ISO/IEC JTC 1/SC 22/OWGV N 0204

Meeting #11 markup of draft language-specific annex for C

Date	15 July 2009
Contributed by	Larry Wagoner
Original file name	
Notes	Markup of N0200

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.

Comment [JWM1]: Needs to list the Corrigenda also.

C.2 General Terminology

C.3.1 Obscure Language Features [BRS]

C.3.1.0 Status and history

**needs work – would like to talk about this one **

Comment [JWM2]: In determining what is "obscure", consider the person who has to review the code—often a systems engineer. Features to be treated should include: comma operator. Larry also plans to analyze a large body of code and do some operator counts.

C.3.1.1 Language-specific terminology

C.3.1.2 Description of application vulnerability

C.3.1.3 Mechanism of failure

C.3.1.4 Avoiding the vulnerability or mitigating its effects in C

C.3.1.5 Implications for standardization in C

C.3.1.6 Bibliography

C.3.2 Unspecified Behaviour [BQF]

C.3.2.0 Status and history

C.3.2.1 Language-specific terminology

C.3.2.2 Description of application vulnerability

Unspecified behavior occurs with behavior where the C standard provides two or more possibilities but does not dictate which one is chosen. Unspecified behavior 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 behavior.

Comment [JWM3]: The notion of a "sequence point" should be described. There should be a reference to Annex J of the C standard; it lists all of the instances of unspecified behaviour.

C.3.2.3 Mechanism of failure

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

```
a = i + b[++i];
```

If i is 0 before the assignment, then the result could be

```
a = 0 + b[1];
```

or

```
a = 1 + b[1];
```

C.3.2.4 Avoiding the vulnerability or mitigating its effects in C

Unspecified behavior should not be relied upon because the behavior can change at each instance. Thus, any code that makes assumptions about the behavior of something that is unspecified should be replaced to make it more correct and portable.

C.3.2.5 Implications for standardization in C

None.

C.3.2.6 Bibliography

C.3.3 Undefined Behaviour [EWF]

C.3.3.0 Status and history

C.3.3.1 Language-specific terminology

C.3.3.2 Description of application vulnerability

Undefined behavior is behavior that results from using erroneous constructs and data. The C standard does not impose any requirements on undefined behavior. Typical undefined behaviors include doing nothing, producing unexpected results, and terminating the program.

C.3.3.3 Mechanism of failure

Relying on undefined behavior makes a program unstable and non-portable. While some cases of undefined behavior may be consistent across multiple implementations, it is still dangerous to rely on them. Relying on undefined behavior 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 behavior, but it may work most of the time.

C.3.3.4 Avoiding the vulnerability or mitigating its effects in C

All cases of undefined behavior should be removed from a program before it is considered to be correct.

C.3.3.5 Implications for standardization in C

None.

C.3.3.6 Bibliography

C.3.4 Implementation-defined Behaviour [FAB]

C.3.4.0 Status and history

C.3.4.1 Language-specific terminology

C.3.4.2 Description of application vulnerability

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

C.3.4.3 Mechanism of failure

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

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

```
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.3.4.4 Avoiding the vulnerability or mitigating its effects in C

Reliance on implementation-defined behavior should be eliminated as much as possible from programs in order to increase portability. However, programs that are intended for a specific implementation may rely on implementation-defined behavior.

C.3.4.5 Implications for standardization in C

None.

C.3.4.6 Bibliography

C.3.5 Deprecated Language Features [MEM]

C.3.5.0 Status and history

Needs work

Comment [JWM4]: There is a list of deprecated features in the C standard.

C.3.5.1 Language-specific terminology

C.3.5.2 Description of application vulnerability

The only deprecated function in C is gets. gets copies a string from standard input into a fixed-size array.

C.3.5.3 Mechanism of failure

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.3.5.4 Avoiding the vulnerability or mitigating its effects in C

Since there isn't a safe and secure way of using gets, gets should not be used.

C.3.5.5 Implications for standardization in C

None.

C.3.5.6 Bibliography

C.3.6 Pre-processor Directives [NMP]

C.3.6.0 Status and history
April 23, 2009 – Created.

C.3.6.1 Language-specific terminology

A function-like macro is a macro that takes textual arguments and inserts them into the body of the macro [1]. 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 application vulnerability

The C pre-processor allows the use of macros that are text-replaced before compilation. Macros exhibit numerous potential security flaws because they look similar to C language features but do not behave the same way [2].
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 behavior 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 behavior [2].
Furthermore, standard library functions are typically implemented using macros. Consequently, it is important to treat all externally-defined functions as if they were macros [2].

C.3.6.3 Mechanism of failure

The following code example demonstrates undefined behavior 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 behavior and 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 programmers must be careful when calling externally-defined functions, such as those in the C standard library, because they may be defined as function-like macros. For example, the following code may be undefined depending on the implementation of `memcpy` [3]:

```
memcpy(dest, src,
#ifdef PLATFORM1
  12
#else
  24
#endif
);
```

Because pre-processor directives constitute undefined behavior when supplied as arguments to function-like macros [1], the previous example exhibits undefined behavior if `memcpy` is implemented as a macro.

C.3.6.4 Avoiding the vulnerability or mitigating its effects in C

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

- Where possible, replace macro-like functions with inline functions. Inline functions offer consistent semantics and benefit from static analysis and debugging tools.
- If a function-like macro must be used, make sure that its arguments and body are parenthesized and do not contain pre-processor directives or side-effects, such as assignment, increment/decrement, volatile access, or function call [2].

C.3.6.5 Implications for standardization in C

None.

C.3.6.6 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 Language-specific terminology

C.3.7.2 Description of application vulnerability

C is reasonably 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 compiler may resolve names to another length resulting in unintended behavior.

C.3.7.3 Mechanism of failure

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.3.7.4 Avoiding the vulnerability or mitigating its effects in C

The choice of clear names and non-confusing names is fairly language independent. It is worth repeating that consistency is desirable in choosing names, and to keep names short in order to understand the code easier, but choose names that are rich in meaning. The reality is that code will be reused and combined in ways that developers never imagine.

Because of scoping in C, names should be made distinguishable within the first few characters. This will also assist in averting problems with compilers resolving to a shorter name than was intended.

C.3.7.5 Implications for standardization in C

None.

C.3.7.6 Bibliography

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

C.3.8.0 Status and history

C.3.8.1 Language-specific terminology

C.3.8.2 Description of application vulnerability

C allows filenames and external identifiers to contain what could be unsafe characters or characters in unsafe positions. For example, C allows control characters, spaces, and leading dashes in filenames. The letters “A” through “Z”, “a” through “z”, digits “0” through “0” through “9”, space and any of the characters “% & + , - . : = _” are considered portable. Other characters than these are implementation dependent and may be unintentionally or intentionally misdirected to a filename or other external resource.

C.3.8.3 Mechanism of failure

Filenames may be interpreted unexpectedly. For example, the filename:

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

will result in the file name “??????” when used on ~~a Red Hat Linux distribution~~some implementations while resulting in the filename <whatever> in other systems.

C.3.8.4 Avoiding the vulnerability or mitigating its effects in C

Filenames and external identifier names should be restricted to the safe set mentioned in C.3.8.2.
Check return codes from the use of filenames or other external identifiers.

C.3.8.5 Implications for standardization in C

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

Libraries supporting the safe subset of characters should be included as part of the standard C library.

C.3.8.6 Bibliography

C.3.9 Unused Variable [XYR]

C.3.9.0 Status and history

C.3.9.1 Language-specific terminology

C.3.9.2 Description of application vulnerability
Variables in C may be declared and remain unused. Most compilers will report this as a warning and the warning can be easily resolved by removing the unused variable.

C.3.9.3 Mechanism of failure
Variables may be declared, but never used when writing the code or the need for a variable may be eliminated in the code, but the declaration may remain.
C.3.9.4 Avoiding the vulnerability or mitigating its effects in C
Resolving a compiler warning for an unused variable 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 ignored by the developer.

If there is a need for an unused variable, the some compilers allow the variable to be “tagged” as unused. For example, gcc uses the 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.

C.3.9.5 Implications for standardization in C
None.

C.3.9.6 Bibliography

C.3.10 Identifier Name Reuse [YOW]

Comment [JWM5]: Does not treat CERT DCL-32C.

C.3.10.0 Status and history

C.3.10.1 Language-specific terminology

C.3.10.2 Description of application 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.

C.3.10.3 Mechanism of failure

The value of a in the following example is undefined as the a in the inner block has not been initialized.

```
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.3.10.4 Avoiding the vulnerability or mitigating its effects in C

- 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.3.10.5 Implications for standardization in C

- C should require mandatory diagnostics for variables with the same name in nested scopes.

- C should require mandatory diagnostics for variable names that exceed the length that the implementation considers unique.

C.3.10.6 Bibliography

C.3.11 Type System [IHN]

C.3.11.0 Status and history

C.3.11.1 Language-specific terminology

C.3.11.2 Description of application 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.

C.3.11.3 Mechanism of failure

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:

```
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.
- 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 be difficult as in the following example:

```
short a=10;
int b=1024, c=2048;
long d=800000000,e=16000000000,f;
float g=100.0,h=200.0;
f = ((b + g) * h – a + e) / c;
```

C.3.11.4 Avoiding the vulnerability or mitigating its effects in C

Consideration of the rules 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. Making casts explicit gives the programmer clearer expectations of conversions.

C.3.11.5 Implications for standardization in C

C 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 weak typing provides. It is suggested that when enforcing strong typing does not detract from the good flexibility that C offers (e.g. adding an integer to a character) 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.3.11.6 Bibliography

C.3.12 Bit Representations [STR]

C.3.12.0 Status and history

Note: p.29, line 43 "Problems can arise when programmers mix their techniques to reference the bits or output the bits." is repeated on p.29, line 44.

C.3.12.1 Language-specific terminology

C.3.12.2 Description of application vulnerability

C supports a variety of sizes for integers such as short, int, long and long long. 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.

C.3.12.3 Mechanism of failure

There are a large variety of representations on different platforms and each can produce remarkably different results when bit operations are performed. A few examples are described below.

Bit manipulations are necessary for some applications and may be the reason a particular application was written in C. Although 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, bit manipulations on signed integers are implementation defined in C, as is shifting by an amount greater than the size of the data type. For instance, on a host where ints contain 32 bits,

