MISRA C:2012 Amendment 1

Additional security guidelines for MISRA C:2012

April 2016

First published April 2016 by HORIBA MIRA Limited
Watling Street
Nuneaton
Warwickshire
CV10 0TU
UK

www.misra.org.uk

ISBN 978-1-906400-16-3 PDF

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

MISRA C:2012 Amendment 1

Additional security guidelines for MISRA C:2012

April 2016

MISRA Mission Statement

We provide world-leading best practice guidelines for the safe and secure application of both embedded control systems and standalone software.

MISRA is a collaboration between manufacturers, component suppliers and engineering consultancies which seeks to promote best practice in developing safety- and security-related electronic systems and other software-intensive applications. To this end MISRA publishes documents that provide accessible information for engineers and management, and holds events to permit the exchange of experiences between practitioners.

Foreword

The vision of MISRA C is set out in the opening paragraph of the Guidelines:

> The MISRA C Guidelines define a subset of the C language in which the opportunity to make mistakes is either removed or reduced.
> Many standards for the development of *safety-related software* require, or recommend, the use of a language subset, and this can also be used to develop any application with *high integrity or high reliability requirements*.

Unfortunately, many people focus on the *safety-related software* reference, and a perception exists that MISRA C is only *safety-related* and not *security-related*.

Subsequent to the publication of MISRA C:2012, ISO/IEC JTC1/SC22/WG14 (the committee responsible for maintaining the C Standard) published their own C language Security Guidelines, as ISO/IEC 17961:2013.

Addendum 2 to MISRA C:2012 sets out the coverage by MISRA C:2012 of ISO/IEC 17961:2013 and justifies the viewpoint that MISRA C is equally applicable in a *security-related* environment as it is in a *safety-related* one. The work to create that matrix highlighted a small number of areas where MISRA C could be enhanced.

This Amendment to MISRA C:2012 sets out a small number of additional guidelines, to improve the coverage of the security concerns highlighted by the ISO C Secure Guidelines. Several of these address specific issues pertaining to the use of untrustworthy data, a well-known security vulnerability.

These additional Guidelines extend MISRA C:2012 and I encourage all users, and all organizations, to consider adoption at the earliest opportunity.

Andrew Banks FBCS CITP
Chairman, MISRA C Working Group

Acknowledgements

The MISRA consortium would like to thank the following individuals for their significant contribution to the writing of this document:

Andrew Banks	Frazer-Nash Research Ltd / Intuitive Consulting
Mike Hennell	LDRA Ltd
Clive Pygott	Columbus Computing Ltd
Chris Tapp	LDRA Ltd
Liz Whiting	LDRA Ltd

The MISRA consortium also wishes to acknowledge contributions from the following members of the MISRA C Working Group during the development and review process:

Fulvio Baccaglini	Programming Research Ltd
Dave Banham	Rolls-Royce plc
Mark Bradbury	Independent Consultant (Formerly Aero Engine Controls)
Jill Britton	Programming Research Ltd
Chris Hills	Phaedrus Systems Ltd

The MISRA consortium also wishes to acknowledge contributions from the following individuals during the development and review process:

David Ward	HORIBA MIRA Ltd

Contents

1 New directives

1.4 Code design

Dir 4.14 The validity of values received from external sources shall be checked

C90 [Undefined 15, 19, 26, 30, 31, 32, 94]
C99 [Undefined 15, 16, 33, 40, 43-45, 48, 49, 113]

Category Required

Applies to C90, C99

Amplification

"External sources" include data:

- Read from a file;
- Read from an environment variable;
- Resulting from user input;
- Received over a communications channel.

Rationale

A program has no control over the values given to data originating from external sources. The values may therefore be invalid, either as the result of errors or due to malicious modification by an external agent. Data from external sources shall therefore be validated before it is used.

In the security domain, external sources of data are usually regarded as untrusted as they may have been modified by someone trying to harm or gain control of the program and/or system it is running on; such data needs to be validated before it can be used safely.

In the safety domain, external sources are regarded as "suspicious" and values obtained from them require validation.

