Software Security

Memory corruption

public enemy number 1

Erik Poll

Digital Security

Radboud University Nijmegen

Security in the development lifecycle

week 3: exercise
Static analysis
PREfast

the deve

week4: group project
fuzzing afl
memory sanitizers Asan, Msan

week 3: exercise
Static analysis
PREfast

the deve

week4: group project
fuzzing afl
memory sanitizers Asan, Msan

More foundational improvements later:
- Safe(r) programming languages (week 5)
- LangSec for safer input languages (week 6)

Overview (next 2 weeks)

1. How do memory corruption flaws work?
2. What can be the impact?
3. How can we spot such problems in C(++) code?

   Next weeks: tool-support for this

   - SAST: PREfast individual project
   - DAST: Fuzzing group project
4. What can 'the platform' do about it?

   ie. the compiler, system libraries, hardware, OS, ..
5. What can the programmer do about it?

Reading material

- SoK article: ‘Eternal War in Memory’ S&P 2013
  - Excl. Section VII.
  - This article is quite dense. You are not expected to be able to reproduce or remember all the discussion here. It’s good enough if you can follow the article, with a steady supply of coffee while googling if the terminology is not clear.

- Chapter 3.1 & 3.2 in lecture notes on memory-safety

  We’ll revisit safe programming languages – incl. other safety features – and rest of Chapter 3 in later lecture

Essence of the problem

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

```
char buffer[4];
```

What happens if the statement below is executed?

```
buffer[4] = 'a';
```

Essence of the problem

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

```
char buffer[4];
```

What happens if the statement below is executed?

```
buffer[4] = 'a';
```

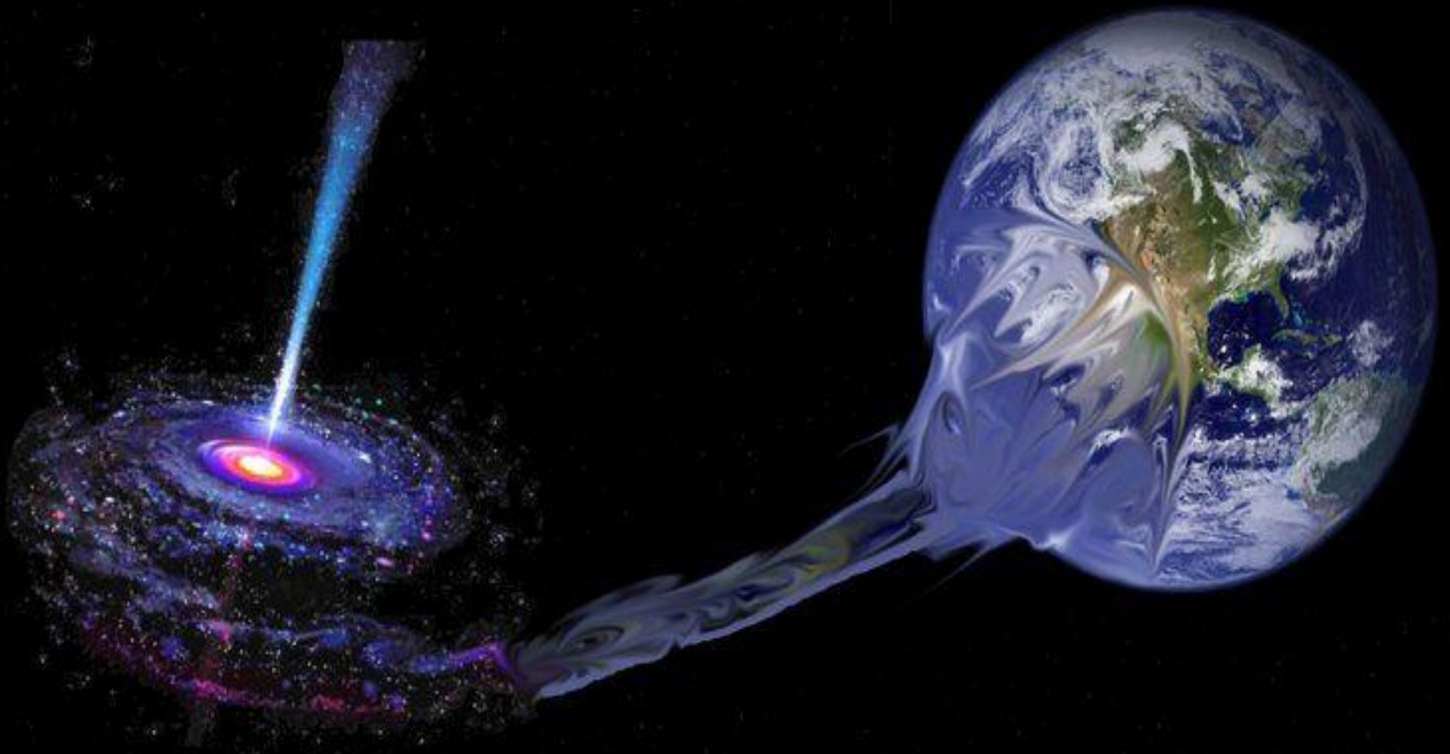
We don’t know!

This is defined to be undefined

ANYTHING can happen

UNDEFINED behaviour: anything can happen

UNDEFINED behaviour: anything can happen



UNDEFINED behaviour: anything can happen

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

```
char buffer[4];
```

What happens if the statement below is executed?

```
buffer[4] = 'a';
```

If the attacker can control the value `'a'`

then *anything that the attacker* wants may happen

- If you are *lucky* : a `SEGMENTATION FAULT`
  - and you'll know that something went wrong
- If you are *unlucky* : remote code execution (RCE)
  - and you *won't* know

UNDEFINED behaviour: anything can happen

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

```
char buffer[4];
```

What happens if the statement below is executed?

```
buffer[4] = 'a';
```

A compiler could remove the statement above,
ie. do *nothing*

- This would be correct compilation by the C standard because *anything* includes *nothing*
- Compiler may actually do this (as part of optimalisation) and this has caused security problems; examples later & in the lecture notes.

Solution to this problem

Regrettably, people often choose performance over security

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

- Fortunately, Perl, Python, Java, C#, PHP, Javascript, and Visual Basic *do* check array bounds

Solution to this problem

- Check array bounds at runtime
  - Algol 60 proposed this back in 1960!

Regrettably, people often choose performance over security

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

- Fortunately, Perl, Python, Java, C#, PHP, Javascript, and Visual Basic *do* check array bounds

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.

Regrettably, people often choose performance over security

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

- Fortunately, Perl, Python, Java, C#, PHP, Javascript, and Visual Basic *do* check array bounds

Solution to this problem

- Check array bounds at runtime
  - Algol 60 proposed this back in 1960!

- Unfortunately, C and C++ have not adopted this solution.
  - *Why?*

Regrettably, people often choose performance over security

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

- Fortunately, Perl, Python, Java, C#, PHP, Javascript, and Visual Basic *do* check array bounds

Solution to this problem

- Check array bounds at runtime
  - Algol 60 proposed this back in 1960!

- Unfortunately, C and C++ have not adopted this solution.
  - *Why?*
  - For EFFICIENCY

  Regrettably, people often choose performance over security

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

- Fortunately, Perl, Python, Java, C#, PHP, Javascript, and Visual Basic *do* check array bounds

Tony Hoare on design principles of ALGOL 60

In his Turing Award lecture in 1980

"The first principle was *security* : ... every subscript was checked at run time against both the upper and the lower declared bounds of the array. Many years later we asked our customers whether they wished an option to switch off these checks in the interests of efficiency. Unanimously, they urged us not to - they knew how frequently subscript errors occur on production runs where failure to detect them could be disastrous.

I note with fear and horror that even in 1980, language designers and users have not learned this lesson. In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law."

[C.A.R. Hoare, The Emperor's Old Clothes, Communications of the ACM, 1980]

Buffer overflow

- The most common security problem in (machine code compiled from) C and C++
  - ever since the first Morris Worm in 1988
- Check out CVEs mentioning buffer (or buffer%20overflow)

  https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=buffer
- Ongoing arms race of attacks & defences:
  attacks are getting cleverer,
  defeating ever better countermeasures

More memory corruption problems

Errors with pointers and with dynamic memory (the heap)

More memory corruption problems

Errors with pointers and with dynamic memory (the heap)

- ***Have you ever written a C(++) program that uses pointers?***

More memory corruption problems

Errors with pointers and with dynamic memory (the heap)

- ***Have you ever written a C(++) program that uses pointers?***
- ***Have you ever had such a program crashing?***

More memory corruption problems

Errors with pointers and with dynamic memory (the heap)

- ***Have you ever written a C(++) program that uses pointers?***
- ***Have you ever had such a program crashing?***
- ***Have you even written a C(++) program that uses dynamic memory, ie.*** **`malloc()`** ***and*** **`free()`** ***?***

More memory corruption problems

Errors with pointers and with dynamic memory (the heap)

- ***Have you ever written a C(++) program that uses pointers?***
- ***Have you ever had such a program crashing?***
- ***Have you even written a C(++) program that uses dynamic memory, ie. `malloc()` and `free()` ?***
- ***Have you ever had such a program crashing?***

More memory corruption problems

Errors with pointers and with dynamic memory (the heap)

- ***Have you ever written a C(++) program that uses pointers?***
- ***Have you ever had such a program crashing?***
- ***Have you even written a C(++) program that uses dynamic memory, ie. `malloc()` and `free()` ?***
- ***Have you ever had such a program crashing?***