```
int i,j;
j = i << 45;
```

is undefined as 45 is greater than 32.

The storage representation for interfacing with external constructs can cause unexpected results. Byte orders that are in little endian or big endian format can unexpectedly alter values.

C.3.12.4 Avoiding the vulnerability or mitigating its effects in C

Unless one can ensure that the code is to be executed on a single known platform, bit operations should be avoided. In instances where there is no alternative, then bit operations should be segregated into distinct routines that are clearly labelled as platform-dependent.

In C, bitwise operators should only be used on unsigned integer operators ~~as~~ because the results of some bitwise operations on signed integers is implementation defined.

~~C provides~~Some environments provide 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 where the Least Significant Byte is first with the network byte order, as used on the Internet, where the Most Significant Byte is first.

C.3.12.5 Implications for standardization in C

C.3.12.6 Bibliography

C.3.13 Floating-point Arithmetic [PLF]

C.3.13.0 Status and history

C.3.13.1 Language-specific terminology

C.3.13.2 Description of application vulnerability

C permits the floating point data types float and 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.

C.3.13.3 Mechanism of failure

As with most data types, C is very flexible in how float and 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:

```
float f;
for (f=0.0, f=0.1; f=2.0)
…
```

may or may not terminate after 20 iterations. The representations used for f and the accumulated effect of 20 iterations may cause f to not be identical to 2.0 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.3.13.4 Avoiding the vulnerability or mitigating its effects in C

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. Tests for equality using floats and doubles should check that an acceptable closeness in value has been achieved to avert rounding and truncation problems.

C.3.13.5 Implications for standardization in C

C.3.13.6 Bibliography

C.3.14 Enumerator Issues [CCB]

C.3.14.0 Status and history

C.3.14.1 Language-specific terminology

C.3.14.2 Description of application vulnerability

The enum type in C is used as follows:

```
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.

C.3.14.3 Mechanism of failure

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.3.14.4 Avoiding the vulnerability or mitigating its effects in C

The use of an enumerated type is not a problem if it is well understood what values are assigned to the members. It is safest to use enumerated types in the default form starting at 0 and incrementing by 1 for each member. However, particular uses may dictate the need for starting at a value other than 0, repeating values or having gaps in the sequence of values. If the need is to start from a value other than 0 and have the rest of the values be sequential, the following format should be used:

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

If gaps are needed or repeated values are desired, then the following format should be used:

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

so as to be explicit as to the values in the enum.

C.3.14.5 Implications for standardization in C

C.3.14.6 Bibliography

C.3.15 Numeric Conversion Errors [FLC]

C.3.15.0 Status and history

C.3.15.1 Language-specific terminology

C.3.15.2 Description of application 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 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 the original 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 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.

C.3.15.3 Mechanism of failure

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 C99 integer conversion rules define how C compilers handle conversions. These rules include 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. Prestandard C usually preferred to preserve signedness of the type.

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.
* 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.

C.3.15.4 Avoiding the vulnerability or mitigating its effects in C

Any conversion from a type with larger precision to a smaller precision type could potentially result in a loss of data. To avoid this, steps should be taken to check the value of the larger type before the conversion to see if it is within the range of the smaller type. In some instances, this loss of precision is desired. Such cases should be explicitly acknowledged in comments.

Compiler warnings will indicate implicit casts. Making a cast in C explicit will both remove the warning and acknowledge that the change in precision is on purpose.

C.3.15.5 Implications for standardization in C

C.3.15.6 Bibliography

C.3.16 String Termination [CJM]

C.3.16.0 Status and history

C.3.16.1 Language-specific terminology

C.3.16.2 Description of application vulnerability

A string in C is composed of 0 or more characters followed by a NUL (‘\0’) to indicate the end of the string. Therefore strings in C cannot contain the NUL character. Inserting a NUL in a string either through a bug or through malicious action can truncate a string unexpectedly. Alternatively, not putting a NUL terminator in a string can cause actions such as string copies to continue well beyond the end of the expected string.

C.3.16.3 Mechanism of failure

Overflowing a string buffer through the intentional lack of a NUL termination character can be used to expose information or to execute malicious code.

C.3.16.4 Avoiding the vulnerability or mitigating its effects in C

The Safe String Library (ISO TR24731 specification) provides alternate 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. Alternative implementations are available as part of other platforms.

C.3.16.5 Implications for standardization in C

ISO SC22 WG14 is working on two TRs on safer C library functions. These Extensions to the C Library (TR 24731-1: Part I: Bounds-checking interfaces and TR 24731-2: Part II: Dynamic allocation functions) should be adopted.

All library calls that make assumptions about the occurrence of a string termination character should be modified or deprecated so that the calls no longer rely on a string ending with a NUL character.

The addition of a string construct that does not rely on the NUL termination character should be considered.

C.3.16.6 Bibliography

C.3.17 Boundary Beginning Violation [XYX]

C.3.17.0 Status and history

C.3.17.1 Language-specific terminology

C.3.17.2 Description of application 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.

C.3.17.3 Mechanism of failure

C does not perform bounds checking on arrays. So in C the following code is legal:

```
int t;
int x[20];
t = x[-10];
```

The variable t will be assigned whatever is in the address pointed to by x[-10]. This could be sensitive information or a return address, which if altered by changing the value of x[-10], could change the program flow.

C.3.17.4 Avoiding the vulnerability or mitigating its effects in C

Since C does not perform bounds checking automatically, it is up to the developer to perform range checking before accessing an array. 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.3.17.5 Implications for standardization in C

The addition of a pointer type that would enable bounds checking should be considered.