In both domains, data from an external source shall be tested to ensure that its value respects all the constraints placed on its use (i.e. its value is not harmful), even if the value cannot be proven to be correct. For example:

- A value used to compute an array index shall not result in an array bounds error;
- A value used to control a loop shall not cause excessive (e.g. infinite) iteration;
- A value used to compute a divisor shall not result in division by zero;
- A value used to compute an amount of dynamic memory shall not result in excessive memory allocation;
- A string used as a query to an SQL database shall be checked to ensure that it does not include a `;` character.

Example

The following example is non-compliant as there is no check made to ensure that a string resulting from user input is null terminated. This may lead to an array bounds error, commonly known as a buffer overrun, when the string is output through the call to *printf*.

```
void f1( void )
{
  char input [ 128 ];

  ( void ) scanf ( "%128c", input );

  ( void ) printf ( "%s", input );      /* Non-compliant */
}
```

2 New rules

2.12 Expressions

Rule 12.5	The *sizeof* operator shall not have an operand which is a function parameter declared as “array of type”

Category	Mandatory
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

The function parameter `A` in `void f ( int32_t A[ 4 ] )` is declared as “array of type”.

Rationale

The *sizeof* operator can be used to determine the number of elements in an array `A`:

```
size_t arraySize = sizeof ( A ) / sizeof ( A[ 0 ] );
```

This works as expected when `A` is an identifier that designates an array, as it has type “array of type”. It does not “degenerate” to a pointer and `sizeof ( A )` returns the size of the array.

However, this is not the case when `A` is a function parameter. The Standard states that a function parameter never has type “array of type” and a function parameter declared as an array will “degenerate” to “pointer to type”. This means that `sizeof ( A )` is equivalent to `sizeof ( int32_t * )`, which does not return the size of an array.

Example

```
int32_t glbA[] = { 1, 2, 3, 4, 5 };

void f ( int32_t A[ 4 ] )
{
  /*
   * The following is non-compliant as it always gives the same answer,
   * irrespective of the number of members that appear to be in the array
   * (4 in this case), because A has type int32_t * and not int32_t[ 4 ].
   * As sizeof ( int32_t * ) is often the same as sizeof ( int32_t ),
   * numElements is likely to always have the value 1.
   */
  uint32_t numElements = sizeof ( A ) / sizeof ( int32_t );

  /*
   * The following is compliant as numElements_glbA will be given the
   * expected value of 5.
   */
  uint32_t numElements_glbA = sizeof ( glbA ) / sizeof ( glbA[ 0 ] );
}
```

2.21 Standard libraries

Rule 21.13 Any value passed to a function in <ctype.h> shall be representable as an *unsigned char* or be the value EOF

C90 [Undefined 63], C99 [Undefined 107]

Category Mandatory

Analysis Undecidable, System

Applies to C90, C99

Rationale

The relevant functions from <ctype.h> are defined to take an *int* argument where the expected value is either in the range of an *unsigned char* or is a negative value equivalent to EOF. The use of any other values results in *undefined behaviour*.

Example

Note: The int casts in the following example are required to comply with Rule 10.3.

```
bool_t f ( uint8_t a )
{
  return (    isdigit ( ( int32_t )  a  )       /* Compliant     */
           && isalpha ( ( int32_t ) 'b' )       /* Compliant     */
           && islower (             EOF )       /* Compliant     */
           && isalpha (             256 ) );    /* Non-compliant */
}
```

See also

Rule 10.3

Rule 21.14 The Standard Library function *memcmp* shall not be used to compare null terminated strings

Category Required

Analysis Undecidable, System

Applies to C90, C99

Amplification

For the purposes of this rule, "null terminated strings" are:

- String literals;
- Arrays having *essentially character type* which contain a *null character*.

Rationale

The Standard Library function

```
int memcmp ( const void *s1, const void *s2, size_t n );
```

performs a byte by byte comparison of the first n bytes of the two objects pointed at by s1 and s2.

If *memcmp* is used to compare two strings and the length of either is less than `n`, then they may compare as different even when they are logically the same (i.e. each has the same sequence of characters before the null terminator) as the characters after a null terminator will be included in the comparison even though they do not form part of the string.

