Software Security

Buffer Overflows

public enemy number 1

Erik Poll

Digital Security

Radboud University Nijmegen

The good news

C is a small language that is close to the hardware

- you can produce highly efficient code
- compiled code runs on raw hardware with minimal infrastructure

C is typically the programming language of choice

- for highly efficient code
- for embedded systems (which have limited capabilities)
- for system software (operating systems, device drivers,...)

The bad news : using C(++) is dangerous

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.

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

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?

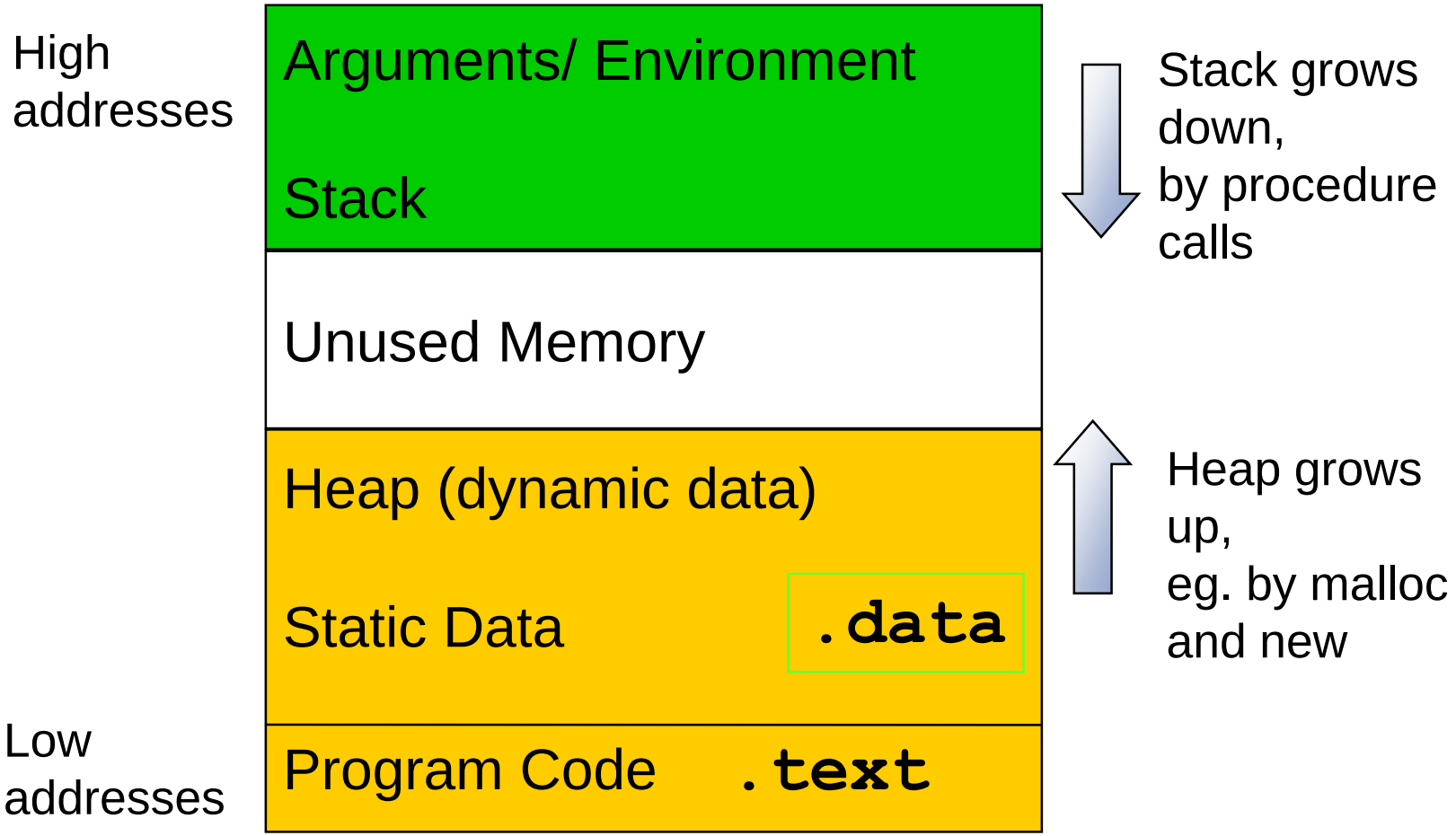
Memory management in C/C++

- A program is 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**

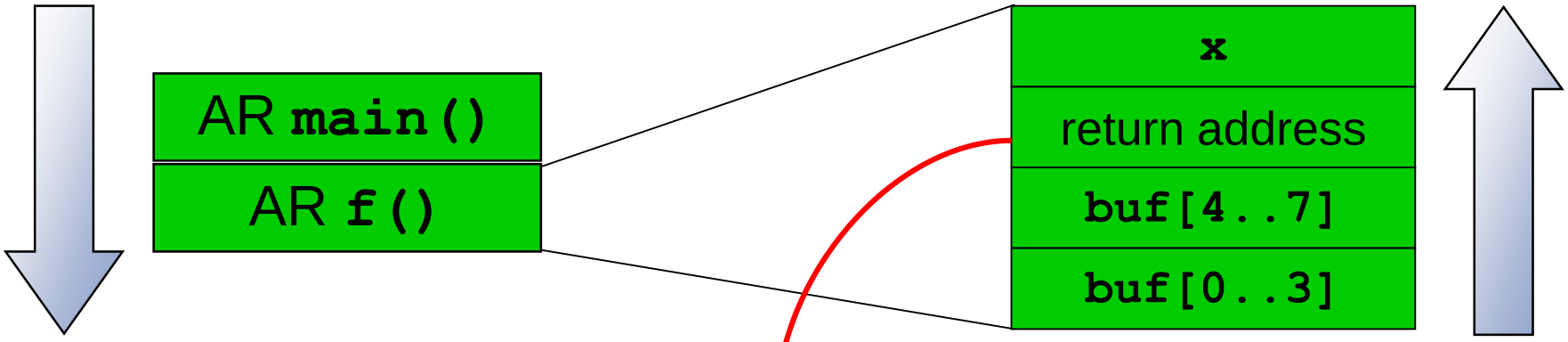
- Typical bugs:
  - Writing past the bound of an array
  - Pointer trouble
    - missing initialisation, bad pointer arithmetic, use after de-allocation (use after free), double de-allocation, failed allocation, forgotten de-allocation (memory leaks)...
- For efficiency, these bugs are not detected at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

Process memory layout



Stack overflow

The stack consists of Activation Records:



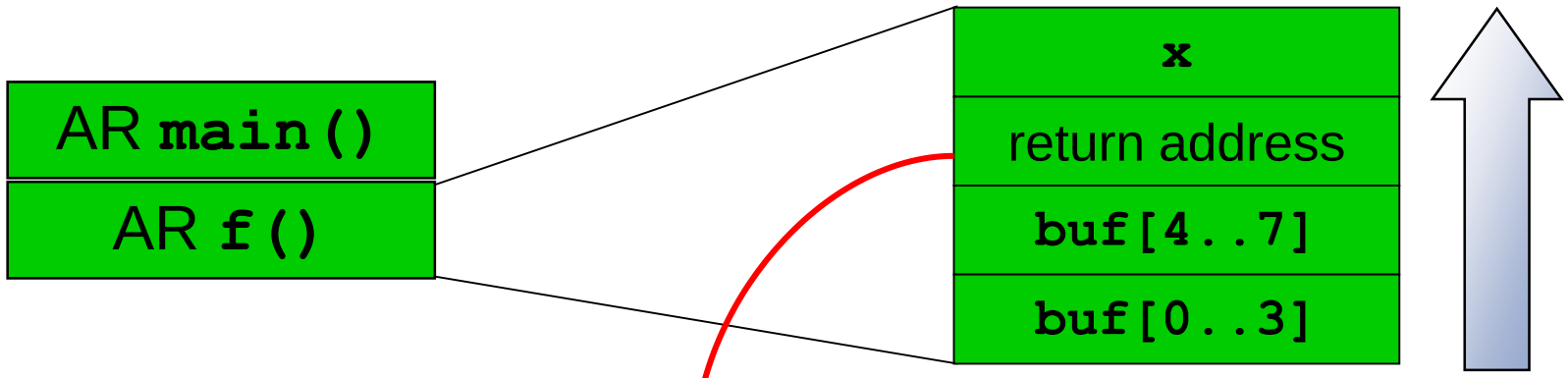
Stack grows downwards

Buffer grows upwards

```
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 ? :

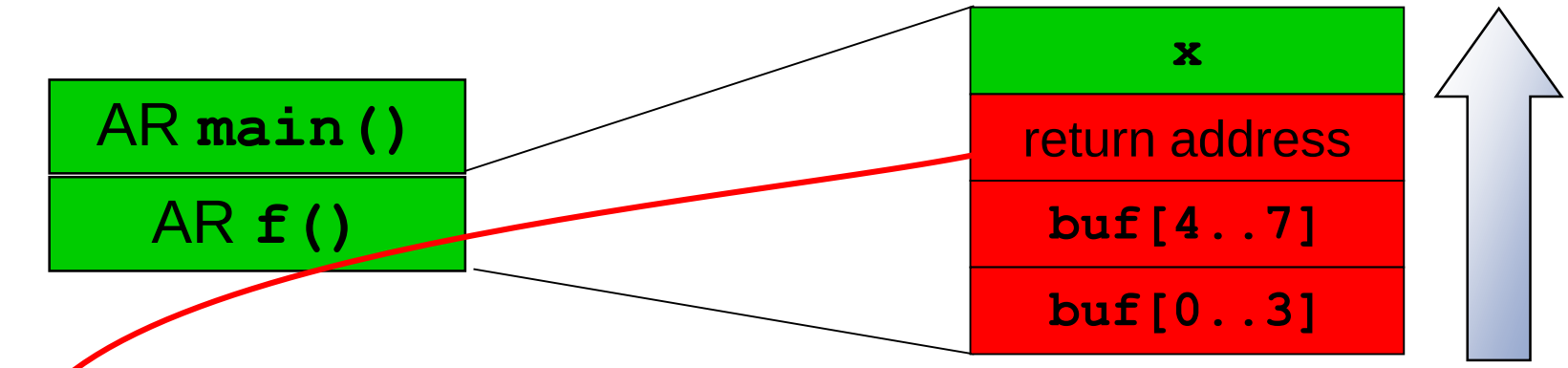



```
void f(int x) {
   char[8] buf;
   gets(buf);
}
void main() {
   f(…); …
}
void format_hard_disk(){…}
```

Buffer grows upwards

Stack overflow

What if gets() reads more than 8 bytes ?

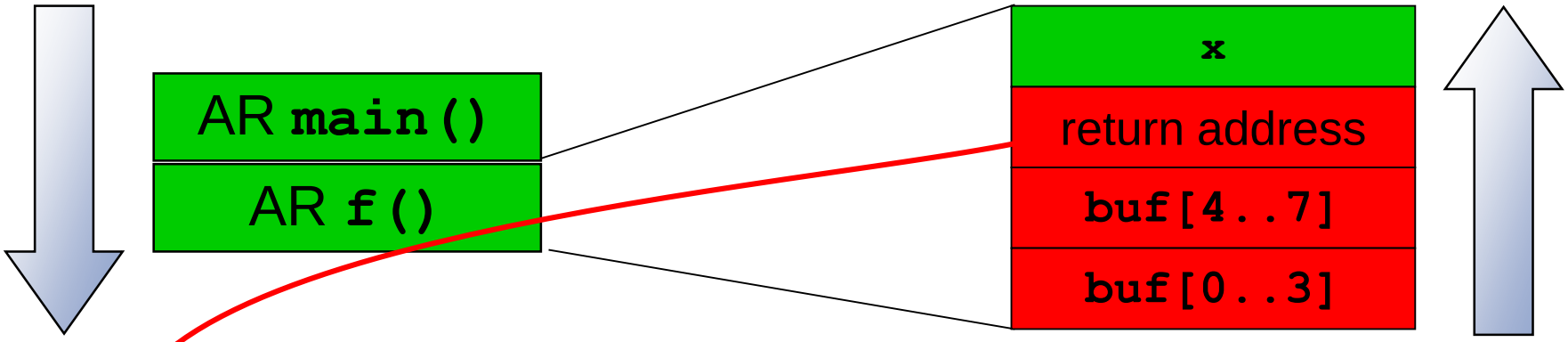



```
void f(int x) {
   char[8] buf;
   gets(buf);
}
void main() {
   f(…); …
}
void format_hard_disk(){…}
```

Buffer grows upwards

Stack overflow

What if gets() reads more than 8 bytes ?



Stack grows downwards

Buffer grows upwards

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

***never* use gets() !**

Stack overflow

- *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 stack

```
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, b1, buf2,..`**

  ...

