

Tru64 UNIX

Protecting Your System Against File Name Spoofing Attacks

January 2003

Product Version: Tru64 UNIX, Versions 4.0F through 5.1A
and higher

This manual discusses the potential for compromise of system security through manipulation of name space collisions (spoofing), also known as symlink attacks, in world-writable directories. It also explains how to address this problem by using system administration tools and safe coding practices in local scripts and programs.

© 2002 Hewlett-Packard Company

UNIX and The Open Group are trademarks of The Open Group in the U.S. and/or other countries. All other product names mentioned herein may be the trademarks of their respective companies.

Confidential computer software. Valid license from Compaq Computer Corporation, a wholly owned subsidiary of Hewlett-Packard Company, required for possession, use, or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

None of Compaq, HP, or any of their subsidiaries shall be liable for technical or editorial errors or omissions contained herein. The information is provided "as is" without warranty of any kind and is subject to change without notice. The warranties for HP or Compaq products are set forth in the express limited warranty statements accompanying such products. Nothing herein should be construed as constituting an additional warranty.

Contents

1 Problem Description

2 Security Options for the System Administrator

2.1	Using System Attributes	2-1
2.1.1	restricted_hardlink_creat (in sec)	2-1
2.1.2	restricted_symlink_follow (in sec)	2-2
2.1.3	restricted_fifo_open (in sec)	2-2
2.1.4	dump_cores (in proc)	2-3
2.1.5	dump_setugid_cores (in proc)	2-3
2.1.6	follow_mkdir_symlinks (in vfs)	2-4
2.2	Using the dirclean Utility	2-4
2.3	Using the ckfsec Utility	2-4
2.4	Using Standard Security Measures	2-5

3 Options for Scripts and Programs

3.1	General Rules for Handling of Temporary Files by Privileged Processes	3-1
3.2	How to Make Script Behavior More Secure	3-3
3.2.1	Using the /usr/bin/mktemp Utility	3-3
3.2.2	Using noclobber Mode in the Shell	3-4
3.2.3	Using a Private Application Directory for Temporary Files	3-6
3.2.4	Using Here Document (<<) Processing in Shells	3-6
3.2.5	Script Examples	3-7
3.2.5.1	Script that Creates Two Temporary Files	3-8
3.2.5.2	Script that Creates a Secure, Temporary Directory	3-9
3.2.5.3	Script Using Multiple Temporary Files Several Times	3-10
3.2.5.4	C Shell (csh) Script	3-12
3.3	How to Make Program Behavior More Secure	3-13
3.3.1	Recommended: Use mkstemp(), mkstemp(), and mkdtemp()	3-13
3.3.2	Recommended: Use safe_open()	3-14
3.3.3	Recommended: Control core File Location with SSI_COREDIR	3-15
3.3.4	Not Recommended: fopen() in World-Writable Directories	3-16

3.3.5	Not Recommended: Certain libc Routines in World-Writable Directories	3-16
3.3.6	Handling of Symbolic Links by mkdir()	3-17

A Reference Pages

Index

Problem Description

This manual addresses a type of security vulnerability that is exploited by file name spoofing (a special case of name space collision). When this type of vulnerability exists, symbolic links, hard links, and FIFOs can all be used in conjunction with the normal operation of privileged programs and scripts to mount an attack against the system. Users who change directory to `/tmp` or `/usr/tmp` before running programs or scripts (a common practice after logging on to a system to which one's home directory is not mounted) add to the security risk.

In the simplest case, privileged programs or scripts that create files in world-writable directories can overwrite protected system files. Any user who can predict the name of a file that will be created by a privileged program or script can create a symbolic link (with the same name as the file used by the program) to point to a protected system file. Unless the privileged program or script is coded defensively (or the system is set up to protect against any that is not coded defensively), the program or script will follow the symbolic link instead of opening or creating the file that it is supposed to be using. As a result, a protected system file to which the symbolic link points can be overwritten when the script or program is run. This type of security breach is commonly known as a “symlink attack;” however, hard links and FIFOs have also been used to spoof file names.

File name spoofing usually targets temporary files created by a privileged program or script, because temporary files traditionally reside in the world-writable `/tmp` and `/var/tmp` directories to which nonprivileged users also have read and write access. However, spoofing attacks sometimes target the names of commonly-used commands to trick the script or program into performing an operation other than the one intended. Spoofing attacks typically aim to create denial-of-service problems by means of system file deletion or corruption. However, sometimes the objective of the attack is to obtain root access.

The following coding practices are traditionally used to safeguard against file name spoofing but provide insufficient protection:

- Randomizing temporary file names

Randomizing the names of temporary files does make the names harder for the attacker to guess. However, by itself, this practice is not enough to protect the system. For example, if the privileged program relies on

appending its PID to randomize a temporary file name, the attacker only needs to determine that PID. While the attacker may not be able to accurately predict the exact PID, knowing that the script is run at boot time or from a cron job or within a day can restrict the range of PIDs that need to be tried to fewer than one thousand. It is relatively easy for an attacker to simply fill the directory name space by creating thousands of symbolic links using likely PID numbers at some point before the privileged program creates its temporary files.

- Deleting a file before creating it or invoking `stat()` on a file before opening it

Deleting a file before creating it or calling `stat()` on the file before opening it does not protect against a file name spoofing attack because the operations are not atomic. Separate file accesses always open a timing window over which the program has limited control. Within this window, the program's process might be rescheduled, a process running on another CPU might alter the file system, and even a process on another cluster member might intervene. The only safe method to open a file is to use exclusive open with owner access only. No matter how small a programmer thinks a timing window is likely to be, there is always the possibility that someone will exploit it.

There are two areas in which actions can be taken to protect the system against a file name spoofing attack: system administration and the coding practices used when writing local programs and shell scripts. The following chapters discuss actions you can take in these two areas. Although the concepts discussed in this document apply to any system, some of the recommended interfaces are not governed by industry standards; therefore, you cannot assume that specific recommendations are portable to other platforms.

