ISO/IEC JTC 1/SC 22/OWGV N 0245

Revised draft language-specific annex for C

Date	23 March 2010
Contributed by	Larry Wagoner
Original file name	C_language_annex_030810.docx
Notes	Replaces N0233

Language Specific Vulnerability Outline

C. Skeleton template for use in proposing language specific information for vulnerabilities

Every vulnerability description of Clause 6 of the main document should be addressed in the annex in the same order even if there is simply a notation that it is not relevant to the language in question.

C.1 Identification of standards

ISO/IEC. *Programming Languages---C, 2nd ed* (ISO/IEC 9899:1999). Geneva, Switzerland: International Organization for Standardization, 1999.

C.2 General Terminology

None

C.3.1 Obscure Language Features [BRS]

C.3.1.0 Status and history

C.3.1.1 Terminology and features

C.3.1.2 Description of vulnerability

C is a relatively small language with a limited syntax set lacking many of the complex features of some other languages. Many of the complex features in C are not implemented as part of the language syntax, but rather implemented as library routines. As such, most of the available features in C are used relatively frequently.

Common use across a variety of languages may make some features less obscure. Because of the unstructured code that is frequently the result of using `goto`'s, the `goto` statement is frequently restricted, or even outright banned, in some C development environments. Even though the `goto` is encountered infrequently and the use of it considered obscure, because it is fairly obvious as to its purpose and since its use is common to many other languages, the functionality of it is easily understood by even the most junior of programmers.

The use of a combination of features adds yet another dimension. Particular combinations of features in C may be used rarely together or fraught with issues if not used correctly in combination. This can cause unexpected results and potential vulnerabilities.

C.3.1.3 Avoiding the vulnerability or mitigating its effects

- Organizations should specify coding standards that restrict or ban the use of features or combinations of features that have been observed to lead to vulnerabilities in the operational environment for which the software is intended.

C.3.1.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.1.5 Bibliography

C.3.2 Unspecified Behaviour [BQF]

C.3.2.0 Status and history

C.3.2.1 Terminology and features

Unspecified behaviour occurs where the C standard provides two or more possibilities but does not dictate which one is chosen. Unspecified behaviour also occurs when an unspecified value is used.

An *unspecified value* is a value that is valid for its type and where the C standard does not impose a choice on the value chosen. Many aspects of the C language result in unspecified behaviour.

C.3.2.2 Description of vulnerability

The C standard has documented, in Annex J.1, 54 instances of unspecified behaviour. Examples of unspecified behaviour are:

- The order in which the operands of an assignment operator are evaluated
- The order in which any side effects occur among the initialization list expressions in an initializer
- The layout of storage for function parameters

Reliance on a particular behaviour that is unspecified leads to portability problems because the expected behaviour may be different for any given instance. Many cases of unspecified behaviour have to do with the order of evaluation of subexpressions and side effects. For example, in the function call

```
f1(f2(x), f3(x));
```

the functions `f2` and `f3` may be called in any order possibly yielding different results depending on the order in which the functions are called.

C.3.2.3 Avoiding the vulnerability or mitigating its effects

- Do not rely on unspecified behaviour because the behaviour can change at each instance. Thus, any code that makes assumptions about the behaviour of something that is unspecified should be replaced to make it less reliant on a particular installation and more portable.

C.3.2.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.2.5 Bibliography

C.3.3 Undefined Behaviour [EWF]

C.3.3.0 Status and history

C.3.3.1 Terminology and features

Undefined behaviour is behaviour that results from using erroneous constructs and data.

C.3.3.2 Description of vulnerability

The C standard does not impose any requirements on undefined behaviour. Typical undefined behaviours include doing nothing, producing unexpected results, and terminating the program.

The C standard has documented, in Annex J.2, 191 instances of undefined behaviour known to exist in C. One example of undefined behaviour occurs when the value of the second operand of the / or % operator is zero. This is generally not detectable through static analysis of the code, but could easily be prevented by a check for a zero divisor before the operation is performed. Leaving this behaviour as undefined lessens the burden on the implementation of the division and modulo operators.

Other examples of undefined behaviour are:

- Referring to an object outside of its lifetime
- The conversion to or from an integer type that produces a value outside of the range that can be represented
- The use of two identifiers that differ only in non-significant characters

Relying on undefined behaviour makes a program unstable and non-portable. While some cases of undefined behaviour may be consistent across multiple implementations, it is still dangerous to rely on them. Relying on undefined behaviour can result in errors that are difficult to locate and only present themselves under special circumstances. For example, accessing memory deallocated by free or realloc results in undefined behaviour, but it may work most of the time.

C.3.3.3 Avoiding the vulnerability or mitigating its effects

- Eliminate to the extent possible all cases of undefined behaviour from a program

C.3.3.4 Implications for standardization

Future standardization efforts should consider:
Making the declarations of undefined behaviour more definitive. The collection of undefined behaviour in Annex J.2 is well done with cross references to sections in the standard. Most of the entries are well defined, but the few that use words such as “proper” or “inappropriately” should be better defined.

C.3.3.5 Bibliography

C.3.4 Implementation-defined Behaviour [FAB]

C.3.4.0 Status and history

C.3.4.1 Terminology and features

Implementation-defined behaviour is unspecified behaviour where the resulting behaviour is chosen by the implementation. Implementation-defined behaviours are typically related to the environment, representation of types, architecture, locale, and library functions.

C.3.4.2 Description of vulnerability

The C standard has documented, in Annex J.3, 112 instances of implementation-defined behaviour. Examples of implementation-defined behaviour are:

- The number of bits in a byte
- The direction of rounding when a floating-point number is converted to a narrower floating-point number
- The rules for composing valid file names

Relying on implementation-defined behaviour can make a program less portable across implementations. However, this is less true than for unspecified and undefined behaviour.

The following code shows an example of reliance upon implementation-defined behaviour:

```
unsigned int x = 50;
x += (x << 2) + 1;  // x = 5x + 1
```

Since the bitwise representation of integers is implementation-defined, the computation on `x` will be incorrect for implementations where integers are not represented in two's complement form.

C.3.4.3 Avoiding the vulnerability or mitigating its effects

- Eliminate to the extent possible any reliance on implementation-defined behaviour from programs in order to increase portability. Even programs that are specifically intended for a particular implementation may in the future be ported to another environment or sections reused for future implementations.

C.3.4.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.4.5 Bibliography

C.3.5 Deprecated Language Features [MEM]

C.3.5.0 Status and history

C.3.5.1 Terminology and features

C.3.5.2 Description of vulnerability

C has deprecated one function, the function `gets`. The `gets` function copies a string from standard input into a fixed-size array. There is no safe way to use `gets` because it performs an unbounded copy of user input. Thus, every use of gets constitutes a buffer overflow vulnerability.

C has deprecated several language features primarily by tightening the requirements for the feature:

- Implicit declarations are no longer allowed.
- Functions cannot be implicitly declared. They must be defined before use or have a prototype.
- The use of the function `ungetc` at the beginning of a binary file is deprecated.
- The deprecation of aliased array parameters has been removed.
- A `return` without expression is not permitted in a function that returns a value (and vice versa).

Violating these new tighter features will generate an error.

C.3.5.3 Avoiding the vulnerability or mitigating its effects

- Do not use the function `gets` as there isn't a safe and secure way to use it.
- Although backward compatibility is sometimes offered as an option for compilers so one can avoid changes to code to be compliant with current language specifications, updating the legacy software to the current standard is a better option.

C.3.5.4 Implications for standardization

Future standardization efforts should consider:

- Creating an Annex that lists deprecated features.

C.3.5.5 Bibliography

C.3.6 Pre-processor Directives [NMP]

C.3.6.0 Status and history

C.3.6.1 Terminology and features

A preprocessing directive of the form

```
# define identifier lparen identifier-listopt ) replacement-list new-line
# define identifier lparen ... ) replacement-list new-line
# define identifier lparen identifier-list , ... ) replacement-list new-line
```

defines a *function-like macro* with parameters, whose use is similar syntactically to a function call. For example, the following function-like macro calculates the cube of its argument by replacing all occurrences of the argument X in the body of the macro.

```
#define CUBE(X) ((X) * (X) * (X))
/* ... */
int a = CUBE(2);
```

The above example expands to:

```
int a = ((2) * (2) * (2));
```

which evaluates to 8.

C.3.6.2 Description of vulnerability

The C pre-processor allows the use of macros that are text-replaced before compilation.

Function-like macros look similar to functions but have different semantics. Because the arguments are text-replaced, expressions passed to a function-like macro may be evaluated multiple times. This can result in unintended and undefined behaviour if the arguments have side effects or are pre-processor directives as described by C99 §6.10 [1]. Additionally, the arguments and body of function-like macros should be fully parenthesized to avoid unintended and undefined behaviour [2].

The following code example demonstrates undefined behaviour when a function-like macro is called with arguments that have side-effects (in this case, the increment operator) [2]:

```
#define CUBE(X) ((X) * (X) * (X))
/* ... */
int i = 2;
int a = 81 / CUBE(++i);
```

The above example expands into:

```
int a = 81 / ((++i) * (++i) * (++i));
```

which is undefined behaviour and is probably not the intended result.

Another mechanism of failure can occur when the arguments within the body of a function-like macro are not fully parenthesized. The following example shows the CUBE macro without parenthesized arguments [2]:

```
#define CUBE(X) (X * X * X)
/* ... */
int a = CUBE(2 + 1);
```

This example expands to:

```
int a = (2 + 1 * 2 + 1 * 2 + 1)
```

which evaluates to 7 instead of the intended 27.

C.3.6.3 Avoiding the vulnerability or mitigating its effects

This vulnerability can be avoided or mitigated in C in the following ways:

- Replace macro-like functions with inline functions where possible. Although making a function inline only suggests to the compiler that the calls to the function be as fast as possible, the extent to which this is done is implementation-defined. Inline functions do offer consistent semantics and allow for better analysis by static analysis tools.
- Ensure that if a function-like macro must be used, that its arguments and body are parenthesized.
- Do not embed pre-processor directives or side-effects such as an assignment, increment/decrement, volatile access, or function call in a function-like macro.

C.3.6.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.6.5 Bibliography

[1] Seacord, Robert C. *The CERT C Secure Coding Standard*. Boston: Addison-Wesley, 2008.
[2] GNU Project. GCC Bugs "Non-bugs" http://gcc.gnu.org/bugs.html#nonbugs_c (2009).

C.3.7 Choice of Clear Names [NAI]

C.3.7.0 Status and history

C.3.7.1 Terminology and features

C.3.7.2 Description of vulnerability

C is somewhat susceptible to errors resulting from the use of similarly appearing names. C does require the declaration of variables before they are used. However, C does allow scoping so that a variable which is not declared locally may be resolved to some outer block and that resolution may not be noticed by a human reviewer. Variable name length is implementation specific and so one implementation may resolve names to one length whereas another implementation may resolve names to another length resulting in unintended behaviour.

As with the general case, calls to the wrong subprogram or references to the wrong data element (when missed by human review) can result in unintended behaviour.

