Software Security

Buffer Overflows

public enemy number 1

Erik Poll

Digital Security

Radboud University Nijmegen

Overview

1. How do buffer overflows work?
2. How can we spot such problems in C(++) code?

   Next week: tool-supported 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 the data written (ie. the "a") is user input that can be controlled by an attacker, this vulnerability can be exploited: *anything that the attacker* wants can happen

- If we are lucky, we will get a SEGMENTATION FAULT.
  - this stops the attacks, but might still cause DoS.
- 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: ...* A consequence of this principle is that *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 and C++

  ever since the Morris Worm in 1988
- Typically, attackers that can feed malicious input to a program can full control over it, incl.
  - services accessible over the network,
    eg. sendmail, 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 clevere**r**, defeating ever better countermeasures

How does buffer overflow work?

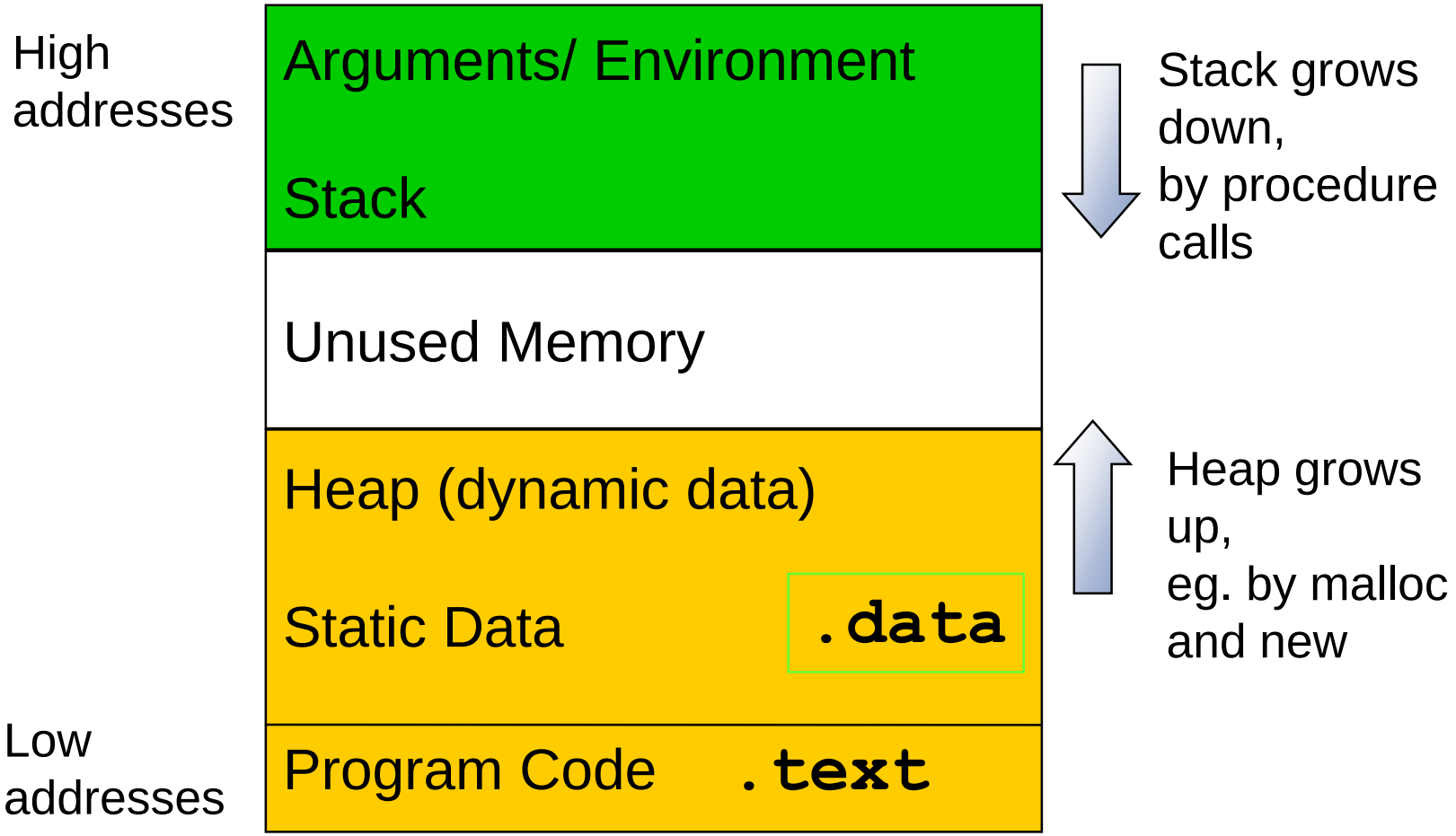
Memory management in C/C++

- Program responsible for its own memory management
- Memory management is very error-prone
  - *Who here has had a C(++) program crash with a segmentation fault?*

  Technical term: C and C++ do not offer **memory-safety**
  (see lecture notes on language-based security, §3.1-3.2)

