Document number: N1967
Authors: Carlos O'Donell, Martin Sebor
Submission Date: September 25, 2015
Subject: Field Experience With Annex K — Bounds Checking Interfaces

Index

About the Authors

Carlos O'Donell is Principal Engineer on the Compiler Toolchain team at Red Hat where he leads the GNU C Library project team responsible for maintaining the Red Hat® Enterprise Linux® and Fedora Linux distributions of the library. Carlos is also one of the Stewards of the GNU C Library, a long time contributor, and a release manager of the project. Prior to joining Red Hat in 2012, Carlos was a software developer on the Sourcery CodeBench and G++ teams at Mentor Graphics (formerly CodeSourcery).

Martin Sebor is Senior Software Engineer on the Compiler Toolchain team at Red Hat where he is responsible for the PowerPC back end. Prior to to joining Red Hat in January 2015, Martin was Technical Leader on the Compiler Toolchain team at Cisco. At Cisco, as one of the authors of the Cisco C Secure Coding Standard, he was involved in deploying the standard and the Cisco implementation of Annex K within the organization.

Summary

Annex K of C11, Bounds-checking interfaces, introduced a set of new, optional functions into the standard C library with the goal of mitigating the security implications of a subset of buffer overflows in existing code. The design of the interfaces has a long history that dates back to 2003.

The annex originated as ISO/IEC TR 24731-1 — Extensions to the C library — Part 1: Bounds-checking interfaces, published in 2007 [TR1]. The technical report was incorporated into the C standard with only minimal changes.

TR 24731-1 is the result of a four-year effort by WG14 that started with the original Proposal for Technical Report on C Standard Library Security submitted on February 24, 2003 [N997], and culminated by a successful publication of the Technical Report in 2007.

With over a decade of both implementation and user experience, this paper takes a critical look at the successes and failures of some of the major goals of the bounds-checking interfaces (*the APIs* for the remainder of this paper) in practice. The paper draws on the experience of the authors with the implementation of the APIs, its deployment in a large code base (50 MLOC), and on the feedback of many software engineers and security experts both at Cisco and from the open source community. The paper concludes in the Suggested Technical Corrigendum by making a recommendation for the committee to consider for the next major revision of the C standard.

Ease Of Adoption

One of the critical objectives of the APIs is that they be easy to deploy in existing code. Adopting the APIs was envisioned to be a straightforward process involving only local edits affecting only a few lines code [RAT, p3]. This section describes some of the challenges that stand in the way of accomplishing this goal.

To ease their adoption in existing code bases, the names and signatures of the APIs were deliberately chosen to be close to those of the corresponding standard functions. The only apparent differences are that the APIs have a `_s` suffix, take an additional argument, and (in the case of the string manipulation functions) return a value of type `errno_t` rather than a pointer to the destination buffer. For example, the API that corresponds to the standard function `strcpy` declared as follows:

```
  char* strcpy (char* restrict s1, const char* restrict s2);
```

is the function `strcpy_s` whose declaration is the following:

```
errno_t strcpy_s (char* restrict s1, rsize_t s1max, const char* restrict s2);
```

The differences are highlighted in red. The additional `s1max` argument specifies the size in characters of the destination buffer pointed to by `s1`. The function return type is an integer. The function returns zero to indicate success and might return a non-zero value on error (termed *runtime-constraint violation* in the specification and the remainder of this paper).

Thanks to the deliberate similarity, users are led to expect that all it takes to adopt the APIs is to perform a search for the standard names, add to each the `_s` suffix, and provide an additional argument specifying the size of the destination buffer. Since the majority of the standard functions cannot fail and simply return their first argument which is commonly ignored (or in some cases reused for convenience as an argument to another similar call), most users also assume that ignoring the return value of the `_s` function is safe as well.

However, these assumptions are wrong for the following reasons:

- The size of the destination buffer isn't always readily available at the site of the replacement and must be computed from the context. In the common case when the same destination buffer is used as an argument to multiple API calls in the same function, it is advantageous to save the result of the size computation in a new temporary variable, possibly adjusting its value as necessary in consecutive calls to the APIs.
- The change to the return type and value from the first argument to an indication of error means that unlike the standard functions, the APIs cannot be passed the results of computations on the first argument and use the return value to store the result, or chain multiple API calls in a single expression. Such uses in existing code require changes when adopting the APIs. For example:

  ```
  strcat (strcpy (d, a), b);
  ```

  must be rewritten as two separate statements:

  ```
  strcpy_s (d, sizeof d, a);
  strcat_s (d, sizeof d, b);
  ```

