Software Security

Memory corruption

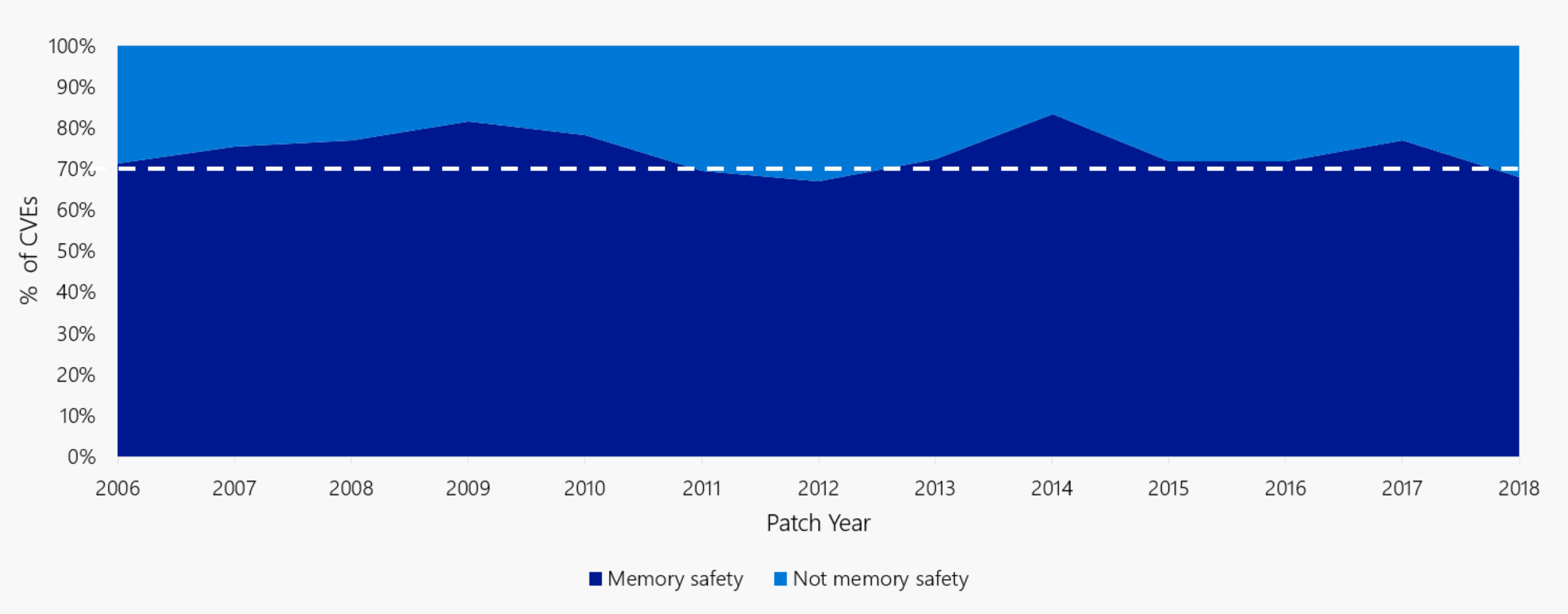
public enemy number 1

Erik Poll

Digital Security

Radboud University Nijmegen

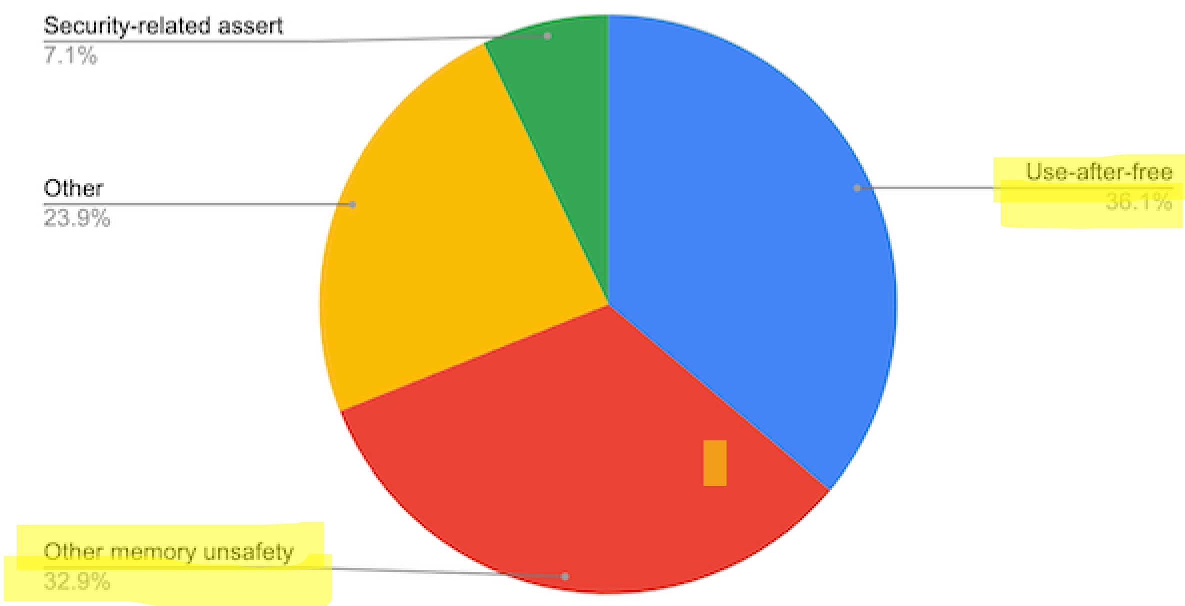
Memory corruption bugs vs rest - Microsoft 2006-2018



[Source: https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code and *"Trends, challenge, and shifts in software vulnerability mitigation*", presentation by Matt Miller at BlueHat IL 2019]