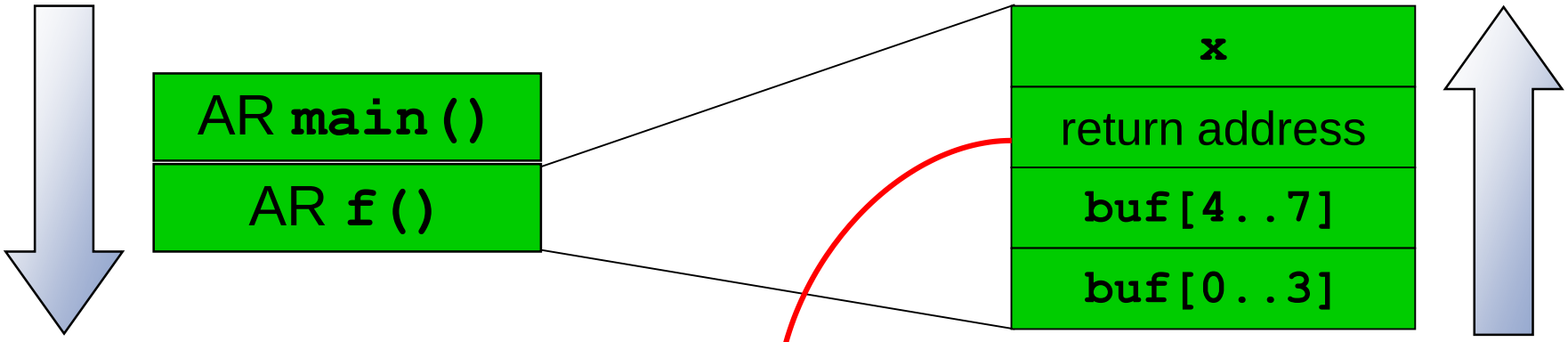
- Typical bugs:
  - Writing past the bound of an array
  - Pointer trouble
    - missing initialisation, bad pointer arithmetic, use after de-allocation (use after free, double free), failed allocation, forgotten de-allocation (memory leaks)...
- For efficiency, these bugs are not detected at run time: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

Process memory layout



Stack overflow

The stack consists of Activation Records:



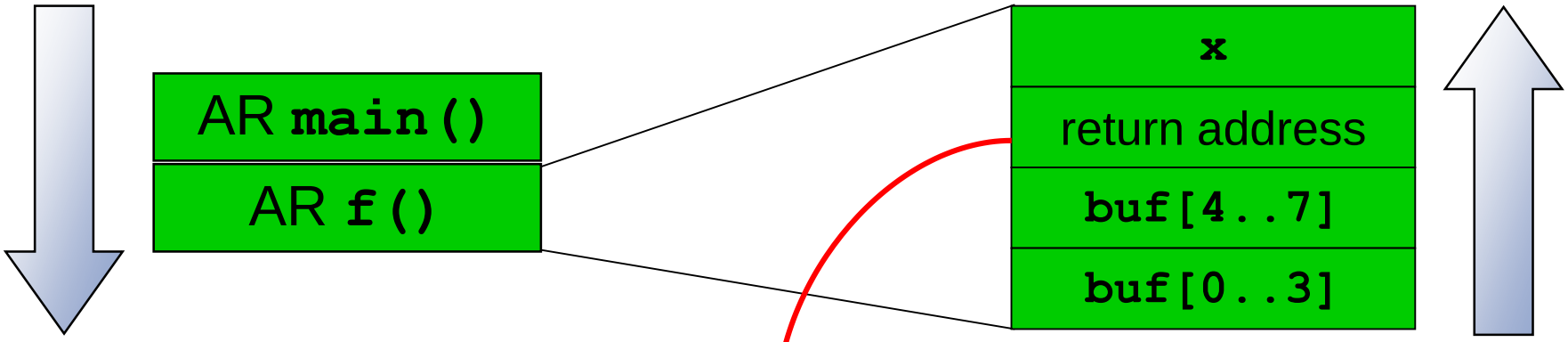
Stack grows downwards

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

Buffer grows upwards

Stack overflow

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



Stack grows downwards

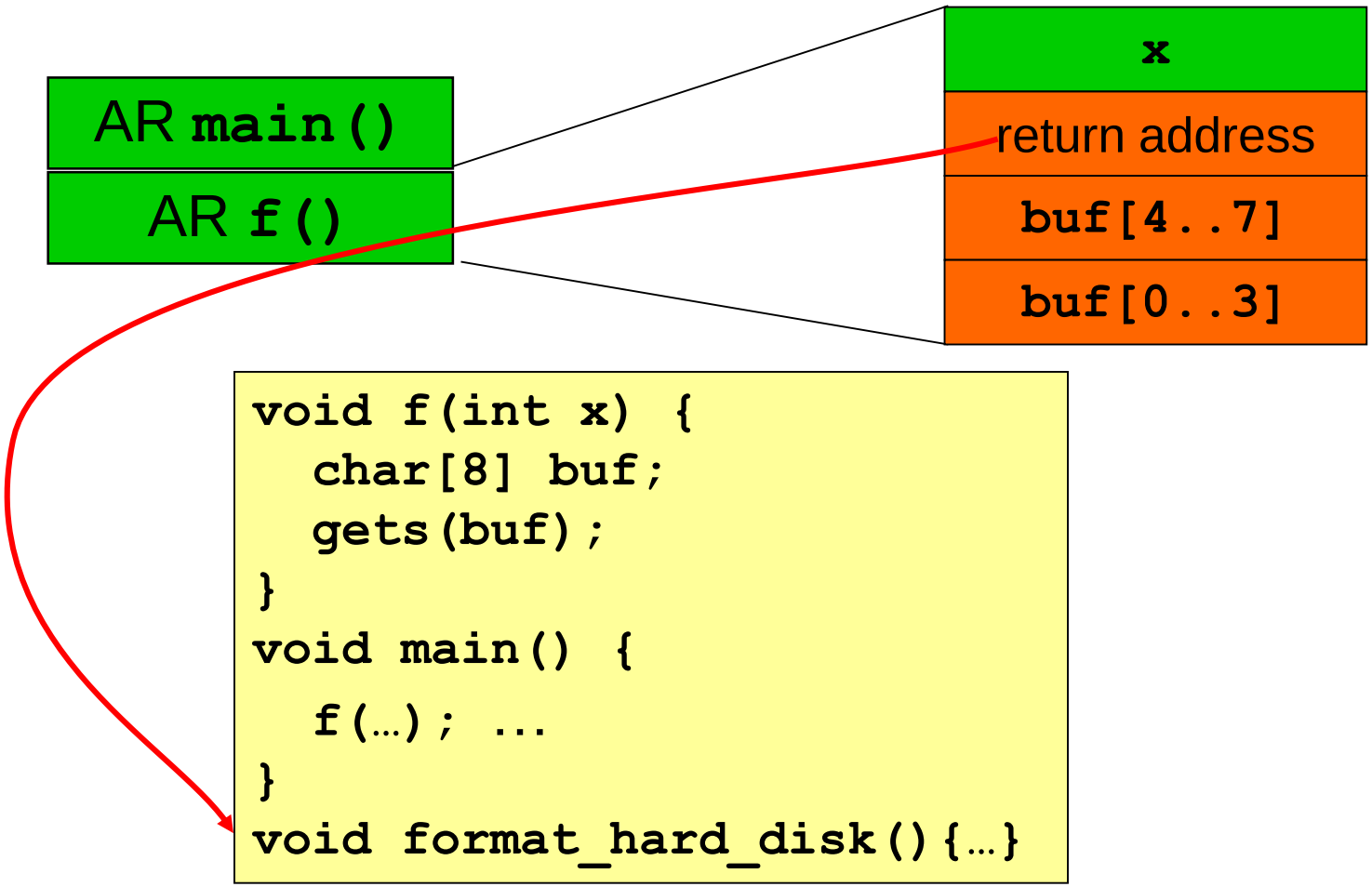
```
void f(int x) {
   char[8] buf;
   gets(buf);
}
void main() {
   f(…); …
}
void format_hard_disk(){…}
```

Buffer grows upwards

Stack overflow

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

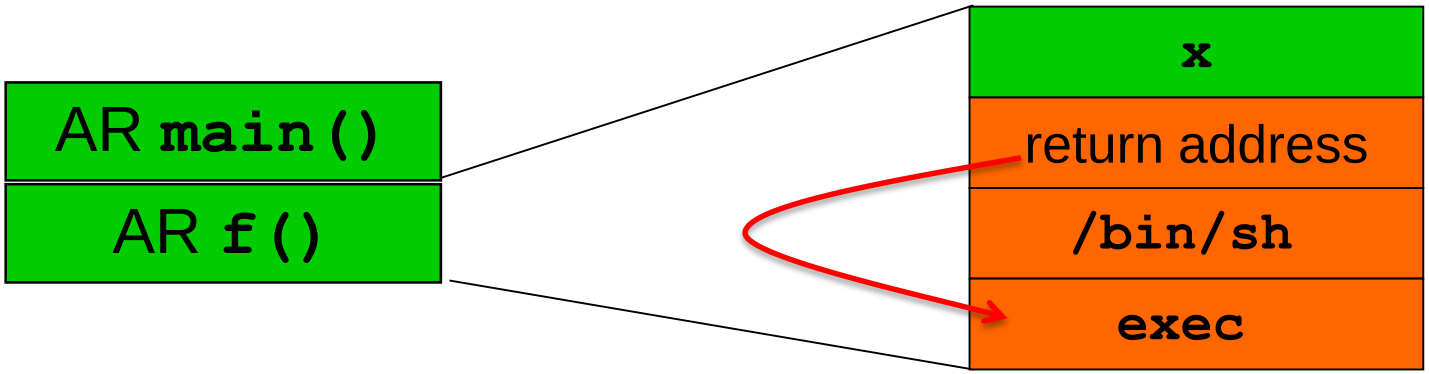
Attacker can jump to any point in the code!



