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

3. What can 'the platform' do about it?
   - ie. the compiler, system libraries, hardware, OS

4. What can the programmer 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 an attacker can cause this,
anything that the attacker wants may happen

- If we are *lucky*, we will get a SEGMENTATION FAULT

  Why is this good luck and not bad luck?
  - It stops the attack, though it might still cause DoS
  - Without SEGFAULT, then 1) we don't know there's a problem and 2) impact could be worse (eg. remote code execution)
- 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.
  - regrettably, people often choose performance over security
  - 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.

Tony Hoare on design principles of ALGOL 60

In his Turing Award lecture

"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 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]

The buffer overflow problem

- The most common security problem in (machine code compiled from) C/C++
  - ever since the first Morris Worm in 1988
- Typically, attackers can get full control, incl. remote code execution in e.g.
  - services accessible over the network, eg. mail servers, web servers, web browser, wireless network driver, ...
  - applications acting on downloaded files or email attachments
  - high privilige processes on the OS (eg. setuid binaries on Linux, as SYSTEM services on Windows)
  - embedded software in routers, phones, cars, ...
- Ongoing arms race of attacks & defences: attacks are getting cleverer, defeating ever better countermeasures

Other memory corruption errors

Other common memory-related bugs in C/C++ code:

errors with pointers and with dynamic memory (the heap)

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

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

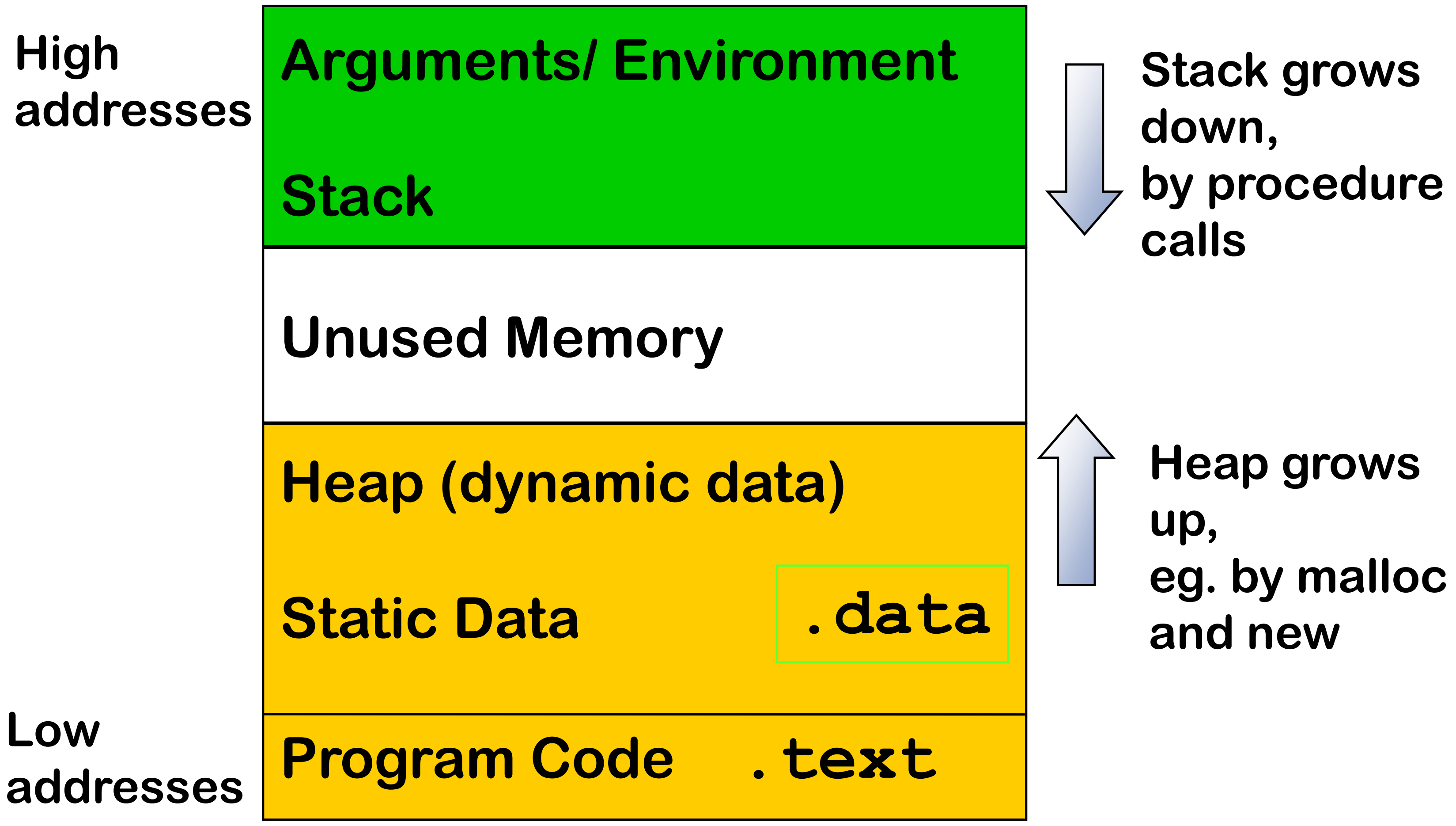
Who ever had their programs crashing?

