Document No.: SC22/WG14/N1358
Date: January 26, 2009
Reply to: David Svoboda, svoboda@cert.org

Extensions to the C1X Library

This document 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.

In Milpitas in September, CERT submitted document N1339, which proposed many functions. Some were regarded favorably, while others were regarded not so favorably. This document formalizes the functions regarded favorably, addressing issues presented in Milpitas, and adds a few new function proposals.

#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 (TOCTOU) 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* (*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()`** 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 GNU libc package has implemented this suggestion by adding the **'x'** mode character to **fopen()** as documented at:
http://www.gnu.org/software/libc/manual/html_mono/libc.html#Opening-Streams

```
"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)."
```

We propose adding an ‘`x`’ character to the mode argument to **fopen()**. This would maintain compatibility with glibc. The necessary revisions to the C standard would be as follows:

Section 7.19.5.3 (The **fopen** function), paragraph 3 would receive the following additions:

wx	`create text file for writing with exclusive access`
wbx	`create binary file for writing with exclusive access`
w+x	`create text file for update with exclusive access`
w+bx *or* **wb+x**	`create binary file for update with exclusive access`

The following paragraph should be inserted before paragraph 5:

> `Opening a file with exclusive mode (`**`'x'`**` as the last character in the `**`mode`**` argument) fails if the file already exists or cannot be created.  Otherwise, the file is created with exclusive (also known as non-shared) access to the extent that the underlying system supports exclusive access.`

#2 fopen_s() exclusive access with “x”

The **fopen_s()** function defined in ISO/IEC TR 24731-1 is designed to improve the security of the **fopen()** function. TR24731-1, section 6.5.2.1, paragraph 6 indicates that when **fopen_s()** is used to open a file for writing, access to the file is exclusive to the program in as far as the underlying system supports exclusivity. 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);
}
...
```

We propose adding an '`x`' character to the mode argument to **`fopen_s().`** The necessary revisions to the C standard would be as follows:

Section 6.5.2.1 (The **`fopen_s`** function), paragraph 5 would receive the following additions:

uw+bx *or* **uwb+x**	create binary file for update with exclusive access, default permissions
wx	create text file for writing, with exclusive access
wbx	create binary file for writing, with exclusive access
w+x	create text file for update, with exclusive access
w+bx *or* **wb+x**	create binary file for update, with exclusive access

The following paragraph should be inserted before paragraph 6:

> `Opening a file with exclusive mode (`**`'x'`**` as the last character in the `**`mode`**` argument) fails if the file already exists or cannot be created.`

#3 rand() disclaimer

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 */
/* ... */
```

At the last WG14 meeting in Milpitas, we proposed adding the **random()** function, as specified by POSIX, to the C standard. The WG14 committee was unfavorable to this proposal. They agreed that **rand()** was a poor random-number generator (RNG), but experts did not agree on what constitutes a 'good' RNG. They suggested that CERT should draft a disclaimer for **rand()** for inclusion in the standard. We hereby provide such a disclaimer:

Add a footnote to 7.20.2.1, paragraph 4, which reads:

> These specifications for a pseudo-random sequence generator do not guarantee that the numbers generated are sufficiently random for applications with strict randomness requirements, such as cryptographic applications. There are several implementations of **rand()** which are known to produce insufficient results. For instance, the low-order bits may follow a short cycle. Specifications for proper pseudo-random number generation are beyond the scope of this document.

#4 resetenv() to clean the environment

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.

At the last WG14 meeting in Milpitas, we proposed adding the **`clearenv()`** function, as implemented by Linux and several other platforms, to the C standard. However, upon further reflection, we realized that a standardized **`clearenv()`** would have the following problems:

- Clearing the environment is not always possible. For instance Windows does not permit users without administrative privileges to clear system environment variables; they may only clear user environment variables.

- An 'empty' environment is not particularly desirable. The purpose of clearing the environment is to prevent unwanted or malicious environment variables from being transmitted to an external program via **`system()`** or **`exec**()`**. But these programs require some environment variables to be set in order to function properly. For instance, CERT Secure Coding Rule **`ENV03-C`** recommends setting **`PATH`**, **`TZ`**, and **`IFS`** before calling an external program on an empty environment.

- Finally, **`clearenv()`** shall soon be standardized in POSIX, however as noted above, it cannot be implemented in Windows, and the C standard should not contain an incompatible definition of **`clearenv()`**.

In particular, we are less interested in an empty, cleared environment, than we are in a sanitized, 'cleaned' environment, for use in calling external programs. Consequently, we propose a **`resetenv()`** function. This function would sanitize the environment, eliminating all permissible environment variables, and leaving a few variables known to have 'good' values. The variables left would be implementation-defined, and designed so that a subsequent call to **`system()`** or **`exec**()`** would be feasible, yet not transmit nefarious information to the external program.

