Secure Programming for Linux HOWTO

Table of Contents

Table of Contents

Secure Programming for Linux HOWTO

David A. Wheeler, dwheeler@dwheeler.com

version 1.30, 9 February 2000

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

Next Previous Contents Next Previous Contents

1. Introduction

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

This paper does not cover assurance measures, software engineering processes, and quality assurance approaches, which are important but widely discussed elsewhere. Such measures include testing, peer review, configuration management, and formal methods. Documents specifically identifying sets of development assurance measures for security issues include the Common Criteria [CC 1999] and the System Security Engineering Capability Maturity Model [SSE−CMM 1999]. More general sets of software engineering methods or processes are defined in documents such as the Software Engineering Institute's Capability Maturity Model for Software (SE−CMM), ISO 9000 (along with ISO 9001 and ISO 9001−3), and ISO 12207.

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

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

You can find the master copy of this document at http://www.dwheeler.com. This document is also part of the Linux Documentation Project (LDP) at http://www.linuxdoc.org (the LDP version may be older than the

master copy).

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

Next Previous ContentsNextPreviousContents

---

10. Conclusions

Designing and implementing a truly secure program is actually a difficult task on Linux. The difficulty is that a truly secure program must respond appropriately to all possible inputs and environments controlled by a potentially hostile user. This is not a problem unique to Linux; other general−purpose operating systems (such as Unix and Windows NT) present developers with similar challenges. 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.

NextPreviousContentsNextPreviousContents

---

11. References

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

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

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

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

[AUSCERT 1996] Australian Computer Emergency Response Team (AUSCERT) and O'Reilly. May 23, 1996 (rev 3C). *A Lab Engineers Check List for Writing Secure Unix Code*.

ftp://ftp.auscert.org.au/pub/auscert/papers/secure_programming_checklist

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[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

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

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

[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.

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

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

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

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

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

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

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

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

[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. Summarized in [Pfleeger 1997, 286].

[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.

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

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

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

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

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

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

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

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

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

---

NextPreviousContents Next PreviousContents

---

12. Document License

```
This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
```

Next PreviousContentsNextPreviousContents

---

2. Background

2.1 Linux and Open Source Software

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 many useful components but was having trouble developing the operating system kernel [FSF 1998]. 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 producing a freely−modifiable and very useful operating system. This paper will term the kernel itself the ``Linux kernel'' and an entire combination as ``Linux'' (many use the term GNU/Linux instead for this combination).

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. This paper is not specific to any distribution; it does presume Linux kernel version 2.2 or greater and the C library glibc 2.1 or greater, which are valid assumptions for essentially all current major Linux distributions.

Increased interest in such ``free software'' has made it increasingly necessary to define and explain it. A widely used term is ``open source software,'' which further defined in [OSI 1999]. Eric Raymond [1997, 1998] wrote several seminal articles examining its development process.

Linux is not derived from Unix source code, but its interfaces are intentionally Unix−like. Therefore, Unix lessons learned apply to Linux, including information on security. Much of the information in this paper actually applies to any Unix−like system, but Linux−specific information has been intentionally added to enable those using Linux to take advantage of its capabilities. This paper intentionally focuses on Linux systems to narrow its scope; including all Unix−like systems would require an analysis of porting issues and other systems' capabilities, which would have greatly increased the size of this document.

Since Linux is intentionally Unix−like, it has Unix security mechanisms. These include user and group ids (uids and gids) for each process, a filesystem with read, write, and execute permissions (for user, group, and other), System V inter−process communication (IPC), socket−based IPC (including network communication), and so on. See Thompson [1974] and Bach [1986] for general information on Unix systems, including their basic security mechanisms. Section 3 summarizes key Linux security mechanisms.

2.2 Security Principles

There are many general security principles which you should be familiar with; consult a general text on computer security such as [Pfleeger 1997].

