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';
```

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.

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.
  (Ada, 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.

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

Eg

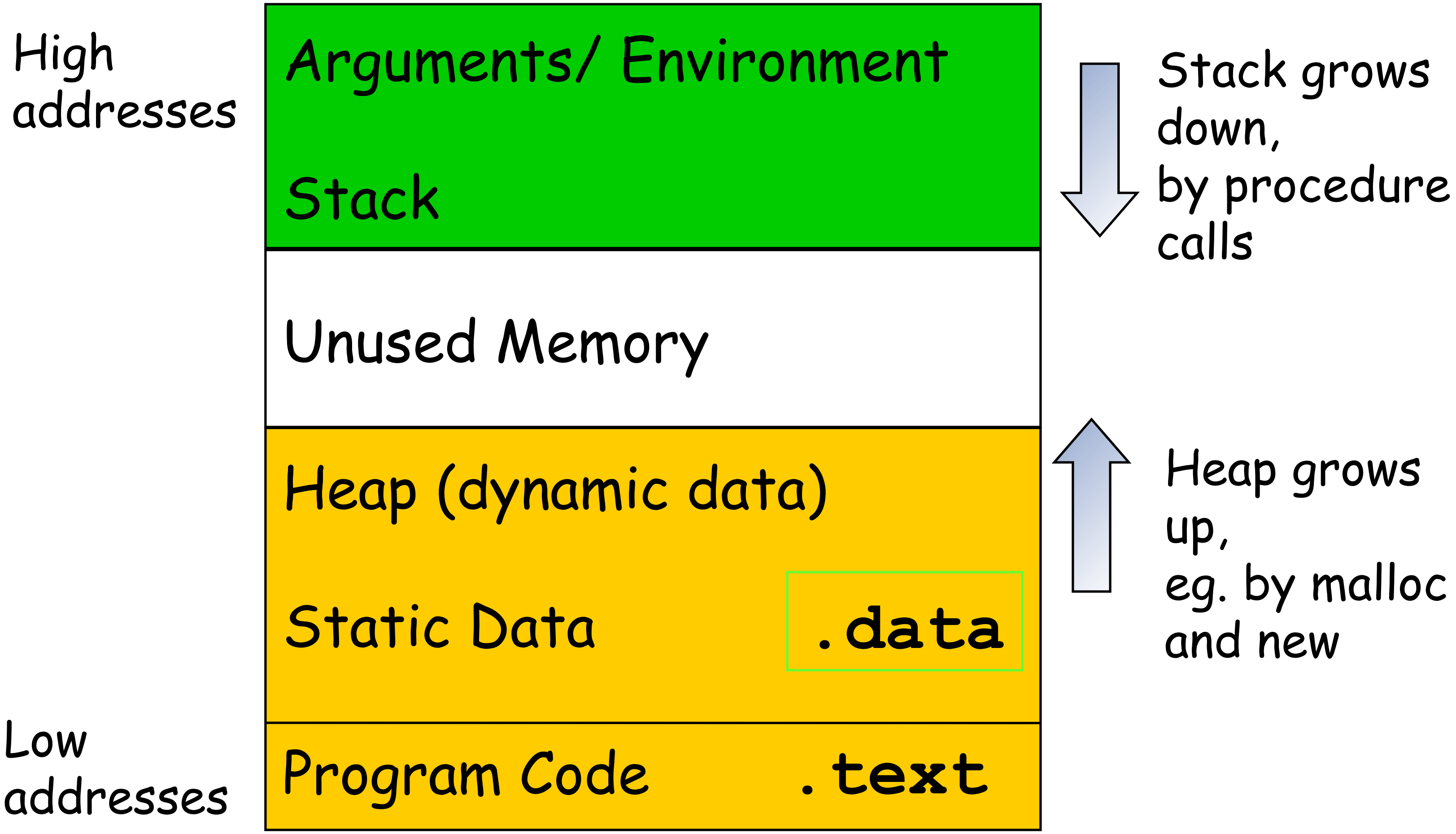
- code taking input over untrusted network
  - eg. sendmail, web browser, wireless network driver,...
- code taking input from untrusted user on multi-user system,
  - esp. services running with high privileges (as ROOT on Unix/Linux, as SYSTEM on Windows)
- code acting on untrusted files
  - that have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

How does buffer overflow work?