2

Security Options for the System Administrator

Through patches to releases of the operating system through Version 5.1A and in unpatched releases following Version 5.1A, several system administration options are available to address the security vulnerabilities discussed in this manual. Included in these releases are the following:

- New kernel subsystem attributes whose settings help protect a system that must run privileged programs and scripts whose coding does not adequately defend against file name spoofing attacks (Section 2.1).

Chapter 3 discusses how locally written privileged scripts and programs can be coded to safely use temporary files in world-writable locations.

- A tool that system administrators can use to safely find and delete files in world-writable directories (Section 2.2).
- A tool that allows system administrators to monitor file system accesses to specified directories and detect creation of symbolic links, hard links, and FIFOs (Section 2.3).

2.1 Using System Attributes

Three tunable system attributes are available through the `sec` kernel subsystem to allow system administration control over the behavior of `link()`, `rename()`, and `open()` system calls on detection of links. Three additional tunable attributes in other subsystems control whether a privileged program can create core dumps and whether `mkdir()` follows a final symbolic link.

2.1.1 `restricted_hardlink_creat` (in `sec`)

The `restricted_hardlink_creat` attribute affects behavior of the `link()` and `rename()` functions. When enabled (set to 1), this attribute:

- Prevents hard link creation by causing `link()` to fail if all of the following conditions are true:
 - The caller is not privileged.
 - The hard link is to be created in a world-writable directory.

- The current user is not the owner of the directory where the hard link is to be created.
- The current user is not the owner of the file object underlying the link.
- Prevents moving a hard link into a world-writable directory by causing `rename()` to fail if all the following conditions are true:
 - The *from* parameter is not a directory.
 - The *from* parameter is not a symbolic link.
 - The link count for *from* is greater than 1.
 - The current user is not privileged.
 - The *to* parameter specifies a world-writable directory.
 - The current user does not own the parent directory of the *to* parameter.

The failure conditions described for `link()` and `rename()` cause [EACCES] to be returned to the caller.

It is recommended that the `restricted_hardlink_creat` be set to 1 (enabled). By default, this attribute is disabled (0).

2.1.2 restricted_symlink_follow (in sec)

The `restricted_symlink_follow` affects the behavior of the `open()` system call.

If enabled (set to 1), the attribute prevents `open()` from following a symbolic link if all the following conditions are true:

- The directory that contains the symbolic link is world-writable.
- The owner of the symbolic link is not root.
- The owner of the symbolic link is not the current user.
- The symbolic link and the directory that contains the symbolic link do not have the same owner.

If these conditions are true, the `open()` system call fails and returns [EACCES] to the caller.

It is recommended that the `restricted_symlink_follow` attribute be enabled (set to 1). By default, this attribute is disabled (0).

2.1.3 restricted_fifo_open (in sec)

The `restricted_fifo_open` attribute affects the behavior of the `open()` call when it opens a fifo.

When enabled (set to 1), `restricted_fifo_open` prevents an `open()` call from opening a fifo if all the following conditions are true:

- The parent directory is world writable.
- The current user is not the fifo owner.
- The fifo owner is not the owner of parent directory.
- The fifo owner is not root.

If these conditions are true, the `open()` call fails, returning `[EACCES]` to the caller.

It is recommended that the `restricted_fifo_open` attribute be enabled (set to 1). By default, this attribute is disabled (0).

2.1.4 `dump_cores` (in `proc`)

The `dump_cores` attribute in the `proc` subsystem is available to control whether user processes are allowed to create core dumps. If disabled (0), this attribute can prevent some denial-of-service attacks that are possible if the `core` file and the directory where it is written are inadequately protected and a privileged user is running the program. Under these conditions, for example, the `/tmp/core` pathname can be spoofed so that the `core` file is created as `/etc/nologin` and prevent users from logging on the system.

By default, many applications dump `core` files to the directory from which the user invoked the program. Note that programs can control the location of `core` files by using the `setsysinfo()` call with an `SSI_COREDIR` request. See Section 3.3.3 for more information.

The `dump_cores` attribute is enabled (1) by default, which allows user applications to dump core. Because `core` files provide important debugging information to application maintainers, it is recommended that this attribute be disabled only if the system is subject to denial-of-service attacks.

2.1.5 `dump_setuid_cores` (in `proc`)

The `dump_setuid_cores` attribute in the `proc` subsystem is available to control `core` dump behavior for privileged processes (specifically, processes of programs running in `setuid/setgid` mode). `Core` dumps from such processes can contain sensitive information that might be used to compromise system security.

By default, this attribute is disabled (0), meaning that privileged processes are not allowed to create `core` dumps. When the `dump_setuid_cores` attribute is enabled (set to 1), privileged processes can create `core` dumps.

It is recommended that this attribute be enabled (set to 1) only for purposes of application debugging and then returned to 0 (the default) after the application is debugged.

2.1.6 follow_mkdir_symlinks (in vfs)

Whether a final symbolic link is followed by the `mkdir()` system call (and, by association, the `mkdir` utility) is controlled by the `follow_mkdir_symlinks` attribute in the `vfs` subsystem. The disabled (0) setting of this attribute (recommended and also the default) addresses a bug that was introduced in Tru64 UNIX Version 5.0. A software patch is available for Version 5 releases (through Version 5.1A) to prevent `mkdir()` from creating the specified directory when the final element of the path already exists as a symbolic link. This patch restores the way the function behaved in releases prior to Version 5.0.

The `follow_mkdir_symlinks` attribute, also included in the patch and in any unpatched releases following Version 5.1A, is a contingency measure to use with any local applications that might have been developed to depend on the incorrect `mkdir()` behavior. (The attribute can be changed to 1 if it is important to keep those applications running until they can be recoded.) However, the attribute should be considered temporary as it will likely be retired in a future release.

See Section 3.3.6 for information about explicit control of symbolic link following in a program and script.

