

## News from CitizenLab last week

# BLASTPASS

## NSO Group iPhone Zero-Click, Zero-Day Exploit Captured in the Wild

September 7, 2023

Last week, while checking the device of an individual employed by a Washington DC-based civil society organization with international offices, Citizen Lab found an actively exploited zero-click vulnerability being used to deliver NSO Group's Pegasus mercenary spyware.

### The BLASTPASS Exploit Chain

We refer to the exploit chain as BLASTPASS. The exploit chain was capable of compromising iPhones running the latest version of iOS (16.6) *without any interaction from the victim.*

The exploit involved *PassKit attachments containing malicious images* sent from an attacker *iMessage* account to the victim.

We expect to publish a more detailed discussion of the exploit chain in the future.

### Disclosure to Apple & CVEs

Citizen Lab immediately disclosed our findings to Apple and assisted in their investigation.

Apple issued two CVEs related to this exploit chain (CVE-2023-41064 and CVE-2023-41061)

### Update Apple Devices Now

We urge everyone to immediately update their devices.

**We encourage everyone who may face increased risk because of who they are or what they do to [enable Lockdown Mode.](#)**

<https://citizenlab.ca/2023/09/blastpass-nso-group-iphone-zero-click-zero-day-exploit-captured-in-the-wild/>

# 🚫 CVE-2023-41061 Detail


## Description

A **validation issue** was addressed with improved logic. This issue is fixed in watchOS 9.6.2, iOS 16.6.1 and iPadOS 16.6.1. A maliciously crafted attachment may result in arbitrary code execution. Apple is aware of a report that this issue may have been actively exploited.

### Severity

CVSS Version 3.x CVSS Version 2.0

**CVSS 3.x Severity and Metrics:**

 **NIST:** NVD **Base Score:** 7.8 HIGH

**Vector:** CVSS:3.1/AV:L/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H

# 🚫 CVE-2023-41064 Detail


## Description

A **buffer overflow** issue was addressed with improved memory handling. This issue is fixed in macOS Monterey 12.6.9, macOS Big Sur 11.7.10, macOS Ventura 13.5.2, iOS 16.6.1 and iPadOS 16.6.1, iOS 15.7.9 and iPadOS 15.7.9. Processing a maliciously crafted image may lead to arbitrary code execution. Apple is aware of a report that this issue may have been actively exploited.

### Severity

CVSS Version 3.x CVSS Version 2.0

**CVSS 3.x Severity and Metrics:**

 **NIST:** NVD **Base Score:** 7.8 HIGH

**Vector:** CVSS:3.1/AV:L/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H

## (Related?) problem reported by Mozilla

# Mozilla patches Firefox, Thunderbird against zero-day exploited in attacks

By [Sergiu Gatlan](#)

September 12, 2023

05:32 PM

0

CVE-ID	
<b>CVE-2023-4863</b>	<a href="#">Learn more at National Vulnerability Database (NVD)</a> <ul style="list-style-type: none"><li>• CVSS Severity Rating</li><li>• Fix Information</li><li>• Vulnerable Software Versions</li><li>• SCAP Mappings</li><li>• CPE Information</li></ul>
Description	
<p>Heap buffer overflow in WebP in Google Chrome prior to 116.0.5845.187 allowed a remote attacker to perform an out of bounds memory write via a crafted HTML page. (Chromium security severity: Critical)</p>	
References	
<p><b>Note:</b> <a href="#">References</a> are provided for the convenience of the reader to help distinguish between vulnerabilities. The list is not intended to be complete.</p>	
<ul style="list-style-type: none"><li>• <a href="https://bugzilla.suse.com/show_bug.cgi?id=1215231">MISC:https://bugzilla.suse.com/show_bug.cgi?id=1215231</a></li><li>• <a href="https://en.bandisoft.com/honeyview/history/">MISC:https://en.bandisoft.com/honeyview/history/</a></li><li>• <a href="https://github.com/webmproject/libwebp/commit/902bc9190331343b2017211debcec8d2ab87e17a">MISC:https://github.com/webmproject/libwebp/commit/902bc9190331343b2017211debcec8d2ab87e17a</a></li><li>• <a href="https://msrc.microsoft.com/update-guide/vulnerability/CVE-2023-4863">MISC:https://msrc.microsoft.com/update-guide/vulnerability/CVE-2023-4863</a></li><li>• <a href="https://news.ycombinator.com/item?id=37478403">MISC:https://news.ycombinator.com/item?id=37478403</a></li></ul>	

<https://www.bleepingcomputer.com/news/security/mozilla-patches-firefox-thunderbird-against-zero-day-exploited-in-attacks/>

