Tony Hoare on Design Principles of ALGOL 60

In his Turing Award lecture

"The first principle was *security: ...* A consequence of this principle is that *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 us to provide 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]

Overview

1. How buffer overflows – and other memory related problems – work?
2. **How to spot such problems in C(++) code?**
3. **What can the compiler do about it?**
4. **What can we do about it?**

Essence of the problem

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

    char buffer[4];

What happens if we execute the statement below ?

    buffer[4] = ‘a’;

This is UNDEFINED! *ANYTHING* can happen !

If the data written (ie. the “a”) is user input that can be controlled by an attacker, this vulnerability can be exploited: *anything that the attacker* wants can happen.

Not only writing outside array bounds in 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, for efficiency reasons.

  (Perl, Python, Java, C#, and even Visual Basic have)

- As a result, buffer overflows have been the no 1 security problem in software ever since

Other root causes

Apart from going outside array bounds there are other root causes

- malloc & free – ie. memory management by program(mer) rather than the language platform (using eg garbage collector)
- integer overflows
- format string attacks
- more generally: weak type systems, esp. for strings
- data races
- ....

Problems caused by buffer overflows

- The first Internet worm, and all subsequent ones (CodeRed, Blaster, ...), exploited buffer overflows

- Buffer overflows cause in the order of 50% of all security alerts
  - Eg check out CERT, cve.mitre.org, or bugtraq

- Trends
  - Attacks are getting cleverer
    - defeating ever more clever countermeasures
  - Attacks are getting easier to do, by script kiddies

Any C(++) code acting on untrusted input is at risk

- code taking input over untrusted network
  - eg. sendmail, web browser, wireless network driver,...
- code taking input from untrusted user on multi-user system,
  - esp. services running with high privileges (as ROOT on Unix/Linux, as SYSTEM on Windows)
- code acting on untrusted files
  - that have been downloaded or emailed
- also embedded software -

  eg. in devices with (wireless) network connections such as mobile phones, RFID card, airplane navigation systems, ...

How does buffer overflow work?

