Members of the Austin Group have been reviewing the proposed Technical Report on "Bounds Checking Functions" over the last year, and wish to express their concerns over its direction.

The proposed interfaces fail to address many of the aspects related to buffer overflow and as a result are only suitable for a narrow range of applications.

The basic idea embodied by the proposed interface is not a new one. For example, the proposed strcpy_s function is similar to the strlcpy function of OpenBSD 2.4 (1998). However, the basic idea has not achieved practical consensus; on the contrary, for reasons discussed below it has been controversial almost since it was introduced. A Technical Report of type 2 does not seem warranted here: the subject is controversial rather than being under technical development, and mere publication of a TR is unlikely to further consensus.

The core of the problem is that memory handling in C is complicated and error prone. Nobody doubts that improvements in the supporting APIs are useful. However, the existing APIs already provide all the means to write correct programs, although it is often cumbersome to do so. The proposed interfaces don't change that and, to the contrary, can make programs even more complex. A better solution would be to take the memory handling off the hands of the programmer as much as possible.

Let's look at the string functions first. Obviously, code like

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

is bad. But it is not written this way because it is impossible to write correct code. Obviously the length of the target buffer can be passed, though this alters the ABI (see below):

```
void f(char *t, size_t tlen, const char *s1, const char *s2) {
  if (strlen(s1)+strlen(s2) >= tlen) abort();
  strcpy(t, s1);
  strcat(t, s2);
}
```

This is cumbersome to write and slow which is why programmers don't do it. But, more importantly, this kind of change cannot retroactively made because it changes both the API and ABI. New interfaces would have to be introduced (give the new function a different name) but then one might as well write a better function than this. If f() is in a third party library, it cannot change without all the customers of the library changing and recompiling/relinking their applications.

The version using the proposed interfaces has exactly the same problem.

```
void f(char *t, rsize_t tlen, const char *s1, const char *s2) {
  if (strcpy_s(t, tlen, s1) != 0 || strcat_s(t, tlen, s2) != 0)
    abort();
}
```

If anything, this code is even less obvious then the previous version even though it is likely a bit faster.

Even using the exception handler to provide the abort requires the ABI to change:

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

Another problem with respect to the string functions being used to fix up existing code can be demonstrated with this code sequence:

```
char *p = malloc (3 * NAME_LEN);
strcpy (p, name1);
strcat (p, name2);
strcat (p, name3);
```

All too often fixed values like NAME_LEN are used which are the basis for overflows. A programmer could certainly use

```
char *p = malloc(strlen(name1) + strlen(name2) + strlen(name3) + 1);
if (p != NULL)
  {
    strcpy (p, name1);
    strcat (p, name2);
    strcat (p, name3);
  }
```

but this is once again cumbersome and therefore won't be used. Now assume the new string functions. The code to correctly handle the code (more correct than either of the previous two code sequences, this is the goal) could look something like this:

```
rsize_t len = 3 * NAME_LEN;
char *p = NULL;
again:
char *p2 = realloc(p, len);
if (p2 == NULL)
  abort ();
p = p2;
if (strcpy_s(p, len, name1) != 0
    || strcat_s(p, len, name2) != 0
    || strcat_s(p, len name3) != 0)
  {
    len += 2;
    goto again;
  }
```

Nobody can say that this is more appealing to the programmer and it is unlikely that code like this will find its way into many programs.

Once again, this can be written more simply as

```
char *p = malloc(3*NAME_LEN);
set_constraint_handler_s(abort_handler_s);
strcpy_s(p, NAME_LEN, name1);
strcat_s(p, strlen(p)+NAME_LEN, name2);
strcat_s(p, strlen(p)+NAME_LEN, name3);
```

but this still has several vulnerabilities and coding weaknesses, resulting in different end results for p than previously correctly functioning code:

1. Is it certain that name1, name2 and name3 really were a maximum of NAMELEN on entry to this code? What if name1 was NAME_LEN+10, while name2 was NAME_LEN-10?

2. The code needs to be aware of the exception handler; if the handler is not the abort handler, the code *should* check the return code and take appropriate steps if it is to function correctly. A simple drop in replacement as above will fail with the ignore-handler.

