

The CERT logo consists of several horizontal, slightly slanted bars of varying lengths, creating a stylized, layered effect. The word "CERT" is centered over these bars in a large, bold, black sans-serif font.

CERT

CoBaSSA 2005

Best Practices for Secure Coding

Robert C. Seacord

Agenda

Strings

Common String Manipulation Errors

Mitigation Strategies

Strings

Comprise most of the data exchanged between an end user and a software system

- command-line arguments
- environment variables
- console input

Software vulnerabilities and exploits are caused by weaknesses in

- string representation
- string management
- string manipulation

Agenda

Strings

Common String Manipulation Errors

Mitigation Strategies

Common String Manipulation Errors

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

Common errors include

- Unbounded string copies
- Null-termination errors
- Truncation
- 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 make errors when copying and concatenating strings because standard functions do not know the size of the destination buffer

```
1. int main(int argc, char *argv[]) {  
2.     char name[2048];  
3.     strcpy(name, argv[1]);  
4.     strcat(name, " = ");  
5.     strcat(name, argv[2]);  
        ...  
6. }
```

C++ Unbounded Copy

Inputting more than 11 characters into following the C++ program results in an out-of-bounds write:

```
1. #include <iostream.h>
2. int main() {
3.     char buf[12];
4.     cin >> buf;
5.     cout << "echo: " << buf << endl;
6. }
```


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];
```

Neither a[] nor b[] are properly terminated

```
    strncpy(a, "0123456789abcdef", sizeof(a));
```

```
    strncpy(b, "0123456789abcdef", sizeof(b));
```

```
    strncpy(c, a, sizeof(c));
```

```
}
```

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.

Write Outside Array Bounds

```
1. int main(int argc, char *argv[]) {
2.     int i = 0;
3.     char buff[128];
4.     char *arg1 = argv[1];
5.     while (arg1[i] != '\0' ) {
6.         buff[i] = arg1[i];
7.         i++;
8.     }
9.     buff[i] = '\0';
10.    printf("buff = %s\n", buff);
11. }
```

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

Agenda

Strings

Common String Manipulation Errors

Mitigation Strategies

Mitigation Strategies

ISO/IEC “Security” TR 24731

Managed string library

Safe/Secure C++

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

ISO/IEC TR 24731 Example

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

```
    char a[16];
```

```
    char b[16];
```

```
    char c[24];
```

strcpy_s() fails and generates a runtime constraint error