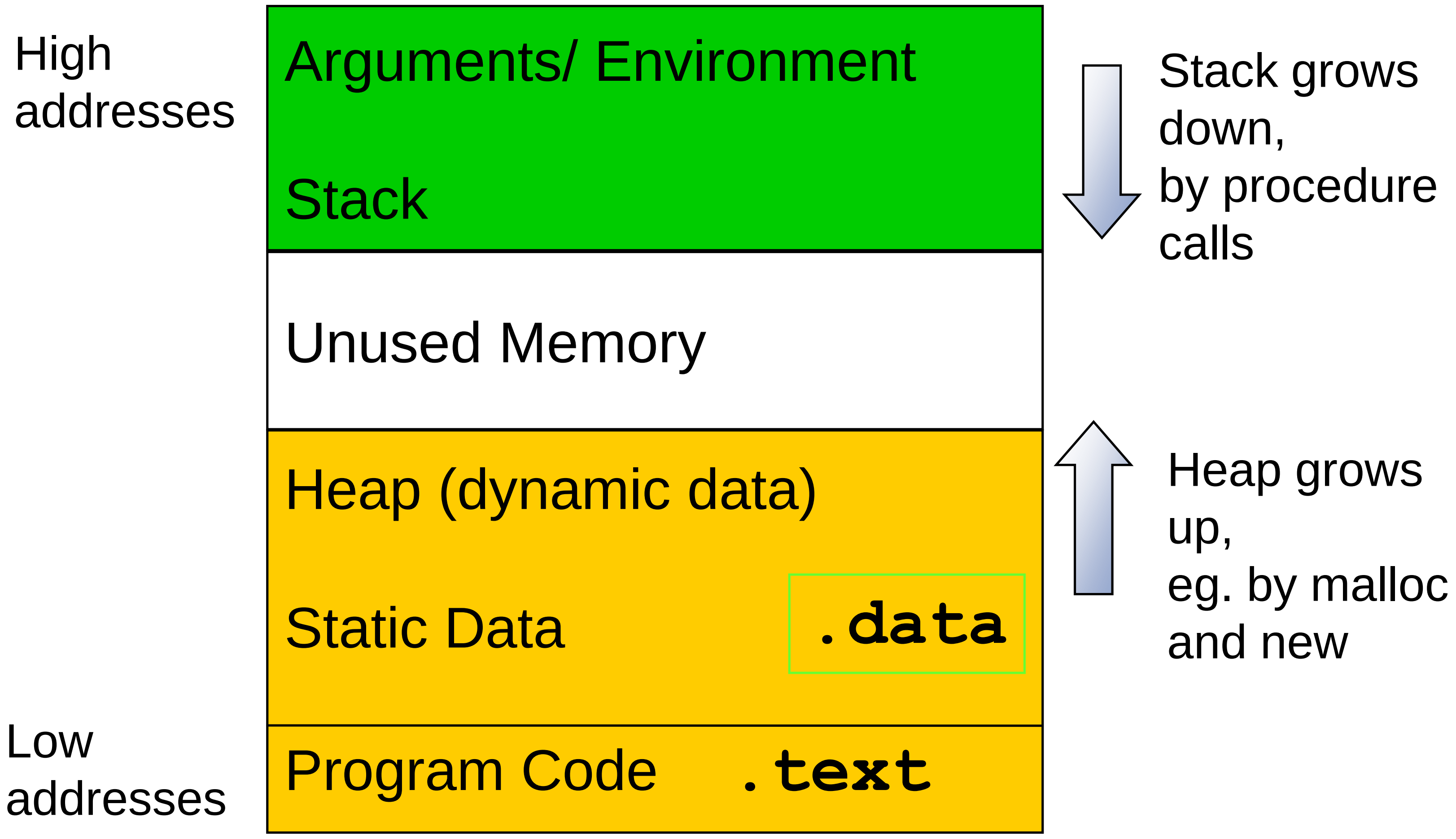
Memory management in C/C++

- Program responsible for its memory management
- Memory management is very error-prone
  - *Who here has had a C(++) program crash with a segmentation fault?*

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

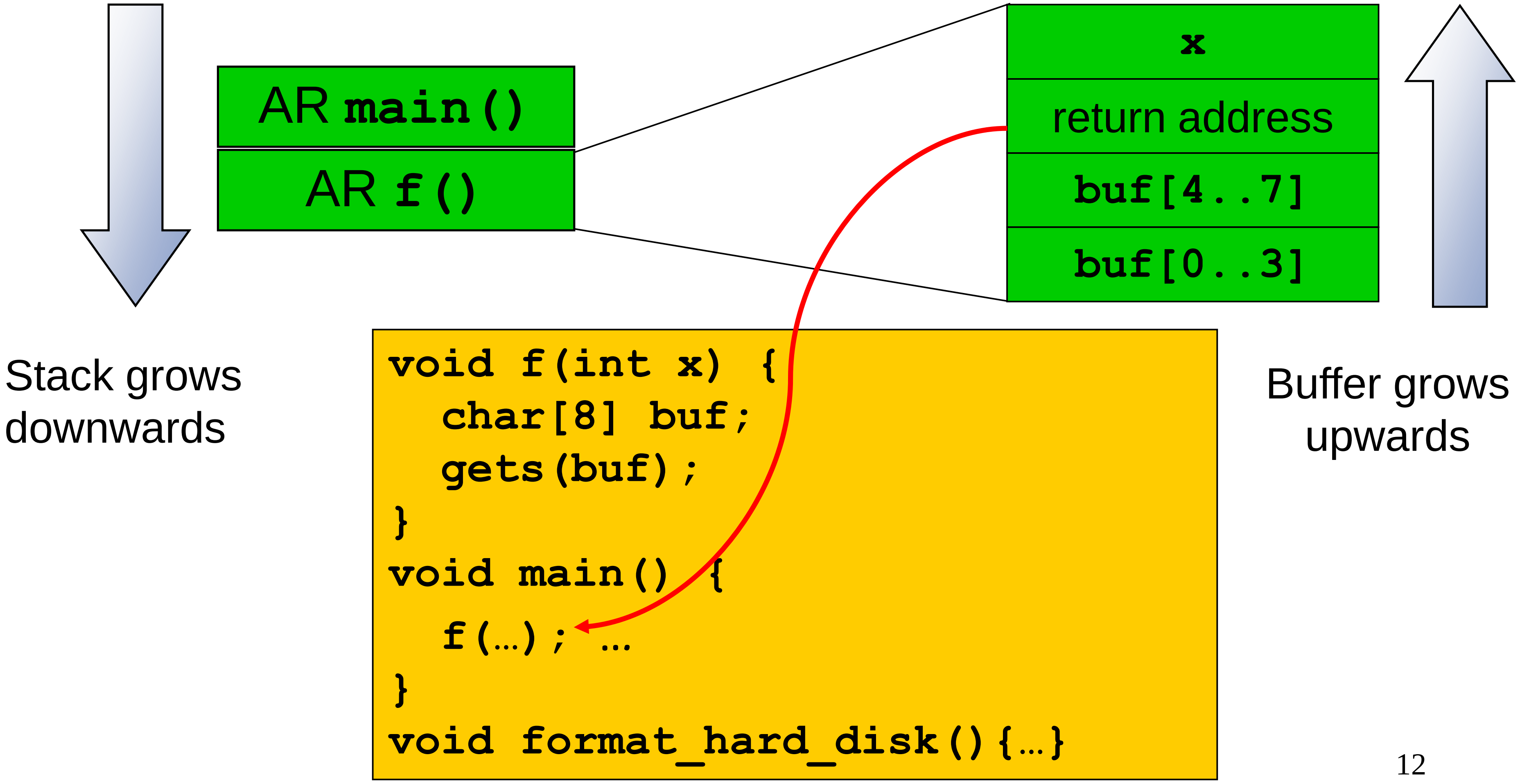
- Typical bugs:
  - Writing past the bound of an array
  - Pointer trouble
    - missing initialisation, bad pointer arithmetic, use after de-allocation (use after free), double de-allocation, failed allocation, forgotten de-allocation (memory leaks)...
- For efficiency, these bugs are not detected at run time:
  - behaviour of a buggy program is *undefined*