The addition of an array type that does automatic bounds checking should be considered.

C.3.17.6 Bibliography

C.3.18 Unchecked Array Indexing [XYZ]

C.3.18.0 Status and history

C.3.18.1 Language-specific terminology

C.3.18.2 Description of application 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.

C.3.18.3 Mechanism of failure

C does not perform bounds checking on arrays. So in C the following code is legal:

```
int t;
int x[20];
t = x[-10];
```

The variable t will be assigned whatever is in the address pointed to by x[-10]. This could be sensitive information or a return address, which if altered by changing the value of x[-10], could change the program flow.

C.3.18.4 Avoiding the vulnerability or mitigating its effects in C

Since C does not perform bounds checking automatically, it is up to the developer to perform range checking before accessing an array. 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.3.18.5 Implications for standardization in C

C.3.18.6 Bibliography

C.3.19 Unchecked Array Copying [XYW]

C.3.19.0 Status and history

C.3.19.1 Language-specific terminology

C.3.19.2 Description of application 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.

C.3.19.3 Mechanism of failure

In the interest of ease and efficiency, C library functions such as memcpy() and memmove() 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 amount 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.3.19.4 Avoiding the vulnerability or mitigating its effects in C

Since C array copying functions such as memcpy() and memmove() do not perform bounds checking automatically, it is up to the developer to perform range checking before calling a memory copying function. 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.3.19.5 Implications for standardization in C

Replacement functions that add an extra parameter for the size of the destination buffer have been proposed for memory copying functions. This solution is very easy to circumvent by simply repeating the parameter for the number of bytes to copy as the parameter for the size of the destination buffer. However such attempts to undo well intentioned safer functions can be relatively easily be detected through static code review. It is suggested that safer versions of functions (e.g. memcpy_s()) be added to the standard in addition to retaining the current functions (e.g. memcpy()). This would allow a safer version of memory copying functions for those applications that want to use them in conjunction with code reviews.

C.3.19.6 Bibliography

C.3.20 Buffer Overflow [XZB]

C.3.20.0 Status and history

C.3.20.1 Language-specific terminology

C.3.20.2 Description of application vulnerability

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

C.3.20.3 Mechanism of failure

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 as is done in many other languages. For instance, consider:

```
char buf[10];
for (i=1; i++; i<=10)
  buf[i] = i + 0x40;
```

will write 0x50 to buf[10] which is one beyond the end of 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 program gets 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.\n");
  gets(buf);
  printf ("You typed: %s\n", buf);

  return 0;
}
```

Typing in a string longer than 499 characters (1 less than the buffer length to account for the string termination '\0') will cause the buffer to overflow. A well crafted string that is the input to this program can cause execution of an attacker's malicious code.

C.3.20.4 Avoiding the vulnerability or mitigating its effects in C

There are many ways in which programmers can keep buffer overflows from occurring. Deprecated functions such as gets() should not be used. Length restrictive functions such as strncpy() should be used instead of strcpy(). There are a variety of libraries that implement safer versions of library functions. All input values should be checked. An array index should be checked before use if there is a possibility the value could be outside the bounds of the array. Stack guarding add-ons can be used to prevent overflows of stack buffers. Even using all of these preventive measures may not be able to stop all buffer overflows from happening. However, the use of them can make it much rarer to discover a buffer overflow and much harder to exploit it.

C.3.20.5 Implications for standardization in C

C should continue to deprecate notoriously unsafe functions and offer safer and more secure replacement functions. C should continue to add routines that help programmers avoid buffer overflows. C should make the easier way the more secure way, such as a way of copying an array through the use of a built in operation instead of having programmers write a copy routine that may contain exploitable errors.

C.3.20.6 Bibliography

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

C.3.21.0 Status and history

C.3.21.1 Language-specific terminology

** NOTE: the CERT/CC Guidelines reference in n0191.pdf should be EXP36-C not EXP36-A

C.3.21.2 Description of application vulnerability

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

C.3.21.3 Mechanism of failure

Pointers in C refer to a specific type, such as integer. If sizeof(int) is 4 bytes, ptr is a pointer to integers and 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.3.21.4 Avoiding the vulnerability or mitigating its effects in C

Maintaining the same type can avert errors introduced through conversions. Compiler warnings will usually be issued for pointer conversion instances. These warnings can be more seriously heeded if the decision is made to avoid conversions so any warnings will be addressed. Note that casting into and out of "void *" pointers will most likely not generate a compiler warning as this is legal in both C99 and C90.

C.3.21.5 Implications for standardization in C

C.3.21.6 Bibliography

C.3.22 Pointer Arithmetic [RVG]

C.3.22.0 Status and history

C.3.22.1 Language-specific terminology

C.3.22.2 Description of application 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. Failing to understand how pointer arithmetic works can lead to miscalculations that result in serious errors, such as buffer overflows.

C.3.22.3 Mechanism of failure

When C does arithmetic involving a pointer, the operation is done relative to the size of the pointer's target. For instance, 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.

C.3.22.4 Avoiding the vulnerability or mitigating its effects in C

Due to the error prone nature of pointer arithmetic, some C guidance recommends an outright ban on pointer arithmetic. If pointer arithmetic is to be used, it must be used carefully to avoid the common pitfalls. For instance, in checking the end of an array, the following method is recommended:

```
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.3.22.5 Implications for standardization in C

Although pointer arithmetic is error prone, the flexibility that it offers offsets alternatives such as restrictions that could be placed on it.