Software Security

# Memory corruption

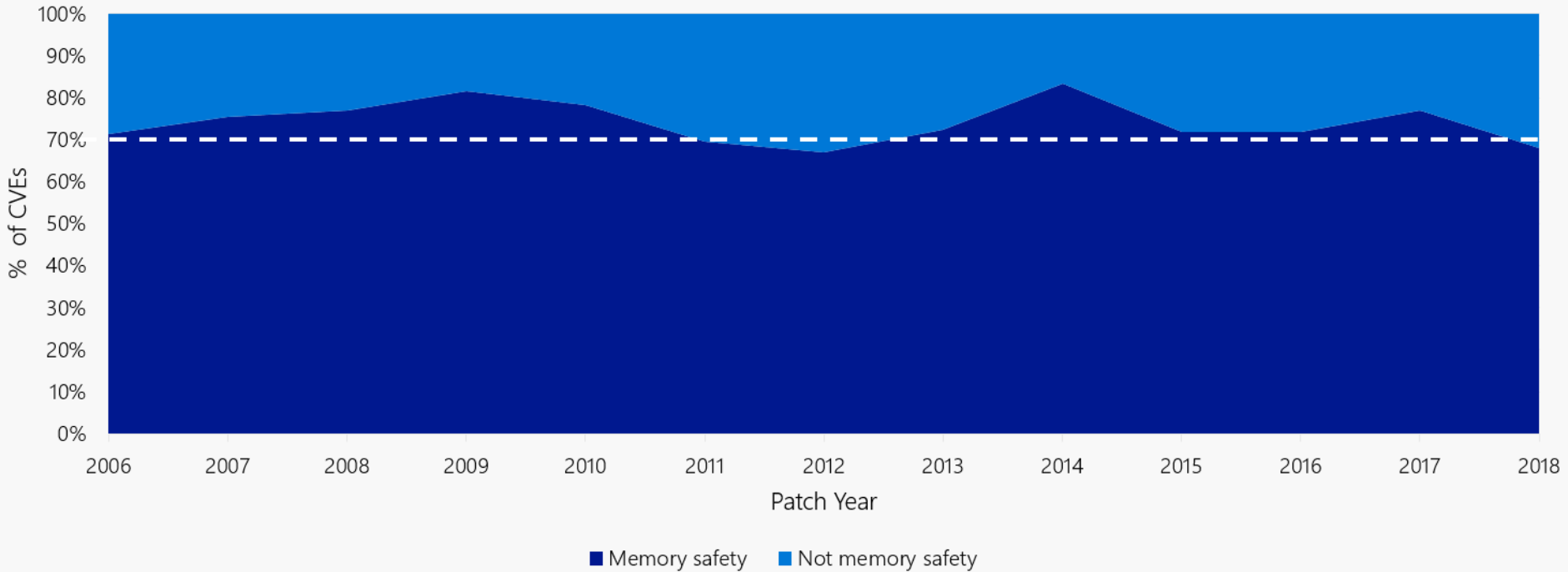
public enemy number 1

**Erik Poll**

Digital Security

Radboud University Nijmegen

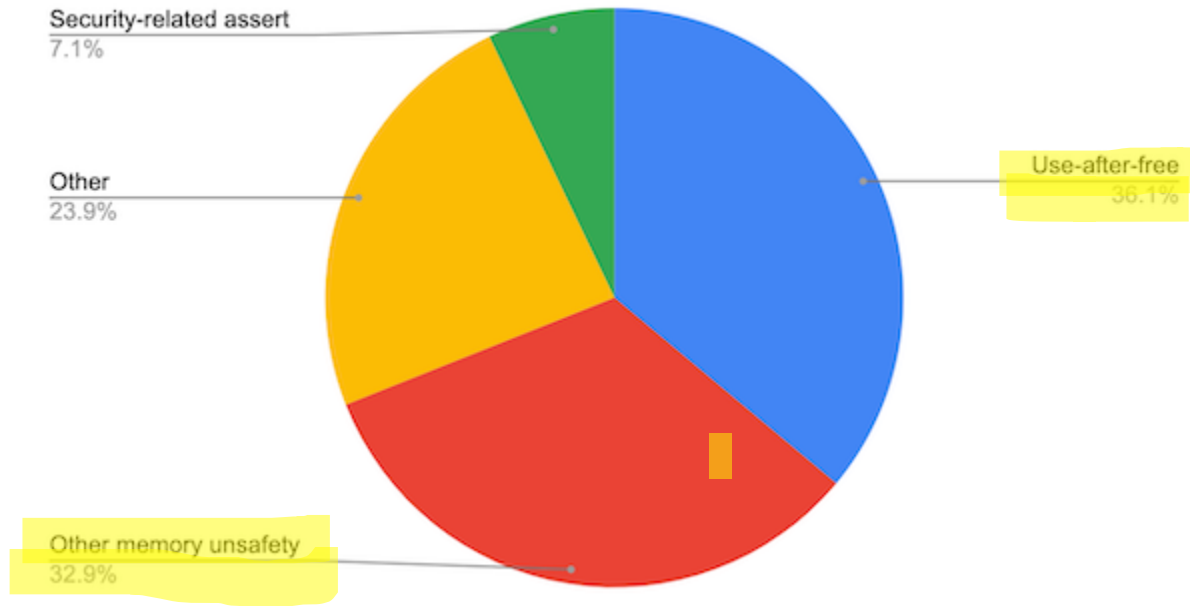
## Memory corruption bugs vs rest - Microsoft 2006-2018



[Source: <https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code> and “Trends, challenge, and shifts in software vulnerability mitigation”, presentation by Matt Miller at BlueHat IL 2019]

# Memory corruption bugs in Chromium project – since 2015

70% of high severity & critical security bugs are memory unsafety problems



[Source: <https://www.chromium.org/Home/chromium-security/memory-safety> ]

# Security in the development lifecycle



aka **SAST**  
(*Static* Application Security Testing)

aka **DAST**  
(*Dynamic* Application Security Testing)

# Finding & fixing memory corruption – next weeks



week 3: exercise  
**Static analysis**  
with **PREfast**

week4: group project  
**fuzzing afl**  
**memory sanitizers Asan, Msan**



# More structural prevention - later

safer programming languages

LangSec for safer parser code



# Overview (next 2 weeks)

1. How do memory corruption flaws work?
2. What can be the impact?
3. How can we spot such problems in C(++) code?

Tool-support for this

- SAST: **PREfast** individual / pair project
- DAST: **Fuzzing** group project

4. What can 'the platform' do about it?  
ie. the compiler, system libraries, hardware, OS, ..
5. What can the programmer do about it?

# Reading material

- **SoK article: ‘Eternal War in Memory’ S&P 2013**
  - Excl. Section VII.
  - This article is quite dense. You are not expected to be able to reproduce or remember all the discussion here. It’s good enough if you can follow the article, with a steady supply of coffee while googling if the terminology is not clear.
- **Chapter 3.1 & 3.2 in lecture notes on memory-safety**

We’ll revisit safe programming languages – incl. other safety features – and rest of Chapter 3 in later lecture

## Essence of the problem

Suppose in a C program you have an array of length 4

```
char buffer[4];
```

What happens if the statement below is executed?

```
buffer[4] = 'a';
```

We don't know!