Process memory layout



Stack overflow

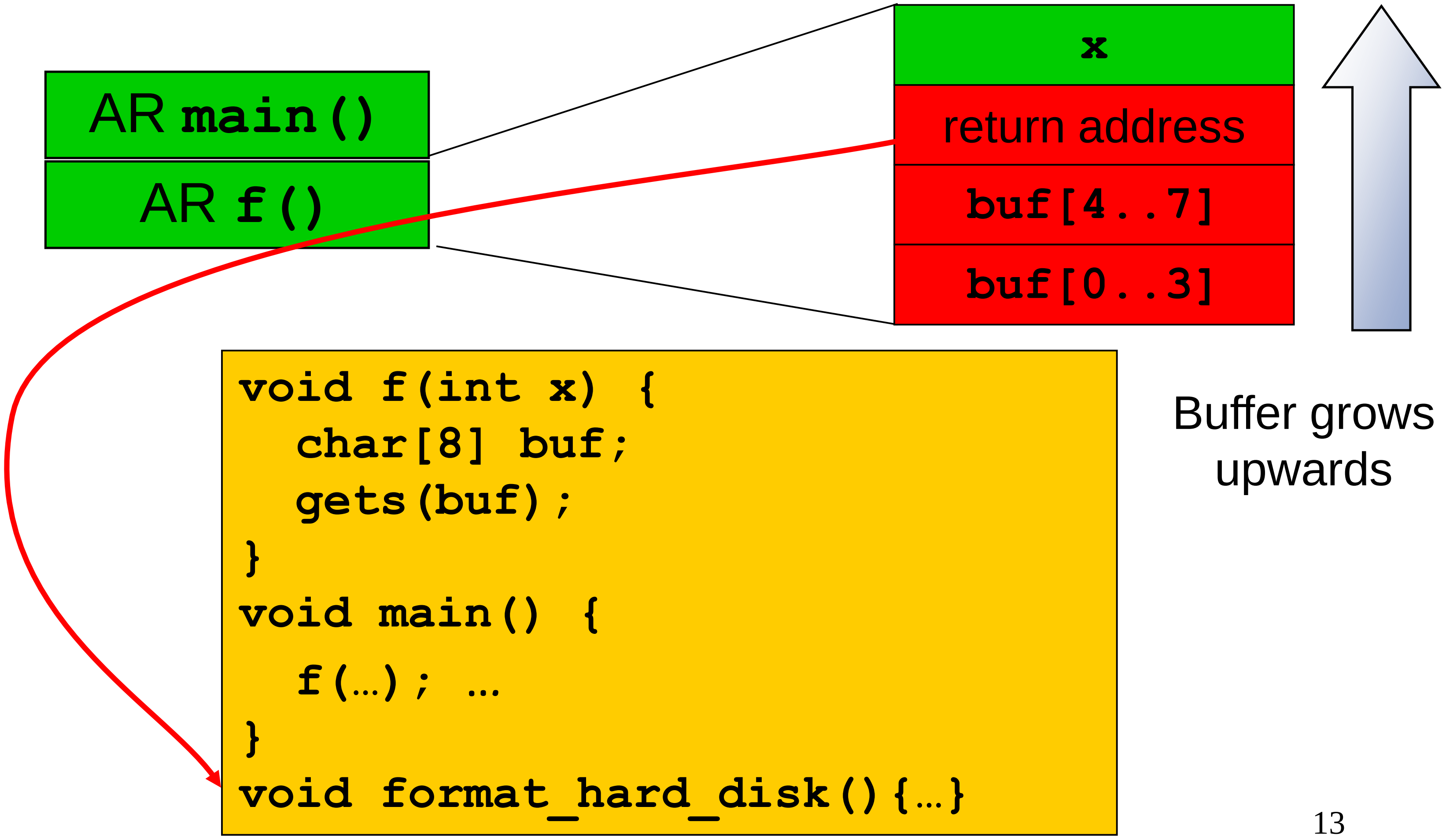
The stack consists of Activation Records:



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

Stack overflow

What if gets() reads more than 8 bytes ?

