ERR34-C. Detect errors when converting a string to a number

The process of parsing an integer or floating-point number from a string can produce many errors. The string might not contain a number. It might contain a number of the correct type that is out of range (such as an integer that is larger than INT_MAX). The string may also contain extra information after the number, which may or may not be useful after the conversion. These error conditions must be detected and addressed when a string-to-number conversion is performed using a C Standard Library function.

The `strtol()`, `strtoll()`, `strtoimax()`, `strtof()`, `strtod()`, and `strtold()` functions convert the initial portion of a null-terminated byte string to a `long int`, `long long int`, `intmax_t`, `unsigned long int`, `unsigned long long int`, `float`, `double`, and `long double representation`, respectively.

Use one of the C Standard Library `strto*()` functions to parse an integer or floating-point number from a string. These functions provide more robust error handling than alternative solutions. Also, use the `strtol()` function to convert to a smaller signed integer type such as `signed int`, `signed short`, and `signed char`, testing the result against the range limits for that type. Likewise, use the `strtoul()` function to convert to a smaller unsigned integer type such as `unsigned int`, `unsigned short`, and `unsigned char`, and test the result against the range limits for that type. These range tests do nothing if the smaller type happens to have the same size and representation for a particular implementation.

Noncompliant Code Example (`atoi()`)  
This noncompliant code example converts the string token stored in the `buff` to a signed integer value using the `atoi()` function:

```c
#include <stdlib.h>

void func(const char *buff) {
    int si;
    if (buff) {
        si = atoi(buff);
    } else {
        /* Handle error */
    }
}
```

Unfortunately, `atoi()` and related functions lack a mechanism for reporting errors for invalid values. Specifically, these functions:

- do not need to set `errno` on an error;
- have `undefined behavior` if the value of the result cannot be represented;
- return 0 (or 0.0) if the string does not represent an integer (or decimal), which is indistinguishable from a correctly formatted, zero-denoting input string.

Noncompliant Example (`sscanf()`)  
This noncompliant example uses the `sscanf()` function to convert a string token to an integer. The `sscanf()` function has the same limitations as `atoi()`:
#include <stdio.h>

void func(const char *buff) {
    int matches;
    int si;

    if (buff) {
        matches = sscanf(buff, "%d", &si);
        if (matches != 1) {
            /* Handle error */
        } else {
            /* Handle error */
        }
    } else {
        /* Handle error */
    }
}

The `sscanf()` function returns the number of input items successfully matched and assigned, which can be fewer than provided for, or even 0 in the event of an early matching failure. However, `sscanf()` fails to report the other errors reported by `strtol()`, such as numeric overflow.

**Compliant Solution (strtol())**

The `strtol()`, `strtoul()`, `strtolmax()`, `strtoull()`, `strtof()`, `strtol()`, and `strtol()()` functions convert a null-terminated byte string to `long int`, `long long int`, `intmax_t`, `unsigned long int`, `unsigned long long int`, `uintmax_t`, `float`, `double`, and `long double` representation, respectively.

This compliant solution uses `strtol()` to convert a string token to an integer and ensures that the value is in the range of `int`:

```
#include <errno.h>
#include <limits.h>
#include <stdlib.h>
#include <stdio.h>

void func(const char *buff) {
    char *end;
    int si;
    errno = 0;

    const long sl = strtol(buff, &end, 10);
    if (end == buff) {
        fprintf(stderr, "\s: not a decimal number\n", buff);
    } else if (*end != 0) {
        fprintf(stderr, "\s: extra characters at end of input: \n", buff, end);
    } else if (LONG_MIN == sl || LONG_MAX == sl) && ERANGE == errno) {
        fprintf(stderr, "\s out of range of type long\n", buff);
    } else if (sl > INT_MAX) {
        fprintf(stderr, "%ld greater than INT_MAX\n", sl);
    } else if (sl < INT_MIN) {
        fprintf(stderr, "%ld less than INT_MIN\n", sl);
    } else {
        si = (int)sl;
        /* Process si */
    }
}
```

**Risk Assessment**

It is rare for a violation of this rule to result in a security vulnerability unless it occurs in security-sensitive code. However, violations of this rule can easily result in lost or misinterpreted data.

