Safe C String Library v1.0.2 (November 19, 2003)

1. Introduction

The goal of the SafeStr library is to provide a rich string-handling library for C that has safe semantics yet interoperates with legacy library code in a straightforward manner. Additionally, porting code that uses standard C string handling should be straightforward. The library should work on all modern Unix-like platforms, as well as any 32-bit Microsoft Windows OS.

The overt security goals of the library are as follows:

1) Buffer overflows should not be possible when using the API.

2) Format string problems should be impossible when using the API.

3) The API should be capable of tracking whether strings are "trusted", a la Perl's taint mode.

The API is meant to provide rich functionality and be easy to use, all the while improving security.

To achieve interoperability with legacy code where you do not have the source or are unwilling to change it, the `safestr_t` type we define is completely compatible with type `char *`. That is, you can cast `safestr_t` structures to `char *`, and then they will act as a `char *` in every way. Without the explicit cast, the compiler will generate either a warning or an error (depending on how strict your compiler is).

The way `safestr_t` works under the hood is by keeping accounting information such as the actual and allocated length in memory directly preceding the memory referenced by the pointer. Therefore, if you cast to a `char *`, then pass the result to a function that modifies the `char *`, you are likely to thwart `safestr_t` length accounting.

The rule of thumb is that it's okay to cast to `char *` for read-only purposes. You should never cast for writing, because you risk buffer overflows. If you do cast for writing, do not subsequently use the data structure as a `safestr_t`, until it's time to free that structure (you cannot call `free()` on the pointer directly).

For example, the following code is safe, albeit rather pointless:

```
char      c_buffer[13];
safestr_t str;

str = safestr_create("hello, world", 0);
strcpy(c_buffer, (char *)str);
```

However, note that there's still no bounds checking on the `strcpy()` operation. When possible, you should use SafeStr's `safestr_copy()` or `safestr_ncopy()` routines, which are analogous to `strcpy()` and `strncpy()`.

On the other hand, the following code is *not* safe, particularly because it modifies the contents of a `safestr_t` structure outside of the Safe C String Library's API:

```
safestr_t str;

str = safestr_alloc(12, 0);
strcpy((char *)str, "hello, world");
```

Note that, if you write functions that take a `safestr_t` and possibly perform multiple operations on a `safestr_t`, you have to do some extra magic to handle temporary strings. See below for more information.

2. Temporary Strings

Sometimes it's convenient to use string constants as an argument to a function. For exactly that purpose the SafeStr API provides a notion of temporary strings, which are strings designed to survive a single invocation to a SafeStr library call so that you don't need to worry about memory management.

For example, you can do the following:

```
safestr_t s = safestr_replace(SAFESTR_TEMP("This is a test"),
                        SAFESTR_TEMP(" "), SAFESTR_TEMP("."));
```

`SAFESTR_TEMP()` is a macro that calls `safestr_create()` with the `SAFESTR_TEMPORARY` flag turned on. Strings created in this manner are also immutable (see below).

You may wish to label temporary strings as "trusted" (see the discussion on trust management below). For those cases, there's a `SAFESTR_TEMP_TRUSTED()` macro that otherwise works identically to `SAFESTR_TEMP()`.

A temporary `safestr_t` string is deallocated automatically after a call to any Safe C String Library API function with the exception of `safestr_reference()`, which will increment the `safestr_t` structure's reference count. If a temporary string's reference count is incremented, it will then survive any number of API calls until its reference count is sufficiently decremented that it will be destroyed. The API functions `safestr_release()` and `safestr_free()` may be used interchangeably to decrement a `safestr_t` structure's reference count.

For example, if you are writing a function that accepts a `safestr_t` structure as an argument (may be temporary, but also may not be) and will perform multiple operations on the string, you should increment the `safestr_t` structure's reference count before operating on the string, and decrement it again when you are finished. This will ensure that the string is not prematurely destroyed if a temporary string is passed in to your function.

```
void some_function(safestr_t *base, safestr_t extra)
{
    safestr_reference(extra);
    if (safestr_length(*base) + safestr_length(extra) < 17)
        safestr_append(base, extra);
    safestr_release(extra);
}
```

In this example, if the calls to `safestr_reference()` and `safestr_release()` were to be omitted, and extra was a temporary string, the call to `safestr_length()` would cause the string to be destroyed. As a result, the `safestr_append()` call would then be operating on an invalid `safestr_t` if the combined length of base and extra was less than 17.

3. Immutable Strings

This library can also prevent you from accidentally using the SafeStr API to overwrite data that shouldn't be changed. That is, the library has a notion of immutable strings. You can set the immutable flag when initializing a string from a `char *`, or you may call the following at any time:

```
void safestr_makereadonly(safestr_t);
```

Note that you can still directly modify the memory. The library can only prevent writes initiated through SafeStr functions. If you'd like to take an immutable string, and make it writable, use the following:

```
void safestr_makewritable(safestr_t);
```

You can check to see if a string is immutable with the following function:

```
int safestr_isreadonly(safestr_t);
```

4. Trust Management

The Safe C String API can help facilitate tracking what data has been checked for potentially malicious input. Strings may be "trusted" or "untrusted". When modifying a string, the trusted property of that string will be set to "untrusted" if any of the operands are untrusted. When creating a new string from operations on other strings, the new string will only be marked as trusted if all the strings that might influence its value are also trusted.

If you circumvent the Safe C String API, the trust property will not properly propagate.

The Safe C String API does not currently provide any routines that actually check the trusted flag. We expect that this will change in future versions. However, you may explicitly check the flag yourself.

For example:

```
#include <stdlib.h>
#include <stdio.h>
#include "strsafe.h"

int safer_system(safestr_t cmd)
{
    if (!safestr_istrusted(cmd))
    {
        fprintf(stderr,
                "Untrusted data flowed into safer_system!\n");
        abort();
    }
    return system((char *)cmd);
}
```

Note that you can explicitly set or unset the trusted flag with the calls `safestr_trust()` and `safestr_untrust()` (described below).

5. Resizable Strings

If you use the recommended API for initializing `safestr_t` structures, you'll get strings that will reallocate themselves if an operation would require that string to grow in size. As a consequence, any function that might cause a `safestr_t` to expand takes a pointer to a `safestr_t`, because, as implemented, reallocating will always move the memory.

6. Error Handling

Error handling is done using XXL (*http://www.zork.org/xxl/*), which is a library that provides both exceptions and asset management for C and C++. SafeStr uses both features provided by XXL extensively. Where previous releases of SafeStr would output a message to `stderr` and call `abort()`, exceptions are now thrown instead. It is expected that the caller will handle the exceptions appropriately (XXL's default action if there is no exception handler when an exception is thrown is to output a message to `stderr` and call `abort()`).

