MSC07-C. Detect and remove dead code

This rule has been deprecated. It has been merged with:
- MSC12-C. Detect and remove code that has no effect or is never executed

10/07/2014 -- Version 2.0

Code that is never executed is known as dead code. Typically, the presence of dead code indicates that a logic error has occurred as a result of changes to a program or the program’s environment. Dead code is usually optimized out of a program during compilation. However, to improve readability and ensure that logic errors are resolved, dead code should be identified, understood, and eliminated.

This recommendation is related to MSC12-C. Detect and remove code that has no effect or is never executed.

Noncompliant Code Example

This noncompliant code example demonstrates how dead code can be introduced into a program [Fortify 2006]. The second conditional statement, if (s)
, will never evaluate true because it requires that s not be assigned NULL, and the only path where s can be assigned a non-null value ends with a return statement.

```c
int func(int condition) {
    char *s = NULL;
    if (condition) {
        s = (char *)malloc(10);
        if (s == NULL) {
            /* Handle Error */
        }
        /* Process s */
        return 0;
    }
    /* ... */
    if (s) {
        /* This code is never reached */
    }
    return 0;
}
```

Compliant Solution

Remediation of dead code requires the programmer to determine why the code is never executed and then to resolve the situation appropriately. To correct the preceding noncompliant code, the return is removed from the body of the first conditional statement.

```c
int func(int condition) {
    char *s = NULL;
    if (condition) {
        s = (char *)malloc(10);
        if (s == NULL) {
            /* Handle Error */
        }
        /* Process s */
    }
    /* ... */
    if (s) {
        /* This code is now reachable */
    }
    return 0;
}
```

Noncompliant Code Example
In this example, the `strlen()` function is used to limit the number of times the function `s_loop()` will iterate. The conditional statement inside the loop evaluates to true when the current character in the string is the null terminator. However, because `strlen()` returns the number of characters that precede the null terminator, the conditional statement never evaluates true.

```c
int s_loop(char *s) {
    size_t i;
    size_t len = strlen(s);
    for (i=0; i < len; i++) {
        /* ... */
        if (s[i] == '\0') {
            /* This code is never reached */
        }
    }
    return 0;
}
```

Compliant Solution

Removing the dead code depends on the intent of the programmer. Assuming the intent is to flag and process the last character before the null terminator, the conditional is adjusted to correctly determine if the `i` refers to the index of the last character before the null terminator.

```c
int s_loop(char *s) {
    size_t i;
    size_t len = strlen(s);
    for (i=0; i < len; i++) {
        /* ... */
        if (s[i+1] == '\0') {
            /* This code is now reached */
        }
    }
    return 0;
}
```

Exceptions

**MSC07-C-EX1:** In some situations, seemingly dead code may make software resilient. An example is the default label in a `switch` statement whose controlling expression has an enumerated type and that specifies labels for all enumerations of the type. (See `MSC01-C. Strive for logical completeness.`) Because valid values of an enumerated type include all those of its underlying integer type, unless enumeration constants are provided for all those values, the default label is appropriate and necessary.

```c
typedef enum { Red, Green, Blue } Color;
const char* f(Color c) {
    switch (c) {
    case Red: return "Red";
    case Green: return "Green";
    case Blue: return "Blue";
    default: return "Unknown color"; /* Not dead code */
    }
}
```

**MSC07-EX2:** It is also permissible to temporarily remove code that may be needed later. (See `MSC04-C. Use comments consistently and in a readable fashion` for an illustration.)

