

Software Security

# Buffer Overflows

public enemy number 1

**Erik Poll**

Digital Security

Radboud University Nijmegen

# Overview

1. How do buffer overflows work?

Or, more generally, **memory corruption** errors

2. How can we spot such problems in C(++) code?

Next week: tool-support for this

Incl. static analysis tool for first exercise.

3. What can 'the platform' do about it?

– ie. the compiler, system libraries, hardware, OS

4. Next week: What can the programmer do about it?

# Reading material

- **Runtime Countermeasures for Code Injection Attacks Against C and C++ Programs**

by Yves Younan, Wouter Joosen, and Frank Piessens

Except Section 3 & 4.6 and all tables

- **Safe languages also discussed in Chapter 3.1 & 3.2 in lecture notes**
  - We'll revisit safe programming languages & 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';
```

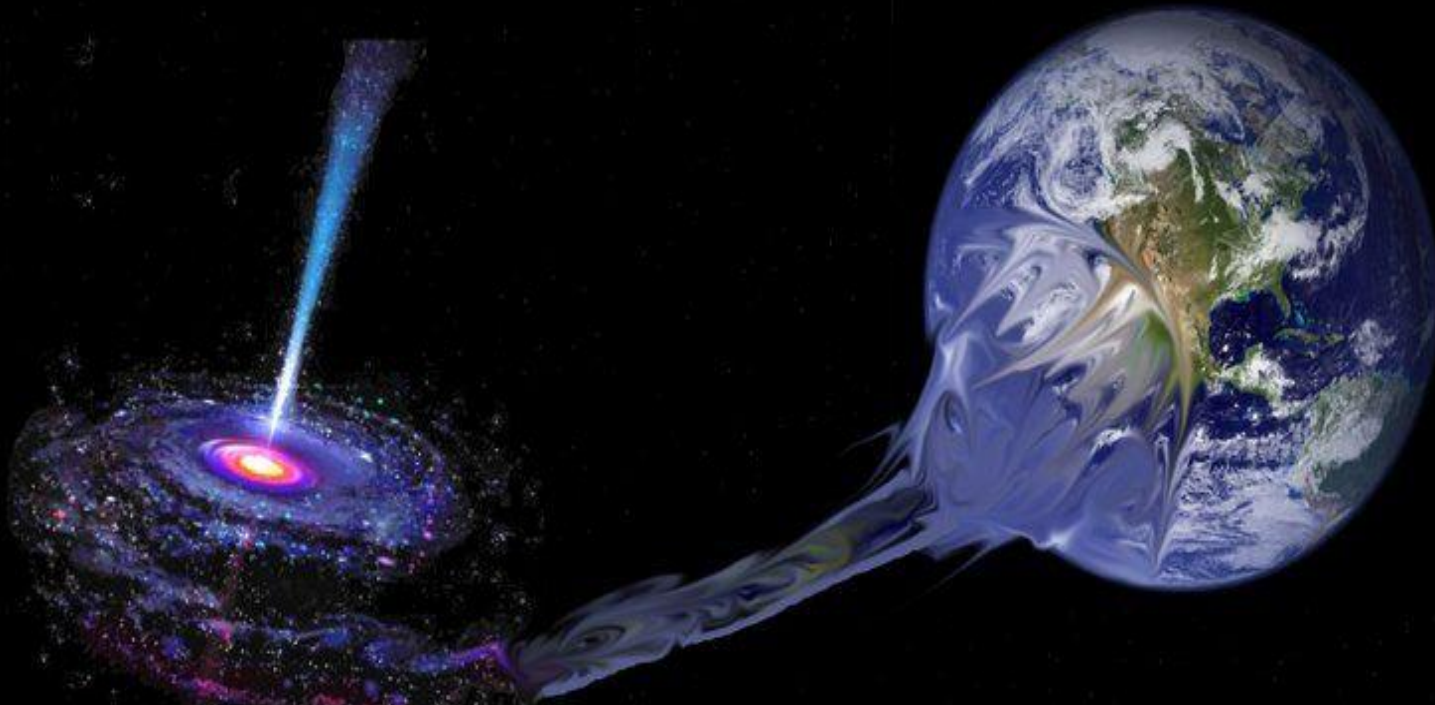
This is **undefined**

***ANYTHING*** can happen

**UNDEFINED** behaviour: anything can happen



**UNDEFINED** behaviour: anything can happen



## 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';
```

If an attacker can trigger this and control the value 'a',  
*anything that the attacker* wants may happen

- If you are *lucky*, you only get a **SEGMENTATION FAULT**
  - and you'll know that something went wrong
- If you are *unlucky*, an attacker will take over a machine with a **remote code execution (RCE)**
  - and *you won't know*
- Not only *writing* outside array bounds is dangerous, but so is *reading* (remember Heartbleed)

## 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.
- 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]

# The buffer overflow problem

- The most common security problem in (machine code compiled from) **C** and **C++**
  - ever since the first Morris Worm in 1988
- Sometime attackers can create **DoS**, by crashing systems; often they get **full control** via **Remote Code Execution (RCE)**
- Check out **CVEs** mentioning buffer (or buffer%20overflow)  
<https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=buffer>
- Ongoing arms race of attacks & defences: attacks are getting cleverer, defeating ever better countermeasures

## Other memory corruption errors

Other memory bugs in C/C++ code, besides array access: errors with **pointers** and with **dynamic memory (the heap)**