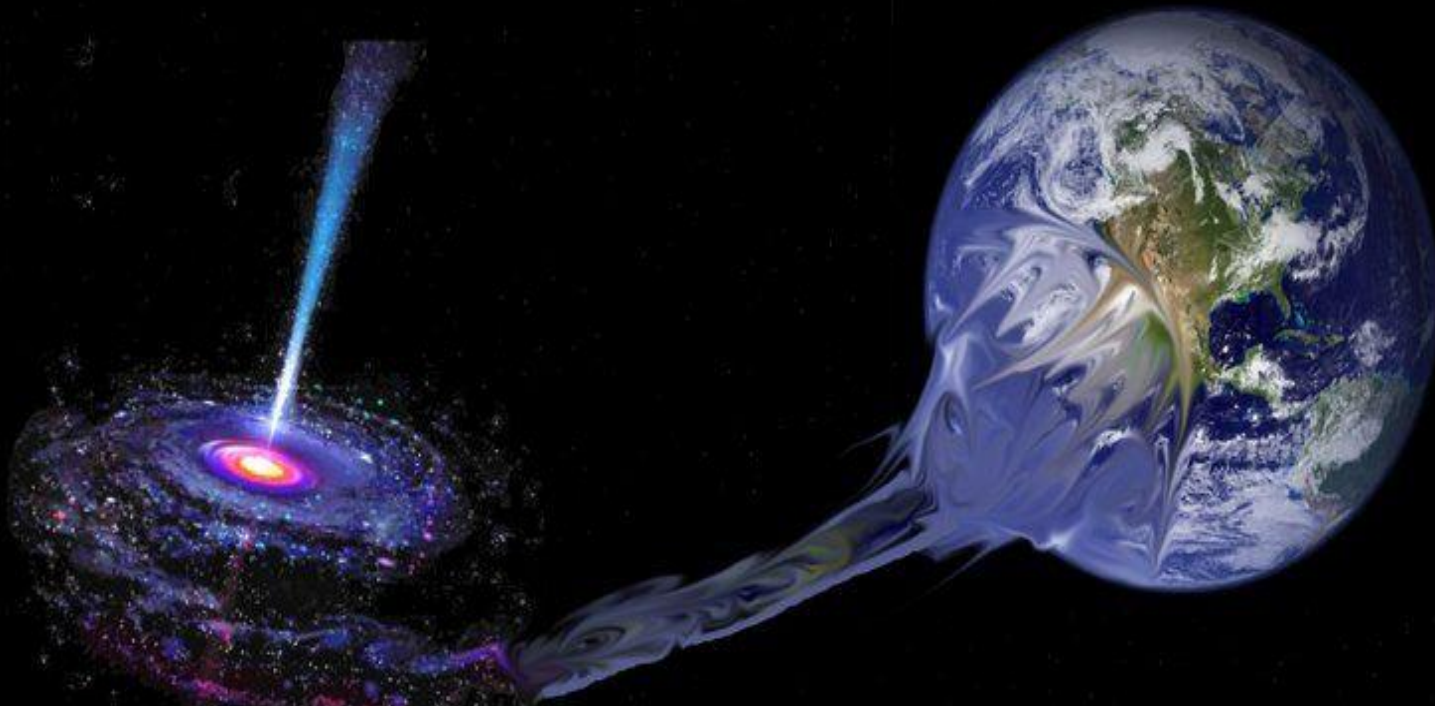
This is defined to be **undefined**

*ANYTHING* can happen

**UNDEFINED** behaviour: anything can happen



**UNDEFINED** behaviour: anything can happen





**UNDEFINED** behaviour: nothing may happen



# Anything attackers wants?

```
char buffer[4];  
buffer[4] = 'a';
```

If the attacker controls the value 'a'  
then anything that the attacker wants may happen ...

- If we are *lucky*: program crashes with **SEGMENTATION FAULT**
- If we are *unlucky*: program does not crash  
but silently allows **data corruption** or **remote code execution (RCE)**  
and we *won't* know till it's too late

# Nothing may happen

```
char buffer[4];  
buffer[4] = 'a';
```

A compiler could remove the assignment above,  
ie. *do nothing*

- Compilers actually do this (as part of optimisation) and this can cause security problems; examples later & in the lecture notes.

## Solution to this problem

- Check array bounds at runtime
  - Algol 60 proposed this back in 1960!
- Unfortunately, C and C++ have not adopted this solution.
  - *Why?*
  - For **EFFICIENCY**  
Regrettably, people often choose **performance** over **security**
- As a result, buffer overflows have been the no 1 security problem in software ever since
  - Check out CVEs mentioning buffer (or buffer%20overflow)  
<https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=buffer>
- Fortunately, Perl, Python, Java, C#, PHP, Javascript, and Visual Basic *do* check array bounds

## Tony Hoare on design principles of ALGOL 60



In his Turing Award lecture in 1980

“The first principle was *security*: ... every subscript was checked at run time against both the upper and the lower declared bounds of the array. Many years later we asked our customers whether they wished an option to switch off these checks in the interests of efficiency. Unanimously, they urged us not to - they knew how frequently subscript errors occur on production runs where failure to detect them could be disastrous.

I note with fear and horror that even in 1980, language designers and users have not learned this lesson. In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law.”

[C.A.R. Hoare, The Emperor’s Old Clothes, Communications of the ACM, 1980]

## More memory corruption problems

Errors with **pointers** and with **dynamic memory** (aka the heap)

- *Have you ever written a C(++) program that uses **pointers**?*
- *Have you ever had such a program crashing?*
  
- *Have you even written a C(++) program that uses **dynamic memory**, ie. **malloc()** and **free()**?*
- *Have you ever had such a program crashing?*

In C/C++, the programmer is responsible for **memory management** and this is very error-prone

- Technical term: C and C++ do not offer **memory-safety**  
(see lecture notes, §3.1-3.2)

# Spot all (potential) defects

```
1000 ...
1001 void f(){
1002     char* buf, buf1, buf24;
1003     buf = malloc(100);
1004     buf[0] = 'a';
    ...
98991     free(buf24);
98992     buf[0] = 'b';
    ...
999991     free(buf);
999992     buf[0] = 'c';
999993     buf1 = malloc(100);
999994     buf[0] = 'd';
999995 }
```