Risk Assessment

The presence of dead code may indicate logic errors that can lead to unintended program behavior. The ways in which dead code can be introduced into a program and the effort required to remove it can be complex. As a result, resolving dead code can be an in-depth process requiring significant analysis.
Automated Detection

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeSonar</td>
<td>5.2p0</td>
<td>DIAG.UNEX.* LANG.STRUCT.RC LANG.STRUCT.UC</td>
<td>Code not exercised by analysis&lt;br&gt;Redundant condition&lt;br&gt;Unreachable (Call, Computation, Conditional, Control Flow, Data Flow)</td>
</tr>
<tr>
<td>Coverity</td>
<td>2017.07</td>
<td>DEADCODE UNREACHABLE</td>
<td>Can detect the specific instance where code can never be reached because of a logical contradiction or a dead “default” in switch statement&lt;br&gt;Can detect the instances where code block is unreachable because of the syntactic structure of the code</td>
</tr>
<tr>
<td>GCC</td>
<td>4.3.5</td>
<td></td>
<td>Can detect violations of this recommendation when the –Wunreachable-code flag is used</td>
</tr>
<tr>
<td>Klocwork</td>
<td>2018</td>
<td>LA_UNUSED UNREACH.GEN UNREACH.RETURN UNREACH.SIZEOF INVARIANT_CONDITION.UNREACH</td>
<td>Fully implemented</td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>1 J 138 S 140 S</td>
<td></td>
</tr>
<tr>
<td>Parasoft C/C++ test</td>
<td>10.4.2</td>
<td>CERT_C-MSC07-a CERT_C-MSC07-b CERT_C-MSC07-c CERT_C-MSC07-d CERT_C-MSC07-e CERT_C-MSC07-f CERT_C-MSC07-g CERT_C-MSC07-h</td>
<td>There shall be no unreachable code in &quot;else&quot; block&lt;br&gt;There shall be no unreachable code after ‘return’, ‘break’, ‘continue’, and ‘goto’ statements&lt;br&gt;There shall be no unreachable code in &quot;if\else/while/for&quot; block&lt;br&gt;There shall be no unreachable code in switch statement&lt;br&gt;There shall be no unreachable code in ‘for’ loop&lt;br&gt;There shall be no unreachable code after ‘if’ or ‘switch’ statement&lt;br&gt;There shall be no unreachable code after “if” or “switch” statement inside while/for/do...while loop&lt;br&gt;Avoid switch with unreachable branches</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2019b</td>
<td>CERT C: Rule MSC07-C</td>
<td>Checks for:&lt;br&gt;- Code does not execute&lt;br&gt;- Default case is missing and may be reached&lt;br&gt;- Code following control-flow statements</td>
</tr>
<tr>
<td>PROQA QA-C</td>
<td>9.7</td>
<td>2006, 2877, 2880, 2881, 2882, 2883, 2980, 2981, 2982, 2983, 2984, 2985, 2986, 3202, 3203, 3205, 3206, 3207, 3210, 3219, 3229, 3404, 3422, 3423, 3425, 3470, 1501, 1503</td>
<td>Fully implemented</td>
</tr>
<tr>
<td>SonarQube C/C++ Plugin</td>
<td>3.11</td>
<td>S1763, S1751</td>
<td>Can detect violations of this recommendation when the –Wunreachable-code flag is used</td>
</tr>
<tr>
<td>Splint</td>
<td>3.1.1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Related Vulnerabilities

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

CVE-2014-1266 results from a violation of this rule. There is a spurious \texttt{goto fail} statement on line 631 of \texttt{sslKeyExchange.c}. This \texttt{goto} statement gets executed unconditionally, even though it is indented as if it were part of the preceding \texttt{if} statement. As a result, the call to \texttt{sslRawVerify()} (which would perform the actual signature verification) becomes dead code. [ImperialViolet 2014]

Related Guidelines

<table>
<thead>
<tr>
<th>SEI CERT C++ Coding Standard</th>
<th>VOID MSC07-CPP. Detect and remove dead code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO/IEC TR 24772</td>
<td>Unspecified functionality [BVQ] Dead and deactivated code [XYQ]</td>
</tr>
<tr>
<td>MISRA C:2012</td>
<td>Rule 2.1 (required)</td>
</tr>
</tbody>
</table>
Bibliography

[Fortify 2006] Code Quality, "Dead Code"