STR30-C. Do not attempt to modify string literals

According to the C Standard, 6.4.5, paragraph 3 [ISO/IEC 9899:2011]:

A character string literal is a sequence of zero or more multibyte characters enclosed in double-quotes, as in "xyz". A UTF8 string literal is the same, except prefixed by u8. A wide string literal is the same, except prefixed by the letter L, U, or U.

At compile time, string literals are used to create an array of static storage duration of sufficient length to contain the character sequence and a terminating null character. String literals are usually referred to by a pointer to (or array of) characters. Ideally, they should be assigned only to pointers to (or arrays of) const char of const wchar_t. It is unspecified whether these arrays of string literals are distinct from each other. The behavior is undefined if a program attempts to modify any portion of a string literal. Modifying a string literal frequently results in an access violation because string literals are typically stored in read-only memory. (See undefined behavior 33.)

Avoid assigning a string literal to a pointer to non-const or casting a string literal to a pointer to non-const. For the purposes of this rule, a pointer to (or array of) const characters must be treated as a string literal. Similarly, the returned value of the following library functions must be treated as a string literal if the first argument is a string literal:

- `strpbrk()`, `strchr()`, `strchr()`, `strstr()`
- `wcsrbrk()`, `wcschr()`, `wcschr()`, `wcsstr()`
- `memchr()`, `wmemchr()`

This rule is a specific instance of EXP40-C. Do not modify constant objects.

Noncompliant Code Example

In this noncompliant code example, the `char` pointer `str` is initialized to the address of a string literal. Attempting to modify the string literal is undefined behavior:

```c
char *str = "string literal";
str[0] = 'S';
```

Compliant Solution

As an array initializer, a string literal specifies the initial values of characters in an array as well as the size of the array. (See STR11-C. Do not specify the bound of a character array initialized with a string literal.) This code creates a copy of the string literal in the space allocated to the character array `str`. The string stored in `str` can be modified safely.

```c
char str[] = "string literal";
str[0] = 'S';
```

Noncompliant Code Example (POSIX)

In this noncompliant code example, a string literal is passed to the (pointer to non-const) parameter of the POSIX function `mkstemp()`, which then modifies the characters of the string literal:

```c
#include <stdlib.h>

void func(void) {
    mkstemp("/tmp/edXXXXXX");
}
```

The behavior of `mkstemp()` is described in more detail in FIO21-C. Do not create temporary files in shared directories.

Compliant Solution (POSIX)

This compliant solution uses a named array instead of passing a string literal:

```c
#include <stdlib.h>

void func(void) {
    char str[] = "string literal";
    str[0] = 'S';
}
```
```c
#include <stdlib.h>

void func(void) {
    static char fname[] = "/tmp/edXXXXXX";
    mkstemp(fname);
}
```

### Noncompliant Code Example (Result of `strrchr()`)  

In this noncompliant example, the `char *` result of the `strrchr()` function is used to modify the object pointed to by `pathname`. Because the argument to `strrchr()` points to a string literal, the effects of the modification are undefined.

```c
#include <stdio.h>
#include <string.h>

const char *get_dirname(const char *pathname) {
    char *slash;
    slash = strrchr(pathname, '/');
    if (slash) {
        *slash = '\0'; /* Undefined behavior */
    }
    return pathname;
}

int main(void) {
    puts(get_dirname(__FILE__));
    return 0;
}
```

### Compliant Solution (Result of `strrchr()`)  

This compliant solution avoids modifying a `const` object, even if it is possible to obtain a non-`const` pointer to such an object by calling a standard C library function, such as `strrchr()`. To reduce the risk to callers of `get_dirname()`, a buffer and length for the directory name are passed into the function. It is insufficient to change `pathname` to require a `char *` instead of a `const char *` because conforming compilers are not required to diagnose passing a string literal to a function accepting a `char *`.

```c
#include <stddef.h>
#include <stdio.h>
#include <string.h>

char *get_dirname(const char *pathname, char *dirname, size_t size) {
    const char *slash;
    slash = strrchr(pathname, '/');
    if (slash) {
        ptrdiff_t slash_idx = slash - pathname;
        if ((size_t)slash_idx < size) {
            memcpy(dirname, pathname, slash_idx);
            dirname[slash_idx] = '\0';
        }
    }
    return 0;
}

int main(void) {
    char dirname[260];
    if (get_dirname(__FILE__, dirname, sizeof(dirname))) {
        puts(dirname);
    }
    return 0;
}
```

### Risk Assessment


Modifying string literals can lead to abnormal program termination and possibly denial-of-service attacks.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR30-C</td>
<td>Low</td>
<td>Likely</td>
<td>Low</td>
<td>P9</td>
<td>L2</td>
</tr>
</tbody>
</table>

Automated Detection

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Astére</td>
<td>19.04</td>
<td>string-literal-modification write-to-string-literal</td>
<td>Fully checked</td>
</tr>
<tr>
<td>Axivion Bauhaus Suite</td>
<td>6.9.0</td>
<td>CertC-STR30</td>
<td>Fully implemented</td>
</tr>
<tr>
<td>Compass/ROSE</td>
<td></td>
<td></td>
<td>Can detect simple violations of this rule</td>
</tr>
<tr>
<td>Coverity</td>
<td>2017.07</td>
<td>PW</td>
<td>Deprecates conversion from a string literal to &quot;char **&quot;</td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>157 S</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>Parsoft C/C++test</td>
<td>10.4.2</td>
<td>CERT_C-STR30-a</td>
<td>A string literal shall not be modified</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CERT_C-STR30-b</td>
<td>Do not modify string literals</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2019b</td>
<td>CERT C: Rule STR30-C</td>
<td>Checks for writing to const qualified object (rule fully covered)</td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>0556, 0752, 0753, 0754</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>PRQA QA-C++</td>
<td>4.4</td>
<td>3063, 3064, 3065, 3066, 3067, 3084</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>PVS-Studio</td>
<td>7.07</td>
<td>V675</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>RuleChecker</td>
<td>19.04</td>
<td>string-literal-modification</td>
<td>Partially checked</td>
</tr>
<tr>
<td>Splint</td>
<td>3.1.1</td>
<td></td>
<td>Exhaustively verified (see one compliant and one non-compliant example)</td>
</tr>
<tr>
<td>TrustInSoft Analyzer</td>
<td>1.38</td>
<td>mem_access</td>
<td></td>
</tr>
</tbody>
</table>

Related Vulnerabilities

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

Related Guidelines

Key here (explains table format and definitions)

<table>
<thead>
<tr>
<th>Taxonomy</th>
<th>Taxonomy item</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>CERT C Secure Coding Standard</td>
<td>EXP05-C. Do not cast away a const qualification</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>CERT C Secure Coding Standard</td>
<td>STR11-C. Do not specify the bound of a character array initialized with a string literal</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
</tbody>
</table>

Bibliography


[Summit 1995] comp.lang.c FAQ List, Question 1.32