In C/C++, the programmer is responsible for memory management, and this is very error-prone

- **Technical term: C and C++ do not offer memory-safety**

  (see lecture notes, §3.1-3.2)

Memory corruption problems

Typical causes

- access outside array bounds
- buggy pointer arithmetic
- dereferencing null pointer
- using a dangling pointer or stale pointer, caused by
  - use-after-free
  - double-free
- forgetting to check for failures in allocation
- forgetting to de-allocate, aka memory leaks
  - not a memory *corruption* issue,
    but a memory *availability* issue

Spot all (potential) defects

```
...
void f (){
  char* buf, buf1;
  buf = malloc(100);
  buf[0] = 'a';
 ...
  free(buf1);
  buf[0] = 'b';
 ...
  free(buf);
  buf[0] = 'c';
  buf1 = malloc(100);
  buf[0] = 'd'
}
```

Spot all (potential) defects

```
...
void f (){
  char* buf, buf1;
  buf = malloc(100);
  buf[0] = 'a';
   ...
  free(buf1);
  buf[0] = 'b';
   ...
  free(buf);
  buf[0] = 'c';
  buf1 = malloc(100);
  buf[0] = 'd'
}
```

use-after-free; buf[0] points to de-allocated memory

Spot all (potential) defects

```
...
void f (){
  char* buf, buf1;
  buf = malloc(100);
  buf[0] = 'a';
 ...
  free(buf1);
  buf[0] = 'b';
 ...
  free(buf);
  buf[0] = 'c';
  buf1 = malloc(100);
  buf[0] = 'd'
}
```

use-after-free; buf[0] points to de-allocated memory

use-after-free, but now buf[0] might point to memory that has now been re-allocated

Spot all (potential) defects

```
...
void f (){
  char* buf, buf1;
  buf = malloc(100);
  buf[0] = 'a';
  ...
  free(buf1);
  buf[0] = 'b';
  ...
  free(buf);
  buf[0] = 'c';
  buf1 = malloc(100);
  buf[0] = 'd'
}
```

possible null dereference (if malloc failed)

use-after-free; buf[0] points to de-allocated memory

use-after-free, but now buf[0] might point to memory that has now been re-allocated

Spot all (potential) defects

```
...
void f (){
  char* buf, buf1;
  buf = malloc(100);
  buf[0] = 'a';
   ...
  free(buf1);
  buf[0] = 'b';
   ...
  free(buf);
  buf[0] = 'c';
  buf1 = malloc(100);
  buf[0] = 'd'
}
```

- `buf[0] = 'a';` — possible null dereference (if malloc failed)
- `buf[0] = 'b';` — potential use-after-free if buf & buf1 are aliased
- `buf[0] = 'c';` — use-after-free; buf[0] points to de-allocated memory
- `buf[0] = 'd'` — use-after-free, but now buf[0] might point to memory that has now been re-allocated

Spot all (potential) defects

```
...
void f (){
  char* buf, buf1;
  buf = malloc(100);
  buf[0] = 'a';
  ...
  free(buf1);
  buf[0] = 'b';
  ...
  free(buf);
  buf[0] = 'c';
  buf1 = malloc(100);
  buf[0] = 'd'
}
```

possible null dereference (if malloc failed)

potential use-after-free if buf & buf1 are aliased

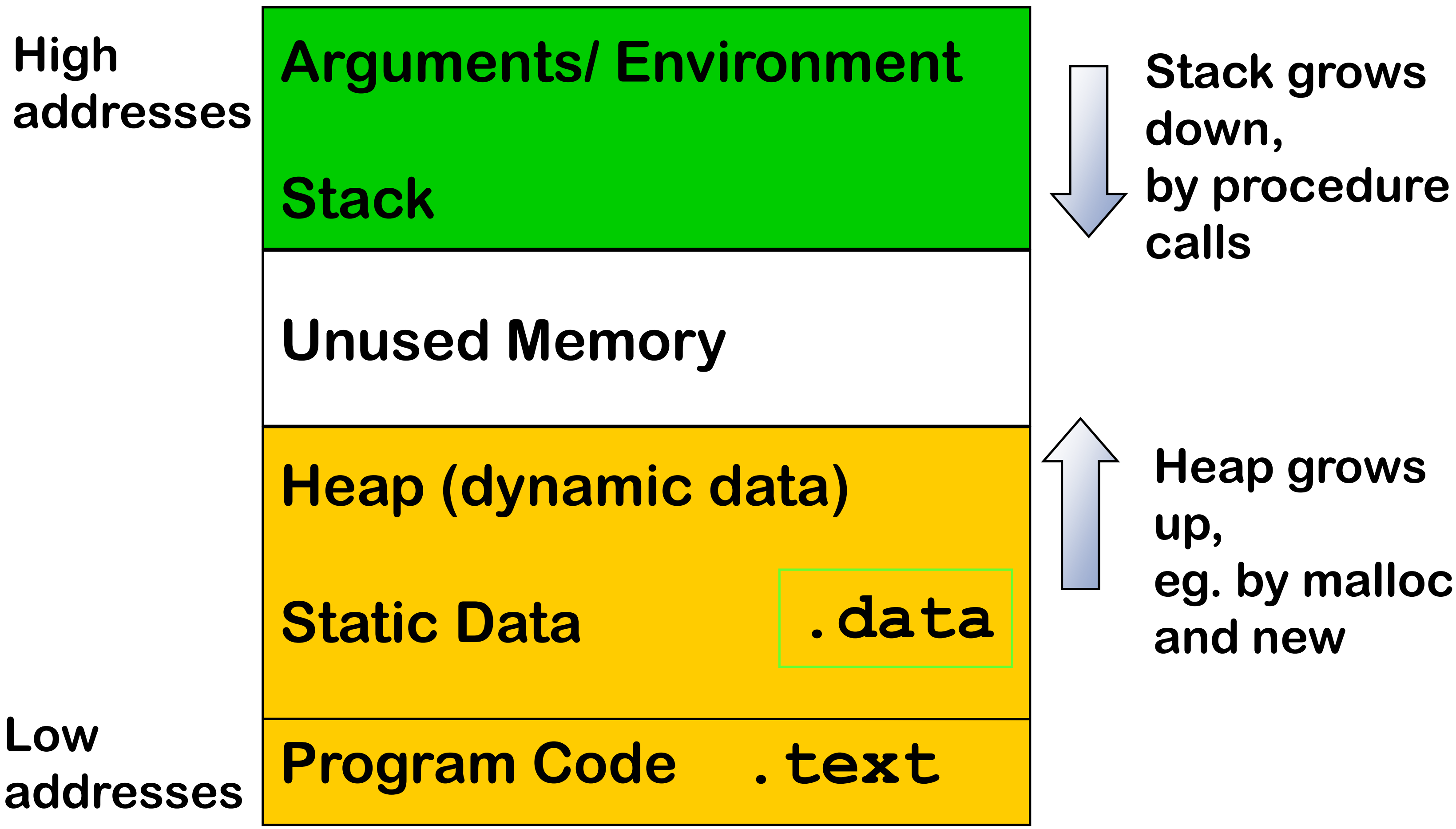
use-after-free; buf[0] points to de-allocated memory

memory leak; pointer `buf1` to this memory is lost & memory is never freed

use-after-free, but now buf[0] might point to memory that has now been re-allocated

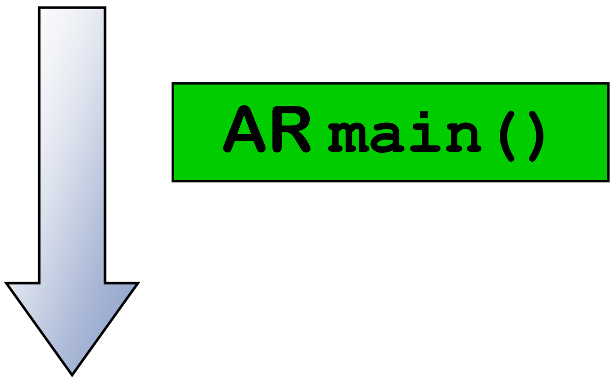
How does classic buffer overflow work?
aka smashing the stack

Process memory layout



Stack layout

The stack consists of Activation Records:

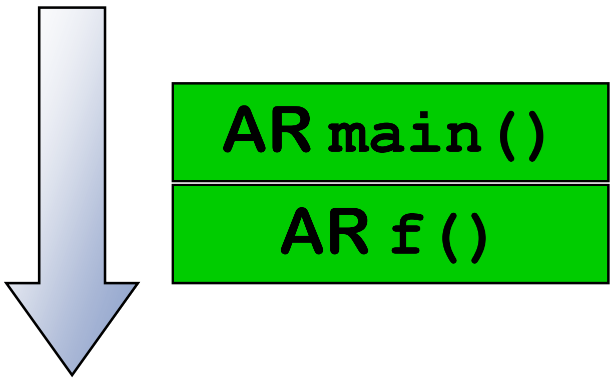


Stack grows downwards

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

Stack layout

The stack consists of Activation Records:

