FIO21-C. Do not create temporary files in shared directories

Programmers frequently create temporary files in directories that are writable by everyone (examples are `/tmp` and `/var/tmp` on UNIX and `%TEMP%` on Windows) and may be purged regularly (for example, every night or during reboot).

Temporary files are commonly used for auxiliary storage for data that does not need to, or otherwise cannot, reside in memory and also as a means of communicating with other processes by transferring data through the file system. For example, one process will create a temporary file in a shared directory with a well-known name or a temporary name that is communicated to collaborating processes. The file then can be used to share information among these collaborating processes.

This practice is dangerous because a well-known file in a shared directory can be easily hijacked or manipulated by an attacker. Mitigation strategies include the following:

1. Use other low-level IPC (interprocess communication) mechanisms such as sockets or shared memory.
2. Use higher-level IPC mechanisms such as remote procedure calls.
3. Use a secure directory or a jail that can be accessed only by application instances (ensuring that multiple instances of the application running on the same platform do not compete).

There are many different IPC mechanisms; some require the use of temporary files, and others do not. An example of an IPC mechanism that uses temporary files is the POSIX `mmap()` function. Berkeley Sockets, POSIX Local IPC Sockets, and System V Shared Memory do not require temporary files. Because the multiuser nature of shared directories poses an inherent security risk, the use of shared temporary files for IPC is discouraged.

When two or more users or a group of users have write permission to a directory, the potential for deception is far greater than it is for shared access to a few files. Consequently, temporary files in shared directories must be

- Created unpredictable file names
- Created with unique names
- Opened only if the file doesn't already exist (atomic open)
- Opened with exclusive access
- Opened with appropriate permissions
- Removed before the program exits

The following table lists common temporary file functions and their respective conformance to these criteria:

Conformance of File Functions to Criteria for Temporary Files

	`tmpnam` **(C)**	**`tmpnam_s`** **(Annex K)**	**`tmpfile`** **(C/POSIX)**	**`tmpfile_s`** **(Annex K)**	**`mktemp`** **(POSIX)**	**`mkstemp`** **(POSIX)**
Unpredictable Name	Not portably	Yes	Not portably	Yes	Not portably	Not portably
Unique Name	Yes	Yes	Yes	Yes	Yes	Yes
Atomic open	No	No	Yes	Yes	No	Yes
Exclusive Access	Possible	Possible	No	If supported by OS	Possible	If supported by OS
Appropriate Permissions	Possible	Possible	If supported by OS*	If supported by OS	Possible	Not portably
File Removed	No	No	Yes*	Yes*	No	No

* If the program terminates abnormally, this behavior is implementation-defined.

Securely creating temporary files is error prone and dependent on the version of the C runtime library used, the operating system, and the file system. Code that works for a locally mounted file system, for example, may be vulnerable when used with a remotely mounted file system. Moreover, none of these functions are without problems. The only secure solution is to not create temporary files in shared directories.

Unique and Unpredictable File Names

Privileged programs that create temporary files in world-writable directories can be exploited to overwrite protected system files. An attacker who can predict the name of a file created by a privileged program 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 is coded securely, the program 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 program is executed [HP 2003]. Unprivileged programs can be similarly exploited to overwrite protected user files.

Exclusive Access

Exclusive access grants unrestricted file access to the locking process while denying access to all other processes and eliminates the potential for a race condition on the locked region. (See *Secure Coding in C and* C++, Chapter 8 [Seacord 2013].)

Files, or regions of files, can be locked to prevent two processes from concurrent access. Windows supports two types of file locks:

- *Shared locks*, provided by `LockFile()`, prohibit all write access to the locked file region while allowing concurrent read access to all processes.
- *Exclusive locks*, provided by `LockFileEx()`, grant unrestricted file access to the locking process while denying access to all other processes.

In both cases, the lock is removed by calling `UnlockFile()`.

Both shared locks and exclusive locks eliminate the potential for a race condition on the locked region. The exclusive lock is similar to a mutual exclusion solution, and the shared lock eliminates race conditions by removing the potential for altering the state of the locked file region (one of the required properties for a race).

These Windows file-locking mechanisms are called *mandatory locks* because every process attempting to access a locked file region is subject to the restriction. Linux implements mandatory locks and advisory locks. An advisory lock is not enforced by the operating system, which severely diminishes its value from a security perspective. Unfortunately, the mandatory file lock in Linux is also largely impractical for the following reasons:

- Mandatory locking works only on local file systems and does not extend to network file systems (such as NFS or AFS).
- File systems must be mounted with support for mandatory locking, and this is disabled by default.
- Locking relies on the group ID bit that can be turned off by another process (thereby defeating the lock).