The new string functions are meant as an aid to secure existing code bases but the requirement to change ABI (new function parameters, additional elements in structures, etc) plus the added complexity of the code makes the adoption of these interfaces higher unlikely.

Instead, a better approach is to eliminate the requirement on the programmer to deal with the allocation him/herself. The runtime should do this. In this users of systems with the GNU C library can simply use

```
char *p;
if (asprintf(&p, "%s%s%s", name1, name2, name2) == -1)
  abort();
```

The runtime makes sure the target string is large enough and that error conditions (out-of-memory etc) are recognized. It does leave the programmer the responsibility of adding

```
free(p);
```

when he/she is finished with the result, but is this really harder than some of the contortions necessary to use the proposed interfaces?

The concept of rsize_t and RSIZE_MAX also cause confusion. The lesson learned from "640 KiB is enough address space" is that there is no fixed limit which people wouldn't want to see lifted over time.

On 32-bit systems we used to have 2 GiB or up to 3 GiB of address space available for user-level code. Nowadays the whole 4 GiB is available because people asked for it. Any RSIZE_MAX chosen for 2 or 3 GiB address spaces would prevent using 4 GiB address spaces. The same is true for any other limit and it will definitely remain true for 64-bit architectures as well.

Any interface introduced solely for the purpose of using rsize_t instead of size_t is completely unnecessary. Aside from the problem of picking a size, using rsize_t for different sizes like tmpnam_s (for a string) wcscpy_s (a wide char string) and qsort_s (a number of element and a type size) makes no sense. How can a function reject handling strings of, say, 2^20 bytes but allow wcscpy_s to handling 2^22 bytes (on platforms with 4 byte wchar_t)?

All of this functionality can be implemented with the existing implementation. It is always possible for the runtime to determine the maximum possible string length, for example, by looking at the gaps in the address space at startup time. This *dynamically* determined value can then be used for sanity checks; no correct program can ever use larger values. It is therefore no violation of the ISO C to handle these situations as error cases.

Then there are the stdio and string functions which are now supposed to gracefully handle NULL pointers. However, as "drop-in replacements" for the original functions, these are likely to lead to security weaknesses in the application where the unmodified program would crash. Programmers far too often don't check for errors and so NULL pointers and the like are used in places where they shouldn't. Now assume that the stream pointer is supposed to be for a stream where security logs are written to.
If an attacker can overwrite the FILE* value with NULL no more output happens and security problems remain unreported and undetected. The only valid exception handler in these cases should be the abort_handler.

There are a myriad of situations like this where unreported invalid input can cause problems. There is a problem with the special handling of printf("%s", NULL) in the GNU C library, which chose to print the string "(null)". In hindsight, this extension probably was not a good idea because it hides problems.

The only reasonable way to handle invalid inputs is by brute force: abort the program or at the very least make absolutely sure the user notices the problem. A Denial of Service attack is much less severe and easier to notice and battle than an attack which causes, for instance, logging to be disabled.

The "drop-in" replacement technique introduces subtle semantic changes in the way an application will behave under a certain set of input data, and these semantic changes may well have unintended side effects that make the resulting program less "secure". As a result, any programmer altering an application to make changes to the interfaces to use these bounds-checking versions *must* do more than simply "drop-in" the new version. Since he/she is going to that much work anyway, moving to a paradigm using malloc'ed memory would be no harder, and is demonstrably safer.

Another problem can arise in cases where the programmer makes incorrect assumptions about buffer sizes. In this case, the programmer believes that he/she has mitigated all buffer overflow problems by using the new interfaces, but in reality there are still buffer overflows possible in the code.

We are not aware of any study demonstrating that the proposed approach leads to safer or more-secure software. On the contrary, when one of us very-briefly attempted to investigate the matter in 2002, he found that the technique (as embodied by the similar strlcpy/strlcat functions of OpenBSD) made C code harder to read and to maintain, while not catching any bugs in the code surveyed. The code surveyed was OpenSSH_3.0.2p1, the then-current version. For some details please see:

http://sources.redhat.com/ml/libc-alpha/2002-01/msg00096.html
http://sources.redhat.com/ml/libc-alpha/2002-01/msg00159.html

In summary, there is no aspect of the proposal which is worth standardizing. Either no change is needed for the new interfaces or there is no chance that the interfaces will find widespread adoption.