Secure Programming for Linux and Unix HOWTO

Table of Contents

Table of Contents

Table of Contents

Secure Programming for Linux and Unix HOWTO

David A. Wheeler, `dwheeler@dwheeler.com`

version 1.60, 4 April 2000

This paper provides a set of design and implementation guidelines for writing secure programs for Linux and Unix systems. Such programs include application programs used as viewers of remote data, CGI scripts, network servers, and setuid/setgid programs.

1.Introduction

A wise man attacks the city of the mighty and pulls down the stronghold in which they trust.
Proverbs 21:22 (NIV)

This paper describes a set of design and implementation guidelines for writing secure programs on Linux and Unix systems. For purposes of this paper, a ``secure program'' is a program that sits on a security boundary, taking input from a source that does not have the same access rights as the program. Such programs include application programs used as viewers of remote data, CGI scripts, network servers, and setuid/setgid programs. This paper does not address modifying the operating system kernel itself, although many of the principles discussed here do apply. These guidelines were developed as a survey of ``lessons learned'' from various sources on how to create such programs (along with additional observations by the author), reorganized into a set of larger principles.

This paper does not cover assurance measures, software engineering processes, and quality assurance approaches, which are important but widely discussed elsewhere. Such measures include testing, peer review, configuration management, and formal methods. Documents specifically identifying sets of development assurance measures for security issues include the Common Criteria [CC 1999] and the System Security Engineering Capability Maturity Model [SSE−CMM 1999]. More general sets of software engineering methods or processes are defined in documents such as the Software Engineering Institute's Capability

Maturity Model for Software (SE−CMM), ISO 9000 (along with ISO 9001 and ISO 9001−3), and ISO 12207.

This paper does not discuss how to configure a system (or network) to be secure in a given environment. This is clearly necessary for secure use of a given program, but a great many other documents discuss secure configurations. For example, information on configuring a Linux system to be secure is available in a wide variety of documents including Fenzi [1999], Seifried [1999], and Wreski [1998].

This paper assumes that the reader understands computer security issues in general, the general security model of Unix−like systems, and the C programming language. This paper does include some information about the Linux and Unix programming model for security.

While this paper covers all Unix−like systems, including Linux and the various strains of Unix, it particularly stresses Linux and provides details about Linux specifically. There are several reasons for this. One simple reason is popularity: according to one survey, in 1999 significantly more servers were installed with Linux than with all Unix operating system types combined (25% for Linux versus 15% for all Unix system types combined) [Shankland 2000]. Also, the original version of this document only discussed Linux, so although its scope has expanded, the Linux information is still noticeably dominant. If you identify areas where this can be improved, please let me know.

You can find the master copy of this document at http://www.dwheeler.com/secure−programs. This document is also part of the Linux Documentation Project (LDP) at http://www.linuxdoc.org It's also mirrored in several other places. Please note that these mirrors, including the LDP copy and/or the copy in your distribution, may be older than the master copy. I'd like to hear comments on this document, but please do not send comments until you've checked to make sure that your comment is valid for the latest version.

This paper first discusses the background of Unix, Linux, and security. The next section describes the general Unix and Linux security model, giving an overview of the security attributes and operations of processes, filesystem objects, and so on. This is followed by the meat of this paper, a set of design and implementation guidelines for developing applications on Linux and Unix systems. This is broken into validating all input, avoiding buffer overflows, structuring program internals and approach, carefully calling out to other resources, judiciously sending information back, and finally information on special topics (such as how to acquire random numbers). The paper ends with conclusions and references.

2.Background

I issued an order and a search was made, and it was found that this city has a long history of revolt against kings and has been a place of rebellion and sedition. Ezra 4:19 (NIV)

2.1 History of Unix, Linux, and Open Source Software

Unix

In 1969−1970, Kenneth Thompson, Dennis Ritchie, and others at AT&T Bell Labs began developing a small operating system on a little−used PDP−7. The operating system was soon christened Unix, a pun on an earlier operating system project called MULTICS. In 1972−1973 the system was rewritten in the programming language C, an unusual step that was visionary: due to this decision, Unix was the first widely−used operating system that could switch from and outlive its original hardware. Other innovations were added to Unix as well, in part due to synergies between Bell Labs and the academic community. In 1979, the ``seventh edition'' (V7) version of Unix was released, the grandfather of all extant Unix systems.

After this point, the history of Unix becomes somewhat convoluted. The academic community, led by Berkeley, developed a variant called the Berkeley Software Distribution (BSD), while AT&T continued developing Unix under the names ``System III'' and later ``System V''. In the late 1980's through early 1990's the ``wars'' between these two major strains raged. After many years each variant adopted many of the key features of the other. Commercially, System V won the ``standards wars'' (getting most of its interfaces into the formal standards), and most hardware vendors switched to AT&T's System V. However, System V ended up incorporating many BSD innovations, so the resulting system was more a merger of the two branches. The BSD branch did not die, but instead became widely used for research, for PC hardware, and for single−purpose servers (e.g., many web sites use a BSD derivative).

The result was many different versions of Unix, all based on the original seventh edition. Most versions of Unix were proprietary and maintained by their respective hardware vendor, for example, Sun Solaris is a variant of System V. Three versions of the BSD branch of Unix ended up as open source: FreeBSD (concentating on ease−of−installation for PC−type hardware), NetBSD (concentrating on many different CPU architectures), and a variant of NetBSD, OpenBSD (concentrating on security). More general information can be found at http://www.datametrics.com/tech/unix/uxhistry/brf−hist.htm. Much more information about the BSD history can be found in [McKusick 1999] and ftp://ftp.freebsd.org/pub/FreeBSD/FreeBSD−current/src/share/misc/bsd−family−tree.

Those interested in reading an advocacy piece that presents arguments for using Unix−like systems should see http://www.unix−vs−nt.org.

Free Software Foundation

In 1984 Richard Stallman's Free Software Foundation (FSF) began the GNU project, a project to create a free version of the Unix operating system. By free, Stallman meant software that could be freely used, read, modified, and redistributed. The FSF successfully built a vast number of useful components, including a C compiler (gcc), an impressive text editor (emacs), and a host of fundamental tools. However, in the 1990's the FSF was having trouble developing the operating system kernel [FSF 1998]; without a kernel the rest of their software would not work.

Linux

In 1991 Linus Torvalds began developing an operating system kernel, which he named ``Linux'' [Torvalds 1999]. This kernel could be combined with the FSF material and other components (in particular some of the BSD components and MIT's X−windows software) to produce a freely−modifiable and very useful operating system. This paper will term the kernel itself the ``Linux kernel'' and an entire combination as ``Linux''. Note that many use the term ``GNU/Linux'' instead for this combination.

In the Linux community, different organizations have combined the available components differently. Each combination is called a ``distribution'', and the organizations that develop distributions are called ``distributors''. Common distributions include Red Hat, Mandrake, SuSE, Caldera, Corel, and Debian. There are differences between the various distributions, but all distributions are based on the same foundation: the Linux kernel and the GNU glibc libraries. Since both are covered by ``copyleft'' style licenses, changes to these foundations generally must be made available to all, a unifying force between the Linux distributions at their foundation that does not exist between the BSD and AT&T−derived Unix systems. This paper is not specific to any Linux distribution; when it discusses Linux it presumes Linux kernel version 2.2 or greater and the C library glibc 2.1 or greater, valid assumptions for essentially all current major Linux distributions.

Open Source Software

Increased interest in such ``free software'' has made it increasingly necessary to define and explain it. A widely used term is ``open source software'', which further defined in [OSI 1999]. Eric Raymond [1997, 1998] wrote several seminal articles examining its development process. Another widely−used term is ``free software'', where the ``free'' is short for ``freedom'': the usual explanation is ``free speech, not free beer''. Neither phrase is perfect. The term ``free software'' is often confused with programs whose executables are given away at no charge, but whose source code cannot be viewed, modified, or redistributed. Conversely, the term ``open source'' is sometime (ab)used to mean software whose source code is visible, but for which there are limitations on use, modification, or redistribution. This paper uses the term ``open source'' for its usual meaning, that is, software which has its source code freely available for use, viewing, modification, and redistribution. Those interested in reading advocacy pieces for open source software should see http://www.opensource.org and http://www.fsf.org.

Comparing Linux and Unix

This paper uses the term ``Unix−like'' to describe systems intentionally like Unix. In particular, the term ``Unix−like'' includes all major Unix variants and Linux distributions.

Linux is not derived from Unix source code, but its interfaces are intentionally like Unix. Therefore, Unix lessons learned generally apply to both, including information on security. Most of the information in this paper applies to any Unix−like system. Linux−specific information has been intentionally added to enable those using Linux to take advantage of Linux's capabilities.

Unix−like systems share a number of security mechanisms, though there are subtle differences and not all systems have all mechanisms available. All include user and group ids (uids and gids) for each process and a filesystem with read, write, and execute permissions (for user, group, and other). See Thompson [1974] and Bach [1986] for general information on Unix systems, including their basic security mechanisms. Section 3 summarizes key Unix and Linux security mechanisms.

2.2 Security Principles

There are many general security principles which you should be familiar with; consult a general text on computer security such as [Pfleeger 1997]. Typically computer security goals are described in terms of three overall goals:

- *Confidentiality* (also known as secrecy), meaning that the computing system's assets are accessible only by authorized parties.
- *Integrity*, meaning that the assets can only be modified by authorized parties in authorized ways.
- *Availability*, meaning that the assets are accessible to the authorized parties. This goal is often referred to by its antonym, denial of service.

Saltzer [1974] and Saltzer and Schroeder [1975] list the following principles of the design of secure protection systems, which are still valid:

- *Least privilege*. Each user and program should operate using the fewest privileges possible. This principle limits the damage from an accident, error, or attack. It also reduces the number of potential interactions among privileged programs, so unintentional, unwanted, or improper uses of privilege are less likely to occur. This idea can be extended to the internals of a program: only the smallest portion of the program which needs those privileges should have them.
- *Economy of mechanism*. The protection system's design should be simple and small as possible. In their words, ``techniques such as line−by−line inspection of software and physical examination of hardware that implements protection mechanisms are necessary. For such techniques to be successful, a small and simple design is essential.''
- *Open design*. The protection mechanism must not depend on attacker ignorance. Instead, the mechanism should be public, depending on the secrecy of relatively few (and easily changeable) items like passwords or private keys. An open design makes extensive public scrutiny possible, and it also makes it possible for users to convince themselves that the system about to be used is adequate. Frankly, it isn't realistic to try to maintain secrecy for a system that is widely distributed; decompilers and subverted hardware can quickly expose any ``secrets'' in an implementation. Bruce Schneier argues that smart engineers should ``demand open source code for anything related to security'', as well as ensuring that it receives widespread review and that any identified problems are fixed [Schneier 1999].
- *Complete mediation*. Every access attempt must be checked; position the mechanism so it cannot be subverted. For example, in a client−server model, generally the server must do all access checking because users can build or modify their own clients.
- *Fail−safe defaults (e.g., permission−based approach)*. The default should be denial of service, and the protection scheme should then identify conditions under which access is permitted.
- *Separation of privilege*. Ideally, access to objects should depend on more than one condition, so that defeating one protection system won't enable complete access.
- *Least common mechanism*. Minimize the amount and use of shared mechanisms (e.g. use of the /tmp or /var/tmp directories). Shared objects provide potentially dangerous channels for information flow and unintended interactions.
- *Psychological acceptability / Easy to use*. The human interface must be designed for ease of use so users will routinely and automatically use the protection mechanisms correctly. Mistakes will be reduced if the security mechanisms closely match the user's mental image of his or her protection goals.

2.3 Types of Secure Programs

Many different types of programs may need to be secure programs (as the term is defined in this paper). Some common types are:

- Application programs used as viewers of remote data. Programs used as viewers (such as word processors or file format viewers) are often asked to view data sent remotely by an untrusted user (this request may be automatically invoked by a web browser). Clearly, the untrusted user's input should not be allowed to cause the application to run arbitrary programs. It's usually unwise to support initialization macros (run when the data is displayed); if you must, then you must create a secure sandbox (a complex and error−prone task). Be careful of issues such as buffer overflow, discussed later, which might allow an untrusted user to force the viewer to run an arbitrary program.

- Application programs used by the administrator (root). Such programs shouldn't trust information that can be controlled by non−administrators.
- Local servers (also called daemons).
- Network−accessible servers (sometimes called network daemons).
- CGI scripts. These are a special case of network−accessible servers, but they're so common they deserve their own category. Such programs are invoked indirectly via a web server, which filters out some attacks but nevertheless leaves many attacks that must be withstood.
- setuid/setgid programs. These programs are invoked by a local user and, when executed, are immediately granted the privileges of the program's owner and/or owner's group. In many ways these are the hardest programs to secure, because so many of their inputs are under the control of the untrusted user and some of those inputs are not obvious.

This paper merges the issues of these different types of program into a single set. The disadvantage of this approach is that some of the issues identified here don't apply to all types of programs. In particular, setuid/setgid programs have many surprising inputs and several of the guidelines here only apply to them. However, things are not so clear−cut, because a particular program may cut across these boundaries (e.g., a CGI script may be setuid or setgid, or be configured in a way that has the same effect), and some programs are divided into several executables each of which can be considered a different ``type'' of program. The advantage of considering all of these program types together is that we can consider all issues without trying to apply an inappropriate category to a program. As will be seen, many of the principles apply to all programs that need to be secured.

There is a slight bias in much of this paper towards programs written in C, with some notes on other languages such as C++, Perl, Python, Ada95, and Java. This is because C is the most common language for implementing secure programs on Unix−like systems (other than CGI scripts, which tend to use Perl), and most other languages' implementations call the C library. This is not to imply that C is somehow the ``best'' language for this purpose, and most of the principles described here apply regardless of the programming language used.

2.4 Paranoia is a Virtue

The primary difficulty in writing secure programs is that writing them requires a different mindset, in short, a paranoid mindset. The reason is that the impact of errors (also called defects or bugs) can be profoundly different.

Normal non−secure programs have many errors. While these errors are undesirable, these errors usually involve rare or unlikely situations, and if a user should stumble upon one they will try to avoid using the tool that way in the future.

In secure programs, the situation is reversed. Certain users will intentionally search out and cause rare or unlikely situations, in the hope that such attacks will give them unwarranted privileges. As a result, when writing secure programs, paranoia is a virtue.

2.5 Why Did I Write This Document?

One question I've been asked is ``why did you write this document''? Here's my answer: Over the last several years I've noticed that many developers for Linux and Unix seem to keep falling into the same security pitfalls, again and again. Auditors were slowly catching problems, but it would have been better if the problems weren't put into the code in the first place. I believe that part of the problem was that there wasn't a single, obvious place where developers could go and get information on how to avoid known pitfalls. The information was publicly available, but it was often hard to find, out−of−date, incomplete, or had other problems. Most such information didn't particularly discuss Linux at all, even though it was becoming widely used! That leads up to the answer: I developed this document in the hope that future software developers for Linux won't repeat past mistakes, resulting in an even more secure form of Linux. I added Unix, since it's often wise to make sure that programs can port between these systems. You can see a larger discussion of this at http://www.linuxsecurity.com/feature_stories/feature_story−6.html.