Under normal circumstances, SafeStr should rarely, if ever, throw any exceptions. If an exception is thrown, it is generally because the programmer has made some error in using the API. The API reference that is provided in this document lists the exceptions that may be thrown by each function.

7. Notes

- Currently, this library does not handle multi-byte characters natively.
- While strings are always stored with a trailing `NUL`, `NUL` characters within the string *do not* delimit the end of a string. That is, you can put arbitrary binary data into a `safestr_t` structure. However, if you pass such strings to a traditional C API, they will treat the string as shorter than

its actual length. ***This can lead to security problems.*** Therefore, we recommend you avoid putting `NUL` characters in `safestr_t` structures, unless you are exceedingly careful.

8. The API

The remaining portion of this document describes the Safe String Library API in its entirety. For each function, its signature, a brief description of what it does, the meaning of each argument, the exceptions it may throw, and a description of the function is provided.

8.1 safestr_alloc()

Name

`safestr_alloc` - Allocates a new `safestr_t` structure.

Synopsis

```
safestr_t safestr_alloc(u_int32_t length, u_int32_t flags);
```

Parameters

`u_int32_t length` This is the size of the string to allocate in bytes, and should not include a `NUL` byte as with C strings.

`u_int32_t flags` This is a bit mask of flags that control the behavior of the string being allocated. Any of the following flags are valid:

```
SAFESTR_TEMPORARY
SAFESTR_IMMUTABLE
SAFESTR_TRUSTED
SAFESTR_ASSET_PERMANENT
SAFESTR_ASSET_TEMPORARY
SAFESTR_ASSET_PROMOTE
SAFESTR_ASSET_DEMOTE
```

Exceptions

```
SAFESTR_ERROR_OUT_OF_MEMORY
SAFESTR_ERROR_PRNG_FAILURE
SAFESTR_ERROR_STRING_TOO_LONG
```

Description

A `safestr_t` structure is always `NUL` terminated as C strings are; however, all lengths as specified to or received from the Safe C String Library API never include the `NUL` terminating byte. The API automatically accounts for this special byte internally. That is, when you pass in the length parameter, you're specifying how much space you'd initially like to be available for storing the actual string itself.

If a string is designated as temporary, the returned string will survive *exactly* one call to any Safe C String Library API function. The single exception to this rule is `safestr_reference()`, which will increment the string's reference count, thus allowing it to then survive any number of API calls until its reference count is decremented by a call to either `safestr_free()` or `safestr_release()`.

If a string is designated as immutable, the API in any way cannot modify the string. This would include appending to, inserting into, and so on.

The `safestr_t` structure that is returned from `safestr_alloc()` will have an initial length of zero, but space is sufficient space is allocated to hold the requested length. If you know how much

space you will need for a string, it is best to allocate the space up front to reduce the need for reallocations later.

8.2 safestr_charat()

Name

`safestr_charat` - Retrieves the character at a specific position within a string.

Synopsis

```
char safestr_charat(safestr_t s, u_int32_t pos);
```

Parameters

`safestr_t s`	This is the string from which the character is to be retrieved.
`u_int32_t pos`	This is the zero-based position from which the character is to be retrieved.

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_INDEX_OUT_OF_RANGE
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function will return the character at the specified position in a `safestr_t` structure. Bounds checking is performed on the specified index position to ensure that it is valid. If it is not valid, an XXL exception will be thrown; otherwise, the character at the specified position will be returned.

8.3 safestr_clone()

Name

`safestr_clone` - Creates a new `safestr_t` structure that has the same contents as the specified string.

Synopsis

```
safestr_t safestr_clone(safestr_t s, u_int32_t flags);
```

Parameters

`safestr_t s`	This is the string that is to be cloned.

`u_int32_t flags`	This is a bit mask of flags that control the behavior of the string being created. Flags are not inherited on cloning, with the exception of the `SAFESTR_TRUSTED` flag. Any of the following flags are valid:
	`SAFESTR_TEMPORARY`
	`SAFESTR_IMMUTABLE`
	`SAFESTR_TRUSTED`
	`SAFESTR_ASSET_PERMANENT`
	`SAFESTR_ASSET_TEMPORARY`
	`SAFESTR_ASSET_PROMOTE`
	`SAFESTR_ASSET_DEMOTE`

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_OUT_OF_MEMORY
SAFESTR_ERROR_PRNG_FAILURE
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

If a string is designated as temporary, the returned string will survive *exactly* one call to any Safe C String Library API function. The single exception to this rule is `safestr_reference()`, which will increment the string's reference count, thus allowing it to then survive any number of API calls until its reference count is decremented by a call to either `safestr_free()` or `safestr_release()`.

If a string is designated as immutable, the API in any way cannot modify the string. This would include appending to, inserting into, and so on.

The `safestr_t` structure that is returned from `safestr_clone()` will initially have the same contents as the cloned string, though the flags may differ.

8.4 safestr_compare()

Name

`safestr_compare` - Lexically compares the contents of two `safestr_t` structures.

Synopsis

```
int safestr_compare(safestr_t s1, safestr_t s2, u_int32_t flags,
                    ...);
```

Parameters

`safestr_t s1`	This is the first `safestr_t` structure for comparison.
`safestr_t s2`	This is the second `safestr_t` structure for comparison.

`u_int32_t flags`	This is a bit mask of flags that control the behavior of the comparison. Any of the following flags are valid:
	`SAFESTR_COMPARE_NOCASE`
	`SAFESTR_COMPARE_LIMIT`

Convenience Macros

1. `safestr_ncompare(safestr_t s1, safestr_t s2, u_int32_t nbytes)`

   Expands to:

   `safestr_compare(s1, s2, SAFESTR_COMPARE_LIMIT, nbytes)`

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function is the rough equivalent of the `strcmp()`, `strncmp()`, `strcasecmp()`, and `strncasecmp()` functions for C strings. Strings are compared lexically without any consideration for collation.

If `SAFESTR_COMPARE_NOCASE` is specified as a flag, the comparison will be performed case insensitively. The default behavior is to compare the two strings case sensitively.

If `SAFESTR_COMPARE_LIMIT` is specified as a flag, an additional parameter is required that specifies the number of characters to consider in the comparison.

The return will be negative if `s1` is compares lexically less than `s2`, positive if `s1` compares lexically greater than `s2`, or zero if `s1` and `s2` are equal.

8.5 safestr_concatenate()

Name

`safestr_concatenate` - Combines two `safestr_t` structures.

