Buffer Overflow

Security 1 2018-19

Università Ca’ Foscari Venezia

www.dais.unive.it/~focardi

secgroup.dais.unive.it

Università
Ca’Foscari
Venezia

Introduction

Buffer overflow is one of the most common vulnerabilities

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

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

- **1988** The Morris Internet Worm uses 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 (Microsoft IIS 5.0)
- **2003** Slammer worm (Microsoft SQL Server 2000)
- **2004** Sasser worm (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 or to **insert specially crafted code** that allows them to gain control of the system."

Sources

Buffer overflow can be caused by

- Reading data from stdin
- Copying/merging data
- Bugs in boundary check (off-by-one)
- Copying strings
- Appending strings
- Creating a string
- ....

Effects

The buffer can be located on the stack, in the heap, or in the data section of the process

Overwriting adjacent memory locations can

- modify other variables (**corruption of data**)
- modify the program control flow data such as return addresses and pointers to previous stack frames (**corruption of control**)

In the worst case, the attacker will execute **arbitrary code** with the **privileges of the attacked process**

Safe vs. unsafe languages

Assembly does not provide any notion of **type**

- data can be interpreted and used in any way
- programmers should enforce safe execution

Safe languages such as Java, ADA, Python are safe

- strong notion of **types**
- overflows are not possible

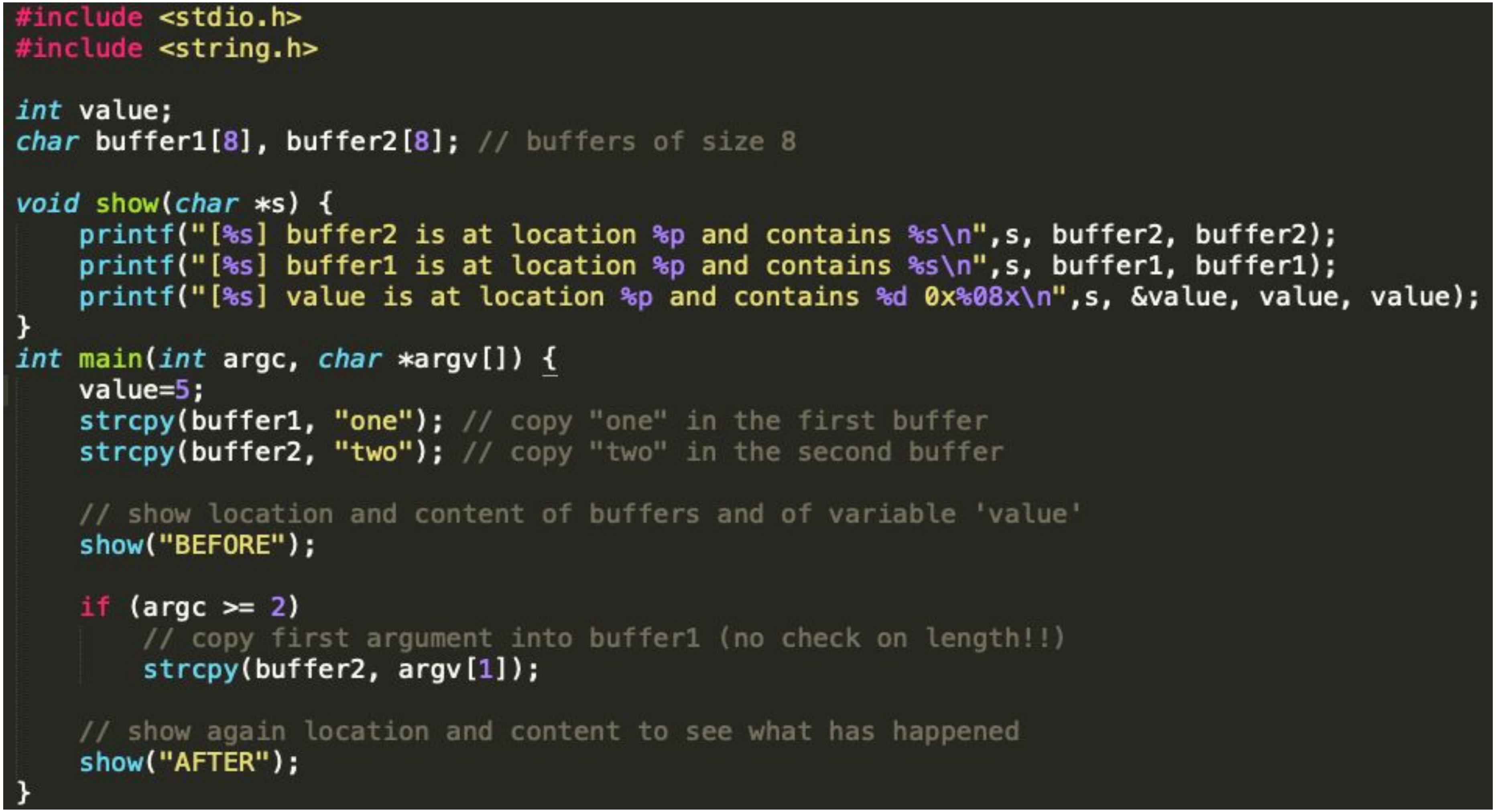
C is in between

- **weaker types** and direct access to memory
- overflows are possible

Common unsafe C functions

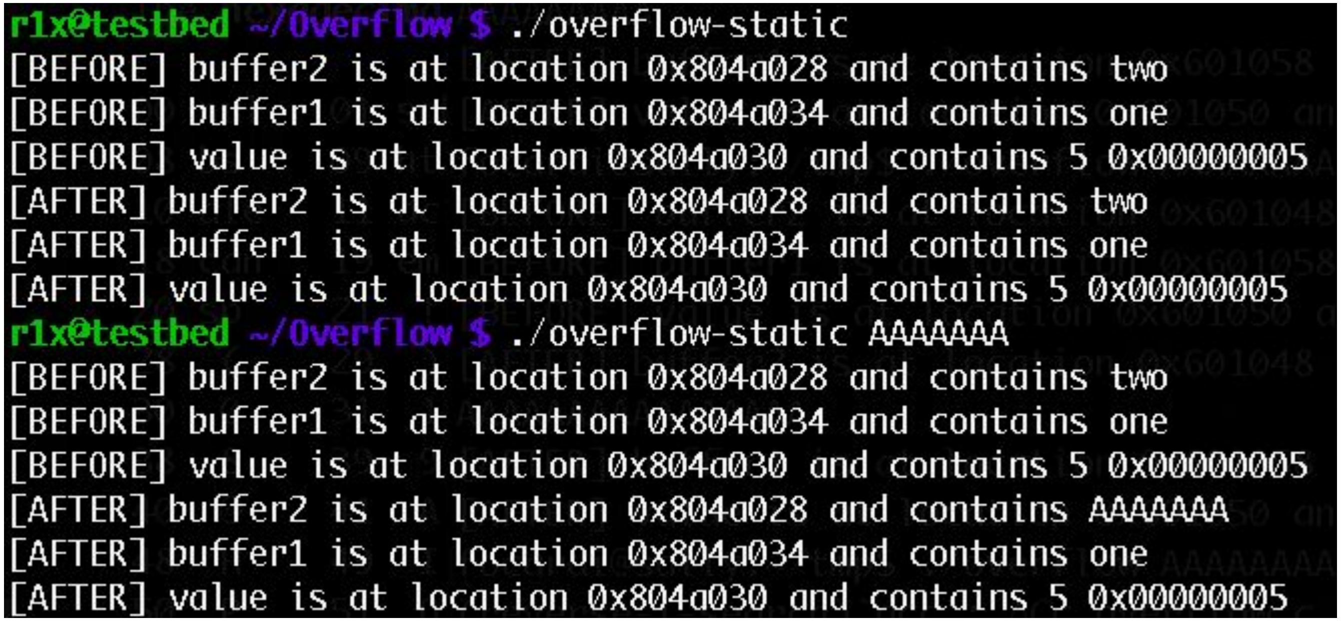
- **gets**(char *str)
  read line from standard input into str
- **sprintf**(char *str, char *format, ...)
  create str according to supplied format and variables
- **strcat**(char *dest, char *src)
  append contents of string src to string dest
- **strcpy**(char *dest, char *src)
  copy contents of string src to string dest
- **vsprintf**(char *str, char *fmt, va_list ap)
  create str according to supplied format and variables