C.3.7.3 Avoiding the vulnerability or mitigating its effects

- Use names which are clear and non-confusing.
- Use consistency in choosing names.
- Keep names short and consise in order to make the code easier to understand.
- Choose names that are rich in meaning.
- Keep in mind that code will be reused and combined in ways that the original developers never imagined.
- Make names distinguishable within the first few characters due to scoping in C. This will also assist in averting problems with compilers resolving to a shorter name than was intended.
- Do not differentiate names through only a mixture of case or the presence/absence of an underscore character.
- Avoid differentiating through characters that are commonly confused visually such as 'O' and '0', 'l' (lower case 'L'), 'I' (capital 'I') and '1', 'S' and '5', 'Z' and '2', and 'n' and 'h'.
- Coding guidelines should be developed to define a common coding style and to avoid the above dangerous practices.

C.3.7.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.7.5 Bibliography

C.3.8 Choice of Filenames and other External Identifiers [AJN]

C.3.8.0 Status and history

C.3.8.1 Terminology and features

C.3.8.2 Description of vulnerability

C allows filenames and external identifiers to contain what could be unsafe characters or characters in unsafe positions. For example, in C, a string can be used to name a file by calling `fopen()` or `rename()`. Control characters, spaces, and leading dashes can be used in filenames which can cause unintended results when these characters are processed by the operating system. The letters “A” through “Z” and “a” through “z”, digits “0” through “9”, period, hyphen and underscore are considered portable.

Filenames may be interpreted unexpectedly if certain sequences of characters are used. For example, the filename:

```
char *file_name ="&#xBB;&#xA3;???&#xAB;";
```

will result in the file name “??????” when used on a Red Hat Linux distribution.

C.3.8.3 Avoiding the vulnerability or mitigating its effects

- Restrict filenames and external identifier names to the portable set mentioned in the previous section.

C.3.8.4 Implications for standardization

Future standardization efforts should consider:

- Language APIs for interfacing with external identifiers should be compliant with ISO/IEC 9945:2003 (IEEE Std 1003.1-2001).

C.3.8.5 Bibliography

C.3.9 Unused Variable [XYR]

C.3.9.0 Status and history

C.3.9.1 Terminology and features

C.3.9.2 Description of vulnerability

Variables may be declared, but never used when writing code or the need for a variable may be eliminated in the code, but the declaration may remain. Most compilers will report this as a warning and the warning can be easily resolved by removing the unused variable.

C.3.9.3 Avoiding the vulnerability or mitigating its effects

- Resolve all compiler warnings for unused variables. This is trivial in C as one simply needs to remove the declaration of the variable. Having an unused variable in code indicates that either warnings were turned off during compilation or were ignored by the developer. The compiler gcc allows the use of an attribute “`((unused))`” to indicate that a variable is intentionally left in the code and unused:

```
int var1 __attribute__ ((unused));
```

This will signify to the compiler not to flag a warning for this variable being unused. However, this is not part of the C standard and thus is not portable.

C.3.9.4 Implications for standardization

Future standardization efforts should consider:

- Defining a standard way of declaring an attribute such as "`__attribute__ ((unused))`" to indicate that a variable is intentionally unused.

C.3.9.5 Bibliography

C.3.10 Identifier Name Reuse [YOW]

C.3.10.0 Status and history

C.3.10.1 Terminology and features

C.3.10.2 Description of vulnerability

C allows scoping so that a variable which is not declared locally may be resolved to some outer block and that resolution may cause the variable to operate on an entity other than the one intended.

Because the variable name var1 was reused in the following example, the printed value of `var1` may be unexpected.

```
int var1;                   /* declaration in outer scope */
var1 = 10;
{
      int var2;
      int var1;             /* declaration in nested (inner) scope */
      var2 = 5;
      var1 = 1;             /* var1 in inner scope is 1*/
}
print (“var1=%d\n”, var1);  /* will print “var1=10” as var1 refers */
                            /* to  var1 in the outer scope */
```

Removing the declaration of `var2` will result in a compiler error of an undeclared variable. However, removing the declaration of `var1` in the inner block will not result in an error as `var1` will be resolved to the declaration in the outer block. That resolution will result in the printing of "`var1=1`" instead of "`var1=10`".

C.3.10.3 Avoiding the vulnerability or mitigating its effects

- Ensure that a definition of an entity does not occur in a scope where a different entity with the same name is accessible and can be used in the same context. A language-specific project coding convention can be used to ensure that such errors are detectable with static analysis.
- Ensure that a definition of an entity does not occur in a scope where a different entity with the same name is accessible and has a type that permits it to occur in at least one context where the first entity can occur.
- Ensure that all identifiers differ within the number of characters considered to be significant by the implementations that are likely to be used, and document all assumptions.

C.3.10.4 Implications for standardization

Future standardization efforts should consider:

- A common warning in Annex I should be added for variables with the same name in nested scopes.

C.3.10.5 Bibliography

C.3.11 Type System [IHN]

C.3.11.0 Status and history

C.3.11.1 Terminology and features

C.3.11.2 Description of vulnerability

C is a statically typed language. In some ways C is both strongly and weakly typed as it requires all variables to be typed, but sometimes allows implicit or automatic conversion between types. For example, C will implicitly convert a `long int` to an `int` and potentially discard many significant digits. Note that integer sizes are implementation defined so that in some implementations, the conversion from a `long int` to an `int` cannot discard any digits since they are the same size. In some implementations, all integer types could be implemented as the same size.

C allows implicit conversions as in the following example:

```
short a = 1023;
int b;
b = a;
```

If an implicit conversion could result in a loss of precision such as in a conversion from a 16 bit `int` to an 8 bit `short int`:

```
int a = 1023;
short b;
a = b;
```

most compilers will issue a warning.

C has a set of rules to determine how conversion between data types will occur. In C, for instance, every integer type has an integer conversion rank that determines how conversions are performed. The ranking is based on the concept that each integer type contains at least as many bits as the types ranked below it. The following rules for determining integer conversion rank are defined in C99:

- No two different signed integer types have the same rank, even if they have the same representation.
- The rank of a signed integer type is greater than the rank of any signed integer type with less precision.
- The rank of `long long int` is greater than the rank of `long int`, which is greater than the rank of `int`, which is greater than the rank of `short int`, which is greater than the rank of `signed char`.
- The rank of any unsigned integer type is equal to the rank of the corresponding signed integer type, if any.
- The rank of any standard integer type is greater than the rank of any extended integer type with the same width.
- The rank of `char` is equal to the rank of `signed char` and `unsigned char`.
- The rank of any extended signed integer type relative to another extended signed integer type with the

same precision is implementation defined but still subject to the other rules for determining the integer conversion rank.

- The rank of `_Bool` shall be less than the rank of all other standard integer types.
- The rank of any enumerated type shall equal the rank of the compatible integer type
- The rank of any extended signed integer type relative to another extended signed integer type with the same precision is implementation-defined, but still subject to the other rules for determining the integer conversion rank.
- For all integer types `T1`, `T2`, and `T3`, if `T1` has greater rank than `T2` and `T2` has greater rank than `T3`, then `T1` has greater rank than `T3`.

The integer conversion rank is used in the usual arithmetic conversions to determine what conversions need to take place to support an operation on mixed integer types.

- If both operands have the same type, no further conversion is needed.
- If both operands are of the same integer type (signed or unsigned), the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.
- If the operand that has unsigned integer type has rank greater than or equal to the rank of the type of the other operand, the operand with signed integer type is converted to the type of the operand with unsigned integer type.
- If the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, the operand with unsigned integer type is converted to the type of the operand with signed integer type.
- Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type. Specific operations can add to or modify the semantics of the usual arithmetic operations.

Other conversion rules exist for other data type conversions. So even though there are rules in place and the rules are rather straightforward, the variety and complexity of the rules can cause unexpected results and potential vulnerabilities. For example, though there is a prescribed order which conversions will take place, determining how the conversions will affect the final result can be difficult as in the following example:

```
long foo (short a, int b, int c, long d, long e, long f) {
     return (((b + f) * d – a + e) / c);
  }
```

The implicit conversions performed in the `return` statement can be nontrivial to discern, but can greatly impact whether any of the variables wrap around during the computation.

C.3.11.3 Avoiding the vulnerability or mitigating its effects

- Consideration of the rules for typing and conversions will assist in avoiding vulnerabilities. However, a lack of full understanding by the programmer of the implications of the rules may cause unexpected results even though the rules may be clear. Complex expressions and intricacies of the rules can cause a difference between what a programmer expects and what actually happens.
- Make casts explicit to give the programmer a clearer vision and expectations of conversions.

C.3.11.4 Implications for standardization

Future standardization efforts should consider:

- Moving in the direction over time to being a more strongly typed language. Much of the use of weak typing is simply convenience to the developer in not having to fully consider the types and uses of variables. Stronger typing forces good programming discipline and clarity about variables while at the same time removing many unexpected run time errors due to implicit conversions. This is not to say that

C should be strictly a strongly typed language – some advantages of C are due to the flexibility that weaker typing provides. It is suggested that when enforcement of strong typing does not detract from the good flexibility that C offers (e.g. adding an integer to a character to step through a sequence of characters) and is only a convenience for programmers (e.g. adding an integer to a floating-point), then the standard should specify the stronger typed solution.

C.3.11.5 Bibliography

C.3.12 Bit Representations [STR]

C.3.12.0 Status and history

C.3.12.1 Terminology and features

C.3.12.2 Description of vulnerability

C supports a variety of sizes for integers such as `short int`, `int`, `long int` and `long long int`. Each may either be signed or unsigned. C also supports a variety of bitwise operators that make bit manipulations easy such as left and right shifts and bitwise operators. These bit manipulations can cause unexpected results or vulnerabilities through miscalculated shifts or platform dependent variations.

Bit manipulations are necessary for some applications and may be one of the reasons that a particular application was written in C. Although many bit manipulations can be rather simple in C, such as masking off the bottom three bits in an integer, more complex manipulations can cause unexpected results. For instance, right shifting a signed integer is implementation defined in C, as is shifting by an amount greater than or equal to the size of the data type. For instance, on a host where an `int` is of size 32 bits,

```
unsigned int foo(const int k) {
      unsigned int i = 1;
      return i << k;
}
```

is undefined for values of `k` greater than or equal to 32.

The storage representation for interfacing with external constructs can cause unexpected results. Byte orders may be in little endian or big endian format and unknowingly switching between the two can unexpectedly alter values.