A related question that could be asked is ``why did you write your own document instead of just referring to other documents''? There are several answers:

- Much of this information was scattered about; placing the critical information in one organized document makes it easier to use.
- Some of this information is not written for the programmer, but is written for an administrator or user.
- Much of the available information emphasizes portable constructs (constructs that work on all Unix−like systems), and failed to discuss Linux at all. It's often best to avoid Linux−unique abilities for portability's sake, but sometimes the Linux−unique abilities can really aid security. Even if non−Linux portability is desired, you may want to support the Linux−unique abilities when running on Linux. And, by emphasizing Linux, I can include references to information that is helpful to someone targeting Linux that is not necessarily true for others.

2.6 Sources of Design and Implementation Guidelines

Several documents help describe how to write secure programs (or, alternatively, how to find security problems in existing programs), and were the basis for the guidelines highlighted in the rest of this paper.

AUSCERT has released a programming checklist [AUSCERT 1996], based in part on chapter 22 of Garfinkel and Spafford's book discussing how to write secure SUID and network programs [Garfinkel 1996]. Matt Bishop [1996, 1997] has developed several extremely valuable papers and presentations on the topic. Galvin [1998a] described a simple process and checklist for developing secure programs; he later updated the checklist in Galvin [1998b]. Sitaker [1999] presents a list of issues for the ``Linux security audit'' team to search for. Shostack [1999] defines another checklist for reviewing security−sensitive code. Other useful information sources include the *Secure Unix Programming FAQ*[Al−Herbish 1999], the *Security−Audit's Frequently Asked Questions*[Graham 1999], and Ranum [1998]. Some recommendations must be taken with caution, for example, Anonymous [unknown] recommends the use of access(3) without noting the dangerous race conditions that usually accompany it. Wood [1985] has some useful but dated advice in its ``Security for Programmers'' chapter. Bellovin [1994] and FreeBSD [1999] also include useful guidelines.

There are many documents giving security guidelines for programs using the Common Gateway Interface (CGI) to interface with the web. These include Gundavaram [unknown], Kim [1996], Phillips [1995], Stein [1999], and Webber [1999].

There are also many documents describing the issue from the other direction (i.e., ``how to crack a system''). One example is McClure [1999], and there's countless amounts of material from that vantage point on the Internet.

This paper is a summary of what I believe are the most useful guidelines; it is not a complete list of all possible guidelines. The organization presented here is my own (every list has its own, different structure), and the Linux−unique guidelines (e.g., on capabilities and the fsuid value) are also my own. Reading all of the referenced documents listed above as well is highly recommended.

2.7 Document Conventions

System manual pages are referenced in the format *name(number)*, where *number* is the section number of the manual. The pointer value that means ``does not point anywhere'' is called NULL; C compilers will convert the integer 0 to the value NULL in most circumstances where a pointer is needed, but note that nothing in the C standard requires that NULL actually be implemented by a series of all−zero bits. C and C++ treat the character '\0' (ASCII 0) specially, and this value is referred to as NIL in this paper (this is usually called ``NUL'', but ``NUL'' and ``NULL'' sound identical). Function and method names always use the correct case, even if that means that some sentences must begin with a lower case letter. I use the term ``Unix−like'' to mean Unix, Linux, or other systems whose underlying models are very similar to Unix; I can't say POSIX, because there are systems such as Windows 2000 that implement portions of POSIX yet have vastly different security models. An attacker is called an ``attacker'', ``cracker'', or ``adversary''. Some journalists use the word ``hacker'' instead of ``attacker''; this paper avoids this (mis)use, because many Linux and Unix developers refer to themselves as ``hackers'' in the traditional non−evil sense of the term. That is, to many Linux and Unix developers, the term ``hacker'' continues to mean simply an expert or enthusiast, particularly regarding computers. This document uses the ``new'' or ``logical'' quoting system, instead of the traditional American quoting system: quoted information does not include any trailing punctuation if the punctuation is not part of the material being quoted. While this may cause a minor loss of typographical beauty, the traditional American system causes extraneous characters to be placed inside the quotes. These extraneous characters have no effect on prose but can be disastrous in code or computer commands.

3.Summary of Linux and Unix Security Features

Discretion will protect you, and understanding will guard you. Proverbs 2:11 (NIV)

Before discussing guidelines on how to use Linux or Unix security features, it's useful to know what those features are from a programmer's viewpoint. This section briefly describes those features that are widely available on nearly all Unix−like systems. However, note that there is considerable variation between different versions of Unix−like systems, and not all systems have the abilities described here. This chapter also notes some extensions or features specific to Linux; Linux distributions tend to be fairly similar to each other from the point−of−view of programming for security, because they all use essentially the same kernel and C library (and the GPL−based licenses encourage rapid dissemination of any innovations). This chapter doesn't discuss issues such as implementations of mandatory access control (MAC) which many Unix−like systems do not implement. If you already know what those features are, please feel free to skip this section.

Many programming guides skim briefly over the security−relevant portions of Linux or Unix and skip important information. In particular, they often discuss ``how to use'' something in general terms but gloss over the security attributes that affect their use. Conversely, there's a great deal of detailed information in the manual pages about individual functions, but the manual pages sometimes obscure key security issues with detailed discussions on how to use each individual function. This section tries to bridge that gap; it gives an overview of the security mechanisms in Linux that are likely to be used by a programmer, but concentrating specifically on the security ramifications. This section has more depth than the typical programming guides, focusing specifically on security−related matters, and points to references where you can get more details.

First, the basics. Linux and Unix are fundamentally divided into two parts: the kernel and ``user space''. Most programs execute in user space (on top of the kernel). Linux supports the concept of ``kernel modules'', which is simply the ability to dynamically load code into the kernel, but note that it still has this fundamental division. Some other systems (such as the HURD) are ``microkernel'' based systems; they have a small kernel with more limited functionality, and a set of ``user'' programs that implement the lower−level functions traditionally implemented by the kernel.

Some Unix−like systems have been extensively modified to support strong security, in particular to support U.S. Department of Defense requirements for Mandatory Access Control (level B1 or higher). This version of this paper doesn't cover these systems or issues; I hope to expand to that in a future version.

When users log in, their usernames are mapped to integers marking their ``UID'' (for ``user id'') and the ``GID''s (for ``group id'') that they are a member of. UID 0 is a special privileged user (role) traditionally called ``root''; on most Unix−like systems (including Unix) root can overrule most security checks and is used to administrate the system. Processes are the only ``subjects'' in terms of security (that is, only processes are active objects). Processes can access various data objects, in particular filesystem objects (FSOs), System V Interprocess Communication (IPC) objects, and network ports. Processes can also set signals. Other security−relevant topics include quotas and limits, libraries, auditing, and PAM. The next few subsections detail this.

3.1 Processes

In Unix−like systems, user−level activities are implemented by running processes. Most Unix systems support a ``thread'' as a separate concept; threads share memory inside a process, and the system scheduler actually schedules threads. Linux does this differently (and in my opinion uses a better approach): there is no essential difference between a thread and a process. Instead, in Linux, when a process creates another process it can choose what resources are shared (e.g., memory can be shared). The Linux kernel then performs optimizations to get thread−level speeds; see clone(2) for more information. When programming, it's usually better to use one of the standard thread libraries that hide these differences.

Process Attributes

Here are typical attributes associated with each process in a Unix−like system:

- RUID, RGID − real UID and GID of the user on whose behalf the process is running
- EUID, EGID − effective UID and GID used for privilege checks (except for the filesystem)
- SUID, SGID − Saved UID and GID; used to support switching permissions ``on and off'' as discussed below. Not all Unix−like systems support this.
- supplemental groups − a list of groups (GIDs) in which this user has membership.
- umask − a set of bits determining the default access control settings when a new filesystem object is created; see umask(2).
- scheduling parameters − each process has a scheduling policy, and those with the default policy SCHED_OTHER have the additional parameters nice, priority, and counter. See sched_setscheduler(2) for more information.
- limits − per−process resource limits (see below).
- filesystem root − the process' idea of where the root filesystem begins; see chroot(2).

Here are less−common attributes associated with processes:

- FSUID, FSGID − UID and GID used for filesystem access checks; this is usually equal to the EUID and EGID respectively. This is a Linux−unique attribute.
- capabilities − POSIX capability information; there are actually three sets of capabilities on a process: the effective, inheritable, and permitted capabilities. See below for more information on POSIX capabilities. Linux kernel version 2.2 and greater support this; some other Unix−like systems do too, but it's not as widespread.

In Linux, if you really need to know exactly what attributes are associated with each process, the most definitive source is the Linux source code, in particular /usr/include/linux/sched.h's definition of task_struct.

The portable way to create new processes it use the fork(2) call. BSD introduced a variant called vfork(2) as an optimization technique. The bottom line with vfork(2) is simple: *don't* use it if you can avoid it. In vfork(2), unlike fork(2), the child borrows the parent's memory and thread of control until a call to execve(2V) or an exit occurs; the parent process is suspended while the child is using its resources. The rationale is that in old BSD systems, fork(2) would actually cause memory to be copied while vfork(2) would not. Linux never had this problem; because Linux used copy−on−write semantics internally, Linux only copies pages when they changed (actually, there are still some tables that have to be copied; in most circumstances their overhead is not significant). Nevertheless, since some programs depend on vfork(2), recently Linux implemented the BSD vfork(2) semantics (previously it had been an alias for fork(2)). The problem with vfork(2) is that it's actually fairly tricky for a process to not interfere with its parent, especially in high−level languages. The result: programs using vfork(2) can easily fail when code changes or even when compiler versions change. Avoid vfork(2) in most cases; its primary use is to support old programs that needed vfork's semantics.

Linux supports the Linux−unique clone(2) call. This call works like fork(2), but allows specification of which resources should be shared (e.g., memory, file descriptors, etc.). Portable programs shouldn't use this call directly; as noted earlier, they should instead rely on threading libraries that use the call to implement threads.

This document is not a full tutorial on writing programs, so I will skip widely−available information handling processes. You can see the documentation for wait(2), exit(2), and so on for more information.

POSIX Capabilities

POSIX capabilities are sets of bits that permit splitting of the privileges typically held by root into a larger set of more specific privileges. POSIX capabilities are defined by a draft IEEE standard; they're not unique to Linux but they're not universally supported by other Unix−like systems either. Linux kernel 2.0 did not support POSIX capabilities, while version 2.2 added support for POSIX capabilities to processes. When Linux documentation (including this one) says ``requires root privilege'', in nearly all cases it really means ``requires a capability'' as documented in the capability documentation. If you need to know the specific capability required, look it up in the capability documentation.

In Linux, the eventual intent is to permit capabilities to be attached to files in the filesystem; as of this writing, however, this is not yet supported. There is support for transferring capabilities, but this is disabled by default. Linux version 2.2.11 added a feature that makes capabilities more directly useful, called the ``capability bounding set''. The capability bounding set is a list of capabilities that are allowed to be held by any process on the system (otherwise, only the special init process can hold it). If a capability does not appear in the bounding set, it may not be exercised by any process, no matter how privileged. This feature can be used to, for example, disable kernel module loading. A sample tool that takes advantage of this is LCAP at http://pweb.netcom.com/~spoon/lcap/.

More information about POSIX capabilities is available at ftp://linux.kernel.org/pub/linux/libs/security/linux−privs.

Process Creation and Manipulation

Processes may be created using fork(2), the non−recommended vfork(2), or the Linux−unique clone(2); all of these system calls duplicate the existing process, creating two processes out of it. A process can execute a different program by calling execve(2), or various front−ends to it (for example, see exec(3), system(3), and popen(3)).

When a program is executed, and its file has its setuid or setgid bit set, the process' EUID or EGID (respectively) is set to the file's value. Note that under Linux this does not occur with ordinary scripts such as shell scripts, because there are a number of security dangers when trying to do this with scripts. Some other Unix−like systems do support setuid shell scripts, but since they're a security problem they're best avoided in new applications. As a special case, Perl includes a special setup to support setuid Perl scripts; if you truly need to support this, examine how Perl does this.

In some cases a process can affect the various UID and GID values; see setuid(2), seteuid(2), setreuid(2), and the Linux−unique setfsuid(2). In particular the saved user id (SUID) attribute is there to permit trusted programs to temporarily switch UIDs. Unix−like systems supporting the SUID use the following rules: If the RUID is changed, or the EUID is set to a value not equal to the RUID, the SUID is set to the new EUID.

Unprivileged users can set their EUID from their SUID, the RUID to the EUID, and the EUID to the RUID.

The Linux−unique FSUID process attribute is intended to permit programs like the NFS server to limit themselves to only the filesystem rights of some given UID without giving that UID permission to send signals to the process. Whenever the EUID is changed, the FSUID is changed to the new EUID value; the FSUID value can be set separately using setfsuid(2), a Linux−unique call. Note that non−root callers can only set FSUID to the current RUID, EUID, SEUID, or current FSUID values.

3.2 Files

On all Unix−like systems, the primary repository of information is the file tree, rooted at ``/''. The file tree is a hierarchical set of directories, each of which may contain filesystem objects (FSOs).

In Linux, filesystem objects (FSOs) may be ordinary files, directories, symbolic links, named pipes (also called first−in first−outs or FIFOs), sockets (see below), character special (device) files, or block special (device) files (in Linux, this list is given in the find(1) command). Other Unix−like systems have an identical or similar list of FSO types.

Filesystem objects are collected on filesystems, which can be mounted and unmounted on directories in the file tree. A filesystem type (e.g., ext2 and FAT) is a specific set of conventions for arranging data on the disk to optimize speed, reliability, and so on; many people use the term ``filesystem'' as a synonym for the filesystem type.

Filesystem Object Attributes

Different Unix−like systems support different filesystem types. Filesystems may have slightly different sets of access control attributes and access controls can be affected by options selected at mount time. On Linux, the ext2 filesystems is currently the most popular filesystem, but Linux supports a vast number of filesystems. Most Unix−like systems tend to support multiple filesystems too.

Most filesystems on Unix−like systems store at least the following:

- owning UID and GID − identifies the ``owner'' of the filesystem object. Only the owner or root can change the access control attributes unless otherwise noted.
- permission bits − read, write, execute bits for each of user (owner), group, and other. For ordinary files, read, write, and execute have their typical meanings. In directories, the ``read'' permission is necessary to display a directory's contents, while the ``execute'' permission is sometimes called ``search'' permission and is necessary to actually enter the directory to use its contents. In a directory ``write'' permission on a directory permits adding, removing, and renaming files in that directory; if you only want to permit adding, set the sticky bit noted below. Note that the permission values of symbolic links are never used; it's only the values of their containing directories and the linked−to file that matter.
- ``sticky'' bit − when set on a directory, unlinks (removes) and renames of files in that directory are limited to the file owner, the directory owner, or root privileges. This is a very common Unix extension and is specified in the Open Group's Single Unix Specification version 2. Old versions of Unix called this the ``save program text'' bit and used this to indicate executable files that should stay in memory. Systems that did this ensured that only root could set this bit (otherwise users could have crashed systems by forcing ``everything'' into memory). In Linux, this bit has no affect on ordinary

files and ordinary users can modify this bit on the files they own: Linux's virtual memory management makes this old use irrelevant.