- Unlike the majority of the corresponding standard functions, the APIs can fail and may indicate failure by returning an error code to their callers (when the runtime-constraint handler returns). Thus, making use of the APIs in many code bases, especially large ones consisting or multiple modules, requires testing their return value and handling possible runtime-constraint violations. Continuing with the example above, the final code would also need to test the results of the two function calls and handle errors:

  ```
  if (strcpy_s (d, sizeof d, a))
      return -1;
  if (strcat_s (d, sizeof d, b))
      return -1;
  ```

As a result, adopting the APIs in an existing code base requires non-trivial changes that are far more intrusive than the simple edits envisioned in the TR 24731-1 Rationale. Testing the return value of each call to the APIs and handling runtime-constraint violations is particularly intrusive. Such changes have been observed to be a source of new defects introduced into the code base in the process, undermining the very purpose their adoption is intended to achieve.

Use In Existing Or New Code

Another important goal of the APIs, to *Provide a library useful to existing code* [RAT, p3], is often emphasized in secure coding standards and guidelines [SECCODE, STR07-C]. Ironically, based on the authors' experience, the APIs often end up required or recommended by corporate security policies primarily or even exclusively for new code instead. This section describes some of the major obstacles in achieving the goal of using the APIs in existing code, examines the reasons why they end up being mainly used in new code, and highlights some of the problems with doing so.

The recommendation to use the APIs in existing code is based on the premise that they will be introduced selectively and only when necessary to mitigate an existing vulnerability. This approach can, in theory, be effective in cases when the handler invoked in response to a runtime-constraint violation can be relied upon to immediately and unconditionally terminate the execution of the program, thus preventing from being exploited beyond denial of service attacks any further flaws exposed by the handler returning control back to the compromised program. This simple approach has the advantage that it doesn't require a deep understanding of affected code, and thus can be carried out even by engineers unfamiliar with it.

However, in practice, and especially in situations where the handler may return to the caller, due to concerns about the risks stemming from the intrusiveness of the use of the APIs in existing code as discussed in Ease Of Adoption above, the APIs ironically end up required or recommended by corporate security policies only or mainly for newly developed code instead.

Requiring to use the APIs in new code as a replacement for the standard functions leads to a paradox. First, it presupposes that the newly developed code is flawed (otherwise, if it were correct, there would not be a need for the extra checking). That isn't in and of itself necessarily a bad position to start with since no code is perfect. However, code that cannot rely on the handler in effect not to return after a runtime-constraint violation must handle the violation itself, sometimes in non-trivial ways. For example, it may be desirable to return an indication of an error to the caller, but before doing so, any acquired resources would need to be freed. Alternatively, it may be appropriate to write an error message to a log file and exit the process. In order to verify that the handling is correct the code must be tested. The challenge is determining how to trigger the runtime-constraint violations to exercise the handling. The paradox is that knowing how to trigger the violation would imply that the programmer who wrote the code knew of the flaw in the code he or she wrote, a highly unlikely scenario. (See also Verification later in this paper.)

A further problem is whether or not to use the APIs in the runtime-constraint violation handling code itself. Statistically, it has been shown that error-handling code tends to be a more frequent source of bugs than other code [CWE-388] and thus subject to exploitation. Therefore, error-handling code would benefit from using the APIs even more than other code. The hard question is how violations encountered in the handling code should be handled. An even tougher problem is how to trigger such violations so the code could be tested. Due to the difficulty of these problems the most common approach is to avoid dealing with them by not testing the return value of the APIs, hoping that any second-order flaws will not be exploited. (See also section Testing Return Values For Errors later in the paper.)

Using the APIs in new code also sometimes gives rise to attempts to use them to validate program input. Such uses are inappropriate because invalid program input is a normal condition expected to arise in routine use of software and not a violation of a constraint inherent in the program. Consequently, invalid input situations must be handled differently from runtime-constraint violations. In particular, aborting in response to invalid input is a security flaw in its own right. This misuse of the APIs is usually found either during unit testing or fuzz testing, but incomplete coverage of such tests can leave some such uses in the code and, when the software makes use of an aborting runtime-constraint handler, making the software vulnerable to denial of service attacks.

Common Mistakes

The close similarity of the names and effects of the APIs to those of their standard counterparts belies some subtle differences in their behavior that tend to lead to mistakes and misuses. The section examines some of those frequently encountered by the authors.

Using Size of Source Instead Of Destination

One of the most common mistakes stems from the APIs taking an additional argument specifying the size of the destination buffer. The mistake is to supply either the size or the length of the source sequence instead of the size of the destination buffer.