**null dereference**  
if malloc failed

**potential use-after-free**  
if buf & buf24 are **aliased**

**use-after-free; buf[0] points**  
to de-allocated memory

**memory leak; pointer buf1**  
to this memory is lost &  
memory is never freed

**use-after-free, but now buf[0]**  
may point to memory that has  
been re-allocated for buf1

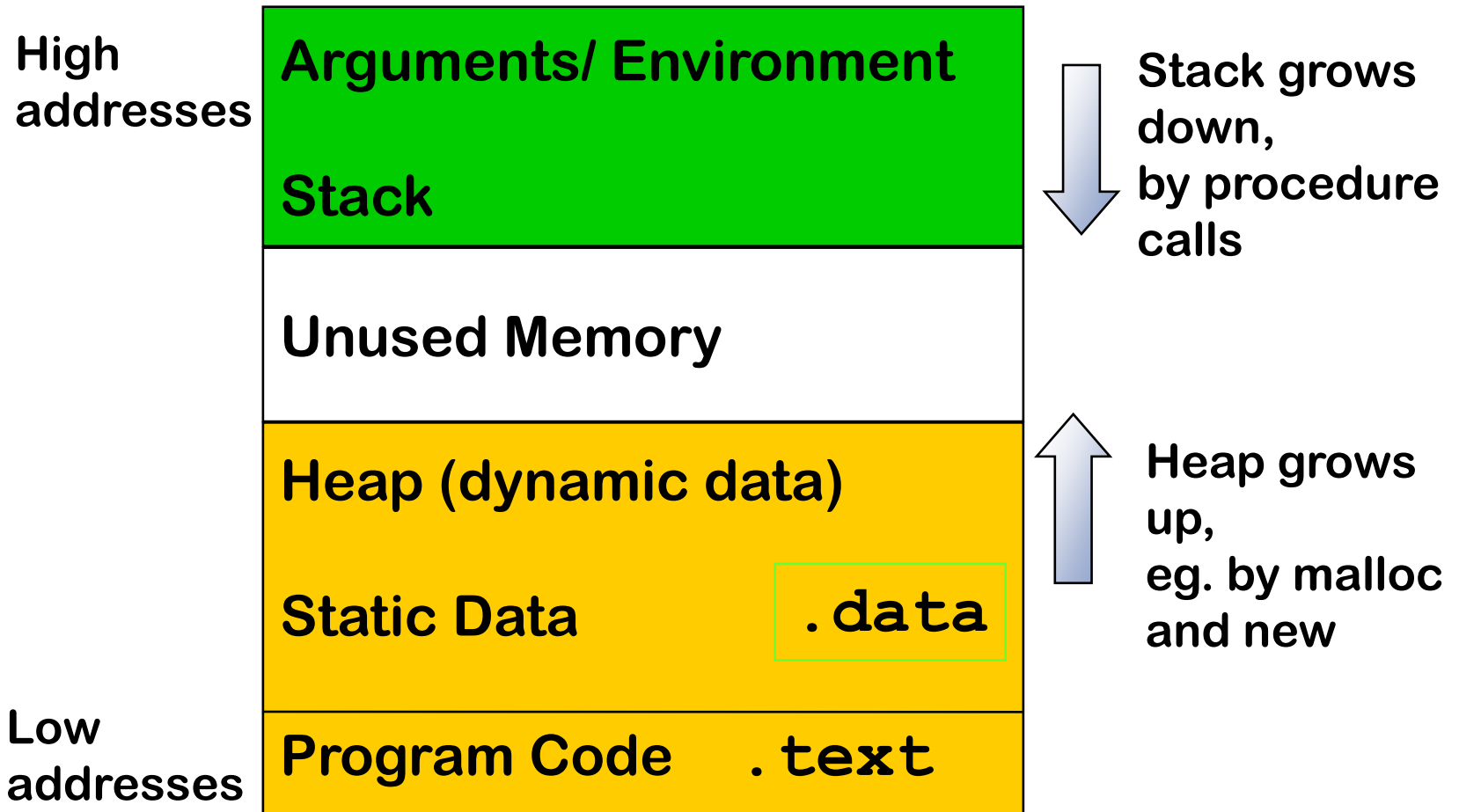
# Causes of memory corruption problems

- **Access outside array bounds** aka **buffer overflow**
  - **overread** or **overwrite**
    - overreads are not a corruption issue, but *confidentiality* issue
- **Pointer trouble:**
  - **buggy pointer arithmetic,**
  - **dereferencing null pointer,**
  - **using a dangling pointer aka stale pointer**
    - caused by e.g. **use-after-free** or **double-free**
- **Memory management problems:**
  - **Forgetting to check for failures in allocation**
  - **Forgetting to de-allocate, aka memory leaks**
    - not a corruption issue, but an *availability* issue
- **Other ways to break memory abstractions:** **missing null terminators, too many null terminators, type casts, type confusion, ...**