Example

```
extern char buffer1[ 12 ];
extern char buffer2[ 12 ];

void f1 ( void )
{
  ( void ) strcpy ( buffer1, "abc" );
  ( void ) strcpy ( buffer2, "abc" );

  /* The following use of memcmp is non-compliant */
  if ( memcmp ( buffer1, buffer2, sizeof ( buffer1 ) ) != 0 )
  {
    /*
     * The strings stored in buffer1 and buffer 2 are reported to be
     * different, but this may actually be due to differences in the
     * uninitialised characters stored after the null terminators.
     */
  }
}

/* The following definition violates other guidelines */
unsigned char headerStart[ 6 ] = { 'h', 'e', 'a', 'd', 0, 164 };

void f2 ( const uint8_t *packet )
{
  /* The following use of memcmp is compliant */
  if ( ( NULL != packet ) && ( memcmp( packet, headerStart, 6 ) == 0 ) )
  {
    /*
     * Comparison of values having essentially unsigned type reports that
     * contents are the same. Any null terminator is simply treated as a
     * zero value and any differences beyond it are significant.
     */
  }
}
```

See also

Rule 21.15, Rule 21.16

Rule 21.15 The pointer arguments to the Standard Library functions *memcpy*, *memmove* and *memcmp* shall be pointers to qualified or unqualified versions of compatible types

Category Required

Analysis Decidable, Single Translation Unit

Applies to C90, C99

Rationale

The Standard Library functions

```
void * memcpy  ( void * restrict s1, const void * restrict s2, size_t n );
void * memmove ( void *s1, const void *s2, size_t n );
 int   memcmp  ( const void *s1, const void *s2, size_t n );
```

perform a byte by byte copy, move or comparison of the first `n` bytes of the two objects pointed at by `s1` and `s2`.

An attempt to call one of these functions with arguments which are pointers to different types may indicate a mistake.

Example

```
/*
 * Is it intentional to only copy part of 's2'?
 */
void f ( uint8_t s1[ 8 ], uint16_t s2[ 8 ] )
{
  ( void ) memcpy ( s1, s2, 8 );    /* Non-compliant */
}
```

See also

Rule 21.14, Rule 21.16

Rule 21.16 The pointer arguments to the Standard Library function *memcmp* shall point to either a pointer type, an *essentially signed* type, an *essentially unsigned* type, an *essentially Boolean* type or an *essentially* enum type

C99 [Unspecified 9]

Category Required

Analysis Decidable, Single Translation Unit

Applies to C90, C99

Rationale

The Standard Library function

```
int memcmp ( const void *s1, const void *s2, size_t n );
```

performs a byte by byte comparison of the first `n` bytes of the two objects pointed at by `s1` and `s2`.

Structures shall not be compared using *memcmp* as it may incorrectly indicate that two structures are not equal, even when their members hold the same values. Structures may contain padding with an indeterminate value between their members and *memcmp* will include this in its comparison. It cannot be assumed that the padding will be equal, even when the values of the structure members are the same. Unions have similar concerns along with the added complication that they may incorrectly be reported as having the same value when the representation of different, overlapping members are coincidentally the same.

Objects with *essentially floating* type shall not be compared with *memcmp* as the same value may be stored using different representations.

If an *essentially char* array contains a null character, it is possible to treat the data as a character string rather than simply an array of characters. However that distinction is a matter of interpretation rather than syntax. Since *essentially char* arrays are most frequently used to store character strings, an attempt to compare such arrays using *memcmp* (rather than *strcmp* or *strncmp*) may indicate an error as the number of characters to be compared will be determined by the value of the `size_t` argument rather than the location of the null characters used to terminate the strings. The result may therefore depend on the comparison of characters which are not part of the respective strings.

Example

```
struct S;

/*
 * Return value may indicate that 's1' and 's2' are different due to padding.
 */
bool_t f1 ( struct S *s1, struct S *s2 )
{
  return ( memcmp ( s1, s2, sizeof ( struct S ) ) != 0 );    /* Non-compliant */
}

union U
{
  uint32_t range;
  uint32_t height;
};

/*
 * Return value may indicate that 'u1' and 'u2' are the same
 * due to unintentional comparison of 'range' and 'height'.
 */
bool_t f2 ( union U *u1, union U *u2 )
{
  return ( memcmp ( u1, u2, sizeof ( union U ) ) != 0 );    /* Non-compliant */
}

const char a[ 6 ] = "task";

/*
 * Return value may incorrectly indicate strings are different as the
 * length of 'a' (4) is less than the number of bytes compared (6).
 */
bool_t f3 ( const char b[ 6 ] )
{
  return ( memcmp ( a, b, 6 ) != 0 );    /* Non-compliant */
}
```