Synopsis

```
void safestr_concatenate(safestr_t *dst, safestr_t src,
                         u_int32_t flags, ...);
```

Parameters

`safestr_t *dst`	This is the first string to be combined. The second string will be appended to this string.
`safestr_t src`	This is the second string to be combined. This string will be appended to the first string.
`u_int32_t flags`	This is a bit mask of flags that control the behavior of the combination. Any of the following flags are valid: `SAFESTR_COPY_LIMIT`

Convenience Macros

1. `safestr_append(safestr_t *dst, safestr_t src)`

   Expands to:

   ```
   safestr_concatenate(dst, src, 0)
   ```

2. `safestr_nappend(safestr_t *dst, safestr_t src, u_int32_t nbytes)`

   Expands to:

   ```
   safestr_concatenate(dst, src, SAFESTR_COPY_LIMIT, nbytes)
   ```

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_IMMUTABLE_STRING
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_OUT_OF_MEMORY
SAFESTR_ERROR_STRING_TOO_LONG
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function will combine the two specified strings together, and place the result in the first string. The destination `safestr_t` structure may need to be grown, thus its address may change, hence the requirement for a pointer to a `safestr_t` structure.

If `SAFESTR_COPY_LIMIT` is specified as a flag, an additional parameter is required that specifies the maximum number of characters that the result can be.

8.6 safestr_convert()

Name

`safestr_convert` - Converts a SafeStr string to upper, lower, or title case.

Synopsis

```
Void safestr_convert(safestr_t str, u_int32_t flags);
```

Parameters

`safestr_t str`	This is the SafeStr string to have its case converted.

`u_int32_t flags`	This is the set of flags to use for creating each `safestr_t` in the resulting array of SafeStr strings.
	`SAFESTR_CONVERT_UPPERCASE`
	`SAFESTR_CONVERT_LOWERCASE`
	`SAFESTR_CONVERT_TITLECASE`

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_IMMUTABLE_STRING
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function converts the contents of a SafeStr string to uppercase, lowercase, or titlecase. One of the three flags (`SAFESTR_CONVERT_UPPERCASE`, `SAFESTR_CONVERT_LOWERCASE`, or `SAFESTR_CONVERT_TITLECASE`) must be specified. They are mutually exclusive, and if more than one is specified (combined via bitwise OR), uppercase will take precedence, followed by lowercase.

The uppercase and lowercase conversions will convert all alphabetical characters to the specified case. The titlecase conversion will capitalize the first letter of each word, and convert other letters to lowercase. A special case for “Mc” will capitalize the letter following the “c”. Any non-alphabetic character will behave as a word separator.

8.7 safestr_convertarray()

Name

`safestr_convertarray` - Converts an array of C strings into an array of SafeStr strings.

Synopsis

```
safestr_t *safestr_convertarray(char **arr, u_int32_t flags);
```

Parameters

`char **arr` This is the array of C strings to be converted into SafeStr strings.

`u_int32_t flags` This is the set of flags to use for creating each `safestr_t` in the resulting array of SafeStr strings.

Exceptions

```
SAFESTR_ERROR_OUT_OF_MEMORY
SAFESTR_ERROR_PRNG_FAILURE
SAFESTR_ERROR_STRING_TOO_LONG
```

Description

This function converts an array of C strings into an array of SafeStr strings using `safestr_create()`. For each element in the original array, `safestr_create()` is called with the specified flags, and the resulting `safestr_t` structure is placed in the new array at the corresponding index position. The original array must be terminated with a `NULL` entry, and the resulting array will also be terminated with a `NULL` entry.

This function is particularly useful for converting `argv` or `envp` as received by `main()` into an array of `safestr_t` structures. We recommend converting these using only the `SAFESTR_IMMUTABLE` flag if any flags are specified at all. For example:

```
int main(int argc, char *argv[])
{
    safestr_t *safe_argv;

    safe_argv = safestr_convertarray(argv, SAFESTR_IMMUTABLE);
    /* ... */
}
```

Never set `SAFESTR_TRUSTED` when directly converting `argv`, `envp`, or any other external input!

8.8 safestr_create()

Name

`safestr_create` - Creates a `safestr_t` structure from a C string.

Synopsis

```
safestr_t safestr_create(char *s, u_int32_t flags);
```

Parameters

`char *s` The C string with which the `safestr_t` structure is to be populated.

`u_int32_t flags` This is a bit mask of flags that control the behavior of the string being created. Any of the following flags are valid:

```
SAFESTR_TEMPORARY
SAFESTR_IMMUTABLE
SAFESTR_TRUSTED
SAFESTR_ASSET_PERMANENT
SAFESTR_ASSET_TEMPORARY
SAFESTR_ASSET_PROMOTE
SAFESTR_ASSET_DEMOTE
```

Convenience Macros

1. `SAFESTR_TEMP(char *s)`

   Expands to:

   ```
   safestr_create(s, SAFESTR_TEMPORARY | SAFESTR_IMMUTABLE)
   ```

2. `SAFESTR_TEMP_TRUSTED(char *s)`

   Expands to:

   ```
   safestr_create(s, SAFESTR_TEMPORARY | SAFESTR_IMMUTABLE |
                  SAFESTR_TRUSTED)
   ```

Exceptions

```
SAFESTR_ERROR_OUT_OF_MEMORY
SAFESTR_ERROR_PRNG_FAILURE
SAFESTR_ERROR_STRING_TOO_LONG
```

Description

If a string is designated as temporary, the returned string will survive *exactly* one call to any Safe C String Library API function. The single exception to this rule is `safestr_reference()`, which will increment the string's reference count, thus allowing it to then survive any number of API calls until its reference count is decremented by a call to either `safestr_free()` or `safestr_release()`.

If a string is designated as immutable, the API in any way cannot modify the string. This would include appending to, inserting into, and so on.

The `safestr_t` structure that is returned from `safestr_create()` will be initialized with the contents of the C string.

8.9 safestr_delete()

Name

`safestr_delete` - Deletes characters from a `safestr_t` structure.

Synopsis

```
void safestr_delete(safestr_t *s, u_int32_t pos, u_int32_t count);
```

Parameters

`safestr_t *s`	This is the string from which characters are to be deleted.
`u_int32_t pos`	This is the zero-based position from which characters are to be deleted.
`u_int32_t count`	This is the number of character to be deleted.

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_IMMUTABLE_STRING
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function will delete the specified number of characters beginning at the specified index from the specified `safestr_t` structure. If more characters to be deleted are specified than exist in the string from the starting position, the string will be truncated.