C.3.12.3 Avoiding the vulnerability or mitigating its effects

- Only use bitwise operators on unsigned integer operators as the results of some bitwise operations on signed integers are implementation defined.
- Use commonly available functions such as `htonl()`, `htons()`, `ntohl()` and `ntohs()`to convert from host byte order to network byte order and vice versa. This would be needed to interface between an i80x86 architecture where the Least Significant Byte is first with the network byte order, as used on the Internet, where the Most Significant Byte is first. ***Note:*** *functions such as these are not part of the C standard and can vary somewhat among different platforms.*
- In cases where there is a possibility that the shift is greater than the size of the variable, perform a check or, as the following example shows, a modulo reduction before the shift:

  ```
  unsigned int i;
  unsigned int k;
  ```

```
unsigned int shifted_i
…
if (k < sizeof(unsigned int)*CHAR_BIT)
  shifted_i = i << k;
else
  // handle error condition
  …
```

C.3.12.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.12.5 Bibliography

C.3.13 Floating-point Arithmetic [PLF]

C.3.13.0 Status and history

C.3.13.1 Terminology and features

C.3.13.2 Description of vulnerability

C permits the floating-point data types float, double and long double. Due to the approximate nature of floating-point representations, the use of float and double data types in situations where equality is needed or where rounding could accumulate over multiple iterations could lead to unexpected results and potential vulnerabilities in some situations.

As with most data types, C is very flexible in how `float`, `double` and `long double` can be used. For instance, C allows the use of floating-point types to be used as loop counters and in equality statements. Even though a loop may be expected to only iterate a fixed number of times, depending on the values contained in the floating-point type and on the loop counter and termination condition, the loop could execute forever. For instance iterating a time sequence using 10 nanoseconds as the increment:

```
float f;
for (f=0.0; f!=1.0; f+=0.00000001)
…
```

may or may not terminate after 10,000,000 iterations. The representations used for `f` and the accumulated effect of many iterations may cause `f` to not be identical to 1.0 causing the loop to continue to iterate forever.

Similarly, the Boolean test

```
float f=1.336;
float g=2.672;
if (f == (g/2))
…
```

may or may not evaluate to true. Given that `f` and `g` are constant values, it is expected that consistent results will be achieved on the same platform. However, it is questionable whether the logic performs as expected when a float that is twice that of another is tested for equality when divided by 2 as above. This can depend on the values selected due to the quirks of floating-point arithmetic.

C.3.13.3 Avoiding the vulnerability or mitigating its effects

- Do not use a floating-point expression in a Boolean test for equality. In C, implicit casts may make an expression floating-point even though the programmer did not expect it.
- Check for an acceptable closeness in value instead of a test for equality when using floats and doubles to avoid rounding and truncation problems.
- Do not convert a floating-point number to an integer unless the conversion is a specified algorithmic requirement or is required for a hardware interface.

C.3.13.4 Implications for standardization

Future standardization efforts should consider:

- A common warning in Annex I should be added for floating-point expressions being used in a Boolean test for equality.

C.3.13.5 Bibliography

C.3.14 Enumerator Issues [CCB]

C.3.14.0 Status and history

C.3.14.1 Terminology and features

C.3.14.2 Description of vulnerability

The enum type in C comprises a set of named integer constant values as in the example:

```
enum abc {A,B,C,D,E,F,G,H} var_abc;
```

The values of the contents of `abc` would be A=0, B=1, C=2, etc. C allows values to be assigned to the enumerated type as follows:

```
enum abc {A,B,C=6,D,E,F=7,G,H} var_abc;
```

This would result in:

A=0, B=1, C=6, D=7, E=8, F=7, G=8, H=9

yielding both gaps in the sequence of values and repeated values.

If a poorly constructed `enum` type is used in loops, problems can arise. Consider the enumerated type `var_abc` defined above used in a loop:

```
int x[8];
…
for (i=A; i<=H; i++)
{
  t = x[i];
…
}
```

Because the enumerated type abc has been renumbered and because some numbers have been skipped, the array will go out of bounds and there is potential for unintentional gaps in the use of x.

C.3.14.3 Avoiding the vulnerability or mitigating its effects

- Use enumerated types in the default form starting at 0 and incrementing by 1 for each member if possible. The use of an enumerated type is not a problem if it is well understood what values are assigned to the members.
- Use an enumerated type to select from a limited set of choices to make possible the use of tools to detect omissions of possible values such as in switch statements.
- Use the following format if the need is to start from a value other than 0 and have the rest of the values be sequential:

```
enum abc {A=5,B,C,D,E,F,G,H} var_abc;
```

- Use the following format if gaps are needed or repeated values are desired and so as to be explicit as to the values in the enum, then:

```
enum abc {
   A=0,
   B=1,
   C=6,
   D=7,
   E=8,
   F=7,
   G=8,
   H=9
} var_abc;
```

C.3.14.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.14.5 Bibliography

C.3.15 Numeric Conversion Errors [FLC]

C.3.15.0 Status and history

C.3.15.1 Terminology and features

C.3.15.2 Description of vulnerability

C permits implicit conversions. That is, C will automatically perform a conversion without an explicit cast. For instance, C allows

```
int i;
float f=1.25;
i = f;
```

This implicit conversion will discard the fractional part of `f` and set `i` to 1. If the value of `f` is greater than `INT_MAX`, then the assignment of `f` to `i` would be undefined.

The rules for implicit conversions in C are defined in the C standard. For instance, integer types smaller than `int` are promoted when an operation is performed on them. If all values of Boolean, character or integer type can be represented as an `int`, the value of the smaller type is converted to an `int`; otherwise, it is converted to an unsigned `int`.

Integer promotions are applied as part of the usual arithmetic conversions to certain argument expressions; operands of the unary +, -, and ~ operators, and operands of the shift operators. The following code fragment shows the application of integer promotions:

```
char c1, c2;
c1 = c1 + c2;
```

Integer promotions require the promotion of each variable (`c1` and `c2`) to `int` size. The two `int` values are added and the sum is truncated to fit into the `char` type.

Integer promotions are performed to avoid arithmetic errors resulting from the overflow of intermediate values. For example:

```
signed char cresult, c1, c2, c3;
c1 = 100;
c2 = 3;
c3 = 4;
cresult = c1 * c2 / c3;
```

In this example, the value of `c1` is multiplied by `c2`. The product of these values is then divided by the value of `c3` (according to operator precedence rules). Assuming that signed char is represented as an 8-bit value, the product of `c1` and `c2` (300) cannot be represented. Because of integer promotions, however, `c1`, `c2`, and `c3` are each converted to `int`, and the overall expression is successfully evaluated. The resulting value is truncated and stored in `cresult`. Because the final result (75) is in the range of the signed `char` type, the conversion from `int` back to `signed char` does not result in lost data. It is possible that the conversion could result in a loss of data should the data be larger than the storage location.

A loss of data (truncation) can occur when converting from a signed type to a signed type with less precision. For example, the following code can result in truncation:

```
signed long int sl = LONG_MAX;
signed char sc = (signed char)sl;
```

The C standard defines rules for integer promotions, integer conversion rank, and the usual arithmetic conversions. The intent of the rules is to ensure that the conversions result in the same numerical values, and that these values minimize surprises in the rest of the computation.

C.3.15.3 Avoiding the vulnerability or mitigating its effects

- Check the value of a larger type before converting it to a smaller type to see if the value in the larger type is within the range of the smaller type. Any conversion from a type with larger precision to a smaller precision type could potentially result in a loss of data. In some instances, this loss of precision is desired. Such cases should be explicitly acknowledged in comments. For example, the following code could be used to check whether a conversion from an unsigned integer to an unsigned character will result in a loss of precision:

```
        unsigned int i;
        unsigned char c;
        …
        if  (i <= UCHAR_MAX) {  // check against the maximum value for an
  object of type unsigned char
          c = (unsigned char) i;
         }
        else
         {
           // handle error condition
          }
        …
```

- Close attention should be given to all warning messages issued by the compiler regarding multiple casts. Making a cast in C explicit will both remove the warning and acknowledge that the change in precision is on purpose.

C.3.15.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.15.5 Bibliography

C.3.16 String Termination [CJM]

C.3.16.0 Status and history

C.3.16.1 Terminology and features

C.3.16.2 Description of vulnerability

A string in C is composed of a contiguous sequence of characters terminated by and including a null character (a byte with all bits set to 0). Therefore strings in C cannot contain the null character except as the terminating character. Inserting a null character in a string either through a bug or through malicious action can truncate a string unexpectedly. Alternatively, not putting a null character terminator in a string can cause actions such as string copies to continue well beyond the end of the expected string. Overflowing a string buffer through the intentional lack of a null terminating character can be used to expose information or to execute malicious code.

C.3.16.3 Avoiding the vulnerability or mitigating its effects

- Use safer and more secure functions for string handling from the ISO TR24731-1, Extensions to the C library–- Part 1: Bounds-checking interfaces. These are alternative string handling library functions to the existing Standard C Library. The functions verify that receiving buffers are large enough for the resulting strings being placed in them and ensure that resulting strings are null terminated. One implementation of these functions has been released as the Safe C Library.

C.3.16.4 Implications for standardization

Future standardization efforts should consider:

- Adopting the two TRs on safer C library functions, Extensions to the C Library (TR 24731-1: Part I: Bounds-checking interfaces and TR 24731-2: Part II: Dynamic allocation functions, that are currently under consideration by ISO SC22 WG14).

- Modifying or deprecating many of the C standard library functions that make assumptions about the occurrence of a string termination character.
- Define a string construct that does not rely on the null termination character.

C.3.16.5 Bibliography

C.3.17 Boundary Beginning Violation [XYX]

C.3.17.0 Status and history

C.3.17.1 Terminology and features

C.3.17.2 Description of vulnerability

A buffer underwrite condition occurs when an array is indexed outside its lower bounds, or pointer arithmetic results in an access to storage that occurs before the beginning of the intended object.

In C, the subscript operator `[]` is defined such that `E1[E2]` is identical to `(*((E1)+(E2)))`, so that in either representation, the value in location `(E1+E2)` is returned. Because C does not perform bounds checking on arrays, the following code:

```
int foo(const int i) {
      int x[] = {0,0,0,0,0,0,0,0,0,0};
      return x[i];
}
```

would return whatever is in location `x[i]` even if, say, `i` were equal to -5 (assuming that `x[-5]` was still within the address space of the program). This could be sensitive information or even a return address, which if altered by changing the value of `x[-5]`, could change the program flow.

C.3.17.3 Avoiding the vulnerability or mitigating its effects

- Perform range checking before accessing an array since C does not perform bounds checking automatically. In the interest of speed and efficiency, range checking only needs to be done when it cannot be statically shown that an access outside of the array cannot occur.
- Use safer and more secure functions for string handling from the ISO TR24731-1, Extensions to the C library–- Part 1: Bounds-checking interfaces. These are alternative string handling library functions to the existing Standard C Library. The functions verify that receiving buffers are large enough for the resulting strings being placed in them and ensure that resulting strings are null terminated. One implementation of these functions has been released as the Safe C Library.

C.3.17.4 Implications for standardization

Future standardization efforts should consider:

- Defining an array type that does automatic bounds checking.