See also

Rule 21.14, Rule 21.15

Rule 21.17 Use of the string handling functions from <string.h> shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters

C90 [Undefined 96], C99 [Undefined 103, 180]

Category	Mandatory
Analysis	Undecidable, System
Applies to	C90, C99

Amplification

The relevant string handling functions from <string.h> are:

```
strcat, strchr, strcmp, strcoll, strcpy, strcspn, strlen, strpbrk, strrchr, strspn,
strstr, strtok
```

Rationale

Incorrect use of a function listed above may result in a read or write access beyond the bounds of an object passed as a parameter, resulting in *undefined behaviour*.

Example

```
char string[] = "Short";

void f1 ( const char *str )
{
  /*
   * Non-compliant use of strcpy as it results in writes beyond the end of 'string'
   */
  ( void ) strcpy ( string, "Too long to fit" );

  /*
   * Compliant use of strcpy as 'string' is only modified if 'str' will fit.
   */
  if ( strlen ( str ) < ( sizeof ( string ) - 1u ) )
  {
    ( void ) strcpy ( string, str );
  }
}


size_t f2 ( void )
{
  char text[ 5 ] = "Token";

  /*
   * The following is non-compliant as it results in reads beyond
   * the end of 'text' as there is no null terminator.
   */
  return strlen ( text );
}
```

See also

Rule 21.18

Rule 21.18 The `size_t` argument passed to any function in `<string.h>` shall have an appropriate value

C90 [Undefined 96], C99 [Undefined 103, 180, 181]

Category	Mandatory
Analysis	Undecidable, System
Applies to	C90, C99

Amplification

The relevant functions in <string.h> are:

`memchr, memcmp, memcpy, memmove, memset, strncat, strncmp, strncpy, strxfrm`

An appropriate value is:

- Positive;
- No greater than the size of the smallest object passed to the function through a pointer parameter.

Rationale

Incorrect use of a function listed above may result in a read or write access beyond the bounds of an object passed as a parameter, resulting in *undefined behaviour*.

Example

```
char buf1[  5 ] = "12345";
char buf2[ 10 ] = "1234567890";

void f ( void )
{
  if ( memcmp ( buf1, buf2, 5 ) == 0 )                  /* Compliant     */
  {
  }

  if ( memcmp ( buf1, buf2, 6 ) == 0 )                  /* Non-compliant */
  {
  }
}
```

See also

Rule 21.17

Rule 21.19 The pointers returned by the Standard Library functions *localeconv*, *getenv*, *setlocale* or, *strerror* shall only be used as if they have pointer to const-qualified type

C90 [Undefined], C99 [Undefined 114, 115, 174]

Category	Mandatory
Analysis	Undecidable, System
Applies to	C90, C99

Amplification

The *localeconv* function returns a pointer of type `struct lconv *`. This pointer shall be regarded as if it had type `const struct lconv *`.

A `struct lconv` object includes pointers of type `char *` and the *getenv*, *setlocale*, and *strerror* functions each return a pointer of type `char *`. These pointers are used to access strings (null terminated arrays of type *char*). For the purpose of this rule, these pointers shall be regarded as if they had type `const char *`.

Rationale

The Standard states that *undefined behaviour* occurs if a program modifies:

- The structure pointed to by the value returned by *localeconv*;
- The strings returned by *getenv*, *setlocale* or *strerror*.

Note: The Standard does not specify the behaviour that results if the strings referenced by the structure pointed to by the value returned by *localeconv* are modified. This rule prohibits any changes to these strings as they are considered to be undesirable.

Treating the pointers returned by the various functions as if they were const-qualified allows an analysis tool to detect any attempt to modify an object through one of the pointers. Additionally, assigning the return values of the functions to const-qualified pointers will result in the compiler issuing a diagnostic if an attempt is made to modify an object.

Note: If a modified version is required, a program should make and modify a copy of any value covered by this rule.