*Who here has ever written C(++) programs with pointers?*

*Who ever had such a program crashing?*

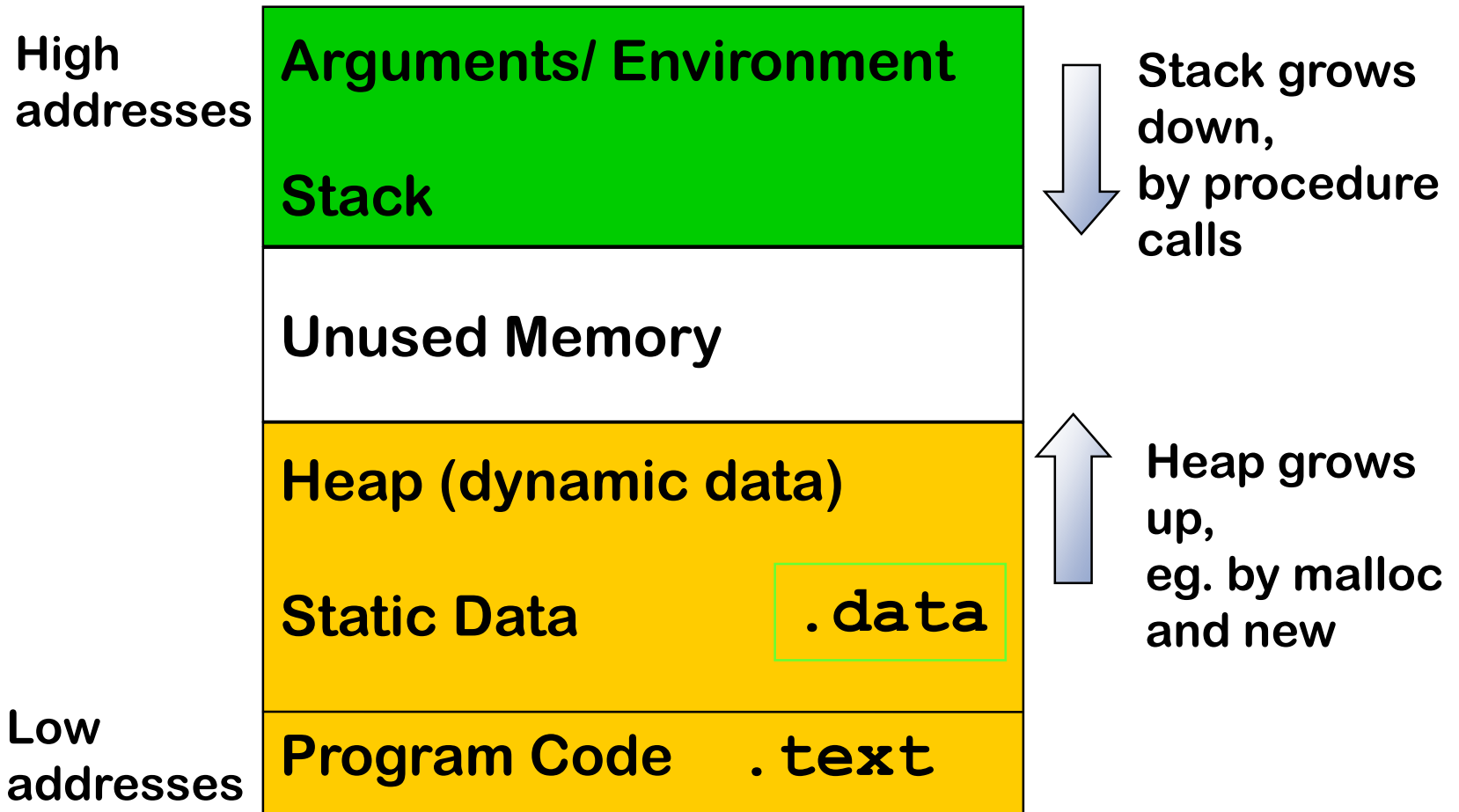
*Or using dynamic memory, ie. using **malloc** & **free**?*

*Who 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 on language-based security, §3.1-3.2)
- Typical problems:
  - **dereferencing null, dangling pointers, use-after-free, double-free, forgotten de-allocation (memory leaks), failed allocation, flawed pointer arithmetic**

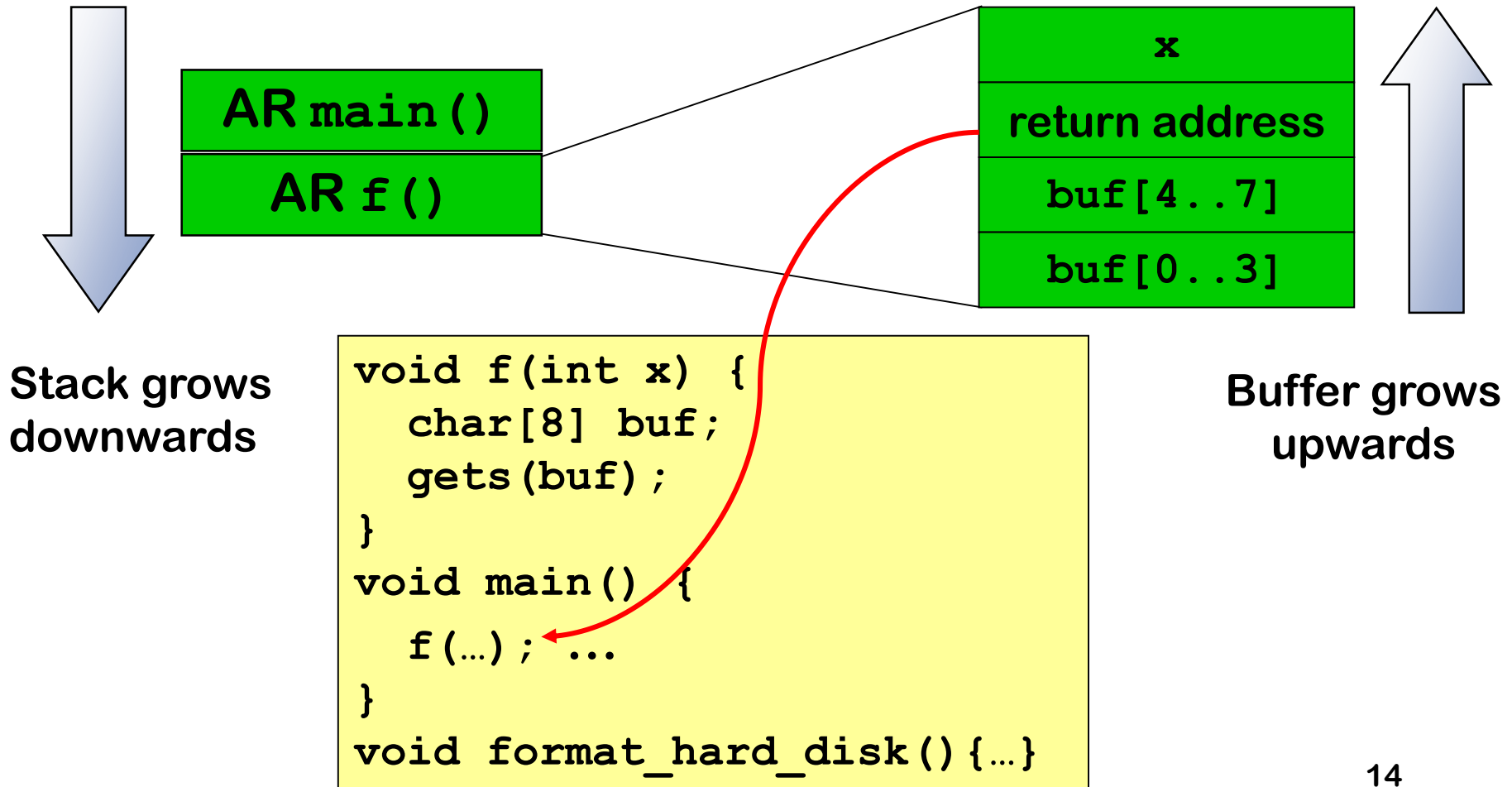
**How does classic buffer overflow work?  
aka smashing the stack**