- 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:
  - dangling pointers, use-after-free, double-free, forgotten de-allocation (memory leaks), failed allocation, flawed pointer arithmetic

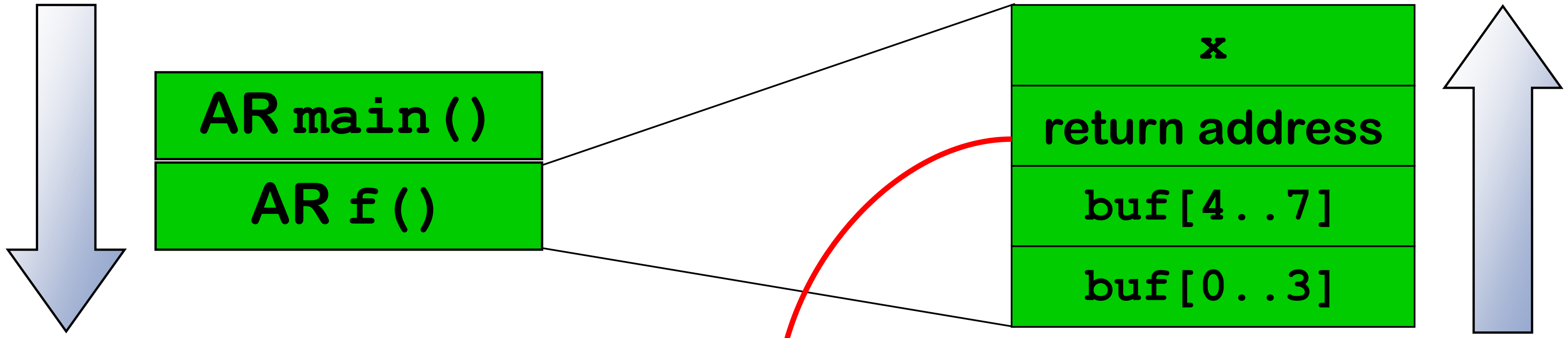
How does classic buffer overflow work?
aka smashing the stack

Process memory layout



Stack layout

The stack consists of Activation Records:



Stack grows downwards

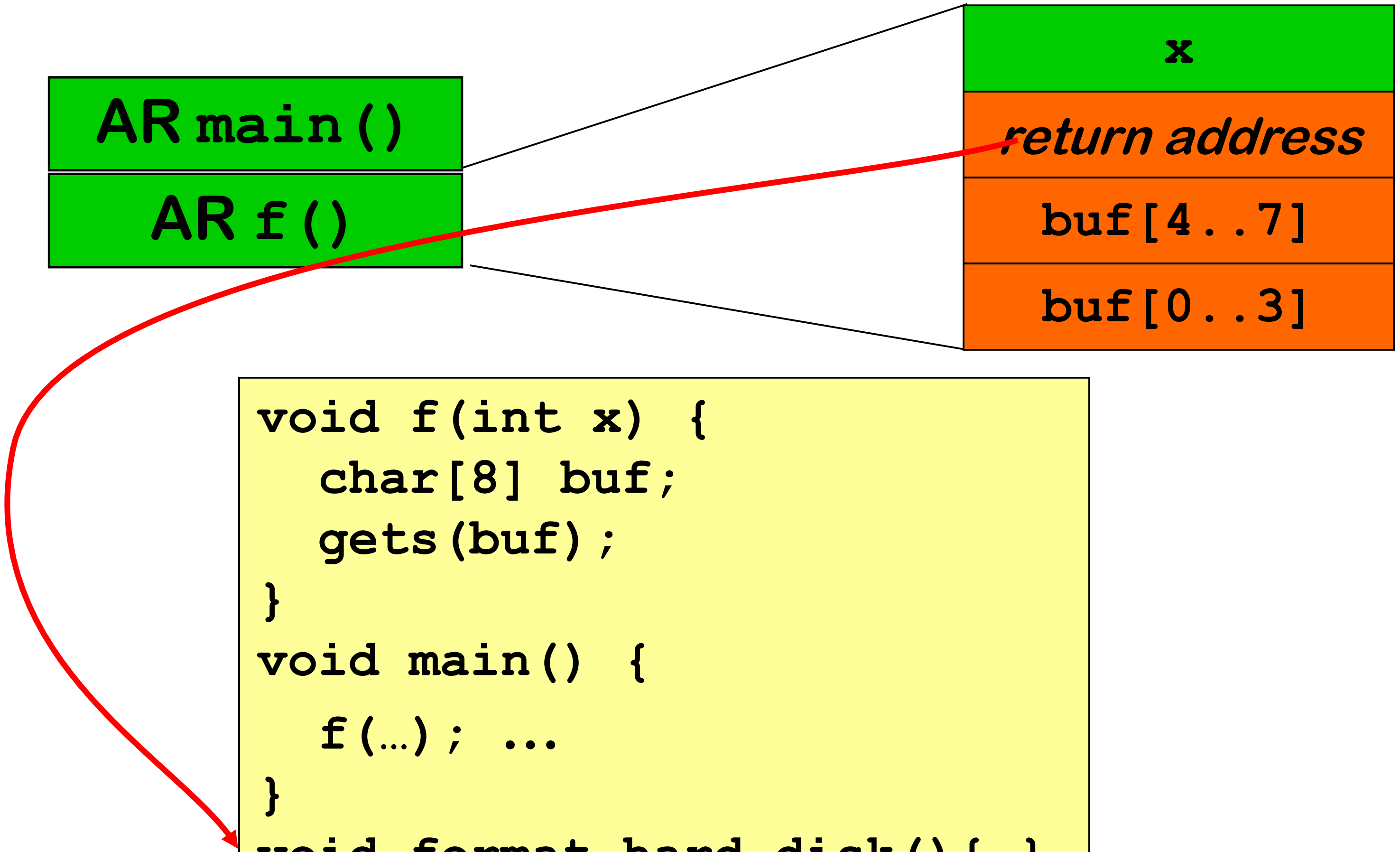
Buffer grows upwards

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

Stack overflow attack (1)

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

Attacker can jump to abitrary point in the code!

