



# Secure Coding

# «Insecure Coding»

## Insecure Coding

- *(Buffer Overflows)*
- String handling mischief
- Integer overflows / underflows
- Information disclosure (uninitialized memory, buffer overread)
- *(Heap related; Use After Free etc.)*
- Format String Vulnerabilities

# **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**

And string function strangeness

# 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] = '8'  
                               // (No overflow)
```

```
strncpy(str, "123456789", 8); // str[7] = '8'  
                               // 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](https://www.youtube.com/watch?v=Jh0G_A7iRac)

# Integer Overflow: 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 ) { }`

# Integer Overflow: 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);
```



# Integer Overflow: 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;
}
```

# Integer Overflow: 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
```

# Integer Overflow: 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 <b>16</b>	<b>16</b>	<b>16</b>	<b>16</b>	<b>16</b>					
<code>short int</code>											
<code>signed short</code>											
<code>signed short int</code>											
<code>unsigned short</code>	<code>unsigned short int</code>	at least <b>16</b>	<b>16</b>	<b>16</b>	<b>16</b>	<b>16</b>					
<code>unsigned short int</code>											
<code>int</code>	<code>int</code>										
<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 <b>32</b>	<b>32</b>	<b>32</b>	<b>32</b>	<b>32</b>
<code>long int</code>											
<code>signed long</code>											
<code>signed long int</code>											
<code>unsigned long</code>	<code>unsigned long int</code>						at least <b>32</b>	<b>32</b>	<b>32</b>	<b>32</b>	<b>64</b>
<code>unsigned long int</code>											
<code>long long</code>	<code>long long int</code> (C++11)										
<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)	at least <b>64</b>	<b>64</b>	<b>64</b>	<b>64</b>	<b>64</b>					
<code>unsigned long long int</code>											



# Information Disclosure

# Heartbleed

<https://www.vulncode-db.com/CVE-2014-0160>

“The Heartbleed bug is an issue with the Heartbeat protocol that is used for [...]. It allows an attacker to exfiltrate up to 16 KB memory data from a target running a vulnerable OpenSSL version.”

ssl/t1\_lib.c    openssl

```
2616    memcpy(bp, p1, payload);  
2617    bp += payload;
```

Copies `payload` bytes (up to 64KB) from `p1` (pointing on the resulting heartbeat buffer) into `bp`.

The critical part is that `payload` (the user-supplied payload length) can be greater than the actual size of the `p1` memory segment leading to an out-of-bounds read effectively copying parts of the memory into `bp`.

# Format String Vulnerabilities

## Format String Vulnerability: Lazy/wrong code

Correct:

```
printf("%s: %i", user_input, some_int);
```

Possible, but wrong:

```
printf(user_input);
```

will create mischief with:

```
user_input = "Its me, mario! %x %x %x"
```

# Format String Vulnerability: User-supplied format string

```
int main (int argc, char **argv)
{
    char buf [100];
    int x = 1 ;
    snprintf ( buf, sizeof buf, argv[1] ) ;
    buf [ sizeof buf -1 ] = 0;
    printf ( "Buffer size is: (%d) \nData input: %s \n" , strlen (buf) , buf ) ;
    printf ( "X equals: %d/ in hex: %#x\nMemory address for x: (%p) \n" , x, x, &x) ;
    return 0 ;
}
```

```
./formattest "Bob %x %x"
```

```
Buffer size is (14)
Data input : Bob bffff 8740
X equals: 1/ in hex: 0x1
Memory address for x (0xbffff73c)
```

Because:

```
int snprintf(char *str, size_t size, const char *format, ...);
snprintf(buf, sizeof(buf), "%s", user_supplied);
```

# Format String Vulnerability Exploitation

[http://www.cis.syr.edu/~wedu/Teaching/cis643/LectureNotes\\_New/Format\\_String.pdf](http://www.cis.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Format_String.pdf)

- Writing an integer to nearly any location in the process memory
  - `%n`: The number of characters written so far is stored into the integer indicated by the corresponding argument.

