«Insecure Coding»

Insecure Coding

- *(Buffer Overflows)*

- String handling mischief

- Integer overflows / underflows

- Information disclosure (unitialized memory, buffer overread)

Secure Coding: Insecure Functions

Secure Coding: Insecure Functions

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
```

data	0x00

Strings in Pascal:

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

0xa1	data

C Strings

Threrefore:

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


strcpy(str, "12345678");  // str[7] = '8'
                          // str[8] = '\0'
strlen(str);              // 8 (9 bytes)


strcpy(str, "123456789");   // str[7] = '8'
                            // str[8] = '9'
                            // str[9] = '\0'
strlen(str);                // 9 (10 bytes)
```

C Strings

Thererefore:

```c
char str[8];

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);


Len str1: 7
Len str2: 15
str1: XXXXYYY
str2: AAAABBBBXXXXYYY
```

C Strings

How to do it correctly

```
#def BUF_SIZE 8
char str1[BUF_SIZE];
char str2[BUF_SIZE];


strncpy(str1, "XXXXYYYY", BUF_SIZE);
str1[BUF_SIZE-1] = "\0";


strncpy(str2, "AAAABBBB", BUF_SIZE);
str2[BUF_SIZE-1] = "\0";


Len str1: 7
Len str2: 7
str1: XXXXYYY
str2: AAAABBB
```

Secure Coding: Integer Overflow

Signed integer

Signed int: can be negative
Halves the amount of numbers it can store

Mapping Signed ↔ Unsigned

Bits	Signed		Unsigned
0000	0		0
0001	1		1
0010	2		2
0011	3	=	3
0100	4		4
0101	5		5
0110	6		6
0111	7		7
1000	−8		8
1001	−7		9
1010	−6		10
1011	−5	+/- 16	11
1100	−4		12
1101	−3		13
1110	−2		14
1111	−1		15

Carnegie Mellon

24

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**
short short int signed short signed short int	short int	at least 16	16	16	16	16
unsigned short unsigned short int	unsigned short int					
int signed signed int	int	at least 16	16	32	32	32
unsigned unsigned int	unsigned int					
long long int signed long signed long int	long int	at least 32	32	32	32	64
unsigned long unsigned long int	unsigned long int					
long long long long int signed long long signed long long int	long long int (C++11)	at least 64	64	64	64	64
unsigned long long unsigned long long int	unsigned long long int (C++11)					

Integer Overflows

"Adding a positive number to an integer might make it smaller"

Unsigned:

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

Integer Overflow: Example 1

signed int

Integer Overflow: example 1 - signed int

NAME
        memcpy - copy memory area

SYNOPSIS
        #include <string.h>

        void *memcpy(void *dest, const void *src, **size_t n**);

DESCRIPTION
        The memcpy() function copies n bytes from memory area src to memory area dest. The
memory areas must not overlap. Use memmove(3) if the memory areas do overlap.

*According to the 1999 ISO C standard (C99), **size_t is an unsigned integer** type of at least 16 bit
(see sections 7.17 and 7.18.3)*

Integer Overflow: example 1 - signed int

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

        if (inputLen > 1024) {
                printf("Not enough space\n");
                return;
        }
        printf("Ok, copying %u\n", inputLen);
        memcpy(arr, input, inputLen);
        …
}
```

Integer Overflow: example 1 - signed int

```c
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: 2147483648

Integer Overflow: example 1

Integer overflow problem:

Programs:
- Usually use "signed int" (can be smaller than 0, half the space)
- Indexes should be "unsigned int" (always positive)
- But: malloc() takes a size_t (unsigned int)!