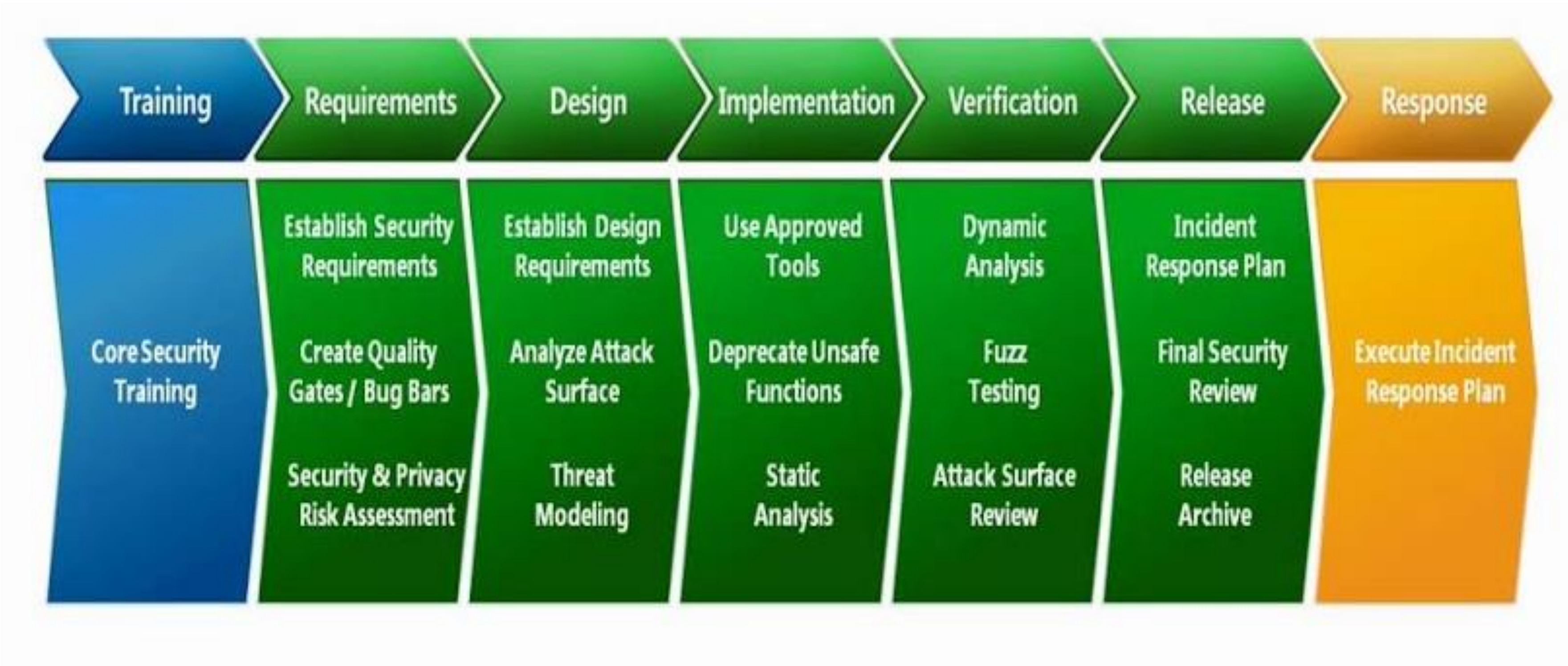
Memory corruption bugs in Chromium project – since 2015

70% of high severity & critical security bugs are memory unsafety problems

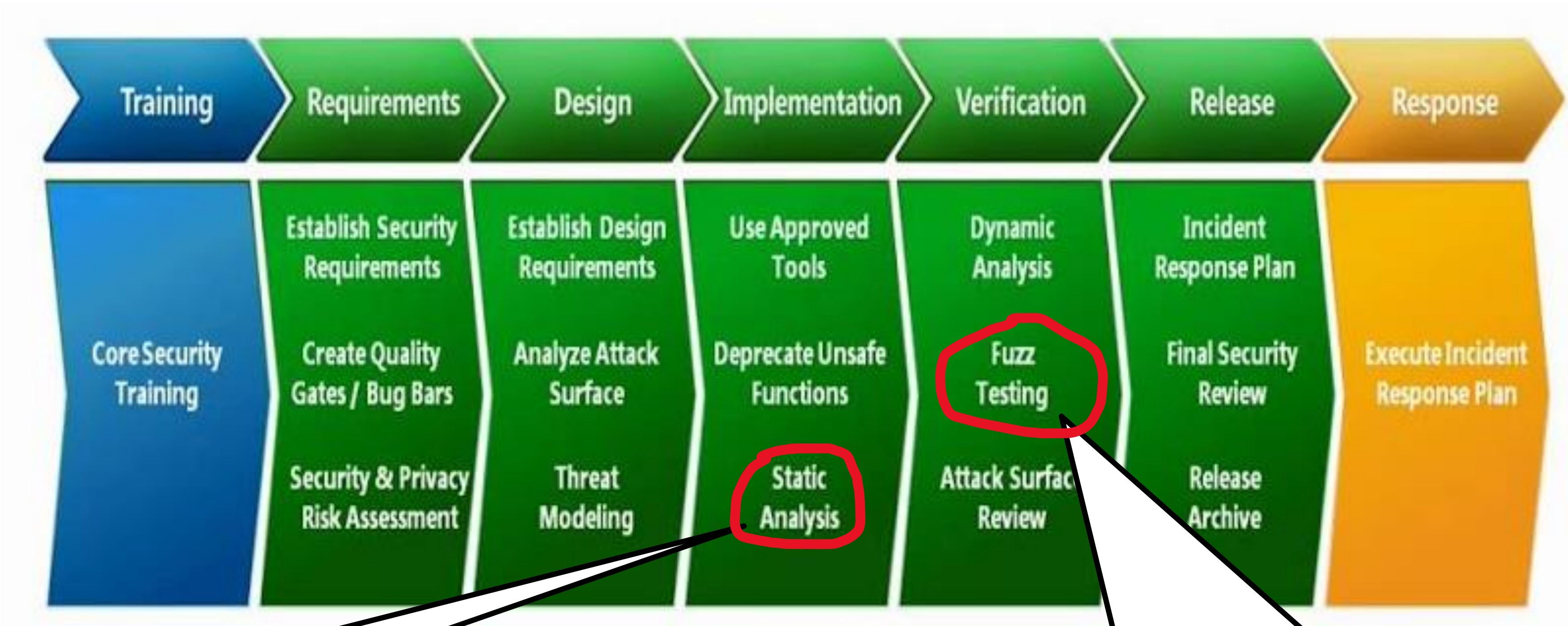


[Source: https://www.chromium.org/Home/chromium-security/memory-safety]

