Doc No: SC22/WG14/N997

Date: February 24, 2003

Project: JTC1.22.32

Reply to: Martyn Lovell
Microsoft Corp.
1 Microsoft Way
Redmond WA USA 98052-6399
Email: martynl@microsoft.com

Proposal for Technical Report on C Standard Library Security

1 Introduction

When the C standard library was originally designed, as part of the evolution of the Unix operating system and the C language, computing and computer internetworking were in their infancy. Security of internal coding was far less of an issue than it is today.

This has caused many of the functions provided by the C standard library to provide an 'insecure' interface. It is easy to accidentally use these functions in a dangerous fashion. Many of today's security advisories result from such dangerous usage. Common security mistakes like buffer overruns are easily made with many of the these functions.

This paper proposes the creation of a Technical Report (TR) to address security weaknesses and where possible remove them from the C standard library. There are functions in the C standard library whose common use can lead to security issues; may are identified below by way of example. One way the TR could choose deal with this is to adopt alternative versions of each of these functions, and let the existing functions be deprecated.

It is important to note that simply switching to these new functions will not, on its own, make any application secure. Secure coding practices, such as threat modeling, code review and rigorous testing are required to build and deploy secure applications. However, use of these functions should reduce the incidence of trivial coding mistakes that can cause security exposure.

2 Problems addressed

There are three kinds of problems in today's implementations of the C standard library.

- Standard-defined interface problems: Some functions do not include appropriate parameters to allow them to be implemented securely. For example, this includes functions which fill output string buffers but do not allow the caller to specify a buffer size.

Resolving these problems requires a new function with appropriate parameters, and a change to the C Standard. For memorability, one way to distinguish such new functions would be with an _s postfix.

- Standard-defined implementation problems: Some functions have an appropriate interface, but the standard requires their implementation to be insecure. For example, returning a non-terminated string in a buffer.
  Resolving these problems requires a change to the standard, and a change to implementations of the functions, but new functions need not be created.
- Standard agnostic implementation problems: Some functions have an appropriate interface, but the standard allows them to be implemented inappropriately. These are quality-of-implementation issues.
  No standard change is required, though the committee may wish to consider doing so.

Examples of issues of all three kinds are presented below. Quality-of-implementation issues are presented for completeness.

2.1 Interface problems

2.1.1 Output buffer sizes

Functions that take a string output buffer must take a size for that buffer, to avoid writing past its end. For example, strxfrm should take a size, and strcpy should be deprecated in favour of strncpy.

Functions that take a binary output buffer must also take a size for it.

2.1.2 Error return

All functions should be documented to return an error in errno. New functions will use errno as a return value to ensure all functions can be seen to return errors.

2.1.3 Callback context

Several library functions (qsort, bsearch) call back to function pointers provided to them. These callbacks often require the caller to store context in static variables to have it accessible from within the callback function. This can cause dangerous problems with reentrancy. To avoid this, functions should always allow a context value (void *) to be passed in, and will pass this back to the callback function.

2.1.4 Static result buffers

Some library functions return a pointer to a library-internal buffer, and specify that the result will be overwritten by the next call. Though libraries are free to implement such functions with one buffer for each thread (to reduce risk of conflict), there is still a significant risk of buffer overrun caused by reentrancy. For example, tmpnam should return its name in a user-provided buffer, rather than a library-internal buffer.

2.1.5 Replacing variables with functions

When the library exposes a variable directly (such as errno), there is no easy way for it to validate that the variable is only used when it is valid, and only set to valid values. Each variable should be replaced with an appropriate get and set function,

2.1.6 Random number quality

Random number generation and initialisation should be performed in a 'safe' manner, using appropriately cryptographically safe generators. For backwards compatibility, we should probably use a new function name for this, since performance will be slower.

2.2 Standard implementation problems

2.2.1 String terminators

All functions writing strings to buffers should terminate the characters written, or return an error if there is no space for termination. For example, strncpy should always write a terminator.