8.10 safestr_duplicate()

Name

`safestr_duplicate` - Copies the contents of one string into another.

Synopsis

```
void safestr_duplicate(string *dst, safestr_t src, u_int32_t flags,
                       ...);
```

Parameters

`safestr_t *dst`	This is the string that will receive the contents of the string to be copied.
`safestr_t src`	This is the string that will be copied into the destination string.
`u_int32_t flags`	This is a bit mask of flags that control the behavior of the copy operation. Any of the following flags are valid: `SAFESTR_COPY_LIMIT`

Convenience Macros

1. `safestr_copy(safestr_t *dst, safestr_t src)`

   Expands to:

   ```
   safestr_duplicate(dst, src, 0)
   ```

2. `safestr_ncopy(safestr_t *dst, safestr_t src, u_int32_t nbytes)`

   Expands to:

   ```
   safestr_ncopy(dst, src, SAFESTR_COPY_LIMIT, nbytes)
   ```

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_OUT_OF_MEMORY
SAFESTR_ERROR_STRING_TOO_LONG
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function will copy the contents of one `safestr_t` structure into another. If the destination string is not large enough to hold the source string, it will be grown to the required size.

If `SAFESTR_COPY_LIMIT` is specified as a flag, an additional parameter is required to specify the number of characters from the source string to be copied into the destination string, which is equivalent to the maximum length of the resulting destination string.

8.11 safestr_equal()

Name

`safestr_equal` - Lexically compares two `safestr_t` structures for equality.

Synopsis

```
int safestr_equal(safestr_t s1, safestr_t s2, u_int32_t flags, ...);
```

Parameters

`safestr_t s1`	This is the first `safestr_t` structure for comparison.
`safestr_t s2`	This is the second `safestr_t` structure for comparison.

`u_int32_t flags`	This is a bit mask of flags that control the behavior of the comparison. Any of the following flags are valid:
	`SAFESTR_COMPARE_NOCASE`
	`SAFESTR_COMPARE_LIMIT`

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function is essentially a convenience version of `safestr_compare()`. It operates in the same manner, performing a lexical comparison of the two strings that are specified according to the rules specified in the flags. A non-zero value will be returned if the two strings are equal; otherwise, a zero return value indicates that they differ in some way.

If `SAFESTR_COMPARE_NOCASE` is specified as a flag, the comparison will be performed case insensitively. The default behavior is to compare the two strings case sensitively.

If `SAFESTR_COMPARE_LIMIT` is specified as a flag, an additional parameter is required that specifies the maximum number of characters to consider in the comparison.

8.12 safestr_fprintf()

Name

`safestr_fprintf` - Writes a formatted string to a stream.

Synopsis

```
u_int32_t safestr_fprintf(FILE *stream, safestr_t fmt, ...);
```

Parameters

`FILE *stream`	This is the stream to which the formatted output will be written.
`safestr_t fmt`	This is the format string to use.

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_ILLEGAL_PERCENT_N
```

```
SAFESTR_ERROR_INVALID_FORMAT_ARG
SAFESTR_ERROR_INVALID_FORMAT_STRING
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_TOO_MANY_FORMAT_ARGS
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function is essentially the same as the standard C `fprintf()` function. The formatting string and additional parameters are the same as with `fprintf()`, except that "%n" is not allowed. Additionally, the "%s" arguments must all map to valid `safestr_t` structures, and the result will be written to the specified stream. This version also does sanity checking of format strings that may not happen in some underlying implementations. When an inconsistency is found, an exception is thrown.

The return value from the function will be the number of characters that were written to the destination stream.

8.13 safestr_free()

Name

`safestr_free` - Decrements the reference count of a `safestr_t` structure.

Synopsis

```
void safestr_free(safestr_t s);
```

Parameters

`safestr_t s` This is the string that will have its reference count decremented.

Convenience Macros

`0. safestr_release(safestr_t s)`

Expands to:

```
safestr_free(s)
```

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function will decrement the reference count of the specified string by one. If the string's reference count reaches zero, the memory allocated for the string will be freed.

8.14 safestr_getpassword()

Name

`safestr_getpassword` – Interactively obtain a password without echo from the terminal.

Synopsis

```
safestr_t safestr_getpassword(FILE *term, safestr_t prompt);
```

Parameters

`FILE *term`	This is the stream from which the password will be read. It may be specified as `NULL`, in which case `_PATH_TTY` (usually `/dev/tty` on Unix or `CONIN$` on Windows) will be used.
`safestr_t prompt`	This is a string containing a prompt to display before reading the password.

Exceptions

```
errno
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_OUT_OF_MEMORY
SAFESTR_ERROR_STRING_TOO_LONG
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function will interactively read a password from the terminal. Before reading the password, the specified prompt will be displayed, and echo will be disabled. The password that was read will be returned in a new SafeStr string.

This function is essentially a wrapper around the Unix `getpass()` function.

8.15 safestr_insert()

Name

`safestr_insert` - Inserts one string into another.

Synopsis

```
void safestr_insert(safestr_t *dst, u_int32_t pos, safestr_t src);
```

Parameters

`safestr_t *dst`	This is the string that will have the other string inserted into it.
`u_int32_t pos`	This is the zero-based position in the destination string to insert the source string.
`safestr_t src`	This is the string to be inserted into the destination string.

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_IMMUTABLE_STRING
SAFESTR_ERROR_INDEX_OUT_OF_RANGE
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_OUT_OF_MEMORY
SAFESTR_ERROR_STRING_TOO_LONG
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function will insert one string into another at the specified zero-based position. The destination string is a pointer to the `safestr_t` structure to be inserted into because it may need to be resized, which may potentially cause its address to change.

8.16 safestr_join()

Name

`safestr_join` - Combines an array of strings into a single string.

Synopsis

```
safestr_t safestr_join(safestr_t *s, safestr_t joiner);
```

Parameters

`safestr_t *s`	This is the array of strings to be joined together.
`safestr_t joiner`	This is the string to combine into the resulting string between each element in the array of strings.

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_OUT_OF_MEMORY
SAFESTR_ERROR_PRNG_FAILURE
SAFESTR_ERROR_STRING_TOO_LONG
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function will combine an array of strings into a single string, separating each element from the array with another string. The array of strings must contain `safestr_t` structures that may or may not be temporary. The last element of the array should be a `NULL` pointer.

The return from this function will be a new mutable string that is trusted if and only if all strings involved in the combination are also trusted.

8.17 safestr_length()

Name

`safestr_length` - Returns the number of characters in a string.

Synopsis