Security in the development lifecycle



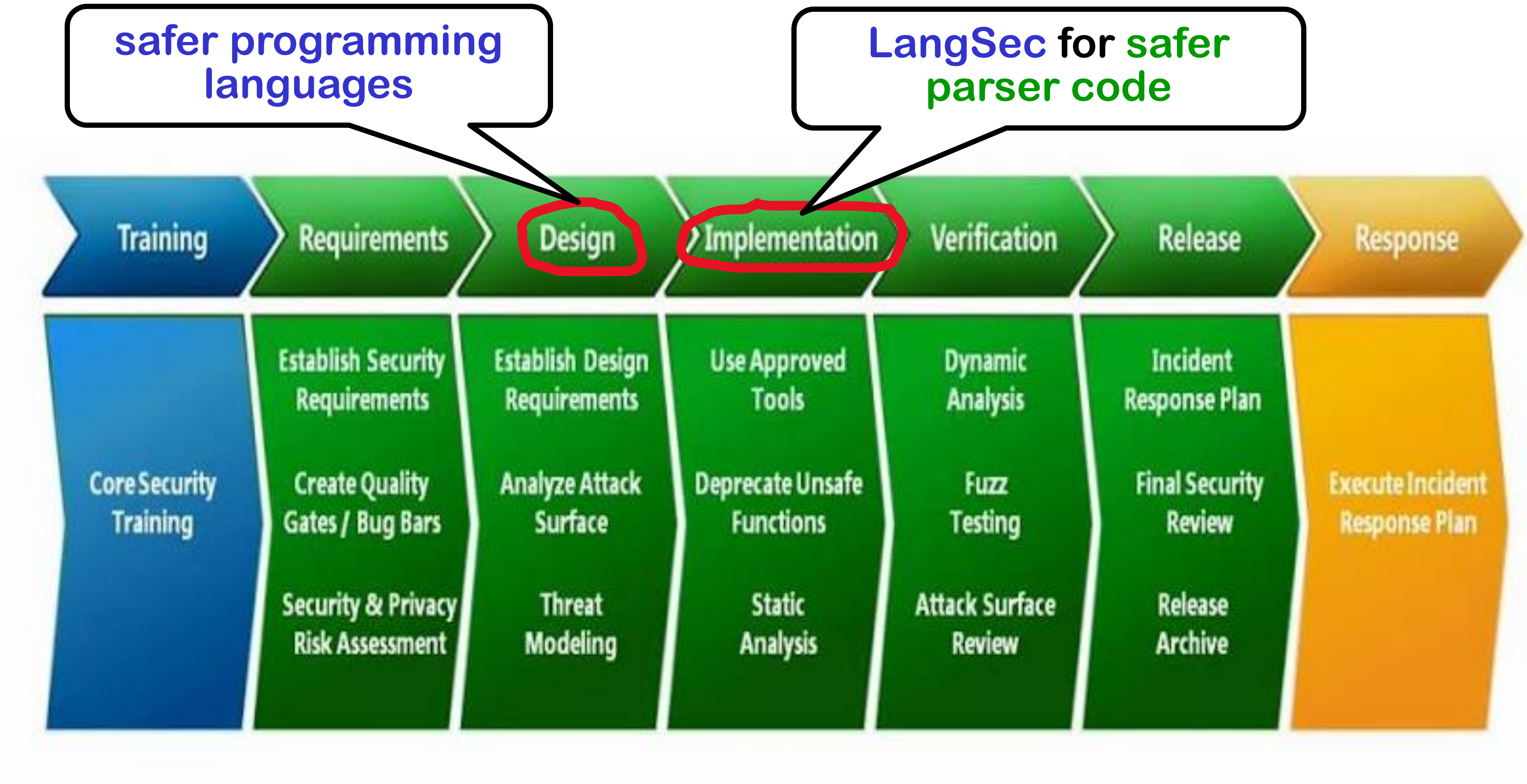
Finding & fixing memory corruption – next weeks



week 3: exercise
Static analysis
with PREfast

week4: group project
fuzzing afl
memory sanitizers Asan, Msan

More structural prevention - later



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?

   Tool-support for this

   - SAST: PREfast individual / pair 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';
```

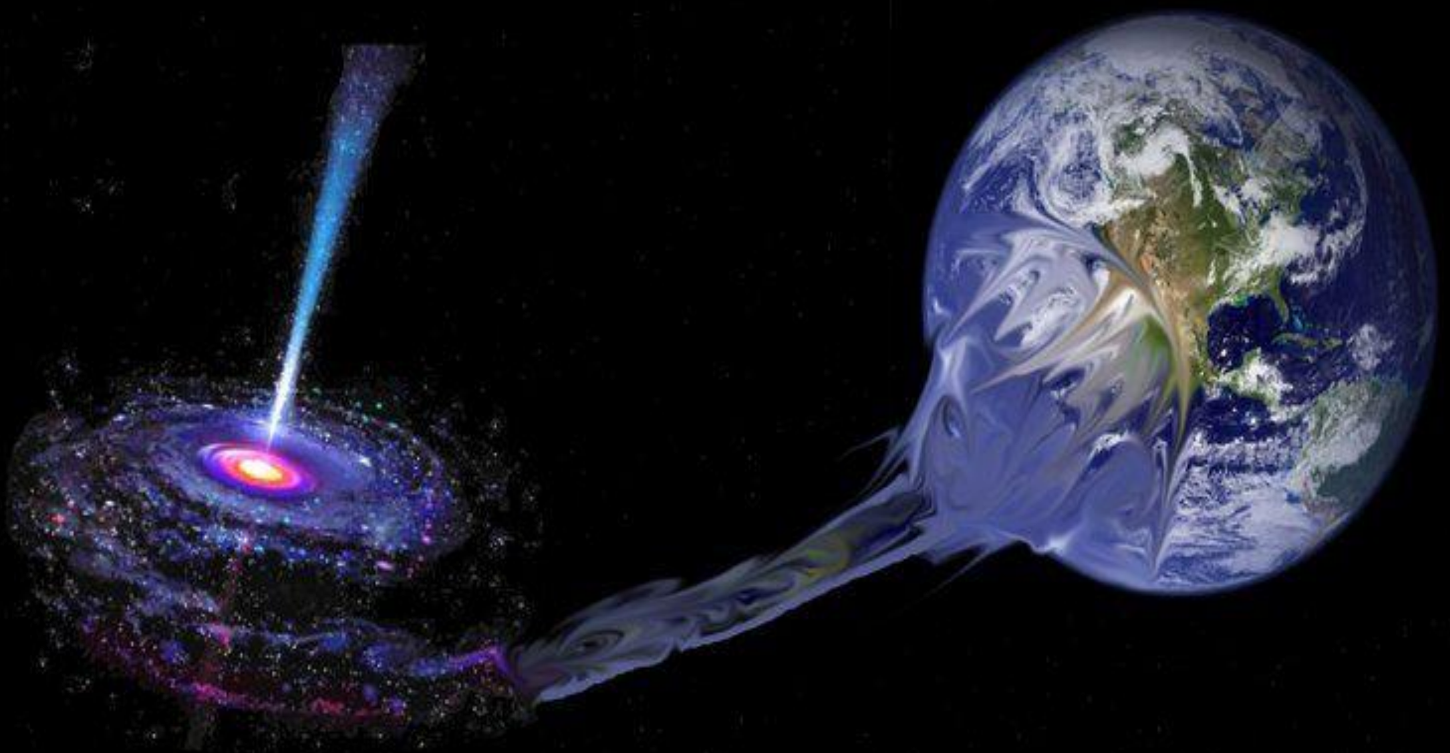
We don't know!

This is defined to be undefined

ANYTHING can happen

UNDEFINED behaviour: <u>anything</u> can happen

UNDEFINED behaviour: anything can happen



UNDEFINED behaviour: <u>nothing</u> may happen

Anything attackers wants?

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

If the attacker *controls the value* `'a'`

then *anything that the attacker wants* may happen …

- If we are *lucky* : program crashes with **`SEGMENTATION FAULT`**
- If we are *unlucky* : program does not crash

  but silently allows data corruption or remote code execution (RCE)

  and we *won't* know till it's too late

Nothing may happen

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

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

- Compilers actually do this (as part of optimisation) and this can cause security problems; examples later & in the lecture notes.

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
  - Check out CVEs mentioning buffer (or buffer%20overflow)

    https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=buffer
- 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]

More memory corruption problems

Errors with pointers and with dynamic memory (aka 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)

Spot all (potential) defects