For instance, when both the source sequence and the destination buffer are arrays, it's not uncommon to see a `sizeof` expression involving the source as the argument specifying the size of the destination.

```
void func (void)
{
  char source [] = "...";
  char dest [N];
  ...
  strcpy_s (dest, sizeof source, source);
  ...
}
```

Similarly, when the size of the destination buffer isn't readily available at the site of the call to the function (the destination isn't an array) and it must be determined from the context, a frequent error is to use the wrong computation (or reuse an inappropriate existing computation) for the size of the destination buffer. A typical mistake of this sort, one that defeats the primary purpose of using the API, is to pass the result of `strlen(source)` as the size of the destination buffer.

```
void func (const char *source)
{
  char dest [N];
  ...
  strcpy_s (dest, strlen (source), source);
  ...
}
```

Unnecessary Uses

A widespread fallacy originated by Microsoft's deprecation of the standard functions [DEPR] in an effort to increase the adoption of the APIs is that every call to the standard functions is necessarily unsafe and should be replaced by one to the "safer" API. As a result, security-minded teams sometimes naively embark on months-long projects rewriting their working code and dutifully replacing all instances of the "deprecated" functions with the corresponding APIs. This not only leads to unnecessary churn and raises the risk of injecting new bugs into correct code, it also makes the rewritten code less efficient.

As a simple example consider the following function. Astute readers will notice that the function is correct and safe and, provided the `str` argument is a valid pointer to a string, cannot result in a buffer overflow.

```
char* string_dup (const char *str)
{
    char *dup = (char*)malloc (strlen (str) + 1);
    if (dup)
        strcpy (dup, str);
    return str;
}
```

Naively rewriting it to avoid using the "deprecated" `strcpy` might result in the following version which makes an extra pass over the source string without actually improving anything.

```
char* string_dup (const char *str)
{
    char *dup = (char*)malloc (strlen (str) + 1);
    if (dup)
        strcpy_s (dup, strlen (str) + 1, str);
    return str;
}
```

We leave it as an exercise to the reader to rewrite the function to also avoid relying on `strlen` and make use of `strnlen_s` instead.

Duplicating Size Computation

In cases where the result of the computation of the destination size isn't stored in a descriptive variable (typically seen when the destination buffer is dynamically allocated) the computation is often reused as the argument to the calls to the bounds checking functions. When there is more than one such call the duplication runs the risk of the duplicate expressions diverging over time as a result of maintenance or other unrelated code changes. When the size of the destination buffer is decreased without adjusting the duplicated expressions, the checks that use the "stale" size computations become ineffective.

For example, naively introducing the APIs in the following function to make it "more secure":

```
char* make_pathname (const char *dir, const char *fname)
{
    char *pathname = malloc (strlen (dir) + strlen (fname) + 2);
    if (!pathname)
        return NULL;
    strcpy (pathname, dir);
    // ...
    strcat (pathname, "/");
    // ...
    strcat (pathname, fname);
    return pathname;
}
```

sometimes results in duplicating the computation of the space remaining in the destination buffer:

```
char* make_pathname (const char *dir, const char *fname)
{
    char *pathname = malloc (strlen (dir) + strlen (fname) + 2);
    if (!pathname)
        return NULL;
    strcpy_s (pathname, strlen (dir) + strlen (fname) + 2, dir);
    // ...
    strcat_s (pathname, strlen (dir) + strlen (fname) + 2, "/");
    // ...
    strcat_s (pathname, strlen (dir) + strlen (fname) + 2, fname);
    return pathname;
}
```

While the duplication is obvious in the contrived example above it can escape notice of even careful reviewers when the API calls are far apart from one another or when some are added independently of other, already existing, calls to the APIs far enough that they don't appear in the code difference submitted for review.

Assuming `strncpy_s` Zeroes Out Entire Buffer

Another common source of errors is to assume that the `strncpy_s` function zeroes out the destination buffer past the first `NUL` character like `strncpy` does. This functionality was deliberately not carried over to `strncpy_s` since the zeroing out was commonly criticized as an inefficiency. The difference in behavior has been seen to lead to unintended information disclosure vulnerabilities in networking code where the previous contents of the buffer beyond the `NUL` were sent to the client.

