1. Possible Defects in TR 24731-1

Since the last meeting, I have received email pointing out possible defects in the Bounds-checking TR. This paper summarizes those issues.

2. Typos

2.1 scanf family 6.5.3., 6.9.1.**

In the Runtime-constraints, “Any argument indirected though” should be “Any argument indirected through” (the “r” in “through” is missing). Due to cut’n’paste, the problem occurs with all scanf functions.

Recommendation: add to Technical Corrigendum.

2.2 6.9.3.1 para 1

“Unlike wcrtomb, wcrtoms_s” should be “Unlike wcrtomb, wcrtomb_s” (“wcrtomb_s” is misspelled.

Recommendation: add to Technical Corrigendum.

3. rsize_t

A valued committee member pointed out two issues:

3.1 A careful reading of the TR is required…

It can be challenging for a reader to determine whether the bounds of some of the wide character arrays is the number of wchar_t elements as opposed to number of bytes. This is true: the Standard rarely tells you what the parameters are. Instead, the Standard describes how the parameters are used in the function. The Bounds-checking TR follows in that tradition. I believe the TR is correct, but does require the infamous “careful reading.” For example, **`mbstowcs_s`** (N1225 Subclause 6.6.5.1), the **`dstmax`** parameter is the number of **`wchar_t`** elements in the **`dst`** array.

Recommendation: Leave it alone, or explain the parameters in the Rationale.

3.2 rsize_t number of elements of a wchar_t array

The C Standard declares parameters in library functions to have type **`size_t`** when the parameter is used to hold a size in bytes or the number of elements in an array. In some cases, the arrays have **`char`** elements. In others, the arrays have **`wchar_t`** elements.

Likewise, the TR declares parameters in library functions to have type **`rsize_t`**. Sometimes, the value in the **`rsize_t`** parameteris a number of bytes; sometimes is a number of **`wchar_t`**s. What might be surprising is that the runtime-constraint on the TR functions is that the parameter must have a value less than **`RSIZE_MAX`**.

This was done intentionally, and was discussed in a previous committee meeting. The idea is that **RSIZE_MAX** is an upper bound on the value in an **rsize_t** parameter. The constraint is the same for all functions in the TR, and is easy to remember.

What might surprise some is that if **sizeof (wchar_t)** is 4, then you can potentially copy four times as many bytes using **wmemcpy_s** than **memcpy_s** since the size for **wmemcpy_s** is number of **wchar_t**s, not the number of bytes.

An implementation that sets **RSIZE_MAX** to the maximum number of bytes actually allowed in an object under that implementation could (but is not required to) issue a runtime-constraint if a wide character function had an array of over **RSIZE_MAX** elements. This comes from the license to turn any undefined behavior into a runtime-constraint.

Recommendation: Add this discussion to the Rationale.

4. May programmers lie about bounds?

Mark Terrel asked the question whether we really prohibit programmers from lying about the size of their arrays when calling the bounds-checking functions. It is clear that the functions prohibit accessing elements beyond the bounds that the programmer passes to the functions, but what prohibits the programmer from claiming the bounds are bigger than the destination array if the amount of data copied happens to fit within the true size of the destination.

Mark's message and my reply appear below. Note that the reflector was dropping messages when these were first sent.

Note that the macro RSIZE_MAX is the largest number guaranteed to be not greater than RSIZE_MAX. An evil programmer might think we gave him a convenient macro for the bounds value that turns off bounds checking!

4.1 Mark Terrel's message

Subject: TR24731: Should pointer overflow be a runtime constraint
Date: Tue, 3 Jul 2007 14:56:50 -0700
From: "Mark Terrel" <mterrel@cisco.com>
To: "WG14" <sc22wg14@open-std.org>, <rmeyers@ix.netcom.com>
[Resending due to delivery failure. Apologies if you get this twice.]

Randy and all-

My team came up with a question on strcpy_s() and friends while looking at our implementation. Should we have required a runtime constraint violation when the programmer passes in a buffer that, during the course of performing it's work, might overflow a pointer? And if so, how strict should we be about it? Please consider the following examples and, just for example purposes, assume a platform with 32-bit

memory space and 32-bit pointers. In each case, should the library function be required to detect a runtime constraint violation? Of course any given implemenation could choose to do whatever additional checks they want, so I think the real question is whether this should be a defect against the TR. Alternately, I suppose we could give a little guidance in the rationale document. Also note that I've used strcpy_s here, but similar issues apply to several other functions.

Example 1:
```
// In this example, dst+len wraps around the end of memory.
// But the actual copy does not.
#define RSIZE_MAX 0x80000000
char *dst = malloc(10);  // assume allocated addr > 0x80000000
char *src  = "abc";
strcpy_s(dst, RSIZE_MAX, src);
```

Example 2:
```
// In this example, the copy to dst does wrap around the end of memory.
#define RSIZE_MAX 0x80000000
char *dst  = malloc(5);  // assume allocated addr  = 0xfffffff0
char *src  = "abcdefghijklmnopqrst";  // more than 16 bytes
strcpy_s(dst, RSIZE_MAX, src);
```

We've pondered this for a bit and am interested in what others think. But here's our $.02:

Example 1: We think this is worth checking because clearly the input parameters are incorrect if the operation dst+len causes overflow. This check is inexpensive and automatically encompasses Example 2. The only counter argument that we've come up with is that lazy programmers may wish to actually pass in RSIZE_MAX (or some other arbitrarily large and obviously incorrect size) to purposely disable the buffer length checking. Thus, in most cases, the null terminator would be encountered before causing pointer overflow. While this is a terrible argument, and counter to the whole point of the function, it may become more relevant as systems choose to completely remove the older, unsafe strcpy and friends.

