



Secure Coding

Secure Coding: Insecure Functions

Secure Coding: Insecure Functions

<http://stackoverflow.com/questions/2565727/what-are-the-c-functions-from-the-standard-library-that-must-should-be-avoided>

Functions which can create a buffer overflow:

- `gets(char *s)`
- `scanf(const char *format, ...)`
- `sprintf(char *str, const char *format, ...)`
- `strcat(char *dest, const char *src)`
- `strcpy(char *dest, const char *src)`

Secure Coding: Insecure Functions

Recap:

- Don't use functions which do not respect size of destination buffer

C Strings

C Strings

Strings in C:

```
Byte 0 to (n-1): String  
Byte n          : \0
```

Strings in Pascal:

```
Byte 0          : Length of string (n)  
Byte 1 to (n+1): String
```

C Strings

Therefore:

```
char str[8];  
strcpy(str, "1234567"); // str[7] = '\0'  
strlen(str);           // 7  
  
strcpy(str, "12345678"); // str[7] = '8'  
                        // str[8] = '\0'  
strlen(str);           // 8  
  
strcpy(str, "123456789"); // str[7] = '8'  
                        // str[8] = '9'  
                        // str[8] = '\0'  
strlen(str);           // 9
```

Overflow for input strings which are too large

```
strcpy(str, "1234567"); // str[7] = '\0'  
strlen(str);           // 7
```

```
strcpy(str, "12345678"); // str[7] = '8'  
                        // str[8] = '\0'  
strlen(str);           // 8
```

```
strcpy(str, "123456789"); // str[7] = '8'  
                        // str[8] = '9'  
                        // str[8] = '\0'  
strlen(str);           // 9
```


C Strings

Therefore:

```
char str[8];  
strcpy(str, "1234567"); // str[7] = '\0'  
  
strncpy(str, "1234567", 8); // str[7] = '\0'  
  
strncpy(str, "12345678", 8); // str[7] = '8'  
                                // (No overflow)  
  
strncpy(str, "123456789", 8); // str[7] = '8'  
                                // No overflow
```

No null terminator if input string is too large (\geq dest_len)

```
strcpy(str, "1234567"); // str[7] = '\0'
```

```
strncpy(str, "1234567", 8); // str[7] = '\0'
```

```
strncpy(str, "12345678", 8); // str[7] = '7'  
// (No overflow)
```

```
strncpy(str, "123456789", 8); // str[7] = '7'  
// No overflow
```

C Strings

Using standard C string functions on strings with missing `\0` terminator is bad

```
char str1[8];  
char str2[8];  
  
strncpy(str1, "XXXXYYY", 8);  
strncpy(str2, "AAAABBBB", 8);
```

- Result: (strlen, printf)

```
Len str1: 7
```

```
Len str2: 15
```

```
str1: XXXXYYY
```

```
str2: AAAABBBBXXXXYYY
```

C Strings

How to do it correctly:

```
strncpy(str2, "AAAABBBB", 8);  
str2[7] = "\0";
```

Or strncpy() (non-standard)

Secure Coding: Integer Overflow

Integer Overflows

“Adding a positive number to an integer might make it smaller”

Signed:

If you add a positive integer to another positive integer, the result is truncated. Technically, if you add two 32-bit numbers, the result has 33 bits.

On the CPU level, if you add two 32-bit integers, the lower 32 bits of the result are written to the destination, and the 33rd bit is signalled out in some other way, usually in the form of a "carry flag".

Integer overflows

Consists of different weaknesses:

- Unsigned Integer Wraparound
- Signed Integer Overflow
- Numeric Truncation Error

Secure Programming Practices in C++ - NDC Security 2018 (Patricia Aas)

- https://www.youtube.com/watch?v=Jh0G_A7iRac

Example 1

Integer Overflow: example 1

```
void test3(int inputLen) {
    char arr[1024];
    printf("Input len : %i / 0x%x\n", inputLen, inputLen);

    if (inputLen > 1024) {
        printf("Not enough space\n");
        return;
    }
    printf("Ok, copying...\n");
    ...
}
```

Integer Overflow: example 1

```
void test3(int inputLen) {  
    char arr[1024];  
    printf("Input len : %i / %u / 0x%x\n",  
          inputLen, inputLen, inputLen);  
    if (inputLen > 1024) {
```

```
test3(0x7fffffff);
```

```
Input len : 2147483647 / 2147483647
```

```
Not enough space
```

```
test3(0x80000000);
```

```
Input len : -2147483648 / 2147483648
```

```
Ok, copying...
```

Integer Overflow: example 1

Integer overflow problem:

Programs:

- Usually use “unsigned int”
- Indexes should be “unsigned int” (cannot be <0)
- malloc() takes a size_t (unsigned int)