What to attack? Fun on the heap

```
struct account {
  int  number;
  bool isSuperUser;
  char name[20];
  int  balance;
}
```

overrun **`name`** to corrupt the values of other fields in the struct

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** will not 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** will not 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(int));
     // read sizeof(int) bytes, ie. an int,
     // and store these in len
buf = malloc(len);
read(fd,buf,len); // read len bytes into buf
```

Spot the defect! (3)

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

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

len might become negative

len cast to unsigned, so 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);
```

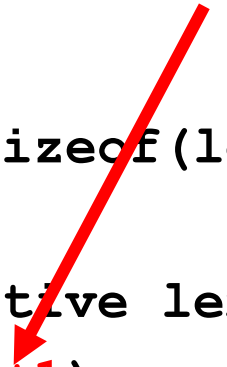
Remaining problem may be that **`buf`** is not null-terminated

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



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! (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);
```

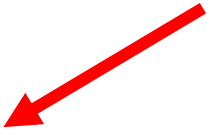
[slide from presentation by Jon Pincus]

Spot the defect! (4)

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

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

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



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

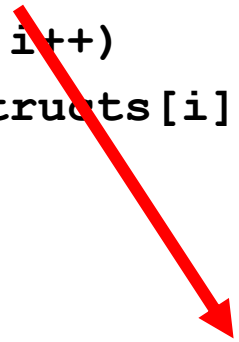
[slide from presentation by Jon Pincus]

Spot the defect! (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 is valid 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 is valid 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! (7)

```
char buff1[MAX_SIZE], buff2[MAX_SIZE];

// make sure url is valid 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);
```

What about 0-length URLs?

Is buff1 always null-terminated?

[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.

- 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?

format strings for printf

```
printf( ”j has the value %i ” , j);
     // %i to print integer value
printf( ”j has the value %x in hex ” , j);
     // %x to print 4-byte hexadecimal value
```

”j has the value %i ” is called a format string

Other printing functions also accept format strings.

Any guess what

```
    printf(”j has the value %x in hex ”);
```

does?

It will print the top 4 bytes of the stack

Leaking data from the stack

```
int main( int argc,  char** argv)
    int pincode = 1234;
    printf(argv[1]);
}
```

How can an attacker learn the value of **`pincode`** ?

Supplying **`%x%x%x`** as input will dump top 12 bytes of the stack

Leaking data from anywhere

```
printf( "str has the value %s " , str);
      // %s to print a string, ie a char*
```

Any guess what

```
printf("str has the value %s ");
```

does?

It interprets the top of the stack as a pointer (an address) and prints the string allocated in memory at that address

Of course, there might not be a string allocated at that address.

printf simply prints whatever is in memory up to the next null terminator

Corrupting data with format string attack

```
int j;

char* msg; ...

printf( ”how long is this? %n”, &j);
```

`%n` causes the number of characters printed to be written to j.

Here it will give `j` the value **`14`**

Any guess what