Example

The following examples are non-compliant as the returned pointers are assigned to non-const qualified pointers. Whilst this will not be reported by a compiler (it is not a constraint violation), an analysis tool will be able to report a violation.

```
void f1 ( void )
{
  char         *s1   = setlocale ( LC_ALL, 0 );   /* Non-compliant       */
  struct lconv *conv = localeconv ();             /* Non-compliant       */

  s1[ 1 ]             = 'A';                      /* Undefined behaviour */
  conv->decimal_point = "^";                      /* Undefined behaviour */
}
```

The following examples are compliant as the returned pointers are assigned to const qualified pointers. Any attempt to modify an object through a pointer will be reported by a compiler or analysis tool as this is a constraint violation.

```
void f2 ( void )
{
  char str[ 128 ];

  ( void ) strcpy ( str,
                    setlocale ( LC_ALL, 0 ) );    /* Compliant - 2nd parameter to
                                                     strcpy takes a const char *  */
  const struct lconv *conv = localeconv ();       /* Compliant                    */

  conv->decimal_point = "^";                      /* Constraint violation         */
}
```

The following example shows that whilst the use of a const-qualified pointer gives compile time protection of the value returned by *localeconv*, the same is not true for the strings it references. Modification of these strings can be detected by an analysis tool.

```
void f3 ( void )
{
  const struct lconv *conv = localeconv ();       /* Compliant     */

  conv->grouping[ 2 ] = 'x';                      /* Non-compliant */
}
```

See also

Rule 7.4, Rule 11.8, Rule 21.8

Rule 21.20 The pointer returned by the Standard Library functions *asctime*, *ctime*, *gmtime*, *localtime*, *localeconv*, *getenv*, *setlocale* or *strerror* shall not be used following a subsequent call to the same function

Category Mandatory

Analysis Undecidable, System

Applies to C90, C99

Amplification

Calls to *setlocale* may change the values accessible through a pointer that was previously returned by *localeconv*. For the purposes of this rule, the *setlocale* and *localeconv* function shall therefore be treated as if they are the same function.

Rationale

The Standard Library functions *asctime*, *ctime*, *gmtime*, *localtime*, *localeconv*, *getenv*, *setlocale* and *strerror* return a pointer to an object within the Standard Library. Implementations are permitted to use static buffers for any of these objects and a second call to the same function may modify the contents of the buffer. The value accessed through a pointer held by the program before a subsequent call to a function may therefore change unexpectedly.

Example

```
void f1( void )
{
  const char *res1;
  const char *res2;
        char  copy[ 128 ];
```

```
  res1 = setlocale ( LC_ALL, 0 );

  ( void ) strcpy ( copy, res1 );

  res2 = setlocale ( LC_MONETARY, "French" );

  printf ( "%s\n", res1 );   /* Non-compliant - use after subsequent call    */
  printf ( "%s\n", copy );   /* Compliant - copy made before subsequent call */
  printf ( "%s\n", res2 );   /* Compliant - no subsequent call before use    */
}
```

2.22 Resources

Rule 22.7 The macro `EOF` shall only be compared with the unmodified return value from any Standard Library function capable of returning `EOF`

Category	Required
Analysis	Undecidable, System
Applies to	C90, C99

Amplification

The value returned by any of these functions shall not be subject to any type conversion if it is later compared with the macro `EOF`. *Note:* indirect type conversions, such as those resulting from pointer type conversions, are included within the scope of this rule.

Rationale

An `EOF` return value from these functions is used to indicate that a stream is either at end-of-file or that a read or write error has occurred. The `EOF` value may become indistinguishable from a valid character code if the value returned is converted to another type. In such cases, testing the converted value against `EOF` will not reliably identify if the end of the file has been reached or if an error has occurred.

If these conditions are to be identified by comparison with `EOF`, the comparison shall be made before any conversion of the value occurs. Alternatively, the Standard Library functions *feof* and *ferror* may be used to directly check the status of the stream, either before or after the conversion takes place.