Stack overflow

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

*Attacker can even jump to his own code in **`buffer`**! (aka shell code)*

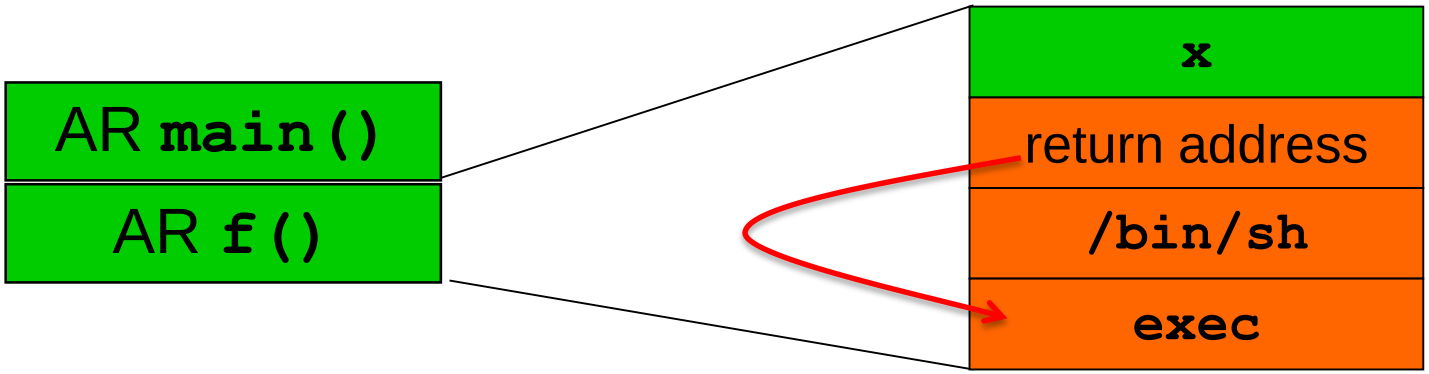



```
void f(int x) {
  char[8] buf;
  gets(buf);
}
void main() {
  f(…); ...
}
void format_hard_disk(){…}
```

Stack overflow

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

*Attacker can even jump to his own code in **`buffer`**! (aka shell code)*




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

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

code injection vs code reuse attacks

The two attack scenarios discussed on these examples

1. code *injection* attack

   attacker inserts his own shell code in a buffer and corrupts the 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]

2. code *reuse* attack

   attacker corrupts the return address to point to existing code,

   In the example, **`format_hard_disk`**

.

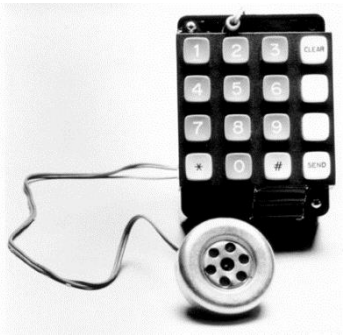
Recurring problem: mixing control & data

In 1950s, Joe Engressia showed the telephone network could be hacked by phone phreaking, ie. whistling at right frequencies

http://www.youtube.com/watch?v=vVZm7I1CTBs

The root cause: in-band signaling

In 1970s, before founding Apple together with Steve Jobs, Steve Wozniak sold Blue Boxes for phone phreaking at university



Stack overflow, ie buffer overflow on the stack

- *How* the attacks works: overflowing buffers to corrupt data

- Lots of details to get right:
  - No nulls in (character-)strings
  - Filling in the correct return address:
    - Fake return address must be precisely positioned
    - Attacker might not know the address of his own string
  - Other overwritten data must not be used before return from function
  - …

- Variant: Heap overflow of a buffer allocated on the heap instead of the stack

What to attack? More fun on the stack

```
void f(void(*error_handler)(int),...) {
  int  diskquota = 200;
  bool is_super_user = false;
  char* filename = "/tmp/scratchpad";
  char[] buf;
  ....
}
```

Suppose the attacker can overflow **buf** .

Apart from corrupting the return address, an attacker could also

- corrupt pointers, eg **filename**
- corrupt function pointers, such as **error_handler**
- corrupt any other data on the stack, eg. **is_super_user, diskquota**

What to attack? Fun on the heap

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

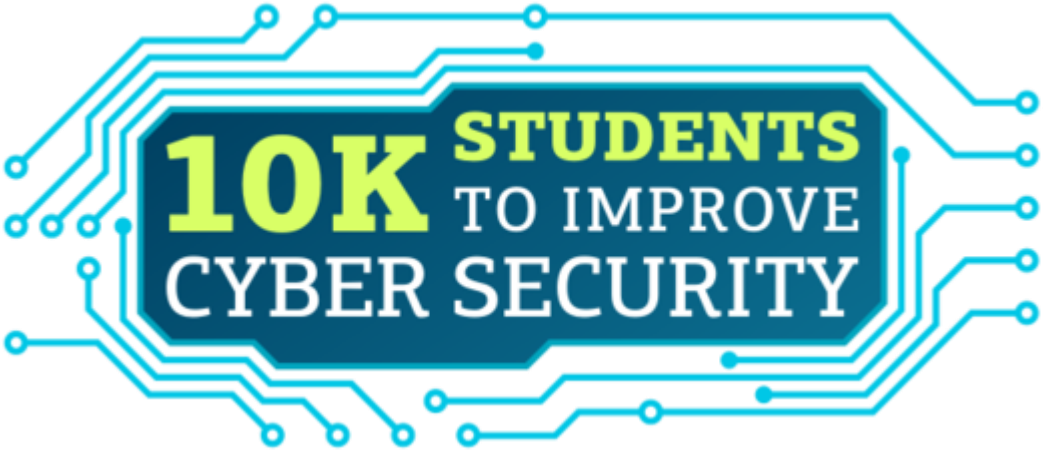
overrun **`name`** to corrupt the values of other fields in the struct

Heap overflows can also corrupt the memory management administration associated with **`malloc()`** and **`free()`**

More background (not exam material!)