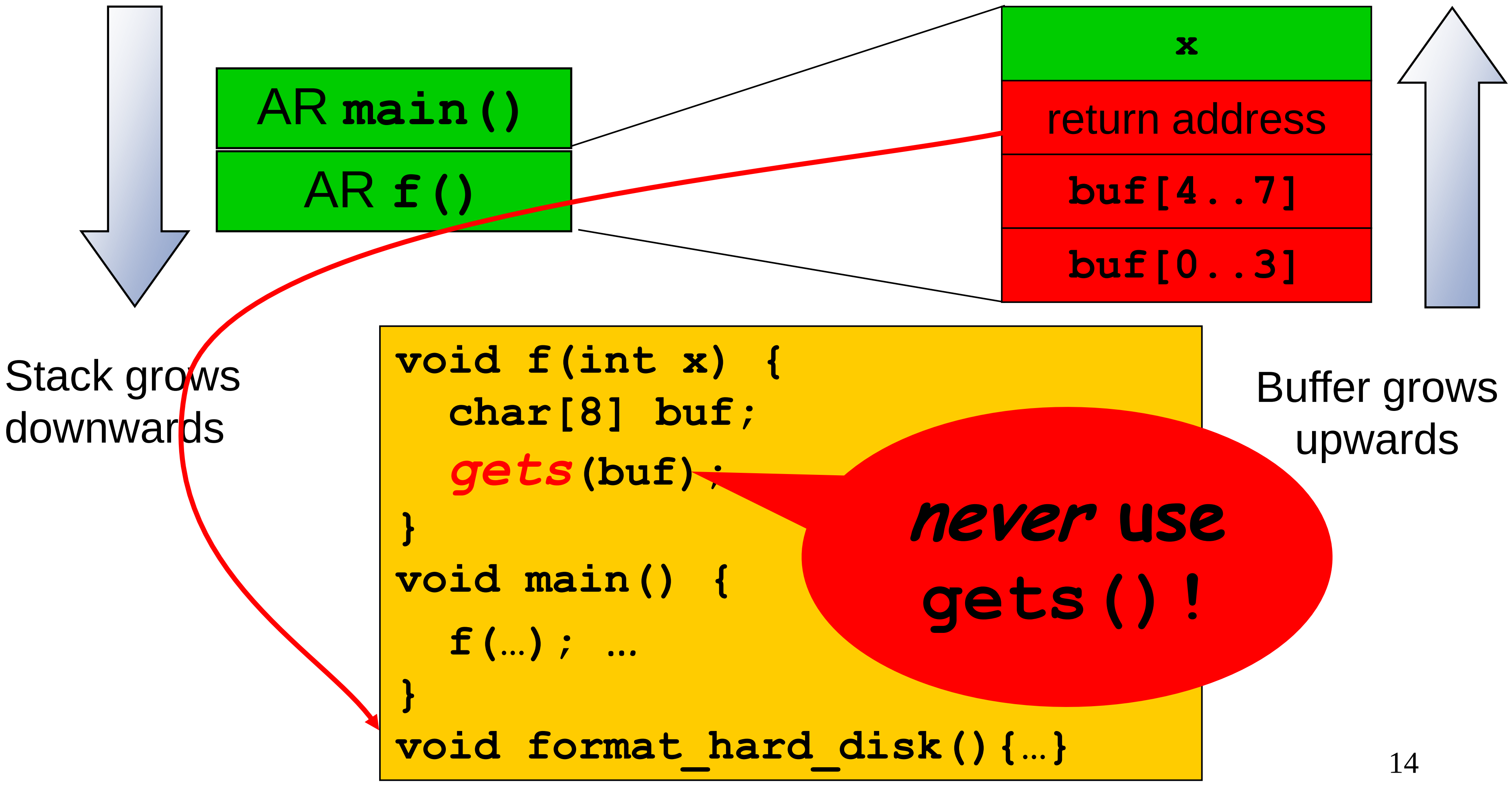


```
void f(int x) {
  char[8] buf;
  gets(buf);
}
void main() {
  f(…); …
}
void format_hard_disk(){…}
```

Buffer grows upwards

Stack overflow

What if gets() reads more than 8 bytes ?