Example

```
void f1 ( void )
{
  char ch;

  ch = ( char ) getchar ();

  /*
   * The following test is non-compliant. It will not be reliable as the
   * return value is cast to a narrower type before checking for EOF.
   */
  if ( EOF != ( int32_t ) ch )
  {
  }
}
```

The following compliant example shows how *feof()* can be used to check for `EOF` when the return value from *getchar()* has been subjected to type conversion:

```
void f2 ( void )
{
  char ch;

  ch = ( char ) getchar ();

  if ( !feof ( stdin ) )
  {
  }
}

void f3 ( void )
{
  int32_t i_ch;

  i_ch = getchar ();

  /*
   * The following test is compliant. It will be reliable as the
   * unconverted return value is used when checking for EOF.
   */
  if ( EOF != i_ch )
  {
    char ch;

    ch = ( char ) i_ch;
  }
}
```

Rule 22.8 The value of `errno` shall be set to zero prior to a call to an *errno-setting-function*

Category	Required
Analysis	Undecidable, System
Applies to	C90, C99

Amplification

An *errno-setting-function* is one of the following:

```
ftell, fgetpos, fsetpos, fgetwc, fputwc
strtoimax, strtoumax, strtol, strtoul, strtoll, strtoull, strtof, strtod, strtold
wcstoimax, wcstoumax, wcstol, wcstoul, wcstoll, wcstoull, wcstof, wcstod, wcstold
wcrtomb, wcsrtombs, mbrtowc
```

Any other function which returns error information using `errno` is also an *errno-setting-function*. *Note:* this may include additional functions from the Standard Library, as permitted by The Standard.

"Prior" requires that `errno` shall be set to zero in the same function and on all paths leading to a call of an *errno-setting-function*. Furthermore, there shall be no calls to functions that may set `errno` in these paths. This includes calls to any function within the Standard Library as these are permitted (but not required) to set `errno`.

Rationale

An *errno-setting-function* writes a non-zero value to `errno` if an error is detected, leaving the value unmodified otherwise. The Standard includes non-normative advice that "a program that uses `errno` for error checking should set it to zero before a library function call, then inspect it before a subsequent library function call".

In order that errors can be detected, this rule requires that `errno` shall be set to zero before an *errno-setting-function* is called. Rule 22.9 then requires that `errno` be tested after the call.

Exception

The value of `errno` need not be set to zero when it can be proven to be zero.

Example

```
void f ( void )
{
  errnoSettingFunction1 ();      /* Non-compliant           */

  if ( 0 == errno )
  {
    errnoSettingFunction2 ();    /* Compliant by exception */

    if ( 0 == errno )
    {
    }
  }

  else
  {
    errno = 0;

    errnoSettingFunction3 ();    /* Compliant              */

    if ( 0 == errno )
    {
    }
  }
}
```

See also

Rule 22.9, Rule 22.10

Rule 22.9 The value of `errno` shall be tested against zero after calling an *errno-setting-function*

Category	Required
Analysis	Undecidable, System
Applies to	C90, C99

Amplification

An *errno-setting-function* is one of those described in Rule 22.8.

The test of `errno` shall occur in the same function on all paths from the call of interest, and before any subsequent function calls.

The results of an *errno-setting-function* shall not be used prior to the testing of `errno`.

Rationale

An *errno-setting-function* writes a non-zero value to `errno` if an error is detected, leaving the value unmodified otherwise. The Standard includes non-normative advice that "a program that uses `errno` for error checking should set it to zero before a library function call, then inspect it before a subsequent library function call".

As the value returned by an *errno-setting-function* is unlikely to be correct when `errno` is non-zero, the program shall test `errno` to ensure that it is appropriate to use the returned value.

Exception

The value of `errno` does not have to be tested when the return value of an *errno-setting-function* can be used to determine if an error has occurred.

Example

```
void f1 ( void )
{
  errno = 0;

  errnoSettingFunction1 ();

  someFunction ();           /* Non-compliant - function call */

  if ( 0 != errno )
  {
  }

  errno = 0;

  errnoSettingFunction2 ();

  if ( 0 != errno )          /* Compliant */
  {
  }
}

void f2 ( FILE *f, fpos_t *pos )
{
  errno = 0;

  if ( fsetpos ( f, pos ) == 0 )
  {
    /* Compliant by exception - no need to test errno as no out-of-band error
reported. */
  }
  else
  {
    /* Something went wrong - errno holds an implementation-defined positive value.
*/
    handleError ( errno );
  }
}
```

See also

Rule 22.8, Rule 22.10

Rule 22.10 The value of `errno` shall only be tested when the last function to be called was an *errno-setting-function*

Category	Required
Analysis	Undecidable, System
Applies to	C90, C99

Amplification

An *errno-setting-function* is one of those described in Rule 22.8.