```
u_int32_t safestr_length(safestr_t s);
```

Parameters

`safestr_t s`	The string from which the length is to be obtained.

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function will return the number of characters contained in the specified `safestr_t` structure. Note that this is not the same as the amount of space allocated to the `safestr_t` structure, which won't necessarily be the same.

8.18 safestr_memzero()

Name

`safestr_memzero` - Securely fill memory with zero bytes.

Synopsis

```
void safestr_memzero(volatile void *str, u_int32_t len);
```

Parameters

`volatile void *str`	This is a pointer to the memory to be filled with zero bytes.
`u_int32_t len`	This is the number of zero bytes with which the buffer shall be filled.

Description

This function will zero out memory in a secure manner, which essentially means to avoid compiler dead code elimination optimizations that can happen when using `memset()`.

8.19 safestr_printf()

Name

`safestr_printf` - Writes a formatted string to `stdout`.

Synopsis

```
u_int32_t safestr_printf(safestr_t fmt, ...);
```

Parameters

`safestr_t fmt`	This is the format string to use.

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_ILLEGAL_PERCENT_N
SAFESTR_ERROR_INVALID_FORMAT_ARG
SAFESTR_ERROR_INVALID_FORMAT_STRING
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_TOO_MANY_FORMAT_ARGS
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function is essentially the same as the standard C `printf()` function. The formatting string and additional parameters are the same as with `printf()`, except that "%n" is not allowed. Additionally, the "%s" arguments must all map to valid `safestr_t` structures, and the result will be written to `stdout`. This version also does sanity checking of format strings that may not happen in some underlying implementations. When an inconsistency is found, an exception is thrown.

The return value from the function will be the number of characters that were written to `stdout`.

8.20 safestr_readline()

Name

`safestr_readline` - Read a line from a stream into a SafeStr string.

Synopsis

```
safestr_t safestr_readline(FILE *stream);
```

Parameters

`FILE *stream`	The stream from which the line is to be read.

Exceptions

```
errno
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_OUT_OF_MEMORY
SAFESTR_ERROR_STRING_TOO_LONG
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function will read a line from a stream into a SafeStr string. Data will be read from the specified stream and stored in a newly allocated SafeStr string until end-of-file (EOF) is reached or a newline character is read, whichever comes first. If a newline is read, it will not be included in the returned string. If the line ends in a carriage return/line feed sequence, the carriage return will also be removed from the returned string.

This function is essentially a wrapper around the `fgets()` function.

8.21 safestr_reference()

Name

`safestr_reference` - Increments the reference count of a string.

Synopsis

```
safestr_t safestr_reference(safestr_t s);
```

Parameters

`safestr_t s`	The `safestr_t` structure to have its reference count incremented.

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function will increment the reference count of the specified string. The maximum value of a reference count is 4294967295. Incrementing a string's reference count any higher will result in a `SAFESTR_ERROR_TOO_MANY_REFERENCES` exception being thrown. This does not seem an

unreasonable limitation. Note that strings having a reference count greater than one are forced to be immutable by SafeStr.

As a convenience, the return from this function will be string passed as its only argument.

8.22 safestr_replace()

Name

`safestr_replace` - Replaces all occurrences of a pattern in one string with another.

Synopsis

```
void safestr_replace(safestr_t *s, safestr_t old, safestr_t new);
```

Parameters

`safestr_t *s`	This is the string that will have occurrences of the specified pattern replaced.
`safestr_t old`	This is the pattern to be replaced.
`safestr_t new`	This is the string that will be used to replace all occurrences of the specified pattern.

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_IMMUTABLE_STRING
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_OUT_OF_MEMORY
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function will replace all occurrences of a specified pattern in a string with another string. The replacement is done in place.

This function is equivalent to splitting the original string using the old pattern, and rejoining the result with the new pattern. Using `safestr_replace()` is recommended, however, because it is faster.

8.23 safestr_search()

Name

`safestr_search` - Finds occurrences of one string inside of another.

Synopsis

```
u_int32_t safestr_search(safestr_t s, safestr_t sub,
                         u_int32_t flags, ...);
```

Parameters

`safestr_t s`	The string to be searched for occurrences of the sub-string.
`safestr_t sub`	The string to be searched for in the primary string.
`u_int32_t flags`	This is a bit mask of flags that control the behavior of the search operation. Any of the following flags are valid:

```
SAFESTR_FIND_REVERSE
SAFESTR_FIND_FROMCHAR
SAFESTR_FIND_FROMNTH
SAFESTR_FIND_NTH
SAFESTR_FIND_NOMATCHCASE
SAFESTR_FIND_CHARACTER
```

Convenience Macros

1. `safestr_find(safestr_t s, safestr_t sub)`

   Expands to:

   ```
   safestr_search(s, sub, 0)
   ```

2. `safestr_findchar(safestr_t s, unsigned char sub)`

   Expands to:

   ```
   safestr_search(s, NULL, SAFESTR_FIND_CHARACTER, sub)
   ```

3. `safestr_rfind(safestr_t, safestr_t sub)`

   Expands to:

   ```
   safestr_search(s, sub, SAFESTR_FIND_REVERSE)
   ```

4. `safestr_rfindchar(safestr_t, unsigned char sub)`

   Expands to:

   ```
   safestr_search(s, NULL, SAFESTR_FIND_REVERSE |
                           SAFESTR_FIND_CHARACTER, sub)
   ```

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_OUT_OF_MEMORY
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function will search a string for occurrences of a substring. The return will be the zero-based index of the occurrence, or it will be `SAFESTR_ERROR_NOTFOUND` if the substring was not found to be present in the primary string according to the behavioral controls.

If `SAFESTR_FIND_REVERSE` is specified, the search begins at the end of the string. It may be combined with any of the other flags. The default is to search forward if this flag is not specified.

If `SAFESTR_FROMCHAR` is specified, an additional parameter is required that specifies the zero-based starting position from the search. If `SAFESTR_FIND_REVERSE` is also specified, the position is from the end of the string. If not specified, the default is zero.

If `SAFESTR_FIND_FROMNTH` is specified, an additional parameter is required that specifies the number of prior occurrences less one to be ignored. If this flag is not specified, the default is zero. This flag acts as a delta for the next flag, which has much the same meaning as this one except that it is absolute.

If `SAFESTR_FIND_NTH` is specified, an additional parameter is required that specifies the number of prior occurrences less one to be ignored. If this flag is not specified, the default is one. This flag is an

absolute count that may be modified by SAFESTR_FIND_FROMNTH, which has much the same meaning as this flag except that it is a delta.

If SAFESTR_FIND_NOMATCHCASE is specified, the search is performed case insensitively. The default is to search case sensitively.