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

Recurring problem: mixing control & data

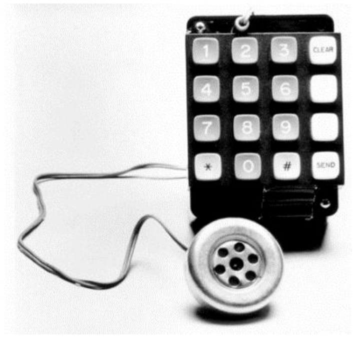
In 1950s, Joe Engressia showed the telephone network could be hacked by phone phreaking:

ie. whistling at right frequencies

The root cause: in-band signaling

http://www.youtube.com/watch?v=vVZm7I1CTBs

In 1970s, before founding Apple together with Steve Jobs, Steve Wozniak sold Blue Boxes for phone phreaking at university



Stack overflow, ie buffer overflow on the stack

- *How* the attacks works: overflowing buffers to corrupt data
- Lots of details to get right:
  - No nulls in (character-)strings
  - Filling in the correct return address:
    - Fake return address must be precisely positioned
    - Attacker might not know the address of his own string
  - Other overwritten data must not be used before return from function
  - …
- Variant: Heap overflow of a buffer allocated on the heap instead of the stack

What to attack? Fun with the <u>stack</u>

```
void f(char* buf1, char* buf2, bool b1) {
  int  i;
  bool b2;
  void (*fp)(int);
  char[] buf3;
  ....
}
```

Overflowing stack-allocated buffer **`buf3`** to

- corrupt return address
  - ideally, let this return address point to another buffers where the attack code is placed
- corrupt function pointers, such as **`fp`**
- corrupt any other data on the stack, eg. **`b2, i, a,`** ...

Spotting the problem

Example: `gets`

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

- *Never* use **`gets`**
- Use **`fgets(buf, size, stdin)`** 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

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
```

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

Another common mistake is giving **`sizeof(path)`** as 3rd argument...

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)

`base_url` is 10 chars long, incl. its null terminator, so **`src`** will not be null-terminated

```
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];

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

base_url is 10 chars long, incl. its null terminator, so **src** will not be null-terminated

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...

Spot the defect! (3)

```
char *buf;
int i, len;

read(fd, &len, sizeof(int));
     // read sizeof(int) bytes, ie. an int,
     // and store these at &len, ie. the
     // memory address of the variable len
buf = malloc(len);
read(fd,buf,len); // read len bytes into buf
```

Spot the defect! (3a)

We forget to check for bytes representing a negative int, so **`len`** might be given a negative value

```
 char *buf;
int i, len;

read(fd, &len, sizeof(int));
     // read sizeof(int) bytes, ie. an int,
     // and store these at &len, ie. the
     // memory address of the variable len
buf = malloc(len);
read(fd,buf,len); // read len bytes into buf
```

`len` cast to unsigned and negative length overflows **`read`** then goes beyond the end of **`buf`**

Spot the defect! (3b)

```
char *buf;
int i, len;

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

Remaining problem may be that **`buf`** is not null-terminated

Spot the defect! (3c)

```
char *buf;
int i, len;

read(fd, &len, sizeof(len));
if (len < 0)
   {error ("negative length"); return; }
buf = malloc(len+1);
read(fd,buf,len);
buf[len] = '\0'; // null terminate buf
```

May result in integer overflow;
we should check that
`len+1` is positive

Spot the defect! (3d)

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

```
char *buf;
int i, len;

read(fd, &len, sizeof(len));
if (len < 0)
   {error ("negative length"); return; }
buf = malloc(len+1);
read(fd,buf,len);
buf[len] = '\0'; // null terminate buf
```

Spot the defect! (3e)

```
char *buf;
int i, len;

read(fd, &len, sizeof(len));
if (len < 0)
   {error ("negative length"); return; }
buf = malloc(len+1);
if (buf==NULL) { exit(EXIT_FAILURE);}
             // or something a bit more graceful
read(fd,buf,len);
buf[len] = '\0'; // null terminate buf
```

Spot the defect! (3f) - Better still?

```
char *buf;
int i, len;

read(fd, &len, sizeof(len));
if (len < 0)
   {error ("negative length"); return; }
buf = calloc(len+1);
      //to initialise allocate memory to 0
if (buf==NULL) { exit(EXIT_FAILURE);}
               // or something a bit more graceful
read(fd,buf,len);
buf[len] = '\0'; // null terminate buf
```

NB (Absence of) Language-Level Security

Note that in other (nicer?) programming languages, we might not have to worry about

- writing past array bounds (the language could throw an IndexOutOfException instead)
- implicit conversions from signed to unsigned integers
- malloc possibly returning null (the language could throw an OutOfMemoryException instead)
- malloc not initialising memory
- integer overflow (the language could throw an IntegeroverflowException instead)
- ...

Spot the defect! (4)

```
#ifdef UNICODE
#define _sntprintf _snwprintf
#define TCHAR wchar_t
#else
#define _sntprintf _snprintf
#define TCHAR char
#endif

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

[slide from presentation by Jon Pincus]

Spot the defect! (4)

```
#ifdef UNICODE
#define _sntprintf _snwprintf
#define TCHAR wchar_t
#else
#define _sntprintf _snprintf
#define TCHAR char
#endif
```

_sntprintf's 2nd param is # of chars in buffer, not # of bytes

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

The CodeRed worm exploited such an mismatch, where code written under the assumption that 1 char was 1 byte allowed buffer overflows after the move from ASCI to Unicode