Example: simple overflow



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

int value;
char buffer1[8], buffer2[8]; // buffers of size 8

void show(char *s) {
    printf("[%s] buffer2 is at location %p and contains %s\n",s, buffer2, buffer2);
    printf("[%s] buffer1 is at location %p and contains %s\n",s, buffer1, buffer1);
    printf("[%s] value is at location %p and contains %d 0x%08x\n",s, &value, value, value);
}
int main(int argc, char *argv[]) {
    value=5;
    strcpy(buffer1, "one"); // copy "one" in the first buffer
    strcpy(buffer2, "two"); // copy "two" in the second buffer

    // show location and content of buffers and of variable 'value'
    show("BEFORE");

    if (argc >= 2)
        // copy first argument into buffer1 (no check on length!!)
        strcpy(buffer2, argv[1]);

    // show again location and content to see what has happened
    show("AFTER");
}
```

Example: simple overflow

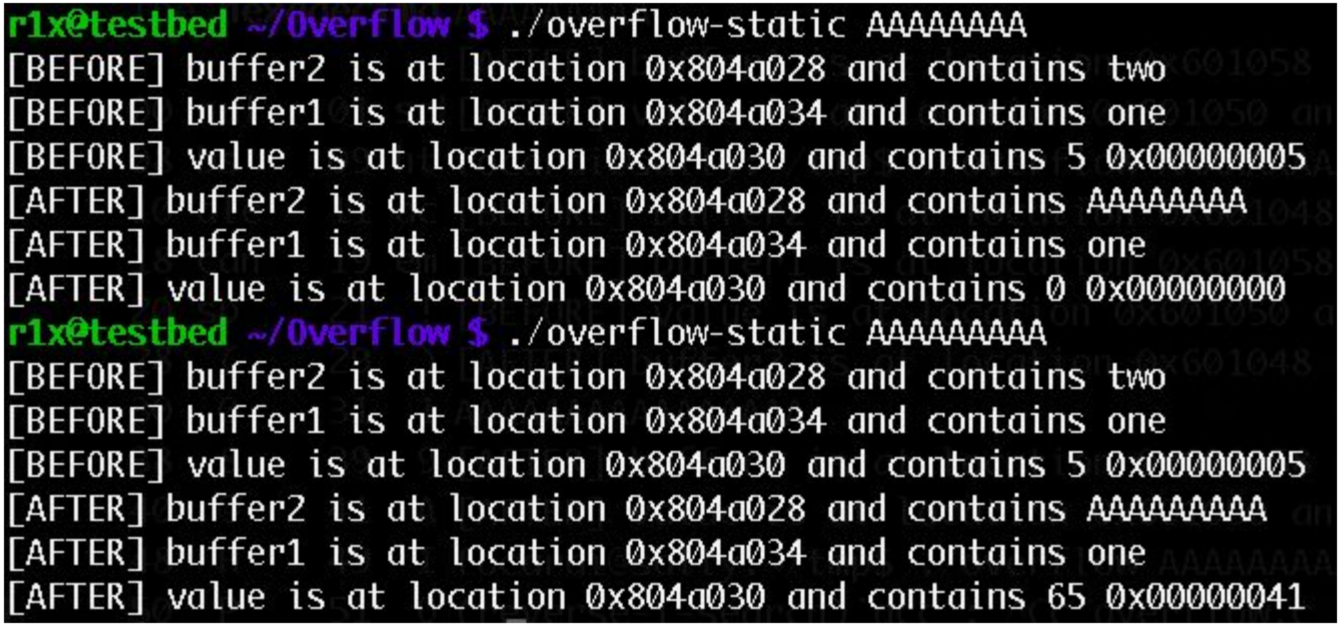


