MSC06-C. Beware of compiler optimizations

Subclause 5.1.2.3 of the C Standard [ISO/IEC 9899:2011] states:

> *In the abstract machine, all expressions are evaluated as specified by the semantics. An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced (including any caused by calling a function or accessing a volatile object).*

This clause gives compilers the leeway to remove code deemed unused or unneeded when building a program. Although this functionality is usually beneficial, sometimes the compiler removes code that it thinks is not needed but has been added for a specific (often security-related) purpose.

Noncompliant Code Example (`memset()`)

An example of unexpected and unwanted compiler optimizations involves overwriting the memory of a buffer that is used to store sensitive data. As a result, care must always be taken when dealing with sensitive data to ensure that operations on it always execute as intended. Some compiler optimization modes can remove code sections if the optimizer determines that doing so will not alter the behavior of the program. In this noncompliant code example, optimization may remove the call to `memset()` (which the programmer had hoped would clear sensitive memory) because the variable is not accessed following the write. Check compiler documentation for information about this compiler-specific behavior and which optimization levels can cause this behavior to occur.

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

For all of the compliant solutions provided for this recommendation, it is strongly recommended that the programmer inspect the generated assembly code in the optimized release build to ensure that memory is actually cleared and none of the function calls are optimized out.

Noncompliant Code Example (`Touching Memory`)

This noncompliant code example accesses the buffer again after the call to `memset()`. This technique prevents some compilers from optimizing out the call to `memset()` but does not work for all implementations. For example, the MIPSpro compiler and versions 3 and later of GCC cleverly nullify only the first byte and leave the rest intact. Check compiler documentation to guarantee this behavior for a specific platform.

```
void getPassword(void) {
  char pwd[64];
  if (retrievePassword(pwd, sizeof(pwd))) {
    /* Checking of password, secure operations, etc. */
  }
  memset(pwd, 0, sizeof(pwd));
  *(volatile char*)pwd= *(volatile char*)pwd;
}
```

Noncompliant Code Example (Windows)

This noncompliant code example uses the `ZeroMemory()` function provided by many versions of the Microsoft Visual Studio compiler:

```
void getPassword(void) {
  char pwd[64];
  if (retrievePassword(pwd, sizeof(pwd))) {
    /* Checking of password, secure operations, etc. */
  }
  ZeroMemory(pwd, sizeof(pwd));
}
```

A call to `ZeroMemory()` may be optimized out in a similar manner to a call to `memset()`.

Compliant Solution (Windows)

This compliant solution uses a `SecureZeroMemory()` function provided by many versions of the Microsoft Visual Studio compiler. The documentation for the `SecureZeroMemory()` function guarantees that the compiler does not optimize out this call when zeroing memory.

```
void getPassword(void) {
  char pwd[64];
  if (retrievePassword(pwd, sizeof(pwd))) {
    /* Checking of password, secure operations, etc. */
  }
  SecureZeroMemory(pwd, sizeof(pwd));
}
```

Compliant Solution (Windows)

The `#pragma` directives in this compliant solution instruct the compiler to avoid optimizing the enclosed code. This `#pragma` directive is supported on some versions of Microsoft Visual Studio and could be supported on other compilers. Check compiler documentation to ensure its availability and its optimization guarantees.

```
void getPassword(void) {
  char pwd[64];
  if (retrievePassword(pwd, sizeof(pwd))) {
    /* Checking of password, secure operations, etc. */
  }
#pragma optimize("", off)
  memset(pwd, 0, sizeof(pwd));
#pragma optimize("", on)
}
```

Compliant Solution (C99)

This compliant solution uses the `volatile` type qualifier to inform the compiler that the memory should be overwritten and that the call to the `memset_s()` function should not be optimized out. Unfortunately, this compliant solution may not be as efficient as possible because of 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 `memset_s()` function as shown in the example may prevent the compiler from using the optimal assembly instructions and can result in less efficient code. Check compiler documentation and the assembly output from the compiler.

```
/* memset_s.c */
errno_t memset_s(void *v, rsize_t smax, int c, rsize_t n) {
  if (v == NULL) return EINVAL;
  if (smax > RSIZE_MAX) return EINVAL;
  if (n > smax) return EINVAL;

  volatile unsigned char *p = v;
  while (smax-- && n--) {
    *p++ = c;
  }

  return 0;
}

/* getPassword.c */
extern errno_t memset_s(void *v, rsize_t smax, int c, rsize_t n);

void getPassword(void) {
  char pwd[64];

  if (retrievePassword(pwd, sizeof(pwd))) {
     /* Checking of password, secure operations, etc. */
  }
  if (memset_s(pwd, sizeof(pwd), 0, sizeof(pwd)) != 0) {
    /* Handle error */
  }
}
```

