N1339 describes extensions to the C1X library to enhance security in the C language. Most of these are existing functions, while some of these are enhancements or minor modifications to existing library functions.

#1 fopen() exclusive access with “x”

The C99 **`fopen()`** and **`freopen()`** 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.

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 24731-1is 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)."
```

#2 sigaction()

The C99 **`signal()`** function provides mechanisms for simple signal handling. However, it suffers from several serious vulnerabilities regarding portability.

Some platforms automatically reset a signal once it is passed to a signal handler. Signal handlers that also wish to catch future invocation of the signal often re-assert themselves to handle the signal, as shown in the following example:

```
...
void handler(int signum){
  if (signal(signum, handler) == SIG_ERR) {
    /* handle error */
  }
  /* handle signal */
}
/* … */
```

```
if (signal(SIGUSR1, handler) == SIG_ERR) {
  /* handle error */
}
```

However, this code suffers from a *Time of Check, Time of Use* (or *TOCTOU*) vulnerability. If two duplicate signals are sent, one immediately after the other, the first signal invokes the handler. But the second signal arrives before the handler can reassert its own signal. Consequently, the second signal triggers default behavior specified by the operating platform, and programs are powerless to avoid this problem.

The POSIX `sigaction()` function assigns handlers to signals in a manner similar to the C99 `signal()` function but also allows signal handler persistence to be controlled via the `SA_RESETHAND` flag. (Leaving the flag clear makes the handler persistent.)

```
/* Equivalent to signal(SIGUSR1, handler) but makes
 * signal persistent */
struct sigaction act;
act.sa_handler = handler;
act.sa_flags = 0;
if (sigemptyset(&act.sa_mask) != 0) {
  /* handle error */
}
if (sigaction(SIGUSR1, &act, NULL) != 0) {
  /* handle error */
}
```

Race conditions also can plague handlers designed to handle multiple signals. This code example registers a single signal handler to process both `SIGUSR1` and `SIGUSR2`. The variable `sig2` should be set to one if one or more `SIGUSR1` signals are followed by `SIGUSR2`, essentially implementing a finite state machine within the signal handler.

```
#include <signal.h>

volatile sig_atomic_t sig1 = 0;
volatile sig_atomic_t sig2 = 0;

void handler(int signum) {
  if (signum == SIGUSR1) {
    sig1 = 1;
  }
  else if (sig1) {
    sig2 = 1;
  }
}

int main(void) {
  if (signal(SIGUSR1, handler) == SIG_ERR) {
    /* handle error */
  }
  if (signal(SIGUSR2, handler) == SIG_ERR) {
    /* handler error */
  }

  while (sig2 == 0) {
    /* do nothing or give up CPU for a while */
  }

  /* ... */

  return 0;
}
```

Unfortunately, there is a race condition in the implementation of `handler()`. If `handler()` is called to handle `SIGUSR1` and is interrupted to handle `SIGUSR2`, it is possible that `sig2` will not be set.

The POSIX `sigaction()` function assigns handlers to signals in a similar manner to the C99 `signal()` function, but it also allows signal masks to be set explicitly. Consequently, `sigaction()` can be used to prevent a signal handler from interrupting itself.

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

volatile sig_atomic_t sig1 = 0;
volatile sig_atomic_t sig2 = 0;

void handler(int signum) {
  if (signum == SIGUSR1) {
    sig1 = 1;
  }
  else if (sig1) {
    sig2 = 1;
  }
}

int main(void) {
  struct sigaction act;
  act.sa_handler = &handler;
  act.sa_flags = 0;
  if (sigemptyset(&act.sa_mask) != 0) {
    /* handle error */
  }
  if (sigaddset(&act.sa_mask, SIGUSR1)) {
    /* handle error */
  }
  if (sigaddset(&act.sa_mask, SIGUSR2)) {
    /* handle error */
  }

  if (sigaction(SIGUSR1, &act, NULL) != 0) {
    /* handle error */
  }
  if (sigaction(SIGUSR2, &act, NULL) != 0) {
    /* handle error */
  }

  while (sig2 == 0) {
    /* do nothing or give up CPU for a while */
  }

  /* ... */

  return 0;
}
```

POSIX states:

> The *sigaction()* function provides a more comprehensive and reliable mechanism for controlling signals; new applications should use *sigaction()* rather than *signal*().

#3 random()

Pseudorandom number generators use mathematical algorithms to produce a sequence of numbers with good statistical properties, but the numbers produced are not genuinely random. The C Standard function `rand()` (available in `stdlib.h`) does not have good random number properties. The numbers generated by `rand()` have a comparatively short cycle, and the numbers may be predictable.