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

- *Least privilege*. Each user and program should operate using the fewest privileges possible. That way, damage from attack is minimized.
- *Economy of mechanism*. The protection system's design should be small, simple, and straightforward.
- *Open design*. The protection mechanism must not depend on the attacker ignorance. Instead, the mechanism should be public, depending on the secrecy of relatively few (and easily changeable) items like passwords. This makes extensive public scrutiny possible. 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.
- *Permission−based*. The default should be denial of service.
- *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*. Shared objects provide potentially dangerous channels for information flow, so physically or logically separate them.
- *Easy to use*. If a mechanism is easy to use it is unlikely to be avoided.

2.3 Types of Secure Programs

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

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

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

This paper merges the issues of these different types of program into a single set. The disadvantage of this approach is that some of the issues identified here don't apply to all types of program. 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). The advantage of considering all of these program types together is that we can consider all issues without trying to apply an inappropriate category to a program. As will be seen, many of the principles apply to all programs that need to be secured.

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

2.4 Paranoia is a Virtue

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

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

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

2.5 Sources of Design and Implementation Guidelines

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

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

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

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

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

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

- Much of this information was scattered about; placing the critical information in one organized document makes it easier to use.
- Some of this information is not written for the programmer, but is written for an administrator or user.
- Some information isn't relevant to Linux. For example, many checklists warn against setuid shell scripts; since Linux doesn't permit them in the normal case, there's no need to warn against them.
- Much of the available information emphasizes portable constructs (constructs that work on all Unix−like systems). It's often best to avoid Linux−unique abilities for portability's sake, but somethimes the Linux−unique abilities can really aid security. Even if non−Linux portability is desired, you may want to support Linux−unique abilities on Linux.
- This approach isn't unique. Other operating systems, such as FreeBSD, have a security programming guide specific to their operating system.

2.6 Document Conventions

System manual pages are referenced in the format *name(number)*, where *number* is the section number of the manual. C and C++ treat the character '\0' (ASCII 0) specially, and this value is referred to as NIL in this paper. 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, but note that nothing in the C standard requires that NULL actually be implemented by a series of all−zero bits.

3. Summary of Linux Security Features

Before discussing guidelines on how to use Linux security features, it's useful to know what those features are from a programmer's viewpoint. This section briefly describes those features; 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 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 the forest with the trees. 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. 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. Unix programmers will be in familiar territory, but there are several Linux extensions and specifics that may surprise them. This section will try to point out those differences.

First, the basics. Linux is fundamentally divided into two parts: the Linux kernel (along with its kernel modules) and ``user space" in which various programs execute on top of the kernel. 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," who 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. The next few sections detail this.

3.1 Processes

In Linux, user−level activities are implemented by running processes. Many systems support separate ``threads"; in Linux, different threads may be implemented by using multiple processes (the Linux kernel then performs optimizations to get thread−level speeds).

Process Attributes

Every process has a set of security−relevant attributes, including the following:

- 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)
- 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.
- SUID, SGID − Saved UID and GID; used to support switching permissions ``on and off" as discussed below.
- 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.
- 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.
- limits − per−process resource limits (see below).
- filesystem root − the process' idea of where the root filesystem begins; see chroot(2).

If you really need to know exactly what attributes are associated with each process, examine the Linux source code, in particular include/linux/sched.h's definition of task_struct.

POSIX Capabilities

Linux version 2.2 added internal support for ``POSIX Capabilities.'' POSIX capabilities supports some 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. 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.

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

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

Process Creation and Manipulation

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

When a program is executed, and its file has its setuid or setgid bit set, the process' EUID or EGID (respectively) is set to the file's. Note that under Linux this does not occur with ordinary scripts such as shell scripts, because there are a number of security dangers when trying to do this with scripts (some other Unix−like systems do support setuid shell scripts). As a special case, Perl includes a special setup to support setuid Perl scripts.