```
    strcpy_s(a, sizeof(a), "0123456789abcde");
```

```
    strcpy_s(b, sizeof(b), "0123456789abcde");
```

```
    strcpy_s(c, sizeof(c), a);
```

```
    strcat_s(c, sizeof(c), b);
```

```
}
```

ISO/IEC TR 24731 Summary

Already available in Microsoft Visual C++ 2005
(being released today, November 7!)

Functions are still capable of overflowing a buffer if the maximum length of the destination buffer is incorrectly specified

The ISO/IEC TR 24731 functions

- are not “fool proof”
- useful in
 - preventive maintenance
 - legacy system modernization

Managed Strings

Manage strings dynamically

- allocate buffers
- resize as additional memory is required

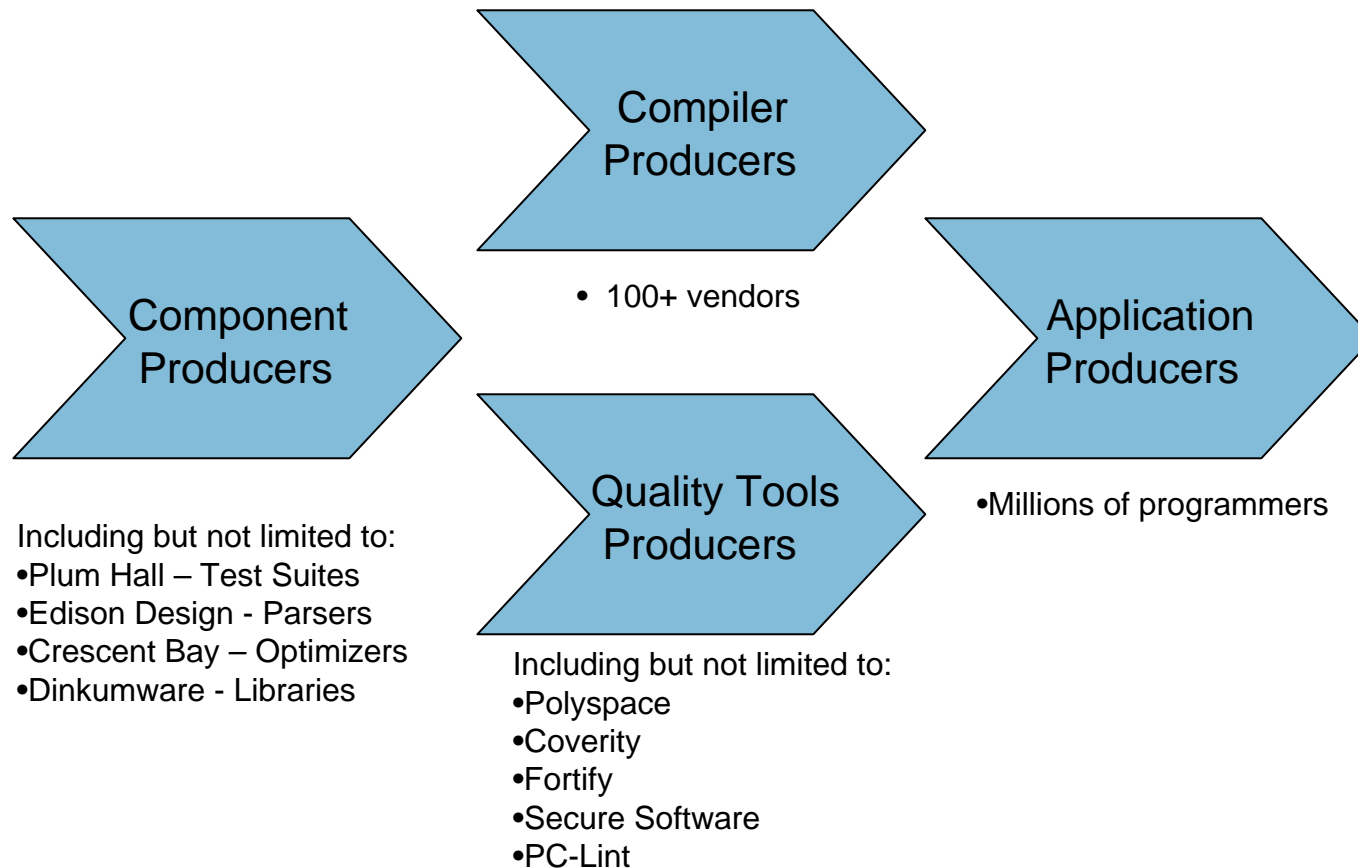
Managed string operations guarantee that

- strings operations cannot result in a buffer overflow
- data is not discarded
- strings are properly terminated (strings may or may not be null terminated internally)

Disadvantages

- unlimited can exhaust memory and be used in denial-of-service attacks
- performance overhead
- mitigation expensive

Software Production Supply Chain



Safe/Secure C++

Commercial offering being developed by Plum Hall, Inc.

Build upon today's compiler and optimizer

Match concepts to programmer intuition

Careless C/C++ code runs safely but probably slower

Performance improved by the 80/20 rule, at compile-time, at link-time, at run-time

Sample Function

```
void hbAssignCodes(
    int *code, unsigned char *length,
    int minLen, int maxLen, int alphaSize ) {
    int n, vec, i;
    vec = 0;
    for (n = minLen; n <= maxLen; n++) {
        for (i = 0; i < alphaSize; i++)
            if (length[i] == n) { code[i] = vec; vec++; };
        vec <<= 1;
    }
}
```

Step 1: Label Fetch and Store