# Process memory layout



# Stack layout

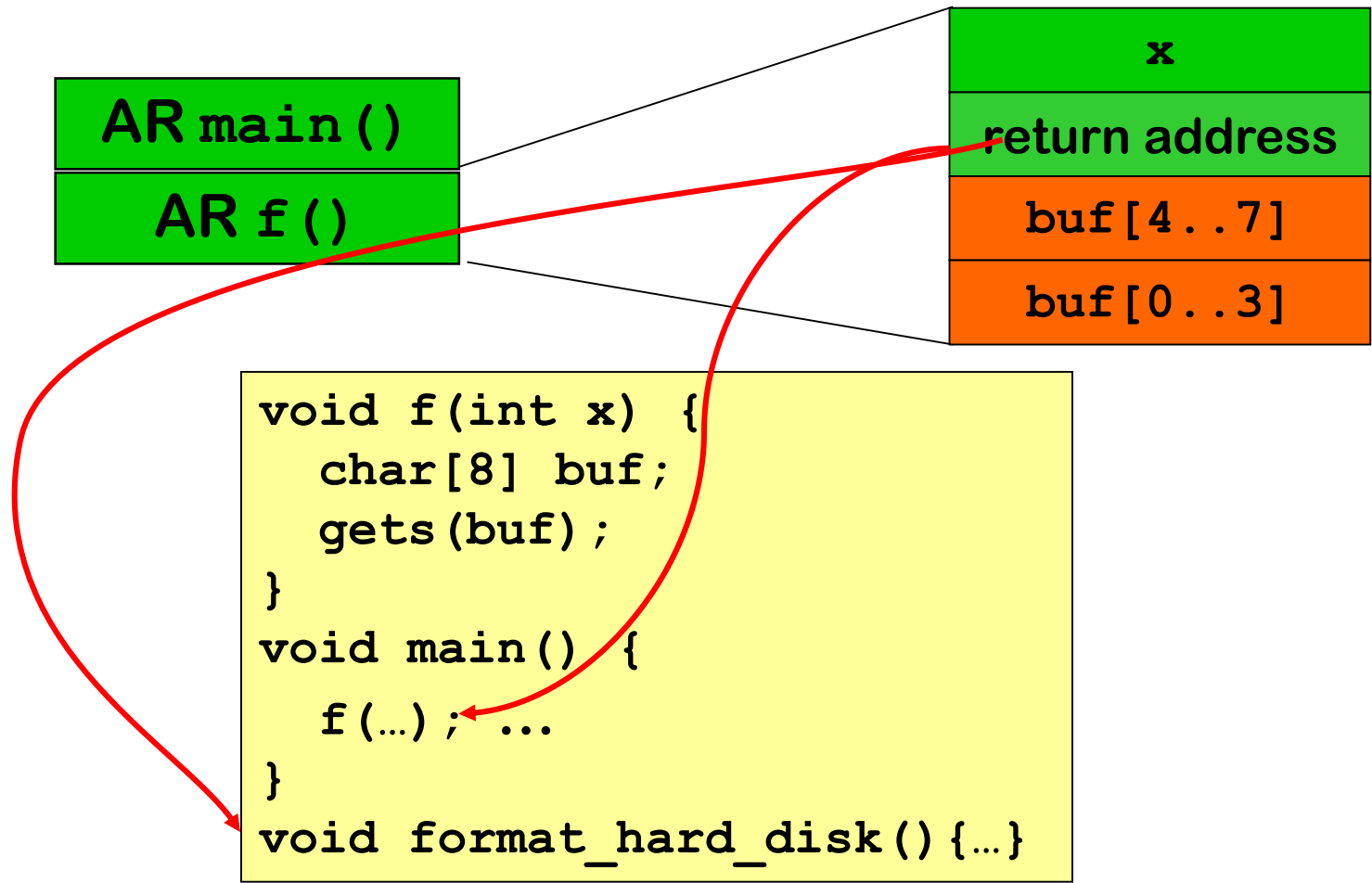
The stack consists of **Activation Records**:



# Stack overflow attack (1)

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

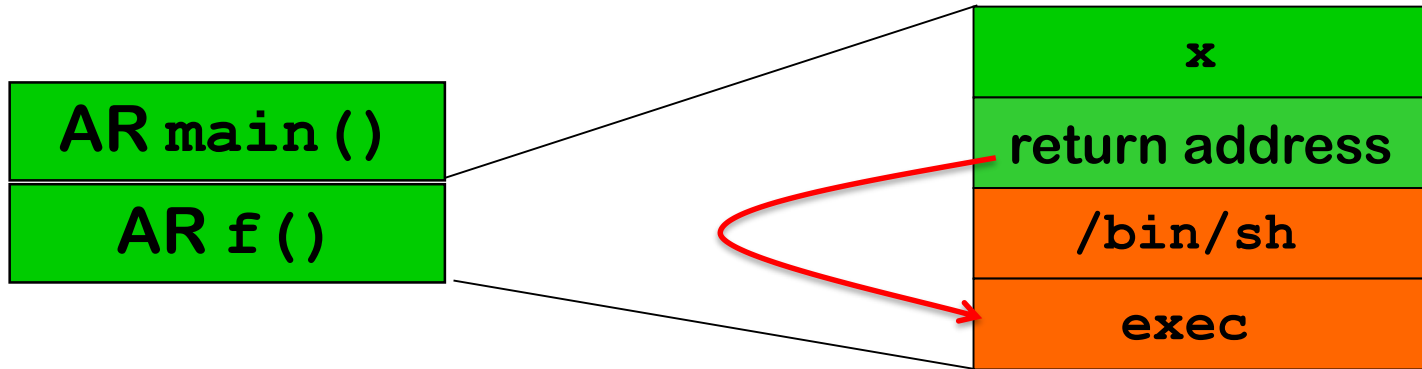
**Attacker can jump to abitrary point in the code!**