In some cases a process can affect the various UID and GID values; see setuid(2), seteuid(2), setreuid(2), setfsuid(2). In particular the SUID attribute is there to permit trusted programs to temporarily switch UIDs. 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 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 Filesystem

Filesystem objects (FSOs) may be ordinary files, directories, symbolic links, named pipes (FIFOs), sockets, character special (device) files, or block special (device) files (this list is shown in the find(1) command). Filesystem objects are collected on filesystems, which can be mounted and unmounted on directories in the filesystem; filesystems may have slightly different sets of access control attributes and access controls can be affected by options selected at mount time.

Filesystem Object Attributes

Currently ext2 is the most popular filesystem on Linux systems; it supports the following attributes on each filesystem object:

- 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.
- 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) are limited to root, the file owner, or the directory owner. This is a very common Unix extension, though not quite universal. The sticky bit has no affect on ordinary files and ordinary users can turn on this bit. Old versions of Unix called this the ``save program text'' bit and used this to indicate executable files that should stay in memory; Linux's virtual memory management makes this 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. 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. 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 not universal across Unix−like systems.
- 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.
- 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).

Many of these values can be influenced at mount time, so that, for example, certain bits can be treated as though they had a certain value (regardless of their values on the media). See mount(1) for more information about this. Some filesystems don't support some of these access control values; again, see mount(1) for how these filesystems are handled.

There is ongoing work to add access control lists (ACLs) and POSIX capability values to the filesystem, but these do not exist in stock Linux 2.2.

Creation Time Initial Values

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

Changing Access Control Attributes

You can set most of these values with chmod(2) or chmod(1), but see also chown(1), chgrp(1), and chattr(1).

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

Using Access Control Attributes

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

Filesystem Hierarchy

Over the years conventions have been built on ``what files to place where''; please follow them and use them when placing information in the hierarchy. A summary of this information is in hier(5). More information is available on the Filesystem Hierarchy Standard (FHS), which is an update to the previous Linux Filesystem Structure standard (FSSTND); see http://www.pathname.com/fhs

3.3 System V IPC

Linux supports System V IPC objects, that is, 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. Socket(2) creates an endpoint for communication and returns a descriptor; see socket(2) for more information and cross−references to other relevant information. Note that binding to TCP and UDP local port numbers less than 1024 requires root privilege in Linux (binding to a remote port number less than 1024 requires no special privilege).

3.5 Quotas and Limits

Linux has mechanisms to support filesystem quotas and process resource limits. Be careful with terminology here, because they both 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 by 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().

3.6 Audit

Currently the most common ``audit'' mechanism is syslogd(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.

3.7 PAM

When authenticating, most Linux systems use Pluggable Authentication Modules (PAM). This permits configuration of authentication (e.g., use of passwords, smart cards, etc.). PAM will be discussed more fully later in this document.

NextPreviousContentsNextPreviousContents

---

4. Validate All Input

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

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

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

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

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

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

untrusted user, so you need only worry about those inputs that are.

4.1 Command line

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

4.2 Environment Variables

By default, environment variables are inherited from a process' parent. However, when a program executes another one it can set the environment variables to arbitrary values. This is dangerous to setuid/setgid programs, because their invoker can control their environment variables, sending them on. Since they are usually inherited, this also applies transitively.

Environment variables are stored in a format that allows multiple values with the same field (e.g., two SHELL values). While typical command shells prohibit doing this, a cracker can create such a situation; some programs may test one value but use a different one in this case. Even worse, many libraries and programs are controlled by environment variables in ways that are obscure, subtle, or even 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, setting IFS to unusual values can subvert apparently−safe calls.

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 by setting the global variable *environ* to NULL, followed by resetting a small set of necessary environment variables to safe values (*not* values from the user). These values usually include PATH (the list of directories to search for programs, which should *not* include the current directory), IFS (to its default of `` \t\n"), and TZ (timezone).

4.3 File Descriptors

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

4.4 File Contents

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

4.5 CGI Inputs

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

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

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

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

4.6 Other Inputs

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

4.7 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.