2.2 Using the `dirclean` Utility

The `/usr/sbin/dirclean` utility was introduced to provide privileged users, scripts, and programs a safe way to find and delete files in world-writable directories. The commonly used `find` and `rm` combinations create a timing window that can be exploited by file name spoofing attacks.

The `/usr/sbin/dirclean` utility is now invoked from the default root `crontab` entry and from the `rmtmpfiles` script.

See `dirclean(8)` for details about this utility.

2.3 Using the `ckfsec` Utility

The `/usr/bin/ckfsec` utility checks one or more directory trees for directories and files that make the system vulnerable to file name spoofing. More specifically, the utility checks for world-writable directories for which the sticky bit is not set and for file system objects (symbolic links, hardlinks, and pipes) that might be used to compromise system security.

See `ckfsec(1)` for information on using this utility.

2.4 Using Standard Security Measures

The software features discussed in this document should be used only as a supplement to the more standard security measures that protect the system from unauthorized file access. Requiring all world-writable directories to have the sticky bit set is important. Although the sticky bit does not prevent privileged programs from following links they didn't create, it does prevent users from deleting or renaming a file created by someone else. Careful tracking and monitoring of user accounts, and requiring users to replace passwords on a regular basis reduces the risk that an unauthorized user can mount an attack against the system.

For complete coverage of Tru64 UNIX security options, see the *Security* or the *Security Administration* guide (title varies, depending on the Tru64 UNIX version that is installed on your system).

Options for Scripts and Programs

This chapter describes coding that allows scripts and programs to protect the system against attacks that rely on weaknesses in the way that privileged processes use the filesystem name space, particularly in world-writable directories.

Using one of the following interfaces is the best way to create and use temporary files in a secure manner:

- For C source programs, use the `mkstemp()` or `mkdtemp()` routine. The `safe_open()` routine is an alternative for programs that must write to a file whose name cannot be randomized because the user or another program expects the file to have a particular name.
- For shell scripts, use the `/usr/bin/mktemp` utility.

Although additional options are described in this chapter, these are the ones most highly recommended.

3.1 General Rules for Handling of Temporary Files by Privileged Processes

Safe, privileged code conforms to the following rules:

Open/create a file so that the open will fail on a symbolic link name collision rather than overwrite the link's target.

Otherwise, no matter what file is intended to be opened, the code can be spoofed. It is not safe to open a file through a symbolic link when the link's target does not yet exist.

In addition to creating the file for exclusive access, randomize the file's name to prevent simple denial-of-service attacks

If an attacker knows that a daemon always creates a consistently named log file in the `/tmp` directory on start up and is coded to fail rather than overwrite an existing file, the attacker can prevent the daemon from starting by creating the file before the daemon does. Randomizing file names is a good idea anyway, for protection in case more than one process runs your code. If you have to produce a file with a predictable name so that a human can find it, it is not a good idea to place that file in a world-writable directory.

Set file protection correctly and atomically.

In a script, use `umask` before opening the file rather than opening the file and then using `chmod` (which requires a second file access).

In program source code, use the mode argument in the `open()` call rather than opening the file and then setting its mode in a second file access. Your code cannot assume that the mode setting at the time the file is opened will prevent damage and, depending on the initial access rights, unexpected garbage might be appended to the file between the time it is opened and the time its mode is adjusted.

For both scripts and programs, a file protection of 0600 (for directories, 0700) is best. A possible exception is when there is a requirement on the application to write to a file that resides in a particular place and is made available for humans to read.

Guarantee that the command invoked is the one expected.

This is a basic principle in defensive programming. In scripts, either define and use `PATH` for utility pathname prefixes or refer by absolute pathname to all utilities that you invoke (for example, `/usr/bin/rm` rather than `rm`).

Don't trust anything read from a file in a world-writable directory.

Unless you verify that the file ownership and permissions would have prevented unauthorized changes, you cannot rely on the content of a file in a world-writable directory, even if the file name is one that you expect. Leaving a file in a world-writable directory for later access by another script, program, or human creates a security vulnerability.

Using the `/usr/bin/mktemp` utility or the `mkstemp()` routine is the best way to ensure that your privileged script or program conforms to these rules.

Always handle file creation or open failures.

It is a fairly common practice to create a program or script that, in response to an unexpected error when opening a file, either continues to run or simply exits (without cleaning up any files that might have been created up to that point). file name spoofing attacks can saturate a world-writable directory with file system objects, such that the directory or entire file system is no longer writable. Therefore, it is always important to handle open or write failures as is appropriate, given the state that the system is in at the time of failure. To the extent possible, always attempt to leave the system in a consistent state and, before exiting, clean up any temporary directories or files that might have been created.

3.2 How to Make Script Behavior More Secure

This section explains how to find and fix script encoding that increases system vulnerability to a file name spoofing attack.

To change a script so that it handles temporary files securely, you need to look at all references to `tmp`, `TMP`, `TEMP`, and `temp` and adjust the coding to conform to the rules in Section 3.1. Wherever possible, invoke the `mktemp` utility to create temporary files. More information about using this utility is provided in Section 3.2.1. Using `mktemp` ensures that the script creates temporary files securely; however, there are other vulnerabilities to consider. You should also take the opportunity to:

- Use an absolute pathname when accessing any file, not just a temporary one. For example, set the `PATH` variable to ensure that the intended command is executed rather than another command that has the same name but resides in a different directory.
- Use `umask` rather than `chmod` to set file permissions. The following example shows how to set permissions for open operations and then, if necessary, to restore the original `umask` after the files are opened:

```
SAVEMASK=`umask`  
umask 077  
{ open files }  
umask $SAVEMASK
```

- Examine all exit paths, under both normal and error conditions, to make sure that the script cleans up all its temporary directories and files.

Before you start to change a script, you should also consider the following options:

- Turning on the shell's `noclobber` option as a debugging aid (to help identify the script's assumptions about files).

This might be difficult to retrofit into existing scripts. See Section 3.2.2 for more information.