**Exploiting this**

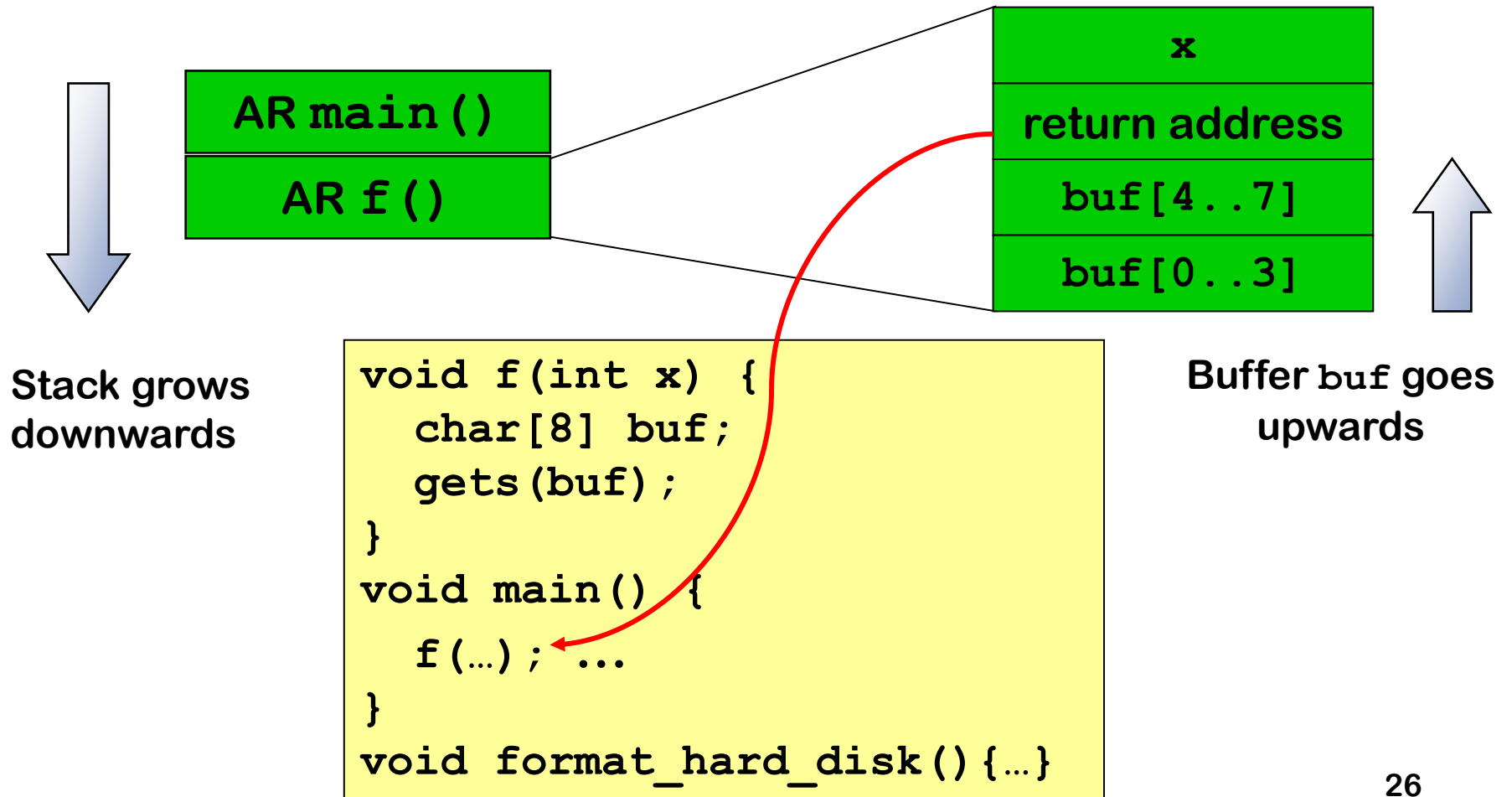


# Process memory layout



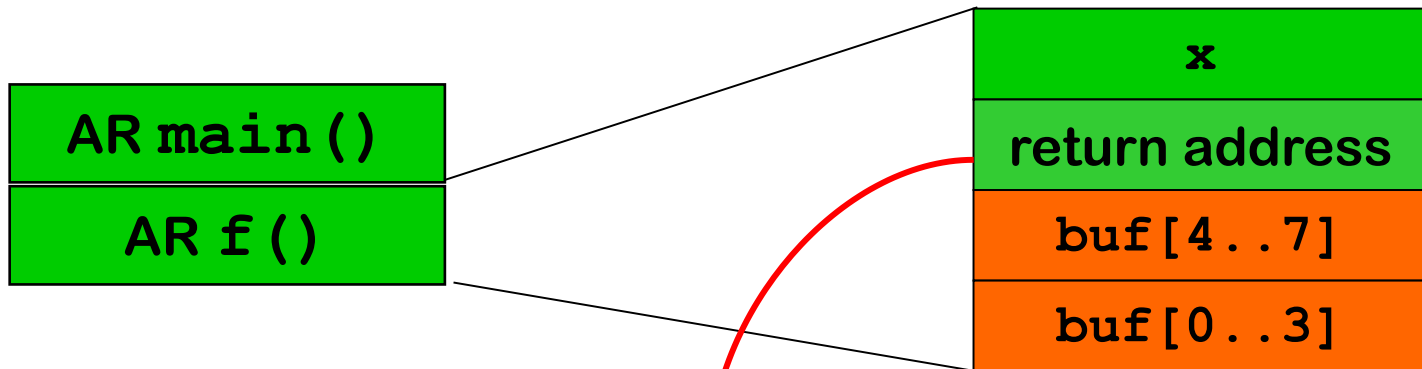
# Stack layout

The stack consists of **Activation Records** aka **stack frames**:



# Stack overflow attack - case 1

*What if gets () reads more than 8 bytes ?*

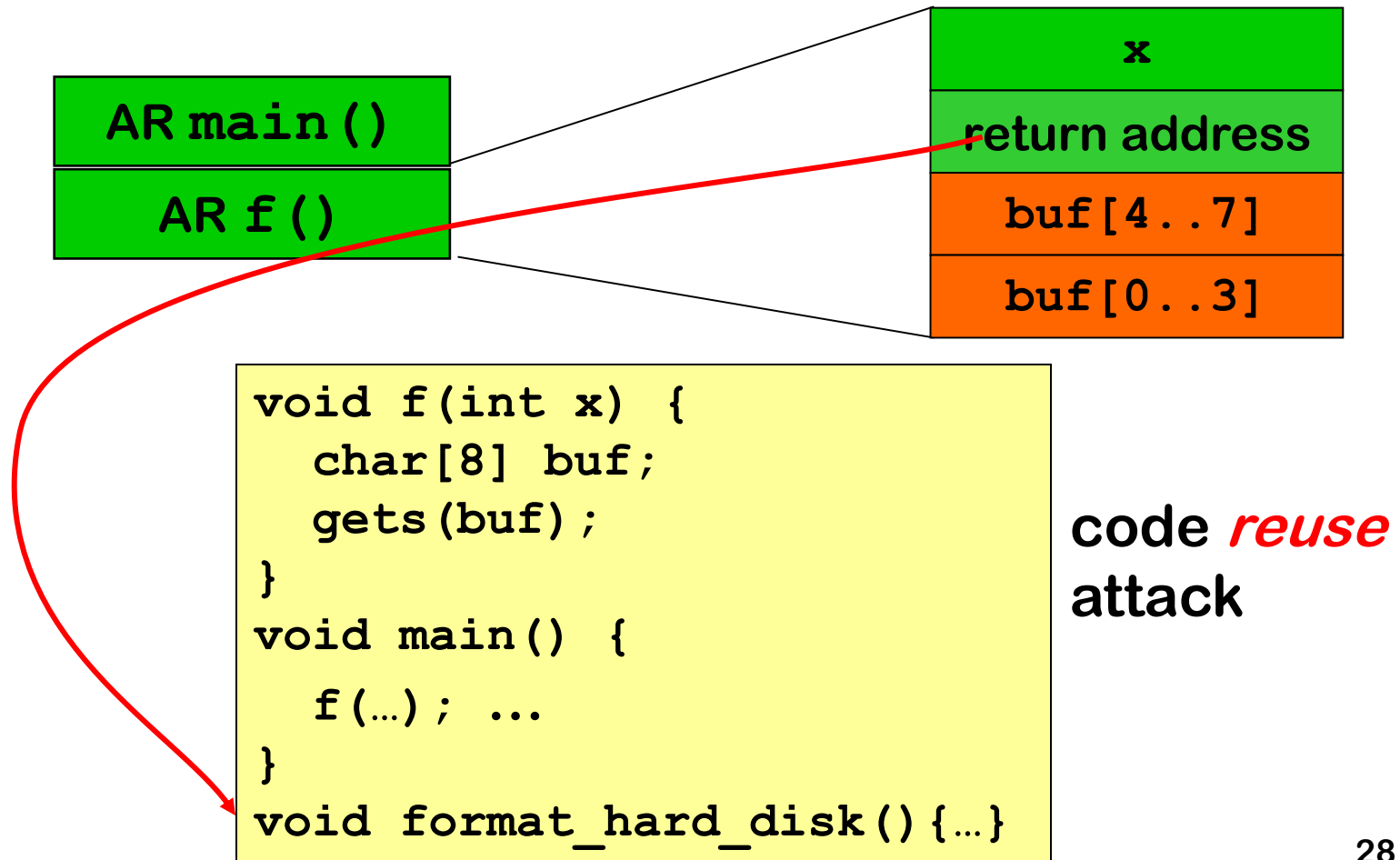


```
void f(int x) {  
    char[8] buf;  
    gets(buf);  
}  
void main() {  
    f(...);  
}  
void format_hard_disk() {...}
```