- setuid, setgid − when set on an executable file, executing the file will set the process' effective UID or effective GID to the value of the file's owning UID or GID (respectively). All Unix−like systems support this. In Linux and System V systems, when setgid is set on a file that does not have any execute privileges, this indicates a file that is subject to mandatory locking during access (if the filesystem is mounted to support mandatory locking); this overload of meaning surprises many and is not universal across Unix−like systems. In fact, the Open Group's Single Unix Specification version 2 for chmod(3) permits systems to ignore requests to turn on setgid for files that aren't executable if such a setting has no meaning. In Linux and Solaris, when setgid is set on a directory, files created in the directory will have their GID automatically reset to that of the directory's GID. The purpose of this approach is to support ``project directories'': users can save files into such specially−set directories and the group owner automatically changes. However, setting the setgid bit on directories is not specified by standards such as the Single Unix Specification [Open Group 1997].
- timestamps − access and modification times are stored for each filesystem object. However, the owner is allowed to set these values arbitrarily (see touch(1)), so be careful about trusting this information. All Unix−like systems support this.

The following are attributes are Linux−unique extensions on the ext2 filesystem, though many other filesystems have similar functionality:

- immutable bit − no changes to the filesystem object are allowed; only root can set or clear this bit. This is only supported by ext2 and is not portable across all Unix systems (or even all Linux filesystems).
- append−only bit − only appending to the filesystem object are allowed; only root can set or clear this bit. This is only supported by ext2 and is not portable across all Unix systems (or even all Linux filesystems).

Other common extensions include some sort of bit indicating ``cannot delete this file''.

Many of these values can be influenced at mount time, so that, for example, certain bits can be treated as though they had a certain value (regardless of their values on the media). See mount(1) for more information about this. Some filesystems don't support some of these access control values; again, see mount(1) for how these filesystems are handled. In particular, many Unix−like systems support MS−DOS disks, which by default support very few of these attributes (and there's not standard way to define these attributes). In that case, Unix−like systems emulate the standard attributes (possibly implementing them through special on−disk files), and these attributes are generally influenced by the mount(1) command.

It's important to note that, for adding and removing files, only the directory ACLs (and file owner) really matter unless the Unix−like system supports more complex schemes (such as POSIX ACLs). Unless the system has special extensions (stock Linux doesn't), a file that has no permissions granted can still be removed if its containing directory permits it. Also, if an ancestor directory permits its children to be changed by some user or group, then any of that directory's descendents can be replaced by that user or group.

The draft IEEE POSIX standard on security defines a technique for true ACLs that support a list of users and groups with their permissions. Unfortunately, this is not widely supported nor supported exactly the same way across Unix−like systems. Stock Linux 2.2, for example, has neither ACLs nor POSIX capability values in the filesystem.

It's worth noting that in Linux, the Linux ext2 filesystem by default reserves a small amount of space for the root user. This is a partial defense against denial−of−service attacks; even if a user fills a disk that is shared with the root user, the root user has a little space left over (e.g., for critical functions). The default is 5% of the filesystem space; see mke2fs(8), in particular its ``−m'' option.

Creation Time Initial Values

At creation time, the following rules apply. On most Unix systems, when a new filesystem object is created via creat(2) or open(2), the FSO UID is set to the process' EUID and the FSO's GID is set to the process' EGID. Linux works slightly differently due to its FSUID and setgid directory extensions; the FSO's UID is set to the process' FSUID, and the FSO GID is set to the process' FSGUID; if the containing directory's setgid bit is set or the filesystem's ``GRPID'' flag is set, the FSO GID is actually set to the GID of the containing directory. This special case supports ``project'' directories: to make a ``project'' directory, create a special group for the project, create a directory for the project owned by that group, then make the directory setgid: files placed there are automatically owned by the project. Similarly, if a new subdirectory is created inside a directory with the setgid bit set (and the filesystem GRPID isn't set), the new subdirectory will also have its setgid bit set (so that project subdirectories will ``do the right thing''.); in all other cases the setgid is clear for a new file. FSO basic access control values (read, write, execute) are computed from (requested values & ~ umask of process). New files always start with a clear sticky bit and clear setuid bit.

Changing Access Control Attributes

You can set most of these values with chmod(2) or chmod(1), but see also chown(1), and chgrp(1). In Linux, some the Linux−unique attributes are manipulated using chattr(1).

Note that in Linux, only root can change the owner of a given file. Some Unix−like systems allow ordinary users to transfer ownership of their files to another, but this causes complications and is forbidden by Linux. For example, if you're trying to limit disk usage, allowing such operations would allow users to claim that large files actually belonged to some other ``victim''.

Using Access Control Attributes

Under Linux and most Unix−like systems, reading and writing attribute values are only checked when the file is opened; they are not re−checked on every read or write. Still, a large number of calls do check these attributes, since the filesystem is so central to Unix−like systems. Calls that check these attributes include open(2), creat(2), link(2), unlink(2), rename(2), mknod(2), symlink(2), and socket(2).

Filesystem Hierarchy

Over the years conventions have been built on ``what files to place where''. Where possible, please follow conventional use when placing information in the hierarchy. For example, place global configuration information in /etc. The Filesystem Hierarchy Standard (FHS) tries to define these conventions in a logical manner, and is widely used by Linux systems. The FHS is an update to the previous Linux Filesystem Structure standard (FSSTND), incorporating lessons learned and approaches from Linux, BSD, and System V systems. See http://www.pathname.com/fhs for more information about the FHS. A summary of these conventions is in hier(5) for Linux and hier(7) for Solaris. Sometimes different conventions disagree; where possible, make these situations configurable at compile or installation time.

3.3 System V IPC

Many Unix−like systems, including Linux and System V systems, support System V interprocess communication (IPC) objects. Indeed System V IPC is required by the Open Group's Single UNIX Specification, Version 2 [Open Group 1997]. System V IPC objects can be one of three kinds: System V message queues, semaphore sets, and shared memory segments. Each such object has the following attributes:

- read and write permissions for each of creator, creator group, and others.
- creator UID and GID − UID and GID of the creator of the object.
- owning UID and GID − UID and GID of the owner of the object (initially equal to the creator UID).

When accessing such objects, the rules are as follows:

- if the process has root privileges, the access is granted.
- if the process' EUID is the owner or creator UID of the object, then the appropriate creator permission bit is checked to see if access is granted.
- if the process' EGID is the owner or creator GID of the object, or one of the process' groups is the owning or creating GID of the object, then the appropriate creator group permission bit is checked for access.
- otherwise, the appropriate ``other'' permission bit is checked for access.

Note that root, or a process with the EUID of either the owner or creator, can set the owning UID and owning GID and/or remove the object. More information is available in ipc(5).

3.4 Sockets and Network Connections

Sockets are used for communication, particularly over a network. Sockets were originally developed by the BSD branch of Unix systems, but they are generally portable to other Unix−like systems: Linux and System V variants support sockets as well, and socket support is required by the Open Group's Single Unix Specification [Open Group 1997]. System V systems traditionally used a different (incompatible) network communication interface, but it's worth noting that systems like Solaris include support for sockets. Socket(2) creates an endpoint for communication and returns a descriptor, in a manner similar to open(2) for files. The parameters for socket specify the protocol family and type, such as the Internet domain (TCP/IP version 4), Novell's IPX, or the ``Unix domain''. A server then typically calls bind(2), listen(2), and accept(2) or select(2). A client typically calls bind(2) (though that may be omitted) and connect(2). See these routine's respective man pages for more information.

The ``Unix domain sockets'' don't actually represent a network protocol; they can only connect to sockets on the same machine. (at the time of this writing for the standard Linux kernel). When used as a stream, they are fairly similar to named pipes, but with significant advantages. In particular, Unix domain socket is connection−oriented; each new connection to the socket results in a new communication channel, a very different situation than with named pipes. Because of this property, Unix domain sockets are often used instead of named pipes to implement IPC for many important services. Just like you can have unnamed pipes, you can have unnamed Unix domain sockets using socketpair(2); unnamed Unix domain sockets are useful for IPC in a way similar to unnamed pipes.

There are several interesting security implications of Unix domain sockets. First, although Unix domain sockets can appear in the filesystem and can have stat(2) applied to them, you can't use open(2) to open them (you have to use the socket(2) and friends interface). Second, Unix domain sockets can be used to pass file descriptors between processes (not just the file's contents). This odd capability, not available in any other IPC mechanism, has been used to hack all sorts of schemes (the descriptors can basically be used as a limited version of the ``capability" in the computer science sense of the term). File descriptors are sent using sendmsg(2), where the msg (message)'s field msg_control points to an array of control message headers (field msg_controllen must specify the number of bytes contained in the array). Each control message is a struct cmsghdr followed by data, and for this purpose you want the cmsg_type set to SCM_RIGHTS. A file descriptor is retrieved through recvmsg(2) and then tracked down in the analogous way. Frankly, this feature is quite baroque, but it's worth knowing about.

Standard Unix convention is that binding to TCP and UDP local port numbers less than 1024 requires root privilege, while any process can bind to an unbound port number of 1024 or greater. Linux follows this convention, more specifically, Linux requires a process to have the capability CAP_NET_BIND_SERVICE to bind to a port number less than 1024; this capability is normally only held by processes with an euid of 0. The adventurous can check this in Linux by examining its Linux's source; in Linux 2.2.12, it's file /usr/src/linux/net/ipv4/af_inet.c, function inet_bind().

3.5 Signals

Signals are a simple form of ``interruption" in the Unix−like OS world, and are an ancient part of Unix. A process can set a ``signal" on another process (say using kill(1) or kill(2)), and that other process would receive and handle the signal asynchronously. For a process to have permission to send a signal to some other process, the sending process must either have root privileges, or the real or effective user ID of the sending process must equal the real or saved set−user−ID of the receiving process.

Although signals are an ancient part of Unix, they've had different semantics in different implementations. Basically, they involve questions such as ``what happens when a signal occurs while handling another signal"? The older Linux libc 5 used a different set of semantics for some signal operations than the newer GNU libc libraries. For more information, see the glibc FAQ (on some systems a local copy is available at /usr/doc/glibc−*/FAQ).

For new programs, just use the POSIX signal system (which in turn was based on BSD work); this set is widely supported and doesn't have the problems that some of the older signal systems did. The POSIX signal system is based on using the sigset_t datatype, which can be manipulated through a set of operations: sigemptyset(), sigfillset(), sigaddset(), sigdelset(), and sigismember(). You can read about these in sigsetops(3). Then use sigaction(2), sigaction(2), sigprocmask(2), sigpending(2), and sigsuspend(2) to set up an manipulate signal handling (see their man pages for more information).

In general, make any signal handlers very short and simple, and look carefully for race conditions. Signals, since they are by nature asynchronous, can easily cause race conditions.

A common convention exists for servers: if you receive SIGHUP, you should close any log files, reopen and reread configuration files, and then re−open the log files. This supports reconfiguration without halting the server and log rotation without data loss. If you are writing a server where this convention makes sense, please support it.

3.6 Quotas and Limits

Many Unix−like systems have mechanisms to support filesystem quotas and process resource limits. This certainly includes Linux. These mechanisms are particularly useful for preventing denial of service attacks; by limiting the resources available to each user, you can make it hard for a single user to use up all the system resources. Be careful with terminology here, because both filesystem quotas and process resource limits have ``hard'' and ``soft'' limits but the terms mean slightly different things.

You can define storage (filesystem) quota limits on each mountpoint for the number of blocks of storage and/or the number of unique files (inodes) that can be used, and you can set such limits for a given user or a given group. A ``hard'' quota limit is a never−to−exceed limit, while a ``soft'' quota can be temporarily exceeded. See quota(1), quotactl(2), and quotaon(8).

The rlimit mechanism supports a large number of process quotas, such as file size, number of child processes, number of open files, and so on. There is a ``soft'' limit (also called the current limit) and a ``hard limit'' (also called the upper limit). The soft limit cannot be exceeded at any time, but through calls it can be raised up to the value of the hard limit. See getrlimit(), setrlimit(), and getrusage(). Note that there are several ways to set these limits, including the PAM module pam_limits.

3.7 Dynamically Linked Libraries

Practically all programs depend on libraries to execute. In most modern Unix−like systems, including Linux, programs are by default compiled to use *dynamically linked libraries* (DLLs). That way, you can update a library and all the programs using that library will use the new (hopefully improved) version if they can.

Dynamically linked libraries are typically placed in one a few special directories. The usual directories include /lib, /usr/lib, /lib/security for PAM modules, /usr/X11R6/lib for X−windows, and /usr/local/lib.

There are special conventions for naming libraries and having symbolic links for them, with the result that you can update libraries and still support programs that want to use old, non−backward−compatible versions of those libraries. There are also ways to override specific libraries or even just specific functions in a library when executing a particular program. This is a real advantage of Unix−like systems over Windows−like systems; I believe Unix−like systems have a much better system for handling library updates, one reason that Unix and Linux systems are reputed to be more stable than Windows−based systems.

On GNU glibc−based systems, including all Linux systems, the list of directories automatically searched during program start−up is stored in the file /etc/ld.so.conf. Many Red Hat−derived distributions don't normally include /usr/local/lib in the file /etc/ld.so.conf. I consider this a bug, and adding /usr/local/lib to /etc/ld.so.conf is a common ``fix'' required to run many programs on Red Hat−derived systems. If you want to just override a few functions in a library, but keep the rest of the library, you can enter the names of overriding libraries (.o files) in /etc/ld.so.preload; these ``preloading'' libraries will take precedence over the standard set. This preloading file is typically used for emergency patches; a distribution usually won't include such a file when delivered. Searching all of these directories at program start−up would be too time−consuming, so a caching arrangement is actually used. The program ldconfig(8) by default reads in the file /etc/ld.so.conf, sets up the appropriate symbolic links in the dynamic link directories (so they'll follow the standard conventions), and then writes a cache to /etc/ld.so.cache that's then used by other programs. So, ldconfig has to be run whenever a DLL is added, when a DLL is removed, or when the set of DLL directories changes; running ldconfig is often one of the steps performed by package managers when installing a library. On start−up, then, a program uses the dynamic loader to read the file /etc/ld.so.cache and then load the libraries it needs.

Various environment variables can control this process, and in fact there are environment variables that permit you to override this process (so, for example, you can temporarily substitute a different library for this particular execution). In Linux, the environment variable LD_LIBRARY_PATH is a colon−separated set of directories where libraries should be searched for first, before the standard set of directories; this is useful when debugging a new library or using a nonstandard library for special purposes. The variable LD_PRELOAD lists object files with functions that override the standard set, just as /etc/ld.so.preload does.