## Stack overflow attack (2)

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

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



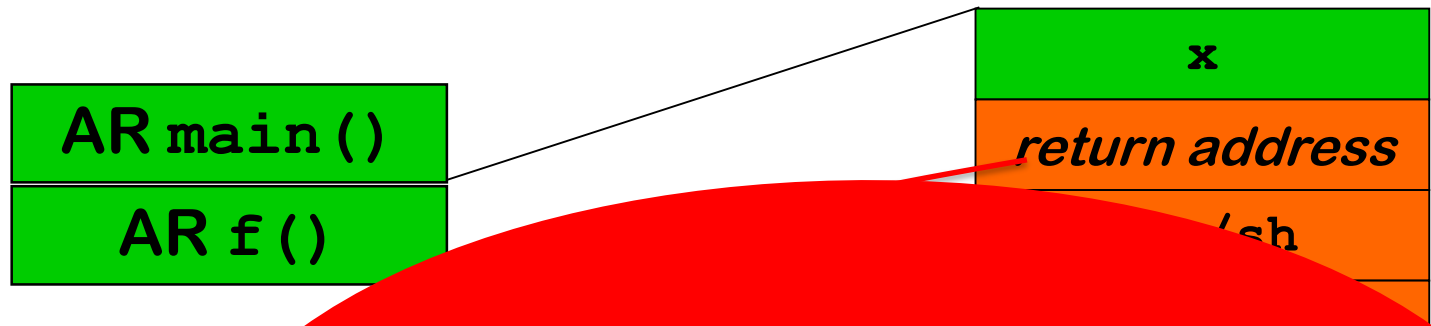
```
void f(int x) {  
    char[8] buf;  
    gets(buf);  
}  
void main() {  
    f(...); ...  
}  
void format_hard_disk() {...}
```



# Stack overflow attack (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 attacks

The two attack scenarios in these examples

(2) is a code *injection* attack

attacker inserts his own shell code in a buffer and corrupts return addresss 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

attacker corrupts 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, ....

## Other attacks using memory errors

Besides messing the return address,  
other ways to exploit buffer overflows & pointer bugs:

- corrupt some data
- illegally read some (confidential) data

This data can be allocated

- on the **stack**
- on the **heap**

## What to attack? More fun on the stack

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

    ...
}
```

Suppose the attacker can overflow `username`

In addition to corrupting the return address, she might corrupt

- **pointers**, eg `filename`
- **other data on the stack**, eg `is_super_user`, `diskquota`
- **function pointers**, eg `error_handler`

But not `j`, unless the compiler chooses to allocate variables in a different order, which the compiler is free to do.

## What to attack? Fun on the heap

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

Overrunning `username` can corrupt other fields in the struct.

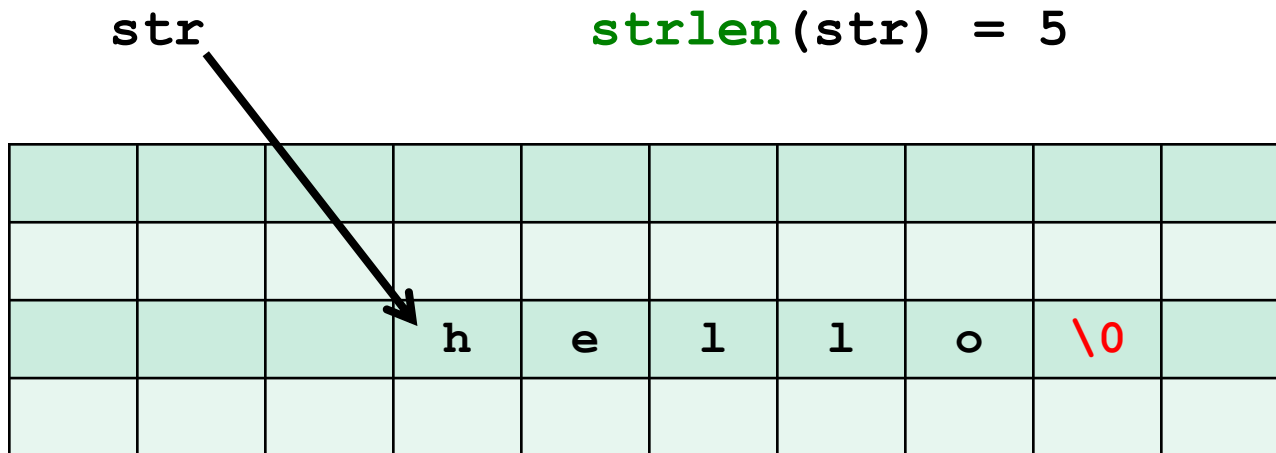
Which field(s) can be corrupted depends on the order of the fields in memory, which the compiler is free to choose.

# Spotting the problem

## Reminder: C strings

- A string is a **sequence of bytes terminated by a NULL byte**
- String variables are **pointers**

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



## Example: gets

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

- *Never* use gets
- Use fgets(buf, size, file) instead



## Example: strcpy

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

- strcpy assumes dest is long enough ,  
and assumes 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://";  
...  
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://";  
...  
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`

## Spot the defect! (2)

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

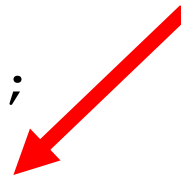
char* base_url = "www.ru.nl";
strncpy(src, base_url, 9);
    // copies base_url to src
strcpy(dest, src);
    // copies src to dest
```

## 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

## 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 dest should 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
```



## 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` system call is unsigned!  
So negative `len` is interpreted as a big positive one!

## Spot the defect! (3)

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

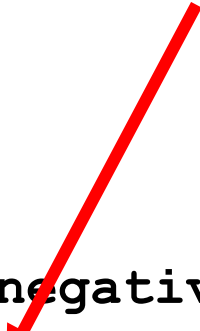
A remaining problem may be that `buf` is not null-terminated;  
we ignore this for now.

## Spot the defect! (3)

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

What if the malloc() fails?  
(because we are 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);
```

## NB absence of language-level security

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

- **writing past array bounds**  
(because you'd get an `IndexOutOfBoundsException` instead)
- **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!

```
#define MAX_BUF 256

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

    len = strlen(in);

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

## Spot the defect!