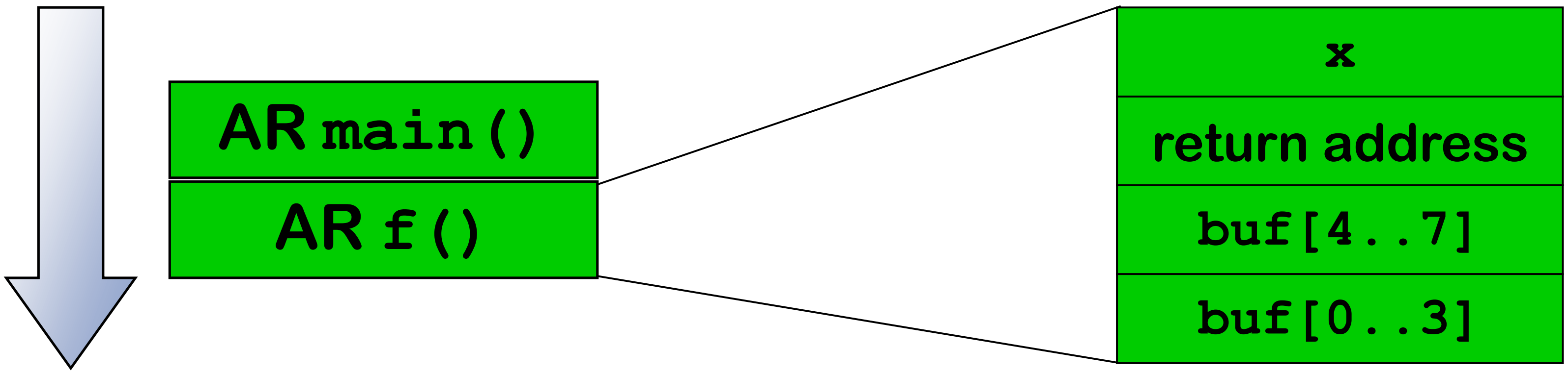


Stack grows downwards

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

Stack layout

The stack consists of Activation Records:

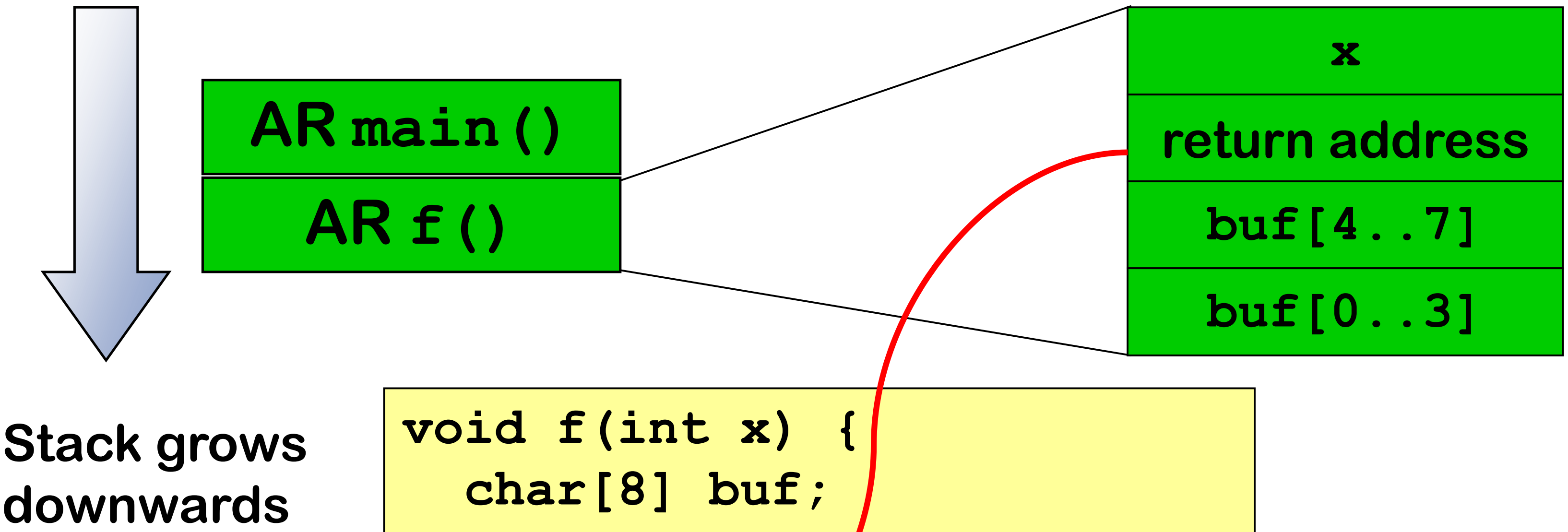


Stack grows downwards

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

Stack layout

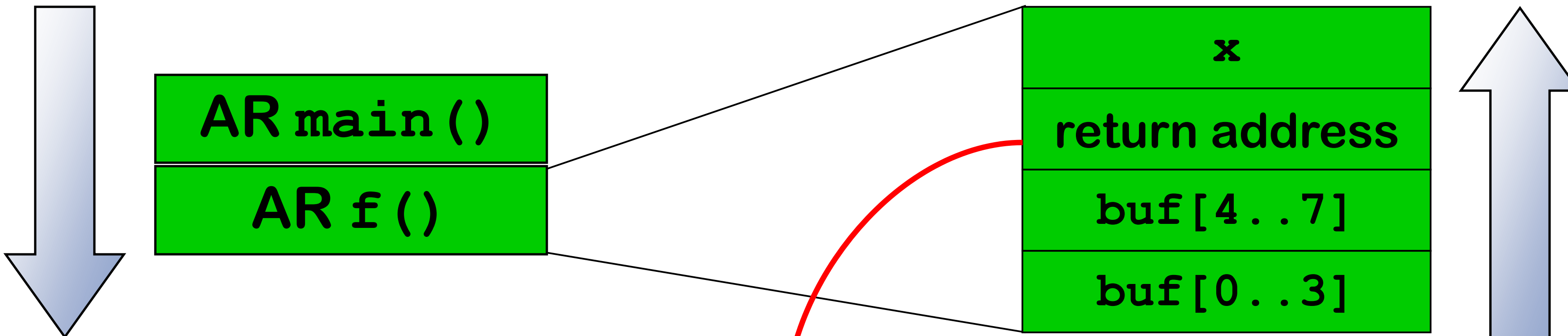
The stack consists of Activation Records:




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

Stack layout

The stack consists of Activation Records:



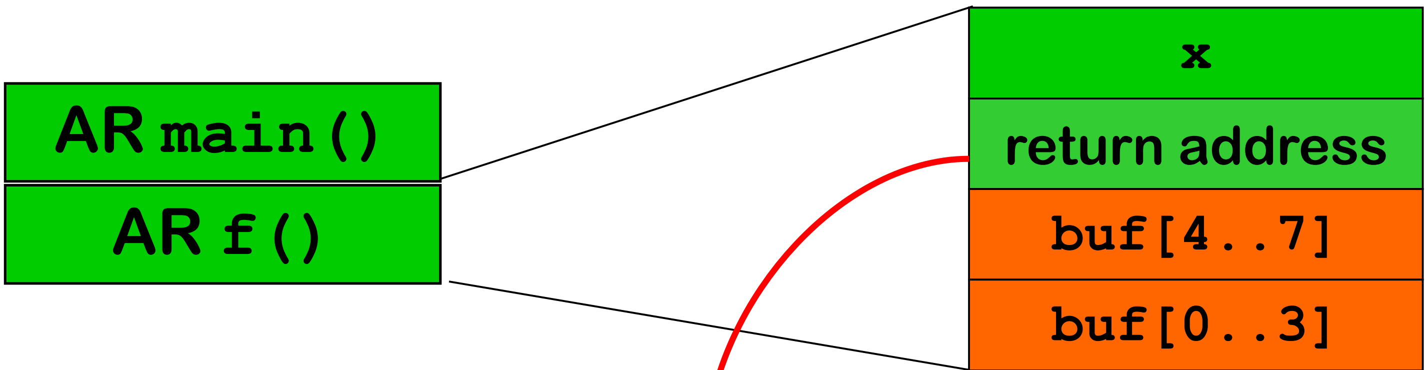
Stack grows downwards

Buffer grows upwards

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

Stack overflow attack - case 1

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

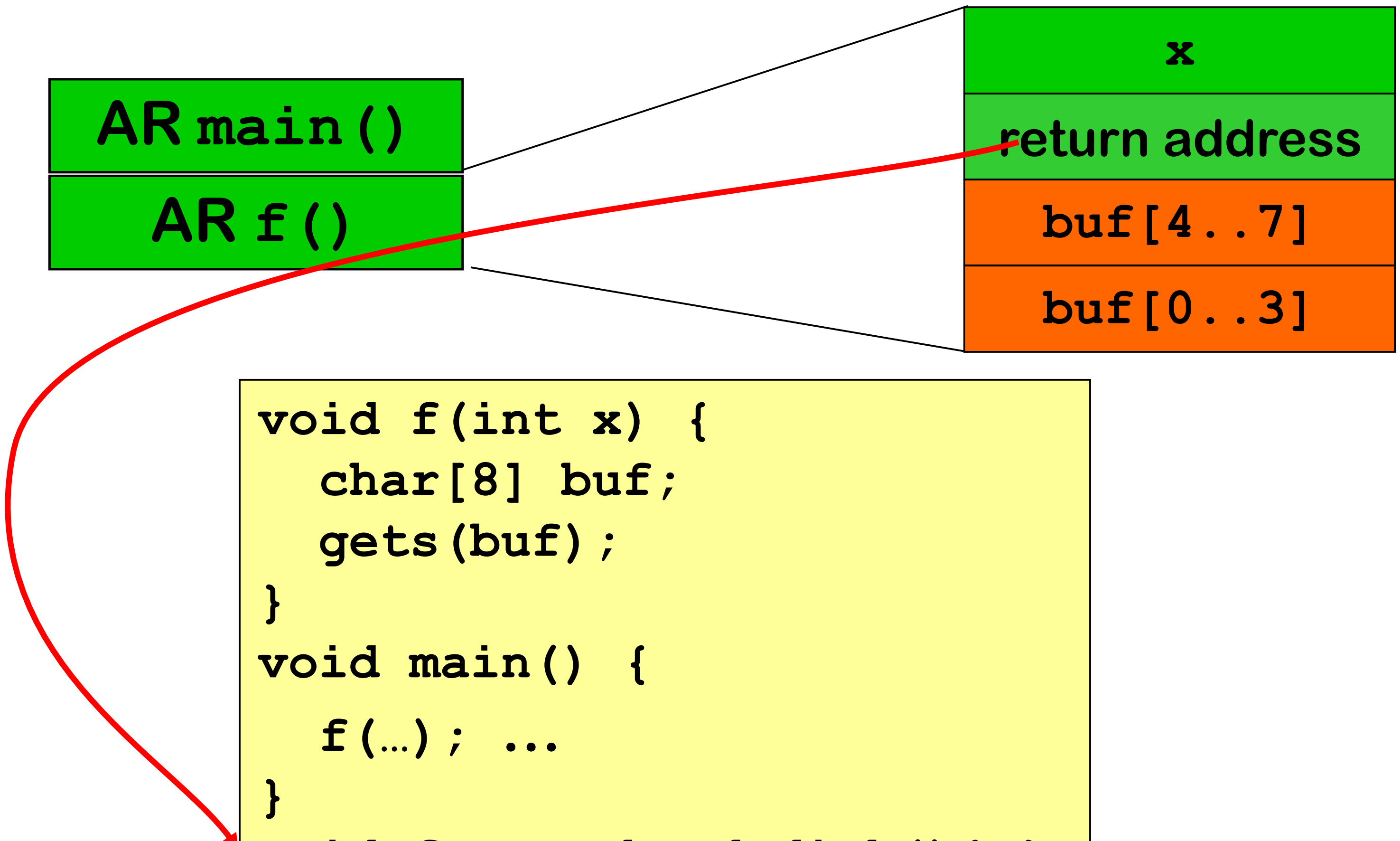



```
void f(int x) {
   char[8] buf;
   gets(buf);
}
void main() {
   f(…); …
}
void format_hard_disk(){…}
```

Stack overflow attack - case 1

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

Attacker can jump to <u>arbitrary point in the code</u>!

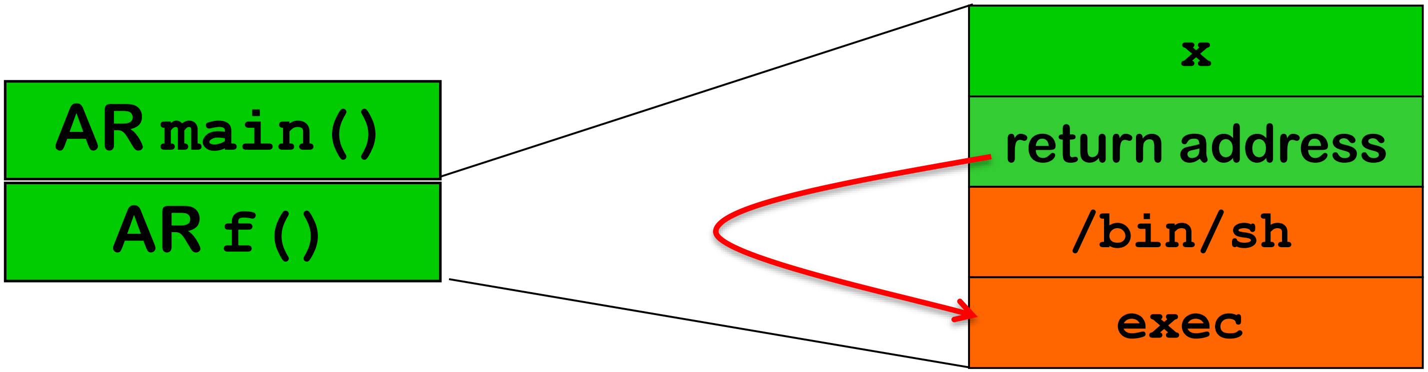



```
void f(int x) {
   char[8] buf;
   gets(buf);
}
void main() {
   f(…); ...
}
void format_hard_disk(){…}
```

Stack overflow attack - case 2

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

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

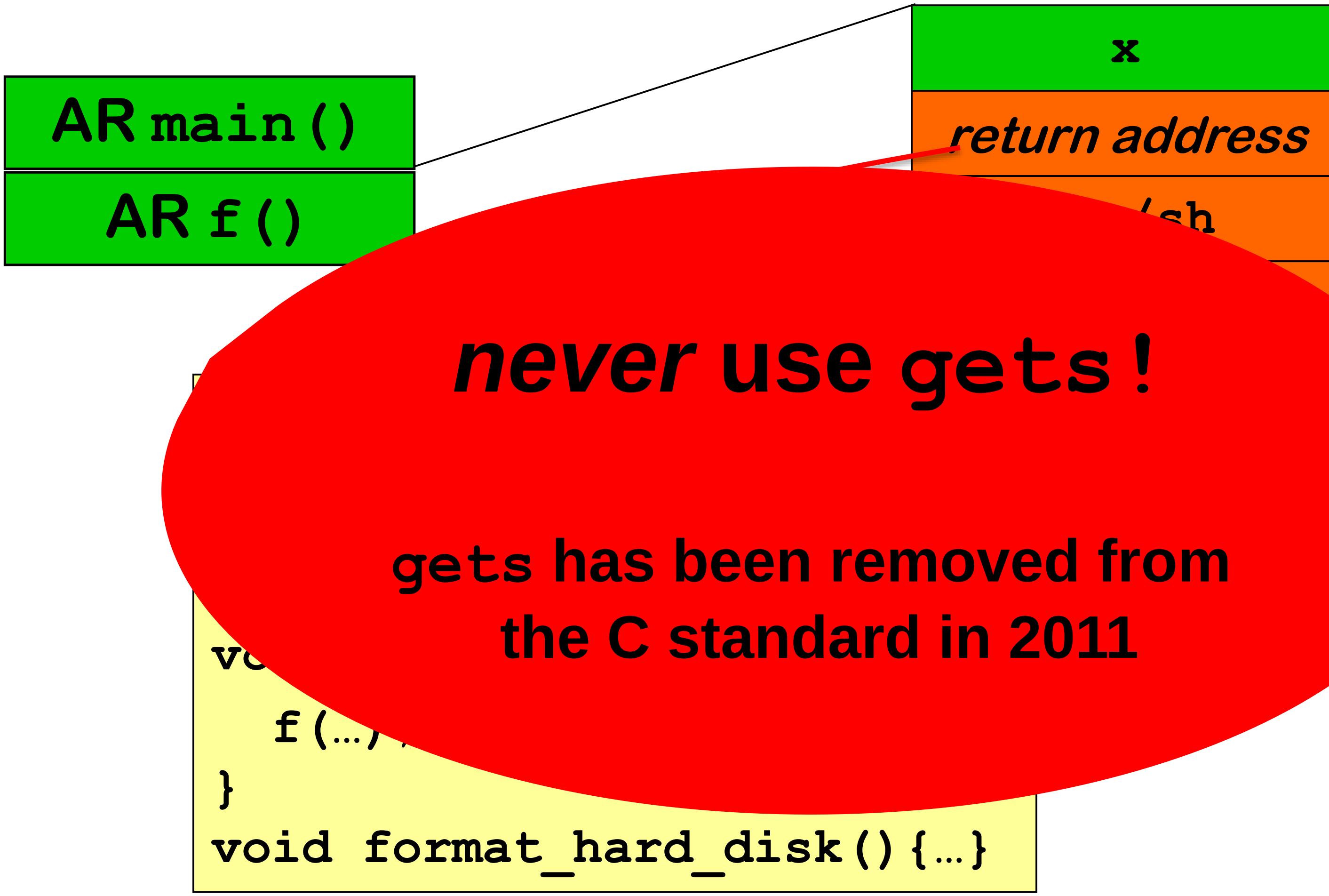



```
void f(int x) {
  char[8] buf;
  gets(buf);
}
void main() {
  f(…); ...
}
void format_hard_disk(){…}
```

Stack overflow attack - case 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

(2) is a code *injection* attack

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

In the example, `exec('/bin/sh')`

This is the classic buffer overflow attack

[Smashing the stack for fun and profit, Aleph One, 1996]

(1) is a code *reuse* attack

attacker corrupts return address to point to existing code

In the example, `format_hard_disk`

Lots of details to get right!