More background on ***how*** buffer overflows & exploits work

- Section 2 of “Low-Level Software Security by Example”, by Ulfar Erlingsson, Yves Younan, and Frank Piessens
- Online lectures Part I & II at http://10kstudents.eu/
- Or take the Hacking in C course at RU or Hacker’s Hut at TU/e



Spotting the problem

Example: `gets`

```
char buf[20];
gets(buf); // read user input until
           // first EoL or EoF character
```

- *Never* use **`gets`**
- Use **`fgets(buf, size, stdin)`** instead

Example: `strcpy`

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

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

Spot the defect! (1)

```
char buf[20];
char prefix[] = ”http://”;
...
strcpy(buf, prefix);
  // copies the string prefix to buf
strncat(buf, path, sizeof(buf));
  // concatenates path to the string buf
```

Spot the defect! (1)

```
char buf[20];
char prefix[] = ”http://”;
...
strcpy(buf, prefix);
  // copies the string prefix to buf
strncat(buf, path, sizeof(buf));
  // concatenates path to the string buf
```

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

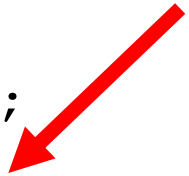
Spot the defect! (2)

```
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 i, len;

read(fd, &len, sizeof(int));
     // read sizeof(int) bytes, ie. an int,
     // and store these at &len, ie. the
     // memory address of the variable len
buf = malloc(len);
read(fd,buf,len); // read len bytes into buf
```

Spot the defect! (3a)

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

read(fd, &len, sizeof(int));
     // read sizeof(int) bytes, ie. an int,
     // and store these at &len, ie. the
     // memory address of the variable len
buf = malloc(len);
read(fd,buf,len); // read len bytes into buf
```

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

len cast to unsigned and negative length overflows

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

Spot the defect! (3b)

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

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

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

Spot the defect! (3c)

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

read(fd, &len, sizeof(len));
if (len < 0)
   {error ("negative length"); return; }
buf = malloc(len+1);
read(fd,buf,len);
buf[len] = '\0'; // null terminate buf
```

May result in integer overflow; we should check that **len+1** is positive

Spot the defect! (3d)

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

read(fd, &len, sizeof(len));
if (len+1 < 0)
    {error ("negative length"); return; }
buf = malloc(len+1);
read(fd,buf,len);
buf[len] = '\0'; // null terminate buf
```

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

Spot the defect! (3e)

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

read(fd, &len, sizeof(len));
if (len < 0)
   {error ("negative length"); return; }
buf = malloc(len+1);
if (buf==NULL) { exit(EXIT_FAILURE);}
              // or something a bit more graceful
read(fd,buf,len);
buf[len] = '\0'; // null terminate buf
```

Spot the defect! (3f) - Better still?

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

read(fd, &len, sizeof(len));
if (len < 0)
   {error ("negative length"); return; }
buf = calloc(len+1);
      //to initialise allocate memory to 0
if (buf==NULL) { exit(EXIT_FAILURE);}
               // or something a bit more graceful
read(fd,buf,len);
buf[len] = '\0'; // null terminate buf
```

NB Absence of language-level security

Note that in other (nicer?) programming languages, the programmer might not have to worry about

- writing past array bounds (the language could throw an IndexOutOfBoundsException instead)
- implicit conversions from signed to unsigned integers
- malloc possibly returning null (the language could throw an OutOfMemoryException instead)
- malloc not initialising memory
- integer overflow (the language could throw an IntegeroverflowException instead)
- ...

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);
```

[slide from presentation by Jon Pincus]

Spot the defect! (4)

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

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

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

The CodeRed worm exploited such an mismatch. Code written under the assumption that 1 character was 1 byte contained many buffer overflows after the move from 1 byte ASCI to 4 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

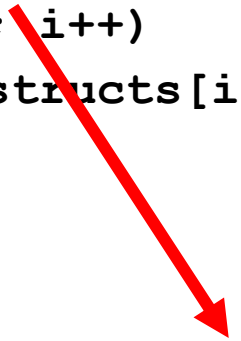
The integer overflow is the root problem, but the (heap) buffer overflow that this enables make it exploitable

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

Spot the defect! (7)

```
char buff1[MAX_SIZE], buff2[MAX_SIZE];
// make sure url a valid URL and fits in buff1 and
   buff2:
if (! isValid(url)) return;
if (strlen(url) > MAX_SIZE - 1) return;
// copy url up to first separator, ie. first '//', to
   buff1
out = buff1;
do {
  // skip spaces
   if (*url != ' ') *out++ = *url;
} while (*url++ != '/');
strcpy(buff2, buff1);
...
```

[slide from presentation by Jon Pincus]