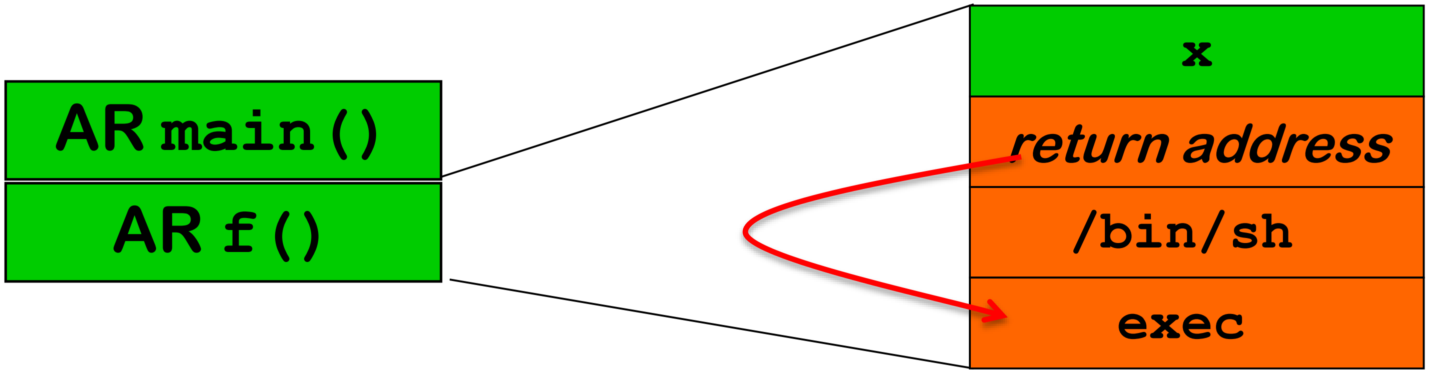


```
void f(int x) {
   char[8] buf;
   gets(buf);
}
void main() {
   f(…); ...
}
void format_hard_disk(){…}
```

Stack overflow attack (2)

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

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

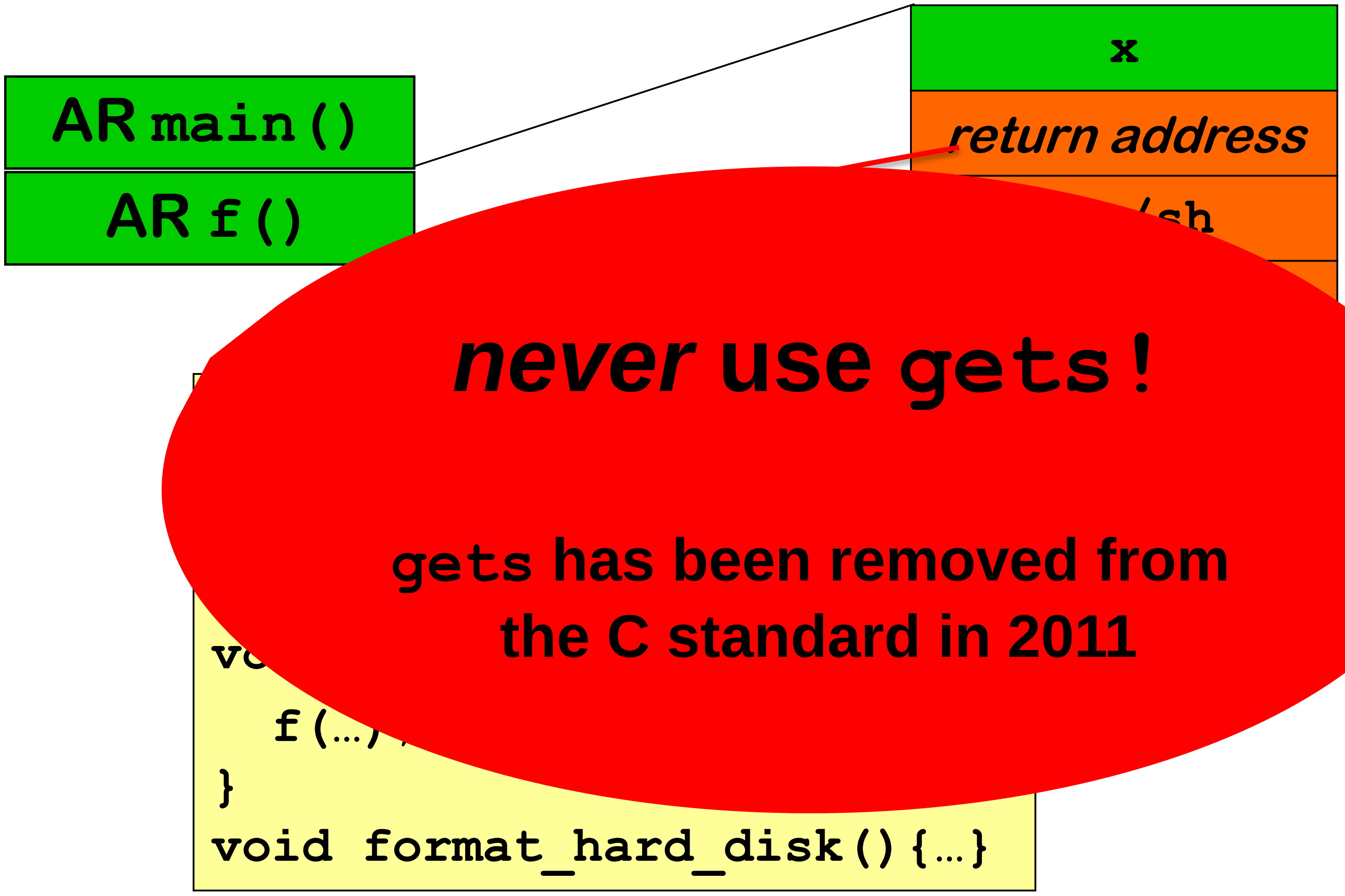


```
void f(int x) {
   char[8] buf;
   gets(buf);
}
void main() {
   f(…); ...
}
void format_hard_disk(){…}
```