Memory management in C/C++

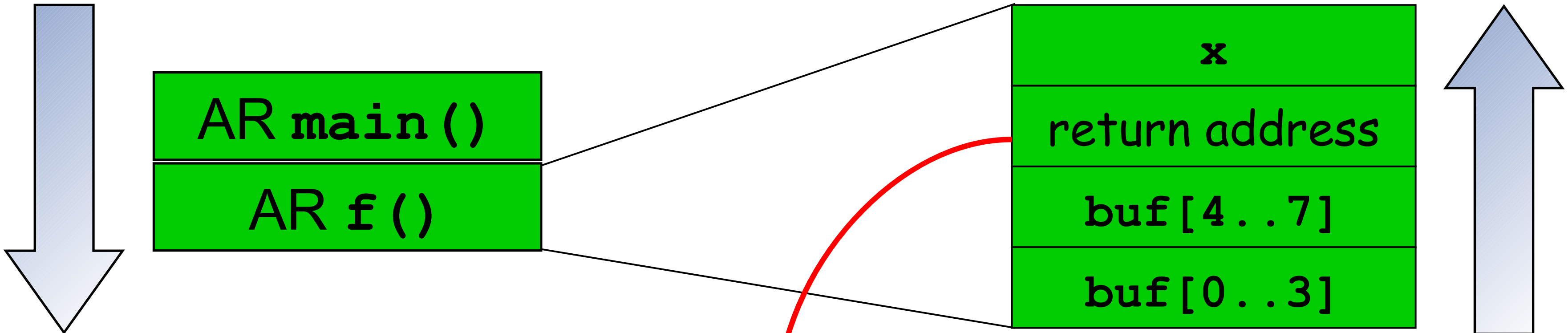
- Program responsible for its memory management
- Memory management is very error-prone
  - *Who here has had a C(++) program crash with a segmentation fault?*
- Typical bugs:
  - Writing past the bound of an array
  - Dangling pointers
    - missing initialisation, bad pointer arithmetic, incorrect de-allocation, double de-allocation, failed allocation, ...
  - Memory leaks
- For efficiency, these bugs are not detected at run time, as discussed before:
  - behaviour of a buggy program is *undefined*

Process memory layout



Stack overflow

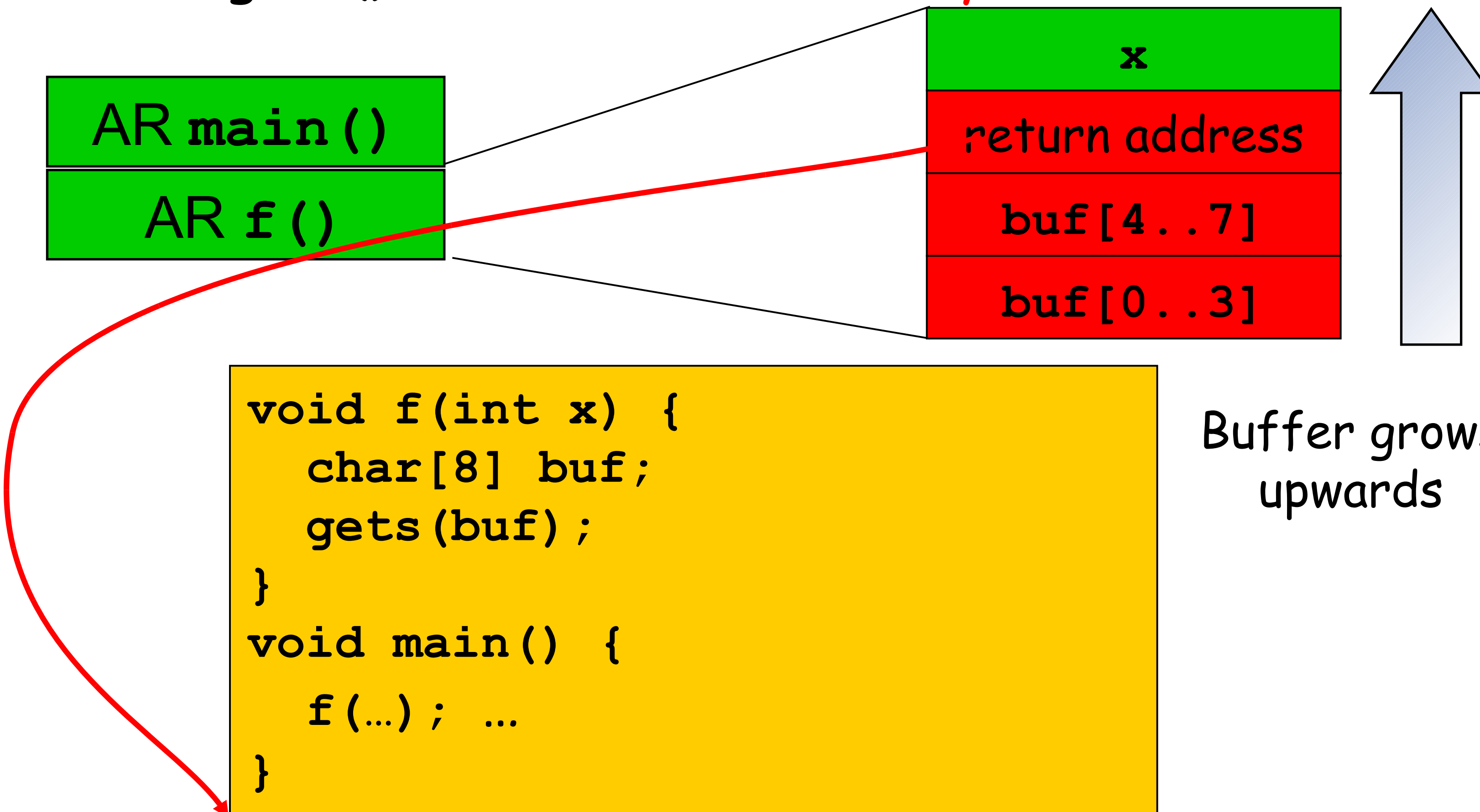
The stack consists of Activation Records:




```
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 ?

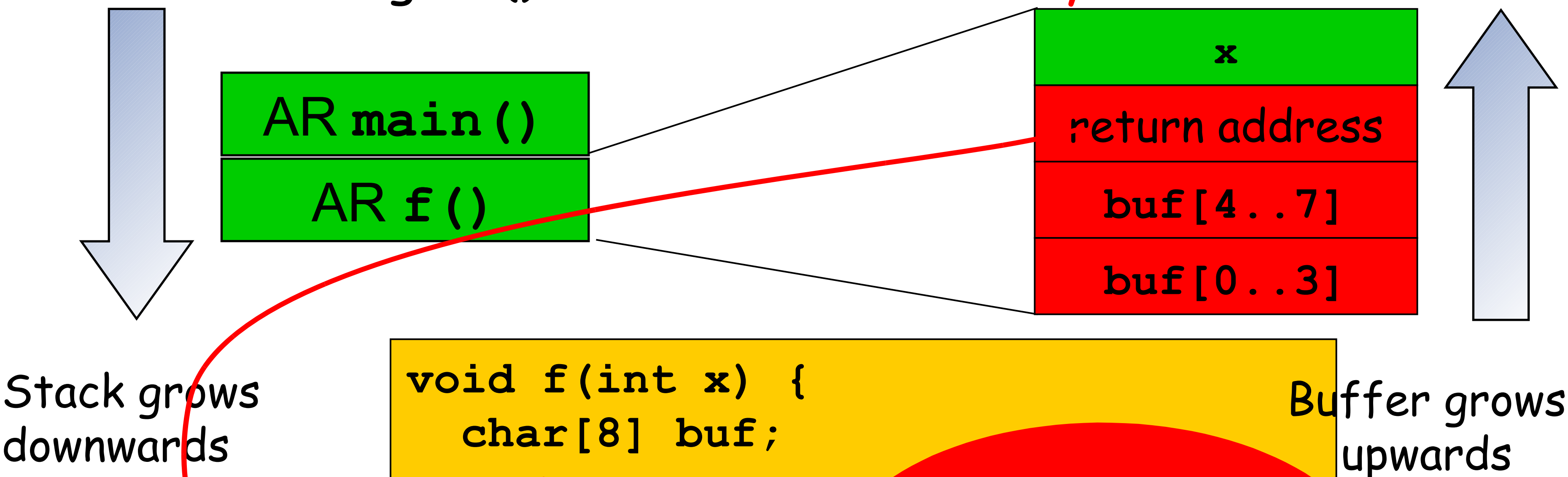



```
void f(int x) {
   char[8] buf;
   gets(buf);
}
void main() {
   f(…); …
}
void format_hard_disk(){…}
```

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(){…}
```

never use gets()!

Stack overflow

- 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
  - …

Variants & causes

- Stack overflow is overflow of a buffer allocated on the stack
- Heap overflow idem, of buffer allocated on the heap

Common causes:

- poor programming with of arrays and strings
  - esp. library functions for null-terminated strings
- problems with format strings

But other low-level coding defects than can result in buffer overflows, eg integer overflows or data races

What causes buffer overflows?

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)

```
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** won't be null-terminated

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** won't be null-terminated

so **strcpy** will overrun the buffer **dest**

Example: `strcpy` and `strncpy`

- Don't replace

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

  by

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

  but by

  ```
  strncpy(dest, src, sizeof(dest)-1)
  dst[sizeof(dest-1)] = `\0`;
  ```

  if `dest` should be null-terminated!

- Btw: a strongly typed programming language could of course enforce that strings are always null-terminated...