NextPreviousContentsNextPreviousContents

---

5. Avoid Buffer Overflow

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

A buffer overflow occurs when you write a set of values (usually a string of characters) into a fixed length buffer and keep writing past its end. These can occur when reading input from the user into a buffer, but they can also occur during other kinds of processing in a program.

If a secure program permits a buffer overflow, it can usually 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. A buffer in the heap isn't much better; attackers can use this to control 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/.

Some 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 absolutely no protection against such problems, and C++ can be easily used in ways to cause this problem too.

5.1 Dangers in C/C++

C users must avoid using dangerous functions that do not check bounds unless they've ensured the bounds will never get exceeded. Functions to avoid in most cases include the functions strcpy(3), strcat(3), sprintf(3), and gets(3). These should be replaced with functions such as strncpy(3), strncat(3), snprintf(3), and fgets(3) respectively, but see the discussion below. The function strlen(3) should be avoided unless you can ensure that there will be a terminating NIL character to find. Other dangerous functions that may permit buffer overruns (depending on their use) include fscanf(3), scanf(3), vsprintf(3), realpath(3), getopt(3), getpass(3), streadd(3), strecpy(3), and strtrns(3).

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 ``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). 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 performance penalty compared to the strcpy(3) it replaces, because strncpy(3) zero−fills the remainder of

the destination.

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 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 BSD−style license at ftp://ftp.openbsd.org/pub/OpenBSD/src/lib/libc/string/strlcpy.3.

Another alternative is to dynamically reallocate all strings instead of using fixed−size buffers. This general approach is recommended by the GNU programming guidelines. 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. The source code is open source; the documentation is not but it is freely available.

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

5.3 Compilation Solutions in C/C++

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

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

You can read Linus Torvalds' explanation for not including this support at http://lwn.net/980806/a/linus−noexec.html.

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

5.4 Other Languages

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

NextPreviousContentsNextPreviousContents

---

6. Structure Program Internals and Approach

6.1 Secure the Interface

Interfaces should be minimal (simple as possible), narrow (provide only the functions needed), and non−bypassable. Trust should be minimized. Applications and data viewers may be used to display files developed externally, so in general don't allow them to accept programs (including auto−executing macros) unless you're willing to do the extensive work necessary to create a secure sandbox.

6.2 Minimize Permissions

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

In Linux, the primary determiner of a process' permission is the set of id's associated with it: each process has a real, effective, filesystem, and saved id for both the user and group. Manipulating these values is critical to keeping permissions minimized.

Permissions should be minimized along several different views:

- Minimize the highest permission granted. Avoid giving a program root privileges if possible. Don't make a program *setuid root* if it only needs access to a single file; consider creating a special group,

make the file owned by the group, and then make the program *setgid* to that group. Similarly, try to make a program *setgid* instead of *setuid*, since group membership grants fewer rights (in particular, it does not grant the right to change file permissions). If a program must constantly switch between user permissions to access files (e.g., an NFS server), consider setting only the Linux−unique value ``file system uid'' (fsuid) since this will limit file accesses without causing race conditions and without permitting users to send signals to the process.

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. For more information on Linux's implementation of POSIX capabilities, see http://linux.kernel.org/pub/linux/libs/security/linux−privs.

- Minimize the time the permission is active. Use setuid(2), seteuid(2), and related functions to ensure that the program only has these permissions active when necessary.
- Minimize the time the permission can become active. As soon as possible, permanently give up permissions. Since Linux implements ``saved'' IDs, 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!
- Minimize the number of modules needing and granted the permission. If only a few modules are granted the permission, then it's much easier to determine if they're secure. One way to do so follows the previous point; 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; one command is 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 (which, if the input is acceptable, is then passed to the first tool). This is especially helpful for GUI−based systems; have the GUI portion run as a normal user, and then pass those requests on to another module that has the special privileges.
- Minimize the resources available. You can set file and directory permissions so that only a small number of them can be written by the program. This is commonly done for game high scores, where games are usually setgid *games*, the score files are owned by the group *games*, and the programs are owned by someone else (say root). Thus, breaking into a game allows the perpetrator to change high scores but won't allow him to change a game's executable or configuration file.