# Stack overflow attack - case 1

*What if gets () reads more than 8 bytes ?*

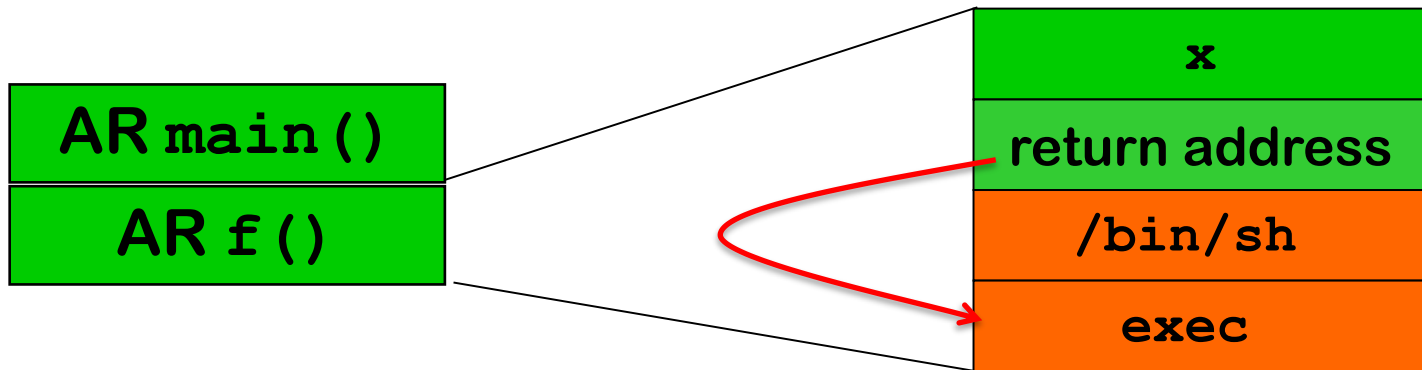
**Attacker can jump to arbitrary point in the code!**



## Stack overflow attack - case 2

*What if gets () reads more than 8 bytes ?*

Attackers can also jump to their own code (aka shell code)



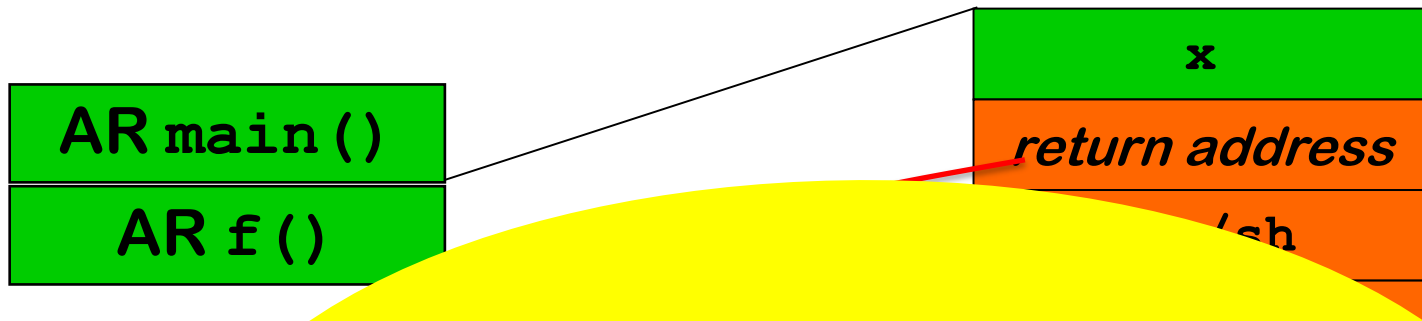
```
void f(int x) {
    char[8] buf;
    gets(buf);
}
void main() {
    f(...); ...
}
void format_hard_disk() {...}
```

code *injection*  
attack

## Stack overflow attack - case 2

*What if gets () reads more than 8 bytes ?*

Attacker can jump to his own code (aka shell code)



***never use gets !***

**gets has been removed from  
the C standard in 2011**

```
void  
f (...)  
}  
void format_hard_disk () {...}
```

## Code *injection* vs code *reuse*

Two types of attacks in these examples

(2) is a code *injection* attack

attackers inject their own shell code in some buffer  
and corrupt return address to point to this code

In the example, `exec('/bin/sh')`

This is the classic buffer overflow attack

[Smashing the stack for fun and profit, Aleph One, 1996]

(1) is a code *reuse* attack

attackers corrupt return address to point to existing code

In the example, `format_hard_disk`

Lots of details to get right!

- knowing precise location of return address and other data on stack, knowing address of code to jump to, ....

## What to attack? Corrupting the stack

```
void f(int x,  
      void(*error_handler)(int),  
      bool b) {  
    int diskquota = 200;  
    bool is_super_user = false;  
    char* filename = "/tmp/scratchpad";  
    char[8] username;  
    int j = 12;  
    ...  
}
```

function pointer

Suppose attacker can overflow `username`

This can corrupt the return address, but also other data on the stack:

`is_super_user`, `diskquota`, `filename`, `x`, `b`, `error_handler`

- But not `j`, unless the compiler chooses to allocate variables in a different order, which the compiler is free to do
- Corruption `function pointers` such as `error_handler` is particularly interesting! *Why?*



## What to attack? Corrupting data on the heap

```
struct BankAccount {  
    int  number;  
    char username[20];  
    int  balance;  
}
```

Suppose attacker can overflow `username`

This can corrupt other fields in the struct

- Which fields depends on the order of the fields in memory.