Stack overflow attack (2)

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

Attacker can jump to <u>his own code</u> (aka shell code)



never **use** `gets` **!**

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

code injection vs code reuse

The two attack scenarios in these examples

- **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]

- **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, he 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.`

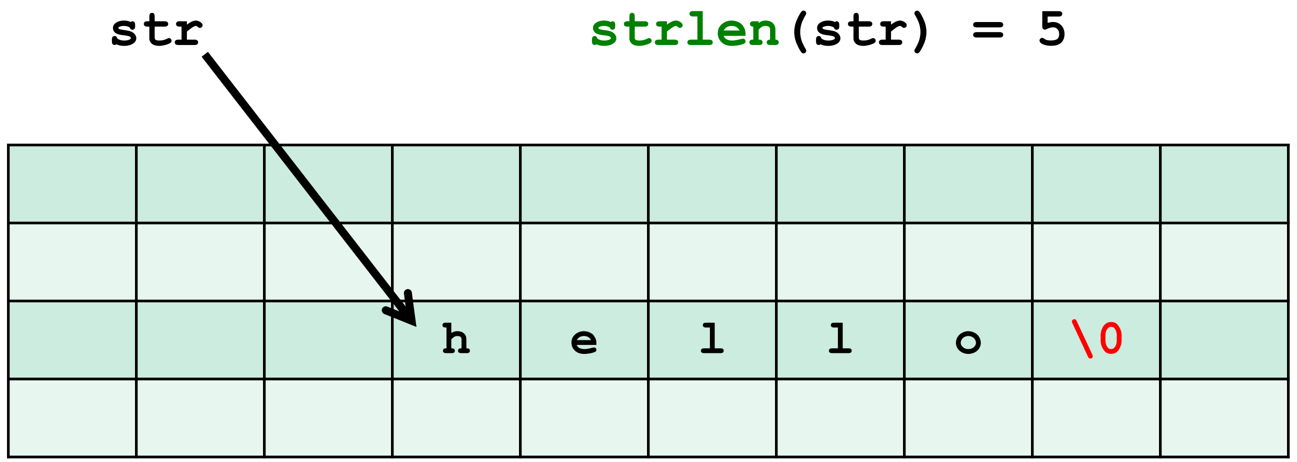
Which field(s) can be corrupted depends on the order of the fields in memory, which the compiler is free to choose.