The following code generates an ID with a numeric part produced by calling the `rand()` function. The IDs produced are predictable and have limited randomness.

```
enum {len = 12};
char id[len];  /* id will hold the ID, starting with
                * the characters "ID" followed by a
                * random integer */
int r;
int num;
/* ... */
r = rand();  /* generate a random integer */
num = snprintf(id, len, "ID%-d", r);  /* generate the ID */
/* ... */
```

A better pseudorandom number generator is the `random()` function, defined in POSIX. While the low dozen bits generated by `rand()` go through a cyclic pattern, all the bits generated by `random()` are usable.

```
enum {len = 12};
char id[len];  /* id will hold the ID, starting with
                * the characters "ID" followed by a
                * random integer */
int r;
int num;
```

```
/* ... */
time_t now = time(NULL);
if (now == (time_t) -1) {
  /* handle error */
}
srandom(now);  /* seed the PRNG with the current time */
/* ... */
r = random();  /* generate a random integer */
num = snprintf(id, len, "ID%-d", r);  /* generate the ID */
/* ... */
```

#4 fremove()

Removing a file via its filename is subject to a race condition. The following code example shows a case where a file is removed. After the program closes the open file, but before the program removes the file, an attacker can substitute the file to be removed with a different file, causing the program to remove the wrong file.

```
char *file_name;
FILE* fd = fopen( file_name, …);

fclose(fd);
if (remove(file_name) != 0) {
  /* Handle error condition */
}
```

File handles are immune to race conditions, but unfortunately, invoking `remove()` on an open file is implementation-defined and hence non-portable.

We propose a function, tentatively called **fremove()** that closes an open file descriptor, and removes the corresponding file as an atomic operation. This would prevent the aforementioned race condition. The fremove() function takes both a file name and an open file descriptor. It would delete the file name only if it still corresponded to the open file descriptor, and it would close the file descriptor, all atomically.

```
FILE *file;
char *file_name;

/* initialize file_name */

file = fopen(file_name, "w+");
if (file == NULL) {
  /* Handle error condition */
}

if (fremove(file, file_name) != 0) {
  /* Handle error condition */
}
```

#5 const char getenv()

C99 defines `getenv` as follows:

> The `getenv` function returns a pointer to a string associated with the matched list member. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the `getenv` function. If the specified name cannot be found, a null pointer is returned.

Consequently, if the string returned by `getenv()` needs to be altered, a local copy should be created to ensure that the environment is not directly and unintentionally modified.

We propose that getenv() should return not a char*, but rather a const char*. This will prevent programmers from inadvertently modifying the string.

#6 setenv()

C99 currently provides no mechanism to modify environment variables, although it provides the getenv() method to access environment.variables. The POSIX setenv() function has become the standard means of modifying individual environment variables.

#7 clearenv()

Because environment variables are inherited from the parent process when a program is executed, an attacker can easily sabotage variables, causing a program to behave in an unexpected and insecure manner [Viega 03].

All programs, particularly those running with higher privileges than the caller should treat their environment as untrusted user input. Because the environment is inherited by processes spawned by calls to the `fork()`, `system()`, or `exec()` functions, it is important to verify that the environment does not contain any values that can lead to unexpected behavior.

C99 states that, "the set of environment names and the method for altering the environment list are implementation-defined." Because some programs may behave in unexpected ways when certain environment variables are not set, it is important to understand which variables are necessary on your system and what are safe values for them.

The nonstandard function `clearenv()` may be used to clear out the environment in an implementation-defined manner that would ensure non-critical values are removed, and critical ones are set to default values.

#8 siglongjmp()

The C99 **longjmp()** function does not specify its behavior with respect to signals. This may result in undefined behavior for programs that rely on signal handling. The POSIX **siglongjmp()** and **sigsetjmp()** functions address this problem by specifying that the signal state is restored. This is particularly significant for programs that use the proposed **sigaction()** function.

#9 fmkstemp()

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 is a dangerous practice, because a well-known file in a shared directory can be easily hijacked or manipulated by an attacker.

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 must be:

1. created with unique and unpredictable file names,
2. opened with exclusive access,
3. removed before the program exits, and
4. opened with appropriate permissions.

The C99 **tmpnam()** function fails to address most of these concerns. A better alternative is the POSIX **mkstemp()** function which uses a cryptographically secure algorithm to generate the filename, and attempts to properly open the file on systems that support exclusive access and permissions.

Because POSIX file semantics differ from C99, a slightly modified version of this function would need to be introduced. This version would return a FILE * instead of an int.

FILE * fmkstemp(char *template);