The compiler is free to choose this.

## What to attack? Corrupting *vtables on the heap*

C++ code uses **late binding** to resolve (so-called virtual) **method calls**

```
Rectangle r;
```

```
Circle c;
```

```
Shape s;
```

```
_surface_area = r.area() + c.area() + s.area();
```

Which code to execute for `s.area()` is determined at runtime.

To do this, a **table of function pointers**, the **vtable**, is maintained that tells which code to execute for each method

This provides many function pointers for attackers to mess with!

# Recurring theme in attacks: **breaking abstractions**

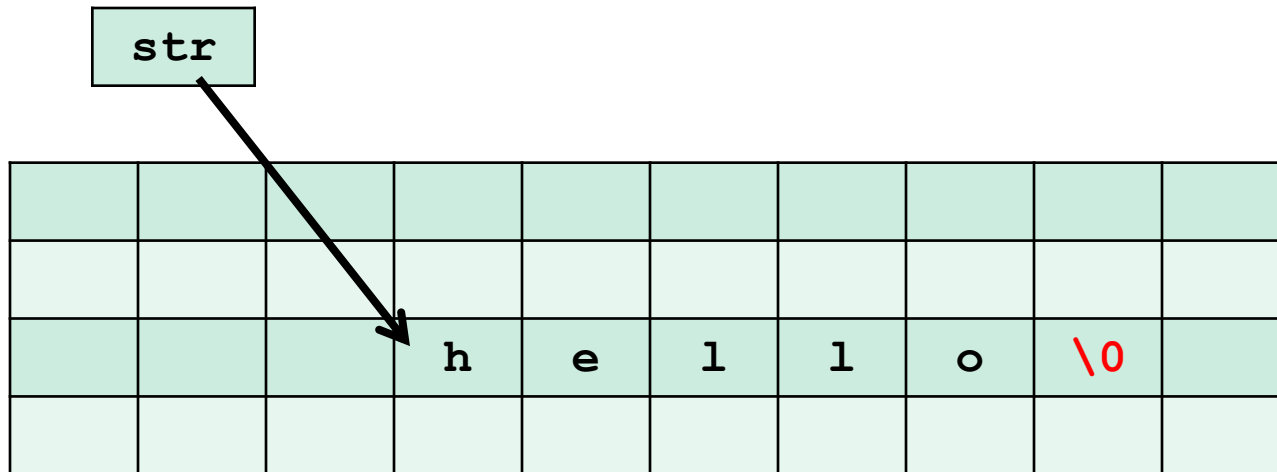


# Spotting the problem

## Reminder: C chars & strings

- A char in C is always exactly one byte
- A string is a **sequence of chars terminated by a NULL byte**
- String variables are **pointers** of type `char*`

```
char* str = "hello"; // a string str
```



Here `strlen(str)` will be 5

## Example: gets

```
char buf[20];  
gets(buf); // read user input until  
           // first EoL or EOF character
```

- *Never use gets*
  - `gets` has been removed from the C library so this code will no longer compile
- Use `fgets(buf, size, file)` instead

## Example: strcpy

```
char dest[20];  
strcpy(dest, src); // copies string src to dest
```

- strcpy assumes that 1 dest is long enough  
and src is null-terminated
- Use strncpy(dest, src, size) instead

Beware of difference between sizeof and strlen

```
sizeof(dest) = 20 // size of an array
```

```
strlen(dest) = number of chars up to first null byte  
// length of a string
```

## Spot the defect!

```
char buf[20];
char prefix[] = "http://";
char* path;
...
strcpy(buf, prefix);
    // copies the string prefix to buf
strncat(buf, path, sizeof(buf));
    // concatenates path to the string buf
```



## Spot the defect! (1)

```
char buf[20];
char prefix[] = "http://";
char* path;
...
strcpy(buf, prefix);
// copies the string prefix to buf
strncat(buf, path, sizeof(buf));
// concatenates path to the string buf
```

strncat's 3rd parameter is number of chars to copy, not the buffer size

So this should be `sizeof(buf) - 7`

# Better libraries?

Keeping track of the space left in buffers when using `strncpy` and `strncpy` is error-prone. Better alternatives:

- `strncpy`(`dst`, `src`, `size`) and `strlcat`(`dst`, `src`, `size`)  
Here `size` is the size of destination array `dst`, not the maximum length copied. These are consistently used in OpenBSD.
- Functions in Microsoft's `Strsafe.h` also always takes destination size as argument. Moreover, they guarantee null-termination.

Other alternatives:

- `glib.h` provides `Gstring` type for dynamically growing null-terminated strings in C
- **C++ string objects** are less error-prone than C strings
  - but `data()` and `c-str()` return a C string, ie. a `char*`, and result of `data()` is not always null-terminated on all platforms.

## Spot the defect! (2)

```
char src[9];
char dest[9];

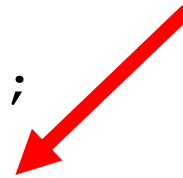
char* base_url = "www.ru.nl";
strncpy(src, base_url, 9);
    // copies base_url to src
strcpy(dest, src);
    // copies src to dest
```

## Spot the defect! (2)

```
char src[9];  
char dest[9];
```

base\_url is 10 chars long, incl.  
its null terminator, so src will not  
be null-terminated

```
char* base_url = "www.ru.nl";  
strncpy(src, base_url, 9);  
    // copies base_url to src  
strcpy(dest, src);  
    // copies src to dest
```



## Spot the defect! (2)

```
char src[9];  
char dest[9];
```

base\_url is 10 chars long, incl.  
its null terminator, so src will not  
be null-terminated

```
char* base_url = "www.ru.nl";  
strncpy(src, base_url, 9);  
    // copies base_url to src  
strcpy(dest, src);  
    // copies src to dest
```

so strcpy will overrun the buffer dest,  
because src is not null-terminated

## Example: strcpy and strncpy

Don't replace

```
strcpy(dest, src)
```

with

```
strncpy(dest, src, sizeof(dest))
```

but with

```
strncpy(dest, src, sizeof(dest)-1)
```

```
dst[sizeof(dest)-1] = '\0';
```

if you want dest to be null-terminated!

**NB:** a **strongly typed programming language** would *guarantee* that strings are always null-terminated, without the programmer having to worry about this...