C.3.17.5 Bibliography

C.3.18 Unchecked Array Indexing [XYZ]

C.3.18.0 Status and history

C.3.18.1 Terminology and features

C.3.18.2 Description of vulnerability

C does not perform bounds checking on arrays, so though arrays may be accessed outside of their bounds, the value returned is undefined and in some cases may result in a program termination. For example, in C the following code is valid, though, for example, if `i` has the value 10, the result is undefined:

```
int foo(const int i) {
      int t;
      int x[] = {0,0,0,0,0};
      t = x[i];
      return t;
}
```

The variable `t` will likely be assigned whatever is in the location pointed to by `x[10]` (assuming that `x[10]` is still within the address space of the program).

C.3.18.3 Avoiding the vulnerability or mitigating its effects

- Perform range checking before accessing an array since C does not perform bounds checking automatically. In the interest of speed and efficiency, range checking only needs to be done when it cannot be statically shown that an access outside of the array cannot occur.
- Use safer and more secure functions for string handling from the ISO TR24731-1, Extensions to the C library–- Part 1: Bounds-checking interfaces. These are alternative string handling library functions to the existing Standard C Library. The functions verify that receiving buffers are large enough for the resulting strings being placed in them and ensure that resulting strings are null terminated. One implementation of these functions has been released as the Safe C Library.

C.3.18.4 Implications for standardization

Future standardization efforts should consider:

- Defining an array type that does automatic bounds checking.

C.3.18.5 Bibliography

C.3.19 Unchecked Array Copying [XYW]

C.3.19.0 Status and history

C.3.19.1 Terminology and features

C.3.19.2 Description of vulnerability

A buffer overflow occurs when some number of bytes (or other units of storage) is copied from one buffer to another and the amount being copied is greater than is allocated for the destination buffer.
In the interest of ease and efficiency, C library functions such as `memcpy(void * restrict s1, const void * restrict s2, size_t n)` and `memmove(void *s1, const void *s2, size_t n)` are used to copy the contents from one area to another. `Memcpy()` and `memmove()` simply copy memory and no checks are made as to whether the destination area is large enough to accommodate the `n` units of data being copied. It is assumed that the calling routine has ensured that adequate space has been provided in the destination. Problems can arise when the destination buffer is too small to receive the amount of data being copied or if the indices being used for either the source or destination are not the intended indices.

C.3.19.3 Avoiding the vulnerability or mitigating its effects

- Perform range checking before calling a memory copying function such as `memcpy()` and `memmove()`. These functions do not perform bounds checking automatically. In the interest of speed and efficiency, range checking only needs to be done when it cannot be statically shown that an access outside of the array cannot occur.

C.3.19.4 Implications for standardization

Future standardization efforts should consider:

- Defining functions that contain an extra parameter in `memcpy` and `memmove` for the maximum number of bytes to copy. In the past, some have suggested that the size of the destination buffer be used as an additional parameter. Some critics state that this solution is very easy to circumvent by simply repeating the parameter that was used for the number of bytes to copy as the parameter for the size of the destination buffer. This analysis and criticism is correct. What is needed is a failsafe check as to the maximum number of bytes to copy. There are several reasons for creating new functions with an additional parameter. This would make it easier for static analysis to eliminate those cases where the memory copy could not be a problem (such as when the maximum number of bytes is demonstrably less than the capacity of the receiving buffer). Manual analysis or more involved static analysis could then be used for the remaining situations where the size of the destination buffer may not be sufficient for the maximum number of bytes to copy. This extra parameter may also help in determining which copies could take place among objects that overlap. Such copying is undefined according to the C standard. It is suggested that safer versions of functions that include a restriction `max_n` on the number of bytes `n` to copy (e.g. `void *memncpy(void * restrict s1,const void * restrict s2,size_t n), const size_t max_n`) be added to the standard in addition to retaining the current corresponding functions (e.g. `memcpy(void * restrict s1,const void * restrict s2,size_t n))`). The additional parameter would be consistent with the copying function pairs that have already been created such as `strcpy/strncpy` and `strcat/strncat`. This would allow a safer version of memory copying functions for those applications that want to use them in to facilitate both safer and more secure code and more efficient and accurate static code reviews.

C.3.19.5 Bibliography

C.3.20 Buffer Overflow [XZB]

C.3.20.0 Status and history

C.3.20.1 Terminology and features

C.3.20.2 Description of vulnerability

C is a very flexible and efficient language due to its rather lax restrictions on memory manipulations. Writing outside of a buffer can occur very easily in C due to a miscalculation of the size of the buffer, a mistake in a loop termination condition or any of dozens of other ways. Egregious violations of a buffer size are often found during testing as crashes of the program occur. However, more subtle or input dependent overflows may go undetected in testing and later be exploited by attackers.

As with other languages, it is very easy to overflow a buffer in C. The main difference is that C does not prevent or detect the occurrence automatically as is done in many other languages. For instance, consider:

```
int foo(const int n) {
      char buf[10];
      for (i=1; i++; i<=n)
       buf[i] = i + 0x40;
      return buf[n];
}
```

A value of 10 for `n` will write 0x50 to `buf[10]` which is one beyond the end of the array `buf` which starts at `buf[0]` and ends at `buf[9]`. Overflows where the amount of the overflow and the content can be manipulated by an attacker can cause the program to crash or execute logic that gives the attacker host access. For instance, the `gets()` function has been deprecated since there isn't a way stop a user from typing in a longer string than expected and overrunning a buffer. Consider:

```
int main()
{
  char buf[500];
  printf "Type something.\");
  gets(buf);
  printf "You typed: %s\", buf);

  return 0;
}
```

Typing in a string longer than 499 characters (1 less than the buffer length to account for the string null termination character) will cause the buffer to overflow. A well crafted string used as input to this program can cause execution of an attacker's malicious code.

C.3.20.3 Avoiding the vulnerability or mitigating its effects

- Validate all input values.
- Check any array index before use if there is a possibility the value could be outside the bounds of the array.
- Use length restrictive functions such as `strncpy()`instead of `strcpy()`.
- Use stack guarding add-ons to prevent overflows of stack buffers.
- Do not use the deprecated functions or other language features such as `gets()`.
- Be aware that the use of all of these preventive measures may still not be able to stop all buffer overflows from happening. However, the use of them can make it much rarer for a buffer overflow to occur and much harder to exploit it.
- Use alternative functions as specified in ISO/IEC TR 24731-1:2007. This TR provides alternative functions for the C Library (as defined in ISO/IEC 9899:1999) that promote safer, more secure programming. The functions verify that output buffers are large enough for the intended result and return a failure indicator if they are not. Optionally, failing functions call a“"runtime-constraint

handle"” to report the error. Data is never written past the end of an array. All string results are null terminated. In addition, the functions in ISO/IEC TR 24731-1:2007 are re-entrant: they never return pointers to static objects owned by the function. ISO/IEC TR 24731-1:2007 also contains functions that address insecurities with the C input-output facilities.

C.3.20.4 Implications for standardization

Future standardization efforts should consider:

- Deprecating less safe functions such as `strcpy()` and `strcat()` where a more secure alternative is available.
- Defining safer and more secure replacement functions such as `memncpy()` and `memncat()` to complement the `memcpy()` and `memcat()` functions (see in Implications for standardization.XYW).
- Adopting the two TRs on safer C library functions, Extensions to the C Library (TR 24731-1: Part I: Bounds-checking interfaces and TR 24731-2: Part II: Dynamic allocation functions, that are currently under consideration by ISO SC22 WG14.

C.3.20.5 Bibliography

C.3.21 Pointer Casting and Pointer Type Changes [HFC]

C.3.21.0 Status and history

C.3.21.1 Terminology and features

C.3.21.2 Description of vulnerability

C allows the value of a pointer to and from another data type. These conversions can cause unexpected changes to pointer values.

Pointers in C refer to a specific type, such as integer. If `sizeof(int)` is 4 bytes, and `ptr` is a pointer to integers that contains the value 0x5000, then `ptr++` would make `ptr` equal to 0x5004. However, if `ptr` were a pointer to char, then `ptr++` would make `ptr` equal to 0x5001. It is the difference due to data sizes coupled with conversions between pointer data types that cause unexpected results and potential vulnerabilities. Due to arithmetic operations, pointers may not maintain correct memory alignment or may operate upon the wrong memory addresses.

C.3.21.3 Avoiding the vulnerability or mitigating its effects

- Maintain the same type to avoid errors introduced through conversions.
- Heed compiler warnings that are issued for pointer conversion instances. The decision may be made to avoid all conversions so any warnings must be addressed. Note that casting into and out of “void *” pointers will most likely not generate a compiler warning as this is valid in both C99 and C90.

C.3.21.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.21.5 Bibliography

C.3.22 Pointer Arithmetic [RVG]

C.3.22.0 Status and history

C.3.22.1 Terminology and features

C.3.22.2 Description of vulnerability

When performing pointer arithmetic in C, the size of the value to add to a pointer is automatically scaled to the size of the type of the pointed-to object. For instance, when adding a value to the byte address of a 4-byte integer, the value is scaled by a factor 4 and then added to the pointer. The effect of this scaling is that if a pointer P points to the `i-th` element of an array object, then `(P) + N` will point to the `i+n-th` element of the array. Failing to understand how pointer arithmetic works can lead to miscalculations that result in serious errors, such as buffer overflows.

The following example will illustrate arithmetic in C involving a pointer and how the operation is done relative to the size of the pointer's target. Consider the following code snippet:

```
int buf[5];
int *buf_ptr = buf;
```

where the address of `buf` is 0x1234. Adding 1 to `buf_ptr` will result in `buf_ptr` being equal to 0x1238 on a host where an `int` is 4 bytes. `Buf_ptr` will then contain the address of `buf[1]`. Not realizing that address operations will be in terms of the size of the object being pointed to can lead to address miscalculations and undefined behaviour.

C.3.22.3 Avoiding the vulnerability or mitigating its effects

- Consider an outright ban on pointer arithmetic due to the error prone nature of pointer arithmetic.
- Avoid the common pitfalls of pointer arithmetic. For instance, in checking the end of an array, the following method can be used:

```
int buf[INTBUFSIZE];
int *buf_ptr = buf;

while (havedata() && (buf_ptr < &buf[INTBUFSIZE])) /* buf[INTBUFSIZE]
                                         is the address of the element
                                         following the buf array */
{
  *buf_ptr++ = parseint(getdata());
}
```

C.3.22.4 Implications for standardization in

Future standardization efforts should consider:

- Restrictions on pointer arithmetic that could eliminate common pitfalls. Pointer arithmetic is error prone and the flexibility that it offers is very useful, but some of the flexibility is simply a shortcut that if restricted could lessen the chance of a pointer arithmetic based error.

C.3.22.5 Bibliography

C.3.23 Null Pointer Dereference [XYH]

C.3.23.0 Status and history

C.3.23.1 Terminology and features

C.3.23.2 Description of vulnerability

C allows memory to be dynamically allocated primarily through the use of `malloc()`, `calloc()`, and `realloc()`. Each will return the address to the allocated memory. Due to a variety of situations, the memory allocation may not occur as expected and a null pointer will be returned. Other operations or faults in logic can result in a memory pointer being set to null. Using the null pointer as though it pointed to a valid memory location can cause a segmentation fault and other unanticipated situations.

