N1210 describes possible defects in ISO/IEC TR 24731-1 Extensions to the C Library, — Part I: Bounds-checking interfaces, document N1199.

Issue #1

The **`fopen_s()`** and **`freopen_s()`** functions are missing a mode character that will cause **`fopen()`** to fail rather than open a file that already exists. This is necessary to eliminate a time-of-creation to time-of-use race condition vulnerability caused when a programmer first problem caused by a .

The ISO/IEC 9899-1999 C standard function **`fopen()`** is typically used to open an existing file or create a new one. However, **`fopen()`** does not indicate if an existing file has been opened for writing or a new file has been created. This may lead to a program overwriting or accessing an unintended file.

In the following example, an attempt is made to check whether a file exists before opening it for writing by trying to open the file for reading.

```
...
FILE *fp = fopen("foo.txt","r");
if( !fp ) { /* file does not exist */
  fp = fopen("foo.txt","w");
  ...
  fclose(fp);
} else {
   /* file exists */
  fclose(fp);
}
...
```

However, this code suffers from a *Time of Check, Time of Use* (or *TOCTOU*) vulnerability. On a shared multitasking system there is a window of opportunity between the first call of **`fopen()`** and the second call for a malicious attacker to, for example, create a link with the given filename to an existing file, so that the existing file is overwritten by the second call of **`fopen()`** and the subsequent writing to the file.

The **`fopen_s()`** function defined in ISO/IEC TR 24731-1 is designed to improve the security of the **`fopen()`** function. However, like **`fopen()`**, **`fopen_s()`** provides no mechanism to determine if an existing file has been opened for writing or a new file has been created. The code below contains the same TOCTOU race condition as in the **`fopen()`** example above.

```
...
FILE *fptr;
errno_t res = fopen_s(&fptr,"foo.txt", "r");
if (res != 0) { /* file does not exist */
  res = fopen_s(&fptr,"foo.txt", "w");
```

```
  ...
  fclose(fptr);
} else {
  fclose(fptr);
}
...
```

The **fopen()** function does not indicate if an existing file has been opened for writing or a new file has been created. However, the **open()** function as defined in the Open Group Base Specifications Issue 6 [Open Group 04] provides such a mechanism. If the **O_CREAT** and **O_EXCL** flags are used together, the **open()** function fails when the file specified by **file_name** already exists.

```
...
int fd = open(file_name, O_CREAT | O_EXCL | O_WRONLY, new_file_mode);
if (fd == -1) {
  /* Handle Error */
}
...
```

The GCC compiler has implemented this suggestion by adding the **'x'** mode character as documented at:

http://www.gnu.org/software/libc/manual/html_mono/libc.html#Opening%20Streams

"The GNU C library defines one additional character for use in opentype: the character x insists on creating a new file--if a file filename already exists, fopen fails rather than opening it. If you use x you are guaranteed that you will not clobber an existing file. This is equivalent to the O_EXCL option to the open function (see Opening and Closing Files)."

Issue #2

The **tmpnam_s()** function should be removed from TR 24731-1 as it cannot be used securely and replaced with a function that can be used to create a temporary directory. TR 24731-1 currently provides the following rationale for including this function under “Recommended practice”:

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.

This capability can be more securely provided by creating a function that is equivalent to the Linux **mkdtemp()** function, for example:

```
errno_t tmpdir_s(FILE * restrict * restrict streamptr);
```

If the directory is successfully created, the pointer to **`FILE`** pointed to by **`streamptr`** is set to the pointer to the object controlling the opened directory. Otherwise, the pointer to **`FILE`** pointed to by **`streamptr`** is set to a null pointer.

Issue #3

There is currently no requirement that **`tmpfile_s()`** and **`tmpnam_s()`** functions produce unpredictable names. An attacker who is able to predict the name of a temporary file can often create a file or a link with the same name to a protected file. If the process which is manipulating the sensitive process has elevated privileges, this could result in an attacker obtaining elevated permissions on a system, or removing or truncating sensitive files. If the process is running with normal user privileges, these vulnerabilities can still be exploited on multi-user systems to trick a user into removing or truncating files that are normally only accessible by the user and system administrators.

The following techniques, which have all been used in various implementations, result in predictable filenames:

- Use the process ID
- Use the user ID
- Use the time of day
- Use a counter
- Use a bad random number generator

The current minimum value of the macro **`TMP_MAX_S`** needs to be increased from 25 to a reasonable limit which would prevent or inhibit the ability of an attacker to create temporary links for every possible filename. In addition, there should be a requirement that the sequence of names be generated in an unpredictable fashion.

TR 24731-1 also provides the following advice in non-normative text:

"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 this suggestion is followed, it reduces the possible name space for temporary file names, making these names more predictable.