<table>
<thead>
<tr>
<th>Recommendation</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERR34-C</td>
<td>Medium</td>
<td>Unlikely</td>
<td>Medium</td>
<td>P4</td>
<td>L3</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>Axivion Bauhaus Suite</td>
<td>7.2.0</td>
<td>CertC-ERR34</td>
<td></td>
</tr>
<tr>
<td>Clang</td>
<td>3.9</td>
<td>cert-err34-c</td>
<td>Checked by clang-tidy</td>
</tr>
<tr>
<td>CodeSonar</td>
<td>6.2p0</td>
<td>BADFUNC.ATOF</td>
<td>Use of atof</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BADFUNC.ATOI</td>
<td>Use of atoi</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BADFUNC.ATOL</td>
<td>Use of atol</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BADFUNC.ATOLL</td>
<td>Use of atoll</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(customization)</td>
<td>Users can add custom checks for uses of other undesirable conversion functions.</td>
</tr>
<tr>
<td>Compass/ROSE</td>
<td></td>
<td></td>
<td>Can detect violations of this recommendation by flagging invocations of the following functions:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• atoi()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• scanf(), fscanf(), sscanf()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Others?</td>
</tr>
<tr>
<td>Helix QAC</td>
<td>2021.3</td>
<td>C5030</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C++5016</td>
<td></td>
</tr>
<tr>
<td>Klocwork</td>
<td>2021.4</td>
<td>CERT.ERR.CONV.STR_TO_NUM</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MISRA.STDLIB.ATOI</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SV.BANNED.RECOMMENDED.SCANF</td>
<td></td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>44 S</td>
<td>Fully implemented</td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>2021.2</td>
<td>CERT_C-ERR34-a</td>
<td>The library functions atof, atoi and atol from library stdlib.h shall not be used</td>
</tr>
<tr>
<td>PC-int Plus</td>
<td>1.4</td>
<td>586</td>
<td>Assistance provided</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2021a</td>
<td>CERT C: Rule ERR34-C</td>
<td>Checks for unsafe conversion from string to numeric value (rule fully covered)</td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>5030</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>PRQA QA-C++</td>
<td>4.4</td>
<td>5016</td>
<td></td>
</tr>
<tr>
<td>SonarQube C/C++ Plugin</td>
<td>3.11</td>
<td>S989</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</td>
<td>INT06-CPP. Use strtol() or a related function to convert a string token to an integer</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>CWE 2.11</td>
<td>CWE-676, Use of potentially dangerous function</td>
<td>2017-05-18: CERT: Rule subset of CWE</td>
</tr>
<tr>
<td>CWE 2.11</td>
<td>CWE-758, Use of potentially dangerous function</td>
<td>2017-06-29: CERT: Partial overlap</td>
</tr>
</tbody>
</table>

CERT-CWE Mapping Notes

Key here for mapping notes

CWE-20 and ERR34-C

Intersection( ERR34-C, CWE-20) = Ø

CERT C does not define the concept of ‘input validation’. String-to-integer conversion (ERR34-C) may qualify as input validation, but this is outside the scope of the CERT rule.
CWE-391 and ERR34-C

CWE-391 = Union( ERR34-C, list) where list =

- Failure to errors outside of string-to-number conversion functions

CWE-676 and ERR34-C

- Independent( ENV33-C, CON33-C, STR31-C, EXP33-C, MSC30-C, ERR34-C)

- ERR34-C implies that string-parsing functions (e.g., atoi() and scanf()) are dangerous.

- CWE-676 = Union( ERR34-C, list) where list =

  - Invocation of dangerous functions besides the following:
    - atoi(), atol(), atoll(), atof(), The scanf() family

CWE-758 and ERR34-C


Intersection( CWE-758, ERR34-C) =

- Undefined behavior arising from a non-representable numeric value being parsed by an ato*() or scanf() function

CWE-758 – ERR34-C =

- Undefined behavior arising from using a function outside of the ato*() or scanf() family

ERR34-C – CWE-758 =

- The ato*() or scanf() family receives input that is not a number when trying to parse one

Bibliography

<table>
<thead>
<tr>
<th>ISO/IEC 9899:2011</th>
<th>Subclause 7.22.1, &quot;Numeric conversion functions&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Subclause 7.21.6, &quot;Formatted input/output functions&quot;</td>
</tr>
<tr>
<td>Klein 2002</td>
<td></td>
</tr>
</tbody>
</table>