The heap

Limitations of the stack

```
int *table_of(int num, int len) {
  int table[len+1];
  for (int i=0; i <= len; i++) {
     table[i] = i*num;
  }
  return table; /* an int[] can be used as an int* */
}
```

What happens if we call the function above, with

```
int *table3 = table_of(3,10);
printf("5 times 3 is %i\n", *(table3+5));
                              // we use pointer arithmetic
                              // to mimic array indexing
```

Limitations of the stack

```
int *table_of(int num, int len) {
  int table[len+1];
  for (int i=0; i <= len; i++) {
     table[i] = i*num;
  }
  return table; /* an int[] can be treated as an int* */
}
```

What happens if we call the function above, with

```
int *table3 = table_of(3,10);
printf("5 times 3 is %i\n", *(table3+5));
int *table4 = table_of(4,10);
printf("5 times 4 is %i\n", *(table4+5));
printf("5 times 3 is %i\n", *(table3+5));
```

The function **`table_of`** is weird, because it returns a reference to a local variable **`table,`** but the memory for this variable is de-allocated when the function returns...

NB you should never write such code, and any decent compiler will warn about this. Still, it is legal C...

A cleaner solution would be to let the caller allocate the memory, and pass in a pointer to that

```
void table_of(int num, int len, int[] table)
```

but note this only works because we know the size of the table needed beforehand.

This is a general limitation of stack-allocated memory:

how can a function allocate some memory that can later still be used by the caller?

the heap – for dynamic memory

(De)allocation of stack memory is very fast, but has its limitations:

- any data we allocate in a function is gone when it returns

Solution: the heap

- The heap is a large piece of scrap paper where functions can create data that will outlive the current function call.
- Functions can use it to share values, using pointers to data stored on the heap
- It is up to the program(er) to organise this; the OS will only keep track of which part of the scrap paper is still unused

malloc

The operation to allocate a chunk of memory on the heap is **malloc**

```
#include <stdlib>
 void* malloc (size_t size)
                // size_t is an unsigned integral type
```

returns an pointer to a contiguous block in memory of **size** bytes,
or NULL if an error occurs

Example use

```
int *table = malloc(len*sizeof(int));
 // allocates enough memory for len int's
```

void* ?

Recall that pointers are typed,

eg **`int*`** is the type of pointers to an **`int`**.

`void*` is the type of untyped pointers.

`malloc` just returns a pointer to a blob of memory, and the result does not have any specific type (yet), so its return type is **`void*`**

A **`void*`** pointer can be converted into any other pointer type, without an explicit cast.

NULL ?

- **`NULL`** is a special value, which is guaranteed to be different from any legal address

  So **`&p`** will never return **`NULL`** for any properly allocated variable **`p`**

- Dereferencing a null pointer, eg

```
int* ptr = NULL;
int i = *ptr;
```

  leads to undefined behaviour:

  the program probably crashes, but basically anything can happen.

  So you should never dereference a **`NULL`** pointer

Check for malloc failure!

Malloc may fail, namely if there is not enough heap space available, in which case it returns **`NULL.`**

Programs should always check the result of malloc!!!

Eg directly after

```
int *table = malloc(len*sizeof(int));
```

there should be a line like

```
if (table == NULL) { exit();}
```

or

```
if (!table) { return -1;}
      // NULL is interpreted as false,
      // so !table will be true when table is NULL.
      // Writing table==NULL is probably clearer?
```

free

You, the programmer, are in charge of freeing heap memory that is no longer needed, by calling

```
void free (void* p)
```

Normal usage pattern

```
long *p = (long*) malloc (10*sizeof(long));
if (p == NULL) { exit();}
... // use p
free(p); // when p is no longer needed
```

Here **free** and **malloc** can be in different functions

Heap vs stack

Data can be allocated on the **stack**
or on the **heap** (aka **dynamic memory**)

- Data on the stack is allocated automatically when we do a function call, and removed when we return

```
f() { ... int table[len]; .... }
```

- Data on the heap must be allocated and de-allocated manually, using **malloc** and **free**