- Creating temporary files in a nonstandard but protected area. The drawback to this option is that your script might not be as portable to other platforms. If portability is not an issue, you can use a private application area that already exists. See Section 3.2.3 for more information.

3.2.1 Using the `/usr/bin/mktemp` Utility

Under almost all circumstances, `/usr/bin/mktemp` will be the fastest and easiest way to generate a temporary file or directory from a script. This utility creates files in `/tmp` or any specified directory using the correct open flags, permissions, and randomized naming. The utility can also create

a private directory for temporary files, which is often the easiest way to retrofit `mktemp` into an existing script. Here are some examples:

1. In the following example, the absolute path to the file is explicitly specified and the script exits if the file cannot be created:

```
TMPFILE='/usr/bin/mktemp /tmp/example.XXXXXXXXXX' || exit 1
echo program output >> $TMPFILE
```

2. In the following example, the value of `TMPDIR` (or `/tmp` if `TMPDIR` is not defined) is prepended to the specified file name, and the script exits if the file cannot be created:

```
TMPFILE='/usr/bin/mktemp -t example.XXXXXXXXXX' || exit 1
echo program output >> $TMPFILE
```

3. In the following example the script creates a directory subordinate to `/tmp` and exits if that directory cannot be created. If the directory was created successfully, the script then creates a file in the new directory:

```
TMPD='/usr/bin/mktemp -d /tmp/exempl.XXXXXX' || exit 1
TMPF='/usr/bin/mktemp -p $TMPD exempl.XXXXXX' || exit 1
echo program output >> $TMPF
```

4. The following example is similar to example 2 in terms of file creation except that the script catches the error:

```
TMP1='/usr/bin/mktemp -t example.1.XXXXXXXXXX' || exit 1
TMP2='/usr/bin/mktemp -t example.2.XXXXXXXXXX'
if [ $? -ne 0 ]; then
    /usr/bin/rm -f $TMP1
    exit 1
fi
```

Note that in all these examples, the absolute path to the directory where temporary files are created is either explicitly specified in the command or the utility creates the path by prepending the value of the `TMPDIR` environment variable (or `/tmp` if `TMPDIR` is undefined) to the path specified in the command.

To create a directory, you must use the `-d` option in a separate command. You cannot create a new directory and a file to go in that directory in a single `mktemp` command. See `mktemp(1)` for more information.

3.2.2 Using noclobber Mode in the Shell

Turning on noclobber mode is one way to minimize a script's vulnerability to file name spoofing; however, it is one of the more difficult methods to use in a complex script.

Noclobber mode affects the behavior of the shell redirection operators (`>` and `>>`). In clobber mode (the default), both redirection operators will follow a specified symbolic link to the file referenced by the link and, if that file does not exist, create the file. This behavior can be exploited by

attacks on system security when files are being created or appended to in world-writable directories. In noclobber mode, the shell returns an error rather than overwriting the specified file or, in some cases, following the symbolic link at all.

The Bourne shell, `/usr/bin/sh`, which did not have noclobber support, has it (in the form of the `-C` option) in releases following Tru64 UNIX Version 5.1A or through a patch to Tru64 UNIX Version 5.1A and prior releases. For the other shells (POSIX/Korn and C shells), which already had noclobber support, the behavior of the redirection operators has been modified to handle symbolic links in a safe manner when noclobber mode is enabled.

- For the Bourne shell in noclobber mode:

If the `>` (create) or `>>` (append) operator redirects output to an existing file, the shell returns an error. On redirection to a symbolic link, the shell also returns an error, whether or not the file referred to by the symbolic link exists. Using the `>|` or `>>|` construct suppresses this check and allows the file or symbolic link target to be created (if it does not exist) or appended to (if it does exist).

- For the POSIX/Korn and C shells in noclobber mode:

If the `>` (create) operator redirects output to a name that identifies an existing symbolic link, the shell returns an error rather than following the symbolic link. This means that, in noclobber mode, you cannot use `>` with the name of a symbolic link to create the link's target. If the `>>` (append) operator redirects output to a name that identifies an existing symbolic link, the shell follows the symbolic link and returns an error only if the target file referenced by the link does not exist. This means that you cannot use `>>` to create a file through a symbolic link but you can use it to append to an existing file through a symbolic link.

Prior to this change, in the case where output was redirected to a symbolic link, both redirection operators in the POSIX/Korn and C shells would always follow the symbolic link and apply conditions in terms of the link's target. This is still true for clobber mode (the default); however, noclobber mode (which is intended to be a more secure mode of operation) now handles symbolic links in a more restrictive manner.

Keep in mind that changing a script to run in noclobber mode means that any file name collision causes a failure, thereby creating a condition that a denial-of-service attack can take advantage of. To guard against this possibility, each attempt to create a file should be preceded by a `rm` operation on the file name. Randomizing the name of the file to be created is another technique that guards against a denial-of-service attack.

Retrofitting noclobber support into an existing script is much more complicated than changing a script to use the `mktemp` utility. Changing an existing script, particularly a complex one, to work in noclobber mode is not

recommended because it introduces new failure conditions that are easy to overlook. If you simply enable `noclobber`, you must review the entire script to verify accesses to all files, not just temporary files, and account for file creation failures because of name collision in all these instances. However, the `noclobber` setting is useful as a debugging tool if you have the time, even if you do not intend to keep `noclobber` enabled once the script is debugged.

3.2.3 Using a Private Application Directory for Temporary Files

If you already have a protected application-specific or subsystem-specific directory, you can use that instead of `/tmp` for your application files. There are two requirements for this directory to be appropriate for use:

- Only root and the subsystem in question can write to the directory. This ensures that no one else can create symbolic links there.
- If the script will be run on diskless systems, the directory cannot reside in the `/usr` file system, which is writable only during system installations on diskless systems. Remember that only the root (`/`) and `/var` file hierarchies are guaranteed to be writable and, of these, only the root hierarchy is available for use in single-user mode.