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

```
char buff1[MAX_SIZE], buff2[MAX_SIZE];
// make sure url a valid URL and fits in buff1 and
   buff2:
if (! isValid(url)) return;
if (strlen(url) > MAX_SIZE - 1) return;
// copy url up to first separator, ie. first '/', to
   buff1
out = buff1;
do {
  // skip spaces
   if (*url != ' ') *out++ = *url;
} while (*url++ != '/');
strcpy(buff2, buff1);
...
```

length up to the first null

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

[slide from presentation by Jon Pincus]

Spot the defect! (8)

```
#include <stdio.h>

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

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

Format string attacks

New type of attack 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);
  ```

  will dump 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 %
  ```

- **%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
  ```

Format string attacks should be history…

because

1. Format string attacks are easy to spot & fix:

   replace **`printf(str)`**

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

2. A simple static analysis tool could warn about any remaining **`print`** statements without a format string

An IDE or compiler could (should?) have such a check built-in

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 the programmer needing to know, include

1. stack canaries
2. non-executable memory (NX, W^X)
3. address space randomization (ASLR)

now standard on many platforms

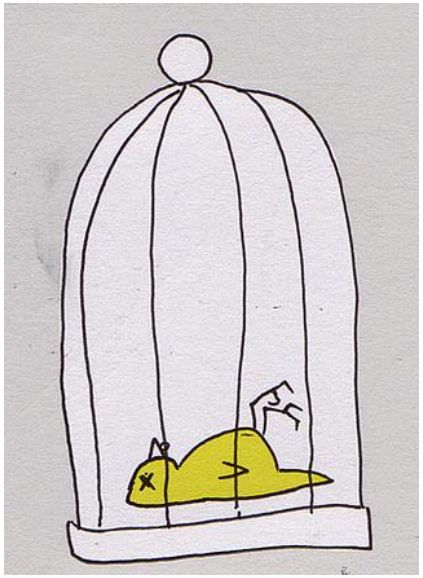
4. control-flow integrity (CFI)
5. pointer encryption
6. execution-aware memory protection

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

History has shown that all new defenses get eventually defeated...

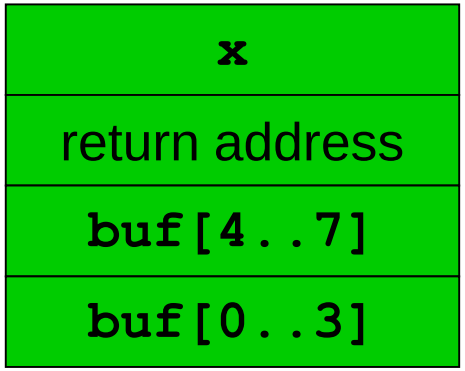
1. stack canaries

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

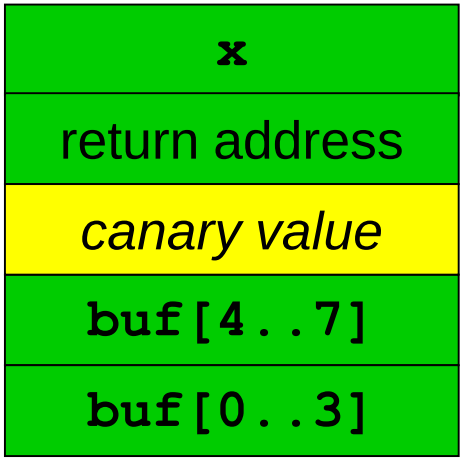


stack canaries

Stack without canary



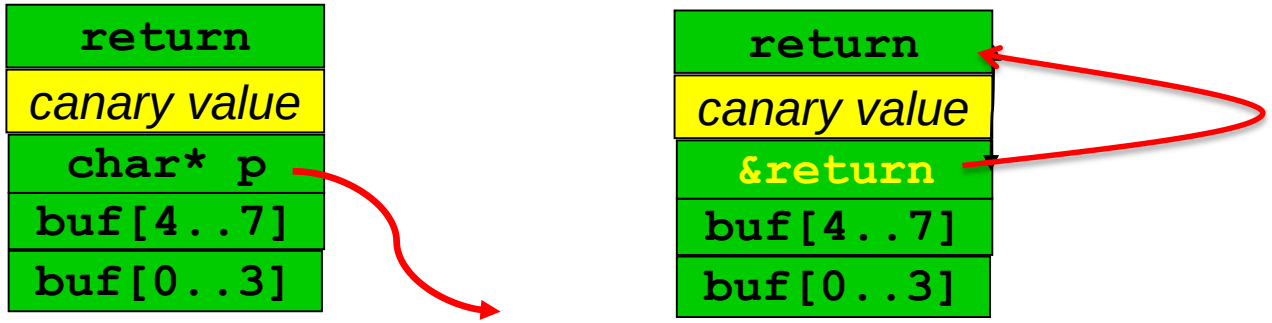
Stack with canary



stack canaries

A careful attacker can defeat this protection, by

- overwriting the canary with the correct value
- corrupting a pointer to point to the return address



Additional countermeasures:

- use a random value for the canary
- XOR this random value with the return address
- include string termination characters in the canary value, as some string copying functions cannot write these.

