ERR02-C. Avoid in-band error indicators

Avoid in-band error indicators while designing interfaces. This practice is commonly used by C library functions but is not recommended. One example from the C Standard of a troublesome in-band error indicator is EOF (see FIO34-C. Distinguish between characters read from a file and EOF or WEOF). Another problematic use of in-band error indicators from the C Standard involving the size_t and time_t types is described by

- INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data
- FLP07-C. Cast the return value of a function that returns a floating-point type
- INT18-C. Evaluate integer expressions in a larger size before comparing or assigning to that size

Noncompliant Code Example (sprintf())

This noncompliant code example is from the Linux Kernel Mailing List archive site, although similar examples are common:

```c
int i;
size_t count = 0;
for (i = 0; i < 9; ++i) {
    count += sprintf(buf + count, "%02x ", ((u8 *)&slreg_num)[i]);
}
count += sprintf(buf + count, "\n");
```

The `sprintf()` function returns the number of characters written in the array, not counting the terminating null character. This number is frequently added to an existing counter to keep track of the location of the index into the array. However, the call to `sprintf()` can (and will) return 1 on error conditions, such as an encoding error. If this error happens on the first call (which is likely), the count variable, already at 0, is decremented. If this index is subsequently used, it will result in an out-of-bounds read or write.

Compliant Solution (sprintf_m())

This compliant solution shows the redesigned API for `sprintf()` from the CERT managed string library [Burch 2006]:

```c
erro_t sprintf_m(
    string_m buf,
    const string_m fmt,
    int *count,
    ...
);
```

The `sprintf_m()` API separates the return status of the function from information about the number of characters written. In this case, *count is set to the number of characters written in `buf`, and the return value indicates the return status. Returning the status as the return value of the function increases the likelihood that a programmer will check the return status of the function.

The preceding code example can be amended as follows:

```c
int i;
size_t count = 0;
erro_t err;
for (i = 0; i < 9; ++i) {
    err = sprintf_m(
        buf + count, "%02x ", &count, ((u8 *)&slreg_num)[i]);
    if (err != 0) {
        /* Handle print error */
    }
}
erro_t err = sprintf_m(
    buf + count, "%02x ", &count, ((u8 *)&slreg_num)[i]);
    if (err != 0) {
        /* Handle print error */
}
```
Noncompliant Code Example (POSIX ssize_t)

The ssize_t data type is designed as a "signed representation of size_t." Consequently, it is often used as a return type for functions that can return an unsigned value upon success and a negative value upon error. For instance, the POSIX read() function has the following signature:

```c
ssize_t read(int fildes, void *buf, size_t nbyte);
```

read() returns 1 if an error occurs; if no errors occur, it returns the number of bytes actually read.

As with all in-band error indicators, this type is not recommended because developers are tempted to ignore the possibility that a ssize_t value is negative.

Compliant Solution (POSIX size_t)

An alternative hypothetical signature for the read() function is

```c
erro_t read(int fildes, void *buf, size_t nbyte, size_t* rbytes);
```

where rbytes is a pointer to a size_t. If no error occurs, and rbytes is not NULL, its value is set to the total number of bytes read, and read() returns 0. If an error occurs, read() returns a nonzero value indicating the error.

Noncompliant Code Example (C11, Annex K)

In this noncompliant code example, the error handler returns normally, but the strcpy_s() function's return value is not checked:

```c
constraint_handler_t handle_errors(void) {
    constraint_handler_t data;
    /* Define what to do when error occurs */
    return data;
}

/* ... */
set_constraint_handler(handle_errors);

/* ... */

// Returns zero on success */
erro_t function(char *dst1, size_t dst_size) {
    char src1[100] = "hello";
    strcpy_s(dst1, dst_size, src1);
    /*
     * At this point strcpy_s may have yielded an
     * error, and handle_errors() might have returned.
     */
    /* ... */
    return 0;
}
```

Compliant Solution (C11, Annex K)

In this compliant solution, the error handler terminates the program, ensuring that strcpy_s() never returns unless it fully succeeds:

```c
constraint_handler_t handle_errors(void) {
    constraint_handler_t data;
    /* Define what to do when error occurs */
    return data;
}

/* ... */
set_constraint_handler(handle_errors);

/* ... */

// Returns zero on success */
erro_t function(char *dst1, size_t dst_size) {
    char src1[100] = "hello";
    strcpy_s(dst1, dst_size, src1);
    /*
     * At this point strcpy_s may have yielded an
     * error, and handle_errors() might have returned.
     */
    /* ... */
    return 0;
}
```
/* The abort_handler_s() function writes 
* a message on the standard error stream and 
* then calls the abort() function. 
*/
set_constraint_handler(abort_handler_s);

/* ... */

/* Returns zero on success */
errno_t function(char *dst1, size_t dst_size) {
    char src[100] = "hello";
    strcpy_s(dst1, dst_size, src);
    /*
    * Because abort_handler_s() never returns,
    * we get here only if strcpy_s() succeeds.
    */
    /* ... */
    return 0;
}

Exceptions

ERR02-EX1: Null pointers are another example of an in-band error indicator. Use of null pointers is allowed because it is supported by the language. According to the C Standard, subclause 6.3.2.3 [ISO/IEC 9899:2011]:

If a null pointer constant is converted to a pointer type, the resulting pointer, called a null pointer, is guaranteed to compare unequal to a pointer to any object or function.

ERR02-EX2: You may use a function returning in-band error indicators if you can securely guarantee the program will not try to continue processing should an error occur in the function. For example, the functions defined in C11 Annex K provide hooks for internal constraint violations. If a constraint violation handler is guaranteed not to return upon an error, then you may safely ignore errors returned by these functions. You might accomplish this by having the constraint-violation handler call abort() or longjmp(), for instance.

See ERR03-C. Use runtime-constraint handlers when calling the bounds-checking interfaces for more on the functions defined in C11 Annex K.

Risk Assessment

The risk in using in-band error indicators is difficult to quantify and is consequently given as low. However, if the use of in-band error indicators results in programmers' failing to check status codes or incorrectly checking them, the consequences can be more severe.

<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>ERR02-C</td>
<td>Low</td>
<td>Unlikely</td>
<td>High</td>
<td>P1</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>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>CERT-C-ERR02-a</td>
<td>The Standard Library input/output functions shall not be used</td>
</tr>
</tbody>
</table>

Related Vulnerabilities

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

Related Guidelines

| SEI CERT C++ Coding Standard | VOID ERR02-CPP. Avoid in-band error indicators |

Bibliography
| [Burch 2006] | Subclause 6.3.2, "Other Operands"