#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:

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.

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

7.21.6.2 The `memset_s` Function

Synopsis

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t memset_s(void * restrict s, rsize_t smax,
     int c, rsize_t n)
```

Runtime-constraints

The argument **s** shall not be a null pointer. Neither **smax** nor **n** shall be greater than **RSIZE_MAX**. **n** shall not be greater than **smax**.

If there is a runtime-constraint violation, the **memset_s** function stores the value of **c** (converted to an **unsigned char**) into each of the first **smax** characters of the object pointed to by **s** if **s** is not a null pointer and **smax** is not greater than **RSIZE_MAX**.

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, 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 zero if there was no runtime-constraint violation. Otherwise, a non-zero value is returned.