[slide from presentation by Jon Pincus]

Spot the defect! (5)

```
#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! (5)

```
#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 `input` is longer than 32K ?

len will be a negative number, due to integer overflow

hence: potential buffer overflow

The integer overflow is the root problem, but the (heap) buffer overflow that this enables make it exploitable

Spot the defect! (6)

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

Spot the defect! (6)

```
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 the Visual Studio C++ compiler adds check to prevent this

Spot the defect! (7)

```
char buff1[MAX_SIZE], buff2[MAX_SIZE];
// make sure url a valid URL and fits in buff1 and
   buff2:
if (! isValid(url)) return;
if (strlen(url) > MAX_SIZE - 1) return;
// copy url up to first separator, ie. first '/', to
   buff1
out = buff1;
do {
  // skip spaces
   if (*url != ' ') *out++ = *url;
} while (*url++ != '/');
strcpy(buff2, buff1);
...
```

[slide from presentation by Jon Pincus]

Spot the defect! (7)
Loop termination (exploited by Blaster)

```
char buff1[MAX_SIZE], buff2[MAX_SIZE];
// make sure url a valid URL and fits in buff1 and
   buff2:
if (! isValid(url)) return;
if (strlen(url) > MAX_SIZE - 1) return;
// copy url up to first separator, ie. first '/', to
   buff1
out = buff1;
do {
  // skip spaces
  if (*url != ' ') *out++ = *url;
} while (*url++ != '/');
strcpy(buff2, buff1);
...
```

length up to the first null

what if there is no '/' in the URL?

[slide from presentation by Jon Pincus]

Spot the defect! (8)

```
#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 attack, invented discovered in 2000.

Another way of corrupting the stack.

- 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 *may* happen if we execute

  ```
  printf(string)
  ```

  where **`string`** is user-supplied ?

  Esp. if it contains special characters, eg %s, %x, %n, %hn?

Format string attacks

- **%x** reads and prints some bytes from stack

  This can leak sensitive data

  Eg

  ```
  printf(s);
  ```

  will dump the stack if attacker supplies as input **s** the string

  ```
  %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x
  %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x
  %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x
  %x
  ```

- **%n** writes the number of characters printed so far onto the stack

  This allows the attacker to corrupt the stack...

Advanced format string attacks

An attacker can try to control which address is used for reading from memory using **`%s`** or for writing to memory using **`%n`** with specially crafted format strings

- **`\xEF\xCD\xCD\xAB %x %x  ... %x %s`**

  With the right number of **`%x`** characters, this will print the string located at address **`ABCDCDEF`**

- **`\xEF\xCD\xCD\xAB %x %x  ... %x %n`**

  With the right number of **`%x`** characters, this will write the number of characters printed so far to location **`ABCDCDEF`**

The tricky things are inserting the right number of **`%x`**, and choosing an interesting address

Format string attacks should be history…

Format string attacks are easy to spot & fix:

replace **`printf(str)`**

with **`printf("%s", str)`**

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: poor programming with arrays and strings
  - esp. library functions for null-terminated strings
- related attacks
  - format string attack: another way of corrupting stack
  - integer overflows: a stepping stone on the way to get buffers to overflows
- Other low level flaws can also cause memory problems, eg data races

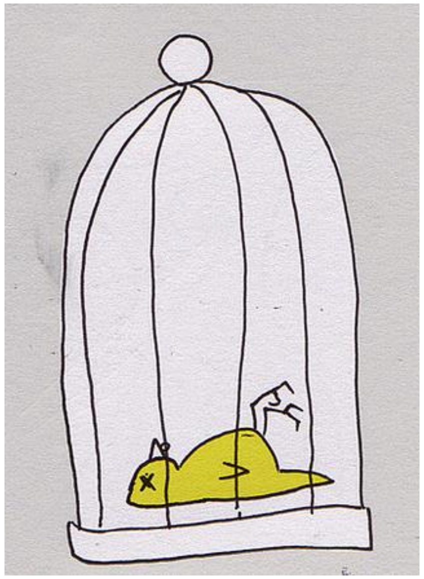
Runtime aka dynamic countermeasures

These are things that the compiler can do,

without the programmer needing to know!

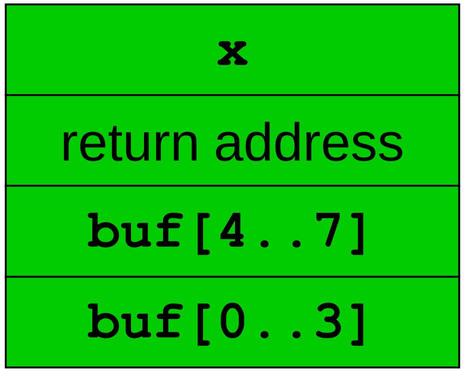
stack canaries

- introduced in StackGuard in gcc
- 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.