Permitting user control over dynamically linked libraries would be disastrous for setuid/setgid programs if special measures weren't taken. Therefore, in the GNU glibc implementation, if the program is setuid or setgid these variables (and other similar variables) are ignored or greatly limited in what they can do. The GNU glibc library determines if a program is setuid or setgid by checking the program's credentials; if the uid and euid differ, or the gid and the egid differ, the library presumes the program is setuid/setgid (or descended from one) and therefore greatly limits its abilities to control linking. If you load the GNU glibc libraries, you can see this; see especially the files elf/rtld.c and sysdeps/generic/dl−sysdep.c. This means that if you cause the uid and gid to equal the euid and egid, and then call a program, these variables will have full effect. Other Unix−like systems handle the situation differently but for the same reason: a setuid/setgid program should not be unduly affected by the environment variables set.

3.8 Audit

Different Unix−like systems handle auditing differently. In Linux, the most common ``audit'' mechanism is syslogd(8), usually working in conjuction with klogd(8). You might also want to look at wtmp(5), utmp(5), lastlog(8), and acct(2). Some server programs (such as the Apache web server) also have their own audit trail mechanisms. According to the FHS, audit logs should be stored in /var/log or its subdirectories.

3.9 PAM

Sun Solaris and nearly all Linux systems use the Pluggable Authentication Modules (PAM) system for authentication. PAM permits run−time configuration of authentication methods (e.g., use of passwords, smart cards, etc.). PAM will be discussed more fully later in this document.

4.Validate All Input

Wisdom will save you from the ways of wicked men, from men whose words are perverse...
Proverbs 2:12 (NIV)

Some inputs are from untrustable users, so those inputs must be validated (filtered) before being used. You should determine what is legal and reject anything that does not match that definition. Do not do the reverse (identify what is illegal and reject those cases), because you are likely to forget to handle an important case. Limit the maximum character length (and minimum length if appropriate), and be sure to not lose control when such lengths are exceeded (see the buffer overflow section below for more about this).

For strings, identify the legal characters or legal patterns (e.g., as a regular expression) and reject anything not matching that form. There are special problems when strings contain control characters (especially linefeed or NIL) or shell metacharacters; it is often best to ``escape'' such metacharacters immediately when the input is received so that such characters are not accidentally sent. CERT goes further and recommends escaping all characters that aren't in a list of characters not needing escaping [CERT 1998, CMU 1998]. see the section on ``limit call−outs to valid values'', below, for more information.

Limit all numbers to the minimum (often zero) and maximum allowed values. Filenames should be checked; usually you will want to not include ``..'' (higher directory) as a legal value. In filenames it's best to prohibit any change in directory, e.g., by not including ``/'' in the set of legal characters. A full email address checker is actually quite complicated, because there are legacy formats that greatly complicate validation if you need to support all of them; see mailaddr(7) and IETF RFC 822 [RFC 822] for more information if such checking is necessary.

These tests should usually be centralized in one place so that the validity tests can be easily examined for correctness later.

Make sure that your validity test is actually correct; this is particularly a problem when checking input that will be used by another program (such as a filename, email address, or URL). Often these tests are have subtle errors, producing the so−called ``deputy problem'' (where the checking program makes different assumptions than the program that actually uses the data).

While parsing user input, it's a good idea to temporarily drop all privileges. This is especially true if the parsing task is complex (e.g., you use to use a lex−like or yacc−like tool), or if the programming language doesn't protect against buffer overflows (e.g., C and C++). See the section below on minimizing permissions.

The following subsections discuss different kinds of inputs to a program; note that input includes process state such as environment variables, umask values, and so on. Not all inputs are under the control of an untrusted user, so you need only worry about those inputs that are.

4.1 Command line

Many programs use the command line as an input interface, accepting input by being passed arguments. A setuid/setgid program has a command line interface provided to it by an untrusted user, so it must defend itself. Users have great control over the command line (through calls such as the execve(3) call). Therefore, setuid/setgid programs must validate the command line inputs and must not trust the name of the program reported by command line argument zero (the user can set it to any value including NULL).

4.2 Environment Variables

By default, environment variables are inherited from a process' parent. However, when a program executes another program, the calling program can set the environment variables to arbitrary values. This is dangerous to setuid/setgid programs, because their invoker can completely control the environment variables they're given. Since they are usually inherited, this also applies transitively; a secure program might call some other program and, without special measures, would pass potentially dangerous environment variables values on to the program it calls.

Some Environment Variables are Dangerous

Some environment variables are dangerous because many libraries and programs are controlled by environment variables in ways that are obscure, subtle, or undocumented. For example, the IFS variable is used by the *sh* and *bash* shell to determine which characters separate command line arguments. Since the shell is invoked by several low−level calls (like system(3) and popen(3) in C, or the back−tick operator in Perl), setting IFS to unusual values can subvert apparently−safe calls. This behavior is documented in bash and sh, but it's obscure; many long−time users only know about IFS because of its use in breaking security, not because it's actually used very often for its intended purpose. What is worse is that not all environment variables are documented, and even if they are, those other programs may change and add dangerous environment variables. Thus, the only real solution (described below) is to select the ones you need and throw away the rest.

Environment Variable Storage Format is Dangerous

Normally, programs should use the standard access routines to access environment variables. For example, in C, you should get values using getenv(3), set them using the POSIX standard routine putenv(3) or the BSD extension setenv(3) and eliminate environment variables using unsetenv(3). I should note here that setenv(3) is implemented in Linux, too. However, crackers need not be so nice; crackers can directly control the environment variable data area passed to a program using execve(2). This permits some nasty attacks, which can only be understood by understanding how environment variables really work. In Linux, you can see environ(5) for a summary how about environment variables really work. In short, environment variables are internally stored as a pointer to an array of pointers to characters; this array is stored in order and terminated by a NULL pointer (so you'll know when the array ends). The pointers to characters, in turn, each point to a NIL−terminated string value of the form ``NAME=value''. This has several implications, for example, environment variable names can't include the equal sign, and neither the name nor value can have embedded NIL characters. However, a more dangerous implication of this format is that it allows multiple entries with the same variable name, but with different values (e.g., more than one value for SHELL). While typical command shells prohibit doing this, a locally−executing cracker can create such a situation using execve(2).

The problem with this storage format (and the way it's set) is that a program might check one of these values (to see if it's valid) but actually use a different one. In Linux, the GNU glibc libraries try to shield programs from this; glibc 2.1's implementation of getenv will always get the first matching entry, setenv and putenv will always set the first matching entry, and unsetenv will actually unset *all* of the matching entries (congratulations to the GNU glibc implementors for implementing unsetenv this way!). However, some programs go directly to the environ variable and iterate across all environment variables; in this case, they might use the last matching entry instead of the first one. As a result, if checks were made against the first matching entry instead, but the actual value used is the last matching entry, a cracker can use this fact to circumvent the protection routines.

The Solution − Extract and Erase

For secure setuid/setgid programs, the short list of environment variables needed as input (if any) should be carefully extracted. Then the entire environment should be erased, followed by resetting a small set of necessary environment variables to safe values. There really isn't a better way if you make any calls to subordinate programs; there's no practical method of listing ``all the dangerous values''. Even if you reviewed the source code of every program you call directly or indirectly, someone may add new undocumented environment variables after you write your code, and one of them may be exploitable.

The simple way to erase the environment is by setting the global variable *environ* to NULL. The global variable environ is defined in <unistd.h>; C/C++ users will want to #include this header file. You will need to manipulate this value before spawning threads, but that's rarely a problem, since you want to do these manipulations very early in the program's execution. Another way is to use the undocumented clearenv() function. clearenv() has an odd history; it was supposed to be defined in POSIX.1, but somehow never made it into that standard. However, clearenv() is defined in POSIX.9 (the Fortran 77 bindings to POSIX), so there is a quasi−official status for it. clearenv() is defined in <stdlib.h>, but before using #include to include it you must make sure that __USE_MISC is #defined.

One value you'll almost certainly re−add is PATH, the list of directories to search for programs; PATH should *not* include the current directory and usually be something simple like ``/bin:/usr/bin''. Typically you'll also set IFS (to its default of `` \t\n'') and TZ (timezone). Linux won't die if you don't supply either IFS or TZ, but some System V based systems have problems if you don't supply a TZ value, and it's rumored that some shells need the IFS value set. In Linux, see environ(5) for a list of common environment variables that you *might* want to set.

If you really need user−supplied values, check the values first (to ensure that the values match a pattern for legal values and that they are within some reasonable maximum length). Ideally there would be some standard trusted file in /etc with the information for ``standard safe environment variable values'', but at this time there's no standard file defined for this purpose. For something similar, you might want to examine the PAM module pam_env on those systems which have that module.

If you're programming a setuid/setgid program in a language that doesn't allow you to reset the environment directly, one approach is to create a ``wrapper'' program. The wrapper sets the environment program to safe values, and then calls the other program. Beware: make sure the wrapper will actually invoke the intended program; if it's an interpreted program, make sure there's no race condition possible that would allow the interpreter to load a different program than the one that was granted the special setuid/setgid privileges.

4.3 File Descriptors

A program is passed a set of ``open file descriptors'', that is, pre−opened files. A setuid/setgid program must deal with the fact that the user gets to select what files are open and to what (within their permission limits). A setuid/setgid program must not assume that opening a new file will always open into a fixed file descriptor id. It must also not assume that standard input, standard output, and standard error refer to a terminal or are even open.

4.4 File Contents

If a program takes directions from a file, it must not trust that file specially unless only a trusted user can control its contents. Usually this means that an untrusted user must not be able to modify the file, its directory, or any of its ancestor directories. Otherwise, the file must be treated as suspect.

If the directions in the file are supposed to be from an untrusted user, then make sure that the inputs from the file are protected as describe throughout this document. In particular, check that values match the set of legal values, and that buffers are not overflowed.

4.5 CGI Inputs

CGI inputs are internally a specified set of environment variables and standard input. These values must be validated.

One additional complication is that many CGI inputs are provided in so−called ``URL−encoded" format, that is, some values are written in the format %HH where HH is the hexadecimal code for that byte. You or your CGI library must handle these inputs correctly by URL−decoding the input and then checking if the resulting byte value is acceptable. You must correctly handle all values, including problematic values such as %00 (NIL) and %0A (newline). Don't decode inputs more than once, or input such as ``%2500" will be mishandled (the %25 would be translated to ``%", and the resulting ``%00" would be erroneously translated to the NIL character).

CGI scripts are commonly attacked by including special characters in their inputs; see the comments above.

Some HTML forms include client−side checking to prevent some illegal values. This checking can be helpful for the user but is useless for security, because attackers can send such ``illegal" values directly to the web server. As noted below (in the section on trusting only trustworthy channels), servers must perform all of their own input checking.

4.6 Other Inputs

Programs must ensure that all inputs are controlled; this is particularly difficult for setuid/setgid programs because they have so many such inputs. Other inputs programs must consider include the current directory, signals, memory maps (mmaps), System V IPC, and the umask (which determines the default permissions of newly−created files). Consider explicitly changing directories (using chdir(2)) to an appropriately fully named directory at program startup.

4.7 Character Encoding

For many years Americans have been using the ASCII encoding of characters, permitting easy exchange of English texts. Unfortunately, ASCII is completely inadequate in handling the character sets of most other languages. For many years different countries have adopted different techniques for exchanging text in different languages. More recently, ISO has developed ISO 10646, a single 31−bit encoding for all of the world's characters. Characters fitting into 16 bits are termed the ``Basic Multilingual Plane" (BMP), and the BMP is intended to cover nearly all spoken languages. The Unicode forum develops the Unicode standard, which concentrates on the 16−bit set and adds some additional conventions to aid interoperability.

However, most software is not designed to handle 16 bit or 32 bit characters, so a special format called ``UTF−8" was developed to encode these characters in a format more easily handled by existing programs. The UTF−8 transformation format is becoming a dominant method for exchanging international text information because it can support all of the world's languages, yet it is backward compatible with U.S. ASCII files. It is defined, among other places, in IETF RFC 2279, and I recommend its use.

The reason to mention UTF−8 is that some byte sequences are not legal UTF−8, and this might be an exploitable security hole. The RFC notes the following:

```
Implementors of UTF−8 need to consider the security aspects of how
they handle illegal UTF−8 sequences.  It is conceivable that in some
circumstances an attacker would be able to exploit an incautious
UTF−8 parser by sending it an octet sequence that is not permitted by
the UTF−8 syntax.

A particularly subtle form of this attack could be carried out
against a parser which performs security−critical validity checks
against the UTF−8 encoded form of its input, but interprets certain
illegal octet sequences as characters.  For example, a parser might
prohibit the NUL character when encoded as the single−octet sequence
00, but allow the illegal two−octet sequence C0 80 and interpret it
as a NUL character.  Another example might be a parser which
prohibits the octet sequence 2F 2E 2E 2F ("/../"), yet permits the
illegal octet sequence 2F C0 AE 2E 2F.
```

A longer discussion about this is available at Markus Kuhn's *UTF−8 and Unicode FAQ for Unix/Linux* at http://www.cl.cam.ac.uk/~mgk25/unicode.html.

The UTF−8 character set is one case where it's possible to enumerate all illegal values (and prove that you've enumerated them all). If you need to determine if you have a legal UTF−8 sequence, you need to check for two things: (1) is the initial sequence legal, and (2) if it is, is the first byte followed by the required number of valid continuation characters? Performing the first check is easy; the following is provably the complete list of all illegal UTF−8 initial sequences:

```
 10xxxxxx                    illegal as initial byte of character (80..BF)
 1100000x                    illegal, overlong (C0 80..BF)
 11100000 100xxxxx           illegal, overlong (E0 80..9F)
 11110000 1000xxxx           illegal, overlong (F0 80..8F)
 11111000 10000xxx           illegal, overlong (F8 80..87)
 11111100 100000xx           illegal, overlong (FC 80..83)
 1111111x                    illegal; prohibited by spec
```

I should note that in some cases, you might want to cut slack (or use internally) the hexadecimal sequence C0 80. This is an overlong sequence that could represent ASCII NUL (NIL). Since C/C++ have trouble including a NIL character in an ordinary string, some people have taken to using this sequence when they want to represent NIL as part of the data stream; Java even enshrines the practice. Feel free to use C0 80 internally while processing data, but technically you really should translate this back to 00 before saving the data. Depending on your needs, you might decide to be ``sloppy'' and accept C0 80 as input in a UTF−8 data stream.

The second step is to check if the correct number of continuation characters are included in the string. If the first byte has the top 2 bits set, you count the number of ``one'' bits set after the top one, and then check that there are that many continuation bytes which begin with the bits ``10''. So, binary 11100001 requires two more continuation bytes.

A related issue is that some phrases can be expressed in more than one way in ISO 10646/Unicode. For example, some accented characters can be represented as a single character (with the accent) and also as a set of characters (e.g., the base character plus a separate composing accent). These two forms may appear identical. There's also a zero−width space that could be inserted, with the result that apparently−similar items are considered different. Beware of situations where such hidden text could interfere with the program.