```
#define MAX_BUF 256
```

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

    len = strlen(in);

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

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>



## 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

## Spot the defect!

```
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 code does if `tun` is null is undefined:

**ANYTHING** may happen then.

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

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

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

## Spot the defect!

```
1. void* f(int start)
2.     if (start+100 < start) return SOME_ERROR;
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

- We cannot assume that overflow produces a negative number; so line 2 is not a good check for integer overflow.
- Worse still, if overflow occurs, behaviour is undefined, and *ANY* compilation is ok
  - so the compiled code is allowed to do *anything* in case `start+100` overflows

This also means compiler may *assume* that `start+100 < start` is *always false* (as it is always false when `start+100` does *not* overflow, at any behaviour is ok when it does overflow), and *remove* line 2

## Spot the defect!

```
// TCHAR is 1 byte ASCII or multiple byte UNICODE
#ifdef UNICODE
# define TCHAR wchar_t
# define _sntprintf _snwprintf
#else
# define TCHAR char
# define _sntprintf _snprintf
#endif

TCHAR buf[MAX_SIZE];
_sntprintf(buf, sizeof(buf), "%s\n", input);
```

*sizeof(buf) is the size in bytes,  
but this parameter gives the number  
of characters that will be printed*

The CodeRed worm exploited such an mismatch.

Lots of code written under the assumption that characters are 1 byte contained many buffer overflows after move from ASCII to Unicode

[slide from presentation by Jon Pincus]

## Spot the defect!

```
#include <stdio.h>

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

This program is vulnerable to **format string attacks**, where calling the program with strings containing special characters can result in a buffer overflow attack.

# Format string attacks

New type of memory corruption invented/discovered in 2000

- Strings can contain special characters, eg `%s` in  

```
printf("Cannot find file %s", filename);
```

  
Such strings are called **format strings**
- What happens if we execute the code below?  

```
printf("Cannot find file %s");
```
- What can happen if we execute  

```
printf(string)
```

  
where `string` is user-supplied?  
Esp. if it contains special characters, eg `%s, %x, %n, %hn`?



# Preventing format string attacks

- Always replace `printf(str)`  
with `printf("%s", str)`
- **Compiler** or **static analysis tool** could warn if the number of arguments does not match the format string  
eg in `printf ("x is %i and y is %i", x);`

Eg gcc has (far too many?) command line options for this:

```
-Wformat -Wformat-no-literal -Wformat-security ...
```

If the format string is not a compile-time constant, we cannot decide this at compile time, so compiler has to give false positives or false negatives

*See <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=format+string> to see how common format strings still are*



## Recap: buffer overflows

- Buffer overflow is #1 weakness in C and C++ programs
  - because these language are **not memory-safe**
- Tricky to spot
- Typical cause: programming with **arrays, pointers, and strings**
  - esp. **library functions for null-terminated strings**
- Related attacks
  - **Format string attack**: another way of corrupting stack
  - **Integer overflows**: often a stepping stone to getting a buffer to overflows
    - but just the integer overflow can already have a security impact; eg think of banking software

# Platform-level defences

# Platform-level defenses

Defensive measures that the 'platform', ie. compiler, hardware, OS can take, without programmer having to know

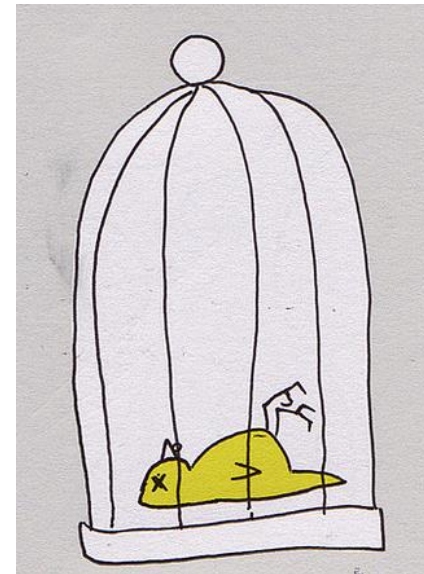
1. stack canaries
  2. non-executable memory (NX, W<sup>X</sup>)
  3. address space layout randomization (ASLR)
  4. control-flow integrity (CFI)
  5. bounds checkers / fat pointers
  6. pointer encryption
  7. instruction set randomisation
  8. execution-aware memory protection
- } now standard on many platforms

These defenses may have overhead (esp. wrt. speed)

*History shows that all new defenses are eventually defeated...*

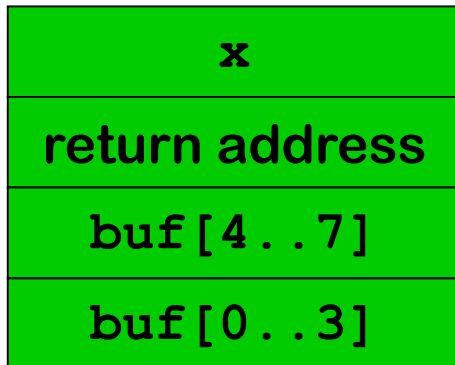
# 1. stack canaries

- A dummy value - **stack canary or cookie** - is written on the stack in front of the return address and checked when function returns
- A careless stack overflow will overwrite the canary, which can then be detected
- introduced in as **StackGuard** in gcc