For example, naively replacing the call to `strncpy` in the following function:

```
void secure_copy_buffer (char *out, const char *in, size_t out_len)
{
    strncpy (out, in, out_len);
    // ...
}
```

with `strncpy_s` as shown below will leave the bytes past the first NUL character in the destination unchanged when `strlen(in)` is smaller than `out_len`, possibly exposing sensitive data to the reader.

```
void secure_copy_buffer (char *out, const char *in, size_t out_len)
{
    strncpy_s (out, in, out_len);
    // ...
}
```

Testing Return Values For Errors

Finally, as mentioned above, the need to test for and handle errors reported by functions that historically could not fail is typically viewed as either unnecessary, or overly intrusive and difficult, and tends to be omitted without an understanding of the consequences of allowing the affected code to continue to run after a runtime-constraint violation. While the immediate adverse effects of a violation are prevented by the use of one of the APIs, second-order adverse effects may be triggered subsequent to it as a result of later flaws and the unanticipated change in program state caused by the API call returning after a runtime-constraint violation.

On the other hand, in code that does check for and attempts to handle errors reported by the APIs, the new error handling paths tend to be poorly tested (if at all) because the runtime-constraint violation can typically be triggered only once, the first time it is found and before it's fixed. After the flaw is removed, the handling code can no longer be tested and, as the code evolves, can become a source of defects in the program.

As an illustrative example, consider the following function that efficiently concatenates the strings `dir`, `fname`, and `ext` to form a pathname. Since it avoids unsigned integer wrapping, allocates sufficient memory for the result, and handles `malloc` failure, the function is safe.

```
char* make_pathname (const char *dir, const char *fname, const char *ext)
{
    size_t dirlen = strlen (dir);
    size_t filelen = strlen (fname);
    size_t extlen = strlen (ext);

    size_t pathlen = dirlen;

    // detect and handle integer wrapping
    if (   (pathlen += filelen) < pathlen
        || (pathlen += extlen) < pathlen
        || (pathlen += 3) < pathlen)
        return 0;

    char *p, *path = malloc (pathlen);
    if (!path)
        return 0;

    p = memcpy (path, dir, dirlen);
    p [dirlen] = '/';

    p = memcpy (p + dirlen + 1, fname, filelen);
    p [filelen] = '.';

    memcpy (p + filelen + 1, ext, extlen + 1);

    return path;
}
```

Contrast the implementation above with the one below that makes use of the APIs to perform the same concatenation. Particularly noteworthy are the differences highlighted in red: the computations adjusting the value of `p`,t he tests of the return value of each of the calls to `memcpy_s`, the `pathlen` computation, and the error handler at the bottom of the function labeled `error`. The adjustments to `p` had to be made because unlike the standard string manipulation functions, the APIs do not return the value of their first argument. The tests for the return value are necessary because without them the function could return an invalid pathname (with one or more of the components missing). If the resulting pathname were to refer to a file used to store sensitive data, returning an invalid pathname could cause the sensitive file to be created in a directory accessible to an attacker. The `pathlen` computation is necessary because the size of the destination buffer must be adjusted to reflect the prior copy. Finally, the error handler at the bottom of the function must free the allocated memory. Neglecting to include this part might lead to resource exhaustion in repeated failed calls to the function, enabling potential denial of service attacks.

It is important to note that since the original function is safe from buffer overflows, none of the added complexities and related problems noted in the paragraph above could arise in it, and the additional computations and error handling are only necessary as a consequence of using the APIs.

```
char* make_pathname_s (const char *dir, const char *fname, const char *ext)
{
    // choose an appropriate maximum length for the strings
    rsize_t maxlen = PATH_MAX < RSIZE_MAX ? PATH_MAX : RSIZE_MAX;

    rsize_t dirlen = strnlen_s (dir, maxlen);
    rsize_t filelen = strnlen_s (fname, maxlen);
    rsize_t extlen = strnlen_s (ext, maxlen);

    rsize_t pathlen = dirlen;

    // detect and handle integer wrapping in case RSIZE_MAX > SIZE_MAX / 2
    if (    (pathlen += filelen) < pathlen
        || (pathlen += extlen) < pathlen
        || (pathlen += 3) < pathlen)
        return 0;

    char *p, *path = (char*)malloc (pathlen);
    if (!path)
        return 0;

    p = path;

    if (memcpy_s (p, pathlen, dir, dirlen))
        goto error;

    pathlen -= dirlen + 1;
    p [dirlen] = '/';
    p += dirlen + 1;
    if (memcpy_s (p, pathlen, fname, filelen))
        goto error;

    pathlen -= filelen + 1;
    p [filelen] = '.';

    p += filelen + 1;
    if (memcpy_s (p, pathlen, ext, extlen + 1))
        goto error;

    return path;

error:
    free (path);
    return 0;
}
```

Multithread Safety

One of the explicit goals outlined by the original Microsoft proposal and later reinforced in the TR 24731-1 Rationale is to *Support re-entrant code* [RAT, p4]. This goal is even more important today as software increasingly attempts to exploit multiple processors and threads than it was when the APIs were being designed in the early 2000's. The individual APIs achieve the goal by making it possible and efficient for implementations to avoid relying on internal state by letting their callers specify where such state should be stored. However, since virtually every function in the Bounds checking library relies on a single process-global runtime-constraint handler, the APIs are ill-suited for multi-threaded components.