```
r1x@testbed ~/Overflow $ ./overflow-static
[BEFORE] buffer2 is at location 0x804a028 and contains two
[BEFORE] buffer1 is at location 0x804a034 and contains one
[BEFORE] value is at location 0x804a030 and contains 5 0x00000005
[AFTER] buffer2 is at location 0x804a028 and contains two
[AFTER] buffer1 is at location 0x804a034 and contains one
[AFTER] value is at location 0x804a030 and contains 5 0x00000005
r1x@testbed ~/Overflow $ ./overflow-static AAAAAAA
[BEFORE] buffer2 is at location 0x804a028 and contains two
[BEFORE] buffer1 is at location 0x804a034 and contains one
[BEFORE] value is at location 0x804a030 and contains 5 0x00000005
[AFTER] buffer2 is at location 0x804a028 and contains AAAAAAA
[AFTER] buffer1 is at location 0x804a034 and contains one
[AFTER] value is at location 0x804a030 and contains 5 0x00000005
```

Memory layout

`buffer2`	(8 bytes)
`value`	(4 bytes)
`buffer1`	(8 bytes)

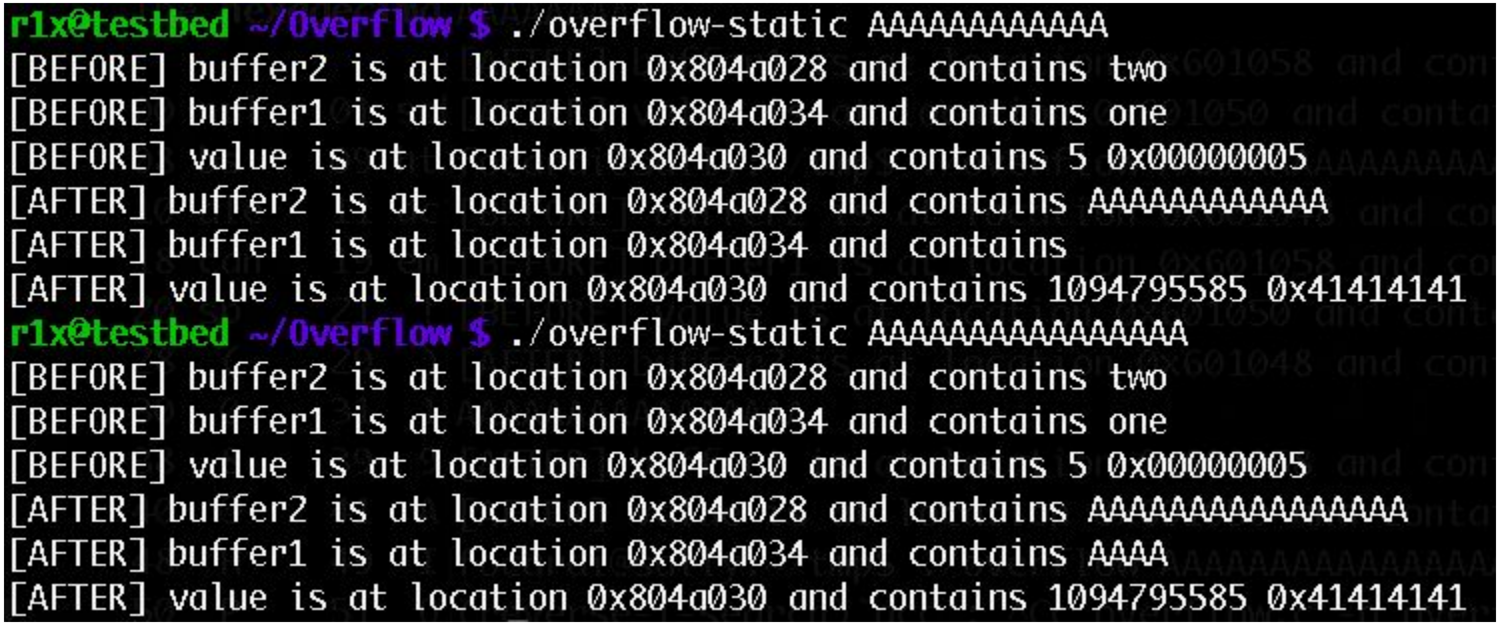
Example: simple overflow