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! (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

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];

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)
```

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! (3a)

```
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! (3b)

```
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 will ignore this for now.

Spot the defect! (3c)

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

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

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

Spot the defect! (3d)

```
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 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! (4)

```
char buffer[100];

void* f(int start)
int end = start + 10;
if (end < start) return SOME_ERROR;
if (end > 100) return SOME_ERROR
for int i=start; i < end; i++ ){
   buffer[i] = 'a';
}
```

Integer overflow is *undefined behaviour*,

- so we cannot assume that overflow produces a negative number.
- so the compiler can simply assume it does not happen.
- In particular, compiler may assume that `x+100 < x` is always false!

Spot the defect! (4)

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

The CodeRed worm exploited such an mismatch. Code written under the assumption that 1 character was 1 byte contained many buffer overflows after the move from 1 byte ASCI to 2 byte Unicode characters

[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 Visual Studio C++ compiler adds check to prevent this

Spot the defect! (7)

```
char buf1[MAX_SIZE], buf2[MAX_SIZE];
// make sure url a valid and fits in buf1 and buf2:
if (! isValid(url)) return;
if (strlen(url) > MAX_SIZE - 1) return;

// Now copy url up to first '/' into buf1
out = buf1;
do {
  // skip spaces
   if (*url != ' ') *out++ = *url;
} while (*url++ != '/');
strcpy(buf2, buf1);
...
```

[slide from presentation by Jon Pincus]

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

```
char buf1[MAX_SIZE], buf2[MAX_SIZE];
// make sure url a valid and fits in buf1 and buf2:
if (! isValid(url)) return;
if (strlen(url) > MAX_SIZE - 1) return;

// Now copy url up to first '/' into buf1
out = buf1;
do {
  // skip spaces
   if (*url != ' ') *out++ = *url;
} while (*url++ != '/');
strcpy(buf2, buf1);
...
```

length up to the first null

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

[slide from presentation by Jon Pincus]

Spot the defect! (8) undefinedness strikes again

```
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.
So what this code does if `tun` is NULL is undefined.

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

Standard compilers (gcc, CLang) actually 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! (9)

```
#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 to illegally read or write memory

- 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 *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 bytes from stack

  This can leak sensitive data

  Eg `printf(s);`

  dumps 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%x%x%x ...
  ```

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

  This allows the attacker to corrupt the stack

- With a carefully constructed format string an attacker may be able to read or write any memory address on the heap or stack

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

Many format string problems easy to spot

- Some format string attacks are easy to prevent:

  eg replace `printf(str)`

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

- The compiler or a 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);
  ```

  But if the format string is not a compile-time constant, we cannot tell at compile time.

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

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^X)
3. address space layout randomization (ASLR)
4. control-flow integrity (CFI)
5. pointer encryption
6. execution-aware memory protection

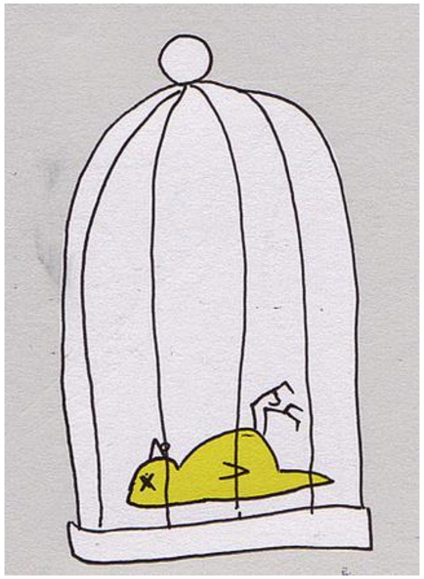
$\left.\right\}$ now standard on many platforms (items 1–3)