- **knowing precise location of return address and other data on stack, knowing address of code to jump to, ...**

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[8] username;
  int j = 12;
...
}
```

Suppose the attacker can overflow `username`

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[8] username;
  int j = 12;
...
}
```

Suppose the attacker can overflow `username`

In addition to corrupting the return address, this 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;
}
```

Suppose attacker can overflow `username`

What to attack? Fun on the heap

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

Suppose attacker can overflow `username`

This can corrupt other fields in the `struct`.

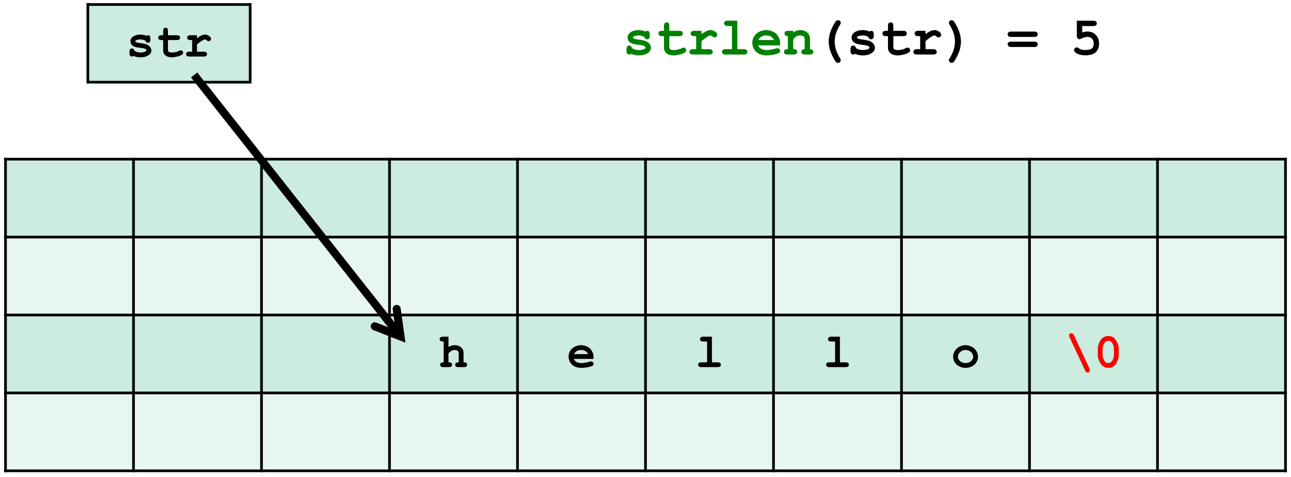
Which field(s) can be corrupted depends on the order of the fields in memory, which the compiler is free to choose.

Spotting the problem

Reminder: C chars & strings

- A `char` in C is always exactly one byte
- A string is a sequence of `chars` terminated by a NULL byte
- String variables are pointers of type `char*`

```
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`
  - `gets` has been removed from the C library
- 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 `src` is null-terminated
- Use `strncpy(dest, src, size)` instead

Example: `strcpy`

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

- `strcpy` assumes `dest` is long enough
  and `src` is null-terminated
- Use `strncpy(dest, src, size)` instead

Beware of difference between `sizeof` and `strlen`

```
sizeof(dest) = 20        // size of an array
strlen(dest) = number of chars up to first null byte
                         // length of a string
```

Spot the defect!

```
char buf[20];
char prefix[] = "http://";
char* path;
...
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://";
char* path;
...
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://";
char* path;
...
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` is now not null-terminated

so `strcpy` will overrun the buffer `dest`, because `src` is not null-terminated

Example: `strcpy` and `strncpy`

Don't replace

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

with

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

but with

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

if `dest` should be null-terminated!

NB: a strongly typed programming language would *guarantee* that strings are always null-terminated, without the programmer having to worry about this...

Spot the defect! (3)

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

buf = malloc(MAX(len,1024)); // allocate buffer
read(fd,buf,len);  // read len bytes into buf
```

Spot the defect! (3)

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

buf = malloc(MAX(len,1024)); // allocate buffer
read(fd,buf,len);  // read len bytes into buf
```

What happens if `len` is negative?

The length parameter of `read` is unsigned!
So negative `len` is interpreted as a big positive one!

(At the exam, you're not expected to remember that `read` treats its 3rd argument as an unsigned int)

Spot the defect! (3)

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

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

Note that `buf` is not guaranteed to be null-terminated; we ignore this for now.

Spot the defect! (3)

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

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

Spot the defect! (3)

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

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

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

Spot the defect! (3)

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

if (len < 0)
   {error ("negative length"); return; }
buf = malloc(MAX(len,1024));
if (buf==NULL) { exit(-1);}
                // or something a bit more graceful
read(fd,buf,len);
```

Better still

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

if (len < 0)
   {error ("negative length"); return; }
buf = calloc(MAX(len,1024));
      //to initialise allocate memory to 0
if (buf==NULL) { exit(-1);}
             // or something a bit more graceful
read(fd,buf,len);
```

Spot the defect!

```
#define MAX_BUF 256

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

    len = strlen(in);

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

Spot the defect!

```
#define MAX_BUF 256

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

     len = strlen(in);

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

Spot the defect!

```
#define MAX_BUF 256

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

     len = strlen(in);

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

What if `in` is longer than 32K ?

Spot the defect!

```
#define MAX_BUF 256

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

     len = strlen(in);

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

What if `in` is longer than 32K ?

len may be a negative number, due to integer overflow

Spot the defect!

```
#define MAX_BUF 256

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

     len = strlen(in);

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

What if `in` is longer than 32K ?

len may be a negative number, due to integer overflow

hence: potential buffer overflow

Spot the defect!

```
#define MAX_BUF 256

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

     len = strlen(in);

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

What if `in` is longer than 32K ?

len may be a negative number, due to integer overflow

hence: potential buffer overflow

The integer overflow is the root problem, the (heap) buffer overflow it causes makes it exploitable

See https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=integer+overflow

Spot the defect!

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

Spot the defect!

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

effectively does a
`malloc(count*sizeof(type))`
which may cause integer overflow

Spot the defect!

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

effectively does a
`malloc(count*sizeof(type))`
which may cause integer overflow

And this integer overflow can lead to a (heap) buffer overflow

Since 2005 Visual Studio C++ compiler adds check to prevent this

NB absence of language-level security

In a safer programming language 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!

```
void* f(int start)
    if (start+100 < start) return SOME_ERROR;
                // checks for overflow
     for (int i=start; i < start+100; i++) {
            . . . // i will not overflow
}     }
```

Spot the defect!

```
void* f(int start)
    if (start+100 < start) return SOME_ERROR;
                // checks for overflow
     for (int i=start; i < start+100; i++) {
            . . . // i will not overflow
}     }
```

Integer overflow is UNDEFINED behaviour! This means

Spot the defect!

```
1. void* f(int start)
2.     if (start+100 < start) return SOME_ERROR;
3.                 // checks for overflow
4.      for (int i=start; i < start+100; i++) {
5.             . . . // i will not overflow
6. }     }
```

Integer overflow is UNDEFINED behaviour! This means

- You cannot assume that overflow produces a negative number; so line 2 is not a good check for integer overflow.

Spot the defect!

```
1. void* f(int start)
2.     if (start+100 < start) return SOME_ERROR;
3.                      // checks for overflow
4.      for (int i=start; i < start+100; i++) {
5.                . . . // i will not overflow
6. }     }
```

Integer overflow is **UNDEFINED behaviour**! This means

- You cannot assume that overflow produces a negative number; so line 2 is not a good check for integer overflow.
- Worse still, if integer overflow occurs, behaviour is undefined, and *ANY* compilation is ok

Spot the defect!

```
1. void* f(int start)
2.      if (start+100 < start) return SOME_ERROR;
3.                      // checks for overflow
4.       for (int i=start; i < start+100; i++) {
5.                 . . . // i will not overflow
6. }      }
```

Integer overflow is UNDEFINED behaviour! This means

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

Spot the defect!

```
1. void* f(int start)
2.     if (start+100 < start) return SOME_ERROR;
3.                     // checks for overflow
4.      for (int i=start; i < start+100; i++) {
5.                . . . // i will not overflow
6. }      }
```

Integer overflow is UNDEFINED behaviour! This means

- You cannot assume that overflow produces a negative number; so line 2 is not a good check for integer overflow.
- Worse still, if integer overflow occurs, behaviour is undefined, and *ANY* compilation is ok
    - So compiled code can do *anything* if `start+100` overflows
    - So compiled code can do *nothing* if `start+100` overflows

Spot the defect!

```
1. void* f(int start)
2.     if (start+100 < start) return SOME_ERROR;
3.                  // checks for overflow
4.      for (int i=start; i < start+100; i++) {
5.              . . . // i will not overflow
6. }      }
```

Integer overflow is UNDEFINED behaviour! This means

- You cannot assume that overflow produces a negative number; so line 2 is not a good check for integer overflow.
- Worse still, if integer overflow occurs, behaviour is undefined, and *ANY* compilation is ok
  - So compiled code can do *anything* if `start+100` overflows
  - So compiled code can do *nothing* if `start+100` overflows
  - This means the compiler may *remove* line 2

Spot the defect!