## Spot the defect! (3)

```
char *buf;  
int len;  
...
```

```
buf = malloc(MAX(len,1024)); // allocate buffer  
read(fd,buf,len); // read len bytes into buf
```



What happens if len is negative?

The length parameter of read is **unsigned!**  
So negative len is interpreted as a big positive one!  
**AAAAAAAAARGH!**

(At the exam, you're not expected to remember  
that read treats its 3<sup>rd</sup> argument as an unsigned int)

## Spot the defect! (3)

```
char *buf;  
int len;  
...  
  
if (len < 0)  
    {error ("negative length"); return; }  
buf = malloc(MAX(len,1024));  
read(fd,buf,len);
```

Note that `buf` is not guaranteed to be null-terminated;  
we ignore this for now.

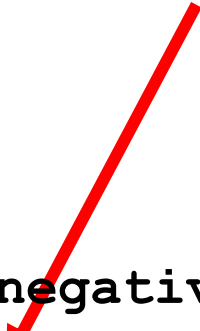


## Spot the defect! (3)

```
char *buf;  
int  len;  
...
```

What if the malloc() fails,  
because we ran out of memory ?

```
if (len < 0)  
    {error ("negative length"); return; }  
buf = malloc(MAX(len,1024));  
read(fd,buf,len);
```



## Spot the defect! (3)

```
char *buf;
int len;
...

if (len < 0)
    {error ("negative length"); return; }
buf = malloc(MAX(len,1024));
if (buf==NULL) { exit(-1);}
    // or something a bit more graceful
read(fd,buf,len);
```

## Better still

```
char *buf;
int len;
...

if (len < 0)
    {error ("negative length"); return; }
buf = calloc(MAX(len,1024));
    //to initialise allocate memory to 0
if (buf==NULL) { exit(-1);}
    // or something a bit more graceful
read(fd,buf,len);
```

## Spot the defect!

```
#define MAX_BUF 256

void BadCode (char* input)
{   short len;
    char buf[MAX_BUF];

    len = strlen(input);

    if (len < MAX_BUF) strcpy(buf,input);
}
```

## Spot the defect!

```
#define MAX_BUF 256
```

```
void BadCode (char* input)
```

```
{  short len;
```

```
  char buf[MAX_BUF];
```

```
  len = strlen(input);
```

```
  if (len < MAX_BUF) strcpy(buf, input);
```

```
}
```

What if `in` is longer than 32K ?

len may be a negative number,  
due to **integer overflow**



hence: potential  
**buffer overflow**




The **integer overflow** is the root problem,  
the (heap) **buffer overflow** it causes makes it exploitable

See [https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=integer+overflow](https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=<u>integer+overflow</u>)

## Spot the defect!

```
bool CopyStructs(InputFile* f, long count)
{
    structs = new Structs[count];
    for (long i = 0; i < count; i++)
        { if !(ReadFromFile(f, &structs[i]))
            break;
        }
}
```



effectively does a  
`malloc(count*sizeof(type))`  
which may cause **integer overflow**

And this integer overflow can lead to a (heap) **buffer overflow**  
Since 2005 Visual Studio C++ compiler adds check to prevent this

## NB absence of language-level security

In a **safer** programming language than C/C++, the programmer would not have to worry about

- **writing past array bounds**  
(because you'd get an `IndexOutOfBoundsException` instead)
- **strings not having a null terminator**
- **implicit conversions from signed to unsigned integers**  
(because the type system/compiler would forbid this or warn)
- **malloc possibly returning null**  
(because you'd get an `OutOfMemoryException` instead)
- **malloc not initialising memory**  
(because language could always ensure default initialisation)
- **integer overflow**  
(because you'd get an `IntegerOverflowException` instead)
- ...

## Spot the defect!

```
1. void* f(int start) {
2.     if (start+100 < start) return SOME_ERROR_CODE;
3.         // checks for overflow
4.     for (int i=start; i < start+100; i++) {
5.         . . . // i will not overflow
6.     } }
```

Integer overflow is **UNDEFINED** behaviour! This means

- You cannot assume that overflow produces a negative number; so line 2 is *not* a good check for integer overflow.
- Worse still, if integer overflow occurs, behaviour is undefined:
  - So compiled code can do *anything* if `start+100` overflows
  - So compiled code can do *nothing* if `start+100` overflows
  - This means the compiler can *remove* line 2

Modern C compilers are clever enough to know that `x+100 < x` is always false, and optimise code accordingly



## Spot the defect! (code from Linux kernel)

```
1. unsigned int tun_chr_poll( struct file *file,  
2.                             poll_table *wait)  
3. { ...  
4.     struct sock *sk = tun->sk; // take sk field of tun  
5.     if (!tun) return POLLERR; // return if tun is NULL  
6.     ...  
7. }
```

## Spot the defect! (code from Linux kernel)

```
1. unsigned int tun_chr_poll( struct file *file,  
2.                             poll_table *wait)  
3. { ...  
4.     struct sock *sk = tun->sk; // take sk field of tun  
5.     if (!tun) return POLLERR; // return if tun is NULL  
6.     ...  
7. }
```

If `tun` is a null pointer, then `tun->sk` is **UNDEFINED**

What this function does when `tun` is null is undefined:

**ANYTHING** may happen then.

So compiler **can remove line 5**: the behaviour when `tun` is `NULL` is undefined anyway, so this check is 'redundant'.

Standard compilers (gcc, clang) do this 'optimisation' !

This is code from the Linux kernel where removing line 5 led to a security vulnerability [CVE-2009-1897]

## Spot the defect! (code from Windows kernel)

```
// TCHAR is 1 byte ASCII or multiple byte UNICODE
#ifdef UNICODE
# define TCHAR wchar_t           wide UNICODE character, > 1 byte
# define _tprintf _wprintf      print-function for wide character strings
#else
# define TCHAR char              ASCII character, 1 byte
# define _tprintf _printf       print-function for ASCII character strings
#endif
```

```
TCHAR buf[MAX_SIZE];
_tprintf(buf, sizeof(buf), input);
```

← **sizeof(buf) is the size in *bytes*, but this parameter should be the number of *characters***

Switch from ASCII to UNICODE caused lots of buffer overflows

## Spot the defect!

```
#include <stdio.h>

int main(int argc, char* argv[])
{  if (argc > 1)
    printf(argv[1]);
   return 0;
}
```