```
r1x@testbed ~/Overflow $ ./overflow-static AAAAAAAA
[BEFORE] buffer2 is at location 0x804a028 and contains two
[BEFORE] buffer1 is at location 0x804a034 and contains one
[BEFORE] value is at location 0x804a030 and contains 5 0x00000005
[AFTER] buffer2 is at location 0x804a028 and contains AAAAAAAA
[AFTER] buffer1 is at location 0x804a034 and contains one
[AFTER] value is at location 0x804a030 and contains 0 0x00000000
r1x@testbed ~/Overflow $ ./overflow-static AAAAAAAAA
[BEFORE] buffer2 is at location 0x804a028 and contains two
[BEFORE] buffer1 is at location 0x804a034 and contains one
[BEFORE] value is at location 0x804a030 and contains 5 0x00000005
[AFTER] buffer2 is at location 0x804a028 and contains AAAAAAAAA
[AFTER] buffer1 is at location 0x804a034 and contains one
[AFTER] value is at location 0x804a030 and contains 65 0x00000041
```

- One more A puts the `0x00` terminator over value!
- An extra A overwrites `value` with `0x41` (A)

Example: simple overflow



```
r1x@testbed ~/Overflow $ ./overflow-static AAAAAAAAAAAA
[BEFORE] buffer2 is at location 0x804a028 and contains two
[BEFORE] buffer1 is at location 0x804a034 and contains one
[BEFORE] value is at location 0x804a030 and contains 5 0x00000005
[AFTER] buffer2 is at location 0x804a028 and contains AAAAAAAAAAAA