4.8 Limit Valid Input Time and Load Level

Place timeouts and load level limits, especially on incoming network data. Otherwise, an attacker might be able to easily cause a denial of service by constantly requesting the service.

5.Avoid Buffer Overflow

An enemy will overrun the land; he will pull down your strongholds and plunder your fortresses. Amos 3:11 (NIV)

An extremely common security flaw is the ``buffer overflow''. Technically, a buffer overflow is a problem with the program's internal implementation, but it's such a common and serious problem that I've placed this information in its own chapter. To give you an idea of how important this subject is, at the CERT, 9 of 13 advisories in 1998 and at least half of the 1999 advisories involved buffer overflows. An informal survey on Bugtraq found that approximately 2/3 of the respondents felt that buffer overflows were the leading cause of security vulnerability (the remaining respondents identified ``misconfiguration'' as the leading cause) [Cowan 1999]. This is an old, well−known problem, yet it continues to resurface [McGraw 2000].

A buffer overflow occurs when you write a set of values (usually a string of characters) into a fixed length buffer and write at least one value outside that buffer's boundaries (usually past its end). A buffer overflow can occur when reading input from the user into a buffer, but it can also occur during other kinds of processing in a program.

If a secure program permits a buffer overflow, the overflow can often be exploited by an adversary. If the buffer is a local C variable, the overflow can be used to force the function to run code of an attackers' choosing. This specific variation is often called a ``stack smashing'' attack. A buffer in the heap isn't much better; attackers may be able to use such overflows to control other variables in the program. More details can be found from Aleph1 [1996], Mudge [1995], or the Nathan P. Smith's "Stack Smashing Security Vulnerabilities" website at http://destroy.net/machines/security/.

Most programming languages are essentially immune to this problem, either because they automatically resize arrays (e.g., Perl), or because they normally detect and prevent buffer overflows (e.g., Ada95). However, the C language provides no protection against such problems, and C++ can be easily used in ways to cause this problem too.

5.1 Dangers in C/C++

C users must avoid using dangerous functions that do not check bounds unless they've ensured the bounds will never get exceeded. Functions to avoid in most cases include the functions strcpy(3), strcat(3), sprintf(3), and gets(3). These should be replaced with functions such as strncpy(3), strncat(3), snprintf(3), and fgets(3) respectively, but see the discussion below. The function strlen(3) should be avoided unless you can ensure that there will be a terminating NIL character to find. Other dangerous functions that may permit buffer overruns (depending on their use) include fscanf(3), scanf(3), vsprintf(3), realpath(3), getopt(3), getpass(3), streadd(3), strecpy(3), and strtrns(3). If you're serious about portability, there's an additional problem: some systems' snprintf do not actually protect against buffer overflows; Linux's version is known to work correctly.

5.2 Library Solutions in C/C++

One solution in C/C++ is to use library functions that do not have buffer overflow problems. The first subsection describes the ``standard C library'' solution, which can work but has its disadvantages. The next subsection describes the general security issues of both fixed length and dynamically reallocated approaches to buffers. The following subsections describe various alternative libraries, such as strlcpy and libmib.

Standard C Library Solution

The ``standard'' solution to prevent buffer overflow in C is to use the standard C library calls that defend against these problems. This approach depends heavily on the standard library functions strncpy(3) and strncat(3). If you choose this approach, beware: these calls have somewhat surprising semantics and are hard to use correctly. The function strncpy(3) does not NIL−terminate the destination string if the source string length is at least equal to the destination's, so be sure to set the last character of the destination string to NIL after calling strncpy(3). If you're going to reuse the same buffer many times, an efficient approach is to tell strncpy() that the buffer is one character shorter than it actually is and set the last character to NIL once before use. Both strncpy(3) and strncat(3) require that you pass the amount of space left available, a computation that is easy to get wrong (and getting it wrong could permit a buffer overflow attack). Neither provide a simple mechanism to determine if an overflow has occurred. Finally, strncpy(3) has a significant performance penalty compared to the strcpy(3) it supposedly replaces, because *strncpy(3) NIL−fills the remainder of the destination*. I've gotten emails expressing surprise over this last point, but this is clearly stated in Kernighan and Ritchie second edition [Kernighan 1988, page 249], and this behavior is clearly documented in the man pages for Linux, FreeBSD, and Solaris. This means that just changing from strcpy to strncpy can cause a severe reduction in performance, for no good reason in most cases.

Static and Dynamically Allocated Buffers

strncpy and friends are an example of statically allocated buffers, that is, once the buffer is allocated it stays a fixed size. The alternative is to dynamically reallocate buffer sizes as you need them. It turns out that both approaches have security implications.

There is a general security problem when using fixed−length buffers: the fact that the buffer is a fixed length may be exploitable. This is a problem with strncpy(3) and strncat(3), snprintf(3), strlcpy(3), strlcat(3), and other such functions. The basic idea is that the attacker sets up a really long string so that, when the string is truncated, the final result will be what the attacker wanted (instead of what the developer intended). Perhaps the string is catenated from several smaller pieces; the attacker might make the first piece as long as the entire buffer, so all later attempts to concatenate strings do nothing. Here are some specific examples:

- Imagine code that calls gethostbyname(3) and, if successful, immediately copies hostent−>h_name to a fixed−size buffer using strncpy or snprintf. Using strncpy or snprintf protects against an overflow of an excessively long fully−qualified domain name (FQDN), so you might think you're done. However, this could result in chopping off the end of the FQDN. This may be very undesirable, depending on what happens next.
- Imagine code that uses strncpy, strncat, snprintf, etc., to copy the full path of a filesystem object to some buffer. Further imagine that the original value was provided by an untrusted user, and that the copying is part of a process to pass a resulting computation to a function. Sounds safe, right? Now imagine that an attacker pads a path with a large number of '/'s at the beginning. This could result in future operations being performed on the file ``/''. If the program appends values in the belief that the

result will be safe, the program may be exploitable. Or, the attacker could devise a long filename near the buffer length, so that attempts to append to the filename would silently fail to occur (or only partially occur in ways that may be exploitable).

When using statically−allocated buffers, you really need to consider the length of the source and destination arguments. Sanity checking the input and the resulting intermediate computation might deal with this, too.

Another alternative is to dynamically reallocate all strings instead of using fixed−size buffers. This general approach is recommended by the GNU programming guidelines, since it permits programs to handle arbitrarily−sized inputs (until they run out of memory). Of course, the major problem with dynamically allocated strings is that you may run out of memory. The memory may even be exhausted at some other point in the program than the portion where you're worried about buffer overflows; any memory allocation can fail. Also, since dynamic reallocation may cause memory to be inefficiently allocated, it is entirely possible to run out of memory even though technically there is enough virtual memory available to the program to continue. In addition, before running out of memory the program will probably use a great deal of virtual memory; this can easily result in ``thrashing'', a situation in which the computer spends all its time just shuttling information between the disk and memory (instead of doing useful work). This can have the effect of a denial of service attack. Some rational limits on input size can help here. In general, the program must be designed to fail safely when memory is exhausted if you use dynamically allocated strings.

strlcpy and strlcat

An alternative, being employed by OpenBSD, is the strlcpy(3) and strlcat(3) functions by Miller and de Raadt [Miller 1999]. This is a minimalist, statically−sized buffer approach that provides C string copying and concatenation with a different (and less error−prone) interface. Source and documentation of these functions are available under a newer BSD−style open source license at ftp://ftp.openbsd.org/pub/OpenBSD/src/lib/libc/string/strlcpy.3.

First, here are their prototypes:

```
size_t strlcpy (char *dst, const char *src, size_t size);
size_t strlcat (char *dst, const char *src, size_t size);
```

Both strlcpy and strlcat take the full size of the destination buffer as a parameter (not the maximum number of characters to be copied) and guarantee to NIL−terminate the result (as long as size is larger than 0). Remember that you should include a byte for NIL in the size.

The strlcpy function copies up to size−1 characters from the NUL−terminated string src to dst, NIL−terminating the result. The strlcat function appends the NIL−terminated string src to the end of dst. It will append at most size − strlen(dst) − 1 bytes, NIL−terminating the result.

One minor disadvantage of strlcpy(3) and strlcat(3) is that they are not, by default, installed in most Unix−like systems. In OpenBSD, they are part of <string.h>. This is not that difficult a problem; since they are small functions, you can even include them in your own program's source (at least as an option), and create a small separate package to load them. You can even use autoconf to handle this case automatically. If more programs use these functions, it won't be long before these are standard parts of Linux distributions and other Unix−like systems.

libmib

One toolset for C that dynamically reallocates strings automatically is the ``libmib allocated string functions'' by Forrest J. Cavalier III, available at http://www.mibsoftware.com/libmib/astring. There are two variations of libmib; ``libmib−open'' appears to be clearly open source under its own X11−like license that permits modification and redistribution, but redistributions must choose a different name, however, the developer states that it ``may not be fully tested.'' To continuously get libmib−mature, you must pay for a subscription. The documentation is not open source, but it is freely available.

Other Libraries

There are other libraries that may help. For example, the glib library is widely available on open source platforms (the GTK+ toolkit uses glib, and glib can be used separately without GTK+). At this time I do not have an analysis showing definitively that the glib library functions protect against buffer overflow, but this seems likely. C++ has a set of string classes and templates as well (see basic_string and string). Hopefully a later edition of this document will confirm which glib and C++ string functions can be used to avoid buffer overflow issues.

5.3 Compilation Solutions in C/C++

A completely different approach is to use compilation methods that perform bounds−checking (see [Sitaker 1999] for a list). In my opinion, such tools are very useful in having multiple layers of defense, but it's not wise to use this technique as your sole defense. There are at least two reasons for this. First of all, most such tools only provide partial defense against buffer overflows (and the ``complete'' defenses are generally 12−30 times slower); C and C++ were simply not designed to protect against buffer overflow. Second of all, for open source programs you cannot be certain what tools will be used to compile the program; using the default ``normal'' compiler for a given system might suddenly open security flaws.

One of the more useful tools is ``StackGuard'', a modification of the standard GNU C compiler gcc. StackGuard works by inserting a ``guard'' value (called a ``canary'') in front of the return address; if a buffer overflow overwrites the return address, the canary's value (hopefully) changes and the system detects this before using it. This is quite valuable, but note that this does not protect against buffer overflows overwriting other values (which they may still be able to use to attack a system). There is work to extend StackGuard to be able to add canaries to other data items, called ``PointGuard''. PointGuard will automatically protect certain values (e.g., function pointers and longjump buffers). However, protecting other variable types using PointGuard requires specific programmer intervention (the programmer has to identify which data values must be protected with canaries). This can be valuable, but it's easy to accidentally omit protection for a data value you didn't think needed protection − but needs it anyway. More information on StackGuard, PointGuard, and other alternatives is in Cowan [1999].

As a related issue, in Linux you could modify the Linux kernel so that the stack segment is not executable; such a patch to Linux does exist (see Solar Designer's patch, which includes this, at http://www.openwall.com/linux/ However, as of this writing this is not built into the Linux kernel. Part of the rationale is that this is less protection than it seems; attackers can simply force the system to call other ``interesting'' locations already in the program (e.g., in its library, the heap, or static data segments). Also, sometimes Linux does require executable code in the stack, e.g., to implement signals and to implement GCC ``trampolines''. Solar Designer's patch does handle these cases, but this does complicate the patch. Personally, I'd like to see this merged into the main Linux distribution, since it does make attacks somewhat more difficult and it defends against a range of existing attacks. However, I agree with Linus Torvalds and others that this does not add the amount of protection it would appear to and can be circumvented with relative ease. You can read Linus Torvalds' explanation for not including this support at http://lwn.net/980806/a/linus−noexec.html.

In short, it's better to work first on developing a correct program that defends itself against buffer overflows. Then, after you've done this, by all means use techniques and tools like StackGuard as an additional safety net. If you've worked hard to eliminate buffer overflows in the code itself, then StackGuard is likely to be more effective because there will be fewer ``chinks in the armor'' that StackGuard will be called on to protect.

5.4 Other Languages

The problem of buffer overflows is an argument for using many other programming languages such as Perl, Python, and Ada95, which protect against buffer overflows. Using those other languages does not eliminate all problems, of course; in particular see the discussion under ``limit call−outs to valid values'' regarding the NIL character. There is also the problem of ensuring that those other languages' infrastructure (e.g., run−time library) is available and secured. Still, you should certainly consider using other programming languages when developing secure programs to protect against buffer overflows.

6.Structure Program Internals and Approach

Like a city whose walls are broken down is a man who lacks self−control. Proverbs 25:28 (NIV)

6.1 Secure the Interface

Interfaces should be minimal (simple as possible), narrow (provide only the functions needed), and non−bypassable. Trust should be minimized. Applications and data viewers may be used to display files developed externally, so in general don't allow them to accept programs unless you're willing to do the extensive work necessary to create a secure sandbox. The most dangerous kind is an auto−executing macro that executes when the application is loaded; from a security point−of−view this is a disaster waiting to happen unless you have extremely strong control over what the macro can do (a ``sandbox''), and past experience has shown that real sandboxes are hard to implement.

6.2 Minimize Privileges

As noted earlier, it is an important general principle that programs have the minimal amount of privileges necessary to do its job (this is termed ``least privilege''). That way, if the program is broken, its damage is limited. The most extreme example is to simply not write a secure program at all − if this can be done, it usually should be.

In Linux and Unix, the primary determiner of a process' privileges is the set of id's associated with it: each process has a real, effective and saved id for both the user and group. Linux also has the filesystem uid and gid. Manipulating these values is critical to keeping privileges minimized, and there are several ways to minimize them (discussed below). You can also use chroot(2) to minimize the files visible to a program.

Minimize the Privileges Granted

Perhaps the most effective technique is to simply minimize the the highest privilege granted. In particular, avoid granting a program root privilege if possible. Don't make a program *setuid root* if it only needs access to a small set of files; consider creating separate user or group accounts for different function.

A common technique is to create a special group, change a file's group ownership to that group, and then make the program *setgid* to that group. It's better to make a program *setgid* instead of *setuid* where you can, since group membership grants fewer rights (in particular, it does not grant the right to change file permissions).

This is commonly done for game high scores. Games are usually setgid *games*, the score files are owned by the group *games*, and the programs themselves and their configuration files are owned by someone else (say root). Thus, breaking into a game allows the perpetrator to change high scores but doesn't grant the privilege to change the game's executable or configuration file. The latter is important; if an attacker could change a game's executable or its configuration files (which might control what the executable runs), then they might be able to gain control of a user who ran the game.

If creating a new group isn't sufficient, consider creating a new pseudouser (really, a special role) to manage a set of resources. Web servers typically do this; often web servers are set up with a special user (``nobody'') so that they can be isolated from other users. Indeed, web servers are instructive here: web servers typically need root privileges to start up (so they can attach to port 80), but once started they usually shed all their privileges and run as the user ``nobody''. Again, usually the pseudouser doesn't own the primary program it runs, so breaking into the account doesn't allow for changing the program itself. As a result, breaking into a running web server normally does not automatically break the whole system's security.