```
printf(”how long is this? %n”, msg”);
```

will do?

It interprets the top of the stack as an address, and writes the value 14 to it

Summary malicious format strings

Interesting inputs for the string **`str`** to attack **`printf(str)`**

- **`%x%x%x%x%x%x%x%x`**

  will print bytes from the top of the stack

- **`%s`**

  will interpret the top bytes of the stack as an address X, and then prints the string starting at that address A in memory, ie. it dumps all memory from A up to the next null terminator

- **`%n`**

  will interpret the top bytes of the stack as an address X, and then *writes* the number of characters output so far to that address

Example *really* malicious format strings

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

- **`\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

Format string attacks are easy to spot & fix:

replace **`printf(str)`**

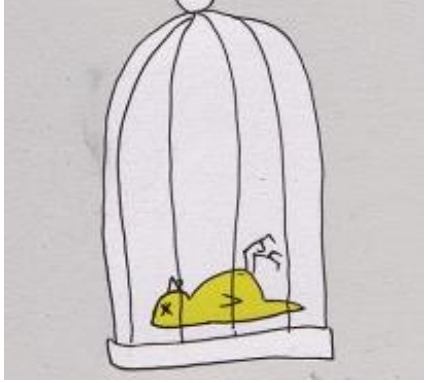
with **`printf("%s", str)`**

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 to get buffer overflows

Runtime aka dynamic countermeasures

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

*N*on-e*X*ecutable memory (NX aka W⊕X)

Distinguish

- executable memory (for storing code)
- 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

*N*on-e*X*ecutable memory (NX aka W⊕X)

Modern CPUs provide such NX bits in hardware:

- Intel calls it eXecute-Disable (XD)
- AMD calls it Enhanced Virus Protection

- Supported by many operating systems
  - MacOs X
  - Data Execution Prevention (DEP) on Windows
  - OpenBSD W^X
  - ExecShield and PAX patches in Linux

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`

Many libraries, incl. **`libc`**, provide enough operations to be Turing complete! So an attacker can do anything with such a library.

Address Space Layout Randomisation (ASLR)

- Attacker needs detailed information on memory layout

- 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

  It prevents the attacker from being able to easily predict target addresses

Dynamic countermeasures (recap)

- canaries
- non-executable memory
- address space layout randomisation (ASLR)

None of these countermeasures are perfect!

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

eg by figuring out cookie values, offset used in address randomisation, key used to encode instructions, returning to libc, etc

Moreover, they do not protect against heap overflows

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
  - http://www.securityfocus.com/bid/8522/info
- 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
- at compilation time
- when testing
- when executing code

to prevent, migitate, or detect buffer overflows problems

Prevention

- Don’t use C or C++

You can write insecure code in any programming language, but some languages make it easier to write insecure programs than others

C(++) programmer is like trapeze artist without safety net

Prevention

Many languages are not prone to memory errors like C(++).

These are often called safe languages, because they offer memory-safety and sometimes also type-safety.

Examples: Java, C#

Typical characteristics of safe languages:

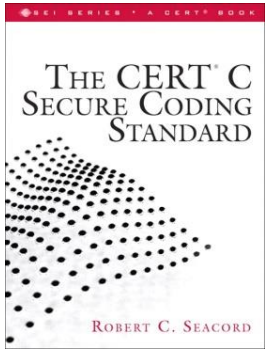
- checking array bounds
- checking for null values
- default initialisation
- no pointer arithmetic
- no dynamic memory management with malloc() and free(), but automatic memory management using garbage collector
- strong type checking
- exception on integer overflow
- more precisely defined semantics

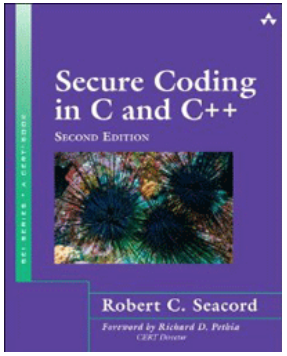
Prevention

- Better programmer awareness & training

  Eg read – and make other people read –

  - C(++) Secure Coding Standards by CERT
    https://www.securecoding.cert.orgps://www.securecoding.cert.org
  - Building Secure Software, J. Viega & G. McGraw, 2002
  - Writing Secure Code, M. Howard & D. LeBlanc, 2002
  - Secure programming for Linux and UNIX HOWTO, D. Wheeler,





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**

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

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

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 overflow weaknesses is **fuzz testing**

- send *random, very long* inputs, to an application
- if the application crashes, with a segmentation fault (segfault), it contains buffer overflows

The nice thing is that this is easy to automate!

Code review & Static Analysis

- Code reviews
  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,
  - PREfix, PREfast by Microsoft

  plus other commercial tools
  Coverity, PolySpace, Klockwork, CodeWizard, Cqual, Fortify
  ....

(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

Summary

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

  or: 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

Moral of the story

- Don’t use C(++), if you can avoid it
  - but use a safer language that provides memory safety

- If you do have to use C(++), become or hire an expert

Want to read more?

- V. van der Veen, N. dutt-Sharma, L. Cavallaro, H. Bos
  Memory Errors: The Past, The Present and the Future

  Nice historical overview of attacks, defences, and trends

- Y. Younan,W. Joosen, F. Piessens,
  Code injection in C and C++:
  a survey of vulnerabilities and countermeasures

  More details on workings of buffer overflows and
  very comprehensive overview of countermeasures