Developers:
- Usually use "int" = "signed int"
- Don't want to type "unsigned…"
- Don't understand size_t
- Want to communicate error: if(result < 0) { }

Integer Overflow: Example 2

unsigned int overflow / truncate

Integer Overflow: example 2 - unsigned int overflow / truncate

```c
#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){     // Truncate if (len1 + len2) > 2^32
      return -1;
  }

  memcpy(mybuf, buf1, len1);
  memcpy(mybuf + len1, buf2, len2); // buffer overflow

  do_some_stuff(mybuf);
```

```
len1: 260  / 260            / 0x104
len2: -4    / 4294967292 / 0xfffffffc

len1 + len2: 256 / 256 / 0x100
```

```
if((len1 + len2) > 256){
    return -1;
}

// We arrive here
memcpy(mybuf, buf1, len1);
memcpy(mybuf + len1, buf2, len2); // len2 = 0xfffffffc

do_some_stuff(mybuf);
```

Integer Overflow: example 2 - unsigned int overflow / truncate

Adding two unsigned int can produce an overflow
Overflow bit is just "forgotten"

```
  0x    FFFFFFFF
+ 0x    FFFFFFFF
= 0x   1FFFFFFFE
= 0x    FFFFFFFE
```

Integer Overflow: Example 3

signed array index

Example 3 - signed array index

```c
int table[500];

int insert_in_table(int val, int pos) {
    if(pos > 500) {
        return -1;
    }

    table[pos] = val; // pos = -1?
    return 0;
}
```

Integer Overflow: Example 4

Integer Overflow – Example 4

Multiplication overflow:

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

    myarray = malloc(len * sizeof(int));    // len < 0? len = 0?
    if(myarray == NULL){
        return -1;
    }

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

    return myarray
}
```

Information Disclosure

Heartbleed

"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."
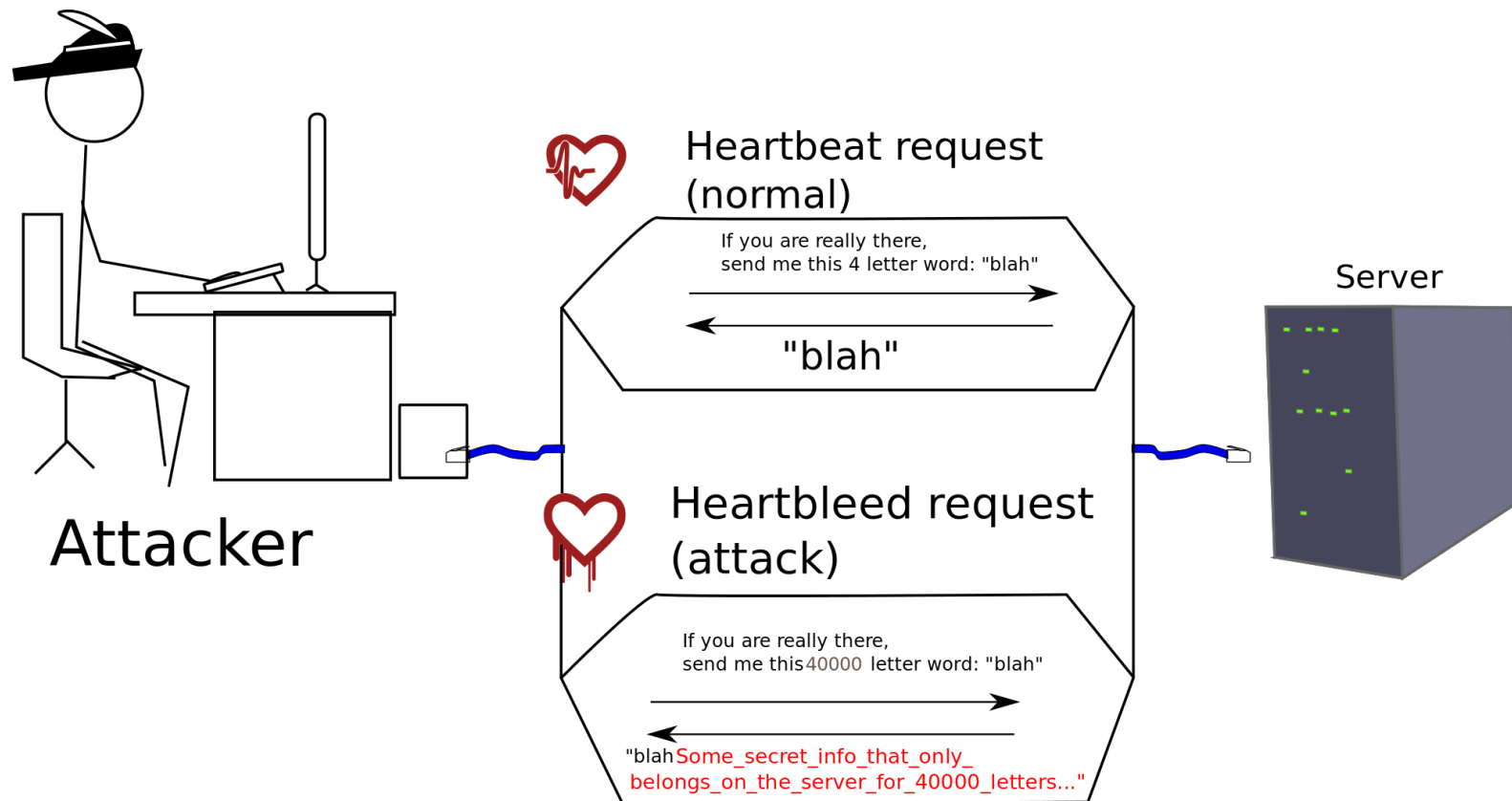