If you *must* give a program root privileges, consider using the POSIX capability features available in Linux 2.2 and greater to minimize them immediately on program startup. By calling cap_set_proc(3) or the Linux−specific capsetp(3) routines immediately after starting, you can permanently reduce the abilities of your program to just those abilities it actually needs. Note that *not* all Unix−like systems implement POSIX capabilities, so this is an approach that can lose portability; however, if you use it merely as an optional safeguard only where it's available, using this approach will not really limit portability. Also, while the Linux kernel version 2.2 and greater includes the low−level calls, the C−level libraries to make their use easy are not installed on some Linux distributions, slightly complicating their use in applications. For more information on Linux's implementation of POSIX capabilities, see http://linux.kernel.org/pub/linux/libs/security/linux−privs.

One Linux−unique tool you can use to simplify minimizing granted privileges is the ``compartment'' tool developed by SuSE. This tool sets the fileystem root, uid, gid, and/or the capability set, then runs the given program. This is particularly handy for running some other program without modifying it. Here's the syntax of version 0.5:

```
Syntax: compartment [options] /full/path/to/program

Options:
```

```
                --chroot path   chroot to path
                --user user     change uid to this user
                --group group   change gid to this group
                --init program  execute this program/script before doing anything
                --cap capset    set capset name. You can specify several capsets.
                --verbose       be verbose
                --quiet         do no logging (to syslog)
```

Thus, you could start a more secure anonymous ftp server using:

```
         compartment --chroot /home/ftp --cap CAP_NET_BIND_SERVICE anon-ftpd
```

At the time of this writing, the tool is immature and not available on typical Linux distributions, but this may quickly change. You can download the program via http://www.suse.de/~marc.

Minimize the Time the Privilege Can Be Used

As soon as possible, permanently give up privileges. Some Unix−like systems, including Linux, implement ``saved'' IDs which store the ``previous'' value. The simplest approach is to set the other id's twice to an untrusted id. In setuid/setgid programs, you should usually set the effective gid and uid to the real ones, in particular right after a fork(2), unless there's a good reason not to. Note that you have to change the gid first when dropping from root to another privilege or it won't work − once you drop root privileges, you won't be able to change much else.

Minimize the Time the Privilege is Active

Use setuid(2), seteuid(2), and related functions to ensure that the program only has these privileges active when necessary. As noted above, you might want ensure that these privileges are disabled while parsing user input, but more generally, only turn on privileges when they're actually needed. Note that some buffer overflow attacks, if successful, can force a program to run arbitrary code, and that code could re−enable privileges that were temporarily dropped. Thus, it's always better to completely drop privileges as soon as possible. Still, temporarily disabling these permissions prevents a whole class of attacks, such as techniques to convince a program to write into a file that perhaps it didn't intent to write into. Since this technique prevents many attacks, it's worth doing if completely dropping the privileges can't be done at that point in the program.

Minimize the Modules Granted the Privilege

If only a few modules are granted the privilege, then it's much easier to determine if they're secure. One way to do so is to have a single module use the privilege and then drop it, so that other modules called later cannot misuse the privilege. Another approach is to have separate commands in separate executables; one command might be a complex tool that can do a vast number of tasks for a privileged user (e.g., root), while the other tool is setuid but is a small, simple tool that only permits a small command subset. The small, simple tool checks to see if the input meets various criteria for acceptability, and then if it determines the input is acceptable, it passes the input is passed to the tool. This can even be layerd several ways, for example, a complex user tool could call a simple setuid ``wrapping'' program (that checks its inputs for secure values) that then passes on information to another complex trusted tool. This approach is especially helpful for

GUI−based systems; have the GUI portion run as a normal user, and then pass security−relevant requests on to another program that has the special privileges for actual execution.

Some operating systems have the concept of multiple layers of trust in a single process, e.g., Multics' rings. Standard Unix and Linux don't have a way of separating multiple levels of trust by function inside a single process like this; a call to the kernel increases privileges, but otherwise a given process has a single level of trust. Linux and other Unix−like systems can sometimes simulate this ability by forking a process into multiple processes, each of which has different privilege. To do this, set up a secure communication channel (usually unnamed pipes or unnamed sockets are used), then fork into different processes and have each process drop as many privileges as possible. Then use a simple protocol to allow the less trusted processes to request actions from the more trusted process(es), and ensure that the more trusted processes only support a limited set of requests.

This is one area where technologies like Java 2 and Fluke have an advantage. For example, Java 2 can specify fine−grained permissions such as the permission to only open a specific file. However, general−purpose operating systems do not typically have such abilities at this time; this may change in the near future.

Consider Using FSUID To Limit Privileges

Each Linux process has two Linux−unique state values called filesystem user id (fsuid) and filesystem group id (fsgid). These values are used when checking against the filesystem permissions. If you're building a program that operates as a file server for arbitrary users (like an NFS server), you might consider using these Linux extensions. To use them, while holding root privileges change just fsuid and fsgid before accessing files on behalf of a normal user. This extension is fairly useful, and provides a mechanism for limiting filesystem access rights without removing other (possibly necessary) rights. By only setting the fsuid (and not the euid), a local user cannot send a signal to the process. Also, avoiding race conditions is much easier in this situation. However, a disadvantage of this approach is that these calls are not portable to other Unix−like systems.

Consider Using Chroot to Minimize Available Files

You can use chroot(2) to limit the files visible to your program. This requires carefully setting up a directory (called the ``chroot jail'') and correctly entering it. This can be a fairly effective technique for improving a program's security − it's hard to interfere with files you can't see. However, it depends on a whole bunch of assumptions, in particular, the program must lack root privileges, it must not have any way to get root privileges, and the chroot jail must be properly set up. I recommend using chroot(2) where it makes sense to do so, but don't depend on it alone; instead, make it part of a layered set of defenses. Here are a few notes about the use of chroot(2):

- The program can still use non−filesystem objects that are shared across the entire machine (such as System V IPC objects and network sockets). It's best to also use separate pseudousers and/or groups, because all Unix−like systems include the ability to isolate users; this will at least limit the damage a subverted program can do to other programs. Note that current most Unix−like systems (including Linux) won't isolate intentionally cooperating programs; if you're worried about malicious programs cooperating, you need to get a system that implements some sort of mandatory access control and/or limits covert channels.
- Be sure to close any filesystem descriptors to outside files if you don't want them used later. In particular, don't have any descriptors open to directories outside the chroot jail, or set up a situation where such a descriptor could be given to it (e.g., via Unix sockets or an old implementation of /proc). If the program is given a descriptor to a directory outside the chroot jail, it could be used to

escape out of the chroot jail.
- The chroot jail has to be set up to be secure. Place the absolute minimum number of files there. Typically you'll have a /bin, /etc/, /lib, and maybe one or two others (e.g., /pub if it's an ftp server). Place in /bin only what you need to run after doing the chroot(); sometimes you need nothing at all (try to avoid placing a shell there, though sometimes that can't be helped). You may need a /etc/passwd and /etc/group so file listings can show some correct names, but if so, try not to include the real system's values, and certainly replace all passwords with "*". In /lib, place only what you need; use ldd(1) to query each program in /bin to find out what it needs, and only include them. On Linux, you'll probably need a few basic libraries like ld−linux.so.2, and not much else. It's usually wiser to completely copy in all files, instead of making hard links; while this wastes some time and disk space, it makes it so that attacks on the chroot jail files do not automatically propogate into the regular system's files. Mounting a /proc filesystem, on systems where this is supported, is generally unwise. In fact, in 2.0.x versions of Linux it's a known security flaw, since there are pseudodirectories in /proc that would permit a chroot'ed program to escape. Linux kernel 2.2 fixed this known problem, but there may be others; if possible, don't do it.
- Chroot really isn't effective if the program can acquire root privilege. For example, the program could use calls like mknod(2) to create a device file that can view physical memory, and then use the resulting device file to modify kernel memory to give itself whatever privileges it desired. Another example of how a root program can break out of chroot is demonstrated at http://www.suid.edu/source/breakchroot.c. In this example, the program opens a file descriptor for the current directory, creates and chroots into a subdirectory, sets the current directory to the previously−opened current directory, repeatedly cd's up from the current directory (which since it is outside the current chroot succeeds in moving up to the real filesystem root), and then calls chroot on the result. By the time you read this, these weaknesses may have been plugged, but the reality is that root privilege has traditionally meant ``all privileges'' and it's hard to strip them away. It's better to assume that a program requiring continuous root privileges will only be mildly helped using chroot(). Of course, you may be able to break your program into parts, so that at least part of it can be in a chroot jail.

6.3 Configure Safely and Use Safe Defaults

Configuration is considered to currently be the number one security problem. Therefore, you should spend some effort to (1) make the initial installation secure, and (2) make it easy to reconfigure the system while keeping it secure.

A program should have the most restrictive access policy until the administrator has a chance to configure it. Please don't create ``sample'' working users or ``allow access to all'' configurations as the starting configuration; many users just ``install everything'' (installing all available services) and never get around to configuring many services. In some cases the program may be able to determine that a more generous policy is reasonable by depending on the existing authentication system, for example, an ftp server could legitimately determine that a user who can log into a user's directory should be allowed to access that user's files. Be careful with such assumptions, however.

Have installation scripts install a program as safely as possible. By default, install all files as owned by root or some other system user and make them unwriteable by others; this prevents non−root users from installing viruses. Indeed, it's best to make them unreadable by all but the trusted user. Allow non−root installation where possible as well, so that users without root privilages and administrators who do not fully trust the installer can still use the program.

Try to make configuration as easy and clear as possible, including post−installation configuration. Make using the ``secure'' approach as easy as possible, or many users will use an insecure approach without understanding the risks. On Linux, take advantage of tools like linuxconf, so that users can easily configure their system using an existing infrastructure.

If there's a configuration language, the default should be to deny access until the user specifically grants it. Include many clear comments in the sample configuration file, if there is one, so the administrator understands what the configuration does.

6.4 Fail Safe

A secure program should always ``fail safe'', that is, it should be designed so that if the program does fail, the safest result should occur. For security−critical programs, that usually means that if some sort of misbehavior is detected (malformed input, reaching a ``can't get here'' state, and so on), then the program should immediately deny service and stop processing that request. Don't try to ``figure out what the user wanted'': just deny the service. Sometimes this can decrease reliability or usability (from a user's perspective), but it increases security. There are a few cases where this might not be desired (e.g., where denial of service is much worse than loss of confidentiality or integrity), but such cases are quite rare.

Note that I recommend ``stop processing the request'', not ``fail altogether''. In particular, most servers should not completely halt when given malformed input, because that creates a trivial opportunity for a denial of service attack (the attacker just sends garbage bits to prevent you from using the service). Sometimes taking the whole server down is necessary, in particular, reaching some ``can't get here'' states may signal a problem so drastic that continuing is unwise.

Consider carefully what error message you send back when a failure is detected. if you send nothing back, it may be hard to diagnose problems, but sending back too much information may unintentionally aid an attacker. Usually the best approach is to reply with ``access denied'' or ``miscellaneous error encountered'' and then write more detailed information to an audit log (where you can have more control over who sees the information).

6.5 Avoid Race Conditions

Secure programs must determine if a request should be granted, and if so, act on that request. There must be no way for an untrusted user to change anything used in this determination before the program acts on it. This kind of race condition is sometimes termed a ``time of check − time of use'' (TOCTOU) race condition.

This issue repeatedly comes up in the filesystem. Programs should generally avoid using access(2) to determine if a request should be granted, followed later by open(2), because users may be able to move files around between these calls. A secure program should instead set its effective id or filesystem id, then make the open call directly. It's possible to use access(2) securely, but only when a user cannot affect the file or any directory along its path from the filesystem root.

6.6 Trust Only Trustworthy Channels

In general, do not trust results from untrustworthy channels.

In most computer networks (and certainly for the Internet at large), no unauthenticated transmission is trustworthy. For example, on the Internet arbitrary packets can be forged, including header values, so don't use their values as your primary criteria for security decisions unless you can authenticate them. In some cases you can assert that a packet claiming to come from the ``inside'' actually does, since the local firewall would prevent such spoofs from outside, but broken firewalls, alternative paths, and mobile code make even this assumption suspect. In a similar vein, do not assume that low port numbers (less than 1024) are trustworthy; in most networks such requests can be forged or the platform can be made to permit use of low−numbered ports.

If you're implementing a standard and inherently insecure protocol (e.g., ftp and rlogin), provide safe defaults and document clearly the assumptions.

The Domain Name Server (DNS) is widely used on the Internet to maintain mappings between the names of computers and their IP (numeric) addresses. The technique called ``reverse DNS'' eliminates some simple spoofing attacks, and is useful for determining a host's name. However, this technique is not trustworthy for authentication decisions. The problem is that, in the end, a DNS request will be sent eventually to some remote system that may be controlled by an attacker. Therefore, treat DNS results as an input that needs validation and don't trust it for serious access control.

If asking for a password, try to set up trusted path (e.g., require pressing an unforgeable key before login, or display unforgeable pattern such as flashing LEDs). Unfortunately, stock Linux doesn't have a trusted path even for its normal login sequence, and since currently normal users can change the LEDs, the LEDs can't currently be used to confirm a trusted path. When handling a password, encrypt it between trusted endpoints.

Arbitrary email (including the ``from'' value of addresses) can be forged as well. Using digital signatures is a method to thwart many such attacks. A more easily thwarted approach is to require emailing back and forth with special randomly−created values, but for low−value transactions such as signing onto a public mailing list this is usually acceptable.

If you need a trustworthy channel over an untrusted network, you need some sort of cryptologic service (at the very least, a cryptologically safe hash); see the section below on cryptographic algorithms and protocols.

Note that in any client/server model, including CGI, that the server must assume that the client can modify any value. For example, so−called ``hidden fields'' and cookie values can be changed by the client before being received by CGI programs. These cannot be trusted unless they are signed in a way the client cannot forge and the server checks the signature.

The routines getlogin(3) and ttyname(3) return information that can be controlled by a local user, so don't trust them for security purposes.

This issue applies to data referencing other data, too. For example, HTML or XML allow you to include by reference other files (e.g., DTDs and style sheets) that may be stored remotely. However, those external references could be modified so that users see a very different document than intended; a style sheet could be modified to ``white out'' words at critical locations, deface its appearance, or insert new text. External DTDs could be modified to prevent use of the document (by adding declarations that break validation) or insert different text into documents [St. Laurent 2000].

6.7 Use Internal Consistency−Checking Code

The program should check to ensure that its call arguments and basic state assumptions are valid. In C, macros such as assert(3) may be helpful in doing so.