If SAFESTR_FIND_CHARACTER is specified, the sub argument is completely ignored (it should be specified as NULL), and the character to search for is obtained from the variable argument list in the appropriate position as described below.

If more than one flag is specified that requires an additional parameter, the parameters should be specified in the order that the flags are listed here in this document: SAFESTR_FIND_FROMCHAR first, followed by SAFESTR_FIND_FROMNTH second, SAFESTR_FIND_NTH third, and SAFESTR_FIND_CHARACTER last.

8.24 safestr_setcharat()

Name

safestr_setcharat - Sets the character at a specific position within a string.

Synopsis

```
void safestr_setcharat(safestr_t s, u_int32_t pos, char c,
                       int trust);
```

Parameters

safestr_t s	This is the string from which the character is to be retrieved.
u_int32_t pos	This is the zero-based position from which the character is to be retrieved.
char c	This is the character to set.
int trust	This is a boolean flag indicating whether the character being set is trusted or not.

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_IMMUTABLE_STRING
SAFESTR_ERROR_INDEX_OUT_OF_RANGE
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function will set the character at the specified position in a safestr_t structure. Bounds checking is performed on the specified index position to ensure that it is valid. If it is not valid, an XXL exception will be thrown; otherwise, the character at the specified position will be set to the specified character. The trust flag of the string will be updated if necessary to reflect the trust setting of the character being set.

Note that safestr_setcharat() will not ever try to grow a string. It is only intended to replace an existing character.

8.25 safestr_setmemfns()

Name

`safestr_setmemfns` - Sets the functions to use for memory management.

Synopsis

```
void safestr_setmemfns(safestr_malloc_t malloc_fn,
                       safestr_realloc_t realloc_fn,
                       safestr_free_t free_fn);
```

Parameters

`safestr_malloc_t malloc_fn`	This is a pointer to the function to call to allocate memory. The default is `safestr_default_malloc()`.
`safestr_realloc_t realloc_fn`	This is a pointer to the function to call to reallocate memory. The default is `safestr_default_realloc()`.
`safestr_free_t free_fn`	This is a pointer to the function to call to free memory. The default is `safestr_default_ree()`.

Description

By default, the Safe C String Library will use `safestr_default_malloc()`, `safestr_default_realloc()`, and `safestr_default_free()` to perform memory management, but this function will allow you to roll your own functions to replace them. Specifying `NULL` for any of the three function pointers will cause the default for that function to be used. The three default functions are simply wrappers around `malloc()`, `realloc()`, and `free()`.

The function pointers for memory allocation, reallocation, and freeing are the same as for `malloc()`, `realloc()`, and `free()` with the addition of two arguments to pass the filename and line number from where the calls to them are made. The default functions throw this information away, but alternative memory management APIs can use this information to tracking memory leaks, usage, etc. The signature for each of these functions is as follows:

```
void *safestr_default_malloc(size_t nbytes, const char *filename,
                             unsigned int lineno);

void *safestr_default_realloc(void *ptr, size_t nbytes,
                              const char *filename,
                              unsigned int lineno);

void safestr_default_free(void *ptr, const char *filename,
                          unsigned int lineno);
```

8.26 safestr_slice()

Name

`safestr_slice` - Creates a new string that is a substring of a string.

Synopsis

```
safestr_t safestr_slice(safestr_t s, u_int32_t pos, u_int32_t end);
```

Parameters

`safestr_t s`	This is the string that a substring is to be extracted from.
`u_int32_t pos`	This is the zero-based starting position to extract characters from.
`u_int32_t end`	This is the one-based ending position to extract characters from.

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_OUT_OF_MEMORY
SAFESTR_ERROR_PRNG_FAILURE
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function will create a new mutable string that will be trusted only if the original string is trusted. The new string will contain the specified portion of the original string. The starting position to extract from is zero-based, and the ending position is essentially one-based.

For example, to extract from the 17th character to the end of the source string, the following call would be used:

```
slice = safestr_slice(source, 16, safestr_length(source));
```

8.27 safestr_split()

Name

`safestr_split` - Splits a string into an array of strings.

Synopsis

```
safestr_t *safestr_split(safestr_t s, safestr_t sub);
```

Parameters

`safestr_t s`	The string to be split into an array of strings.
`safestr_t sub`	The string to be used as a delimiter for breaking the primary string.

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_OUT_OF_MEMORY
SAFESTR_ERROR_PRNG_FAILURE
SAFESTR_ERROR_STRING_TOO_LONG
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function will create an array of `safestr_t` structures from a single string. For each occurrence of the delimiting string, a new `safestr_t` non-resizable, permanent `safestr_t` structure will be created that contains the characters between the previous occurrence of the delimiting string and the current one.

The array that is returned will have one additional element that is the `NULL` pointer to mark the end of the array. The delimiting string will be omitted from the returned array. If the delimiting string is the

empty string (it has a length of zero), an array containing `safestr_t` structures for each character in the source string will be returned.

The resulting array and delimiter can be passed to `safestr_join()` to obtain the original string:

```
safestr_t *array, delimiter, rebuilt, source;

array = safestr_split(source, delimiter);
rebuilt = safestr_join(array, delimiter);
safestr_equal(source, rebuilt) ==> 1
```

Examples

1. `safestr_split(SAFESTR_TEMP("abc"), SAFESTR_TEMP(""))`

   Yields:

   ```
   "a"
   "b"
   "c"
   NULL
   ```

2. `safestr_split(SAFESTR_TEMP("/usr/local/etc"), SAFESTR_TEMP("/"))`

   Yields:

   ```
   ""
   "usr"
   "local"
   "etc"
   NULL
   ```

8.28 safestr_sprintf()

Name

`safestr_sprintf` - Writes a formatted string into another string.

Synopsis

```
u_int32_t safestr_sprintf(safestr_t *s, safestr_t fmt, ...);
```

Parameters

`safestr_t *s`	This is the string into which the formatted output will be written.
`safestr_t fmt`	This is the format string to use.

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_ILLEGAL_PERCENT_N
SAFESTR_ERROR_IMMUTABLE_STRING
SAFESTR_ERROR_INVALID_FORMAT_ARG
SAFESTR_ERROR_INVALID_FORMAT_STRING
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_TOO_MANY_FORMAT_ARGS
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function is essentially the same as the standard C `sprintf()` function. The formatting string and additional parameters are the same as with `sprintf()`, except that "%n" is not allowed. Additionally, the "%s" arguments must all map to valid `safestr_t` structures, the result will be written to a `safestr_t` structure, and the string will be resized as needed to hold the formatted string. This version also does sanity checking of format strings that may not happen in some underlying implementations. When an inconsistency is found, an exception is thrown.