Removal before Termination

Removing temporary files when they are no longer required allows file names and other resources (such as secondary storage) to be recycled. In the case of abnormal termination, there is no sure method that can guarantee the removal of orphaned files. For this reason, temporary file cleaner utilities, which are invoked manually by a system administrator or periodically run by a daemon to sweep temporary directories and remove old files, are widely used. However, these utilities are themselves vulnerable to file-based exploits and often require the use of shared directories. During normal operation, it is the responsibility of the program to ensure that temporary files are removed either explicitly or through the use of library routines, such as `tmpfile_s`, which guarantee temporary file deletion upon program termination.

Noncompliant Code Example (`fopen()/open()` with `tmpnam()`)

This noncompliant code example creates a file with a hard-coded `file_name` (presumably in a shared directory such as `/tmp` or `C:\Temp`):

```
#include <stdio.h>

void func(const char *file_name) {
  FILE *fp = fopen(file_name, "wb+");
  if (fp == NULL) {
    /* Handle error */
  }
}
```

Because the name is hard coded and consequently neither unique nor unpredictable, an attacker need only replace a file with a symbolic link, and the target file referenced by the link is opened and truncated.

This noncompliant code example attempts to remedy the problem by generating the file name at runtime using `tmpnam()`. The C `tmpnam()` function generates a string that is a valid file name and that is not the same as the name of an existing file. Files created using strings generated by the `tmpnam()` function are temporary in that their names should not collide with those generated by conventional naming rules for the implementation. The function is potentially capable of generating `TMP_MAX` different strings, but any or all of them may already be in use by existing files.

```
#include <stdio.h>

void func(void) {
  char file_name[L_tmpnam];
  FILE *fp;

  if (!tmpnam(file_name)) {
    /* Handle error */
  }

  /* A TOCTOU race condition exists here */

  fp = fopen(file_name, "wb+");
  if (fp == NULL) {
     /* Handle error */
  }
}
```

Because `tmpnam()` does not guarantee a unique name and `fopen()` does not provide a facility for an exclusive open, this code is still vulnerable.

The next noncompliant code example attempts to remedy the problem by using the POSIX `open()` function and providing a mechanism to indicate whether an existing file has been opened for writing or a new file has been created [IEEE Std 1003.1:2013]. If the `O_CREAT` and `O_EXCL` flags are used together, the `open()` function fails when the file specified by `file_name` already exists. To prevent an existing file from being opened and truncated, include the flags `O_CREAT` and `O_EXCL` when calling `open()`:

```
#include <stdio.h>

void func(void) {
  char file_name[L_tmpnam];
  int fd;

  if (!(tmpnam(file_name))) {
    /* Handle error */
  }

  /* A TOCTOU race condition exists here */

  fd = open(file_name, O_WRONLY | O_CREAT | O_EXCL | O_TRUNC,
            0600);
  if (fd < 0) {
     /* Handle error */
  }
}
```

This call to `open()` fails whenever `file_name` already exists, including when it is a symbolic link, but a temporary file is presumably still required. Additionally, the method used by `tmpnam()` to generate file names is not guaranteed to be unpredictable, which leaves room for an attacker to guess the file name ahead of time.

Care should be observed when using `O_EXCL` with remote file systems because it does not work with NFS version 2. NFS version 3 added support for `O_EXCL` mode in `open()`; see IETF RFC 1813 [Callaghan 1995], particularly the `EXCLUSIVE` value to the `mode` argument of `CREATE`.

Moreover, the `open()` function, as specified by the *Standard for Information Technology—Portable Operating System Interface (POSIX®), Base Specifications, Issue 7* [IEEE Std 1003.1:2013], does not include support for shared or exclusive locks. However, BSD systems support two additional flags that allow you to obtain these locks:

- `O_SHLOCK`: Atomically obtain a shared lock.
- `O_EXLOCK`: Atomically obtain an exclusive lock.

Noncompliant Code Example (`tmpnam_s()/open()`, Annex K, POSIX)

The C Standard function `tmpnam_s()` function generates a string that is a valid file name and that is not the same as the name of an existing file. It is almost identical to the `tmpnam()` function except for an added `maxsize` argument for the supplied buffer.

```
#define __STDC_WANT_LIB_EXT1__
#include <stdio.h>

void func(void) {
  char file_name[L_tmpnam_s];
  int fd;

  if (tmpnam_s(file_name, L_tmpnam_s) != 0) {
    /* Handle error */
  }

  /* A TOCTOU race condition exists here */

  fd = open(file_name, O_WRONLY | O_CREAT | O_EXCL | O_TRUNC,
            0600);
  if (fd < 0) {
    /* Handle error */
  }
}
```

Nonnormative text in the C Standard, subclause K.3.5.1.2 [ISO/IEC 9899:2011], also recommends the following:

> *Implementations should take care in choosing the patterns used for names returned by `tmpnam_s`. For example, making a thread id part of the names avoids the race condition and possible conflict when multiple programs run simultaneously by the same user generate the same temporary file names.*

If implemented, the space for unique names is reduced and the predictability of the resulting names is increased. In general, Annex K does not establish any criteria for the predictability of names. For example, the name generated by the `tmpnam_s` function from Microsoft Visual Studio consists of a program-generated file name and, after the first call to `tmpnam_s()`, a file extension of sequential numbers in base 32 (.1-.1vvvvvu).