6.8 Self−limit Resources

In network daemons, shed or limit excessive loads. Set limit values (using setrlimit(2)) to limit the resources that will be used. At the least, use setrlimit(2) to disable creation of ``core'' files. For example, by default Linux will create a core file that saves all program memory if the program fails abnormally, but such a file might include passwords or other sensitive data.

7.Carefully Call Out to Other Resources

Do not put your trust in princes, in mortal men, who cannot save. Psalms 146:3 (NIV)

7.1 Limit Call−outs to Valid Values

Ensure that any call out to another program only permits valid and expected values for every parameter. This is more difficult than it sounds, because there are many library calls or commands call lower−level routines in potentially surprising ways. For example, several system calls, such as popen(3) and system(3), are implemented by calling the command shell, meaning that they will be affected by shell metacharacters. Similarly, execlp(3) and execvp(3) may cause the shell to be called. Many guidelines suggest avoiding popen(3), system(3), execlp(3), and execvp(3) entirely and use execve(3) directly in C when trying to spawn a process [Galvin 1998b]. In a similar manner the Perl and shell backtick (`) also call a command shell.

One of the nastiest examples of this problem are shell metacharacters. The standard Unix−like command shell (stored in /bin/sh) interprets a number of characters specially. If these characters are sent to the shell, then their special interpretation will be used unless escaped; this fact can be used to break programs. According to the WWW Security FAQ [Stein 1999, Q37], these metacharacters are:

```
& ; ` ' \ " | * ? ~ < > ^ ( ) [ ] { } $ \n \r
```

Forgetting one of these characters can be disastrous, for example, many programs omit backslash as a metacharacter [rfp 1999]. As discussed in the section on validating input, a recommended approach is to immediately escape at least all of these characters when they are input.

A related problem is that the NIL character (character 0) can have surprising effects. Most C and C++ functions assume that this character marks the end of a string, but string−handling routines in other languages (such as Perl and Ada95) can handle strings containing NIL. Since many libraries and kernel calls use the C convention, the result is that what is checked is not what is actually used [rfp 1999].

When calling another program or referring to a file always specify its full path (e.g, /usr/bin/sort). For program calls, this will eliminate possible errors in calling the ``wrong'' command, even if the PATH value is incorrectly set. For other file referents, this reduces problems from ``bad'' starting directories.

7.2 Check All System Call Returns

Every system call that can return an error condition must have that error condition checked. One reason is that nearly all system calls require limited system resources, and users can often affect resources in a variety of ways. Setuid/setgid programs can have limits set on them through calls such as setrlimit(3) and nice(2). External users of server programs and CGI scripts may be able to cause resource exhaustion simply by making a large number of simultaneous requests. If the error cannot be handled gracefully, then fail open as discussed earlier.

8.Send Information Back Judiciously

Do not answer a fool according to his folly, or you will be like him yourself. Proverbs 26:4 (NIV)

8.1 Minimize Feedback

Avoid giving much information to untrusted users; simply succeed or fail, and if it fails just say it failed and minimize information on why it failed. Save the detailed information for audit trail logs. For example:

- If your program requires some sort of user authentication (e.g., you're writing a network service or login program), give the user as little information as possible before they authenticate. In particular, avoid giving away the version number of your program before authentication. Otherwise, if a particular version of your program is found to have a vulnerability, then users who don't upgrade from that version advertise to attackers that they are vulnerable.
- If your program accepts a password, don't echo it back; this creates another way passwords can be seen.

8.2 Handle Full/Unresponsive Output

It may be possible for a user to clog or make unresponsive a secure program's output channel back to that user. For example, a web browser could be intentionally halted or have its TCP/IP channel response slowed. The secure program should handle such cases, in particular it should release locks quickly (preferably before replying) so that this will not create an opportunity for a Denial−of−Service attack. Always place timeouts on outgoing network−oriented write requests.

9.Special Topics

Understanding is a fountain of life to those who have it, but folly brings punishment to fools.
Proverbs 16:22 (NIV)

9.1 Locking

There are often situations in which a program must ensure that it has exclusive rights to something (e.g., a file, a device, and/or existence of a particular server process). On Unix−like systems this has traditionally been done by creating a file to indicate a lock, because this is very portable.

However, there are several traps to avoid. First, a program with root privileges can open a file, even if it sets the exclusive mode (O_EXCL) when creating the file (O_EXCL mode normally fails if the file already exists). So, if you want to use a file to indicate a lock, but you might do this as root, don't use open(2) and the exclusive mode. A simple approach is to use link(2) instead to create a hard link to some file in the same directory; not even root can create a hard link if it already exists.

Second, if the lock file may be on an NFS−mounted filesystem, then you have the problem that NFS version 2 doesn't completely support normal file semantics. This can even be a problem for work that's supposed to be ``local'' to a client, since some clients don't have local disks and may have *all* files remotely mounted via NFS. The manual for *open(2)* explains how to handle things in this case (which also handles the case of root programs):

> ... programs which rely on [the O_CREAT and O_EXCL flags of open(2)] for performing locking tasks will contain a race condition. The solution for performing atomic file locking using a lockfile is to create a unique file on the same filesystem (e.g., incorporating hostname and pid), use link(2) to make a link to the lockfile and use stat(2) on the unique file to check if its link count has increased to 2. Do not use the return value of the link(2) call.

Obviously, this solution only works if all programs doing the locking are cooperating, and if all non−cooperating programs aren't allowed to interfere. In particular, the directories you're using for file locking must not have permissive file permissions for creating and removing files.

NFS version 3 added support for O_EXCL mode in open(2); see IETF RFC 1813, in particular the "EXCLUSIVE" value to the "mode" argument of "CREATE". Sadly, not everyone has switched to NFS version 3 at the time of this writing, so you you can't depend on this in portable programs.

If you're locking a device or the existence of a process on a local machine, try to use standard conventions. I recommend using the Filesystem Hierarchy Standard (FHS); it is widely referenced by Linux systems, but it also tries to incorporate the ideas of other Unix−like systems. The FHS describes standard conventions for such locking files, including naming, placement, and standard contents of these files [FHS 1997]. If you just want to be sure that your server doesn't execute more than once on a given machine, you should usually create a process identifier as /var/run/NAME.pid with the pid as its contents. In a similar vein, you should place lock files for things like device lock files in /var/lock. This approach has the minor disadvantage of leaving files hanging around if the program suddenly halts, but it's standard practice and that problem is easily handled by other system tools.

It's important that the programs which are cooperating using files to represent the locks use the "same" directory, not just the same directory name. This is an issue with networked systems: the FHS explicitly notes that /var/run and /var/lock are unshareable, while /var/mail is shareable. Thus, if you want the lock to work on a single machine, but not interfere with other machines, use unshareable directories like /var/run (e.g., you want to permit each machine to run its own server). However, if you want all machines sharing files in a network to obey the lock, you need to use a directory that they're sharing; /var/mail is one such location. See FHS section 2 for more information on this subject.

Of course, you need not use files to represent locks. Network servers often need not bother; the mere act of binding can act as a kind of lock, since if there's an existing server bound to a given port, no other server will be able to bind to that port.

Another approach to locking is to use POSIX record locks, implemented through fcntl(2) as a ``discretionary lock''. These are discretionary, that is, using them requires the cooperation of the programs needing the locks (just as the approach to using files to represent locks does). There's a lot to recommend POSIX record locks: POSIX record locking is supported on nearly all Unix−like platforms (it's mandated by POSIX.1), it can lock portions of a file (not just a whole file), and it can handle the difference between read locks and write locks. Even more usefully, if a process dies, its locks are automatically removed, which is usually what is desired.

You can also use mandatory locks, which are based on System V's mandatory locking scheme. These only apply to files where the locked file's setgid bit is set, but the group execute bit is not set. Also, you must mount the filesystem to permit mandatory file locks. In this case, every read(2) and write(2) is checked for locking; while this is more thorough than advisory locks, it's also slower. Also, mandatory locks don't port as widely to other Unix−like systems (they're available on Linux and System V−based systems, but not necessarily on others). Note that processes with root privileges can be held up by a mandatory lock, too, making it possible that this could be the basis of a denial−of−service attack.

9.2 Passwords

Where possible, don't write code to handle passwords. In particular, if the application is local, try to depend on the normal login authentication by a user. If the application is a CGI script, try to depend on the web server to provide the protection. If the application is over a network, avoid sending the password as cleartext (where possible) since it can be easily captured by network sniffers and reused later. ``Encrypting'' a password using some key fixed in the algorithm or using some sort of shrouding algorithm is essentially the same as sending the password as cleartext.

For networks, consider at least using digest passwords. Digest passwords are passwords developed from hashes; typically the server will send the client some data (e.g., date, time, name of server), the client combines this data with the user password, the client hashes this value (termed the ``digest pasword'') and replies just the hashed result to the server; the server verifies this hash value. This works, because the password is never actually sent in any form; the password is just used to derive the hash value. Digest passwords aren't considered ``encryption'' in the usual sense and are usually accepted even in countries with laws constraining encryption for confidentiality. Digest passwords are vulnerable to active attack threats but protect against passive network sniffers. One weakness is that, for digest passwords to work, the server must have all the unhashed passwords, making the server a very tempting target for attack.

If your application must handle passwords, overwrite them immediately after use so they have minimal exposure. In Java, don't use the type String to store a password because Strings are immutable (they will not be overwritten until garbage−collected and reused, possibly a far time in the future). Instead, in Java use char[] to store a password, so it can be immediately overwritten.

If your application permits users to set their passwords, check the passwords and permit only ``good'' passwords (e.g., not in a dictionary, having certain minimal length, etc.). You may want to look at information such as http://consult.cern.ch/writeup/security/security_3.html on how to choose a good password.

9.3 Random Numbers

``Random'' numbers generated by many library routines are intended to be used for simulations, games, and so on; they are *not* sufficiently random for use in security functions such as key generation. The problem is that these library routines use algorithms whose future values can be easily deduced by an attacker (though they may appear random). For security functions, you need random values based on truly unpredictable values such as quantum effects.

The Linux kernel (since 1.3.30) includes a random number generator, which is sufficient for many security purposes. This random number generator gathers environmental noise from device drivers and other sources into an entropy pool. When accessed as /dev/random, random bytes are only returned within the estimated number of bits of noise in the entropy pool (when the entropy pool is empty, the call blocks until additional environmental noise is gathered). When accessed as /dev/urandom, as many bytes as are requested are returned even when the entropy pool is exhausted. If you are using the random values for cryptographic purposes (e.g., to generate a key), use /dev/random. More information is available in the system documentation random(4).

9.4 Cryptographic Algorithms and Protocols

Often cryptographic algorithms and protocols are necessary to keep a system secure, particularly when communicating through an untrusted network such as the Internet. Where possible, use session encryption to foil session hijacking and to hide authentication information, as well as to support privacy.

Cryptographic algorithms and protocols are difficult to get right, so do not create your own. Instead, use existing standard−conforming protocols such as SSL, SSH, IPSec, GnuPG/PGP, and Kerberos. Use only encryption algorithms that have been openly published and withstood years of attack (examples include triple DES, which is also not encumbered by patents). In particular, do not create your own encryption algorithms unless you are an expert in cryptology and know what you're doing; creating such algorithms is a task for experts only.

In a related note, if you must create your own communication protocol, examine the problems of what's gone on before. Classics such as Bellovin [1989]'s review of security problems in the TCP/IP protocol suite might help you, as well as Bruce Schneier [1998] and Mudge's breaking of Microsoft's PPTP implementation and their follow−on work. Of course, be sure to give any new protocol widespread review, and reuse what you can.

For background information and code, you should probably look at the classic text ``Applied Cryptography'' [Schneier 1996]. Linux−specific resources include the Linux Encryption HOWTO at http://marc.mutz.com/Encryption−HOWTO/.

9.5 Java

Some security−relevant programs on Linux may be implemented using the Java language and/or the Java Virtual Machine (JVM). Developing secure programs on Java is discussed in detail in material such as Gong [1999]. The following are a few key points extracted from Gong [1999]:

- Do not use public fields or variables; declare them as private and provide accessors to them so you can limit their accessibility.
- Make methods private unless these is a good reason to do otherwise.
- Avoid using static field variables. Such variables are attached to the class (not class instances), and classes can be located by any other class. As a result, static field variables can be found by any other class, making them much more difficult to secure.
- Never return a mutable object to potentially malicious code (since the code may decide to change it).

9.6 PAM

Pluggable Authentication Modules (PAM) is a flexible mechanism for authenticating users. Many Unix−like systems support PAM, including Solaris, nearly all Linux distributions (e.g., Red Hat Linux, Caldera, and Debian as of version 2.2), and FreeBSD as of version 3.1. By using PAM, your program can be independent of the authentication scheme (passwords, SmartCards, etc.). Basically, your program calls PAM, which at run−time determines which ``authentication modules'' are required by checking the configuration set by the local system administrator. If you're writing a program that requires authentication (e.g., entering a password), you should include support for PAM. You can find out more about the Linux−PAM project at http://www.kernel.org/pub/linux/libs/pam/index.html.

9.7 Miscellaneous

Have your program check at least some of its assumptions before it uses them (e.g., at the beginning of the program). For example, if you depend on the ``sticky'' bit being set on a given directory, test it; such tests take little time and could prevent a serious problem. If you worry about the execution time of some tests on each call, at least perform the test at installation time, or even better at least perform the test on application start−up.

Write audit logs for program startup, session startup, and for suspicious activity. Possible information of value includes date, time, uid, euid, gid, egid, terminal information, process id, and command line values. You may find the function syslog(3) helpful for implementing audit logs. There is the danger that a user could create a denial−of−service attack (or at least stop auditing) by performing a very large number of events that cut an audit record until the system runs out of resources to store the records. One approach to counter to this threat is to rate−limit audit record recording; intentionally slow down the response rate if ``too many'' audit records are being cut. You could try to slow the response rate only to the suspected attacker, but in many situations a single attacker can masquerade as potentially many users.

If you have a built−in scripting language, it may be possible for the language to set an environment variable which adversely affects the program invoking the script. Defend against this.

If you need a complex configuration language, make sure the language has a comment character and include a number of commented−out secure examples. Often '#' is used for commenting, meaning ``the rest of this line is a comment''.

If possible, don't create setuid or setgid root programs; make the user log in as root instead.

Sign your code. That way, others can check to see if what's available was what was sent.

Consider statically linking secure programs. This counters attacks on the dynamic link library mechanism by making sure that the secure programs don't use it.

When reading over code, consider all the cases where a match is not made. For example, if there is a switch statement, what happens when none of the cases match? If there is an ``if'' statement, what happens when the condition is false?