```
ssl/t1_lib.c      openssl
2616              memcpy(bp, pl, payload);
2617              bp += payload;
```

Copies `payload` bytes (up to 64KB) from `pl` (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 `pl` memory segment leading to an out-of-bounds read effectively copying parts of the memory into `bp`.

Heartbleed

```
char buffer[1024];
write(socket, buffer, len_attacker);  // e.g. 40'000
```


Heartbeat request (normal)

If you are really there,
send me this 4 letter word: "blah"

"blah"

Attacker

Heartbleed request (attack)

If you are really there,
send me this 40000 letter word: "blah"

"blah Some_secret_info_that_only_
belongs_on_the_server_for_40000_letters..."

Server

Some Buffer Overflow Bugs

Some Bugs: Mongoose MQTT

```
MG_INTERNAL int parse_mqtt(struct mbuf *io, struct mg_mqtt_message *mm) {
    const char *p = &io->buf[1], *end;
[...]
    /* decode mqtt variable length */
    // In Fixed header
    do {
      len += (*p & 127) << 7 * (p - &io->buf[1]);
    } while ((*p++ & 128) != 0 && ((size_t)(p - io->buf) <= io->len));

    // end = p for (attacker controlled) len = 0
    end = p + len;
    if (end > io->buf + io->len + 1) {
      return -1;
    }
```

Some Bugs: Mongoose MQTT

```
case MG_MQTT_CMD_SUBSCRIBE:
    mm->message_id = getu16(p); // Variable header
    p += 2; // p > end for len = 0
    /*
     * topic expressions are left in the payload and can be parsed with
     * `mg_mqtt_next_subscribe_topic`
     */
    mm->payload.p = p;
    mm->payload.len = end - p; // mm->payload.len < 0 for len = 0
    printf("MQTT Subscribe 1: p: %p   len: %lx\n",
      mm->payload.p, mm->payload.len);
    break;
```

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

```
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++
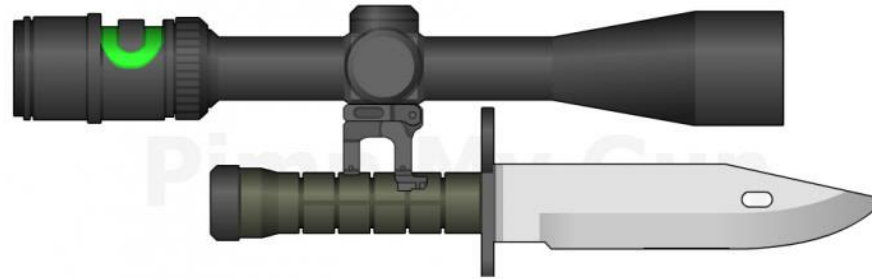
References

References:

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