C.3.22.6 Bibliography

C.3.23 Null Pointer Dereference [XYH]

C.3.23.0 Status and history

C.3.23.1 Language-specific terminology

C.3.23.2 Description of application vulnerability

C allows memory to be dynamically allocated primarily through the use of malloc(), calloc(), 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 the value return will be NULL. Other operations or faults in logic can result in a memory pointer to be 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.

C.3.23.3 Mechanism of failure

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

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

Malloc() will return the address of the memory allocation or NULL if insufficient memory is available for the allocation. It is good practice 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 */
```

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 set to NULL as good practice would indicate.

C.3.23.4 Avoiding the vulnerability or mitigating its effects in C

Before dereferencing a pointer in C, it can be checked to see that it is not equal to NULL. 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.3.23.5 Implications for standardization in C

Since pointer dereference problems can be difficult to determine, a compiler option would force a check for non-NULL before each memory dereference.

C.3.23.6 Bibliography

C.3.24 Dangling Reference to Heap [XYK]

C.3.24.0 Status and history

C.3.24.1 Language-specific terminology

C.3.24.2 Description of application vulnerability

C allows memory to be dynamically allocated primarily through the use of malloc(), calloc(), 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.

C.3.24.3 Mechanism of failure

Consider the following segment of code:

```
        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 freed memory in C is undefined. Depending on the execution path taken in the program, the 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, overwriting of another piece of data resulting in data corruption may occur. Determining that a dangling reference is the cause of a problem and locating it can be very difficult.

C.3.24.4 Avoiding the vulnerability or mitigating its effects in C

After every free() call, the pointer should be set to NULL as illustrated in the following code:

```
free (ptr);
ptr = NULL;
```

Note that this will not mitigate all dangling reference problems as the following code segment shows:

```
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*/
        …
}
…
```

This situation can be avoided by not setting and using additional pointers to dynamically allocated memory and only using the pointer used to allocate the memory.

C.3.24.5 Implications for standardization in C

The free(ptr) should be modified so that it sets ptr to NULL to prevent its reuse.

C.3.24.6 Bibliography

C.3.25 Templates and Generics [SYM]

Does not apply to C.

C.3.25.0 Status and history

C.3.25.1 Language-specific terminology

C.3.25.2 Description of application vulnerability

C.3.25.3 Mechanism of failure

C.3.25.4 Avoiding the vulnerability or mitigating its effects in C

C.3.25.5 Implications for standardization in C

C.3.25.6 Bibliography

C.3.26 Inheritance [RIP]

Does not apply to C.

C.3.26.0 Status and history

C.3.26.1 Language-specific terminology

C.3.26.2 Description of application vulnerability

C.3.26.3 Mechanism of failure

C.3.26.4 Avoiding the vulnerability or mitigating its effects in C

C.3.26.5 Implications for standardization in C

C.3.26.6 Bibliography

C.3.27 Initialization of Variables [LAV]

C.3.27.0 Status and history

C.3.27.1 Language-specific terminology

C.3.27.2 Description of application 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.

C.3.27.3 Mechanism of failure

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

C.3.27.4 Avoiding the vulnerability or mitigating its effects in C

In most cases, compilers warn about uninitialized variables. These warnings should be resolved as recommended to achieve a clean compile at high warning levels.

Additionally, memory allocated by functions such as malloc() should not be used before being initialized as its contents are indeterminate.

C.3.27.5 Implications for standardization in C

C.3.27.6 Bibliography

C.3.28 Wrap-around Error [XYY]

C.3.28.0 Status and history

C.3.28.1 Language-specific terminology

C.3.28.2 Description of application 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.

C.3.28.3 Mechanism of failure

Consider the following code for a short int containing 16 bits:

```
short i;
i = 65535;
i++;
```

would result in i containing the value -65536. Manipulating I 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, so the following code would be implementation dependent:

```
short i;            /* short int is 16 bits */
i = 0x1357;
j = 21;
k = i >> j;
```

which can yield unexpected results.

C.3.28.4 Avoiding the vulnerability or mitigating its effects in C

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
```

Defensive programming techniques should be used 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.

Bit manipulations in C should be conducted only 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.3.28.5 Implications for standardization in C

C.3.28.6 Bibliography

C.3.29 Sign Extension Error [XZI]

C.3.29.0 Status and history

C.3.29.1 Language-specific terminology

C.3.29.2 Description of application vulnerability

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

C.3.29.3 Mechanism of failure

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 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.3.29.4 Avoiding the vulnerability or mitigating its effects in C

Conversions from one data type to another should be performed with the appropriate conversion routines. Using an unsigned conversion routine to convert a signed integer type to a larger integer data type can yield unexpected results.

C.3.29.5 Implications for standardization in C

C.3.29.6 Bibliography

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

C.3.30.0 Status and history

C.3.30.1 Language-specific terminology

C.3.30.2 Description of application vulnerability

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

C.3.30.3 Mechanism of failure

Consider the following:

```
a | 0x7  == 0
```

designed to mask off and test the lower three bits of "a". However, due to the precedence rules in C, the effect of this expression is to perform the "0x7 == 0" and then bitwise OR that with "a" which may or may not just happen to be the correct answer.

C.3.30.4 Avoiding the vulnerability or mitigating its effects in C

Parenthesis should be used generously to avoid any uncertainty or lack of portibility in the order of evaluation of an expression. If parenthesis were used in the previous example, as in:

```
(a | 0x7) == 0
```

the order of the evaluation would be clear and as expected.

C.3.30.5 Implications for standardization in C

It is suggested that the C language committee join with other language designers to create one or at most a few accepted precedence orders. Standardizing on one or a few will help to eliminate the confusing intricaticies that exist between languages. 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.3.30.6 Bibliography

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

C.3.31.0 Status and history

C.3.31.1 Language-specific terminology

C.3.31.2 Description of application vulnerability

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

```
i = v[i++];
```

then since the order of evaluation is undefined, 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.

Assignments statif ((a == b) | (c = (d-1))) /* the assignment to c may not occur if a==b*/

C.3.31.3 Mechanism of failure

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 behavior, 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.3.31.4 Avoiding the vulnerability or mitigating its effects in C

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

C.3.31.5 Implications for standardization in C

C.3.31.6 Bibliography

C.3.32 Likely Incorrect Expression [KOA]

C.3.32.0 Status and history

C.3.32.1 Language-specific terminology

C.3.32.2 Description of application 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 conducting statements. This freedom, if misused, can result in unexpected results and potential vulnerabilities.

C.3.32.3 Mechanism of failure

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

```
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 legal 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. This would change the example code to:

```
x = y;
if (x)
 {
   …
}
```

This would make it clear 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 legal as in the following example:

```
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, there is an extremely high chance that the programmer did not intend to put the semi-colon there.

C.3.32.4 Avoiding the vulnerability or mitigating its effects in C

The flexibility of C permits a programmer to create extremely complex expressions. The following sub-expression, though legal, would be a nightmare to understand:

```
(h+=*d++-h)&&(-'1'^(h-='1'))&&(i<<=4 & i||!++I--&&(h--||(k|=i))- i/=2;
```

Simplifying statements with interspersed comments would aid considerably in accurately programming functionality and help future maintainers understand the intent and nuances of the code.

Assignments embedded within other statements can be potentially problematic. Each of the following would be clearer and have less potential for problems if the assignments were conducted outside of the expression. Consider:

```
if ((a == b) | (c = (d-1))) /* the assignment to c may not occur if a==b*/
```

or:

```
foo (a=b, c);
```

Each is a legal C statement, but each may have unintended results.

C.3.32.5 Implications for standardization in C

C.3.32.6 Bibliography

C.3.33 Dead and Deactivated Code [XYQ]

C.3.33.0 Status and history

C.3.33.1 Language-specific terminology

C.3.33.2 Description of application vulnerability

As with any programming language that contains branching statements, C can potentially contain dead code. It is of concern primarily since dead code reveals 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 type 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 are 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.3.33.3 Mechanism of failure

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:

```
if (a = b)
…
```

can cause portions of code to become dead code.

The comment indicator in C can cross many lines of code until an ending comment indicator is found. For example,

```
/* this is a comment
i = 1;
j = 2;
/* this is another comment */
```

Inadvertently leaving out the ending comment indicator on the first line will cause the two assignment statements to be considered as part of the comment as the comment will not end until the end of the fourth line. Eliminating statements can be a problem with multi-line comments, but the opposite is also true. Code that is commented out to deactivate it may become active due to misplaced or inadvertently deleted comment indicators.

C.3.33.4 Avoiding the vulnerability or mitigating its effects in C

Dead code to the extent that it is possible to identify should be eliminated from C programs. Compilers and analysis tools can help identify unreachable code. Deactivated code should be deleted from programs due to the possibility of accidentally activating it.

C.3.33.5 Implications for standardization in C

C.3.33.6 Bibliography

C.3.34 Switch Statements and Static Analysis [CLL]

C.3.34.0 Status and history

C.3.34.1 Language-specific terminology

C.3.34.2 Description of application 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.3.34.3 Mechanism of failure

C contains a switch statement of the form:

```
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.3.34.4 Avoiding the vulnerability or mitigating its effects in C

Only direct fall throughs 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:

```
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 as it is not a direct fall through due to the
*/
        …         /* j++ statement */
  }
```

Except for switches on enumerated type where all possible values can be exhausted, all switch statements should have a default value if only to indicate that there exists a case that was unanticipated and thought impossible by the developers. 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.3.34.5 Implications for standardization in C

It is suggested that C consider adding the “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.3.34.6 Bibliography

C.3.35 Demarcation of Control Flow [EOJ]

C.3.35.0 Status and history

C.3.35.1 Language-specific terminology

C.3.35.2 Description of application vulnerability

C is a block structured language, but is not a comb structured language like Ada or Pascal. Therefore, it may not be readily apparent what statements are part of a loop construct or if statement.

C.3.35.3 Mechanism of failure

Consider the following section of code:

```
int a=0, i;
for (i=1; i<10; i++);
  {
    a = a + I;
  }
```

At first it may appear that a will be a sum of the numbers of 1 through 9. However, even though the code is structured so that the "a = a + 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 + i;" statement to only be executed once. In this case, this mistake may be readily apparent during development or testing. More subtle not cases may 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.3.35.4 Avoiding the vulnerability or mitigating its effects in C

The bodies of if, else, while, for, etc. should always be enclosed in braces. This will reduce confusion and potential problems when modifying the software. For example:

```
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*/
```

All if, else if statements should contain a final else statement or a comment stating why the final else isn't necessary.