[AFTER] buffer1 is at location 0x804a034 and contains
[AFTER] value is at location 0x804a030 and contains 1094795585 0x41414141
r1x@testbed ~/Overflow $ ./overflow-static AAAAAAAAAAAAAAAA
[BEFORE] buffer2 is at location 0x804a028 and contains two
[BEFORE] buffer1 is at location 0x804a034 and contains one
[BEFORE] value is at location 0x804a030 and contains 5 0x00000005
[AFTER] buffer2 is at location 0x804a028 and contains AAAAAAAAAAAAAAAA
[AFTER] buffer1 is at location 0x804a034 and contains AAAA
[AFTER] value is at location 0x804a030 and contains 1094795585 0x41414141
```

- Three more A’s fully overwrite value and put `0x00` over `buffer1`
- Extra A’s overwrite `buffer1`

Università Ca’Foscari Venezia

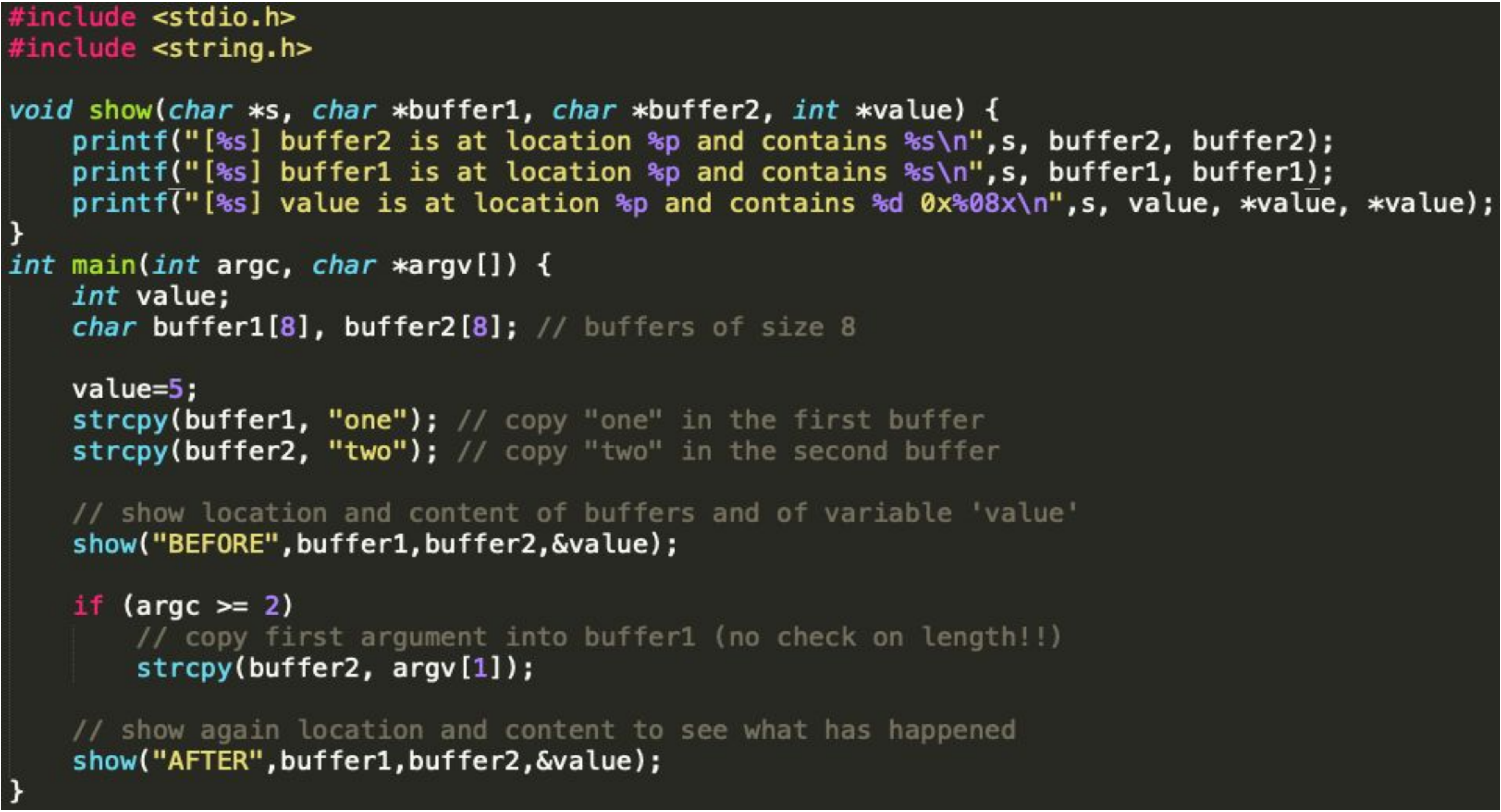
Stack protector

A set of mechanisms to **mitigate** buffer overflow attacks

- Variables on the stack are rearranged so to minimize the effects of overflows
- buffers are put **after** non-buffers
- overflow might affect other buffers but NOT variables that are not buffers (e.g. integers)
- non-buffers will never overflow so putting them before buffers is safe

(Stack protector does more but we will see this next week)

Example: stack protector

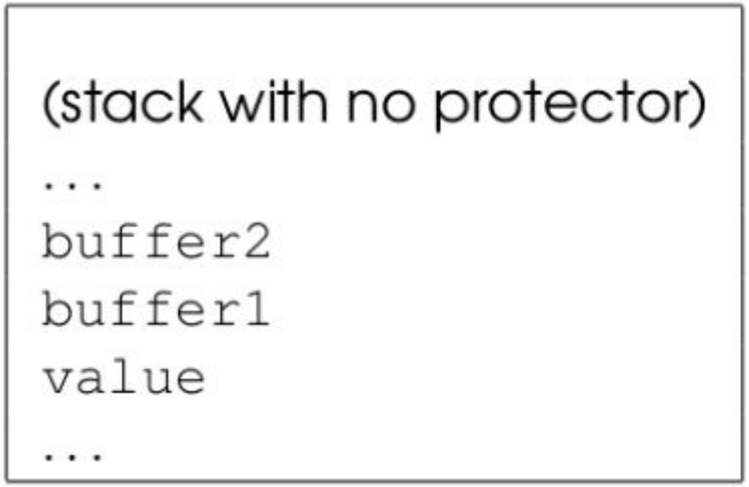