Implementations of the APIs are required to provide a default runtime-constraint handler with implementation-defined semantics that is invoked by the API when it detects a runtime-constraint violation. In particular, it is implementation-defined whether the default handler exits the program or returns to the caller. Implementations are also required to provide two predefined handlers for programs to replace the default handler with.

- `abort_handler_s` that writes a message to `stderr` in implementation-defined format and calls `abort`, and
- `ignore_handler_s` that does nothing and simply returns to its caller.

The `set_constraint_handler_s()` function is provided to make it possible for programs to replace the default implementation-defined handler with one with semantics that better fit a program's security policy (for example, one that reports the runtime-constraint violation in a log file and calls `_Exit()`). The same function is also intended to be used by components of those programs to temporarily replace the currently installed handler with one with known semantics for the duration of their execution, and then restore the original before returning control to the main program. For example, a component that maintains important program state and that cannot risk being aborted in the middle of a computation might need to temporarily replace the installed handler with the provided `ignore_handler_s` that returns, perform the computation, and then either commit the transaction on the program state, restore the original handler, and continue, or roll back the started transaction and abort in case of a runtime-constraint violation.

Unfortunately, since there is only one runtime-constraint handler per-process, components that run in distinct threads cannot make use of the `set_constraint_handler()` function since doing so could change the handler installed and relied on by other threads.

As a result, components in multi-threaded programs must avoid making changes to or assumptions about the nature of the currently installed runtime-constraint handler. Those that can tolerate the risk of a call to any of the APIs of terminating the program must, however, avoid assuming that the handler will not return and instead must check the return value of each call to the APIs. Components that maintain important program state in non-volatile storage must avoid using he APIs between checkpoints altogether since doing otherwise might cause the state to become lost or corrupted as a result of the handler terminating the process.

This problem has been submitted in N1866.

Verification

Since every runtime-constraint violation is by definition a violation of a requirement and thus a logic error in a program, one with potential security consequences, each detected instance of a runtime-constraint violation must be removed from the program. As a result, tests cannot be developed to reliably trigger runtime-constraint violations or to exercise individual runtime-constraint handling code. This in turn leads to the risk that runtime-constraint violation handling code paths contains bugs. Since error checking code is a notorious source of bugs and thus a favorite target of hackers [CWE-388, OWASP07], this is a serious flaw in the design of the Bounds checking APIs.

Customizable Runtime-Constraint Handler

One of the major security concerns with the design of the Bounds checking interfaces is the runtime-constraint handlers in general, and the ability of programs to provide their own runtime-constraint handlers and replace the system-default in particular. The concerns with the thread-safety of the handler is explored in the section titled Multithreaded Safety.

The reason for the concern with the ability to replace the handler is the following. Every runtime-constraint violation represents a potentially exploitable security flaw in the software. Once a program enters an unanticipated state there is no way for it to proceed without potentially opening itself up to the possibility of an attack. Thus, the only way to prevent such an attack is to abort the execution of the program and immediately exit. The ability to install a user-defined handler that gives the compromised program the opportunity to either try to recover from such state, or even temporarily defer termination while some cleanup is done, increases the window of opportunity for an attacker to gain control of the vulnerable program. This concern can be only partially mitigated by enforcing a global security policy that installs a runtime-constraint handler that aborts the program and that forbids programs from even temporarily replacing it with a custom handler. However, this approach imposes severe restrictions on using the APIs in code that maintains important program state.

Efficiency

The use of the APIs introduces a number of inefficiencies into a code base. While some of the inefficiencies could be addressed by improving the quality of implementation of the APIs, others are impossible to resolve without making extensive modifications to compilers, and others still are inherent in the design and thus unavoidable.