```
int i;  
printf ("12345%n", &i);
```

- It causes `printf()` to write 5 into variable `i`.
- Using the same approach as that for viewing memory at any location, we can cause `printf()` to write an integer into any location. Just replace the `%s` in the above example with `%n`, and the contents at the address `0x10014808` will be overwritten.

# Format String Vulnerabilities: Solved

Anti-Formatstring Vulnerability:

Compiler:

`-Wformat-security`

Code:

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

Warning:

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

# Some Buffer Overflow Bugs



# Some Bugs: Mongoose MQTT

```
static void mg_mqtt_broker_handle_subscribe(struct mg_connection *nc,
                                           struct mg_mqtt_message *msg) {
    struct mg_mqtt_session *ss = (struct mg_mqtt_session *) nc->user_data;
    uint8_t qoss[512]; // static size, will be overflowed
    size_t qoss_len = 0;
    struct mg_str topic;
    uint8_t qos;
    int pos;
    struct mg_mqtt_topic_expression *te;

    for (pos = 0;
         (pos=mg_mqtt_next_subscribe_topic(msg, &topic, &qos, pos)) != -1;) {
        qoss[qoss_len++] = qos; // Stack based buffer overflow here
    }
    [...]
}
```

# Some Bugs: Exim Off By One buffer overflow

<https://devco.re/blog/2018/03/06/exim-off-by-one-RCE-exploiting-CVE-2018-6789-en/>

```
b64decode(const uschar *code, uschar **ptr)
{
    int x, y;
    uschar *result = store_get(3*(Ustrlen(code)/4) + 1);

    *ptr = result;
    // perform decoding
}
```

As shown above, exim allocates a buffer of  $3*(len/4)+1$  bytes to store decoded base64 data. However, when the input is not a valid base64 string and the length is  $4n+3$ , exim allocates  $3n+1$  but consumes  $3n+2$  bytes while decoding. This causes one byte heap overflow (aka off-by-one).

## Some Bugs: Netkit-telnetd buffer overflow

```
static void
encrypt_keyid(struct key_info *kp, unsigned char *keyid, int len)
{
    ...
    if (!(ep = (*kp->getcrypt)(*kp->modep))) {
        ...
    } else if ((len != kp->keylen)
               || (memcmp(keyid, kp->keyid, len) != 0)) {
        /* Length or contents are different */
        kp->keylen = len;
        memcpy(kp->keyid, keyid, len);
    }
}
```

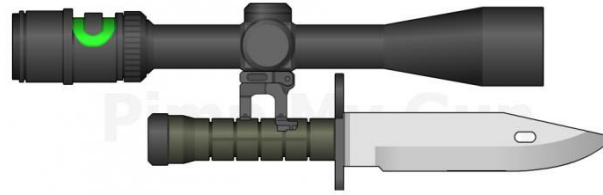
# Some Bugs: iOS 11 Multipath TCP

Let's first take a quick look at the offending code in `mptcp_usr_connect()`, which is the handler for the `connectx` syscall for the `AP_MULTIPATH` socket family:

```
1  if (src) {
2      // verify sa_len for AF_INET
3      if (src->sa_family == AF_INET &&
4          src->sa_len != sizeof(mpte->__mpte_src_v4)) {
5          mptcplog((LOG_ERR, "%s IPv4 src len %u\n", __func__,
6                  src->sa_len),
7                  MPTCP_SOCKET_DBG, MPTCP_LOGLVL_ERR);
8          error = EINVAL;
9          goto out;
10     }
11
12     // verify sa_len for AF_INET6
13     if (src->sa_family == AF_INET6 &&
14         src->sa_len != sizeof(mpte->__mpte_src_v6)) {
15         mptcplog((LOG_ERR, "%s IPv6 src len %u\n", __func__,
16                 src->sa_len),
17                 MPTCP_SOCKET_DBG, MPTCP_LOGLVL_ERR);
18         error = EINVAL;
19         goto out;
20     }
21
22     // code doesn't bail if sa_family is neither AF_INET nor AF_INET6
23     if ((mp_so->so_state & (SS_ISCONNECTED|SS_ISCONNECTING)) == 0) {
24         memcpy(&mpte->mpte_src, src, src->sa_len);
25     }
26 }
```

The code does not validate the `sa_len` field if `src->sa_family` is neither `AF_INET` nor `AF_INET6` so the function directly falls through to `memcpy` with a user specified `sa_len` value up to 255 bytes.

## Assembly



## C



## C++



## Python



# References

References:

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