Ensure the program works with compile−time and run−time checks turned on, and leave them on where practical. Perl programs should turn on the warning flag (−w), which warns of potentially dangerous or obsolete statements. Perl programs involving security should probably also include the taint flag (−T), which prevents the direct use of untrusted inputs without performing some sort of filtering. Security−relevant programs should compile cleanly with all warnings turned on. For C or C++ compilations using gcc, use at least the following as compilation flags (which turn on a host of warning messages) and try to eliminate all warnings:

```
gcc −Wall −Wpointer−arith −Wstrict−prototypes
```

10.Conclusion

The end of a matter is better than its beginning, and patience is better than pride.
Ecclesiastes 7:8 (NIV)

Designing and implementing a truly secure program is actually a difficult task on Unix−like systems such as Linux and Unix. The difficulty is that a truly secure program must respond appropriately to all possible inputs and environments controlled by a potentially hostile user. Developers of secure programs must deeply understand their platform, seek and use guidelines (such as these), and then use assurance processes (such as peer review) to reduce their programs' vulnerabilities.

In conclusion, here are some of the key guidelines from this paper:

- Validate all your inputs, including command line inputs, environment variables, CGI inputs, and so on. Don't just reject ``bad'' input; define what is an ``acceptable'' input and reject anything that doesn't match.
- Avoid buffer overflow. This is the primary programmatic error at this time.
- Structure Program Internals. Secure the interface, minimize privileges, make the initial configuration and defaults safe, and fail safe. Avoid race conditions and trust only trustworthy channels (e.g., most servers must not trust their clients for security checks).
- Carefully call out to other resources. Limit their values to valid values (in particular be concerned about metacharacters), and check all system call return values.

- Reply information judiciously. In particular, minimize feedback, and handle full or unresponsive output to an untrusted user.

11.References

The words of the wise are like goads, their collected sayings like firmly embedded nails−−given by one Shepherd. Be warned, my son, of anything in addition to them. Of making many books there is no end, and much study wearies the body. Ecclesiastes 12:11−12 (NIV)

Note that there is a heavy emphasis on technical articles available on the web, since this is where most of this kind of technical information is available.

[Al−Herbish 1999] Al−Herbish, Thamer. 1999. *Secure Unix Programming FAQ*. http://www.whitefang.com/sup.

[Aleph1 1996] Aleph1. November 8, 1996. ``Smashing The Stack For Fun And Profit". *Phrack Magazine*. Issue 49, Article 14. http://www.phrack.com/search.phtml?view&article=p49−14 or alternatively http://www.2600.net/phrack/p49−14.html.

[Anonymous unknown] *SETUID(7)*http://www.homeport.org/~adam/setuid.7.html.

[AUSCERT 1996] Australian Computer Emergency Response Team (AUSCERT) and O'Reilly. May 23, 1996 (rev 3C). *A Lab Engineers Check List for Writing Secure Unix Code*. ftp://ftp.auscert.org.au/pub/auscert/papers/secure_programming_checklist

[Bach 1986] Bach, Maurice J. 1986. *The Design of the Unix Operating System*. Englewood Cliffs, NJ: Prentice−Hall, Inc. ISBN 0−13−201799−7 025.

[Bellovin 1989] Bellovin, Steven M. April 1989. "Security Problems in the TCP/IP Protocol Suite" Computer Communications Review 2:19, pp. 32−48. http://www.research.att.com/~smb/papers/ipext.pdf

[Bellovin 1994] Bellovin, Steven M. December 1994. *Shifting the Odds −− Writing (More) Secure Software*. Murray Hill, NJ: AT&T Research. http://www.research.att.com/~smb/talks

[Bishop 1996] Bishop, Matt. May 1996. ``UNIX Security: Security in Programming". *SANS '96*. Washington DC (May 1996). http://olympus.cs.ucdavis.edu/~bishop/secprog.html

[Bishop 1997] Bishop, Matt. October 1997. ``Writing Safe Privileged Programs". *Network Security 1997* New Orleans, LA. http://olympus.cs.ucdavis.edu/~bishop/secprog.html

[CC 1999] *The Common Criteria for Information Technology Security Evaluation (CC)*. August 1999. Version 2.1. Technically identical to International Standard ISO/IEC 15408:1999. http://csrc.nist.gov/cc/ccv20/ccv2list.htm

[CERT 1998] Computer Emergency Response Team (CERT) Coordination Center (CERT/CC). February 13, 1998. *Sanitizing User−Supplied Data in CGI Scripts*. CERT Advisory CA−97.25.CGI_metachar. http://www.cert.org/advisories/CA−97.25.CGI_metachar.html.

[CMU 1998] Carnegie Mellon University (CMU). February 13, 1998 Version 1.4. ``How To Remove Meta−characters From User−Supplied Data In CGI Scripts". ftp://ftp.cert.org/pub/tech_tips/cgi_metacharacters.

[Cowan 1999] Cowan, Crispin, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. ``Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade". Proceedings of DARPA Information Survivability Conference and Expo (DISCEX), http://schafercorp−ballston.com/discex To appear at SANS 2000, http://www.sans.org/newlook/events/sans2000.htm. For a copy, see http://immunix.org/documentation.html.

[Fenzi 1999] Fenzi, Kevin, and Dave Wrenski. April 25, 1999. *Linux Security HOWTO*. Version 1.0.2. http://www.linuxdoc.org/HOWTO/Security−HOWTO.html

[FHS 1997] Filesystem Hierarchy Standard (FHS 2.0). October 26, 1997. Filesystem Hierarchy Standard Group, edited by Daniel Quinlan. Version 2.0. http://www.pathname.com/fhs.

[FreeBSD 1999] FreeBSD, Inc. 1999. ``Secure Programming Guidelines". *FreeBSD Security Information*. http://www.freebsd.org/security/security.html

[FSF 1998] Free Software Foundation. December 17, 1999. *Overview of the GNU Project*. http://www.gnu.ai.mit.edu/gnu/gnu−history.html

[Galvin 1998a] Galvin, Peter. April 1998. ``Designing Secure Software". *Sunworld*. http://www.sunworld.com/swol−04−1998/swol−04−security.html.

[Galvin 1998b] Galvin, Peter. August 1998. ``The Unix Secure Programming FAQ". *Sunworld*. http://www.sunworld.com/sunworldonline/swol−08−1998/swol−08−security.html

[Garfinkel 1996] Garfinkel, Simson and Gene Spafford. April 1996. *Practical UNIX & Internet Security, 2nd Edition*. ISBN 1−56592−148−8. Sebastopol, CA: O'Reilly & Associates, Inc. http://www.oreilly.com/catalog/puis

[Graham 1999] Graham, Jeff. May 4, 1999. *Security−Audit's Frequently Asked Questions (FAQ)*. http://lsap.org/faq.txt

[Gong 1999] Gong, Li. June 1999. *Inside Java 2 Platform Security*. Reading, MA: Addison Wesley Longman, Inc. ISBN 0−201−31000−7.

[Gundavaram Unknown] Gundavaram, Shishir, and Tom Christiansen. Date Unknown. *Perl CGI Programming FAQ*. http://language.perl.com/CPAN/doc/FAQs/cgi/perl−cgi−faq.html

[Kernighan 1988] Kernighan, Brian W., and Dennis M. Ritchie. 1988. *The C Programming Language*. Second Edition. Englewood Cliffs, NJ: Prentice−Hall. ISBN 0−13−110362−8.

[Kim 1996] Kim, Eugene Eric. 1996. *CGI Developer's Guide*. SAMS.net Publishing. ISBN: 1−57521−087−8 http://www.eekim.com/pubs/cgibook

[McClure 1999] McClure, Stuart, Joel Scambray, and George Kurtz. 1999. *Hacking Exposed: Network Security Secrets and Solutions*. Berkeley, CA: Osbourne/McGraw−Hill. ISBN 0−07−212127−0.

[McKusick 1999] McKusick, Marshall Kirk. January 1999. ``Twenty Years of Berkeley Unix: From AT&T−Owned to Freely Redistributable." *Open Sources: Voices from the Open Source Revolution*. http://www.oreilly.com/catalog/opensources/book/kirkmck.html.

[McGraw 2000] McGraw, Gary and John Viega. March 1, 2000. Make Your Software Behave: Learning the Basics of Buffer Overflows. http://www−4.ibm.com/software/developer/library/overflows/index.html.

[Miller 1999] Miller, Todd C. and Theo de Raadt. ``strlcpy and strlcat −− Consistent, Safe, String Copy and Concatenation" *Proceedings of Usenix '99*. http://www.usenix.org/events/usenix99/millert.html and http://www.usenix.org/events/usenix99/full_papers/millert/PACKING_LIST

[Mudge 1995] Mudge. October 20, 1995. *How to write Buffer Overflows*. l0pht advisories. http://www.l0pht.com/advisories/bufero.html.

[Open Group 1997] The Open Group. 1997. *Single UNIX Specification, Version 2 (UNIX 98)*. http://www.opengroup.org/online−pubs?DOC=007908799.

[OSI 1999]. Open Source Initiative. 1999. *The Open Source Definition*. http://www.opensource.org/osd.html.

[Pfleeger 1997] Pfleeger, Charles P. 1997. *Security in Computing.* Upper Saddle River, NJ: Prentice−Hall PTR. ISBN 0−13−337486−6.

[Phillips 1995] Phillips, Paul. September 3, 1995. *Safe CGI Programming*. http://www.go2net.com/people/paulp/cgi−security/safe−cgi.txt

[Raymond 1997] Raymond, Eric. 1997. *The Cathedral and the Bazaar*. http://www.tuxedo.org/~esr/writings/cathedral−bazaar

[Raymond 1998] Raymond, Eric. April 1998. *Homesteading the Noosphere*. http://www.tuxedo.org/~esr/writings/homesteading/homesteading.html

[Ranum 1998] Ranum, Marcus J. 1998. *Security−critical coding for programmers − a C and UNIX−centric full−day tutorial*. http://www.clark.net/pub/mjr/pubs/pdf/.

[RFC 822] August 13, 1982 *Standard for the Format of ARPA Internet Text Messages*. IETF RFC 822. http://www.ietf.org/rfc/rfc0822.txt.

[rfp 1999]. rain.forest.puppy. ``Perl CGI problems". *Phrack Magazine*. Issue 55, Article 07. http://www.phrack.com/search.phtml?view&article=p55−7.

[St. Laurent 2000] St. Laurent, Simon. February 2000. *XTech 2000 Conference Reports*. ``When XML Gets Ugly". http://www.xml.com/pub/2000/02/xtech/megginson.html.

[Saltzer 1974] Saltzer, J. July 1974. ``Protection and the Control of Information Sharing in MULTICS". *Communications of the ACM*. v17 n7. pp. 388−402.

[Saltzer 1975] Saltzer, J., and M. Schroeder. September 1975. ``The Protection of Information in Computing Systems". *Proceedings of the IEEE*. v63 n9. pp. 1278−1308.

http://www.mediacity.com/~norm/CapTheory/ProtInf. Summarized in [Pfleeger 1997, 286].

[Schneier 1996] Schneier, Bruce. 1996. *Applied Cryptography, Second Edition: Protocols, Algorithms, and Source Code in C*. New York: John Wiley and Sons. ISBN 0−471−12845−7.

[Schneier 1998] Schneier, Bruce and Mudge. November 1998. *Cryptanalysis of Microsoft's Point−to−Point Tunneling Protocol (PPTP)* Proceedings of the 5th ACM Conference on Communications and Computer Security, ACM Press. http://www.counterpane.com/pptp.html.

[Schneier 1999] Schneier, Bruce. September 15, 1999. ``Open Source and Security''. *Crypto−Gram*. Counterpane Internet Security, Inc. http://www.counterpane.com/crypto−gram−9909.html

[Seifried 1999] Seifried, Kurt. October 9, 1999. *Linux Administrator's Security Guide*. http://www.securityportal.com/lasg.

[Shankland 2000] Shankland, Stephen. ``Linux poses increasing threat to Windows 2000''. CNET. http://news.cnet.com/news/0−1003−200−1549312.html

[Shostack 1999] Shostack, Adam. June 1, 1999. *Security Code Review Guidelines*. http://www.homeport.org/~adam/review.html.

[Sitaker 1999] Sitaker, Kragen. Feb 26, 1999. *How to Find Security Holes*http://www.pobox.com/~kragen/security−holes.html and http://www.dnaco.net/~kragen/security−holes.html

[SSE−CMM 1999] SSE−CMM Project. April 1999. *System Security Engineering Capability Maturity Model (SSE CMM) Model Description Document*. Version 2.0. http://www.sse−cmm.org

[Stein 1999]. Stein, Lincoln D. September 13, 1999. *The World Wide Web Security FAQ*. Version 2.0.1 http://www.w3.org/Security/Faq/www−security−faq.html

[Thompson 1974] Thompson, K. and D.M. Richie. July 1974. ``The UNIX Time−Sharing System''. *Communications of the ACM* Vol. 17, No. 7. pp. 365−375.

[Torvalds 1999] Torvalds, Linus. February 1999. ``The Story of the Linux Kernel''. *Open Sources: Voices from the Open Source Revolution*. Edited by Chris Dibona, Mark Stone, and Sam Ockman. O'Reilly and Associates. ISBN 1565925823. http://www.oreilly.com/catalog/opensources/book/linus.html

[Webber 1999] Webber Technical Services. February 26, 1999. *Writing Secure Web Applications*. http://www.webbertech.com/tips/web−security.html.

[Wood 1985] Wood, Patrick H. and Stephen G. Kochan. 1985. *Unix System Security*. Indianapolis, Indiana: Hayden Books. ISBN 0−8104−6267−2.

[Wreski 1998] Wreski, Dave. August 22, 1998. *Linux Security Administrator's Guide*. Version 0.98. http://www.nic.com/~dave/SecurityAdminGuide/index.html

---

12.Credits

As iron sharpens iron, so one man sharpens another. Proverbs 27:17 (NIV)

My thanks to the following people who kept me honest by sending me emails noting errors, suggesting areas to cover, asking questions, and so on. Where email addresses are included, they've been shrouded by prepending my ``thanks." so bulk emailers won't easily get these addresses; inclusion of people in this list is *not* an authorization to send unsolicited bulk email to them.

- Neil Brown (thanks.neilb@cse.unsw.edu.au)
- Scott Ingram (thanks.scott@silver.jhuapl.edu)
- John Levon (thanks.moz@compsoc.man.ac.uk)
- Ryan McCabe (thanks.odin@numb.org)
- Paul Millar (thanks.paulm@astro.gla.ac.uk)
- Chuck Phillips (thanks.cdp@peakpeak.com)
- Martin Pool (thanks.mbp@humbug.org.au)
- Eric S. Raymond (thanks.esr@snark.thyrsus.com)
- Eric Werme (thanks.werme@alpha.zk3.dec.com)

If you want to be on this list, please send me a constructive suggestion at dwheeler@dwheeler.com. If you send me a constructive suggestion, but do *not* want credit, please let me know that when you send your suggestion, comment, or criticism; normally I expect that people want credit, and I want to give them that credit. My current process is to add contributor names to this list in the document, with more detailed explanation of their comment in the ChangeLog for this document (available on−line). Note that although these people have sent in ideas, the actual text is my own, so don't blame them for any errors that may remain. Instead, please send me another constructive suggestion.

13.Document License

A copy of the text of the edict was to be issued as law in every province and made known to the people of every nationality so they would be ready for that day. Esther 3:14 (NIV)