The return value from the function will be the number of characters that were written to the destination string.

8.29 safestr_strdup()

Name

`safestr_strdup` - Duplicates a C string.

Synopsis

```
char *safestr_strdup(char *s);
```

Parameters

`char *s`	The C string to be duplicated.

Exceptions

`SAFESTR_ERROR_OUT_OF_MEMORY`

Description

This function performs the same operation as the standard C `strdup()` function, except that it uses the memory allocation routines specified using `safestr_setmemfns()`. The function will always return a pointer to a new copy of the C string. If an error occurs while trying to create the copy, an appropriate exception will be thrown, and the function will not return normally.

8.30 safestr_todouble()

Name

`safestr_todouble` - Convert a string to a `double` floating point number.

Synopsis

```
double safestr_todouble(safestr_t str);
```

Parameters

`safestr_t str`	This is the string to be converted to a `double` floating point number.

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_INVALID_FORMAT
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_OUT_OF_MEMORY
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function will convert a string representation of a floating point number to a `double`. It is currently implemented as simply a wrapper around the `strtod()` function.

8.31 safestr_toint32()

Name

`safestr_toint32` - Convert a string to a signed 32-bit integer.

Synopsis

```
int32_t safestr_toint64(safestr_t str, u_int32_t base);
```

Parameters

`safestr_t str`	This is the string to be converted to an integer.
`u_int32_t base`	This is the base to use (usually 0, 8, 10, or 16). If 0 is specified, an attempt will be made to guess the proper base.

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_INVALID_FORMAT
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_OUT_OF_MEMORY
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function will convert a string representation of a number to a signed 32-bit integer. If no base is specified, the function will attempt to determine the base that the number is encoded in by looking for “0b” or “0B” for base 2, “0x” or “0X” for base 16, or a leading “0” for base 8. If none of these are matched, base 10 is assumed. If a base is specified, valid bases are 2 through 36.

Leading whitespace (ASCII 127 or less than or equal to 32) is ignored. After that, once parsing has begun any invalid character will cause a `SAFESTR_ERROR_INVALID_FORMAT` exception to be thrown. The only exception is a whitespace character, which will cause parsing to terminate as if the end of the string was reached.

This function is essentially a wrapper around the `strtol()` function.

8.32 safestr_toint64()

Name

`safestr_toint64` - Convert a string to a signed 64-bit integer.

Synopsis

```
int64_t safestr_toint64(safestr_t str, u_int32_t base);
```

Parameters

`safestr_t str`	This is the string to be converted to an integer.
`u_int32_t base`	This is the base to use (usually 0, 8, 10, or 16). If 0 is specified, an attempt will be made to guess the proper base.

Exceptions

SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_INVALID_FORMAT
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_OUT_OF_MEMORY
SAFESTR_ERROR_TOO_MANY_REFERENCES

Description

This function will convert a string representation of a number to a signed 64-bit integer. If no base is specified, the function will attempt to determine the base that the number is encoded in by looking for "0b" or "0B" for base 2, "0x" or "0X" for base 16, or a leading "0" for base 8. If none of these are matched, base 10 is assumed. If a base is specified, valid bases are 2 through 36.

Leading whitespace (ASCII 127 or less than or equal to 32) is ignored. After that, once parsing has begun any invalid character will cause a SAFESTR_ERROR_INVALID_FORMAT exception to be thrown. The only exception is a whitespace character, which will cause parsing to terminate as if the end of the string was reached.

This function is essentially a wrapper around the strtoll() function.

8.33 safestr_touint32()

Name

safestr_touint32 - Convert a string to an unsigned 32-bit integer.

Synopsis

```
u_int32_t safestr_touint64(safestr_t str, u_int32_t base);
```

Parameters

safestr_t str	This is the string to be converted to an integer.
u_int32_t base	This is the base to use (usually 0, 8, 10, or 16). If 0 is specified, an attempt will be made to guess the proper base.

Exceptions

SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_INVALID_FORMAT
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_OUT_OF_MEMORY
SAFESTR_ERROR_TOO_MANY_REFERENCES

Description

This function will convert a string representation of a number to an unsigned 32-bit integer. If no base is specified, the function will attempt to determine the base that the number is encoded in by looking for "0b" or "0B" for base 2, "0x" or "0X" for base 16, or a leading "0" for base 8. If none of these are matched, base 10 is assumed. If a base is specified, valid bases are 2 through 36.

Leading whitespace (ASCII 127 or less than or equal to 32) is ignored. After that, once parsing has begun any invalid character will cause a SAFESTR_ERROR_INVALID_FORMAT exception to be thrown. The only exception is a whitespace character, which will cause parsing to terminate as if the end of the string was reached.

This function is essentially a wrapper around the strtol() function.

8.34 safestr_touint64()

Name

`safestr_touint64` - Convert a string to an unsigned 64-bit integer.

Synopsis

```
u_int64_t safestr_touint64(safestr_t str, u_int32_t base);
```

Parameters

`safestr_t str`	This is the string to be converted to an integer.
`u_int32_t base`	This is the base to use (usually 0, 8, 10, or 16). If 0 is specified, an attempt will be made to guess the proper base.

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_INVALID_FORMAT
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_OUT_OF_MEMORY
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function will convert a string representation of a number to an unsigned 64-bit integer. If no base is specified, the function will attempt to determine the base that the number is encoded in by looking for “0b” or “0B” for base 2, “0x” or “0X” for base 16, or a leading “0” for base 8. If none of these are matched, base 10 is assumed. If a base is specified, valid bases are 2 through 36.

Leading whitespace (ASCII 127 or less than or equal to 32) is ignored. After that, once parsing has begun any invalid character will cause a `SAFESTR_ERROR_INVALID_FORMAT` exception to be thrown. The only exception is a whitespace character, which will cause parsing to terminate as if the end of the string was reached.

This function is essentially a wrapper around the `strtoll()` function.

8.35 safestr_trim()

Name

`safestr_trim` - Trim whitespace from a string.

Synopsis

```
void safestr_trim(safestr_t str, u_int32_t flags);
```

Parameters

`safestr_t str`	This is the string to be trimmed.
`u_int32_t flags`	This is a set of flags that determine how the string is to be trimmed.

```
SAFESTR_TRIM_LEADING
SAFESTR_TRIM_TRAILING
SAFESTR_TRIM_BOTH
```

Exceptions

SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_IMMUTABLE_STRING
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_OUT_OF_MEMORY
SAFESTR_ERROR_TOO_MANY_REFERENCES

Description

This function will trim whitespace from the specified string. If no flags are specified, SAFESTR_TRIM_BOTH is assumed, which is equivalent to combining both SAFESTR_TRIM_LEADING and SAFESTR_TRIM_TRAILING. Any character with an ASCII value of 127 or less than or equal to 32 is considered whitespace.