  Consider creating separate user or group accounts for different functions, so breaking into one system will not automatically allow damage to others.

  You can use the chroot(2) command so that the program has only a limited number of files available to it. This requires carefully setting up a directory (called the ``chroot jail''). A program with root permission can still break out (using calls like mknod(2) to modify system memory), but otherwise such a jail can significantly improve a program's security.

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 permission, 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 permissions. To do this, set up a secure communication channel (usually unnamed pipes are used), then fork into different processes and drop as many permissions as possible. Then use a simple protocol to allow the less trusted processes to request actions from more trusted processes, 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.

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 for filesystem permissions. Programs with root privileges should consider changing just fsuid and fsgid before accessing files on behalf of a normal user. The reason is that setting a process' euid allows the corresponding user to send a signal to that process, while just setting fsuid does not. The disadvantage is that these calls are not portable to other POSIX systems.

6.3 Use Safe Defaults

On installation the program should deny all accesses until the user has had a chance to configure it. Installed files and directories should certainly not be world writable, and in in fact it's best to make them unreadable by all but the trusted user. If there's a configuration language, the default should be to deny access until the user specifically grants it.

6.4 Fail Open

A secure program should always ``fail open,'' that is, it should be designed so that if the program does fail, the program will deny all access (this is also called ``failing safe''). If there seems to be some sort of bad behavior (malformed input, reaching a ``can't get here'' state, and so on), then the program should immediately deny service. 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.

6.5 Avoid Race Conditions

Secure programs must determine if a request should be granted, and if so, act on that request. There must be no way for an untrusted user to change anything used in this determination before the program acts on it.

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

6.6 Trust Only Trustworthy Channels

In general, do not trust results from untrustworthy channels.

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

trustworthy; in most networks such requests can be forged or the platform can be made to permit use of low−numbered ports.

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

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

If asking for a password, try to set up trusted path (e.g., require pressing an unforgeable key before login, or display unforgeable pattern such as flashing LEDs). When handling a password, encrypt it between trusted endpoints.

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

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

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

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

6.7 Use Internal Consistency−Checking Code

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

6.8 Self−limit Resources

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

7. Carefully Call Out to Other Resources

7.1 Limit Call−outs to Valid Values

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

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

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

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

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

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

7.2 Check All System Call Returns

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

NextPreviousContentsNextPreviousContents

8. Send Information Back Judiciously

8.1 Minimize Feedback

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

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

8.2 Handle Full/Unresponsive Output

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

NextPreviousContentsNextPreviousContents

9. Special Topics

9.1 Locking

There are often situations in which a program must ensure that it has exclusive rights to something. On POSIX systems this is traditionally done by creating a file to indicate a lock, because this is portable to many systems.

However, there are several traps to avoid. First, a program with root privileges can open a file, even if it sets the O_EXCL mode (which normally fails if the file already exists). If that can happen, don't use open(2), use

link(2) to create the file. If you just want to be sure that your server doesn't execute more than once on a given machine, consider creating a lockfile as /var/log/NAME.pid with the pid as its contents. This has the disadvantage of hanging around if the program prematurely halts, but it's common practice and is easily handled by other system tools.

Second, if the lock file may be on an NFS−mounted filesystem, then you have the problem that NFS doesn't completely support normal file semantics. 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.

9.2 Passwords

Where possible, don't write code to handle passwords. In particular, if the application is local, try to depend on the normal login authentication by a user. If the application is a CGI script, 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. For networks, consider at least using digest passwords (which are vulnerable to active attack threats but protect against passive network sniffers).

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

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

9.3 Random Numbers

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

9.4 Cryptographic Algorithms and Protocols

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

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

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

9.5 Java

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

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

9.6 PAM

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

9.7 Miscellaneous

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

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.

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. Allow non−root installation where possible as well, so that users without root permission and administrators who do not fully trust the installer to still use the program.

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

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

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

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

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

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

NextPreviousContents