Further improvements of stack canaries

- re-order elements on the stack to reduce potential for trouble:
  - swapping parameters **`buf`** and **`fp`** on the stack changes whether overrunning **`buf`** can corrupt **`fp`**
    - this is especially dangerous if **`fp`** is a function pointer
  - better to allocated buffers 'above' other local variables

  First introduced by IBM's ProPolice

- a separate shadow stack for return addresses
  - to keep copies of return addresses, and use it to check for corrupted return addresses

  Of course, attackers 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
- 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. Address Space Layout Randomisation (ASLR)

- Attacker needs detailed information on memory layout , eg
  - to jump to specific piece of code
  - 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 OS 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 program Ming (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 (CFI): determine control flow graph (cfg) and monitor execution to spot such attacks

- many variants, with different levels of precision, overhead, ...

Note: not all attacks results in unusual control flow

- eg buffer overflows that only corrupt data will not, so cannot be detected by CFI

Control Flow Graph

For a large part, the cfg is static & can be determined at compile-time.

But complications in determining the cfg come from

- function calls through function pointers

  void **`sort(int[] src, int[] dest,`**
  `boolean (*compare)(int,int) )`

  But whole program analysis can determine the set of values a function pointer can take

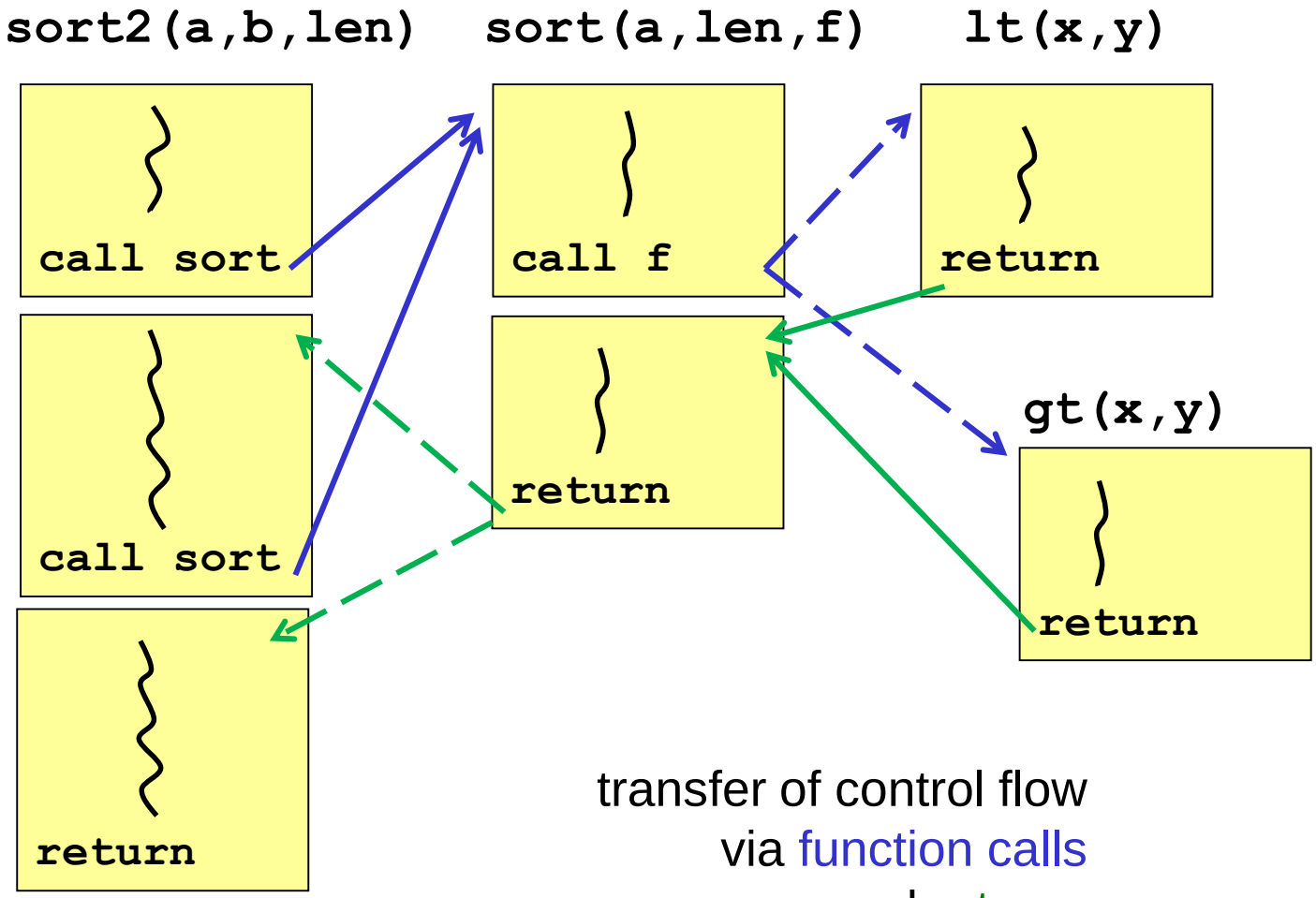
- return's from functions

  But whole program analysis can determine the set of valid return destitations

Example code

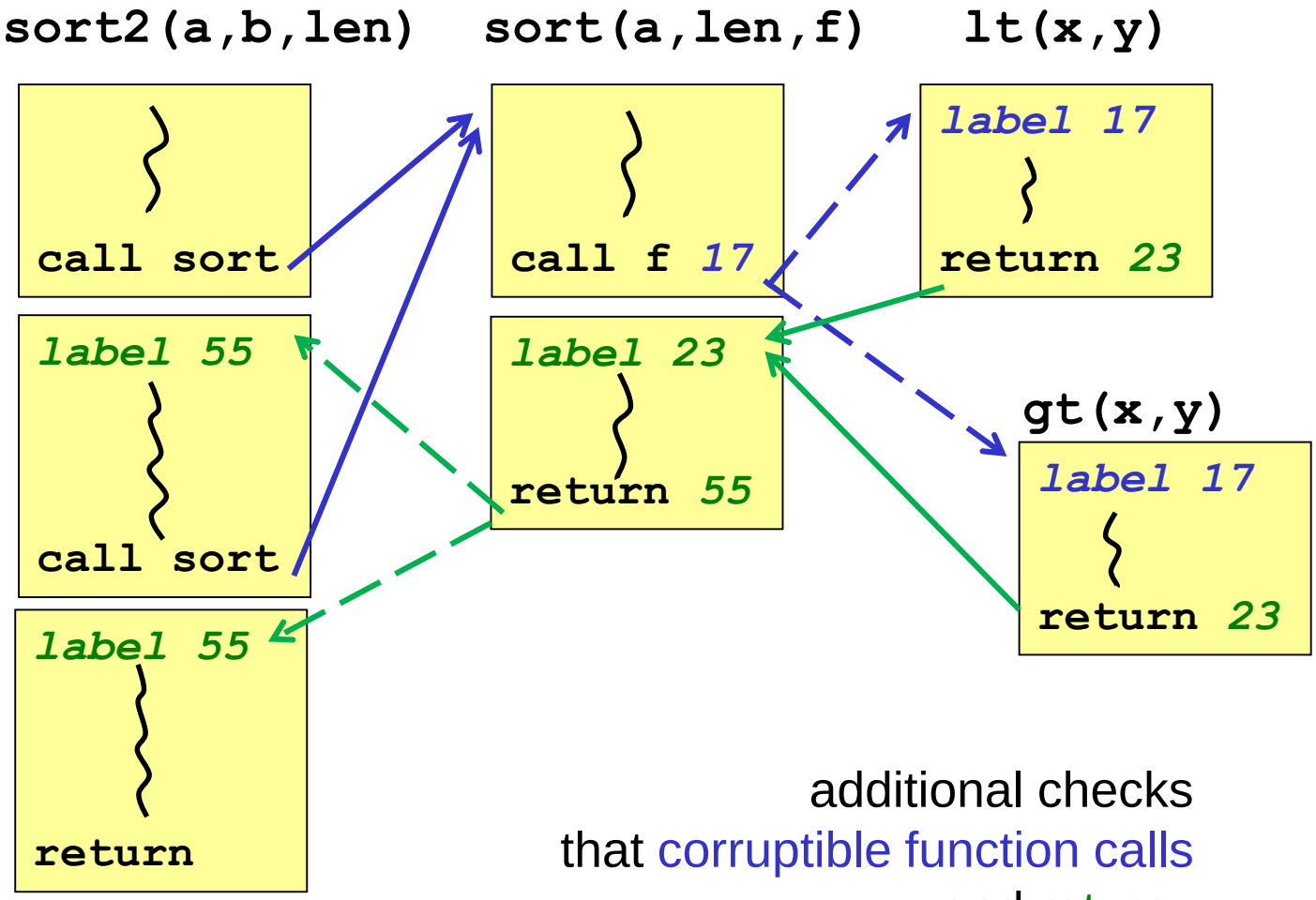
```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