2.3 Quality of implementation problems

2.3.1 Null pointer checks

All functions should check for invalid or null pointers and fail to act if the input pointers are not valid. For example, if a null pointer is passed to strncat, it should fail.

2.3.2 Parameter validation

Functions should ensure they were provided with appropriate and correct inputs, and return error if not. For example, if an invalid open mode is provided to fopen, the function should fail.

2.3.3 Stack depth

Functions should not copy unbounded user input to the stack, as this can allow a denial of service attack. Long strings should always be allocated on the heap where overflow can be safely dealt with.

2.3.4 File permissions

File functions should default to secure permissions (exclusive/single user), and secure locations (temporary files) to ensure that squatting attacks are not possible. For example fopen should default to exclusive access.

2.3.5 scanf family problems

scanf makes extensive use of unsized buffers. While there is no entirely satisfactory way to fix this, one possible proposal would be to require that each scanf buffer parameter have a size passed. The function already allows this, but it is possible to propose a new function that requires the size.

3 Proposed Technical Report

We propose the creation of a technical report specifying security-related risks and security-enabling changes to the C library in detail, potentially including items such as the following: deprecation of functions that cannot be fixed without change to the function's signature; replacement functions that are secure, specifying the necessary behaviour changes; and new functions to be added. We would also be able to provide various sample implementations of such functions.

4 Appendix: Function Shape Changes

We are working through the whole library to apply these principles. This table summarises the changes we're making to standard functions. We are also making similar changes to our many functions that are extensions to the standard.

Area	Old prototype	New prototype	Security Act

Algorithms	`void *bsearch(`
	`const void *key,`
	`const void *base,`
	`size_t num,`
	`size_t width,`
	`int ( __cdecl *compare ) ( const void *, const void *)`
	`);`
		`void *bsearch_s(`
		`const void *key,`
		`const void *base,`
		`size_t num,`
		`size_t width,`
		`int ( __cdecl *compare ) (void *context, const void *, const void *),`
		`void *context`
		`);`
			Passes context to avoid static vars

Algorithms	`void qsort(`
	`void *base,`
	`size_t num,`
	`size_t width,`
	`int (__cdecl *compare )(const void *, const void *)`
	`);`
		`void qsort_s(`
		`void *base,`
		`size_t num,`
		`size_t width,`
		`int (__cdecl *compare )(void *context, const void *, const void *),`
		`void *context`
		`);`
			Avoid static vars with context

Filesystem	`char *tmpnam(`
	`char *string`
	`);`
		`errcode tmpnam_s(`
		`char *buffer,`
		`size_t`
			Standard validations

		sizeInBytes, char *string);	
General	char *getenv(const char *varname);	errcode getenv_s(char *buffer, size_t sizeInBytes, const char *varname);	Standard validations
Math	int rand(void);	int rand_s (void)	Crypto-safe
Stream IO	char *fgets(char *string, int n, FILE *stream); wchar_t *fgetws(wchar_t *string, int n, FILE *stream);	char *fgets_s(char *string, size_t sizeInBytes, int n, FILE *stream); wchar_t *fgetws_s(wchar_t *string, size_t sizeInWords, int n, FILE *stream);	Standard validations
Stream IO	int fscanf(FILE *stream, const char *format [, argument]...); int fwscanf(FILE *stream, const wchar_t *format [, argument]...);	int fscanf_s(FILE *stream, const char *format [, argument]...); int fwscanf_s(FILE *stream, const wchar_t *format [, argument]...);	Require buffer lengths