```
#include <stdio.h>
#include <string.h>

void show(char *s, char *buffer1, char *buffer2, int *value) {
    printf("[%s] buffer2 is at location %p and contains %s\n",s, buffer2, buffer2);
    printf("[%s] buffer1 is at location %p and contains %s\n",s, buffer1, buffer1);
    printf("[%s] value is at location %p and contains %d 0x%08x\n",s, value, *value, *value);
}
int main(int argc, char *argv[]) {
    int value;
    char buffer1[8], buffer2[8]; // buffers of size 8

    value=5;
    strcpy(buffer1, "one"); // copy "one" in the first buffer
    strcpy(buffer2, "two"); // copy "two" in the second buffer

    // show location and content of buffers and of variable 'value'
    show("BEFORE",buffer1,buffer2,&value);

    if (argc >= 2)
        // copy first argument into buffer1 (no check on length!!)
        strcpy(buffer2, argv[1]);

    // show again location and content to see what has happened
    show("AFTER",buffer1,buffer2,&value);
}
```

Example: stack protector

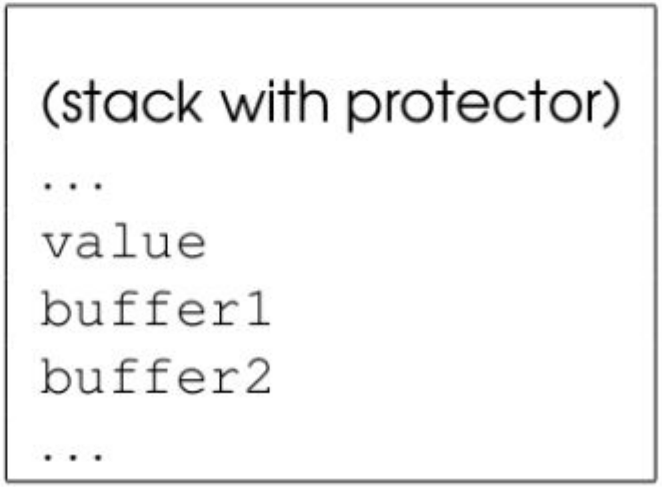
Compile the program with or without

`-fno-stack-protector`



(stack with no protector)

```
...
buffer2
buffer1
value
...
```



(stack with protector)