```
void hbAssignCodes(  
    int *code, unsigned char *length,  
    int minLen, int maxLen, int alphaSize ) {  
    int n, vec, i;  
    vec = 0;  
    for (n = minLen; n <= maxLen; n++) {  
        for (i = 0; i < alphaSize; i++)  
            if (length[i] == n) { code[i] = vec; vec++; };  
        vec <<= 1;  
    }  
}
```

i SUB4 length;

i SUB4(code)

Step 2: Look For { counted } Loops

```
void hbAssignCodes(  
    int *code, unsigned char *length,  
    int minLen, int maxLen, int alphaSize ) {  
    int n, vec, i;  
    vec = 0; { counted-plus }  
    for (n = minLen; n <= maxLen; n++) { { counted-plus }  
        for (i = 0; i < alphaSize; i++)  
            if (length[i] == n) { code[i] = vec; vec++; };  
            vec <<= 1;  
        }  
    }  
}
```

Step 3: Look for Limits

```
void hbAssignCodes(  
    int *code, unsigned char *length,  
    int minLen, int maxLen, int alphaSize ) {  
    int n, vec, i;  
    vec = 0;  
    for (n = minLen; n <= maxLen; n++) {  
        for (i = 0; i < alphaSize; i++)  
            if (length[i] == n) { code[i] = vec; vec++; };  
        vec <<= 1;  
    }  
}
```

alphaSize SUB5(length)

alphaSize SUB5(length)

Step 4: New Signatures

```
void hbAssignCodes(  
    int *code, unsigned char *length,  
    int minLen, int maxLen, int alphaSize  
);
```

```
hbAssignCodes(code; length;  
    minLen; maxLen;  
    alphaSize SUB4(length),  
    SUB4(code)  
);
```


Step 5: Evaluate Code in Context

```
unsigned char len [6][258];
```

```
int code [6][258];
```

```
alphaSize SUB5(len) so alphaSize SUB5(length)
```

```
alphaSize SUB5(len) so alphaSize SUB5(code)
```

```
hbAssignCodes(code; length;
```

```
    minLen; maxLen;
```

```
    alphaSize SUB4(length),
```

```
    SUB4(code)
```

```
);
```

Bounds-checking Example

`memcpy(targ, src, num)`

becomes

`memcpy_s(targ, tsize, src, num)`

The target size of targ must be determined so it can be inserted as a new argument.

Summary

ISO/IEC TR 24731 good approach for remediation

Managed strings good approach for new development that is not performance critical

Analysis techniques based solely on detection and mitigation of dangerous functions is targeted at the wrong level of abstraction

Safe-secure C/C++ promising technology for eliminating buffer overflows and improving security of C/C++ programs

For More Information

Visit the CERT® web site

<http://www.cert.org/>

Contact Presenter

Robert C. Seacord

rsc@cert.org

Jason Rafail

jrafail@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

The image features a stylized globe with a grid pattern, showing continents in light gray and oceans in a pale blue. A large, bold, black 'CERT' logo is centered over the globe. To the left of the globe, there are several horizontal, gray, trapezoidal bars of varying lengths, stacked vertically. A thick, curved red and black border frames the left side of the image. The text 'CERT' is rendered in a bold, sans-serif font.

CERT

Back up

Data Type

Managed strings use an opaque data type

```
struct string_mx;  
  
typedef struct string_mx *string_m;
```

The representation of this type is

- private
- implementation specific

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

Create / Retrieve String Example

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

if (retValue = strcreate_m(&str1, "hello, world")) {
    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
}
```


Data Sanitization

The managed string library provides a mechanism for dealing with data sanitization by (optionally) ensuring that all characters in a string belong to a predefined set of “safe” characters.

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

Performance Breakthrough, Combining Static and Dynamic

SPEC case	SSCC penalty (raw)	SSCC penalty (adjusted)
164.gzip	11. %	1.6%
176.gcc	5.7%	0.9%
181.mcf	5.4%	0.8%
197.parser	24. %	5.8%
256.bzip2	0.0%	0.0%
300.twolf	8.7%	1.8%
AVERAGE	9.3%	1.8%