Noncompliant Code Example (`mktemp()/open()`, POSIX)

The POSIX function `mktemp()` takes a given file name template and overwrites a portion of it to create a file name. The template may be any file name with exactly six X's appended to it (for example, `/tmp/temp.XXXXXX`). The six trailing X's are replaced with the current process number and/or a unique letter combination.

```
#include <stdio.h>
#include <stdlib.h>

void func(void) {
  char file_name[] = "tmp-XXXXXX";
  int fd;

  if (!mktemp(file_name)) {
    /* Handle error */
  }

  /* A TOCTOU race condition exists here */

  fd = open(file_name, O_WRONLY | O_CREAT | O_EXCL | O_TRUNC,
            0600);
  if (fd < 0) {
    /* Handle error */
  }
}
```

The `mktemp()` function is marked "LEGACY" in the Open Group Base Specifications Issue 6 [Open Group 2004]. The manual page for `mktemp()` gives more detail:

> *Never use `mktemp()`. Some implementations follow BSD 4.3 and replace XXXXXX by the current process id and a single letter, so that at most 26 different names can be returned. Since on the one hand the names are easy to guess, and on the other hand there is a race between testing whether the name exists and opening the file, every use of `mktemp()` is a security risk. The race is avoided by `mkstemp(3)`.*

Noncompliant Code Example (`tmpfile()`)

The `tmpfile()` function creates a temporary binary file that is different from any other existing file and that is automatically removed when it is closed or at program termination.

It should be possible to open at least `TMP_MAX` temporary files during the lifetime of the program. (This limit may be shared with `tmpnam()`.) Subclause 7.21.4.4, paragraph 6, of the C Standard allows for the value of the macro `TMP_MAX` to be as small as 25.

Most historic implementations provide only a limited number of possible temporary file names (usually 26) before file names are recycled.

```
#include <stdio.h>

void func(void) {
  FILE *fp = tmpfile();
  if (fp == NULL) {
    /* Handle error */
  }
}
```

Noncompliant Code Example (`tmpfile_s()`, Annex K)

The `tmpfile_s()` function creates a temporary binary file that is different from any other existing file and is automatically removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined.

The file is opened for update with `"wb+"` mode, which means "truncate to zero length or create binary file for update." To the extent that the underlying system supports the concepts, the file is opened with exclusive (nonshared) access and has a file permission that prevents other users on the system from accessing the file.

It should be possible to open at least `TMP_MAX_S` temporary files during the lifetime of the program. (This limit may be shared with `tmpnam_s()`.) The value of the macro `TMP_MAX_S` is required to be only 25 [ISO/IEC 9899:2011].

The C Standard, subclause K3.5.1.2, paragraph 7, notes the following regarding the use of `tmpfile_s()` instead of `tmpnam_s()` [ISO/IEC 9899:2011]:

> *After a program obtains a file name using the `tmpnam_s` function and before the program creates a file with that name, the possibility exists that someone else may create a file with that same name. To avoid this race condition, the `tmpfile_s` function should be used instead of `tmpnam_s` when possible. One situation that requires the use of the `tmpnam_s` function is when the program needs to create a temporary directory rather than a temporary file.*

```
#define __STDC_WANT_LIB_EXT1__
#include <stdio.h>

void func(void) {
  FILE *fp;

  if (tmpfile_s(&fp)) {
    /* Handle error */
  }
}
```

The `tmpfile_s()` function should not be used with implementations that create temporary files in a shared directory, such as `/tmp` or `C:`, because the function does not allow the user to specify a directory in which the temporary file should be created.

Compliant Solution (`mkstemp()`, POSIX)

The `mkstemp()` algorithm for selecting file names has shown to be immune to attacks. The `mkstemp()` function is available on systems that support the Open Group Base Specifications Issue 4, version 2 or later.