Example 2: This seems like a real corner case to me. Although some embedded 16-bit platforms may often have memory all the way to 0xffff, it seems like there's not much value in trying to catch this error while specifically not catching the error from example 1. If you try to catch the error in example 2 without catching example 1, then you must know the actual length of src to check this. That makes this check much more expensive and therefore, in our humble opinion, not worth doing independent of Example 1. Plus, on most systems, I'm guessing you're more likely to hit the end of your memory space (either virtual or physical) before hitting pointer overflow and I'm fairly certain there's no portable way to check for end of memory.

So I'm interested in 1) whether you think checking for Example 1 is the right approach and 2) whether you think the TR should change to require this.

Thanks in advance for your thoughts.
Mark

4.2 My Reply

Date: Thu, 05 Jul 2007 22:43:45 -0400
From: Randy Meyers <rmeyers@ix.netcom.com>
To: "Mark Terrel (mterrel)" <mterrel@cisco.com>
CC: WG14 <sc22wg14@open-std.org>
Subject: Re: TR24731: Should pointer overflow be a runtime constraint

Our clear intent was that the functions in the Bounds-checking TR prevent buffer overflows. Any attempted use of these functions that lies by exaggerating the number of elements in the destination array completely subverts that goal. I don't think that the committee can bless any such usage under any conditions.

You mention a "terrible argument" (your words) that some programmers might wish to call the new bounds-checking functions with bad lengths because the old functions are flagged or even inaccessible. I agree completely that this is terrible, and I'll observe that such programmers have made their programs much worse than if they left them alone. At least if they were calling the old functions, it would be easy to find all of the dangerous calls. If they start using the new bounds-checking functions with exaggerated bounds, they have all the old problems, but now have made those problems harder to find. That type of usage cannot be tolerated.

As for the examples in your message, there is good justification for the runtime-constraints you wish to add. TR Subclause 6.7.1.3 paragraph 5 says:

"All elements following the terminating null character (if any) written by strcpy_s in the array of s1max characters pointed to by s1 take unspecified values when strcpy_s returns.[38]"

Footnote 38 says:

"This allows an implementation to copy characters from s2 to s1 while simultaneously checking if any of those characters are null. Such an approach might write a character to every element of s1 before discovering that the first element should be set to the null character."

While strcpy_s may stop storing elements of the destination array as soon as it sees a null character, it need not do so. Paragraph 5 gives strcpy_s a license to alter every element of s1 (the destination array) from s1[0] to s1[s1max-1]. Therefore, s1[s1max-1] must be a legitimate element of the array pointed to by s1. If s1[s1max-1] is not a legitimate element, there is undefined behavior.

As you point out, both the TR and its Rationale encourage implementations to diagnose additional cases of undefined behavior by making them runtime-constraints. If an implementation can cheaply diagnose that s1[s1max-1] is not a legitimate element (because the address has wrapped around or the address is too big for the data segment), it is free to trigger a runtime-constraint.

The TR does not require this undefined behavior to be a runtime-constraint. I suspect that the cases where is can be easily diagnosed are too machine specific (like the address arithmetic wrapping) to be a requirement. But, it is a good idea.

Here's the list of functions that include some version of paragraph 5:
gets_s, mbstowcs_s, wcstombs_s, strcpy_s, strncpy_s, strcat_s, strncat_s, wcscpy_s, wcsncpy_s, wcscat_s, wcsncat_s, mbsrtowcs_s, wcsrtombs_s

At one point I suggested to the committee that there be some sort of blanket statement like paragraph 5 applying to all of the functions in the TR. That was rejected basically because any blanket wording would be clumsy (functions don't use the same names for the array and bounds parameters, for example), and the committee worried that a blanket statement might apply in unexpected and undesired ways.

strcpy_s and a few other functions, like memcpy_s, have a runtime-constraint if the objects being copied overlap. I believe that your examples have a 50% chance of triggering the overlap runtime-constraint even without checking the reasonableness of the address arithmetic. I believe that the overlap runtime-constraint is a further argument to programmers that using any sort of exaggerated array bounds is likely trouble (you never know if the source lies within the exaggerated bounds of the destination).

While my "Paragraph 5" and "overlap" arguments apply to the examples in your message, they don't apply to other functions in the TR. For example, calling asctime_s with a destination array of 26 characters but a claimed bounds of RSIZE_MAX characters. The result will fit, but array size is wrong, and simple checks can determine that. I believe the committee has to take the position that:
1. Such programming is bad, and programmers should not do it.
2. An implementation must be free to raise a runtime-constraint.

Perhaps we need to say more to make this clear. I'm not pushing for a DR against the TR, but I don't oppose one either. I could imagine adding to the Description part of each function a paragraph saying something like:

It undefined behavior if s1[s1max-1] is not an element of the array pointed to by s1.

for every array and bounds referenced by the function. That would make clear to programmers that the implementation is free (but not required) to nail them if they exaggerate the array bounds.

Perhaps something could be added to the Rationale instead. I believe that this should be brought up at the next committee meeting. Unless there is strong opposition, I'll submit Mark's and my messages as a paper.