```
...
void f(){
  char* buf, buf1, buf24;
  buf = malloc(100);
  buf[0] = 'a';
...
  free(buf24);
  buf[0] = 'b';
...
  free(buf);
  buf[0] = 'c';
  buf1 = malloc(100);
  buf[0] = 'd'
}
```

null dereference if malloc failed

***potential* use-after-free** if buf & buf24 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] may point to memory that has been re-allocated for buf1

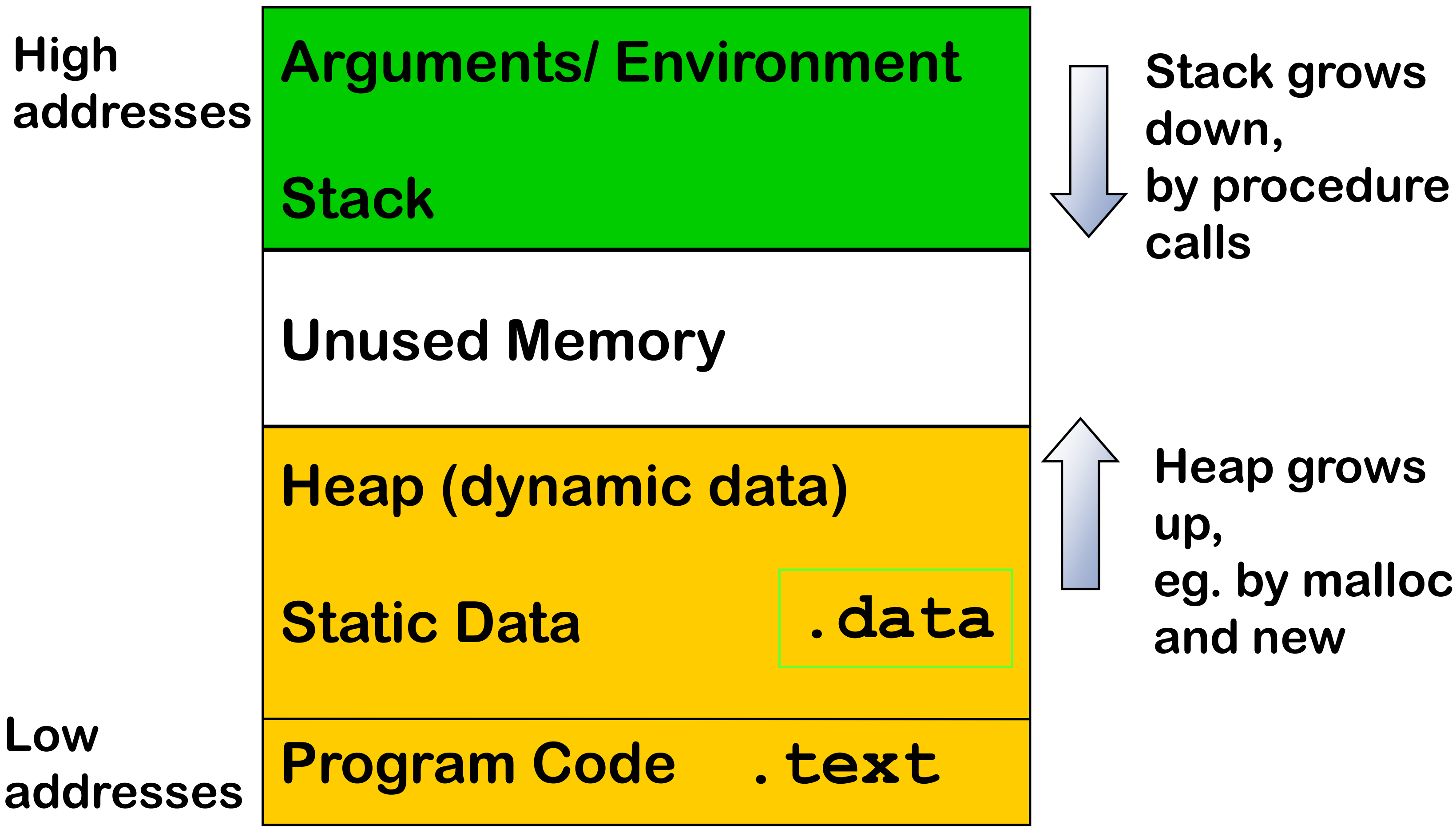
Causes of memory corruption problems

- Access outside array bounds aka buffer overflow
  - overread or overwrite

    overreads are not a corruption issue, but *confidentiality* issue
- Pointer trouble:
  - buggy pointer arithmetic,
  - dereferencing null pointer,
  - using a dangling pointer aka stale pointer
    - caused by e.g. use-after-free or double-free
- Memory management problems:
  - Forgetting to check for failures in allocation
  - Forgetting to de-allocate, aka memory leaks
    - not a corruption issue, but an *availability* issue
- Other ways to break memory abstractions: missing null terminators, too many null terminators, type casts, type confusion, ...

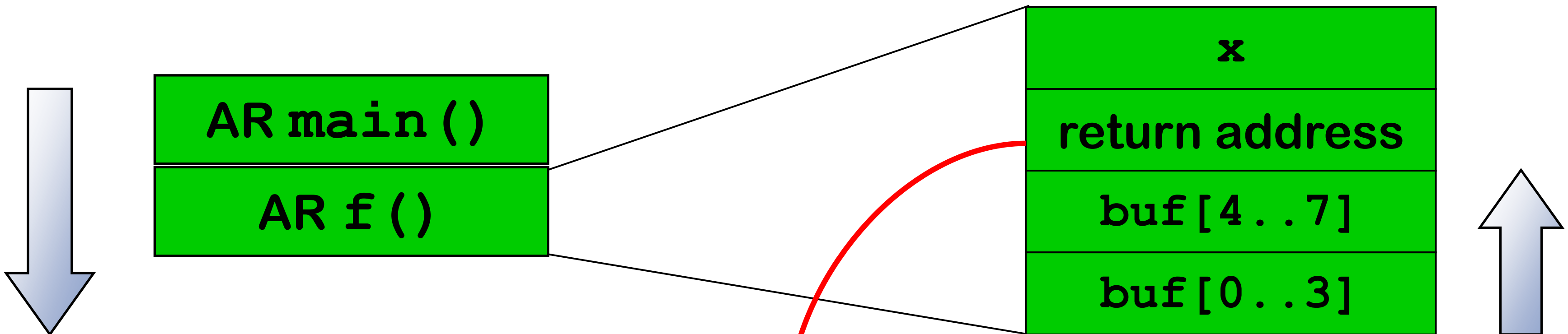
Exploiting this

Process memory layout



Stack layout

The stack consists of Activation Records aka stack frames:



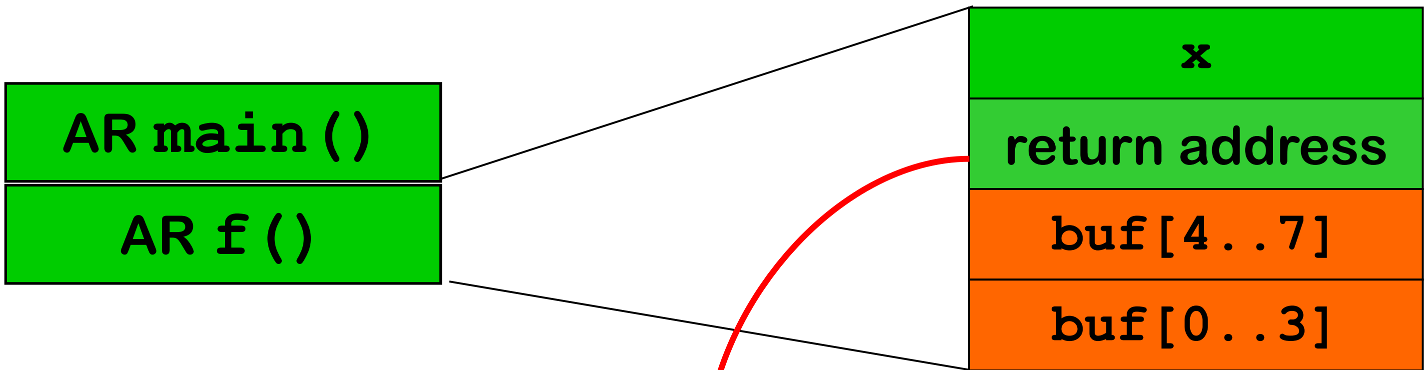
Stack grows downwards

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

Buffer buf goes upwards

Stack overflow attack - case 1

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

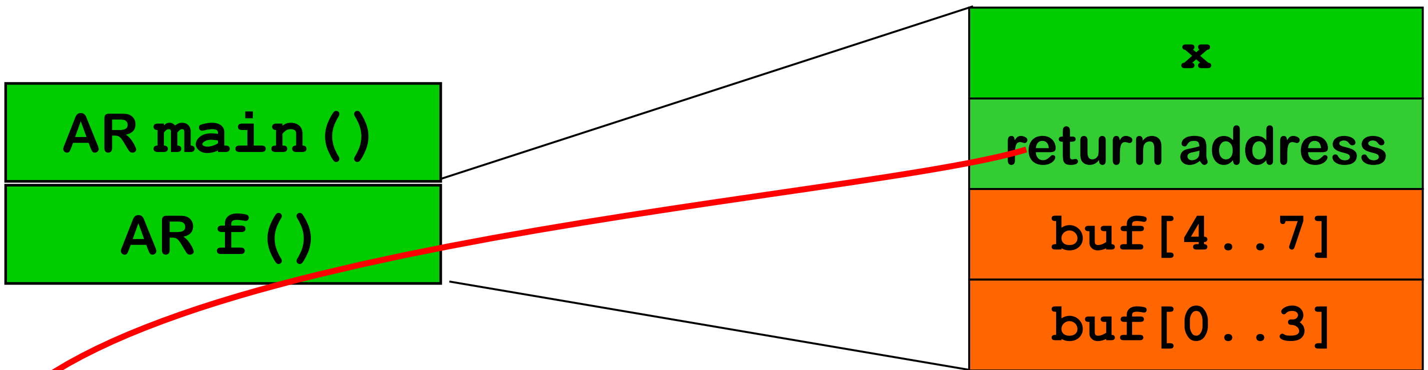



