ISO/IEC JTC 1/SC 22/WG 23 N 0270

Possible new vulnerability, Buffer overflow (HCB)--Language-independent and C versions

Date	2010-08-31
Contributed by	*John Benito*
Original file name	Buffer Overflow _HCB_.pdf and C - HCB.pdf
Notes	*Action Item #14-08*

6.nn Buffer Boundary Violation (Buffer Overflow) [HCB]

6.nn.1 Description of application vulnerability

A buffer boundary violation arises when, due to unchecked array indexing or unchecked array copying, storage outside the buffer is accessed. Usually boundary violations describe the situation where such storage is then written. Depending on where the buffer is located, logically unrelated portions of the stack or the heap could be modified maliciously or unintentionally. Usually, buffer boundary violations are accesses to contiguous memory beyond either end of the buffer data, accessing before the beginning or beyond the end of the buffer data is equally possible, dangerous and maliciously exploitable.

6.nn.2 Cross reference

CWE:

120. Buffer copy without Checking Size of Input ('Classic Buffer Overflow')
122. Heap-based Buffer Overflow
124. Boundary Beginning Violation ('Buffer Underwrite')
129. Unchecked Array Indexing
787: Out-of-bounds Write

JSF AV Rule: 15 and 25
MISRA C 2004: 21.1
MISRA C++ 2008: 5-0-15 to 5-0-18
CERT C guidelines: ARR30-C, ARR32-C, ARR33-C, ARR38-C, MEM35-C and STR31-C

6.nn.3 Mechanism of failure

The program statements that cause buffer boundary violations are often difficult to find.

There are several kinds of failures (in all cases an exception may be raised if the accessed location is outside of some permitted range of the run-time environment):

- A read access will return a value that has no relationship to the intended value, such as, the value of another variable or uninitialized storage.
- An out-of-bounds read access may be used to obtain information that is intended to be confidential.
- A write access will not result in the intended value being updated and may result in the value of an unrelated object (that happens to exist at the given storage location) being modified.
- When an array has been allocated storage on the stack an out-of-bounds write access may modify internal runtime housekeeping information (for example, a function's return address) which might change a program's control flow.
- An inadvertent or malicious overwrite of function pointers that may be in memory, pointing them to the attacker's code. Even in applications that do not explicitly use function pointers, the

run-time will usually store pointers to functions in memory. For example, object methods in object-oriented languages are generally implemented using function pointers in a data structure or structures that are kept in memory. The consequence of a buffer boundary violation can be targeted to cause arbitrary code execution; this vulnerability may be used to subvert any security service.

6.nn.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:

- Languages that do not detect and prevent an array being accessed outside of its declared bounds (either by means of an index or by pointer[1]).
- Languages that do not automatically allocate storage when accessing an array element for which storage has not already been allocated.
- Languages that provide bounds checking but permit the check to be suppressed.
- Languages that allow a copy or move operation without an automatic length check ensuring that source and target locations are of at least the same size. The destination target can be larger than the source being copied.

6.nn.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Use of implementation-provided functionality to automatically check array element accesses and prevent out-of-bounds accesses.
- Use of static analysis to verify that all array accesses are within the permitted bounds. Such analysis may require that source code contain certain kinds of information, such as, that the bounds of all declared arrays be explicitly specified, or that pre- and post-conditions be specified.
- Sanity checks should be performed on all calculated expressions used as an array index or for pointer arithmetic.

Some guideline documents recommend only using variables having an unsigned data type when indexing an array, on the basis that an unsigned data type can never be negative. This recommendation simply converts an indexing underflow to an indexing overflow because the value of the variable will wrap to a large positive value rather than a negative one. Also some languages support arrays whose lower bound is greater than zero, so an index can be positive and be less than the lower bound.

In the past the implementation of array bound checking has sometimes incurred what has been considered to be a high runtime overhead (often because unnecessary checks were performed). It is now practical for translators to perform sophisticated analysis that significantly reduces the runtime

[1] Using the physical memory address to access the memory location.

overhead (because runtime checks are only made when it cannot be shown statically that no bound violations can occur).