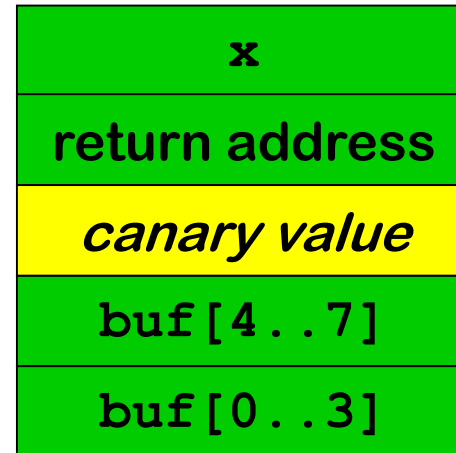


# stack canaries

Stack without canary



Stack with canary

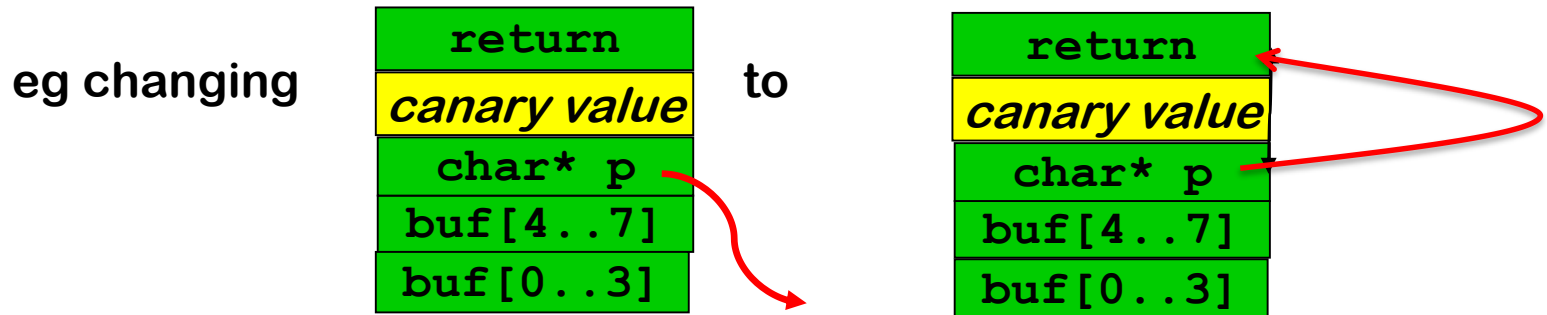


## Further improvements

- **More variation in canary values:** eg not a fixed values hardcoded in binary but a random values chosen for each execution
- Better still, **XOR the return address into the canary value**
- **Include a null byte in the canary value**, because C string functions cannot write nulls inside strings

A careful attacker can still defeat canaries, by

- overwriting the canary with the correct value
- corrupting a pointer to point to the return address to then change the return address without killing the canary



## Further improvements

- **Re-order elements on the stack to reduce the potential impact of overruns**
  - swapping parameters `buf` and `fp` on stack changes whether overrunning `buf` can corrupt `fp`
    - which is especially dangerous if `fp` is a function pointer
  - hence it is safer to allocated array buffers 'above' all other local variables

First introduced by IBM's ProPolice.

- A separate **shadow stack for return addresses**
  - with copies of return addresses, used to check for corrupted return addresses

Of course, the attacker should not be able to corrupt the shadow stack

# Windows 2003 Stack Protection

*Nice example of the ways in which things can go wrong...*

- Enabled with /GS command line option in Visual Studio
- When canary is corrupted, control is transferred to an exception handler
- Exception handler information is stored ...  
on the stack!
- Attacker could corrupt the exception handler info on the stack, in the process corrupt the canaries, and then let Stack Protection mechanism transfer control for him

*[<http://www.securityfocus.com/bid/8522/info>]*

- Countermeasure: only allow transfer of control to registered exception handlers



## 2. ASLR (Address Space Layout Randomisation)

- Attacker needs detailed info about memory layout
  - eg to jump to specific piece of code
  - or to corrupt a pointer at a know position on the stack
- Attacks become harder if we **randomise** the memory layout every time we start a program
  - **ie. change the offset of the heap, stack, etc, in memory by some random value**
- Attackers can still analyse memory layout on their own laptop, but will have to determine the offsets used on the victim's machine to carry out an attack
- NB **security by obscurity**, despite its bad reputation, is a really great defense mechanism to annoy attackers!
- Once the offset leaks, we're back to square one..

### 3. Non-eXecutable memory (NX , W $\oplus$ X)

Distinguish

- X: executable memory (for storing code)
  - W: writeable, non-executable memory (for storing data)
- and let processor refuse to execute non-executable code

How does this help?

Attacker can no longer jump to his own attack code,  
as any input he provides as attack code will be  
non-executable

Aka DEP (Data Execution Prevention).

Intel calls it eXecute-Disable (XD)

AMD calls it Enhanced Virus Protection

## Defeating NX: return-to-libc attacks

Code *injection* attacks no longer possible,  
but code *reuse* attacks still are...

So instead of jumping to own attack code in non-executable  
buffer

overflow the stack to jump to code that is already there,  
esp. library code in `libc`

`libc` is a rich library that offers lots of functionality,  
eg. `system()`, `exec()`,  
which provides attackers with all they need...

# reTURN oriented program Ming (ROP)

Next stage in evolution of attacks, as people removed or protected dangerous libc calls such as `system()`

Instead of using entire library call, the attacker

- looks for **gadgets**, small snippets of code which end with a **return**, in the existing code base

```
...; ins1 ; ins2 ; ins3 ; ret
```

- strings these gadgets together as subroutines to form a program that does what he wants

This turns out to be doable

- Most libraries contain enough gadgets to provide a **Turing complete programming language**
- **ROP compilers** can then translate arbitrary code to a string of these gadgets

## 4. Control Flow Integrity (CFI)

Return-to-libc and ROP give rise to **unusual control flow jumps between code blocks**

Eg a function  $f()$  never calls library routine `exec()`, because `exec()` does not even occur in the code of  $f()$ , but when supplied with malicious input  $f()$  suddenly does call `exec()`

Idea behind Control Flow Integrity:

**determine the control flow graph (cfg) and monitor execution to spot deviant behaviour**

- Many variants, with different levels of precision, overhead, ...
- Of course, not all attacks results in unusual control flow. Eg buffer overflows that only corrupt data will not, so cannot be detected by CFI.

[Called **Execution Monitors/Policy Enforcement** in Younan et al]

# Control Flow Graph

For a large part, the cfg is static & can be determined at compile-time.