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

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

1. stack canaries

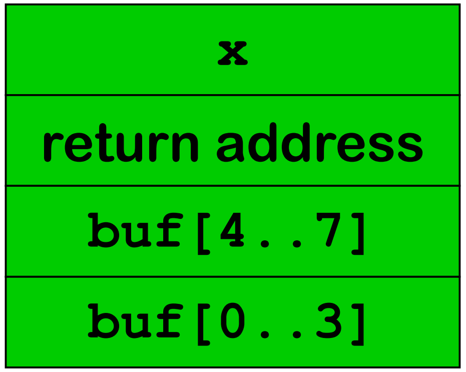
- A dummy value - stack canary or cookie - is written on the stack in front of the return address and checked when function returns
- A careless stack overflow will overwrite the canary, which can then be detected
- introduced in as StackGuard in gcc

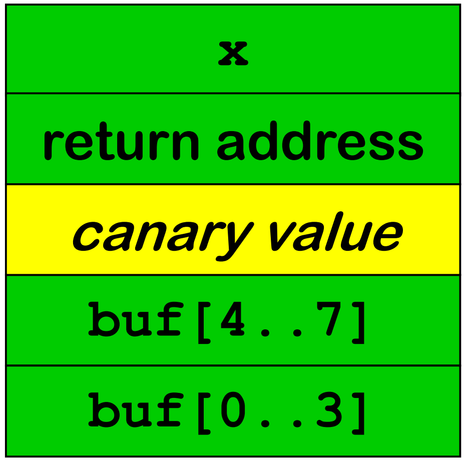


stack canaries

Stack without canary

Stack with canary





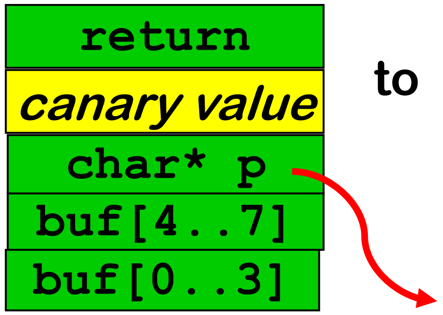
Further improvements

- More variation in canary values: fixed values hardcoded in binary vs random values chosen for each execution vs ...
- 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

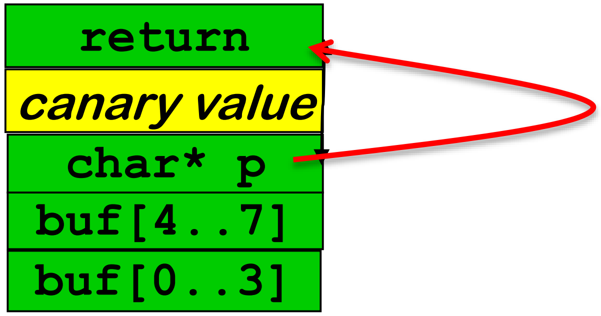
A careful attacker can still defeat canaries, by

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

eg changing



to



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 subtle 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 information 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 information 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

- An attacker can still analyse the memory layout of say a Windows process on his own laptop, but he will have to determine these offsets used on the victim’s machine to carry out an attack there.

- NB security by obscurity, despite its bad reputation, is a really great defense mechanism to annoy attackers!

3. Non-eXecutable memory (NX , W⊕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 are no longer possible,
but code *reuse* attacks 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 many possibilities for attacker, eg. `system()`, `exec()`,
which provide the attacker with any functionality he wants...

re TURN oriented programMing (ROP)

Next stage in evolution of attacks, as people removed or protected dangerous library calls like `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 bas,

```
        ... ; ... ; ... ; 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
- An ROP compiler can then translate any code to a string of these gadgets

4. Control Flow Integrity (CFI)

Return-to-libc or ROP attacks give rise to unusual control flow jumps between code blocks

Eg a function `f()` never calls library routine `exec()`, and `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 such attacks

- 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.