However, note that both calling functions and accessing `volatile`-qualified objects can still be optimized out (while maintaining strict conformance to the standard), so this compliant solution still might *not* work in some cases. The `memset_s()` function introduced in C11 is the preferred solution (see the following solution for more information). If `memset_s()` function is not yet available on your implementation, this compliant solution is the best alternative, and can be discarded once supported by your implementation.

Compliant Solution (C11)

The C Standard includes a `memset_s` function. Subclause K.3.7.4.1, paragraph 4 [ISO/IEC 9899:2011], states:

> *Unlike `memset`, any call to the `memset_s` function shall be evaluated strictly according to the rules of the abstract machine as described in (5.1.2.3). That is, any call to the `memset_s` function shall assume that the memory indicated by s and n may be accessible in the future and thus must contain the values indicated by c.*

```
void getPassword(void) {
  char pwd[64];

  if (retrievePassword(pwd, sizeof(pwd))) {
     /* Checking of password, secure operations, etc. */
  }
  memset_s(pwd, 0, sizeof(pwd));
}
```

Noncompliant Code Example

In rare cases, use of an empty infinite loop may be unavoidable. For example, an empty loop may be necessary on a platform that does not support `sleep(3)` or an equivalent function. Another example occurs in OS kernels. A task started before normal scheduler functionality is available may not have access to `sleep(3)` or an equivalent function. An empty infinite loop that does nothing within the loop body is a suboptimal solution because it consumes CPU cycles but performs no useful operations. An optimizing compiler can remove such a loop, which can lead to unexpected results. According to the C Standard, subclause 6.8.5, paragraph 6 [ISO/IEC 9899:2011],

> *An iteration statement whose controlling expression is not a constant expression, that performs no input/output operations, does not access volatile objects, and performs no synchronization or atomic operations in its body, controlling expression, or (in the case of a **`for`** statement) its* expression-3, *may be assumed by the implementation to terminate.*[157]
>
> *157) This is intended to allow compiler transformations, such as removal of empty loops, even when termination cannot be proven.*

This noncompliant code example implements an idle task that continuously executes a loop without executing any instructions within the loop. An optimizing compiler could remove the `while` loop in the example.

```
static int always = 1;
int main(void) {
  while (always) { }
}
```

Compliant Solution (`while`)

To avoid the loop being optimized away, this compliant solution uses a constant expression `(1)` as the controlling expression in the `while` loop:

```
int main(void) {
  while (1) { }
}
```

Compliant Solution (`for`)

According to the C Standard, subclause 6.8.5.3, paragraph 2, omitting the *expression-2* from a `for` loop will replace that expression with a nonzero constant.

```
int main(void) {
  for (;;) { }
}
```

Risk Assessment

If the compiler optimizes out memory-clearing code, an attacker can gain access to sensitive data.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
MSC06-C	Medium	Probable	Medium	P8	L2

Related Vulnerabilities

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

Automated Detection

Tool	Version	Checker	Description
CodeSonar	5.2p0	**BADFUNC.MEMSET**	Use of memset
LDRA tool suite	9.7.1	**35 S, 57 S, 8 D, 65 D, 76 D, 105 D, I J, 3 J**	Partially implemented
PVS-Studio	6.23	**V597**, **V712**	

Related Guidelines

SEI CERT C++ Coding Standard	VOID MSC06-CPP. Be aware of compiler optimization when dealing with sensitive data
CERT Oracle Secure Coding Standard for Java	MSC01-J. Do not use an empty infinite loop
MITRE CWE	CWE-14, Compiler removal of code to clear buffers

Bibliography

[ISO/IEC 9899:2011]	Subclause 6.8.5, "Iteration Statements" Subclause K.3.7.4.1, "The `memset_s` Function"
[MSDN]	"SecureZeroMemory" "Optimize (C/C++)"
[PVS-Studio]	"Safe Clearing of Private Data"
[US-CERT]	"MEMSET"
[Wheeler 2003]	Section 11.4, "Specially Protect Secrets (Passwords and Keys) in User Memory"