But complications in determining the cfg come from

- **function calls through function pointers**

```
void sort(int[] src, int[] dest,  
         boolean (*compare)(int,int) )
```

But whole program analysis can determine **the set of values a function pointer** can take

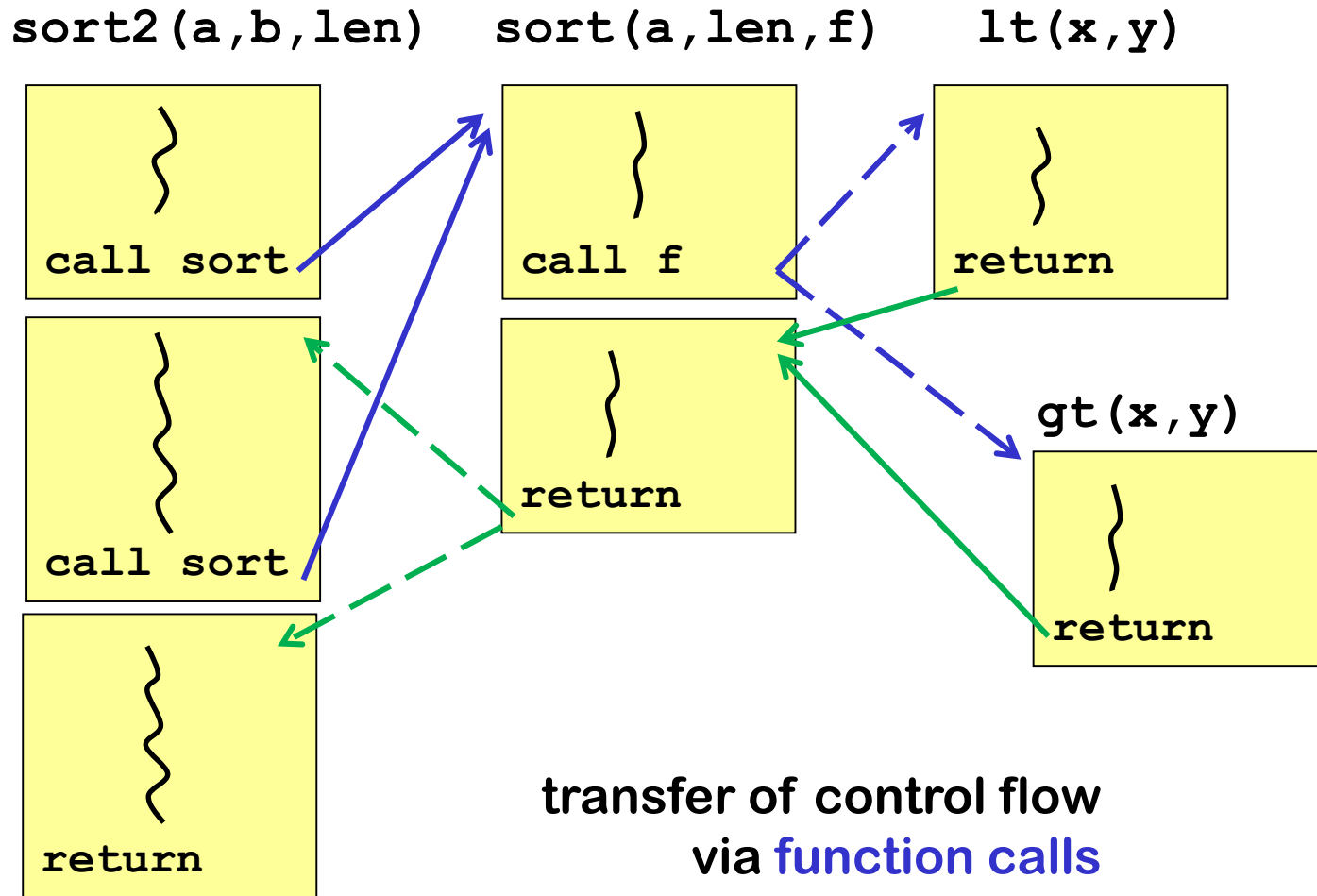
- **return's from functions**

But whole program analysis can determine the **set of valid return destinations**