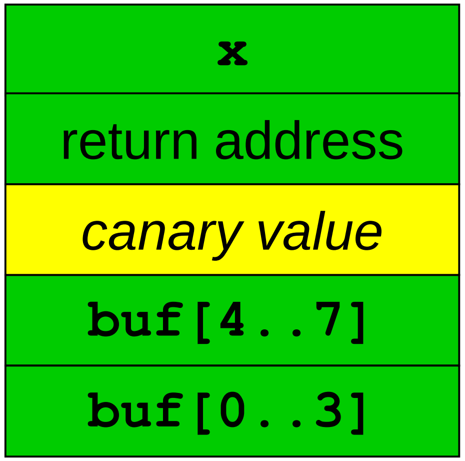


stack canaries

Stack without canary



Stack with canary



stack canaries

- introduced in StackGuard in gcc
- 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

- a careful attacker can overwrite the canary with the correct value
- additional countermeasures:
  - use a random value for the canary
  - XOR this random value with the return address
  - include string termination characters in the canary value

Further improvements of stack canaries

- PointGuard
  - also protects other data values, eg function pointers, with canaries

- ProPolice's Stack Smashing Protection (SSP) by IBM
  - also re-orders stack elements to reduce potential for trouble:

    swapping parameters x and y on the stack changes whether overrunning x can corrupt y

    this is especially dangerous if y is a function pointer

- Stackshield has a special stack for return addresses, and can disallow function pointers to the data segment

Non-eXecutable memory (NX aka W⊕X)

Distinguish

- X: executable memory (for storing code)
- W: writeable, non-executable memory (for storing data)

and let processor refuses to execute non-executable code

This can be done for the stack, or for arbitrary memory pages

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

Return-to-libc attacks

Way to get around non-executable memory:

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

instead of jumping to your own attack code.

`libc` is a rich library that offers many possibilities for attacker, eg.
`system, exec, fork`

which provide the attacker with any functionality he wants..

Return-oriented programming

Next stage in evolution of attacks, esp. on 64 bit x86, as people removed or protected dangerous library calls like **`system()`**

Instead of using entire library call, the attacker

- looks for small snippets of code which end with a return (so-called gadgets)

```
... ; ... ; ... ; ret
```

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

Most libraries contain enough gadgets to provide a Turing complete programming language.

Return-to-libc is essentially simple case of return oriented programming.

Control Flow Integrity (CFI)

A return-to-libc attack can be spotted as an unusual call pattern

Eg a function **`foo()`** never calls library routine **`bar()`** - and **`bar()`** does not even occur in the code of **`foo()`** - but when supplied with malicious input **`foo()`** suddenly does call **`bar()`**

With some extra administrative overhead for function calls, such strange call patterns can be detected and stopped at runtime.

Address Space Layout Randomisation (ASLR)

- Attacker needs detailed information on memory layout
  (eg to jump to specific piece of code)

- By randomising the layout every time we start a program
  - ie. moving the offset of the heap, stack, etc, by some random value

  the attacker’s life becomes much harder

- When to randomise? Only when we start a new program, or should we re-randomise when we fork() a process?

Dynamic countermeasures (recap)

- canaries
- non-executable memory (W⊕X)
- address space layout randomisation (ASLR)
- Control Flow Integrity

None of these countermeasures are perfect!

A determined attacker can and will find a way around them.

eg by figuring out canary values, figuring out the offset used in address randomisation, returning to libc, etc.

Moreover, they do not protect against heap overflows

Other countermeasures

Countermeasures

- We can take countermeasures at different points in time
  - before we even begin programming
  - during development
  - at compilation time
  - when testing
  - when executing code

to prevent, migitate, or detect buffer overflows problems

Generic defence mechanisms

- Reducing attack surface

  Not running or even installing certain software, or enabling all features by default, mitigates the threat

- Mitigating impact by reducing permissions

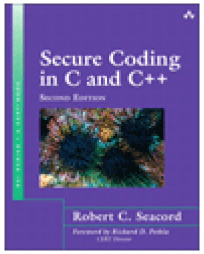
  Reducing OS permissions of software (or user) will restrict the damage that an attack can have

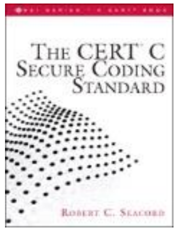
  - principle of least privilege

Prevention

- Don’t use C or C++
  - you can write insecure code in any programming language, but some make it easier...
- Better programmer awareness & training