If you decide to use a private application directory for temporary files, be sure to clean it up on exit, especially if your script operates in `noclobber` mode. Most systems have procedures in place to periodically clean out the standard locations for temporary files, but these procedures do not operate on private directories.

3.2.4 Using Here Document (`<<`) Processing in Shells

A command line of the form `command << eof_string` causes a shell to create a temporary file called a “here document.” All shells now implement the following changes when processing these temporary files. (A patch is required for Tru64 UNIX Version 5.1A and prior releases.)

- Permissions on the temporary file grant read and write permission only to its owner.
- Existing files are never overwritten on a name collision, and if the collision occurs on the name of a symbolic link, the link is not followed. Instead, the shells try file creation with another random name. If all attempts fail, they return the error “Unable to create temporary file.”
- More randomness is available for generating temporary file names. The larger name space reduces the likelihood of name collisions.

These changes are mentioned here mostly to make you aware that here-document processing is now done in a safe manner. However, the “Unable to create temporary file” diagnostic can now be displayed for a

condition under which it was not displayed by previous versions of the shells. This would be a rare condition, however, and would usually indicate active file name spoofing aggression on the system where the script is run.

3.2.5 Script Examples

It is inherent to scripts that there will always be race conditions (operations trying to occur before other operations) and possible failure modes. This is primarily because not all operations can be made atomic. However, failure modes can be made safe rather than insecure.

The examples demonstrate how to secure temporary files that are created in world writable directories which have the sticky bit set (/tmp). If you create files in other world writable directories, you might have to make additional changes to your scripts.

As an alternative, consider modifying your script so it does not use a temporary file or create the temporary file in a private directory rather than in a world writable directory.

In addition to modifying your scripts, you need to be aware of the following warnings:

- Do not try to delete a temporary file twice, rather clear the variable that holds the temporary file name. By clearing the variable, the trap (or other coding) does not try to delete the file again, after you have deleted it. Alternatively, you can reset the trap so that a trap does not occur.

If you delete a temporary file twice, you might delete someone else's file that was created simultaneously to your deleting your temporary file the first time.

- Ensure that the temporary files are deleted, even if the script exits unexpectedly.

You might have to set up a trap that catches any signals that might cause the script to exit unexpectedly as well as those signals that cause the script to exit normally.

The different shells handle signals differently. The signal values are listed in /usr/include/signal.h . Consider trapping the following signals: 0 (exit), 1 (HUP), 2 (INT also known as control-C), 3 (quit), 13 (PIPE) and 15 (TERM).

The following sample scripts show several methods of properly handling temporary files. You might need to combine or change these scripts, depending on what you want the script to do. The examples are written for Bourne shell (sh), Korn shell (ksh) and C shell (csh). You might have to modify them to work with other shells. Document your script to explain what you are doing and why.

3.2.5.1 Script that Creates Two Temporary Files

This script creates two temporary files and uses `trap` to delete the files automatically on exit. When the trap contains 0 (exit), you do not have to delete the temporary files anywhere else in the script.

```
#!/usr/bin/sh

# Secure the path
#
# Secure scripts should not depend on PATH being set correctly by the
# user. The path should be set to a known path at the beginning of
# the script. The exception to this is a script that is only called
# from another script that exports PATH. Set PATH now so you can be
# sure of the results that you get when the commands execute.
# Your script may need additional entries in the path,
# for example, entries that are command or subsystem specific.

PATH=/usr/sbin:/sbin:/usr/bin
Export PATH

# Set up variables to hold the temporary file names
# Initialize the variables that will hold the temporary file names
# to the null string at this time. This helps secure
# the script in two ways: 1) You won't accidentally use a value
# that was set up before the script was called. 2) You won't
# accidentally delete an existing file that happens to have your
# temporary file template name.
#
tmp1=""
tmp2=""

# Trap on exit to delete temporary files
#
# Set up the trap to delete the temporary files BEFORE you create
# the temporary files. This reduces the possibility that the files
# might not be deleted if the script exits unexpectedly between
# the time that the files are created and the time when the trap command is executed.
# As long as you have cleared the variables to hold the
# temporary file names (see above), you won't accidentally delete
# the wrong files. Having 0 (trap on normal exit) in a trap that
# calls exit can cause the trap to trigger twice.
# Clear the trap on normal exit or clear the contents of
# the temporary file variables in the trap. Include any signals
# that might cause the script to exit.

trap "rm -f $tmp1 $tmp2; trap 0; exit" 0 1 2 3 15

# Create temporary files
#
# After mktemp executes, tmp1 will either contain the name
# of a successfully created temporary file or the null string.
# To reduce the possibility of deleting a file that
# has not been created, if the script exits unexpectedly,
# do mktemp and set in one command.
tmp1=`/usr/bin/mktemp /tmp/myfile1.XXXXXXX`
if [ $? -ne 0 ]
then
# Output any additional error message and/or do any
# additional cleanup that your script requires.
exit
fi
tmp2=`/usr/bin/mktemp /tmp/myfile2.XXXXXXX`
```

```

if [ $? -ne 0 ]
then
#   Output any additional error message and/or do any
#   additional clean up that your script requires.
    exit
fi

# Do whatever work is necessary on the temporary files.

# Successful exit
#
# Do not delete the temporary files within the script. In this
# example script, the trap will be called on exit.
# The trap will lose any value passed as a parameter to exit.
# If the exit value is important, set a variable to pass that
# value to the trap for it to set when it exits.
# Refer to example 3.
exit

```

3.2.5.2 Script that Creates a Secure, Temporary Directory

The advantage of this Korn shell script (ksh) is that you can create many temporary files, and you do not have to keep track of them to delete them. Simply remove the temporary directory and its contents on exit.

```

#!/bin/ksh -p

# Secure the path
#
# If this script calls other scripts, make sure to export
# PATH so that the other scripts are also protected.
PATH=/sbin:/usr/sbin:/usr/bin:/usr/ccs/bin:/usr/bin/X11
export PATH

# Set up variables to hold the temporary file names
tmpdir=""

# A cleanup function can be used to ensure that all
# exits from the script do the proper cleanup, including
# deleting the temporary directory. Ensure that the cleanup
# function is called before each exit. If the cleanup
# function does the exit, pass the value for the exit
# parameter to the function.
function cleanup
{
    # Remove temp files
    rm -rf ${tmpdir}

    # Do any other cleanup that is necessary

    return 0
} # end of cleanup

# You can do multiple traps for different signals
trap 'echo "Received cntl-C"; cleanup; exit 0' INT
trap 'echo "Received interrupt"; cleanup; exit 1' HUP TERM QUIT

# Create a directory in /tmp that could be used
# to create temporary files. Exit if the script
# fails to create a temp directory.