A call to `mkstemp()` replaces the six X's in the template string with six randomly selected characters and returns a file descriptor for the file (opened for reading and writing), as in this compliant solution:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

extern int secure_dir(const char *sdn);

void func(void) {
  const char *sdn = "/home/usr1/";
  char sfn[] = "/home/usr1/temp-XXXXXX";
  FILE *sfp;

  if (!secure_dir(sdn)) {
    /* Handle error */
  }

  int fd = mkstemp(sfn);
  if (fd == -1) {
    /* Handle error */
  }

  /*
   * Unlink immediately to hide the file name. The race
   * condition here is inconsequential if the file is created
   * with exclusive permissions (glibc >= 2.0.7).
   */
  if (unlink(sfn) == -1) {
    /* Handle error */
  }

  sfp = fdopen(fd, "w+");
  if (sfp == NULL) {
    close(fd);
    /* Handle error */
  }

  /* Use temporary file */

  fclose(sfp); /* Also closes fd */
}
```

This solution is not serially reusable, however, because the `mkstemp()` function replaces the `"XXXXXX"` in `template` the first time it is invoked. This is not a problem as long as `template` is reinitialized before calling `mkstemp()` again. If `template` is not reinitialized, the `mkstemp()` function will return `-1` and leave `template` unmodified because `template` did not contain six `X`'s.

The Open Group Base Specification Issue 6 [Open Group 2004] does not specify the permissions the file is created with, so these are implementation-defined. However, IEEE Std 1003.1, 2013 Edition [IEEE Std 1003.1:2013] specifies them as `S_IRUSR|S_IWUSR` (0600).

This compliant solution invokes the user-defined function `secure_dir()` (such as the one defined in FIO15-C. Ensure that file operations are performed in a secure directory) to ensure the temporary file resides in a secure directory.

Implementation Details

For GLIBC, versions 2.0.6 and earlier, the file is created with permissions 0666; for GLIBC, versions 2.0.7 and later, the file is created with permissions 0600. On NetBSD, the file is created with permissions 0600. This creates a security risk in that an attacker will have write access to the file immediately after creation. Consequently, programs need a private version of the `mkstemp()` function in which this issue is known to be fixed.

In many older implementations, the name is a function of process ID and time, so it is possible for the attacker to predict the name and create a decoy in advance. FreeBSD changed the `mk*temp()` family to eliminate the process ID component of the file name and replace the entire field with base-62 encoded randomness. This raises the number of possible temporary files for the typical use of six X's significantly, meaning that even `mktemp()` with six X's is reasonably (probabilistically) secure against guessing except under frequent usage [Kennaway 2000].

Exceptions

FIO21-C-EX1: The Annex K `tmpfile_s()` function can be used if all the targeted implementations create temporary files in secure directories.

Risk Assessment

Insecure temporary file creation can lead to a program accessing unintended files and permission escalation on local systems.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
FIO21-C	Medium	Probable	Medium	P8	L2

Automated Detection

Tool	Version	Checker	Description
CodeSonar	5.2p0	**BADFUNC.TEMP.***	A collection of checks that report uses of library functions associated with temporary file vulnerabilities
Compass/ROSE			Can detect violations of this recommendation. Specifically, Rose reports use of `tmpnam()`, `tmpnam_s()`, `tmpfile()`, and `mktemp()`
Coverity	6.5	**SECURE_TEMP**	Fully implemented
LDRA tool suite	9.7.1	**44 S**	Enhanced enforcement
Parasoft C/C++test	10.4.2	**CERT_C-FIO21-a**	Usage of functions prone to race is not allowed
Polyspace Bug Finder	R2019b	CERT C: Rec. FIO21-C	Checks for non-secure temporary file (rec. partially covered)
PRQA QA-C	9.7	**5016**	Partially implemented

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Related Guidelines

CERT C Secure Coding Standard	FIO15-C. Ensure that file operations are performed in a secure directory
SEI CERT C++ Coding Standard	VOID FIO19-CPP. Do not create temporary files in shared directories
CERT Oracle Secure Coding Standard for Java	FIO03-J. Remove temporary files before termination
ISO/IEC TR 24772:2013	Path Traversal [EWR]
MITRE CWE	CWE-379, Creation of temporary file in directory with insecure permissions

Bibliography

[HP 2003]	
[IEEE Std 1003.1:2013]	XSH, System Interfaces: `open` XSH, System Interfaces: `mkdopen`, `mksopen`
[ISO/IEC 9899:2011]	Subclause K.3.5.1.2, "The `tmpnam_s` Function" Subclause 7.21.4.4, "The `tmpnam` Function
[Kennaway 2000]	

[Open Group 2004]	`mkstemp()`
	`mktemp()`
	`open()`

[Seacord 2013]	Chapter 3, "Pointer Subterfuge" Chapter 8, "File I/O"
[Viega 2003]	Section 2.1, "Creating Files for Temporary Use"
[Wheeler 2003]	Chapter 7, "Structure Program Internals and Approach"