```
void f(int x) {
  char[8] buf;
  gets(buf);
}
void main() {
  f(…); …
}
void format_hard_disk(){…}
```

Stack overflow attack - case 1

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

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



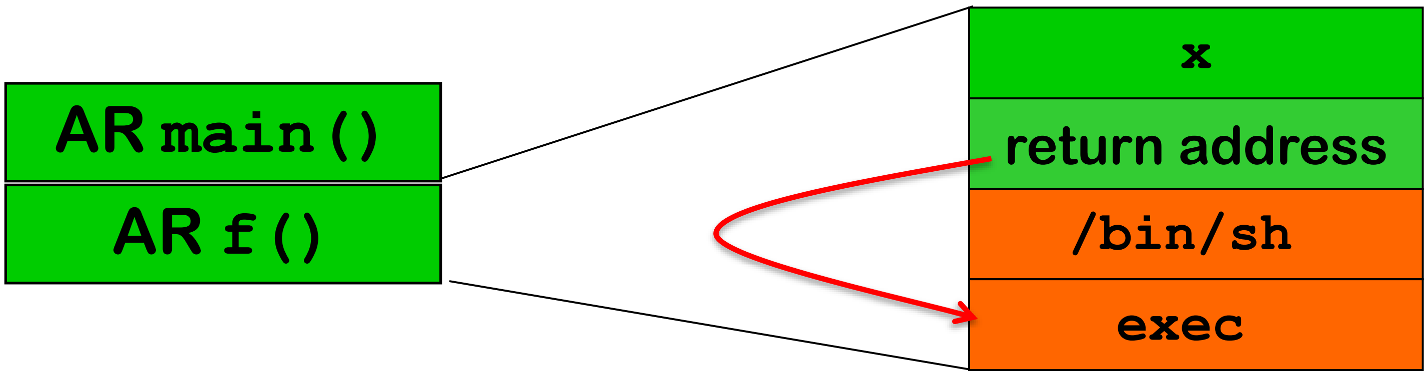