8.36 safestr_truncate()

Name

safestr_truncate - Truncates a string.

Synopsis

```
void safestr_truncate(safestr_t *s, u_int32_t length);
```

Parameters

safestr_t *s	This is the string to be truncated.
u_int32_t length	This is the length to which the string is to be truncated.

Exceptions

SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_IMMUTABLE_STRING
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_OUT_OF_MEMORY
SAFESTR_ERROR_STRING_TOO_LONG
SAFESTR_ERROR_TOO_MANY_REFERENCES

Description

This function will truncate the specified string to the specified length. If the string is not as long as the specified length, it will be grown to the necessary length and filled with space characters.

8.37 safestr_vfprintf()

Name

safestr_vprintf - Writes a formatted string to a stream.

Synopsis

```
u_int32_t safestr_vfprintf(FILE *stream, safestr_t fmt, va_list ap);
```

Parameters

FILE *stream	This is the stream to which the formatted output will be written.
safestr_t fmt	This is the format string to use.

`va_list ap` This is the pointer to the variable arguments for substitution into the formatted string.

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_ILLEGAL_PERCENT_N
SAFESTR_ERROR_INVALID_FORMAT_ARG
SAFESTR_ERROR_INVALID_FORMAT_STRING
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_TOO_MANY_FORMAT_ARGS
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function is essentially the same as the standard C `fprintf()` function. The formatting string and additional parameters are the same as with `fprintf()`, except that "%n" is not allowed. Additionally, the "%s" arguments must all map to valid `safestr_t` structures, and the result will be written to the specified stream. This version also does sanity checking of format strings that may not happen in some underlying implementations. When an inconsistency is found, an exception is thrown.

The return value from the function will be the number of characters that were written to the destination stream.

8.38 safestr_vprintf()

Name

`safestr_vprintf` - Writes a formatted string to `stdout`.

Synopsis

```
u_int32_t safestr_vprintf(safestr_t fmt, va_list ap);
```

Parameters

`safestr_t fmt` This is the format string to use.

`va_list ap` This is the pointer to the variable arguments for substitution into the formatted string.

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_ILLEGAL_PERCENT_N
SAFESTR_ERROR_INVALID_FORMAT_ARG
SAFESTR_ERROR_INVALID_FORMAT_STRING
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_TOO_MANY_FORMAT_ARGS
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function is essentially the same as the standard C `printf()` function. The formatting string and additional parameters are the same as with `printf()`, except that "%n" is not allowed. Additionally, the "%s" arguments must all map to valid `safestr_t` structures, and the result will be written to `stdout`. This version also does sanity checking of format strings that may not happen in some underlying implementations. When an inconsistency is found, an exception is thrown.

The return value from the function will be the number of characters that were written to `stdout`.

8.39 safestr_vsprintf()

Name

`safestr_vsprintf` - Writes a formatted string into another string.

Synopsis

```
u_int32_t safestr_vsprintf(safestr_t *s, safestr_t fmt, va_list ap);
```

Parameters

`safestr_t *s`	This is the string into which the formatted output will be written.
`safestr_t fmt`	This is the format string to use.
`va_list ap`	This is the pointer to the variable arguments for substitution into the formatted string.

Exceptions

```
SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_ILLEGAL_PERCENT_N
SAFESTR_ERROR_IMMUTABLE_STRING
SAFESTR_ERROR_INVALID_FORMAT_ARG
SAFESTR_ERROR_INVALID_FORMAT_STRING
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_TOO_MANY_FORMAT_ARGS
SAFESTR_ERROR_TOO_MANY_REFERENCES
```

Description

This function is essentially the same as the standard C `vsprintf()` function. The formatting string and additional parameters are the same as with `vsprintf()`, except that "%n" is not allowed. Additionally, the "%s" arguments must all map to valid `safestr_t` structures, the result will be written to a `safestr_t` structure, and the string will be resized as needed to hold the formatted string. This version also does sanity checking of format strings that may not happen in some underlying implementations. When an inconsistency is found, an exception is thrown.

The return value from the function will be the number of characters that were written to the destination string.

8.40 safestr_trust(), safestr_untrust(), safestr_istrusted()

Name

`safestr_trust` - Set the `SAFESTR_TRUSTED` flag on a SafeStr string.
`safestr_untrust` - Clear the `SAFESTR_TRUSTED` flag on a SafeStr string.
`safestr_istrusted` - Test the `SAFESTR_TRUSTED` flag on a SafeStr string.

Synopsis

```
void safestr_trust(safestr_t s);
void safestr_untrust(safestr_t s);
int safestr_istrusted(safestr_t s);
```

Parameters

`safestr_t s`	The string for which the `SAFESTR_TRUSTED` flag is to be manipulated.

Exceptions

SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_TOO_MANY_REFERENCES

Description

These functions manipulate the SAFESTR_TRUSTED flag on a string. The function safestr_trust() will set the flag, and safestr_untrust() will clear it. Whether the SAFESTR_TRUSTED flag is set on a string can be determined by calling safestr_istrusted(), which will return zero if it is not set; otherwise, it will return non-zero.

8.41 safestr_makereadonly(), safestr_makewritable(), safestr_isreadonly()

Name

safestr_makereadonly - Set the SAFESTR_IMMUTABLE flag on a SafeStr string.
safestr_makewritable - Clear the SAFESTR_IMMUTABLE flag on a SafeStr string.
safestr_isreadonly - Test the SAFESTR_IMMUTABLE flag on a SafeStr string.

Synopsis

```
void safestr_makereadonly(safestr_t s);
void safestr_makewritable(safestr_t s);
int safestr_isreadonly(safestr_t s);
```

Parameters

safestr_t s	The string for which the SAFESTR_IMMUTABLE flag is to be manipulated.

Exceptions

SAFESTR_ERROR_BAD_ADDRESS
SAFESTR_ERROR_INVALID_PARAMETER
SAFESTR_ERROR_TOO_MANY_REFERENCES

Description

These functions manipulate the SAFESTR_IMMUTABLE flag on a string. The function safestr_makereadonly() will set the flag, and safestr_makewritable() will clear it. Whether the SAFESTR_IMMUTABLE flag is set on a string can be determined by calling safestr_isreadonly(), which will return zero if it is not set; otherwise, it will return non-zero.