C.3.35.5 Implications for standardization in C

C.3.35.6 Bibliography

C.3.36 Loop Control Variables [TEX]

C.3.36.0 Status and history

C.3.36.1 Language-specific terminology

C.3.36.2 Description of application 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 responsibly.

C.3.36.3 Mechanism of failure

Since the modification of a loop control variable within a loop is rarely 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 legal:

```
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 occured.

C.3.36.4 Avoiding the vulnerability or mitigating its effects in C

Although the capability exists in C, it is still considered to be a poor programming practice to modify a loop control variable within a loop.

C.3.36.5 Implications for standardization in C

The addition of 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 structure programming.

C.3.36.6 Bibliography

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

C.3.37.0 Status and history

C.3.37.1 Language-specific terminology

C.3.37.2 Description of application 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 a[10];
int I;
for (i=0, i<=10, i++)
…
```

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

C.3.37.3 Mechanism of failure

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.3.37.4 Avoiding the vulnerability or mitigating its effects in C

Careful programming, testing of border conditions and freely available static analysis tools can be used to detect off by one errors in C.

C.3.37.5 Implications for standardization in C

C.3.37.6 Bibliography

C.3.38 Structured Programming [EWD]

C.3.38.0 Status and history

C.3.38.1 Language-specific terminology

C.3.38.2 Description of application vulnerability

It is as easy to write structured programs in C as it is not to. C contains *goto*, *continue*, *break* and other branching constructs that can create very unstructured code 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.

C.3.38.3 Mechanism of failure

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.3.38.4 Avoiding the vulnerability or mitigating its effects in C

In an attempt to encourage more structured programming, many guidance documents for programming in C restrict the use of goto, continue, break and other branching constructs and 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 test 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. Adherence to writing clear and concise structured code should be the goal to make the code as understandable as possible. 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.3.38.5 Implications for standardization in C

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's (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.3.38.6 Bibliography

C.3.39 Passing Parameters and Return Values [CSJ]

C.3.39.0 Status and history

C.3.39.1 Language-specific terminology

C.3.39.2 Description of application vulnerability

Contrary to the wishes and beliefs of some, not everything that is useful is only written in C. At times, it is useful to interface 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.3.39.3 Mechanism of failure

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). Even minor issues 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.3.39.4 Avoiding the vulnerability or mitigating its effects in C

Interfacing with other languages can be error prone. There exist interface packages for many language combinations that 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. It is recommended that additional testing be conducted on sections of code that interface with other languages.

C.3.39.5 Implications for standardization in C

A standardized interface package should be developed 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.3.39.6 Bibliography

C.3.40 Dangling References to Stack Frames [DCM]

C.3.40.0 Status and history

C.3.40.1 Language-specific terminology

C.3.40.2 Description of application 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 expired as the memory has now been recycled for some other use.

C.3.40.3 Mechanism of failure

Using the address of a local variable or other part of perishable memory after the lifetime has expired can lead to unexpected results.

C.3.40.4 Avoiding the vulnerability or mitigating its effects in C

In order to avoid the possibility of a dangling reference, the address of an object should not be assigned to an object which persists after the object has ceased to exist. Once the object ceases to exist, then so will the stored address of the object preventing accidental dangling references.

C.3.40.5 Implications for standardization in C

C.3.40.6 Bibliography

C.3.41 Subprogram Signature Mismatch [OTR]

C.3.41.0 Status and history

C.3.41.1 Language-specific terminology

C.3.41.2 Description of application vulnerability

Functions in C may be called with more or less than the number of parameters the receiving function expects. Some compilers do not produce a warning about this in some situations. This can lead to unexpected results when the count or types of the parameters differs from the calling to the receiving function.

C.3.41.3 Mechanism of failure

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.

If too few arguments are sent to a function, then the function will still pop the expected number of arguments from the stack leading to unexpected results.

C.3.41.4 Avoiding the vulnerability or mitigating its effects in C

C allows function prototypes that allow a declaration of a function with its expected parameters which allows 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.

C also allows functions with a variable number of arguments such as is used in printf(). Because a variable argument feature is difficult to use in a type safe manner, it is recommended that the variable argument feature not be used except in rare instances such as printf().

C.3.41.5 Implications for standardization in C

C.3.41.6 Bibliography

C.3.42 Recursion [GDL]

C.3.42.0 Status and history

C.3.42.1 Language-specific terminology

C.3.42.2 Description of application vulnerability

C permits recursive calls and the use of recursion can make implementation of some mathematical functions much simpler. 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. As C is frequently cited for its performance efficiency, recursion is usually very inefficient both in execution time and memory usage.

C.3.42.3 Mechanism of failure

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.3.42.4 Avoiding the vulnerability or mitigating its effects in C

Although recursion can shorten programs considerably, there is a high performance penalty which is contrary to the usual efficiency of C. Recursion should in general not be used. Only in very rare instances should it be used and only if it can be proven that adequate resources exist to support the maximum level of recursion possible.

C.3.42.5 Implications for standardization in C

C.3.42.6 Bibliography

C.3.43 Returning Error Status [NZN]

C.3.43.0 Status and history

C.3.43.1 Language-specific terminology

C.3.43.2 Description of application vulnerability

C provides the include file errno.h that contains a large number of defined error values. Though these values are defined, inconsistencies in responding to error conditions can lead to vulnerabilities.