```
void f(int x) {
  char[8] buf;
  gets(buf);
}
void main() {
  f(…); ...
}
void format_hard_disk(){…}
```

code *reuse* attack

Stack overflow attack - case 2

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

Attackers can also jump to <u>their own code</u> (aka shell code)



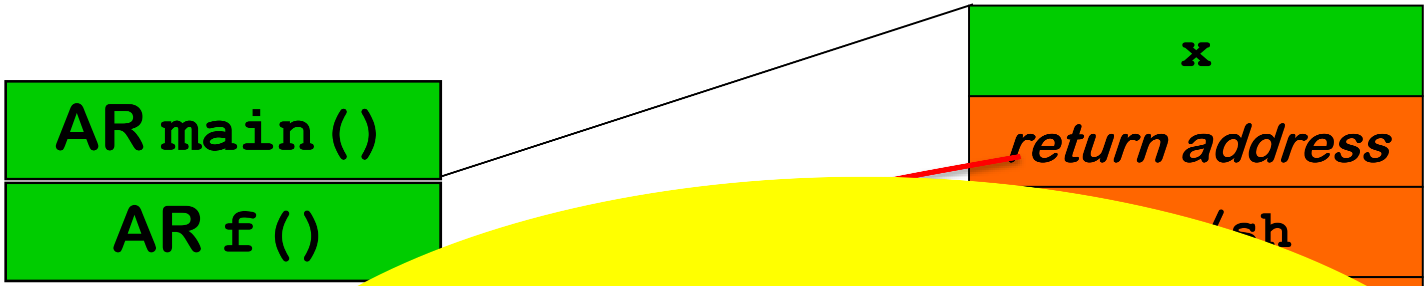

```
void f(int x) {
  char[8] buf;
  gets(buf);
}
void main() {
  f(…); ...
}
void format_hard_disk(){…}
```

code *injection* attack

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

```
vo
   f(...)
}
void format_hard_disk(){...}
```

Code *injection* vs code *reuse*

Two types of attacks in these examples

(2) is a code *injection* attack

attackers inject their own shell code in some buffer and corrupt 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

attackers corrupt 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? Corrupting the *stack*

function pointer