```

```

tmpdir=`usr/bin/mktemp -d /tmp/mydirXXXXXX` || {
    echo 1>&2 "Could not create temporary directory, aborting"
    cleanup
    exit 2
}

# You do not need to use mktemp for each temporary file,
# if you are creating it in the secure, temporary directory.
echo "this is my first temporary file" > ${tmpdir}/tmp1
echo "this is my second temporary file" > ${tmpdir}/tmp2

if [something unexpected happens]
then
    echo "create an unexpected file" > ${tmpdir}/unexpected
    # Do any other necessary work.
    cleanup
    exit 42
fi

# Other processing, create as many temporary files as you want!
# Ensure that they are in your temporary directory.

# Success! Clean up and exit

#
# The script does not trap on exit. Specifically call the
# cleanup function before every call to exit in the script.
#
cleanup
exit 0

```

3.2.5.3 Script Using Multiple Temporary Files Several Times

The following script uses multiple temporary files several times.

```

#!/bin/sh

# If you can not define PATH in the script, make sure to use
# the full path for ANY command that you use! E.g.:
AWK=/bin/awk
CAT=/bin/cat
CHMOD=/bin/chmod
ECHO=/bin/echo
GREP=/bin/grep
MKTEMP=/usr/bin/mktemp
RM=/bin/rm
SED=/bin/sed

# When defining an exit status, default to an error status
status=1

# Set up variables to hold the temporary file names. Clear
# the variables, even those that are used within only one
# function, at the start. Then you do not have to worry
# about it if the script changes.
TMPA=
TMPB=
TMPC=
TMPD=

# Another way to define what needs to be done on exit
QUIT='

```

```

if [ -r "$TMPA" ]
then
    RM -f $TMPA
fi
if [ -r "$TMPB" ]
then
    RM -f $TMPB
fi
if [ -r "$TMPC" ]
then
    RM -f $TMPC
fi
if [ -r "$TMPD" ]
then
    RM -f $TMPD
fi
if [ $status -ne 0 ]
then
    $ECHO "script terminated."
else
    $ECHO "Successful completion."
fi
exit $status
,

trap 'status=3; eval "$QUIT"' 1 2 3 15

#
TMPA=`$MKTEMP /tmp/myA.XXXXXX 2> /dev/null` &&
TMPB=`$MKTEMP /tmp/myB.XXXXXX 2> /dev/null` &&
TMPC=`$MKTEMP /tmp/myC.XXXXXX 2> /dev/null` || {
    $ECHO "Unable to create a temporary file."
    status=2
    eval "$QUIT"
}

# This function is not called very often; therefore, do not
# create the temporary file outside the function.
# Create and clean up the temporary file within the function.
one ()
{
    TMPD=`$MKTEMP /tmp/myD.XXXXXX`
    if [ $? -ne 0 ]
    then
        status=4
        eval "$QUIT"
    fi

    # use TMPD

    # To make sure that you never delete someone else's
    # file, clear the variable before deleting
    # the temporary file. This leaves a window
    # where the file might not be deleted (if the script
    # is aborted after the variable is cleared and before
    # the file is deleted). This is a more benign case
    # than the double delete case. In this example, it
    # would be best to let QUIT delete the TMPD file instead
    # of doing it here. This example shows the
    # right way to delete the file, if you have to do it in line.
    tmpx=$TMPD
    TMPD=""
    RM -f $tmpx
    return 0
}

```

```

}

# The use of temp files in functions two and three is totally
# contained within the functions. You can use the same
# temp files for both if you overwrite rather than delete the
# files. As long as the files exist, no one can delete
# or overwrite them.
two ()
{
    $ECHO "two overwrites file TMPA" > TMPA
    return 0
}
three ()
{
    $ECHO "three overwrites file TMPA, too" > TMPA
    $ECHO "but never, ever deletes it!" >> TMPA

    if [function three decides that the script is done]
    then
        status=0
        $ECHO "exit from within function three"
        eval "$QUIT"
    fi
    return 0
}

# main script
if [function one should be executed]
then
    one
fi

two
three

# Do clean up.
status=0
eval "$QUIT"

```

3.2.5.4 C Shell (csh) Script

The C shell script appears to be different, but the mechanics are the same.

```

#!/bin/csh -fb

# Secure the path
set PATH = (/usr/bin /usr/sbin /sbin)

# Set up variables to hold the temporary file names
set TMP=""

# Trap on exit to delete temporary files
# onintr catches the signals.
onintr catch_sig

# Create temporary files
set TMP = `/usr/bin/mktemp /tmp/Init.XXXXXX`
# csh has an internal status variable!
if ($status != 0) exit 1

# Use temporary files. Before any exit in the body of
# the script, make sure to delete the temporary files.

```

```
# normal exit
/usr/bin/rm -f $TMP
exit 0

# Catching the signals
catch_sig:
/usr/bin/rm -f $TMP
exit 1
```

3.3 How to Make Program Behavior More Secure

This section explains how to find and fix C program encoding that increases system vulnerability to a file name spoofing attack. In C source files, the most highly recommended option for creating temporary files is the C library's `mkstemp()` routine, which is discussed in Section 3.3.1.