   return x > y;
}
sort2(int a[ ], int b[ ], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```

Example control flow graph



transfer of control flow
via function calls
and returns

Example control flow checks



additional checks
that corruptible function calls
and returns
go to *legal destinations*

CFI with more precise return checks

A more precise check on return addresses could be enforced by recording a shadow stack for return addresses at runtime,

- to know which one of these possible return addresses the current call really came from

an option already mentioned as improvement of stack canaries

5. Pointer encryption

Many buffer overflow attacks involve corrupting pointers,
either pointers to data or function pointers.

PointGuard compiler extension encrypts pointers

- pointers encrypted in main memory, unencrypted in registers
  - simple & fast encryption scheme: XOR with a fixed value, randomly chosen when a process starts
- attacker can stll to corrupt (encrypted) pointers in memory, but these will not decrypt to predictable values
- NB this is using *encryption* to ensure *integrity*. This is normally NOT a good idea, but here it works.

[Cowan et al, PointGuard: protecting pointer from buffer overflow vulnerabilities, Usenix Security 2003]

6. Execution-aware memory protection

More fine-grained memory protection that normal OS provides:

- access control based on the value of the program counter, so that some memory region can only be accessed by a specific part of the program code
  - eg. crypto keys only accessible from the module with the encryption code
- The possible impact of an buffer overflow attack is the rest of the code is then reduced.

[Google, US patent 9395993 B2, July 2016]

[Koeberl et al., TrustLite: A security architecture for tiny embedded devices, *European Conference on Computer Systems*. ACM, 2014]