Rationale

The *errno-setting-functions* are the only functions which are required to set `errno` when an error is detected. Other functions, including those defined in the Standard Library that are not *errno-setting-functions*, may or may not set `errno` to indicate that an error has occurred. The use of `errno` to detect errors within these functions will fail for an implementation that does not set `errno` as it will be left unmodified.

Given that a zero value for `errno` does not therefore guarantee the absence of an error within a function that is not an *errno-setting-function*, its value shall not be tested as the outcome must be considered unreliable.

Example

In the following example:

- *atof* may or may not set `errno` when an error is detected;
- *strtod* is an *errno-setting-function*.

```
void f ( void )
{
  float64_t f64;

  errno = 0;

  f64 = atof ( "A.12" );

  if ( 0 == errno )                /* Non-compliant */
  {
    /* f64 may not have a valid value in here.      */
  }

  errno = 0;

  f64 = strtod ( "A.12", NULL );

  if ( 0 == errno )                /* Compliant     */
  {
    /* f64 will have a valid value in here.         */
  }
}
```

See also

Rule 22.8, Rule 22.9

3 Changes to existing rules

Rule 21.8

The following changes have been made to Rule 21.8 as the outright prohibition on using *getenv* is no longer necessary after the introduction of Rule 21.19 and Rule 21.20.

- Remove references to *getenv* from the rule headline and the amplification;
- Add a “See also” section with cross-references to Rule 21.19 and Rule 21.20.

4 References

[1] ISO/IEC TS 17961:2013, *Information technology – Programming languages, their environments and system software interfaces – C secure coding rules*, International Organization for Standardization, 1990

Appendix A Summary of guidelines

Code design

Dir 4.14	Required	The validity of values received from external sources shall be checked

Expressions

Rule 12.5	Mandatory	The sizeof operator shall not have an operand which is a function parameter declared as “array of type”

Standard libraries

Rule 21.13	Mandatory	Any value passed to a function in <ctype.h> shall be representable as an unsigned char or be the value EOF
Rule 21.14	Required	The Standard Library function memcmp shall not be used to compare null terminated strings
Rule 21.15	Required	The pointer arguments to the Standard Library functions memcpy, memmove and memcmp shall be pointers to qualified or unqualified versions of compatible types
Rule 21.16	Required	The pointer arguments to the Standard Library function memcmp shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type
Rule 21.17	Mandatory	Use of the string handling functions from <string.h> shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters
Rule 21.18	Mandatory	The size_t argument passed to any function in <string.h> shall have an appropriate value
Rule 21.19	Mandatory	The pointers returned by the Standard Library functions localeconv, getenv, setlocale or, strerror shall only be used as if they have pointer to const-qualified type
Rule 21.20	Mandatory	The pointer returned by the Standard Library functions asctime, ctime, gmtime, localtime, localeconv, getenv, setlocale or strerror shall not be used following a subsequent call to the same function

Resources

Rule 22.7	Required	The macro EOF shall only be compared with the unmodified return value from any Standard Library function capable of returning EOF
Rule 22.8	Required	The value of errno shall be set to zero prior to a call to an errno-setting-function
Rule 22.9	Required	The value of errno shall be tested against zero after calling an errno-setting-function

Rule 22.10	Required	The value of errno shall only be tested when the last function to be called was an errno-setting-function

Appendix B Guideline attributes

Rule	Category	Applies to	Analysis
Dir 4.14	Required	C90, C99	
Rule 12.5	Mandatory	C90, C99	Decidable, Single Translation Unit
Rule 21.13	Mandatory	C90, C99	Undecidable, System
Rule 21.14	Required	C90, C99	Undecidable, System
Rule 21.15	Required	C90, C99	Decidable, Single Translation Unit
Rule 21.16	Required	C90, C99	Decidable, Single Translation Unit
Rule 21.17	Mandatory	C90, C99	Undecidable, System
Rule 21.18	Mandatory	C90, C99	Undecidable, System
Rule 21.19	Mandatory	C90, C99	Undecidable, System
Rule 21.20	Mandatory	C90, C99	Undecidable, System
Rule 22.7	Required	C90, C99	Undecidable, System
Rule 22.8	Required	C90, C99	Undecidable, System
Rule 22.9	Required	C90, C99	Undecidable, System
Rule 22.10	Required	C90, C99	Undecidable, System