Reply to Austin Group Concerns on PDTR 24731

Introduction

WG14 agrees with the Austin group that there are drawbacks to the functions in DTR 24731, Part 1. However, WG14 believes that there are also large benefits to the functions, and that these functions are the appropriate solution for certain sets of engineering and cost constraints.

These functions are not the only solution to code security remediation, and the committee has changed the name of the document to reflect this philosophy. The new name of the document is *ISO/IEC TR 24731 Extensions to the C Library—Part 1: Bounds-checking interfaces*. The "Part 1" is because the committee expects to produce a part 2 describing a dynamic-storage approach to security remediation. The name change to "Bounds-checking interfaces" is to be more specifically describe that this document concerns itself with bounds-checking stores to fixed-size (non-dynamic) memory buffers.

Both the bounds-checking approach and the dynamic-memory approach have pluses and minuses, and the differing trade-offs between these two sets of functions mean that one set will appeal to users with one set of engineering and cost constraints, and the other set will appeal to users with differing constraints.

The remainder of this document will reply to specific issues raised by the Austin Group. The section titles will be quotes from the Austin Group paper.

"the basic idea has not achieved practical consensus"

There is no "one-size fits all" solution in security remediation. The bounds-checking approach has been adopted by some groups, just as the dynamic memory approach has been adopted by others. This reflects that different groups had different needs and are willing to pay different costs. The lesson of experience here is that it is unwise to present a single solution and expect everyone to believe it is the best for them.

"strcpy_s function is similar to the strlcpy"

Only in the general approach of using bounds checking. There are some important differences that make strcpy_s easier to use and more secure. As will be discussed in further sections, the valid criticisms of strlcpy do not really apply to strcpy_s.

"[using these functions] changes both the API and ABI"

Unfortunately, this is a problem equally true of the dynamic memory approach. Consider the example function needing remediation the Austin paper:

```
void f(char *t, const char *s1, const char *s2) {
  strcpy(t, s1);
  strcat(t, s2);
}
```

After being modified to use dynamic memory management, the function interface is likely to change to either:

```
char *f(const char *s1, const char *s2);
```
or
```
void f(char **t, const char *s1, const char *s2);
```

because the function needs to change from a function where the caller passes in a pointer to the array to hold the result to a function that dynamically allocates an array to hold the result and then pass it back to the caller either as a return value or a pointer parameter.

A much larger interface cost in the dynamic memory solution is that the caller of function **`f`** now is responsible for freeing the memory allocated by **`f`**. In the general case, this is a much harder and error-prone problem to solve. The recent interest in garbage-collecting languages attests to this.

There is no magic bullet in security remediation. Both the bounds-checking approach and the dynamic memory approach in general require API and ABI changes.

"This is cumbersome to write… The version using the proposed interfaces has exactly the same problem"

This appears to be a case of trying to force a valid criticism of strlcpy onto strcpy_s. The original code is:

```
void f(char *t, const char *s1, const char *s2) {
  strcpy(t, s1);
  strcat(t, s2);
}
```

In most cases, after remediation the code will be:

```
void f(char *t, rsize_t tlen, const char *s1, const char *s2){
  strcpy_s(t, tlen, s1);
  strcat_s(t, tlen, s2);
}
```

It is hard to argue that the code is in any way cumbersome to write, to maintain, or to read. It is also fully protected against buffer overrun, null pointers, and even outlandish string lengths.

Because the exception handler is easier and safer to use than checking return values by hand, it is likely to be overwhelmingly preferred by programmers. The feedback from early adopters has been than most applications set the appropriate runtime-constraint handler at program startup, and never touch it again.

It is true that a general purpose library function would likely have to set the runtime-constraint handler to its preferred value upon entry, and to restore it before returning. That operation is reasonably easy to do, and similar to other global or processor state information that freestanding libraries sometimes have to set and restore.

"All too often fixed values…are the basis for overflows"

WG14 agrees that a dynamic approach has several advantages over bounds-checked statically-allocated buffers, and this is motivation for Part 2 of the TR. WG14 and the Austin group are in agreement on this point.

The examples of mixing the Part 1 functions and dynamic allocation are just silly. Real programmers would write those examples this way:

```
size_t len = strlen(name1) + strlen(name2) + strlen(name3) + 1;
char *p = malloc(len);
assert(p != null);
strcpy_s(p, len, name1);
strcat_s(p, len, name2);
strcat_s(p, len, name3);
```

while this code is easily readable, that fact does not mean that a dynamic approach lacks unique advantages or that WG14 should stop investigating Part 2.

"It does leave the programmer the responsibility of adding free(p);"

This is a major drawback to the dynamic allocation approach which the Austin group dismisses far too lightly. The problems of finding the right place to free memory and the right conditions to free it are very hard problems because it is a dynamic property of the executing program rather than a static problem. Solving this problem has been one of the driving forces in the language design of C++, and now the more recent languages that incorporate garbage collection.