1. **Tests for runtime-constraint violations.** Each of the APIs performs a number of tests for runtime-constraint violations. Among those are the following:
    - Verifying that pointers are non-null. This involves one test for each pointer.
    - Verifying that pointers do not point to overlapping regions of memory. For a two-argument function such as `memcpy_s` this computation involves six comparisons and two additions as shown in the snippet below copied from the Safe C [SafeC] implementation of the function:

      ```
      /*
       * overlap is undefined behavior, do not allow
       */
      if( ((dp > sp) &&(dp < (sp + smax))) ||
          ((sp > dp) && (sp < (dp + dmax))) ) {
          mem_prim_set (dp, dmax, 0);
          invoke_safe_mem_constraint_handler ("memcpy_s: overlap undefined",
                                              NULL, ESOVRLP);
          return (ESOVRLP);
      }
      ```

    - Verify that neither the arguments nor the result of the operation exceeds the value of the `RSIZE_MAX` constant and that the size of the destination buffer is sufficient to hold the result. For a two-argument function such as `memcpy_s` this computation involves six comparisons.

2. **Redundant tests in chains of calls.** In chains of calls to the APIs involving the same source sequence or the same destination buffer, most of the tests for runtime-constraint violations in the second and subsequent calls are redundant. Since, unlike many of

the corresponding traditional functions, the APIs are not implemented as compiler intrinsics, the redundant checks typically aren't optimized away, making calls to them much less efficient. (It is possible to optimize some of the redundancy away when using Whole Program Optimization.)

3. **Not expanded inline.** Today's optimizing compilers provide *intrinsics* or *built-ins* as highly efficient equivalents of the traditional C library string manipulation functions and expand calls to the functions inline. Besides avoiding the overhead of a branch instruction to jump to the library implementation of the functions, the intrinsics have the important benefit of enabling other optimizations across multiple calls to the same function that aren't possible otherwise.

   Although the intent of the authors of the original proposal for the APIs was that they would eventually be implemented in compilers in the form of efficient intrinsics, to date this has not happened (not even in the Microsoft compiler). As a result, each call to one of the APIs incurs the overhead of a function call, and redundant tests for runtime-constraint violations aren't eliminated. Since many of the APIs are performance sensitive (most of the string manipulation functions), this makes such APIs unsuitable in performance sensitive situations.

4. **Implemented as wrappers around standard equivalents.** Some of the existing implementations of the APIs are little more than thin "wrappers" for the standard equivalents. Each wrapper starts by performing the necessary checks to detect and diagnosing runtime-constraint violations. When no such violations are detected, the wrapper then calls the corresponding standard equivalent. While such implementations avoid duplicating code, they tend to be quite inefficient.

As an illustrative example of all the problems outlined above, consider the following implementation of the `strncpy_s` function from slibc 0.9.3 [7]. The function starts by performing the required checks of runtime-constraints. This involves 10 tests and branches, plus two calls to the `strnlen_s` function, each of which iterates over all the elements in the source sequence. Since the `strnlen_s` function is not implemented as a compiler intrinsic the second (redundant) call to it cannot be optimized away. Finally, after all the tests pass, the function finally calls the standard function `strncpy` to perform the copy.

Whereas implementations of the standard `strncpy` function avoid all of the runtime-constraint violation tests and make a single pass over the source sequence, this `strncpy_s` makes three. For a performance sensitive string manipulation function that is usually hand-written in assembly for maximum efficiency such an implementation would be unsuitable for most programs.

```
errno_t strncpy_s (char * restrict s1, rsize_t s1max, const char* restrict s2, rsize_t n)
{
  rsize_t m;

  _CONSTRAINT_VIOLATION_IF ((s1 == NULL), EINVAL, EINVAL);
  _CONSTRAINT_VIOLATION_IF ((s1max == 0), EINVAL, EINVAL);
  _CONSTRAINT_VIOLATION_IF ((s1max > RSIZE_MAX), EINVAL, EINVAL);
  _CONSTRAINT_VIOLATION_CLEANUP_IF (s2 == NULL, s1[0] = 0, EINVAL, EINVAL);
  _CONSTRAINT_VIOLATION_CLEANUP_IF (n > RSIZE_MAX, s1[0] = 0,  EINVAL, EINVAL);
  _CONSTRAINT_VIOLATION_CLEANUP_IF ((n >= s1max && s1max <= strnlen_s (s2, s1max)), s1[0] = 0,  EINVAL, EINVAL);

  rsize_t s2_size = MIN (strnlen_s (s2, n) + 1, n);
  _CONSTRAINT_VIOLATION_CLEANUP_IF (REGIONS_OVERLAP_CHECK (s1, s1max, s2, s2_size), s1[0] = 0,  EINVAL, EINVAL);

  m = MIN (n, s1max - 1);
  strncpy (s1, s2, m);
  s1 [MIN (n, s1max - 1)] = '\0';

  return 0;
}
```

Available Implementations

Despite the specification of the APIs having been around for over a decade only a handful of implementations exist with varying degrees of completeness and conformance. The following is a survey of implementations that are known to exist and their status.