```
void f(int x,
       void(*error_handler)(int),
       bool b) {
   int  diskquota = 200;
   bool is_super_user = false;
   char* filename = "/tmp/scratchpad";
   char[8] username;
   int j = 12;
   ...
}
```

Suppose attacker can overflow `username`

This can corrupt the return address, but also other data on the stack:

`is_super_user, diskquota, filename, x, b, error_handler`

- But not `j`, unless the compiler chooses to allocate variables in a different order, which the compiler is free to do
- Corruption function pointers such as `error_handler` is particularly interesting! *Why?*

What to attack? Corrupting data 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 fields depends on the order of the fields in memory.

  The compiler is free to choose this.

What to attack? Corrupting *vtables on the heap*

C++ code uses late binding to resolve (so-called virtual) method calls

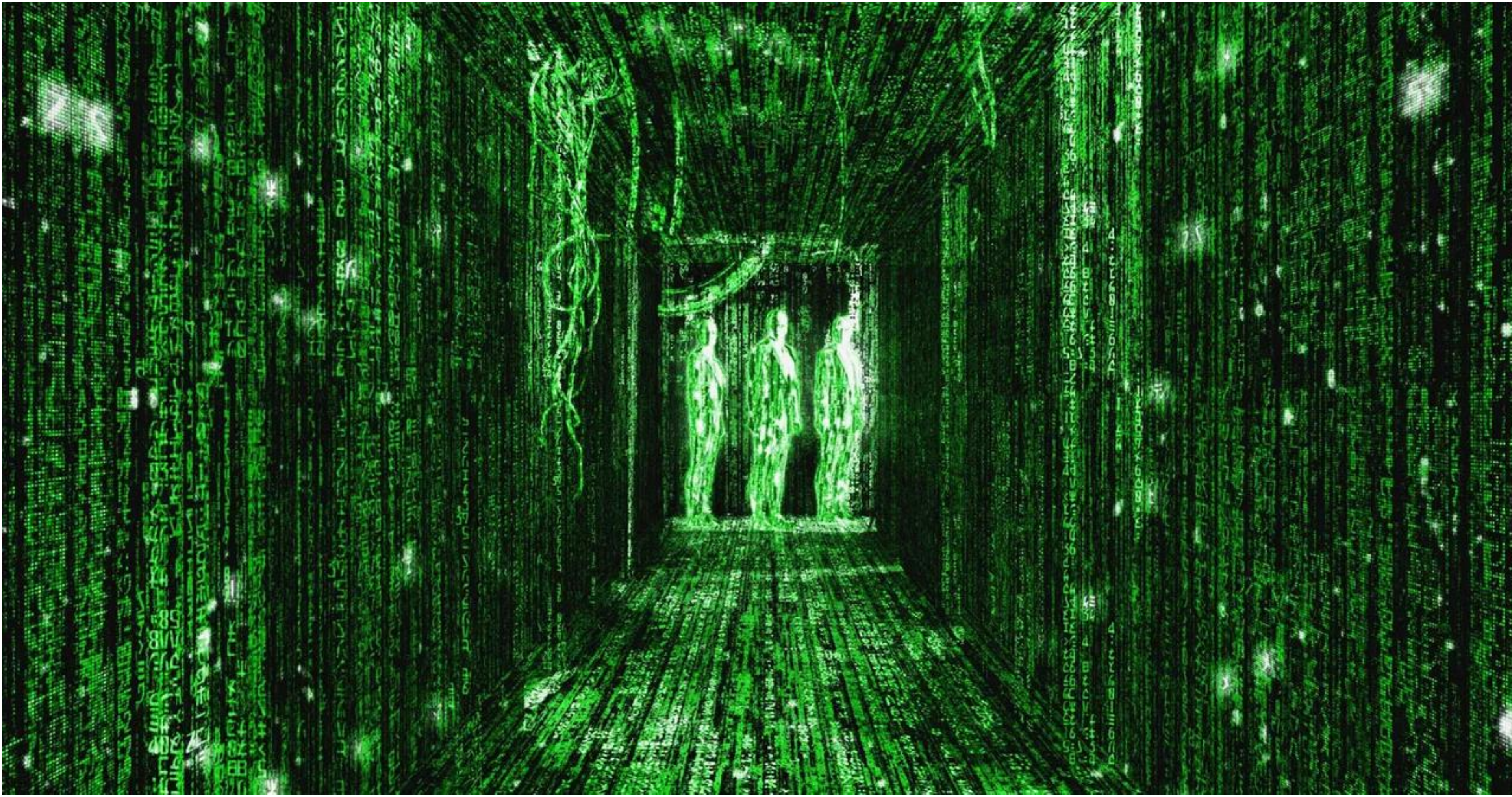
```
Rectangle r;
Circle c;
Shape s;

_surface_area = r.area() + c.area() + s.area();
```

Which code to execute for `s.area()` is determined at runtime.

To do this, a table of function pointers, the `vtable`,
is maintained that tells which code to execute for each method

This provides many function pointers for attackers to mess with!

Recurring theme in attacks: breaking abstractions

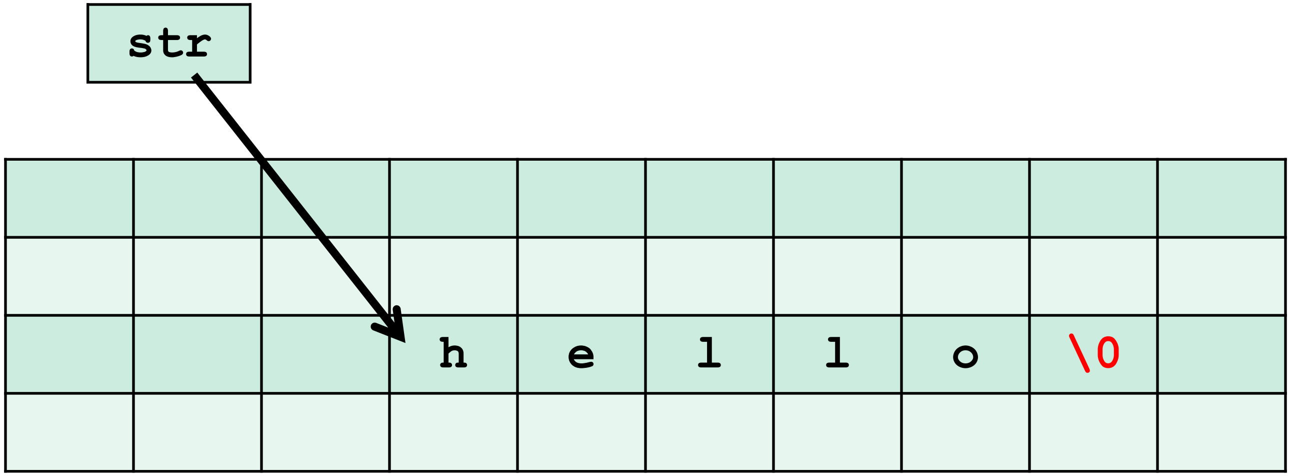


Spotting the problem

Reminder: C chars & strings

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

```
char* str = "hello";    // a string str
```



Here `strlen(str)` will be `5`

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 so this code will no longer compile
- Use `fgets(buf, size, file)` instead

Example: `strcpy`

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

- `strcpy` assumes that 1 `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
```

strncat's 3rd parameter is number of chars to copy, not the buffer size

So this should be `sizeof(buf)-7`

Better libraries?

Keeping track of the space left in buffers when using `strncpy` and `strncpy` is error-prone. Better alternatives:

- `strlcpy(dst,src,size)` and `strlcat(dst,src,size)`
  Here `size` is the size of destination array `dst`, not the maximum length copied. These are consistently used in OpenBSD.
- Functions in Microsoft's Strsafe.h also always takes destination size as argument. Moreover, they guarantee null-termination.

Other alternatives:

- glib.h provides `Gstring` type for dynamically growing null-terminated strings in C
- C++ string objects are less error-prone than C strings
  - but `data()` and `c-str()` return a C string, ie. a `char*`, and result of `data()` is not always null-terminated on all platforms.

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`, 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 you want `dest` to 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
```

What happens if `len` is negative?

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

(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);
```

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* input)
{   short len;
    char buf[MAX_BUF];

    len = strlen(input);

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

Spot the defect!

```
#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 `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;
        }
 }
```

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)
- strings not having a null terminator
- 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!

```
1. void* f(int start) {
2.     if (start+100 < start) return SOME_ERROR_CODE;
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:
  - So compiled code can do *anything* if `start+100` overflows
  - So compiled code can do *nothing* if `start+100` overflows
  - This means the compiler can *remove* line 2

  Modern C compilers are clever enough to know that `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)

```
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 when `tun` is null is undefined:
ANYTHING may happen then.