Space for 10000 integers can be dynamically allocated in C in the following way:

```
int *ptr = malloc(10000*sizeof(int));  /*allocate space for 10000 ints*/
```

`Malloc()` will return the address of the memory allocation or a null pointer if insufficient memory is available for the allocation. It is good practice after the attempted allocation to check whether the memory has been allocated via an `if` test against `NULL`:

```
if (ptr != NULL)  /* check to see that the memory could be allocated */
```

Memory allocations usually succeed, so neglecting this test and using the memory will usually work which is why neglecting the null test will frequently go unnoticed. An attacker can intentionally create a situation where the memory allocation will fail leading to a segmentation fault.

Faults in logic can cause a code path that will use a memory pointer that was not dynamically allocated or after memory has been deallocated and the pointer was set to null as good practice would indicate.

C.3.23.3 Avoiding the vulnerability or mitigating its effects

- Check whether a pointer is null before dereferencing it. As this can be overly extreme in many cases (such as in a `for` loop that performs operations on each element of a large segment of memory), judicious checking of the value of the pointer at key strategic points in the code is recommended.

C.3.23.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.23.5 Bibliography

C.3.24 Dangling Reference to Heap [XYK]

C.3.24.0 Status and history

C.3.24.1 Terminology and features

C.3.24.2 Description of vulnerability

C allows memory to be dynamically allocated primarily through the use of `malloc()`, `calloc()`, and `realloc()`. C allows a considerable amount of freedom in accessing the dynamic memory. Pointers to the dynamic memory can be created to perform operations on the memory. Once the memory is no longer needed, it can be released through the use of `free()`. However, freeing the memory does not prevent the use of the pointers to the memory and issues can arise if operations are performed after memory has been freed.

Consider the following segment of code:

```
 int foo() {
      int *ptr = malloc (100*sizeof(int));/* allocate space for 100 integers*/
      if (ptr != NULL)  /* check to see that the memory could be allocated */
       {
            …           /* perform some operations on the dynamic memory */
            free (ptr); /* memory is no longer needed, so free it */
            …           /* program continues performing other operations */
            ptr[0] = 10;/* ERROR – memory is being used after it has been
released */
            …
       }
      …
 }
```

The use of memory in C after it has been freed is undefined. Depending on the execution path taken in the program, freed memory may still be free or may have been allocated via another `malloc()` or other dynamic memory allocation. If the memory that is used is still free, use of the memory may be unnoticed. However, if the memory has been reallocated, altering of the data contained in the memory can result in data corruption. Determining that a dangling memory reference is the cause of a problem and locating it can be very difficult.

Setting and using another pointer to the same section of dynamically allocated memory can also lead to undefined behaviour. Consider the following section of code:

```
   int foo() {
      int *ptr = malloc (100*sizeof(int));/* allocate space for 100 integers*/
      if (ptr != NULL)  /* check to see that the memory could be allocated */
       {
        int ptr2 = &ptr[10];  /* set ptr2 to point to the 10th element of the
      allocated memory */
            …           /* perform some operations on the dynamic memory */
        free (ptr); /* memory is no longer needed, so free it */
        ptr = NULL; /* set ptr to NULL to prevent ptr from being used again */
            …       /* program continues performing other operations */
      ptr2[0] = 10; /* ERROR – memory is being used after it has been released
via ptr2*/
            …
       }
      return (0);
   }
```

Dynamic memory was allocated via a `malloc` and then later in the code, `ptr2` was used to point to an address in the dynamically allocated memory. After the memory was freed using `free(ptr)` and the good practice of setting `ptr` to NULL was followed to avoid a dangling reference by `ptr` later in the code, a dangling reference still existed using `ptr2`.

C.3.24.3 Avoiding the vulnerability or mitigating its effects

- Set a freed pointer to null immediately after a `free()` call, as illustrated in the following code:
```
free (ptr);
ptr = NULL;
```
- Do not create and use additional pointers to dynamically allocated memory.
- Only reference dynamically allocated memory using the pointer that was used to allocate the memory.

C.3.24.4 Implications for standardization

Future standardization efforts should consider:
- Modifying the library `free(void *ptr)` so that it sets `ptr` to NULL to prevent reuse of `ptr`.

C.3.24.5 Bibliography

C.3.25 Templates and Generics [SYM]

Does not apply to C.

C.3.25.0 Status and history

C.3.25.1 Terminology and features

C.3.25.2 Description of vulnerability

C.3.25.3 Avoiding the vulnerability or mitigating its effects

C.3.25.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.25.5 Bibliography

C.3.26 Inheritance [RIP]

Does not apply to C.

C.3.26.0 Status and history

C.3.26.1 Terminology and features

C.3.26.2 Description of vulnerability

C.3.26.3 Avoiding the vulnerability or mitigating its effects

C.3.26.4 Implications for standardization

Future standardization efforts should consider:

None

C.3.26.5 Bibliography

C.3.27 Initialization of Variables [LAV]

C.3.27.0 Status and history

C.3.27.1 Terminology and features

C.3.27.2 Description of vulnerability

Local, automatic variables can assume unexpected values if they are used before they are initialized. C99 specifies, "If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate" [ISO/IEC 9899:1999]. In the common case, on architectures that make use of a program stack, this value defaults to whichever values are currently stored in stack memory. While uninitialized memory often contains zeros, this is not guaranteed. Consequently, uninitialized memory can cause a program to behave in an unpredictable or unplanned manner and may provide an avenue for attack.

Assuming that an uninitialized variable is 0 can lead to unpredictable program behaviour when the variable is initialized to a value other than 0.

C.3.27.3 Avoiding the vulnerability or mitigating its effects

- Heed compiler warnings about uninitialized variables. These warnings should be resolved as recommended to achieve a clean compile at high warning levels.
- Do not use memory allocated by functions such as `malloc()` before the memory is initialized as the memory contents are indeterminate.

C.3.27.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.27.5 Bibliography

C.3.28 Wrap-around Error [XYY]

C.3.28.0 Status and history

C.3.28.1 Terminology and features

C.3.28.2 Description of vulnerability

Given the limited size of any computer data type, continuously adding one to the data type eventually will cause the value to go from a the maximum possible value to a very small value. C permits this to happen without any detection or notification mechanism.

C is often used for bit manipulation. Part of this is due to the capabilities in C to mask bits and shift them. Another part is due to the relative closeness C has to assembly instructions. Manipulating bits on a signed value can inadvertently change the sign bit resulting in a number potentially going from a large positive value to a large negative value.

For example, consider the following code for a `short int` containing 16 bits:

```
int foo(short int i) {
      i++;
      return i;
}
```

Calling `foo` with the value of 65535 would return -65536. Manipulating a value in this way can result in unexpected results such as overflowing a buffer.

In C, bit shifting by a value that is greater than the size of the data type or by a negative number is undefined. The following code, where a `short int` is 16 bits, would be undefined when `j` is greater than or equal to 16 or negative:

```
int foo(short int i, const short int j) {
      return i>>j;
}
```

C.3.28.3 Avoiding the vulnerability or mitigating its effects

- Be aware that any of the following operators have the potential to wrap in C:

```
a + b       a – b       a * b       a++         a--    a += b
a -= b      a *= b      a << b      a >> b      -a
```

- Use defensive programming techniques to check whether an operation will overflow or underflow the receiving data type. These techniques can be omitted if it can be shown at compile time that overflow or underflow is not possible.
- Only conduct bit manipulations on unsigned data types. The number of bits to be shifted by a shift operator should lie between 1 and (n-1), where n is the size of the data type.

C.3.28.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.28.5 Bibliography

C.3.29 Sign Extension Error [XZI]

C.3.29.0 Status and history

C.3.29.1 Terminology and features

C.3.29.2 Description of vulnerability

C contains a variety of integer sizes: `short`, `int`, `long int` and `long long int`. Converting from a smaller to a larger size signed integer will cause the sign bit to extend which could lead to unexpected results.

The number of bits in a `short`, `int`, `long int` and `long long int` have been left vague by the C standard in order to avoid constraints on the hardware architecture. Therefore it is quite possible that the a `short`, `int`, `long int` and `long long int` could be contain the identical number of bits. On an architecture where all are the same size, there would not be a conversion issue.

When going from a smaller signed integer data type to a larger one, all of the lower order bits are copied to the larger data type. In order to transfer the signedness of the smaller integer to the larger one in a 2’s complement architecture, the sign bit must be extended. That is, if the sign bit of the smaller data type is 0, then the additional bits are set to 0. If the sign bit is 1, the additional bits are set to 1. Not modifying the bits (i.e. extending the sign bit) in this manner can cause a negative number to become a relatively large positive number upon conversion.

C.3.29.3 Avoiding the vulnerability or mitigating its effects

- Use appropriate conversion routines when converting from one data type to another. For example, do not use an unsigned conversion routine to convert a signed integer type to a larger integer data type as doing so can yield unexpected results.

C.3.29.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.29.5 Bibliography

C.3.30 Operator Precedence/Order of Evaluation [JCW]

C.3.30.0 Status and history

C.3.30.1 Terminology and features

C.3.30.2 Description of vulnerability

The order in which an expression is evaluated can drastically alter the result of the expression. The order of evaluation of the operands in C is clearly defined, but misinterpretations by programmers can lead to unexpected results.

Consider the following:

```
int foo(short int a, short int b) {
      if (a | 0x7 = b)
      ...
}
```

designed to mask off and test the lower three bits of “a” for equality to “b”. However, due to the precedence rules in C, the effect of this expression is to perform the “`0x7 == b`” and then bitwise OR that with “a” which may or may not be the expected answer.

C.3.30.3 Avoiding the vulnerability or mitigating its effects

- Use parentheses generously to avoid any uncertainty or lack of portability in the order of evaluation of an expression. If parenthesis were used in the previous example, as in:

```
int foo(short int a, short int b) {
      if ((a | 0x7) = b)
      ...
}
```

  the order of the evaluation would be clear.

C.3.30.4 Implications for standardization

Future standardization efforts should consider:

- Creating a few standardized precedence orders. Standardizing on a few precedence orders will help to eliminate the confusing intricacies that exist between languages. This would not affect current languages as altering precedence orders in existing languages is too onerous. However, this would set a basis for the future as new languages are created and adopted. Stating that a language uses "ISO precedence order A" would be very useful rather than having to spell out the entire precedence order that differs in a conceptually minor way from some other languages, but in a major way when programmers attempt to switch between languages.

C.3.30.5 Bibliography

C.3.31 Side-effects and Order of Evaluation [SAM]

C.3.31.0 Status and history

C.3.31.1 Terminology and features

C.3.31.2 Description of vulnerability

C allows expressions to have side effects. If two or more side effects modify the same expression as in:

```
int v[10];
int i;
/* … */
i = v[i++];
```

the behaviour is undefined and this can lead to unexpected results. Either the "`i++`" is performed first or the assignment "`i=v[i]`" is performed first. Because the order of evaluation can have drastic effects on the functionality of the code, this can greatly impact portability.