Developers:

- Usually use “signed int”
- Don't want to type “unsigned...”
- Don't understand size_t
- Want to communicate error: `if(result < 0) { }`

Example 2

Integer Overflow: example 2

```
#define BUF_SIZE 256

int catvars(char *buf1, char *buf2,
            unsigned int len1, unsigned int len2)
{
    char mybuf[BUF_SIZE];

    if((len1 + len2) > BUF_SIZE) {        /* [3] */
        return -1;
    }

    memcpy(mybuf, buf1, len1);            /* [4] */
    memcpy(mybuf + len1, buf2, len2);

    do_some_stuff(mybuf);
}
```

len1: 260 / 260 / 0x104

len2: -4 / 4294967292 / 0xfffffffffc

len1 + len2: 256 / 256 / 0x100

```
if((len1 + len2) > 256) { /* [3] */
    return -1;
}

memcpy(mybuf, buf1, len1); /* [4] */
memcpy(mybuf + len1, buf2, len2);

do_some_stuff(mybuf);
```

Example 3

Example 3

```
int table[500];

int insert_in_table(int val, int pos) {
    if(pos > (sizeof(table) / sizeof(int)) ) {
        return -1;
    }

    table[pos] = val;

    return 0;
}
```


Example 4

Example 4

```
#define BUF_SIZE 32
void concat_print(
    char *first, unsigned int *first_len,
    char *second, unsigned int *second_len)
{
    char buf[BUF_SIZE];

    if (*first_len + *second_len > BUF_SIZE) {
        return;
    }

    for(unsigned int n=0; n<*first_len; n++) {
        buf[n] = first[n];
    }

    for(unsigned int n=0; n<*second_len; n++) {
        buf[*first_len + n] = second[n];
    }
}
```

Example 4

```
char first[16];  
char second[16];
```

Defined behaviour:

```
concat_print(first, 16, second, 16)    // OK (copy 32 bytes)  
concat_print(first, 16, second, 256)   // OK (wont copy anything)
```

Undefined behaviour:

```
concat_print(first, 16, second, UINT_MAX) // -> BOF
```

Example 5

Integer Overflow – Example 5

Multiplication overflow:

```
int myfunction(int *array, int len){
    int *myarray, i;

    myarray = malloc(len * sizeof(int)); /* [1] */
    if(myarray == NULL){
        return -1;
    }

    for(i = 0; i < len; i++){           /* [2] */
        myarray[i] = array[i];
    }

    return myarray
}
```

Integer overflows

C types

<http://en.cppreference.com/w/cpp/language/types>

Type specifier	Equivalent type	Width in bits by data model									
		C++ standard	LP32	ILP32	LLP64	LP64					
<code>short</code>	<code>short int</code>	at least 16	16	16	16	16					
<code>short int</code>											
<code>signed short</code>											
<code>signed short int</code>											
<code>unsigned short</code>	<code>unsigned short int</code>										
<code>unsigned short int</code>											
<code>int</code>	<code>int</code>	at least 16	16	32	32	32					
<code>signed</code>											
<code>signed int</code>											
<code>unsigned</code>											
<code>unsigned int</code>	<code>unsigned int</code>										
<code>long</code>	<code>long int</code>	at least 32	32	32	32	64					
<code>long int</code>											
<code>signed long</code>											
<code>signed long int</code>											
<code>unsigned long</code>							<code>unsigned long int</code>				
<code>unsigned long int</code>											
<code>long long</code>	<code>long long int</code> (C++11)	at least 64	64	64	64	64					
<code>long long int</code>											
<code>signed long long</code>											
<code>signed long long int</code>											
<code>unsigned long long</code>							<code>unsigned long long int</code> (C++11)				
<code>unsigned long long int</code>											

Compiler Warnings

Compiler Warnings

Anti-Formatstring Vulnerability:

Compiler:

```
-Wformat-security
```

Code:

```
printf(argv[1]);
```

Warning:

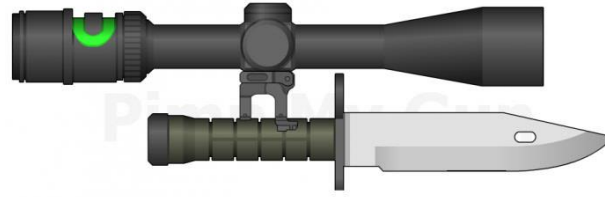
```
warning: format not a string literal and no format arguments [-Wformat-security]
```


Compiler Warnings

Make warnings into errors:

`-Werror`

Assembly



C



VIA 9SAG.COM

C++



Python



References

References:

- Catching Integer Overflows in C
 - <https://www.fefe.de/intof.html>