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

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

This is code from the Linux kernel where 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 _tprintf _wprintf
#else
#   define TCHAR char
#   define _tprintf _printf
#endif

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

wide UNICODE character, > 1 byte

print-function for wide character strings

ASCI character, 1 byte

print-function for ASCI character strings

`sizeof(buf)` is the size in *bytes*, but this parameter should be the number of *characters*

Switch from ASCI to UNICODE caused lots of buffer overflows

[slide from presentation by Jon Pincus]

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

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

If attacker can control malicious input `s` to `printf(s)` then this can

- ***read* the stack**

  `%x` reads and prints bytes from stack

  so 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 the value 8 to the stack

- ***read arbitrary memory***

  a carefully crafted input 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

1. Always replace `printf(str)`
   with `printf("%s", str)`

2. Compiler or static analysis (SAST) tool could warn if the number of arguments does not match the format string

As e.g. in `printf ("x is %i and y is %i", x);`

- `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 then want your compiler or SAST tool to a give false positive or false negative?

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

Recap: memory corruption

- #1 weakness in C / C++
  - because these language are not memory-safe and programmer is responsible for memory management
- Tricky to spot
- Typical cause: programming with arrays, pointers, strings & and dynamic (ie heap-allocated) memory

- Related attacks
  - Format string attack: another way of corrupting stack
  - Integer overflows: useful a stepping stone to getting a buffer to overflows, or dangerous in its own right

Platform-level defences

Platform-level defences

- Defenses the compiler, hardware, OS,… can take, without the programmer having to know
- Some defenses need OS & hardware support
- Some defenses cause overhead
  - if this overhead is unacceptable in production code, we can still use it in testing phase
- Some defenses may break binary compatibility
  - if the compiler adds extra book-keeping & checks, 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:

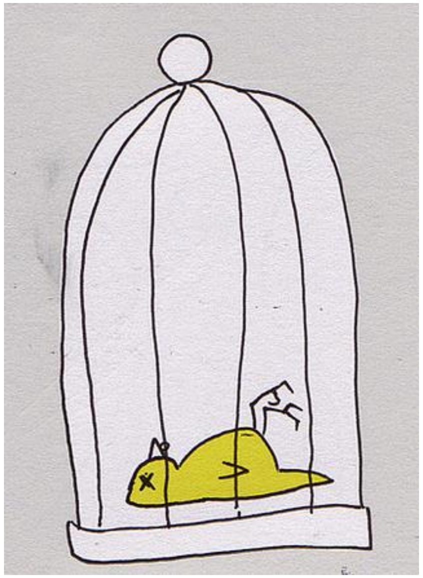
   eg. checks on bounds (spatial) or on allocation (temporal)
3. Checks on control flow
4. Execution-aware memory protection

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

1. Stack canaries

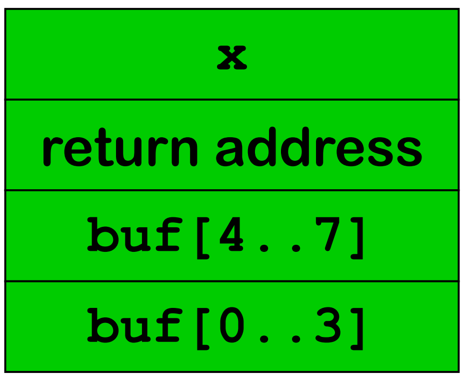
- Stack canary aka stack cookie is written on the stack in front of the return address and checked when function returns
- A careless stack overflow will overwrite the canary, which can then be detected

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

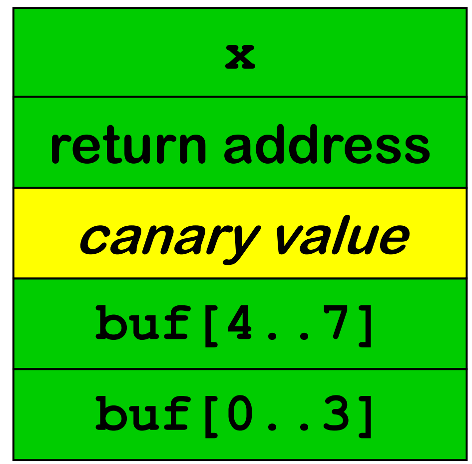


Stack canaries

Stack without canary



Stack with canary



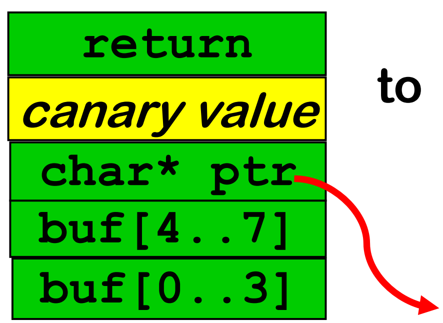
Further improvements

- 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

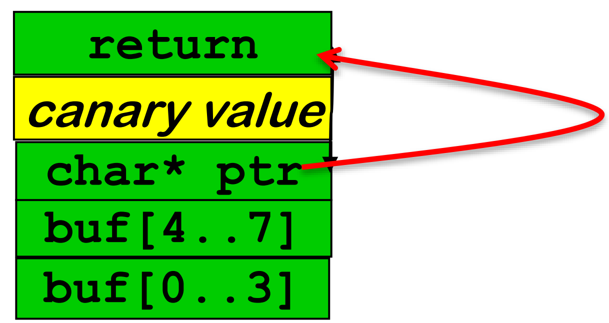
A careful attacker can still defeat canaries, by

- 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



to



Aside: corrupting pointers

Overwriting pointers is especially interesting because subsequent uses of that pointer then read/write data in another place that attacker can choose.

```
100   char* ptr;
101   char[8] buf;
      ...
200   fgets(buf, 12, stdin);  // overflow corrupts ptr,
       // e.g. to point to the position of return address
     ...
210   fgets(ptr, 100, stdin);
             // corrupts any location chosen by the
             // attacker when overflowing buf in line 200
```

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...

- **/GS command line option in Visual Studio add stack canaries**
- **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 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)

- 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…