There are several situations in C where the order of evaluation of subexpressions or the order in which side effects take place is unspecified including:

- The order in which the arguments to a function are evaluated (C99, Section 6.5.2.2,"Function calls").
- The order of evaluation of the operands in an assignment statement (C99, Section 6.5.16,"Assignment operators").
- The order in which any side effects occur among the initialization list expressions is unspecified. In particular, the evaluation order need not be the same as the order of subobject initialization (C99, Section 6.7.8, "Initialization").

Because these are unspecified behaviours, testing may give the false impression that the code is working and portable, when it could just be that the values provided cause evaluations to be performed in a particular order that causes side effects to occur as expected.

C.3.31.3 Avoiding the vulnerability or mitigating its effects

- Expressions should be written so that the same effects will occur under any order of evaluation that the C standard permits since side effects can be dependent on an implementation specific order of evaluation.

C.3.31.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.31.5 Bibliography

C.3.32 Likely Incorrect Expression [KOA]

C.3.32.0 Status and history

C.3.32.1 Terminology and features

C.3.32.2 Description of vulnerability

C has several instances of operators which are similar in structure, but vastly different in meaning. This is so common that the C example of confusing the Boolean operator "==" with the assignment "=" is frequently cited as an example among programming languages. Using an expression that is technically correct, but which may just be a null statement can lead to unexpected results.

C is also provides a lot of freedom in constructing statements. This freedom, if misused, can result in unexpected results and potential vulnerabilities.

The flexibility of C can obscure the intent of a programmer. Consider:

```
int x,y;
/* … */
if (x = y)
 {
  /* … */
 }
```

A fair amount of analysis may need to be done to determine whether the programmer intended to do an assignment as part of the `if` statement (perfectly valid in C) or whether the programmer made the common mistake of using an "=" instead of a "==". In order to prevent this confusion, it is suggested that any assignments in contexts that are easily misunderstood be moved outside of the Boolean expression. This would change the example code to:

```
int x,y;
/* … */
x = y;
if (x == 0)
{
 /* … */
```

```
}
```

This would clearly state what the programmer meant and that the assignment of y to x was intended.

Programmers can easily get in the habit of inserting the “`;`” statement terminator at the end of statements. However, inadvertently doing this can drastically alter the meaning of code, even though the code is valid as in the following example:

```
int a,b;
/* … */
if (a == b);  /* the semi-colon will make this a null statement */
{
 /* … */
}
```

Because of the misplaced semi-colon, the code block following the `if` will always be executed. In this case, it is extremely likely that the programmer did not intend to put the semi-colon there.

C.3.32.3 Avoiding the vulnerability or mitigating its effects

- Simplify statements with interspersed comments to aid in accurately programming functionality and help future maintainers understand the intent and nuances of the code. The flexibility of C permits a programmer to create extremely complex expressions. For example, the following sub-expression, though valid, would be a nightmare to understand:

```
int d,h,i,k;
/* … */
(h+=*d++-h)&&(‘'’'^(h-’'’'))&&(i<<=4 & i||!++i--&&(h--||(k|=i))-
        i/=2);
```

- Do not embed assignments inside of expressions. Assignments embedded within other statements can be potentially problematic. Each of the following would be clearer and have less potential for problems if the embedded assignments were conducted outside of the expressions:

```
int a,b,c,d;
/* … */
if ((a == b) || (c = (d-1)))   /* the assignment to c may not occur */
                               /* if a is equal to b */
```

  or:

```
int a,b,c;
/* … */
foo (a=b, c);
```

  Each is a valid C statement, but each may have unintended results.
- Null statements should have a source line of their own. This, combined with enforcement by static analysis, would make clearer the intention that the statement was meant to be a null statement.

C.3.32.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.32.5 Bibliography

C.3.33 Dead and Deactivated Code [XYQ]

C.3.33.0 Status and history

C.3.33.1 Terminology and features

C.3.33.2 Description of vulnerability

As with any programming language that contains branching statements, C programs can potentially contain dead code. It is of concern primarily since dead code may reveal a logic flaw or an unintentional mistake on the part of the programmer. Sometimes statements can be inserted in C programs as defensive programming such as adding a default case to a switch statement even though the expectation is that the default can never be reached – until through some twist of logic or through modifications to the code the notifying error message reveals the surprising event. These types of defensive statements may be able to be shown to be computationally impossible and thus are dead code. Those are not the focus. The focus is on those statements which are not defensive and which are unreachable. It is impossible to identify all such cases and therefore only those which are blatant and that indicate deeper issues of flawed logic may be able to be identified and removed.

C uses some operators that are easily confused with other operators. For instance, the common mistake of using an assignment operator in a Boolean test as in:

```
int a,b;
/* … */
if (a = b)
…
```

can cause portions of code to become dead code since unless `b` can contain the value 0, the `else` portion of the `if` statement cannot be reached.

C.3.33.3 Avoiding the vulnerability or mitigating its effects

- Eliminate dead code to the extent possible from C programs.
- Use compilers and analysis tools to assist in identifying unreachable code.
- Use "`//`" comment syntax instead of "`/*…*/`" comment syntax to avoid the inadvertent commenting out of sections of code.
- Delete deactivated code from programs due to the possibility of accidentally activating it.

C.3.33.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.33.5 Bibliography

C.3.34 Switch Statements and Static Analysis [CLL]

C.3.34.0 Status and history

C.3.34.1 Terminology and features

C.3.34.2 Description of vulnerability

Because of the way in which the switch-case statement in C is structured, it is relatively easy to unintentionally omit the `break` statement between cases causing unintended execution of statements for some cases.

C contains a `switch` statement of the form:

```
char abc;
/* … */
switch (abc)
{
   case 1:
      sval = “a”;
      break;
   case 2:
      sval = “b”;
      break;
   case 3:
      sval = “c”;
      break;
   default:
      printf (“Invalid selection\n”);
```

If there isn’t a default case and the switched expression doesn’t match any of the cases, then control simply shifts to the next statement after the switch statement block. Unintentionally omitting a `break` statement between two cases will cause subsequent cases to be executed until a `break` or the end of the switch block is reached. This could cause unexpected results.

C.3.34.3 Avoiding the vulnerability or mitigating its effects

- Only a direct fall through should be allowed from one case to another. That is, every nonempty `case` statement should be terminated with a `break` statement as illustrated in the following example:

```
int i;
/* … */
switch (i)
 {
   case 1:
   case 2:
      i++;        /*  fall through from case 1 to 2 is permitted */
      break;
   case 3:
      j++;
   case 4:        /* fall through from case 3 to 4 is not permitted */
                  /* as it is not a direct fall through due to the */
                  /* j++ statement */
  }
```

- All `switch` statements should have a default value if only to indicate that there could exist a case that was unanticipated and thought impossible by the developers. The only exception is for switches on an enumerated type where all possible values can be exhausted. Even in the case of enumerated types, it is suggested that a default be inserted in anticipation of possible code changes to the enumerated type.

C.3.34.4 Implications for standardization

Future standardization efforts should consider:

- Defining a “fallthru” construct that will explicitly bind multiple switch cases together and eliminate the need for the `break` statement. The default would be for a case to break instead of falling through to the next case. Granted this is a major shift in concept, but if it could be accomplished, less unintentional errors would occur.

C.3.34.5 Bibliography

C.3.35 Demarcation of Control Flow [EOJ]

C.3.35.0 Status and history

C.3.35.1 Terminology and features

A *block-structured language* is a language that has a syntax for enclosing structures between bracketed keywords, such as an `if` statement bracketed by `if` and `endif,` as in FORTRAN, or a code section bracketed by `BEGIN` and END, as in PL/1.

A *comb-structured language* is a language that has an ordered set of keywords to define separate sections within a block, analogous to the multiple teeth or prongs in a comb separating sections of the comb. For example, in Ada, a block is a 4-pronged comb with keywords `declare`, `begin`, `exception`, `end`, and the `if` statement in Ada is a 4-pronged comb with keywords `if`, `then`, `else`, `end if`.

C.3.35.2 Description of vulnerability

C is a block-structured language, while languages such as Ada and Pascal are comb-structured languages. Therefore, it may not be readily apparent which statements are part of a loop construct or an `if` statement.

Consider the following section of code:

```
int foo(int a, const int *b) {
      int i=0;

      /* … */
      a = 0;
      for (i=0; i<10; i++);
        {
         a = a + b[i];
        }

}
```

At first it may appear that a will be a sum of the numbers `b[0]` to `b[9]`. However, even though the code is structured so that the “`a = a + b[i]`” code is structured to appear within the `for` loop, the “`;`” at the end of the `for` statement causes the loop to be on a null statement (the “`;`”) and the “`a = a + b[i];`” statement to only be executed once. In this case, this mistake may be readily apparent during development or testing. More subtle cases may not be as readily apparent leading to unexpected results.

`If` statements in C are also susceptible to control flow problems since there isn’t a requirement in C for there to be an `else` statement for every `if` statement. An `else` statement in C always belong to the most recent `if` statement without an else. However, the situation could occur where it is not readily apparent to which if statement an else due to the way the code is indented or aligned.

C.3.35.3 Avoiding the vulnerability or mitigating its effects

- Enclose the bodies of if, else, while, for, etc. in braces. This will reduce confusion and potential problems when modifying the software. For example:

```
int a,b,i;

/* … */

if (i = 10)
 {
   a = 5;          /* this is correct */
   b = 10;
  }
else
    a = 10;        /* this is incorrect -- the assignments to b */
                   /* were added later and were expected to */
    b = 5;         /* be part of the if and else and indented */
                   /* as such, but did not become part of the else*/
```

- Use a final else statement or a comment stating why the final else isn't necessary in all if and else if statements.

C.3.35.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.35.5 Bibliography

C.3.36 Loop Control Variables [TEX]

C.3.36.0 Status and history

C.3.36.1 Terminology and features

C.3.36.2 Description of vulnerability

C allows the modification of loop control variables within a loop. Though this is usually not considered good programming practice as it can cause unexpected problems, the flexibility of C expects the programmer to use this capability responsibly.

Since the modification of a loop control variable within a loop is infrequently encountered, reviewers of C code may not expect it and hence miss noticing the modification. Modifying the loop control variable can cause unexpected results if not carefully done. In C, the following is valid:

```
int a,i;

for (i=1; i<10; i++)
```

```
{
   …
   if (a > 7)
     i = 10;
  …
}
```

which would cause the `for` loop to exit once `a` is greater than 7 regardless of the number of loops that have occurred.

C.3.36.3 Avoiding the vulnerability or mitigating its effects

- Do not modify a loop control variable within a loop. Even though the capability exists in C, it is still considered to be a poor programming practice.

C.3.36.4 Implications for standardization

Future standardization efforts should consider:

- Defining an identifier type for loop control that cannot be modified by anything other than the loop control construct would be a relatively minor addition to C that could make C code safer and encourage better structured programming.