6.nn.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Languages should provide safe copying of arrays as built-in operation.
- Languages should consider only providing array copy routines in libraries that perform checks on the parameters to ensure that no buffer overrun can occur.
- Languages should perform automatic bounds checking on accesses to array elements. This capability may need to be optional for performance reasons.
- Languages that use pointer types should consider specifying a standardized feature for a pointer type that would enable array bounds checking.

C.HCB Buffer Boundary Violation [HCB]

C.HCB.1 Terminology and features

C.HCB.2 Description of vulnerability

A buffer boundary violation condition occurs when an array is indexed outside its bounds, or pointer arithmetic results in an access to storage that occurs outside the bounds of the object accessed.

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. C does not perform bounds checking on arrays, so the following code:

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

will return whatever is in location `x[i]` even if, `i` were equal to -10 or 10 (assuming either subscript was still within the address space of the program). This could be sensitive information or even a return address, which if altered by changing the value of `x[-10]`or `x[10]`, could change the program flow.

The following code is more appropriate and would not violate the boundaries of the array `x`:

```
int foo( const int i) {
      int x[X_SIZE] = {0};
      if (i < 0 || i >= X_SIZE) {
        return ERROR_CODE;
      }

      else {
            return x[i];
      }
}
```

A buffer boundary violation may also occur when coping, initializing, writing or reading a buffer if attention to the index or addresses used are not taken. For example, in the follow move operation there is a buffer boundary violation:

```
char buffer_src[]={“abcdefg”};
char buffer_dest[5]={0};

strcpy(buffer_dest, buffer_src);
```

the `buffer_src` is longer than the `buffer_dest`, and the code does not check for this before the actual copy operation is invoked. A safer way to accomplish this copy would be:

```
char buffer_src[]={“abcdefg”];
char buffer_dest[5]={0};

strncpy(buffer_dest, buffer_src, sizeof(buffer_dest) -1);
```

this would not cause a buffer bounds violation, however, because the destination buffer is smaller than the source buffer, the destination buffer will now hold “abcd”, the 5th element of the array would hold the NULL character.

C.HCB.3 Avoiding the vulnerability or mitigating its effects

- Validate all input values.
- Check any array index before use if there is a possibility the value could be outside the bounds of the array.
- Use length restrictive functions such as `strncpy()`instead of `strcpy()`.
- Use stack guarding add-ons to prevent overflows of stack buffers.
- Do not use the deprecated functions or other language features such as `gets()`.
- Be aware that the use of all of these preventive measures may still not be able to stop all buffer overflows from happening. However, the use of them can make it much rarer for a buffer overflow to occur and much harder to exploit it.
- Use alternative functions as specified in ISO/IEC TR 24731-1:2007 or TR 24731-2:2010. These Technical Reports provides alternative functions for the C Library (as defined in ISO/IEC 9899:1999) that promotes safer, more secure programming. The functions verify that output buffers are large enough for the intended result and return a failure indicator if they are not. Optionally, failing functions call a“"runtime-constraint handle"” to report the error. Data is never written past the end of an array. All string results are null terminated. In addition, the functions in ISO/IEC TR 24731-1:2007 are re-entrant: they never return pointers to static objects owned by the function. ISO/IEC TR 24731-1:2007 also contains functions that address insecurities with the C input-output facilities.

C.HCB.4 Implications for standardization

Future standardization efforts should consider:

- Defining an array type that does automatic bounds checking.
- Deprecating less safe functions such as `strcpy()` and `strcat()` where a more secure alternative is available.
- Defining safer and more secure replacement functions such as `memncpy()` and `memncat()` to complement the `memcpy()` and `memcat()` functions (see in Implications for standardization.XYW).
- Adopting one of the Technical Reports on safer C library functions, Extensions to the C Library (TR 24731-1: Part I: *Bounds-checking interfaces* or TR 24731-2: Part II: *Dynamic allocation functions*, that have been developed by WG 14.