While two of the implementations below are available in portable source code form as Open Source projects, none of the popular Open Source distribution such as BSD or Linux has chosen to make either available to their users. At least one (GNU C Library) has repeatedly rejected proposals for inclusion for some of the same reasons as those noted by the Austin Group in their initial review of TR 24731-1 N1106]. It appears unlikely that the APIs will be provided by future versions of these distributions.

Microsoft Visual Studio

Microsoft Visual Studio implements an early version of the APIs. However, the implementation is incomplete and conforms neither to C11 nor to the original TR 24731-1. For example, it doesn't provide the `set_constraint_handler_s` function but instead defines a `_invalid_parameter_handler _set_invalid_parameter_handler(_invalid_parameter_handler)` function with similar behavior but a slightly different and incompatible signature. It also doesn't define the `abort_handler_s` and `ignore_handler_s` functions, the `memset_s` function (which isn't part of the TR), or the `RSIZE_MAX` macro.The Microsoft implementation also doesn't treat overlapping source and destination sequences as runtime-constraint violations and instead has undefined behavior in such cases.

As a result of the numerous deviations from the specification the Microsoft implementation cannot be considered conforming or portable.

Open Watcom

Starting with version 1.5, the Open Watcom compiler ships with an implementation of TR 24731. According to the provided documentation [OWDOC] it defines the `__STDC_LIB_EXT1__` macro to 200509L, indicating that it conforms to the final draft of the Technical Report [FDTR].

Since the final draft of TR 24731 is essentially identical to the published technical report, which is very close to Annex K (although not identical since the `memset_s` function specified by the Annex does not appear in the TR), the Open Watcom implementation can be, at least on paper, be considered a nearly conforming implementation.

Safe C Library

Safe C Library [SafeC] is a fairly efficient and portable but unfortunately very incomplete implementation of Annex K with support for the string manipulation subset of functions declared in `<string.h>`.

Due to its lack of support for Annex K facilities beyond the `<string.h>` functions the Safe C Library cannot be considered a conforming implementation.

Slibc

Slibc is a complete, open source implementation of Annex K designed to be used with GNU C library typically distributed with Linux. The implementation claims to be complete and to fully conform to C11. An inspection of the implementation reveals that it is quite inefficient and thus unsuitable for production use without considerable changes. It does provide a good referefence implementation of the library. A proposal to incorporate slibc into the GNU C library was submitted in 2012 to the GNU C library community and rejected [SLIBC-PROP].

Alternate Solutions

The Bounds checking interfaces rely on the programmer to provide additional arguments to avoid buffer overflows. There are many other solutions to the problem of buffer overflow available today, including ISO/IEC TR 24731-2:2010 — Extensions to the C library — Part 2: Dynamic Allocation Functions [TR2], and the Managed Strings Library [N1158]. They all take the similar approach of providing alternate functions that programmers must be trained to use instead of the standard functions they are used to. They will not be discussed here.

However, since the inception of the Bounds checking interfaces a number of technologies that either weren't available then or weren't viable have become efficient and commonplace. The following is a brief survey of alternate, portable solutions that do not require retraining of programmers or changes to legacy code and that are readily available today.

Address Sanitizer

Clang Address Sanitizer [ASAN] is an efficient detector of the most common kinds of memory errors including buffer overflows, out-of-bounds accesses and heap errors. It relies on compiler instrumentation to detect these kinds of errors. Its advertised runtime slowdown factor is 2X ranking it among the fastest such solutions available. Address Sanitizer, an Open Source project, comes integrated into the Clang and GCC compilers.

Intel Pointer Checker

Intel Pointer Checker [PTRCHK] is a tool for the detection of all incorrect uses of arrays and pointers, including out-of-bounds accesses. Like other similar solutions, it relies on compiler instrumentation of the program and dynamic analysis to detect and prevent memory errors. While reasonably efficient when fully implemented in software, it imposes little to no runtime overhead when used on processors that implement the Intel MPX technology (Memory Protection Extensions) [MPX]. This makes the tool the perfect solution to the buffer overflow problem on Intel hardware.

Object Size Checking

Object Size Checking [OSC] (also known under the moniker `_FORTIFY_SOURCE` based on the name of the macro used to control it) is an extensible feature of the GNU C and C++ compilers and the GNU C library that relies on the compiler's knowledge of data types and sizes to detect a subset of buffer overflows in calls to instrumented functions at compile time and another subset at runtime. Object Size Checking has no runtime overhead but it only prevents buffer overflows in cases when the size of at least one of the buffers being operated on is known. In our experience with deploying the feature in large code base (millions of lines of code), its effectiveness was below 40%.

