ISO/IEC JTC 1/SC 22/OWGV N 0221

Draft C Annex

Date	24 August 2009
Contributed by	Larry Wagoner
Original file name	c language annex partial 082509.docx
Notes	Replaces N0215

C.3.10 Identifier Name Reuse [YOW]

C.3.10.0 Status and history

C.3.10.1 C-specific terminology and features

C.3.10.2 Description of vulnerability in C

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

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

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

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

C.3.10.3 Avoiding the vulnerability or mitigating its effects 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.3.10.4 Implications for standardization in C

Future standardization efforts should consider:

- Requiring mandatory diagnostics for variables with the same name in nested scopes.
- Requiring mandatory diagnostics for variable names that exceed the length that the implementation considers unique.

C.3.10.5 Bibliography

C.3.11 Type System [IHN]

C.3.11.0 Status and history

C.3.11.1 C-specific terminology and features

C.3.11.2 Description of vulnerability in C

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 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.
- The rank of `_Bool` shall be less than the rank of all other standard integer types.

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

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

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

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

```
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;
```

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

C.3.11.3 Avoiding the vulnerability or mitigating its effects 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.3.11.4 Implications for standardization in C

Future standardization efforts should consider:

- Moving in the direction over time to being a more strongly typed language. Much of the use of weak typing is simply convenience to the developer in not having to fully consider the types and uses of variables. Stronger typing forces good programming discipline and clarity about variables while at the same time removing many unexpected run time errors due to implicit conversions. This is not to say that C should be strictly a strongly typed language – some advantages of C are due to the flexibility that weaker typing provides. It is suggested that when enforcement of strong typing does not detract from the good

flexibility that C offers (e.g. adding an integer to a character to step through a sequence of characters) and is only a convenience for programmers (e.g. adding an integer to a floating point), then the standard should specify the stronger typed solution.

C.3.11.5 Bibliography

C.3.12 Bit Representations [STR]

C.3.12.0 Status and history

C.3.12.1 C-specific terminology and features

C.3.12.2 Description of vulnerability in C

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

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

```
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 may be in little endian or big endian format and unknowingly switching between the two can unexpectedly alter values.

C.3.12.3 Avoiding the vulnerability or mitigating its effects in C

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

Functions such as `htonl()`, `htons()`, `ntohl()` and `ntohs()` are commonly available to convert from host byte order to network byte order and vice versa. This would be needed to interface between an i80x86 architecture where the Least Significant Byte is first with the network byte order, as used on the Internet, where the Most Significant Byte is first.

C.3.12.4 Implications for standardization in C

Future standardization efforts should consider:

None

C.3.12.5 Bibliography

C.3.13 Floating-point Arithmetic [PLF]

C.3.13.0 Status and history

C.3.13.1 C-specific terminology and features

C.3.13.2 Description of vulnerability in C

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

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

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

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

Similarly, the Boolean test

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

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

C.3.13.3 Avoiding the vulnerability or mitigating its effects 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.3.13.4 Implications for standardization in C

Future standardization efforts should consider:

None

C.3.13.5 Bibliography

C.3.14 Enumerator Issues [CCB]

C.3.14.0 Status and history

C.3.14.1 C-specific terminology and features

C.3.14.2 Description of vulnerability in C

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

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

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

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

This would result in:

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

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

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

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

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

C.3.14.3 Avoiding the vulnerability or mitigating its effects 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.3.14.4 Implications for standardization in C

Future standardization efforts should consider:
None

C.3.14.5 Bibliography

C.3.15 Numeric Conversion Errors [FLC]

C.3.15.0 Status and history

C.3.15.1 C-specific terminology and features

C.3.15.2 Description of vulnerability in C

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

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

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

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

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

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

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

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

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

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

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

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

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

C.3.15.3 Avoiding the vulnerability or mitigating its effects 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.3.15.4 Implications for standardization in C

Future standardization efforts should consider:
None

C.3.15.5 Bibliography

C.3.16 String Termination [CJM]

C.3.16.0 Status and history

C.3.16.1 C-specific terminology and features

C.3.16.2 Description of vulnerability in C

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

C.3.16.3 Avoiding the vulnerability or mitigating its effects in C

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

C.3.16.4 Implications for standardization in C

Future standardization efforts should consider:

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

- Modifying or deprecating all library calls that make assumptions about the occurrence of a string termination character so that the calls no longer rely on a string ending with a null termination character.

- Adding a string construct that does not rely on the null termination character.

C.3.16.5 Bibliography

C.3.17 Boundary Beginning Violation [XYX]

C.3.17.0 Status and history

C.3.17.1 C-specific terminology and features

C.3.17.2 Description of vulnerability in C

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

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

```
int t;
int x[] = {0,0,0,0,0,0,0,0,0,0};
t = x[-5];
```

The variable t will be assigned whatever is in the location pointed to by `x[-5]`. For example, this could be sensitive information or even a return address, which if altered by changing the value of `x[-5]`, could change the program flow.

C.3.17.3 Avoiding the vulnerability or mitigating its effects 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.3.17.4 Implications for standardization in C

Future standardization efforts should consider:

- The addition of an array type that does automatic bounds checking.

C.3.17.5 Bibliography

C.3.18 Unchecked Array Indexing [XYZ]

C.3.18.0 Status and history

C.3.18.1 C-specific terminology and features

C.3.18.2 Description of vulnerability in C

C does not perform bounds checking on arrays, so though arrays may be accessed outside of their bounds, the value returned is undefined and in some cases may result in a program ending unexpectantly. For example, in C the following code is legal, though the result is undefined:

```
int t;
int x[] = {0,0,0,0,0};
t = x[10];
```

The variable t will be assigned whatever is in the location pointed to by `x[10]`.

C.3.18.3 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.3.18.4 Implications for standardization in C

Future standardization efforts should consider:

- The addition of an array type that does automatic bounds checking.

C.3.18.5 Bibliography

C.3.19 Unchecked Array Copying [XYW]

C.3.19.0 Status and history

C.3.19.1 C-specific terminology and features

C.3.19.2 Description of vulnerability in C

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

C.3.19.3 Avoiding the vulnerability or mitigating its effects 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.3.19.4 Implications for standardization in C

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

C.3.19.5 Bibliography

C.3.20 Buffer Overflow [XZB]

C.3.20.0 Status and history

C.3.20.1 C-specific terminology and features

C.3.20.2 Description of vulnerability in C

C is a very flexible and efficient language due to its rather lax restrictions on memory manipulations. The same lax restrictions and the popularity of C make C notorious for buffer overflows. 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 often 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.

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

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

```
int main()
{
  char buf[500];
  printf("Type something.\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 null termination character) 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.3.20.3 Avoiding the vulnerability or mitigating its effects in C

There are many ways in which programmers can keep buffer overflows from occurring. The deprecated function `gets()` should not be used. Although not a foolproof solution to buffer overflows, length restrictive functions such as `strncpy()` should be used instead of `strcpy()`. 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. Using all of these preventive measures may still not be able to stop all buffer overflows from happening. However, the use of them can make it much rarer for a buffer overflow to occur and much harder to exploit it.

C.3.20.4 Implications for standardization in C

Future standardization efforts should consider:

- Continuing to deprecate less safe functions where a safer alternative is available such as `strcpy()` and `strcat()`. C should add safer and more secure replacement functions such as `memncpy()` and `memncat()` to complement the `memcpy()` and `memcat()` functions (see in Implications for standardization in C.XYW).

C.3.20.5 Bibliography