After discussion of various program coding practices, there is information about other changes to the system infrastructure that can affect program run-time behavior.

For information about coding scripts, see Section 3.2.

3.3.1 Recommended: Use `mkstemp()`, `mkstemp()`, and `mkdtemp()`

The easiest way to safely create a file in a world-writable directory is to call `mkstemp()`. For example:

```
tmpfd = mkstemp(tmp_filename);
```

This routine automatically generates random names (a greater range of these in the patched version of the routine), performs an exclusive open with owner access only, and retries names until they succeed or exhaust the name space.

Trying to replicate `mkstemp()` by creating the file with an exclusive open and, on failure, retrying additional random names is not recommended. The call `open(path, O_CREAT|O_EXCL)` always fails if the target file exists, whether it is a symbolic link or a regular file.

The `mkstemp()` function is included in the XSH specification, starting with XSH4.2, and is therefore portable to platforms that conform to this standard.

The `mkdtemp()` function is available for safely creating a temporary directory in a world-writable area, and the `mkstemp()` function is available for ease in porting code from other platforms, such as Linux. Both of these routines are available through a patch to Tru64 UNIX Versions 4.0F to 5.1A and are included in any unpatched releases following Tru64 UNIX Version 5.1A.

See `mktemp(3)` for more information about these routines.

Although the `mkstemp()`, `mkstemp()`, and `mkdtemp()` routines provide the easiest-to-use protection, and protect against hard link and fifo, as well as symlink, attacks, they are not appropriate for all cases, such as one where you must use a particular name for your temporary file. In this case, `safe_open()` is the recommended option.

3.3.2 Recommended: Use `safe_open()`

Typically, programs try to protect against a symlink attack by using `access()`, `stat()`, `lstat()` or `remove()` before `open()`. However, this call sequence requires multiple file accesses and therefore creates a timing window that cannot be safely controlled. In addition, some programs open files whose pathnames are input as parameters. Such files cannot be safely created or opened unless the entire pathname is verified.

If your code is forced to use pathnames received as input parameters, use the `safe_open()` routine rather than a less secure workaround (such as repetitive sequences of `lstat()`, `open()`, `fstat()` and trying to compare the results). See `safe_open(3)` for details about this routine.

A program that controls the pathname (does not receive it as input) can use `open()` only in relative safety and only if the call includes one of the following:

- `O_CREAT|O_EXCL` and a *mode* argument if creating a file
Mode bits should be set to `0600` for a file and to `0700` for a directory.
- `O_NOFOLLOW` if writing to an existing file

Both of these methods are guaranteed not to follow symbolic links, and they are the only secure defense against a symlink attack if you use an `open()` call. However, protection is much more difficult when opening files that already exist in world-writable areas. More specifically, the `O_NOFOLLOW` flag provides protection against being spoofed by a symbolic link, but unfortunately does not provide protection from being spoofed by hard links or FIFOs. The risk increases even more if the program receives the pathname as a parameter. For this reason, privileged code should not trust a file in a world-writable directory.

The `safe_open()` routine and the `open()` call's `O_NOFOLLOW` flag are not currently included in industry standards, so you should not assume that they are portable to other platforms.

No matter which of the defensive strategies that your program uses for opening temporary files in world-writable directories, it should not create temporary directories that are world writable. In fact, it is best if temporary directories are not even created to be world readable.

3.3.3 Recommended: Control core File Location with SSI_COREDIR

Applications that create a core file on abnormal termination, typically create the file in the current working directory. If that directory is world writable, the core file (whose name is obviously predictable) might leave the system vulnerable to a file name spoofing attack. Furthermore, core files created by some programs (those that run in `setuid/setgid` mode) often contain sensitive system information that should never be written to an insecure location.

The `SSI_COREDIR` request of the `setsysinfo()` function allows a program to set the directory where a core file would be created. The following example illustrates the call syntax for this request:

```
#include <sys/signal.h>
#include <sys/sysinfo.h>
#include <stdlib.h>

main (argc, argv)
int argc;
char *argv[];
{
    char *coredir;

    /*
     * fetch core directory value from environment variable
     */
    if (!(coredir = getenv("COREDIR"))) {
        printf("please \"setenv COREDIR\" to desired core directory\n");
        exit(1);
    }

    /*
     * set core directory
     */
    if (setsysinfo(SSI_COREDIR, NULL, 0, coredir, 0) == -1) {
        perror("setsysinfo(SSI_COREDIR)");
        exit(1);
    }

    /*
     * Now kill the process.  If core dumps are allowed, the core file will be
     * written to the specified core directory.
     */
    kill(getpid(), SIGSEGV);

    exit(1);
}
```

The `SSI_COREDIR` request is available only on systems running Tru64 UNIX Version 5.0A or higher. The request is not available through a patch.

3.3.4 Not Recommended: fopen() in World-Writable Directories

Using `fopen()` to open files in a world-writable directory and in any mode other than read only is unsafe. The fix is to replace `fopen()` with one of the following:

- A `mkstemp()` call, followed by an `fdopen()` call
- A `safe_open()` call

Following is an example of code that needs to be fixed:

```
FILE *fp;

fp = fopen (tempfile, "w");
```

Following is an example of how this code might be made safe:

```
FILE *fp;
int fd;

if (((fd = mkstemp (tempfile)) < 0) || ((fp = fdopen( fd, "w")) ==NULL))
{
/* Add your error handling here */
}
```

Note that the *mode* argument of the `fdopen()` call must be compatible with the mode (0600) that is used by `mkstemp()` when it creates the file.