For instance, a Linux **`resetenv()`** function might clear the environment, and set **`PATH`**, **`TZ`**, and **`IFS`** to default values. Likewise, a Windows **`resetenv()`** function might remove all user environment variables, and if the program had administrator access would also clear the system environment variables except **`PATH`**, which would be set to a default value.

This proposal does not specify details of the environment that **`resetenv()`** will create; those are considered implementation-dependent. Consequently, this proposal cannot ensure that an implementation of **`resetenv()`** would be sufficient; implementations could fail to clean nefarious environment variables, or could clean variables necessary to a subsequent call to an external program.

The proposal is as follows:

Add this section after 7.20.4.5 (The **`getenv`** Function)

```
  7.20.4.6 The resetenv function
  Synopsis
          #include <stdlib.h>
          void resetenv();
  Description

  The resetenv function replaces the current host
environment with a minimal host environment. This minimal
environment is independent of the environment state before
the call to resetenv, and is a suitable environment for a
subsequent call to system. The set of names and values in
the minimal environment and the method for altering the
environment list are implementation-defined.

  The implementation shall behave as if no library function
calls the resetenv function.

  Returns
  The resetenv function returns no value.
```

#5 memset_s() to clear memory, without fear of removal

The **`memset()`** function, defined in Section 7.21.6.1, sets a range of memory to a value, and is often used to zero out a series of bytes. However, this function is insufficient in circumstances involving sensitive data, as described in CERT Secure Coding rule MSC06-C. Consider the following code:

```
  void getPassword(void) {
    char pwd[64];
    if (GetPassword(pwd, sizeof(pwd))) {
      /* checking of password, secure operations, etc */
    }
    memset(pwd, 0, sizeof(pwd));
  }
```

This code is subject to a potential vulnerability. An optimizing compiler could employ "dead store removal"; that is, it could decide that **`pwd`** is never accessed after the call to **`memset`**(), ergo the call to **`memset()`** could be optimized away. Consequently, the password remains in memory, possibly to be discovered by some other process requesting memory.

There are several solutions to this problem, but no solution appears to be both portable and optimal. The solutions currently known are as follows:

1. Append a volatile access after the **`memset()`**:

```
memset(pwd, 0, sizeof(pwd));
*(volatile char*)pwd = *(volatile char*)pwd;
```

However, the MIPSpro compiler and versions 3 and later of GCC cleverly zero out only the first byte and leave the rest of the **`pwd`** array intact.

2. Replace **`memset()`** with **`ZeroMemory()`**:

```
ZeroMemory(pwd, sizeof(pwd));
```

This function also might be optimized away, and is only available on Windows.

3. Replace **`memset()`** with **`SecureZeroMemory()`**:

```
SecureZeroMemory(pwd, sizeof(pwd));
```

This function is guaranteed not to be optimized away, but it is only available on Windows.

4. Pragmas

```
#pragma optimize("", off)
/* clear memory */
#pragma optimize("", on)
```

This approach will prevent the clearing of memory from being optimized away. However, this pragma is not portable.

5. Platform-independent ' secure-memset' solution:

```
void *secure_memset(void *v, int c, size_t n) {
  volatile unsigned char *p = v;
  while (n--) *p++ = c;
  return v;
}
```

This approach will prevent the clearing of memory from being optimized away, and it should work on any standard-compliant platform. There has been recent notice that some compilers violate the standard by not always respecting the **`volatile`** qualifier. Also, this compliant solution may not be as efficient as possible due to the nature of the volatile type qualifier preventing the compiler from optimizing the code at all. Typically, some compilers are smart enough to replace calls to **`memset()`** with equivalent assembly instructions that are much more efficient than the **`memset()`** implementation. Implementing a **`secure_memset()`** function as shown in the example may prevent the compiler from using the optimal assembly instructions and may result in less efficient code.

We propose a **`memset_s()`** function that behaves like **`memset()`**, with the added stipulation that the call to **`memset_s()`** is guaranteed not to be optimized away. It may be implemented like **`SecureZeroMemory()`**, or it might be implemented like the **`secure_memset()`** described above. The implementation is encouraged to implement it in an optimal fashion. We thus propose the following:

Add the following section after section 7.21.6.1 *The **`memset()`** function*:

```
  7.21.6.2 The memset_s function

  Synopsis
     #include <string.h>
     void *memset_s(void *s, int c, size_t n);

  Description
  The memset_s function copies the value of c (converted to
an unsigned char) into each of the first n characters of
the object pointed to by s. Unlike memset, any call to
memset_s shall be evaluated strictly according to the rules
of the abstract machine, as described in 5.1.2.3. That is
to say, any call to memset_s shall assume that the memory
indicated by s and n may be accessible in the future and
therefore must contain the values indicated by c.

  Returns
  The memset_s function returns the value of s.
```

One final note: While necessary for working with sensitive information, this **`memset_s()`** function may not be sufficient, as it does nothing to prevent memory from being swapped to disk, or written out in a core dump. More information on such issues is available at the CERT C Secure Coding rule MEM06-C.