C.3.43.3 Mechanism of failure

C.3.43.4 Avoiding the vulnerability or mitigating its effects in C

The C standard library functions provide an error status as the return value and sometimes in an additional global error value. Any returned error statuses should be checked upon return from a function.

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`. Using errno_t will make it readily apparent that the function is returning an error code.

C.3.43.5 Implications for standardization in C

C 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.3.43.6 Bibliography

C.3.44 Termination Strategy [REU]

C.3.44.0 Status and history

C.3.44.1 Language-specific terminology

C.3.44.2 Description of application vulnerability

C provides several ways of terminating a program including *abort(), exit(), _Exit()* and a *return()* issued from the main() program. Some of these are clearly defined as to their functionality and what clean-up will occur (removal of temporary files, flushing of buffers, etc.), while others are implementation defined. This can leave a system in an unexpected state.

C.3.44.3 Mechanism of failure

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. Creating a situation where a C program uses abort() to halt the program may leave temporary buffers or files available for perusal.

C provides the function *atexit()* that allows functions to be registered so that when exit() is invoked, these 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.3.44.4 Avoiding the vulnerability or mitigating its effects in C

The cleanest way to exit in C is to perform a return from the main() program. However, quickly exiting from a deeply nested function may require the use of exit(). There may be situations where an abrupt halt is needed in which case calling abort() may be desirable. If abort() is necessary, the design should protect critical data from being exposed after an abrupt halt of the program.

C.3.44.5 Implications for standardization in C

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.3.44.6 Bibliography

C.3.45 Extra Intrinsics [LRM]

C.3.45.0 Status and history

C.3.45.1 Language-specific terminology

needs work

C.3.45.2 Description of application vulnerability

C.3.45.3 Mechanism of failure

C.3.45.4 Avoiding the vulnerability or mitigating its effects in C

C.3.45.5 Implications for standardization in C

C.3.45.6 Bibliography

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

C.3.46.0 Status and history

C.3.46.1 Language-specific terminology

C.3.46.2 Description of application vulnerability

One 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 Mechanism of failure

C is fairly liberal in its freedom to manipulate the contents of memory. The use of *unions* can lead to unexpected interpretations of data.

C.3.46.4 Avoiding the vulnerability or mitigating its effects in C

The unions should be avoided as it is relatively easy for there to exist an unexpected program flow that leads to a misinterpretation of the union data.

C.3.46.5 Implications for standardization in C

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 source of errors. Therefore many C programming guidelines specifically prohibit the use of unions. It is suggested that unions be considered for deprecation in C.

C.3.46.6 Bibliography

C.3.47 Memory Leak [XYL]

C.3.47.0 Status and history

C.3.47.1 Language-specific terminology

C.3.47.2 Description of application vulnerability

C is very prone to memory leaks as many programs use dynamically allocated memory and C does not have a built in garbage collector to reclaim unreachable memory.

C.3.47.3 Mechanism of failure

Memory is dynamically allocated in C using a library call such as *malloc* and when the program no longer needs the memory, it can be released using library call such as *free*. Should there be a flaw in the logic of the program, memory is allocated but becomes unreachable or more and more memory is allocated, but is not used even though it is still reachable.

C.3.47.4 Avoiding the vulnerability or mitigating its effects in C

There are several very good debugging tools that can be used to help identify unreachable memory. Additionally, built in garbage collectors are available that replace the usual *malloc* memory allocator to allow memory to be recycled when it is no longer reachable. Some performance degradation may occur, so for particular instances, this may need to be used with caution.

C.3.47.5 Implications for standardization in C

C.3.47.6 Bibliography

C.3.48 Argument Passing to Library Functions [TRJ]

C.3.48.0 Status and history

C.3.48.1 Language-specific terminology

C.3.48.2 Description of application 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.

C.3.48.3 Mechanism of failure

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.3.48.4 Avoiding the vulnerability or mitigating its effects in C

No assumptions should be made about the values of parameters or whether 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.

C.3.48.5 Implications for standardization in C

It is suggested that a naming standard for routines be made where 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.3.48.6 Bibliography

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

C.3.49.0 Status and history

C.3.49.1 Language-specific terminology

C.3.49.2 Description of application vulnerability

C compilers allow dynamically linked libraries also known as shared libraries. 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.

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.

C.3.49.3 Mechanism of failure

Code is designed and tested using a suite of shared libraries which are loaded at execution time. The libraries are selected from directories on the host by using an environment variable that contains the search path to be used. 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.

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.3.49.4 Avoiding the vulnerability or mitigating its effects in C

Code should be as tested. Signatures can be used to verify that the shared libraries used are identical to the libraries with which the code was tested.

Except in very rare instances, C code should not be self-modifying. In those rare instances, self-modifying code in C can and should be very contained to a particular section of the code.

C.3.49.5 Implications for standardization in C

C should consider standardizing on an easy to use signature mechanism for libraries. All standard C libraries should have the signature.

C.3.49.6 Bibliography

C.3.50 Library Signature [NSQ]

C.3.50.0 Status and history

C.3.50.1 Language-specific terminology
needs work

C.3.50.2 Description of application vulnerability

C.3.50.3 Mechanism of failure

C.3.50.4 Avoiding the vulnerability or mitigating its effects in C

C.3.50.5 Implications for standardization in C

C.3.50.6 Bibliography