Even freeing dynamic memory used only locally in straight-line code can be difficult if the program uses signal handlers and longjmp.

The bugs associated with dynamic memory management are well known, and it is no surprise that many programmers would consider it too expensive in development and debugging to retrofit an existing program to use dynamic memory.

That is not to say that other programmers would not be willing to take on the burden of storage management in order to reap its benefits.

Neither group of programmers is wrong or ignorant of the costs and benefits of dynamic memory versus bounds checking. They are just examples of different engineering needs.

"there is no fixed limit which people wouldn't want to see lifted"

Do not confuse the idea of rsize_t and RSIZE_MAX with a fixed architectural limit like the "640K address space." RSIZE_MAX is a tool to aid in finding program bugs by setting a reasonable limit on lengths of objects manipulated by the TR's functions. RSIZE_MAX is not a fixed architectural limit. Nothing prohibits RSIZE_MAX changing from run to run of the same executable, or even during a single run of an executable. (Although RSIZE_MAX is a macro, it is not a macro required to expand into a constant expression.) The example of an RSIZE_MAX that changes to reflect the amount of memory actually available to the program is given as a possible implementation in the Rational for the TR.

"would prevent using 4 GiB address spaces"

While RSIZE_MAX would likely prevent having a single 4GB string, it would not prevent having lots of strings filling a 4GB address space.

The above statement is true if the implementation chose to follow the recommendation to merely eliminate the "negative" sizes. Implementations that chose not to put any limit would not have any problem with a single 4GB string. Implementations that grow RSIZE_MAX to reflect the size of memory allocated would not have problems with a legitimately allocated single 4GB string.

Even at its most restrictive, RSIZE_MAX does not prohibit programs from filling the address space.

"using rsize_t for different sizes"

It is not really a problem that the same value of RSIZE_MAX as used as a limit for objects measured in different units (chars versus wchar_ts versus element sizes for qsort). First, all of these limits have traditionally been expressed by the same size_t type. Second, the RSIZE_MAX limit is not very precise, but merely a rough and hopefully generous approximation of reasonableness in the size of a single object.

Most existing programs that fill a 4 gigabyte address space would never notice if any single object was limited to only half of memory. Such programs could be rewritten to use the functions in the TR Part 1, and the only time the runtime constraint-handler would be called was when the program contains a legitimate bug (like subtracting pointers in the wrong order, or miscalculating offsets).

"the only valid exception handler … should be the abort_handler"

WG14 agrees that the abort_handler (or a user handler that aborts the program) should be used in most cases to guarantee that a compromised program halts. However, we have received feedback from one early adopter (Oracle) who strongly disagrees. Their application must succeed. For their purposes, they set a handler that logs every function failure. However, the failing function must then return so that they can check the return code and then properly recover from the failed operation.

"make absolutely sure the user notices the problem"

This has been part of the philosophy of the TR. In addition to calling the runtime-constraint handler, in addition to returning a failure code, many of the functions in the TR set their outputs to known bad values if the function fails. If string functions fail, the result string is set to a null string. If memcpy_s fails, it clears the output buffer.

"the programmer makes incorrect assumptions about buffer sizes"

WG14 agrees that if the programmer uses wrong buffer sizes, the bound checking is defeated. This is one of the motivations for Part 2, and one of the tradeoffs that programmers must evaluate when choosing whether to use Part 1 or Part 2 functions.

"strlcpy/strlcat… made C code harder to read and to maintain, while not catching any bugs"

WG14 reviewed the URLs given, and found that the criticisms while valid for strlcpy/strlcat, were not valid for the functions in the TR Part 1. The points raised:

- The return values of the functions were not checked. This is necessary for strlcpy/strlcat, but not necessary for Part 1 functions because of the runtime-constraint handler.
- strlcpy/strlcat quietly truncate their results leading to a possible vulnerability. The Part 1 functions do not truncate, and would call the runtime-constraint handler.

At one point the criticism was made that the strlcat functions should not have been used because it was obvious that the buffer was of correct size, and so the small amount of runtime used to check the bounds was wasted. This issue is so small, one should have argued that the original code should have used a strcpy rather than strcat to copy the first string, since strcpy does not have to waste time finding the null at offset zero.

Conclusion

It is a mistake to assume that problems with strlcpy and strlcat apply equally to the functions in DTR 24731, Part 1. The Part 1 functions allow a much more readable writing style, while providing automatic (but configurable) reporting of any failing function.

The functions in DTR 24731, Part 1, fulfill a real need and a number of groups have expressed an interest in them. The functions are one approach to mitigating security problems; dynamic memory allocating functions are another. The two approaches have different costs and benefits, with neither approach being a solution acceptable to everyone under all conditions. They both fulfill needs that the other cannot satisfy.

Neither is a silver bullet that saves programmers from interface changes or paying a cost to mitigate security problems. That is not to say that different programmers in different situations will see an equal cost for both solutions, or even be unwilling to adopt a more costly solution. Different groups will pick different solutions as being right for them.