Spot the defect! (3)

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

read(fd, &len, sizeof(len));
     // read sizeof(len) bytes, ie. an int
     // and store these in len
buf = malloc(len);
read(fd,buf,len);
```

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

len cast to unsigned and negative length overflows

read then goes beyond the end of **buf**

Spot the defect! (3)

```
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! (3)

```
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

Absence of language-level security

In programming languages with "security" provisions, the programmer would not have to worry about

- writing past the bounds of the array (IndexOutOfBoundsException for example)
- implicit conversion from signed to unsigned integers (forbidden or warned by compiler/typechecker)
- `malloc` returning null value (OutOfMemoryException for example)
- `malloc` non initializing memory (by default)
- integer overflow (IntegerOverflowException for example)

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);
```

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);
```

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

The CodeRed worm exploited such an ANSI/Unicode mismatch

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.

(Microsoft Visual Studio 2005(!) 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);
...
```

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?

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++ != '/') && (*url != 0);
strcpy(buff2, buff1);
...
```

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++ != '/') && (*url != 0);
strcpy(buff2, buff1);
...
```

Order of tests is wrong (note the first test includes ++)

What about 0-length URLs?

Is buff1 always null-terminated?

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

- Complete new type of attack, invented/discovered in 2000. Like integer overflows, it can lead to buffer overflows.
- 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 4 bytes from stack
  - this may leak sensitive data
- **%n** writes the number of characters printed so far onto the stack
  - this allow stack overflow attacks...

- Note that format strings break the "don't mix data & code" principle.
- Easy to spot & fix:

  replace `printf(str)` by `printf("%s", str)`

Dynamic countermeasures incl. stack canaries

Dynamic countermeasures

protection by kernel

- non-executable memory (NOEXEC)
  - prevents attacker executing her code
- address space layout randomisation (ASLR)
  - generally makes attacker's life harder
- instruction set randomisation
  - hardware support needed to make this efficient enough

protection inserted by the compiler

- stack canaries to prevent or detect malicious changes to the stack; examples to follow
- obfuscation of memory addresses

Doesn't prevent against heap overflows

Dynamic countermeasure: 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

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

Dynamic countermeasures

NB none of these protections is perfect!

Eg

- even if attacks to return addresses are caught, integrity of other data other the stack can still be abused
- clever attacks may leave canaries intact
- where do you store the "master" canary value
  - a cleverer attack could change it
- none of this protects against heap overflows
  - eg buffer overflow within a struct...
- ....

Windows 2003 Stack Protection

The subtle ways in which things can still go wrong...

- Enabled with /GS command line option
- Similar to StackGuard, except that when canary is corrupted, control is transferred to an exception handler
- Exception handler information is stored ... on the stack

- Countermeasure: register exception handlers, and don't trust exception handlers that are not registered or on the stack
- Attackers may still abuse existing handlers or point to exception outside the loaded module...

Other countermeasures

Countermeasures

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

  to prevent, to detect – at (pre)compile time or at runtime -, and to migitate problems with buffer overflows

Prevention

- Don't use C or C++
- Better programmer awareness & training

Eg read – and make other people read -

  - Building Secure Software, J. Viega & G. McGraw, 2002
  - Writing Secure Code, M. Howard & D. LeBlanc, 2002
  - 19 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

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`

High risk (cntd)

- `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...

Detection before shipping

- Testing
  - Difficult! How to hit the right cases?
  - Fuzz testing - test for crash on long, random inputs – can be succesful in finding some weaknesses
- Code reviews
  - Expensive & labour intensive
- Code scanning tools (aka static analysis)

  Eg
  - RATS () – also for PHP, Python, Perl
  - Flawfinder , ITS4, Deputy, Splint
  - PREfix, PREfast by Microsoft

  plus other commercial tools
  - Coverity
  - Parasoft
  - Klockwork.

More prevention & detection

- Bounds Checkers
  - add additonal bounds info for pointers and check these at run time
  - eg Bcc, RTcc, CRED, ...
  - RICH prevents integer overflows
- Safe variants of C
  - adding bound checks, but also type checks

  and more: eg garbage collection or region-based memory management)
  - eg Cyclone (http://cyclone.thelanguage.org), CCured, Vault, Control-C, Fail-Safe C, …

More prevention & detection

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:

https://link.springer.com/chapter/10.1007/978-3-642-05089-3_51

https://www.microsoft.com/en-us/research/project/vcc-a-verifier-for-concurrent-c/

Reducing attack surface

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

Summary

- Buffer overflows are the top security vulnerability
- Any C(++) code acting on untrusted input 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

More general

Buffer overflow is an instance of three more general problems:

1) lack of input validation
2) mixing data & code
   – data and return address on the stack
1) believing in & relying on an abstraction
   – in this case, the abstraction of procedure calls offered by C

- Attacks often exploit holes in abstractions that are not 100% enforced

Moral of the story

- Don't use C(++), if you can avoid it
  - but use a language that provides memory safety, such as Java or C#
- If you do have to use C(++), become or hire an expert
- Reading
  - *A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention*, by John Wilander and Mariam Kamkar