```
...
value
buffer1
buffer2
...
```

Exercises

1. Try the overflow with and without stack protector
2. Exploit the password checking example

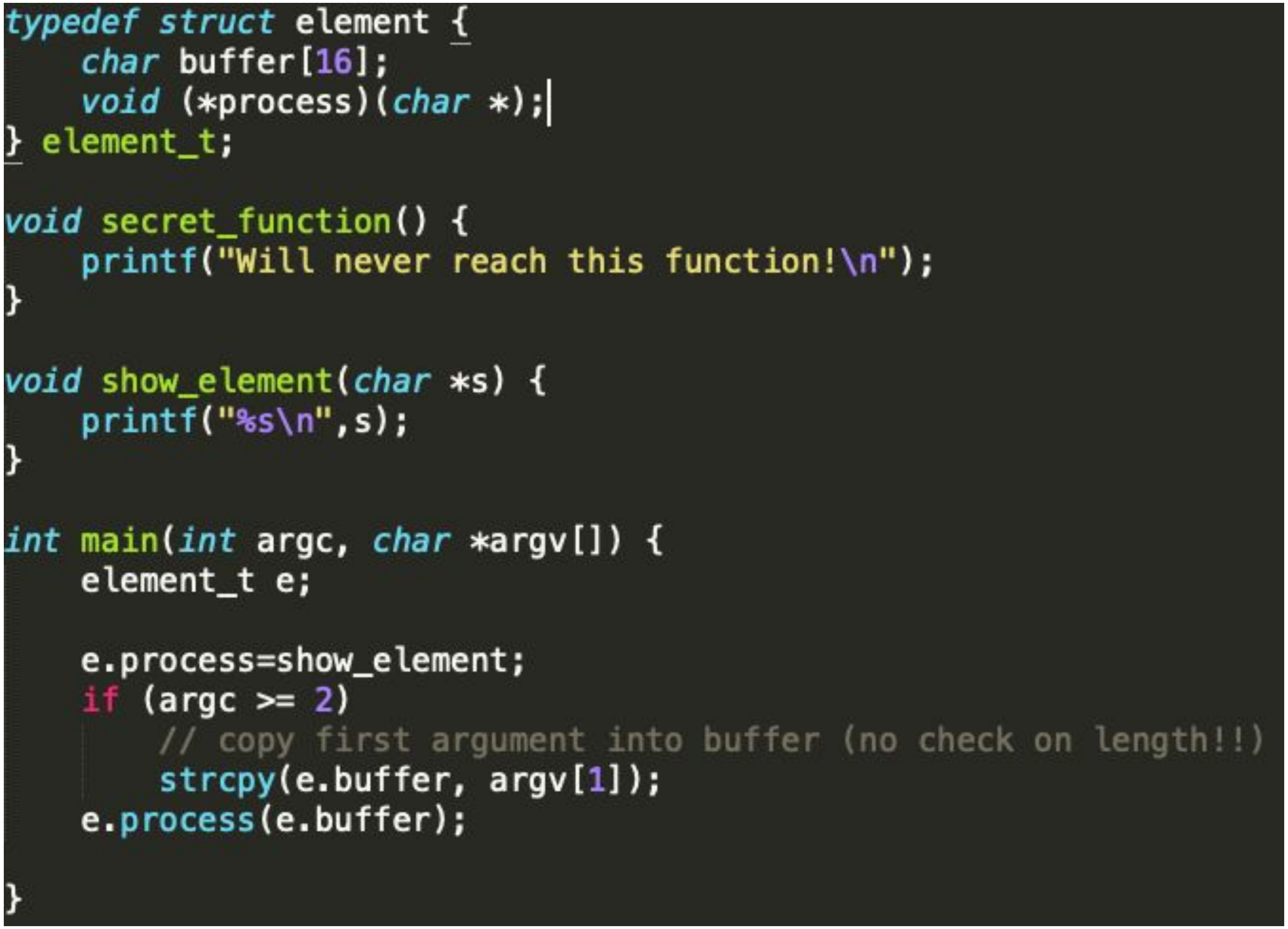
Modifying the control flow

It can happen that an overflow overwrites an address that corresponds to **code** o to some **structured data**

- A **function pointer**
- The return address of a **function**
- A pointer to a structure (**stack**, **heap**, …)
- …

This can change the program control flow!

Example: subverting control flow



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

void secret_function() {
    printf("Will never reach this function!\n");
}

void show_element(char *s) {
    printf("%s\n",s);
}

int main(int argc, char *argv[]) {
    element_t e;

    e.process=show_element;
    if (argc >= 2)
        // copy first argument into buffer (no check on length!!)
        strcpy(e.buffer, argv[1]);
    e.process(e.buffer);

}
```

Example: subverting control flow

We can overwrite the function address in `e.process`

To do so:

- We need to fill the **buffer** (e.g. with A's)
- We need to look for an **interesting address** to jump to (e.g. `secret_function`)
- We need to inject the address in the buffer (as **bytes!**)
- Don't forget **endianness!!**

Let's try it