```
1. void* f(int start)
2.     if (start+100 < start) return SOME_ERROR;
3.                 // checks for overflow
4.      for (int i=start; i < start+100; i++) {
5.             . . . // i will not overflow
6. }     }
```

Integer overflow is UNDEFINED behaviour! This means

- You cannot assume that overflow produces a negative number; so line 2 is not a good check for integer overflow.
- Worse still, if integer overflow occurs, behaviour is undefined, and *ANY* compilation is ok
  - So compiled code can do *anything* if `start+100` overflows
  - So compiled code can do *nothing* if `start+100` overflows
  - This means the compiler may *remove* line 2

  Modern C compilers are clever enough to know `x+100 < x` is always false, and optimise code accordingly

Spot the defect! (code from Linux kernel)

```
unsigned int tun_chr_poll(  struct file *file,
                             poll_table *wait)
{ ...
 struct sock *sk = tun->sk; // take sk field of tun
 if (!tun) return POLLERR; // return if tun is NULL
 ...
}
```

Spot the defect! (code from Linux kernel)

```
unsigned int tun_chr_poll(  struct file *file,
                             poll_table *wait)
{ ...
 struct sock *sk = tun->sk; // take sk field of tun
 if (!tun) return POLLERR; // return if tun is NULL
 ...
}
```

If `tun` is a null pointer, then `tun->sk` is **UNDEFINED**

Spot the defect! (code from Linux kernel)

```
unsigned int tun_chr_poll(  struct file *file,
                             poll_table *wait)
{ ...
  struct sock *sk = tun->sk; // take sk field of tun
  if (!tun) return POLLERR; // return if tun is NULL
  ...
}
```

If `tun` is a null pointer, then `tun->sk` is **UNDEFINED**

What this function does if `tun` is null is undefined:
ANYTHING may happen then.

Spot the defect! (code from Linux kernel)

```
1. unsigned int tun_chr_poll(  struct file *file,
2.                              poll_table *wait)
3. { ...
4.   struct sock *sk = tun->sk; // take sk field of tun
5.   if (!tun) return POLLERR; // return if tun is NULL
6.   ...
7. }
```

If `tun` is a null pointer, then `tun->sk` is **UNDEFINED**

What this function does if `tun` is null is undefined:
ANYTHING may happen then.

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

Spot the defect! (code from Linux kernel)

```
1. unsigned int tun_chr_poll(  struct file *file,
2.                              poll_table *wait)
3. { ...
4.   struct sock *sk = tun->sk; // take sk field of tun
5.   if (!tun) return POLLERR; // return if tun is NULL
6.   ...
7. }
```

If `tun` is a null pointer, then `tun->sk` is **UNDEFINED**

What this function does if `tun` is null is undefined:
ANYTHING may happen then.

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

Standard compilers (gcc, clang) do this 'optimalisation' !

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

Spot the defect! (code from Windows kernel)

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

TCHAR buf[MAX_SIZE];
_sntprintf(buf, sizeof(buf), input);
```

For code handling ASCI: 1 character is one byte

For code handling UNICODE: 1 character is several bytes

Spot the defect! (code from Windows kernel)

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

TCHAR buf[MAX_SIZE];
_sntprintf(buf, sizeof(buf), input);
```

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

For code handling ASCI: 1 character is one byte

For code handling UNICODE: 1 character is several bytes

Lots of code written under the assumption that characters are one byte contained overflows after switch from ASCI to Unicode

The CodeRed worm exploited such an mismatch.

Spot the defect!

```
#include <stdio.h>

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

Spot the defect!

```
#include <stdio.h>

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

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

Format string attacks

New type of memory corruption 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 string attacks

Suppose attacker can feed malicious input string `s` to `printf(s)`. This can

Format string attacks

Suppose attacker can feed malicious input string `s` to `printf(s)`. This can

- ***read* the stack**

  `%x` reads and prints bytes from stack so the input

  ```
  %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...
  ```

  dumps the stack ,including passwords, keys,… stored on the stack

Format string attacks

Suppose attacker can feed malicious input string `s` to `printf(s)`. This can

- *read* the stack

  %x reads and prints bytes from stack so the input

  ```
  %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. . .
  ```

  dumps the stack ,including passwords, keys,… stored on the stack

- *corrupt* the stack

  %n writes the number of characters printed to the stack,

  so input `12345678%n` writes value 8 to the stack

Format string attacks

Suppose attacker can feed malicious input string `s` to `printf(s)`. This can

- *read* the stack

  %x reads and prints bytes from stack so the input

  `%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%x...`

  dumps the stack ,including passwords, keys,… stored on the stack

- *corrupt* the stack

  %n writes the number of characters printed to the stack, so input `12345678%n` writes value 8 to the stack

- *read arbitrary memory*

  a carefully crafted format string of the form

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

  print the string at memory address ABCDCDEF

Preventing format string attacks is EASY

- Always replace `printf(str)`

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

- Compiler or static analysis tool could warn if the number of arguments does not match the format string, eg in

```
printf ("x is %i and y is %i", x);
```

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

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

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

Preventing format string attacks is **EASY**

- Always replace `printf(str)`

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

- Compiler or static analysis tool could warn if the number of arguments does not match the format string, eg in