3.3.5 Not Recommended: Certain libc Routines in World-Writable Directories

There are a number of `libc` routines that are required for standards conformance (and cannot be changed for the same reason) but are unsafe to use in world-writable directories. These include the following:

- `mktemp()`: Although this routine generates random file names, it does not open the file and thereby creates a timing window that can be exploited.
- `tmpnam()` and `tempname()`: These routines use `mktemp()` to create temporary file names. Therefore, they are subject to the same shortcomings as `mktemp()`.
- `creat()`: This function uses `open(path, O_WRONLY|O_CREAT|O_TRUNC, mode)`. It will therefore follow a symbolic link, possibly overwriting or creating the file in a location other than the one intended.
- `tmpfile()`: Although this routine uses `O_CREAT|O_EXCL` when opening the file, it can set file permissions to allow read and write access for the world.

3.3.6 Handling of Symbolic Links by `mkdir()`

This section discusses changes made for security reasons to the `mkdir()` call behavior. Using this function is not the safest way to create a directory in a world-writable area; using `mkdtemp()` is a safer way to do that. (Section 3.3.1.) However, your program might use `mkdir()` in another capacity, so you should be aware of the security issues associated with this function.

It is possible for the directory path specified to the `mkdir()` system call (or `mkdir` utility) to identify an existing symbolic link whose target is a directory that does not yet exist. If the symbolic link's existence is not expected by a privileged program (the program is being spoofed), creating the directory named by the symbolic link compromises system security. In versions of the operating system prior to Tru64 UNIX Version 5.0, `mkdir()` does not follow a final symbolic link at all and therefore cannot be spoofed into creating a symbolic link's target directory. In the unpatched Version 5.0, Version 5.1, and Version 5.1A releases, `mkdir()` does follow a final symbolic link and can be spoofed into creating the directory pointed to by the link rather than returning a "file exists" error.

It is possible that applications and scripts have been developed for or ported to the Version 5 release stream and that these applications are dependent on the changed behavior of the `mkdir()` function. In other words, these applications might not be privileged, create the specified symbolic link under the safe control of application procedures, and rely on `mkdir()` to follow the symbolic link and to create the directory pointed to by the link. This behavior is, in fact, allowed by some other UNIX implementations, but only if the specified directory path ends in a slash (/).

To balance security for most applications against the use of symbolic links that are known in advance and used by applications, there are two methods for controlling how `mkdir()` handles a final symbolic link:

- The `follow_mkdir_symlinks` attribute (discussed in Section 2.1.6), which can and should be left disabled to guard against file name spoofing attacks

This attribute is disabled (set to 0) by default so that `mkdir()` does not follow symbolic links. Although it can be enabled to allow symbolic links to be followed if a customer script or program was developed on the Version 5 release stream and inadvertently has a dependency on this behavior, no new script or program should be developed to depend on an enabled setting of this attribute.

- Directory path specification

Including a final slash (/) on the directory path enables `mkdir()` (or the `mkdir` utility) to follow a symbolic link and create the link target if it does

not exist. This method allows select programs and scripts to override the “disabled” setting of the `follow_mkdir_symlinks` attribute.

Omit the final slash in cases where the existence of the symbolic link is not designed into the program. Note that privileged programs should not include the final slash when creating a directory in a world-writable area.

A

Reference Pages

This appendix contains reference pages for the utilities and routines discussed in preceding chapters.

Index

A

access function, 3–14
attributes
(*See* system attributes)

B

Bourne shell
here document changes, 3–6
noclobber mode, 3–5

C

-C option of Bourne shell, 3–5
C shell
here document changes, 3–6
noclobber mode, 3–5
chmod utility, 3–3
ckfsec utility, 2–4
core files
controlling location of, 3–15
disabling for
 setuid/setgid programs, 2–3
 user processes, 2–3
creat function, 3–16

D

dirclean utility, 2–4
dump_cores attribute, 2–3
dump_setugid_cores attribute,
2–3

F

FIFOs
open restrictions, 2–2
file names
safe creation of random, 3–13
safe opening of specific, 3–14
follow_mkdir_symlinks attribute,
2–4, 3–17
fopen function, 3–16
fstat function, 3–14

H

hard links
and link function, 2–1
and rename function, 2–2
here documents, 3–6

K

Korn shell
here document changes, 3–6
noclobber mode, 3–5

L

link function, 2–1
lstat function, 3–14

M

mkdir function, 3–17
mkdir utility, 3–17
mkdtemp function, 3–13
mkstemp function, 3–13

mkstemp function, 3–13
mktemp function, 3–16
mktemp utility, 3–3
mode
for fdopen, 3–16
recommendations for setting, 3–2

N

noclobber mode in shell scripts,
3–4

O

O_CREAT flag of open call, 3–14
O_EXCL flag of open call, 3–14
O_NOFOLLOW flag of open call,
3–14
open function, 3–14
fifo open restrictions, 2–2
symlink follow restrictions, 2–2

P

path specifications, 3–2
for temporary files, 3–4
in mkdir function, 3–17
POSIX shell
here document changes, 3–6
noclobber mode, 3–5

R

random file name creation, 1–1
redirection operations in shell
scripts, 3–4
remove function, 3–14
rename function, 2–2
restricted_fifo_open attribute, 2–2
restricted_hardlink_creat
attribute, 2–1
restricted_symlink_follow
attribute, 2–2

S

safe_open function, 3–14
setsysinfo function
SSI_COREDIR request, 3–15
setsysinfo function,
SSI_COREDIR request, 2–3
shells
here document changes, 3–6
noclobber mode changes, 3–4
spoofing attacks, 1–1
prevention guidelines, 3–1
SSI_COREDIR request, 2–3, 3–15
stat function, 3–14
symbolic links
and mkdir function, 2–4
and open function, 2–2
system attributes, 2–1
dump_cores, 2–3
dump_setugid_cores, 2–3
follow_mkdir_symlinks, 2–4, 3–17
restricted_fifo_open, 2–2
restricted_hardlink_creat, 2–1
restricted_symlink_follow, 2–2

T

temporary files
cleaning out, 2–4
directory alternatives to /tmp
application-specific directory,
3–6
TMPDIR variable, 3–4
tmpfile function, 3–16
tmpnam function, 3–16
tmpname function, 3–16
tunable parameters
(See system attributes)

U

umask settings, 3–3