C.3.36.5 Bibliography

C.3.37 Off-by-one Error [XZH]

C.3.37.0 Status and history

C.3.37.1 Terminology and features

C.3.37.2 Description of vulnerability

Arrays are a common place for off by one errors to manifest. In C, arrays are indexed starting at 0, causing the common mistake of looping from 0 to the size of the array as in:

```
int foo() {
      int a[10];
      int i;
      for (i=0, i<=10, i++)
      …
      return (0);
      }
```

Strings in C are also another common source of errors in C due to the need to allocate space for and account for the string sentinel value. A common mistake is to expect to store an n length string in an n length array instead of length n+1 to account for the sentinel '\0'. Interfacing with other languages that do not use sentinel values in strings can also lead to an off by one error.

C does not flag accesses outside of array bounds, so an off by one error may not be as detectable in C as in some other languages. Several very good and freely available tools for C can be used to help detect accesses beyond the bounds of arrays that are caused by an off by one error. However, such tools will not help in the case where only a portion of the array is used and the access is still within the bounds of the array.

Looping one more or one less is usually detectable by good testing. Due to the structure of the C language, this may be the main way to avoid this vulnerability. Unfortunately some cases may still slip through the development and test phase and manifest themselves during operational use.

C.3.37.3 Avoiding the vulnerability or mitigating its effects

- Use careful programming, testing of border conditions and static analysis tools to detect off by one errors in C.

C.3.37.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.37.5 Bibliography

C.3.38 Structured Programming [EWD]

C.3.38.0 Status and history

C.3.38.1 Terminology and features

C.3.38.2 Description of vulnerability

It is as easy to write structured programs in C as it is not to. C contains the `goto` statement, which can create unstructured code. Also, C has `continue`, `break`, and `return` that can create a complicated control flow, when used in an undisciplined manner. Spaghetti code can be more difficult for C static analyzers to analyze and is sometimes used on purpose to intentionally obfuscate the functionality of software. Code that has been modified multiple times by an assortment of programmers to add or remove functionality or to fix problems can be prone to become very unstructured.

Because unstructured code in C can cause problems for analyzers (both automated and human) of code, problems with the code may not be detected as readily or at all as would be the case if the software was written in a structured manner.

C.3.38.3 Avoiding the vulnerability or mitigating its effects

- Write clear and concise structured code to make code as understandable as possible.
- Restrict the use of `goto`, `continue`, `break` and `return` to encourage more structured programming.
- Encourage the use of a single exit point from a function. At times, this guidance can have the opposite effect, such as in the case of an `if` check of parameters at the start of a function that requires the remainder of the function to be encased in the `if` statement in order to reach the single exit point. If, for example, the use of multiple exit points can arguably make a piece of code clearer, then they should be used. However, the code should be able to withstand a critique that a restructuring of the code would have made the need for multiple exit points unnecessary.

C.3.38.4 Implications for standardization

Future standardization efforts should consider:
- Deprecating the `goto` statement. The use of the `goto` construct is very often spotlighted as the

antithesis of good structured programming. Though its deprecation will not instantly make all C code structured, deprecating the `goto` and leaving in place the restricted `goto` variations (e.g. `break` and `continue`) and possibly adding other restricted `goto`'s could assist in encouraging safer and more secure C programming in general.

C.3.38.5 Bibliography

C.3.39 Passing Parameters and Return Values [CSJ]

C.3.39.0 Status and history

C.3.39.1 Terminology and features

C.3.39.2 Description of vulnerability

At times, it is useful to interface a C program with routines written in other languages. Other languages may have different data types, storage orders or parameter passing semantics. These differences in interfacing with other languages can lead to unexpected interpretations or manipulations of data.

C only passes parameters by value. That is, the receiving function will get the value of the parameter. Call by reference can be achieved by passing a reference as a value. Interfacing with another language, such as Fortran, that uses call by reference can yield some surprising results. Therefore, the addresses of the arguments must be passed when calling a Fortran subroutine from C. There are many other major and minor issues in interfacing to other languages all of which can lead to unexpected results and even potential vulnerabilities. For example, arrays in C are stored in row major order (last index varies fastest) whereas Fortran stores arrays in column major order (first index varies fastest). Other issues are minor annoyances, such as the inability of C to be able to pass a constant as a parameter to a Fortran subroutine since there isn't an address to pass (that is, &7) to satisfy the call by reference expectation.

C.3.39.3 Avoiding the vulnerability or mitigating its effects

- Use caution when interfacing with other languages as this can be error prone.
- Use interface packages that are available for many language combinations which can assist in avoiding some problems in interfacing. Even with an interface package, there will likely still be some issues that need to be addressed for a successful interface.
- Conduct additional rigorous testing on sections of code that interface with other languages.

C.3.39.4 Implications for standardization

Future standardization efforts should consider:

- Defining a standardized interface package for interfacing C with many of the top programming languages and a reciprocal package should be developed of the other top languages to interface with C.

C.3.39.5 Bibliography

C.3.40 Dangling References to Stack Frames [DCM]

C.3.40.0 Status and history

C.3.40.1 Terminology and features

C.3.40.2 Description of vulnerability

C allows the address of a variable to be stored in a variable. Should this variable's address be, for example, the address of a local variable that was part of a stack frame, then using the address after the local variable has been deallocated can yield unexpected behaviour as the memory will have been made available for further allocation and may indeed been allocated for some other use. Any use of perishable memory after it has been deallocated can lead to unexpected results.

C.3.40.3 Avoiding the vulnerability or mitigating its effects

- Do not assign the address of an object to any entity which persists after the object has ceased to exist. This is done in order to avoid the possibility of a dangling reference. Once the object ceases to exist, then so will the stored address of the object preventing accidental dangling references.
- Pointers should be assigned the null-pointer value before executing a return for any block-local addresses that have been stored in longer-lived storage.

C.3.40.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.40.5 Bibliography

C.3.41 Subprogram Signature Mismatch [OTR]

C.3.41.0 Status and history

C.3.41.1 Terminology and features

C.3.41.2 Description of vulnerability

Functions in C may be called with more or less than the number of parameters the receiving function expects. However, most C compilers will generate a warning or an error about this situation. If the number of arguments does not equal the number of parameters, the behaviour is undefined. This can lead to unexpected results when the count or types of the parameters differs from the calling to the receiving function. If too few arguments are sent to a function, then the function could still pop the expected number of arguments from the stack leading to unexpected results.

C allows a variable number of arguments in function calls. A good example of an implementation of this is the `printf` function. This is specified in the function call by terminating the list of parameters with an ellipsis (`, ...`). After the comma, no information about the number or types of the parameters is supplied. This can be a very useful feature for situations such as `printf`, but the use of this feature outside of very special situations can be the basis for vulnerabilities.

Functions may or may not be defined with a function definition. The function definition may or may not contain a parameter type list. If a function that accepts a variable number of arguments is defined without a parameter type list that ends with the ellipsis notation, the behaviour is undefined.

If the calling and receiving functions differ in the type of parameters, C will, if possible, do an implicit conversion such as the call to `sqrt` that expects a double:

```
double sqrt(double)
```

the call:

```
root2 = sqrt(2);
```

coerces the integer 2 into the double value 2.0.

C.3.41.3 Avoiding the vulnerability or mitigating its effects

- Use a function prototype to declare a function with its expected parameters to allow the compiler to check for a matching count and types of the parameters. The prototype contains just the name of the function and its parameters without the body of code that would normally follow.
- Do not use the variable argument feature except in rare instances. The variable argument feature such as is used in `printf()`is difficult to use in a type safe manner.

C.3.41.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.41.5 Bibliography

C.3.42 Recursion [GDL]

C.3.42.0 Status and history

C.3.42.1 Terminology and features

C.3.42.2 Description of vulnerability

C permits recursive calls both directly and indirectly through any chain of other functions. However, recursive functions must be implemented carefully in C as C lacks some of the protective mechanisms that could avert serious problems such as an overly large consumption of resources or an overrun of buffers. Since C is frequently cited for its high performance efficiency, the use of recursion in C is counter to this as recursion is usually very inefficient both in execution time and memory usage.

As with many languages, the high consumption of resources for recursive calls applies to C. It is difficult to predict the complete range of values that a recursive function can execute that will lead to a manageable consumption of resources. Part of this difficulty is that the range of values can change depending on the current load of the host. Manipulation of the input values to a recursive function can result in an intentional exhaustion of system resources leading to a denial of service.

C.3.42.3 Avoiding the vulnerability or mitigating its effects

- Only use recursion only in very rare instances. Although recursion can shorten programs considerably, there is a high performance penalty which is contrary to the usual high efficiency of C.
- Only use recursion if it can be proven that adequate resources exist to support the maximum level of recursion possible.

C.3.42.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.42.5 Bibliography

C.3.43 Returning Error Status [NZN]

C.3.43.0 Status and history

C.3.43.1 Terminology and features

C.3.43.2 Description of vulnerability

C provides the include file `errno.h` that defines the macros `EDOM`, `EILSEQ` and `ERANGE`, which expand to integer constant expressions with type `int`, distinct positive values and which are suitable for use in `#if` preprocessing directives. C also provides the integer `errno` that can be set to a nonzero value by any library function (if the use of **`errno`** is not documented in the description of the function in the C Standard, `errno` could be used whether or not there is an error). Though these values are defined, inconsistencies in responding to error conditions can lead to vulnerabilities.

C.3.43.3 Avoiding the vulnerability or mitigating its effects

- Check the returned error status upon return from a function. The C standard library functions provide an error status as the return value and sometimes in an additional global error value.
- Set `errno` to zero before a library function call in situations where a program intends to check `errno` before a subsequent library function call.
- Use `errno_t` to make it readily apparent that a function is returning an error code. Often a function that returns an `errno` error code is declared as returning a value of type `int`. Although syntactically correct, it is not apparent that the return code is an `errno` error code. TR 24731-1 introduced the new type `errno_t` in `errno.h` that is defined to be type `int`.

C.3.43.4 Implications for standardization

Future standardization efforts should consider:

- Joining with other languages in developing a standardized set of mechanisms for detecting and treating error conditions so that all languages to the extent possible could use them. Note that this does not mean that all languages should use the same mechanisms as there should be a variety (e.g. label parameters, auxiliary status variables), but each of the mechanisms should be standardized.

C.3.43.5 Bibliography

C.3.44 Termination Strategy [REU]

C.3.44.0 Status and history

C.3.44.1 Terminology and features

C.3.44.2 Description of vulnerability

Choosing when and where to exit is a design issue, but choosing how to perform the exit may result in the host being left in an unexpected state. C provides several ways of terminating a program including `exit()`, `_Exit()`, and `abort()`. A `return` from the initial call to the `main` function is equivalent to calling the `exit()` function with the value returned by the `main` function as its argument (this is if the return type of the `main` function is a type compatible with `int`, otherwise the termination status returned to the host environment is unspecified) or simply reaching the "}" that terminates the `main` function returns a value of 0.

All of the termination strategies in C have undefined, unspecified, and/or implementation defined behaviour associated with them. For example, if more than one call to the `exit()` function is executed by a program, the behaviour is undefined. The amount of clean-up that occurs upon termination such as the removal of temporary files or the flushing of buffers varies and may be implementation defined.