Static Analysis

A number of dedicated static analysis tools exist that have the ability to track the flow of data through a program and detect limited subsets of buffer overflows. Coverity Prevent has several checkers designed specifically for the detection of buffer overflows: `BUFFER_SIZE`, `SIZECHECK`, `STRING_OVERFLOW`, and `STRING_SIZE`. HP Fortify SCA also has checkers capable of detecting buffer overflows in a program.

Static analysis is also increasingly being incorporated into compilers. The Clang open source compiler offers the Clang Static Analyzer [CLANGSA] as either a standalone or integrated option. Similarly, HP Code Adviser [cadvise] is available as a standalone tool and also comes bundled with the HP aCC compiler. Intel ICC also offers an integrated static analyzer [INTCSA]. Likewise, Microsoft Visual Studio comes with a built-in static analysis tool [MSFTSA]. The advantage of compiler solutions over didicated static analysis tools is that they allow programmers to uncover errors earlier in the development cycle, typically during the build step, and under a familiar user interface.

Valgrind

Valgrind [VALGRIND] is a powerful, feature-rich dynamic analysis tool capable of detecting many kinds of memory management and other bugs. Like many other dynamic analysis tools, it works by instrumenting a program after it has been compiled and linked and detecting errors in code paths exercised by the running program. Valgrind is very powerful but its considerable runtime overhead makes it suitable only as a debugging tool and not as runtime protection facility.

Suggested Technical Corrigendum

Despite more than a decade since the original proposal and nearly ten years since the ratification of ISO/IEC TR 24731-1:2007, and almost five years since the introduction of the Bounds checking interfaces into the C standard, no viable conforming implementations has emerged. The APIs continue to be controversial and requests for implementation continue to be rejected by implementers.

The design of the Bounds checking interfaces, though well-intentioned, suffers from far too many problems to correct. Using the APIs has been seen to lead to worse quality, less secure software than relying on established approaches or modern technologies. More effective and less intrusive approaches have become commonplace and are often preferred by users and security experts alike.

Therefore, we propose that Annex K be either removed from the next revision of the C standard, or deprecated and then removed.

References

1. N997 Proposal for Technical Report on C Standard Library Security, February 24, 2003
2. N1106 — Austin Group Review of ISO/IEC WDTR 24731 Specification for Secure C Library Functions, March 2005
3. ISO/IEC TR 24731-1:2007 Information technology — Programming languages, their environments and system software interfaces — Extensions to the C library — Part 1: Bounds-checking interfaces
4. N1173 — Rationale for TR 24731 Extensions to the C Library Part I: Bounds-checking interfaces
5. ISO/IEC TR 24731-2:2010 Information technology — Programming languages, their environments and system software interfaces — Extensions to the C library — Part 2: Dynamic Allocation Functions
6. Implement C11 annex K? — A discussion on the GNU libc mailing list of a proposal to implement the APIs.
7. ISO/IEC SC22/WG14 N1866 — thread safety of `set_constraint_handler_s`
8. slibc — Implementation of C11 Annex K "Bounds-checking interfaces"
9. The CERT C Secure Coding Standard, Robert C. Seacord
10. Implementation of C11 Bounds-checking interfaces — rejected proposal to include the slibc implementation of Annex K in GNU libc
11. Safe C Library — A partial implementation of Annex K
12. N1199 — Extensions to the C Library — Part I: Bounds-checking interfaces, final draft technical report, October 2006
13. Address Sanitizer — Fast memory error detector
14. Intel Pointer Checker — Easily Catch Out-of-Bounds Memory Accesses
15. Introduction to Intel Memory Protection Extensions
16. Valgrind — An instrumentation framework for building dynamic analysis tools
17. Object Size Checking Built-in Functions — GCC manual
18. Open Watcom Library Reference
19. Deprecated CRT Functions — Microsoft Developer Network
20. CWE-388 : Error Handling, Common Weakness Enumeration, Mitre
21. Information Leakage and Improper Error Handling — OWASP Top 10 2007 bugs
22. N1158 — Specification for Managed Strings

23. HP Code Advisor — HP static analysis tool for C and C++ programs
24. Intel Static Analysis, User And Reference Guide for the Intel C++ Compiler 15.0
25. Analyzing C/C++ Code Quality by Using Code Analysis, Microsoft Visual Studio 2015