```
printf ("x is %i and y is %i", x);
```

Eg gcc has (far too many?) command line options for this:
`-Wformat -Wformat-no-literal -Wformat-security ...`

- If the format string is not a compile-time constant, we cannot decide this at compile time ☹

  Would you want your compiler or SAST tool to give false positive or false negative?

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

Recap: buffer overflows

- Buffer overflow is #1 weakness in C and C++ programs
  - because these language are not memory-safe
- Tricky to spot
- Typical cause: programming with arrays, pointers, and strings
  - esp. library functions for null-terminated strings

- Related attacks
  - Format string attack: another way of corrupting stack
  - Integer overflows: often a stepping stone to getting a buffer to overflows
    - just the integer overflow can already have a security impact, eg think of banking software

Platform-level defences

Platform-level defences

- Defenses the compiler, hardware, OS,… can take, without the programmer having to know
- Some defenses may need OS & hardware support
- Some defenses cause overhead
  - if the overhead is unacceptable in production code, we can still use it when testing
- Some defenses may break binary compatibility
  - eg if a compiler adds extra book-keeping & checks, then all libraries may need to be re-compiled with that compiler

Platform-level defenses

1. Stack canaries
2. Non-executable memory (NX, W⊕X)
3. Address space layout randomization (ASLR)

} now standard on many platforms

More advanced defenses

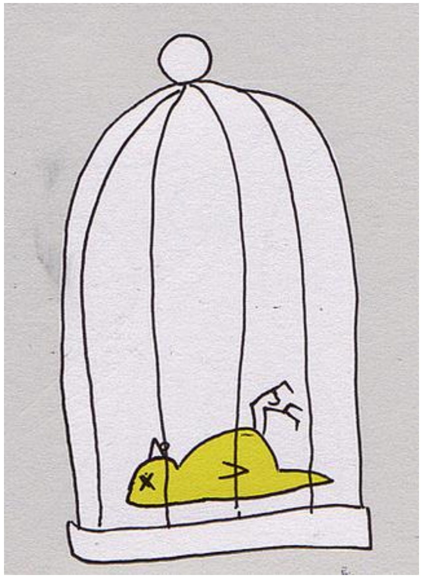
1. More randomisation: eg. pointer & memory encryption
2. More memory safety checks:
   eg. checks on bounds (spatial) or on allocation (temporal)
3. Checks on control flow
4. Execution-aware memory protection

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

1. Stack canaries

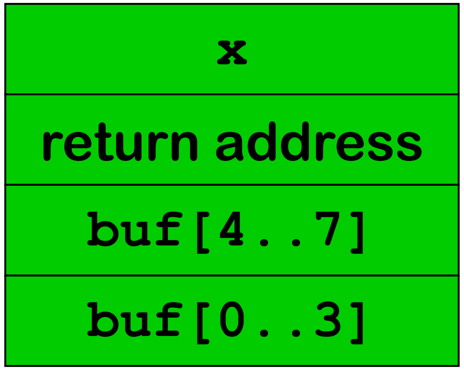
- A dummy value - stack canary or cookie - is written on the stack in front of the return address and checked when function returns
- A careless stack overflow will overwrite the canary, which can then be detected

  - first introduced in as StackGuard in gcc
  - only very small runtime overhead

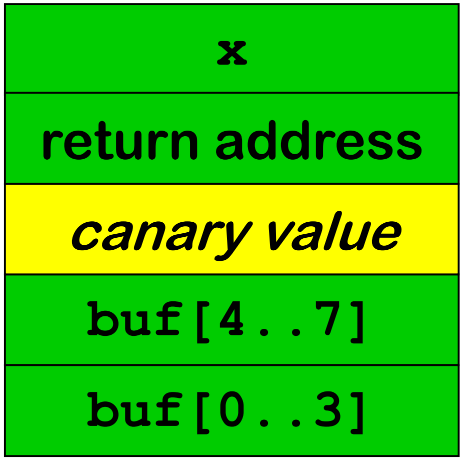


Stack canaries

Stack without canary



Stack with canary



Further improvements

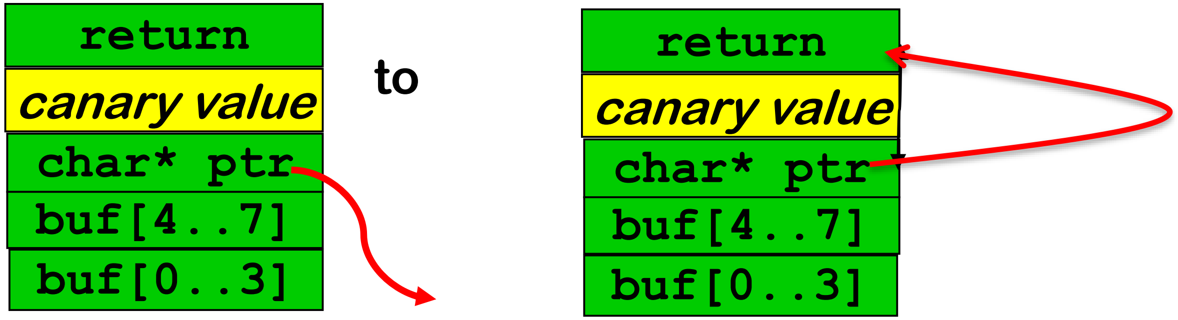
- More variation in canary values: eg not a fixed values hardcoded in binary but a random values chosen for each execution
- Better still, XOR the return address into the canary value
- Include a null byte in the canary value, because C string functions cannot write nulls inside strings

Further improvements

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

A careful attacker can still defeat canaries, by

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

eg changing 

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
  - with copies of return addresses, used to check for corrupted return addresses
  - Of course, the attacker should not be able to corrupt the shadow stack

Windows 2003 Stack Protection

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

- **Enabled with /GS command line option in Visual Studio**
- **When canary is corrupted, control is transferred to an exception handler**

Windows 2003 Stack Protection

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

- **Enabled with /GS command line option in Visual Studio**
- **When canary is corrupted, control is transferred to an exception handler**
- **Exception handler information is stored ...**

Windows 2003 Stack Protection

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

- Enabled with /GS command line option in Visual Studio
- When canary is corrupted, control is transferred to an exception handler
- Exception handler information is stored ...

  on the stack!

Windows 2003 Stack Protection

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

- **Enabled with /GS command line option in Visual Studio**
- **When canary is corrupted, control is transferred to an exception handler**
- **Exception handler information is stored ...**

  on the stack!
- **Attacker can corrupt the exception handler info on the stack, in the process corrupt the canaries, and then let Stack Protection mechanism transfer control to a malicious exception handler**

Windows 2003 Stack Protection

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

- Enabled with /GS command line option in Visual Studio
- When canary is corrupted, control is transferred to an exception handler
- Exception handler information is stored ...

  on the stack!
- Attacker can corrupt the exception handler info on the stack, in the process corrupt the canaries, and then let Stack Protection mechanism transfer control to a malicious exception handler

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

Windows 2003 Stack Protection

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

- **Enabled with /GS command line option in Visual Studio**
- **When canary is corrupted, control is transferred to an exception handler**
- **Exception handler information is stored ...**

  on the stack!
- **Attacker can corrupt the exception handler info on the stack, in the process corrupt the canaries, and then let Stack Protection mechanism transfer control to a malicious exception handler**

  [http://www.securityfocus.com/bid/8522/info]
- **Countermeasure: only allow transfer of control to registered exception handlers**

2. ASLR (Address Space Layout Randomisation)

2. ASLR (Address Space Layout Randomisation)

- Attacker needs detailed info about memory layout
  - eg to jump to specific piece of code
  - or to corrupt a pointer at known position on the stack

2. ASLR (Address Space Layout Randomisation)

- Attacker needs detailed info about memory layout
  - eg to jump to specific piece of code
  - or to corrupt a pointer at known 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

2. ASLR (Address Space Layout Randomisation)

- **Attacker needs detailed info about memory layout**
  - **eg to jump to specific piece of code**
  - **or to corrupt a pointer at known position on the stack**
- **Attacks become harder if we randomise the memory layout every time we start a program**
  - **ie. change the offset of the heap, stack, etc, in memory by some random value**

- **Attackers can still analyse memory layout on their own laptop, but will have to determine the offsets used on the victim's machine to carry out an attack**

2. ASLR (Address Space Layout Randomisation)

- Attacker needs detailed info about memory layout
  - eg to jump to specific piece of code
  - or to corrupt a pointer at known position on the stack
- Attacks become harder if we randomise the memory layout every time we start a program
  - ie. change the offset of the heap, stack, etc, in memory by some random value

- Attackers can still analyse memory layout on their own laptop, but will have to determine the offsets used on the victim's machine to carry out an attack

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

2. ASLR (Address Space Layout Randomisation)

- **Attacker needs detailed info about memory layout**
  - **eg to jump to specific piece of code**
  - **or to corrupt a pointer at known position on the stack**
- **Attacks become harder if we randomise the memory layout every time we start a program**
  - **ie. change the offset of the heap, stack, etc, in memory by some random value**

- **Attackers can still analyse memory layout on their own laptop, but will have to determine the offsets used on the victim's machine to carry out an attack**

- **NB security by obscurity, despite its bad reputation, is a really great defense mechanism to annoy attackers!**
- **Once the offset leaks, we're back to square one…**

3. Non-eXecutable memory (NX , W⊕X,DEP)

Distinguish

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

and let processor refuse to execute non-executable code

Attackers can then no longer jump to their own attack code, as any input provide as attack code will be non-executable

Aka DEP (Data Execution Prevention).

Intel calls it eXecute-Disable (XD)

AMD calls it Enhanced Virus Protection

3. Non-eXecutable memory (NX , W⊕X,DEP)

Distinguish

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

and let processor refuse to execute non-executable code

Attackers can then no longer jump to their own attack code, as any input provide as attack code will be non-executable

Aka DEP (Data Execution Prevention).

Intel calls it eXecute-Disable (XD)

AMD calls it Enhanced Virus Protection

Limitation: this technique does not work for JIT (Just In Time) compilation, where e.g. JavaScript is compiled to machine code at run time.

Defeating NX: return-to-libc attacks

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

- Attackers can no longer corrupt code or insert their own code, but can still corrupt code pointers
- Called control-flow hijack in SoK paper

So instead of jumping to own attack code
corrupt return address to jump to existing code
esp. library code in `libc`

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

reTURN oriented programMing (ROP)

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

Instead of using entire library call, attackers can

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

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

- chain these gadgets together as subroutines to form a program that does what they want

This turns out to be doable

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

A newer variant is Jump-Oriented Programming (JOP) which uses a different kind of code fragment as gadgets

More advanced defences

[See SoK Eternal War in Memory paper]

Types of (building blocks for) attacks

- **Code corruption attack**

  Overwrite the original program code in memory; impossible with W⊕X

- **Control-flow hijack attack**

  Overwrite a code pointer, eg return address, jump address, function pointer, or pointer in `vtable` of C++ object

- **Data-only attack**

  Overwrite some data, eg `bool isAdmin;`

- **Information leak**

  Only reading some data; recall Heartbleed attack on TLS

Control flow hijack via code pointers

- A compiler translates function calls in source code to `call <address>` or `JSR <address>` in machine code where `<address>` is the location of the code for the function.

Control flow hijack via code pointers

- A compiler translates function calls in source code to `call <address>` or `JSR <address>` in machine code where `<address>` is the location of the code for the function.

- For a function call `f(...)` in C a static address (or offset) of the code for `f` may be known at compile time.

  If compiler can hard-code this static address in the binary, W⊕X can prevent attackers from corrupting this address

Control flow hijack via code pointers

- A compiler translates function calls in source code to `call` <address> or `JSR` <address> in machine code where <address> is the location of the code for the function.

- For a function call `f(...)` in C a static address (or offset) of the code for `f` may be known at compile time.

  If compiler can hard-code this static address in the binary, W⊕X can prevent attackers from corrupting this address

- For a virtual function call `o->m(...)` in C++ the address of the code for `m` typically has to be determined at runtime, by inspecting the virtual function table (`vtable`)

  W⊕X does not prevent attackers from corrupting code pointers in these tables

Classification of defences [SoK paper]

- **Probabilistic methods**

  Basic idea: add randomness to make attacks harder

  - in location where certain data is located (eg ASLR), or in the way data is represented in memory (eg pointer encryption)

- **Memory Safety**

  Basic idea: do additional bookkeeping & add runtime checks to prevent some illegal memory access

- **Control-Flow Hijack Defenses**

  Basic idea: do additional bookkeeping & add runtime check to prevent strange control flow

More randomness: Pointer Encryption (PointGuard)

- Many buffer overflow attacks involve corrupting pointers, pointers to data or code pointers

More randomness: Pointer Encryption (PointGuard)

- Many buffer overflow attacks involve corrupting pointers,
  pointers to data or code pointers
- To complicate this: store pointers encrypted in main memory, unencrypted in registers
  - simple & fast encryption scheme: eg. XOR with a fixed value, randomly chosen when a process starts

More randomness: Pointer Encryption (PointGuard)

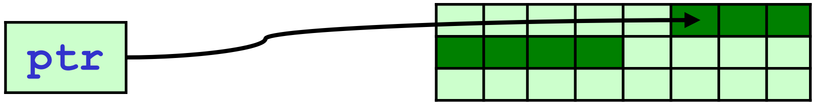
- Many buffer overflow attacks involve corrupting pointers,
  pointers to data or code pointers
- To complicate this: store pointers encrypted in main memory, unencrypted in registers
  - simple & fast encryption scheme: eg. XOR with a fixed value, randomly chosen when a process starts
- Attacker can still corrupt encrypted pointers in memory, but these will not decrypt to predictable values

More randomness: Pointer Encryption (PointGuard)

- Many buffer overflow attacks involve corrupting pointers,
  pointers to data or code pointers
- To complicate this: store pointers encrypted in main memory, unencrypted in registers
  - simple & fast encryption scheme: eg. XOR with a fixed value, randomly chosen when a process starts
- Attacker can still corrupt encrypted pointers in memory, but these will not decrypt to predictable values
  - This uses *encryption* to ensure *integrity*.
    Normally NOT a good idea, but here it works.
- More extreme variant: Data Space Randomisation (DSR)
  - store not just pointers encrypted in main memory, but store all data encrypted in memory

More memory safety

Additional book-keeping of meta-data
& extra runtime checks to prevent illegal memory access



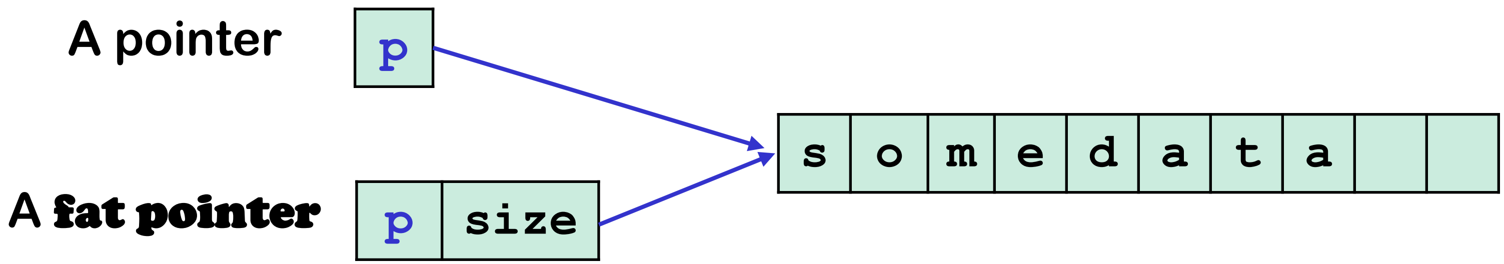
Different possibilities

- add information to pointer about size of memory chunks it points to (fat pointers)
- add information to memory chunks about their size (Spatial safety with object bounds)
- ...

Fat pointers

The compiler

- records size information for all pointers
- adds runtime checks for pointer arithmetic & array indexing



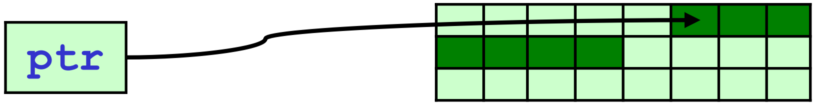
Downsides

- Considerable execution time overhead
- Not binary compatible – ie all code needs to be compiled to add this book-keeping for all pointers

More memory safety

Additional book keeping of meta-data
& extra runtime checks to prevent illegal memory access

Different possibilities



- add information to pointer about size of memory chunks it points to (fat pointers)
- add information to memory chunks about their size (Spatial safety with object bounds)
- keep a shadow administration of this meta-data, separate from the pointers & the existing memory (SoftBounds)
- keep a shadow administration of which memory cells have been allocated (Valgrind, Memcheck, AddressSanitizer or ASan)
  - to also spot temporal bugs, ie. malloc/free bugs

Object-based temporal safety (Valgrind, Memcheck, ASan)

Shadow admin

1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0
0	0	1	1	1	1	1	1

of allocated memory

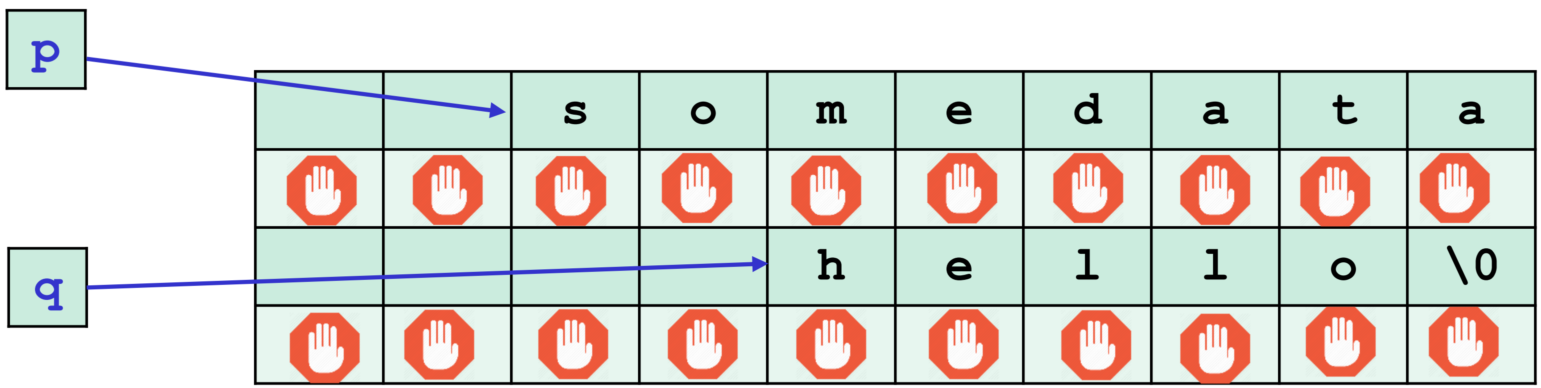
s	o	m	e	d	a	t	a
o	l	d	j	u	n	k	X
Y	Z	h	e	l	l	o	\0

to keep track of which memory is allocated, to generate runtime error when code tries to read/write unallocated memory

- Can also catch spatial bugs, ie. small buffer overruns, by keeping empty space between allocated chunks (unless overrun is huge)
  - small overrun will end up in this unallocated space
- Cannot spot illegal access via a stale pointer if the data chunk it points to has been re-allocated
  - Eg the last bug, line 3004, on slide 15

Guard pages to improve memory safety

Allocate chunks with the end at a page boundary with a non-readable, non-writeable page between them



Buffer overwrite or overread will cause a memory fault.

Small execution overhead, but big memory overhead

Control Flow Integrity (CFI)

Extra bookkeeping & checks to spot unexpected control flow

- **Dynamic return integrity**

  Stack canaries, or shadow stack that keeps copies of all return addresses, providing extra check against corruption of return addresses

- **Static control flow integrity**

  Idea: determine the control flow graph (cfg) and monitor jumps in the control flow to spot deviant behavior

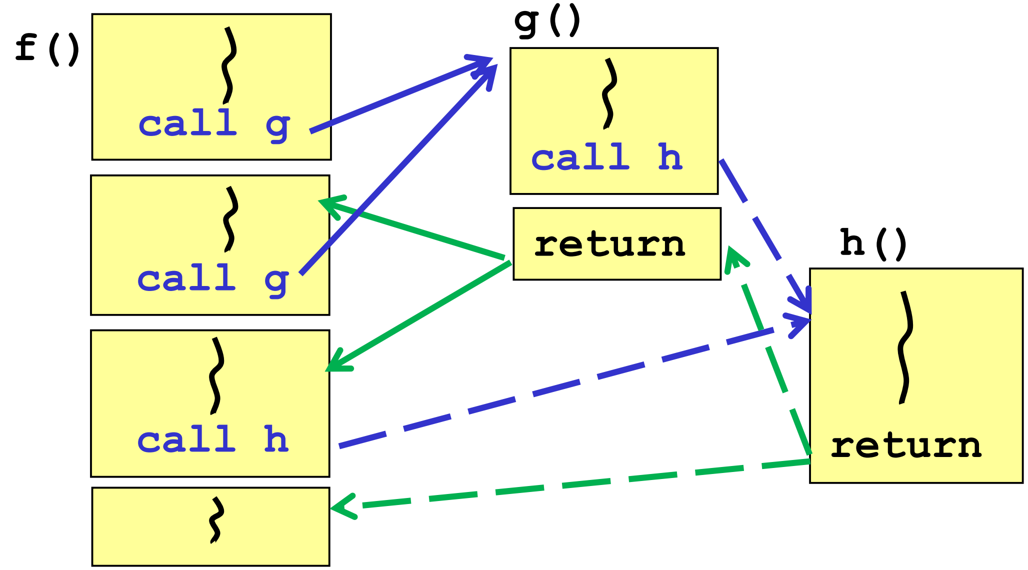
  > **If `f()` never calls `g()`,**
  > **because `g()` does not even occur in the code of `f()`,**
  > **then call from `f()` to `g()` is suspicious,**
  > **as is a return from `g()` to `f()`**

  We could interrupt execution when this happens

  This can detect Return-to-libc and ROP attacks

Static control flow integrity: example code & CFG

