >/tmp/x$$
        sed 'ld
            /^# mark$/q' $0 >>/tmp/x$$
        sed ld $i >>/tmp/x$$
        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 a number of 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, and within an hour had infected about 140 files, at which time 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 of 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.

UNIX system administrators and users can take many measures to limit the danger of viral attack.

- 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. In the best situation 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 persistent media (disks, tapes, Williams tubes, etc.) can be reformatted and reloaded with old data if any infection appears. Ideal conditions often do not obtain. 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] 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 of 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 very well have been infected.

## 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 indicates a system organization in which all of a program's accesses to files are authorized by the user running the program. [Lai & Gray] 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) are actually 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 one of the system programs 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]

### 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 602 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 the first change 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.

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 requires. In such cases, the `chown` command can easily be made root-owned and `set-userid` and modified to implement any appropriate policy.

### 8. Acknowledgements

Some of the ideas described here arose in conversations with Norman Wilson and Fred Grampp. Ron Gomes helped make the Bourne shell virus more portable.

MH-11273-TD

Tom Duff

Att.  
References



## References

- [Grampp & Morris] 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, pp. 1649-1672
- [Koenig] Andrew R. Koenig, *Automatic Software Distribution*, Usenix 1984 Summer Conference Proceedings, pp. 312-322
- [Lai & Gray] Nick Lai and Terence E. Gray, *Strengthening Discretionary Access Controls to Inhibit Trojan Horses and Computer Viruses*, Proceedings of the Summer 1988 USENIX Conference, pp. 275-286
- [Reeds] Jim Reeds, */bin/sh: the biggest UNIX security loophole*, AT&T Bell Laboratories Technical Memo, 1988
- [Thompson] Ken Thompson, *Reflections on Trusting Trust*, Comm. ACM Vol 27 Number 8 (August 1984) pp. 761-763 (1983 Turing Award lecture)
- [Ritchie & Thompson] D. M. Ritchie and K. Thompson, *The UNIX Time-Sharing System*, Comm. ACM, 17, No. 7 (July 1974), pp. 365-375.
- [Weinberger] Peter J. Weinberger, *The Version 8 Network File System (abstract)*, Usenix 1984 Summer Conference Proceedings, pg. 86