Stream IO	char *gets(char *buffer);	errcode gets_s(char *buffer, size_t sizeInBytes);	Standard validations
Stream IO	int scanf(const char *format [, argument]...); int wscanf(const wchar_t *format [, argument]...);	int scanf_s(const char *format [, argument]...); int wscanf_s(const wchar_t *format [, argument]...);	Requires buffer sizes
Stream IO	void setbuf(FILE *stream, char *buffer);	Deprecate	Standard validations
Stream IO	int vsprintf(const char *format, va_list argptr); int vswprintf(const wchar_t *format, va_list argptr);	int vsprintf_s(const char *format, size_t count, va_list argptr); int vswprintf_s(const wchar_t *format, size_t count, va_list argptr);	Nul terminate Buffer size Parameter validate
String	void *memcpy(void *dest, const void *src, size_t count); wchar_t *wmemcpy(	errcode memcpy_s(void *dest, size_t sizeInBytes, const void *src, size_t count	Standard validations Don’t deprecate old

	wchar_t *dest, const wchar_t *src, size_t count);	); errcode wmemcpy_s(wchar_t *dest, size_t sizeInWords, const wchar_t *src, size_t count);	
String	void *memmove(void *dest, const void *src, size_t count); wchar_t *wmemmove(wchar_t *dest, const wchar_t *src, size_t count);	void *memmove(void *dest, size_t sizeInBytes, const void *src, size_t count); wchar_t *wmemmove(wchar_t *dest, size_t sizeInWords, const wchar_t *src, size_t count);	Standard validations
String	int sprintf(char *buffer, const char *format [, argument] ...);	Deprecate	Null terminate and validate
String	int sscanf(const char *buffer, const char *format [, argument] ...); int swscanf(	int sscanf_s(const char *buffer, const char *format [, argument] ...);	Require buffer sizes

	const wchar_t *buffer, const wchar_t *format [, argument] ...);	int swscanf_s(const wchar_t *buffer, const wchar_t *format [, argument] ...);	
String	char *strcat(char *strDestination, const char *strSource); wchar_t *wcscat(wchar_t *strDestination, const wchar_t *strSource);	Deprecate	Standard validations
String	char *strcpy(char *strDestination, const char *strSource); wchar_t *wcscpy(wchar_t *strDestination, const wchar_t *strSource); unsigned char *_mbscpy(unsigned char *strDestination, const unsigned char *strSource);	Deprecate	Standard validations
String	char *strerror(int errnum	errcode strerror_s(char *buffer, size_t	Standard validations

	);	sizeInBytes int errnum,);	
String		size_t strnlen(const char *string); size_t wcsnlen(const wchar_t *string);	Standard validations
String	char *strncat(char *strDest, const char *strSource, size_t count); wchar_t *wcsncat(wchar_t *strDest, const wchar_t *strSource, size_t count);	errcode strncat_s(char *strDest, size_t sizeInBytes, const char *strSource, size_t count); errcode wcsncat_s(wchar_t *strDest, size_t sizeInWords, const wchar_t *strSource, size_t count);	Ensure null termination Validate parameters
String	char *strncpy(char *strDest, const char *strSource, size_t count); wchar_t *wcsncpy(wchar_t *strDest, const wchar_t *strSource, size_t count	char *strncpy_s(char *strDest, size_t sizeInBytes, const char *strSource, size_t count); wchar_t *wcsncpy_s(wchar_t *strDest, size_t	Standard validations

	);	sizeInWords, const wchar_t *strSource, size_t count);	
String	char *strtok(char *strToken, const char *strDelimit); wchar_t *wcstok(wchar_t *strToken, const wchar_t *strDelimit); unsigned char *_mbstok(unsigned char*strToken, const unsigned char *strDelimit);	We should do strtok_r to avoid static reentrancy issues	Standard validations
String	size_t wcstombs(char *mbstr, const wchar_t *wcstr, size_t count);	size_t wcstombs_s(char *mbstr, size_t sizeInBytes, const wchar_t *wcstr, size_t count);	Standard validations
String	size_t mbstowcs(wchar_t *wcstr, const char *mbstr, size_t count);	errcode mbstowcs_s(size_t *pConvertedMBChars, wchar_t *wcstr, size_t sizeInWords, const char *mbstr, size_t count);	Takes a buffer size and won't write past

Time	`char *ctime(`
	`const time_t *timer`
	`);`
		`errcode ctime_s(`
		`char *buffer,`
		`size_t sizeInBytes,`
		`const time_t *timer`
		`);`
			Output buffer, no static buffers