  Read – and make other people read – books like
  - Building Secure Software, J. Viega & G. McGraw, 2002
  - Writing Secure Code, M. Howard & D. LeBlanc, 2002
  - 24 deadly sins of software security, M. Howard, D LeBlanc & J. Viega, 2005
  - Secure programming for Linux and UNIX HOWTO, D. Wheeler,
  - Secure C coding, T. Sirainen
  - CERT secure coding guidelines for C and C++ www.securecoding.cert.org





Dangerous C system calls

source: Building secure software, J. Viega & G. McGraw, 2002

Extreme risk

- `gets`

High risk

- `strcpy`
- `strcat`
- `sprintf`
- `scanf`
- `sscanf`
- `fscanf`
- `vfscanf`
- `vsscanf`
- `streadd`
- `strecpy`
- `strtrns`
- `realpath`
- `syslog`
- `getenv`
- `getopt`
- `getopt_long`
- `getpass`

Moderate risk

- `getchar`
- `fgetc`
- `getc`
- `read`
- `bcopy`

Low risk

- `fgets`
- `memcpy`
- `snprintf`
- `strccpy`
- `strcadd`
- `strncpy`
- `strncat`
- `vsnprintf`

Prevention – use better string libraries

- there is a choice between using statically vs dynamically allocated buffers
  - static approach easy to get wrong, and chopping user input may still have unwanted effects
  - dynamic approach susceptible to out-of-memory errors, and need for failing safely

Better string libraries (1)

- libsafe.h provides safer, modified versions of eg. strcpy
  - prevents buffer overruns beyond current stack frame in the dangerous functions it redefines

- libverify enhancement of libsafe
  - keeps copies of the stack return address on the heap, and checks if these match

- **`strlcpy(dst,src,size)`** and **`strlcat(dst,src,size)`**

  with **`size`** the size of **`dst`**, not the maximum length copied.

  Consistently used in OpenBSD

Better string libraries (2)

- glib.h provides Gstring type for dynamically growing null-terminated strings in C
  - but failure to allocate will result in crash that cannot be intercepted, which may not be acceptable

- Strsafe.h by Microsoft guarantees null-termination and always takes destination size as argument

- C++ string class
  - but **`data()`** and **`c-str()`** return low level C strings, ie **`char*`**, with result of **`data()`** is not always null-terminated on all platforms...

Runtime detection on instrumented binaries

There are many memory error detection tools that instrument binaries to allow runtime detection of memory errors, esp.

- out-of-bounds access
- use-after-free bugs on heap

with some overhead (time, memory space) but no false positives

For example Valgrind (Memcheck), Dr. Memory, Purify, Insure++, BoundsChecker, Cachegrind, Intel Parallel Inspector, Discoverer, AddressSanitizer,…

Safer variants of C

Some approaches go further and propose safer dialects of C which include

- bound checks,
- type checks
- automated memory management, to ensure memory safety
  - by garbage collection or region-based memory management

Examples are Cyclone, CCured, Vault, Control-C, Fail-Safe C, …

Fuzzing aka fuzz testing

Testing for security can be difficult!

- How to hit the right cases?

A classic technique to find buffer weaknesses is **fuzz testing**

- send *random, very long* inputs, to an application
- if there are buffer overflow weaknesses,
  this is likely to crash the application with a segmentation fault

The nice thing is that this is easy to automate!

More on fuzz testing in the security testing lecture.

Code review & static analysis

- Code reviews
  ie. someone reviewing the code manually
  Expensive & labour intensive

- Code scanning tools aka static analysis
  Automated tools that look for suspicious patterns in code;
  ranges for **CTRL-F** or **`grep`**, to advanced analyses

  Incl. free tools
  - RATS – also for PHP, Python, Perl
  - Flawfinder , ITS4, Deputy, Splint
  - PREfix, PREfast by Microsoft

  plus other commercial tools
  Coverity, PolySpace, Klockwork, ...

(formal) verification

The most extreme form of static analysis:

- Program verification

  proving by mathematical means (eg Hoare logic) that memory management of a program is safe

  - extremely labour-intensive ☹
  - eg hypervisor verification project by Microsoft & Verisoft:
    - http://www.microsoft.com/emic/verisoft.mspx

Beware: in industry “verification” means testing,

in academia it means formal program verification

Conclusions

“C is a terse and unforgiving abstraction of silicon”

Summary

- Buffer overflows are a top security vulnerability
- Any C(++) code acting on untrusted input is at risk

  so effectively

  Any C(++) code is at risk
- Getting rid of buffer overflow weaknesses in C(++) code is hard and may prove to be impossible
  - Ongoing arms race between countermeasures and ever more clever attacks.
  - Attacks are not only getting cleverer, using them is getting easier

Common themes

Buffer overflows involve three more general problems:

1. lack of input validation
2. mixing data & code
   namely data and return address on the stack
3. believing in & relying on an abstraction that is not 100% guaranteed & enforced
   namely types and procedure interfaces in C

   ```
   int f(float f, boolean b, char* buf);
   ```