## Example code

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
sort2(int src[ ], int dst[ ], int len)
{
    sort( src, len, lt );
    sort( dst, len, gt );
}
```

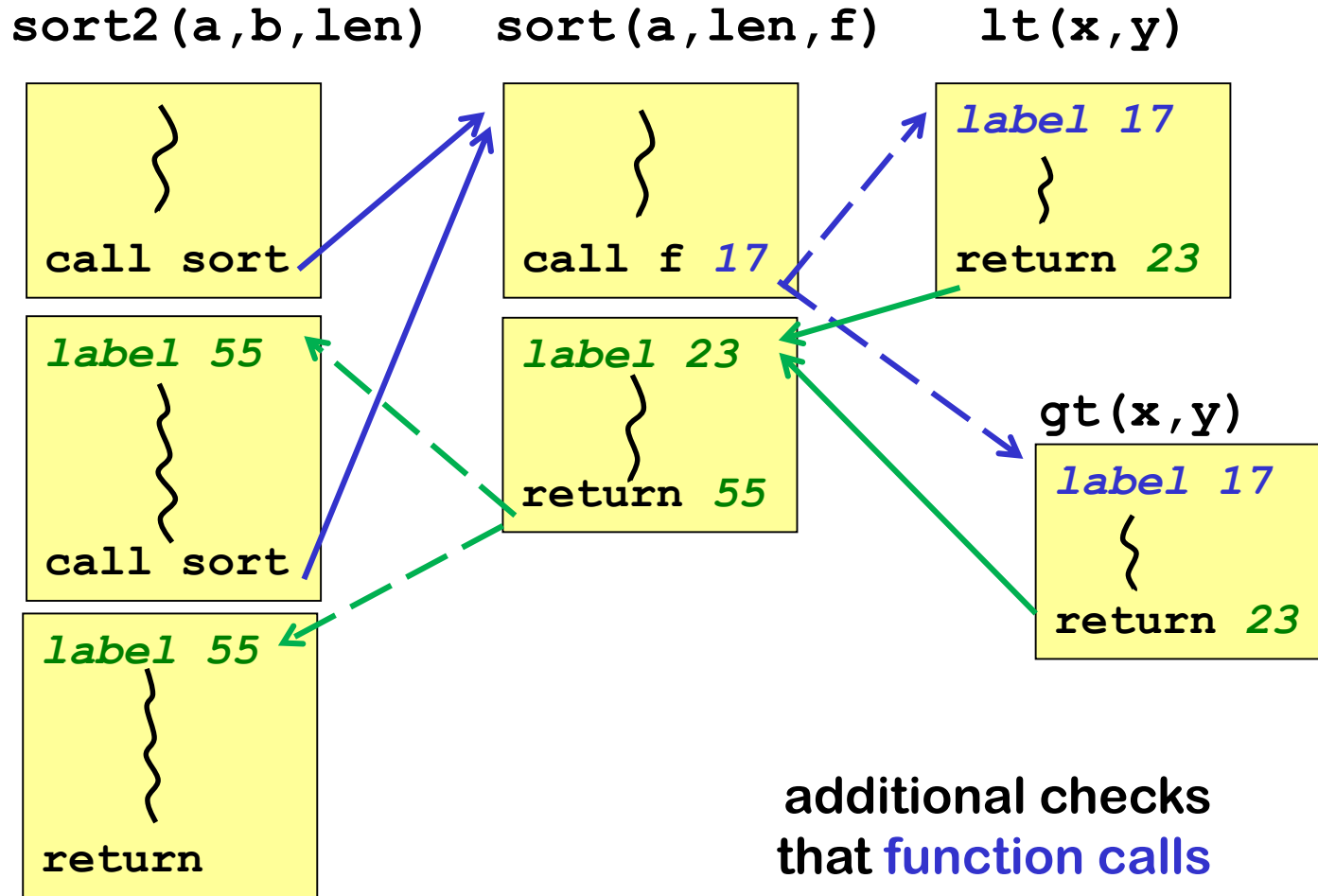
## Example control flow graph



transfer of control flow  
via **function calls**  
and **returns**



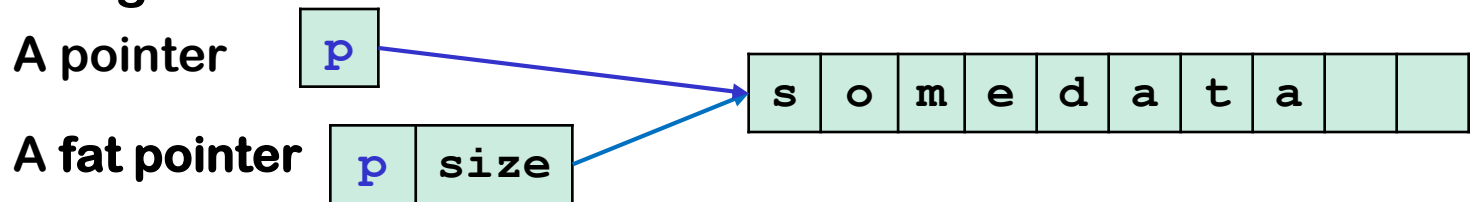
## Example control flow checks



## 5. Bound checkers


### Compiler

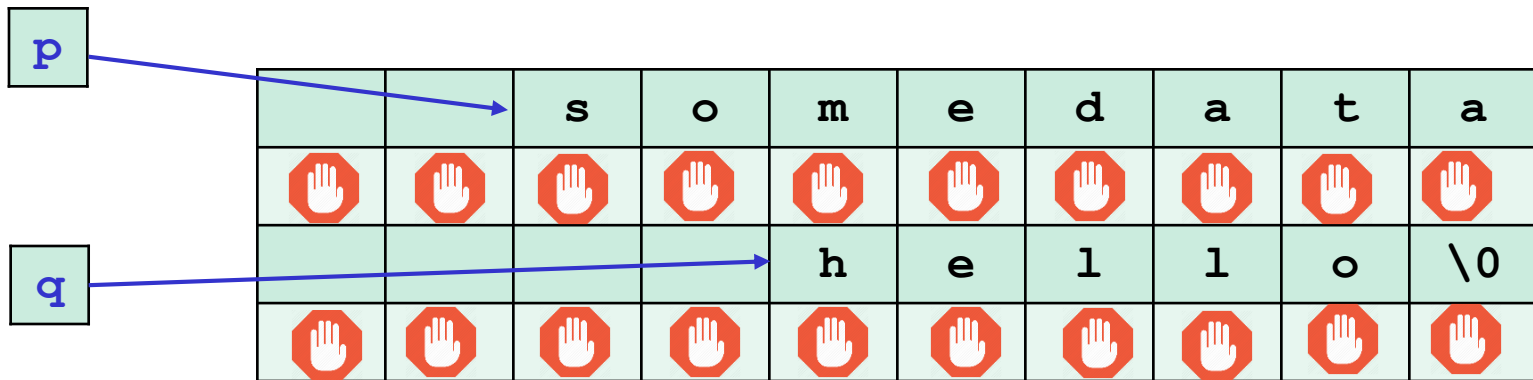
- **records size information** for all pointers (eg using so-called **fat pointers**)
- **adds runtime checks** for pointer arithmetic & array indexing



Considerable **execution time overhead**

## 6. Guard pages

Allocate heap chunks with the end at a **page boundary**  
with a non-readable, non-writable page  between chunks



Buffer over-write or over-read will cause a memory fault.

Considerable **memory overhead**

## 7. Pointer encryption

- Many buffer overflow attacks involve corrupting pointers, **pointers to data** or **function pointers**
- To complicate this: **Encrypt pointers in main memory, unencrypted in registers**
  - simple & fast encryption scheme: XOR with a fixed value, randomly chosen when a process starts
- **Attacker can still corrupt encrypted pointers in memory, but these will not decrypt to predictable values**
  - This uses *encryption* to ensure *integrity*. Normally NOT a good idea, but here it works.

[Cowan et al, PointGuard: protecting pointer from buffer overflow vulnerabilities, Usenix Security 2003]

[Called **Obfuscation of Memory Addresses** in Younan et al]

## 8. Instruction set randomisation

Basically the same thing as pointer encryption, but now for **instructions** (ie code in memory)

- If attackers can inject their own code, this will not decrypt to predictable instructions

Not so interesting any more, as with NX memory attackers no longer inject their own code, but abuse existing code

## 9. Execution-aware memory protection

- More fine-grained memory protection by OS & hardware:
  - not just access control based on **process id**
  - but also based on value of the **program counter (PC)**
- So some memory regions only accessible from specific part of the program code
  - eg. crypto keys only accessible if PC points to module with the crypto code
- Does not prevent buffer overflow, but reduces the *impact*
  - eg. only buffer overflows in the crypto code can leak keys

[Google, US patent 9395993 B2, July 2016]

[Koeberl et al., TrustLite: A security architecture for tiny embedded devices, *European Conference on Computer Systems*. ACM, 2014]