A call to `exit()` or `_Exit()` will terminate a program normally. Abnormal program termination will occur when `abort()` is used to exit a program (unless the signal `SIGABRT` is caught and the signal handler does not return). Unlike a call to `exit()`, when either `_Exit()` or `abort()` are used to terminate a program, it is implementation defined as to whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed. This can leave a system in an unexpected state.

C provides the function `atexit()` that allows functions to be registered so that at normal program termination, the registered functions will be executed to perform desired functions. C99 requires the capability to register *at least* 32 functions. Implementations expecting more than 32 registered functions may yield unexpected results.

C.3.44.3 Avoiding the vulnerability or mitigating its effects

- Use a return from the `main()` program as it is the cleanest way to exit a C program.
- Use `exit()` to quickly exit from a deeply nested function.
- Use `abort()` in situations where an abrupt halt is needed. If `abort()` is necessary, the design should protect critical data from being exposed after an abrupt halt of the program.
- Become familiar with the undefined, unspecified and/or implementation aspects of each of the termination strategies.

C.3.44.4 Implications for standardization

Future standardization efforts should consider:

- Since fault handling and exiting of a program is common to all languages, it is suggested that common terminology such as the meaning of fail safe, fail hard, fail soft, etc. along with a core API set such as `exit`, `abort`, etc. be standardized and coordinated with other languages.

C.3.44.5 Bibliography

C.3.45 Extra Intrinsics [LRM]

Does not apply to C.

C.3.45.0 Status and history

C.3.45.1 Terminology and features

C.3.45.2 Description of vulnerability

C.3.45.3 Avoiding the vulnerability or mitigating its effects

C.3.45.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.45.5 Bibliography

C.3.46 Type-breaking Reinterpretation of Data [AMV]

C.3.46.0 Status and history

C.3.46.1 Terminology and features

C.3.46.2 Description of vulnerability

The primary way in C that a reinterpretation of data is accomplished is through a `union` which may be used to interpret the same piece of memory in multiple ways. If the use of the union members is not managed carefully, then unexpected and erroneous results may occur.

C allows the use of pointers to memory so that an integer pointer could be used to manipulate character data. This could lead to a mistake in the logic that is used to interpret the data leading to unexpected and erroneous results.

C.3.46.3 Avoiding the vulnerability or mitigating its effects

- Avoid the use of unions as it is relatively easy for there to exist an unexpected program flow that leads to a misinterpretation of the union data.

C.3.46.4 Implications for standardization

Future standardization efforts should consider:
- Deprecating unions. The primary reason for the use of unions to save memory has been diminished considerably as memory has become cheaper and more available. Unions are not statically type safe and are historically known to be a common source of errors, leading to many C programming guidelines specifically prohibiting the use of unions.

C.3.46.5 Bibliography

C.3.47 Memory Leak [XYL]

C.3.47.0 Status and history

C.3.47.1 Terminology and features

C.3.47.2 Description of vulnerability

C is prone to memory leaks as many programs use dynamically allocated memory. C relies on manual memory management rather than a built in garbage collector primarily since automated memory management can be unpredictable, impact performance and is limited in its ability to detect unused memory such as memory that is still referenced by a pointer, but is never used.

Memory is dynamically allocated in C using the library calls `malloc()`, `calloc()`, and `realloc()`. When the program no longer needs the dynamically allocated memory, it can be released using the library call `free()`. Should there be a flaw in the logic of the program, memory continues to be allocated but is not freed when it is no longer needed. A common situation is where memory is allocated while in a function, the memory is not freed before the exit from the function and the lifetime of the pointer to the memory has ended upon exit from the function.

C.3.47.3 Avoiding the vulnerability or mitigating its effects

- Use debugging tools such as leak detectors to help identify unreachable memory.
- Allocate and free memory in the same module and at the same level of abstraction to make it easier to determine when and if an allocated block of memory has been freed.
- Use `realloc()` only to resize dynamically allocated arrays.
- Use garbage collectors that are available to replace the usual C library calls for dynamic memory allocation which allocate memory to allow memory to be recycled when it is no longer reachable. The use of garbage collectors may not be acceptable for some applications as the delay introduced when the allocator reclaims memory may be noticeable or even objectionable leading to performance degradation.

C.3.47.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.47.5 Bibliography

C.3.48 Argument Passing to Library Functions [TRJ]

C.3.48.0 Status and history

C.3.48.1 Terminology and features

C.3.48.2 Description of vulnerability

Parameter passing in C is either pass by reference or pass by value. There isn't a guarantee that the values being passed will be verified by either the calling or receiving functions. So values outside of the assumed range may be received by a function resulting in a potential vulnerability.

A parameter may be received by a function that was assumed to be within a particular range and then an operation or series of operations is performed using the value of the parameter resulting in unanticipated results and even a potential vulnerability.

C.3.48.3 Avoiding the vulnerability or mitigating its effects

- Do not make assumptions about the values of parameters.
- Do not assume that the calling or receiving function will be range checking a parameter. It is always safest

to not make any assumptions about parameters used in C libraries. Because performance is sometimes cited as a reason to use C, parameter checking in both the calling and receiving functions is considered a waste of time. Since the calling routine may have better knowledge of the values a parameter can hold, it may be considered the better place for checks to be made as there are times when a parameter doesn't need to be checked since other factors may limit its possible values. However, since the receiving routine understands how the parameter will be used and it is good practice to check all inputs, it makes sense for the receiving routine to check the value of parameters. Therefore, in C it is very difficult to create a blanket statement as to where the parameter checks should be made and as a result, parameter checks are recommended in both the calling and receiving routines unless knowledge about the calling or receiving routines dictates that this isn't needed.

C.3.48.4 Implications for standardization

Future standardization efforts should consider:

- Creating a recognizable naming standard for routines such that one version of a library does parameter checking to the extent possible and another version does no parameter checking. The first version would be considered safer and more secure and the second could be used in certain situations where performance is key and the checking is assumed to be done in the calling routine. A naming standard could be made such that the library that does parameter checking could be named as usual, say "library_xyz" and an equivalent version that does not do checking could have a "_p" appended, such as "library_xyz_p". Without a naming standard such as this, a considerable number of wasted cycles will be conducted doing a double check of parameters or even worse, no checking will be done in both the calling and receiving routines as each is assuming the other is doing the checking.

C.3.48.5 Bibliography

C.3.49 Dynamically-linked Code and Self-modifying Code [NYY]

C.3.49.0 Status and history

C.3.49.1 Terminology and features

C.3.49.2 Description of vulnerability

Most loaders allow dynamically linked libraries also known as shared libraries. Code is designed and tested using a suite of shared libraries which are loaded at execution time. The process of linking and loading is outside the scope of the C standard, but many popular platforms select libraries from directories on the host in a similar way through the use of an environment variable that contains the search path to be used. For example, the environment variable for UNIX based systems

```
LD_LIBRARY_PATH=.:/opt/gdbm-1.8.3/lib:/net/lib
```

specifies the directories to be searched to locate needed shared libraries (on Windows platforms, the `PATH` variable is used). By altering the path or location of libraries, it is possible that the library that is used for testing is not the same as the one used for operation.

Shared libraries can call other shared libraries. It can be very difficult to exactly determine the location and depth of the dependencies of shared libraries.

Modifying the `LD_LIBRARY_PATH` or `PATH` can alter which shared libraries are loaded. If an attacker is able to insert the `/tmp` path in the library path as follows:

```
LD_LIBRARY_PATH=/tmp:.:/opt/gdbm-1.8.3/lib:/net/lib
```

and inserts a malicious library in the `/tmp` directory, the malicious library will be used instead of the one the developer had intended and tested with the code. Even with the original path:

```
LD_LIBRARY_PATH=.:/opt/gdbm-1.8.3/lib:/net/lib
```

the use of the current directory path, ".", at the start of the library path would mean that if an attacker is able to insert a malicious library in the directory where the code is executed, the malicious library would be used.

C also allows self-modifying code. Since in C there isn't a distinction between data space and code space, executable commands can be altered as desired during the execution of the program. Although self modifying code may be easy to do in C, it can be difficult to understand, test and fix leading to potential vulnerabilities in the code.

Self-modifying code can be done intentionally in C to obfuscate the effect of a program or in some special situations to increase performance. Because of the ease with which executable code can be modified in C, accidental (or maliciously intentional) modification of C code can occur if pointers are misdirected to modify code space instead of data space or code is executed in data space. Accidental modification usually leads to a program crash. Intentional modification can also lead to a program crash, but used in conjunction with other vulnerabilities can lead to more serious problems that affect the entire host.

C.3.49.3 Avoiding the vulnerability or mitigating its effects

- Use signatures to verify that the shared libraries used are identical to the libraries with which the code was tested.
- Do not use self-modifying code except in very rare instances. In those rare instances, self-modifying code in C can and should be constrained to a particular section of the code and well commented.

C.3.49.4 Implications for standardization

Future standardization efforts should consider:
- Standardizing on an easy to use signature mechanism for libraries. Standard C libraries should be signed to allow for verification.

C.3.49.5 Bibliography

C.3.50 Library Signature [NSQ]

C.3.50.0 Status and history

C.3.50.1 Terminology and features

C.3.50.2 Description of vulnerability

Integrating C and another language into a single executable relies on knowledge of how to interface the function calls, argument lists and data structures so that symbols match in the object code during linking. Byte alignments can be a source of data corruption.

For instance, when calling Fortran from C, several issues arise. Neither C nor Fortran check for mismatch argument types or even the number of arguments. C passes arguments by value and Fortran passes arguments by reference, so addresses must be passed to Fortran rather than values in the argument list. Multidimensional arrays in C are stored in row major order, whereas Fortran stores them in column major order. Strings in C are terminated by a null character, whereas Fortran uses the declared length of a string. These are just some of the issues that arise when calling Fortran programs from C. Each language has its differences with C, so different issues arise with each interface.

Writing a library wrapper is the traditional way of interfacing with code from another language. However, this can be quite tedious and error prone.

C.3.50.3 Avoiding the vulnerability or mitigating its effects

- Use a tool, if possible, to automatically create the interface wrappers.
- Minimize the use of those issues known to be error prone when interfacing from C, such as passing character strings, passing multi-dimensional arrays to a column major language, interfacing with other parameter formats such as call by reference or name and receiving return codes.

C.3.50.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.50.5 Bibliography

C.3.51 Unanticipated Exceptions from Library Routines [HJW]

C.3.50.0 Status and history

C.3.50.1 Terminology and features

C.3.50.2 Description of vulnerability

Calling software routines produced outside of the control of the main application developer puts all of the code at the mercy of the called routines. An unanticipated exception generated from a library routine could have devastating consequences.

C.3.50.3 Avoiding the vulnerability or mitigating its effects

- Check the values of parameters to ensure appropriate values are passed to libraries in order to reduce or eliminate the chance of an unanticipated exception

C.3.50.4 Implications for standardization

Future standardization efforts should consider:
None

C.3.50.5 Bibliography