```
void f() {
    ... ; g();
    ... ; g();
    ... ; h();
    ...
}
void g(){ ..h();}
void h(){ ... }
```



Before and/or after every control transfer (function call or return) we could check if it is legal – ie. allowed by the cfg

Some weird returns would still be allowed

- eg if we call `h()` from `g()`, and the return is to `f()`, this would be allowed by the static cfg
- Additional *dynamic* return integrity check can narrow this down to actual call site – using recorded call site on shadow stack

Downsides of static control flow integrity checks

- Requires a whole program analysis

- Use of function pointers in C or virtual functions in C++ (that both result in so-called indirect control transfers) complicate compile-time analysis of the cfg: we'd need
  - a points-to analysis to determine where such code pointers can point to

    eg in C++, if `Animal->eat()` can resolve to `Cat->eat()` or `Dog->eat()`, so both these addresses are valid targets for transferring control
  - or: simply allow transfer to any function entry point

New(er) features of main OS [not exam material]

- Pointer encryption in iOS (2018)

- Hardware-enforced Stack Protection in Windows 10 (2020)
  - with a shadow stack,
    using Intel Control-flow Enforcement Technology (CET)

  https://techcommunity.microsoft.com/t5/windows-kernel-internals/understanding-hardware-enforced-stack-protection/ba-p/1247815

- Evolution of CFI at Microsoft discussed by Joe Bialek

  https://www.youtube.com/watch?v=oOqpl-2rMTw

  The Evolution of CFI Attacks and Defenses @ OffensiveCON 18

Exam questions: you should be able to

- **Explain how simple buffer overflows work & what root causes are**
- **Spot a *simple* buffer overflow, memory-allocation problem, format string attack, or integer overflow in some C code**
- **Explain how countermeasures - such as stack canaries, non-executable memory, ASLR, CFI, bounds checkers, pointer encryption, ... - work**
- **Explain why they might not always work**