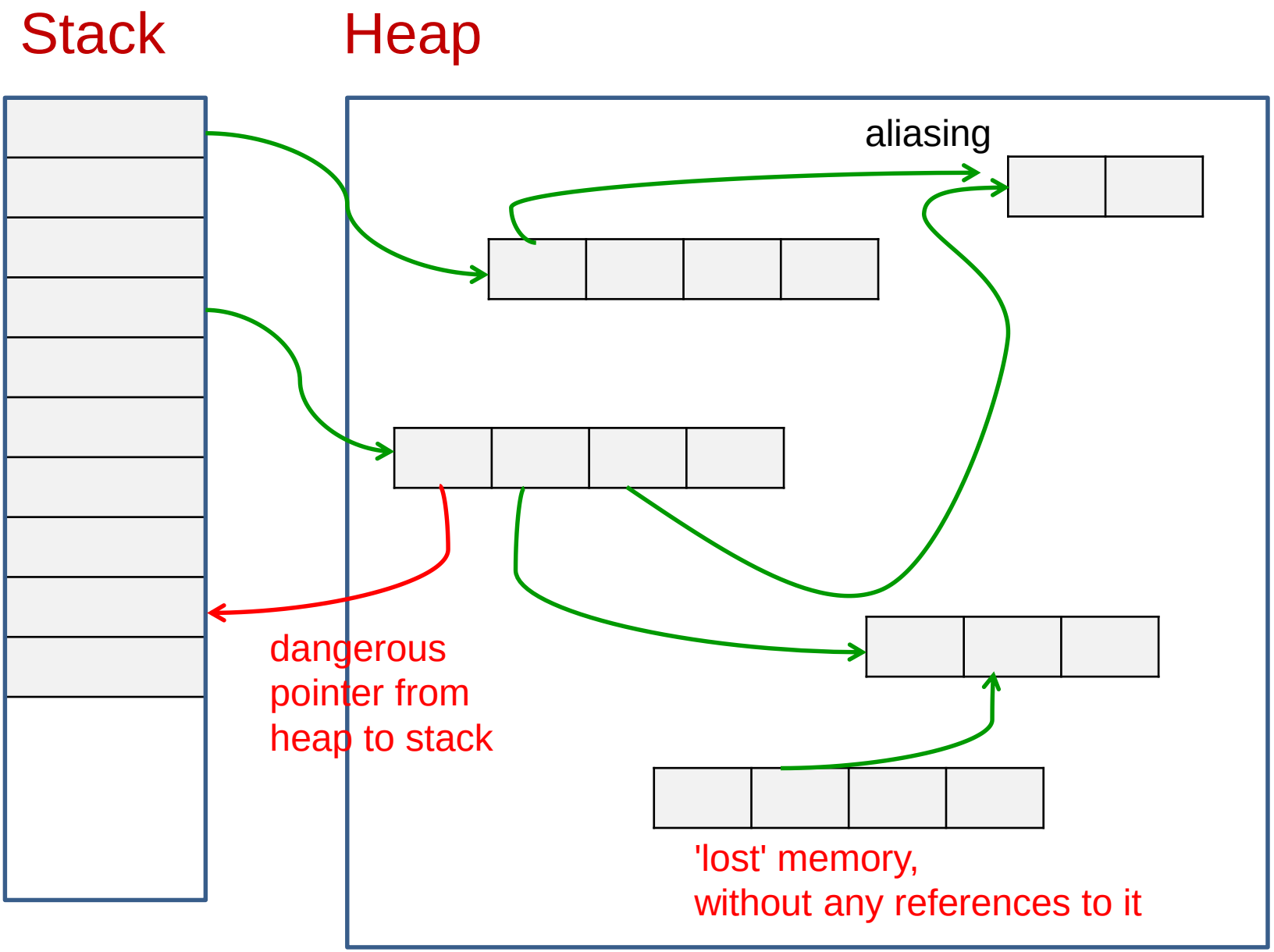
```
int *table = malloc(len*sizeof(int));
if (table == NULL) { ...  // How to proceed? }
...
free(table);
```

Stack, heap & pointers

- To use data on the heap, we *must* use pointers!
  - otherwise the data is lost and we cannot use it

- Pointers to data allocated on the heap can be
  - on the stack
  - in the heap itself

You can have pointers from the heap to the stack,
but typically you do not need them, or want them!


Stack
Heap
aliasing
dangerous
pointer from
heap to stack
'lost' memory,
without any references to it

Stack & heap issues

Memory (security) problems

C(++) does not provide memory-safety

Malicious, buggy, and insecure code can access data *anywhere* on the heap and stack

- by doing pointer arithmetic
- by overrunning array bounds

More generally, security problems with memory can be due to

1. running out of memory
2. lack of initialisation of memory
3. bugs in program code

esp for heap, as dynamic memory is more complex & error-prone

Hence MISRA-C guidelines for safety-critical software include

Rule 20.4 (REQUIRED) Dynamic heap allocation shall not be used

Running out of stack memory (aka stack overflow)

- Max size of the stack is finite and typically fixed on start-up of a process
- Normally, stack overflow will simply crash a program
  - demo! see .../hic/code/lecture3/stack_overflow.c

- *Are there sensible alternatives?*
- *Are there more dangerous alternatives?*

Memory initialisation

What will this program print?

```
int i;
printf("i is %i .\n", i); // %i to print int
```

In C memory is *not initialised*, so `i` can have any value.

- Demo: see .../hic/code/lecture4/print_unitinitialsed.c

Some programming language do provide a default initialisation.

Why is that nicer and more secure?

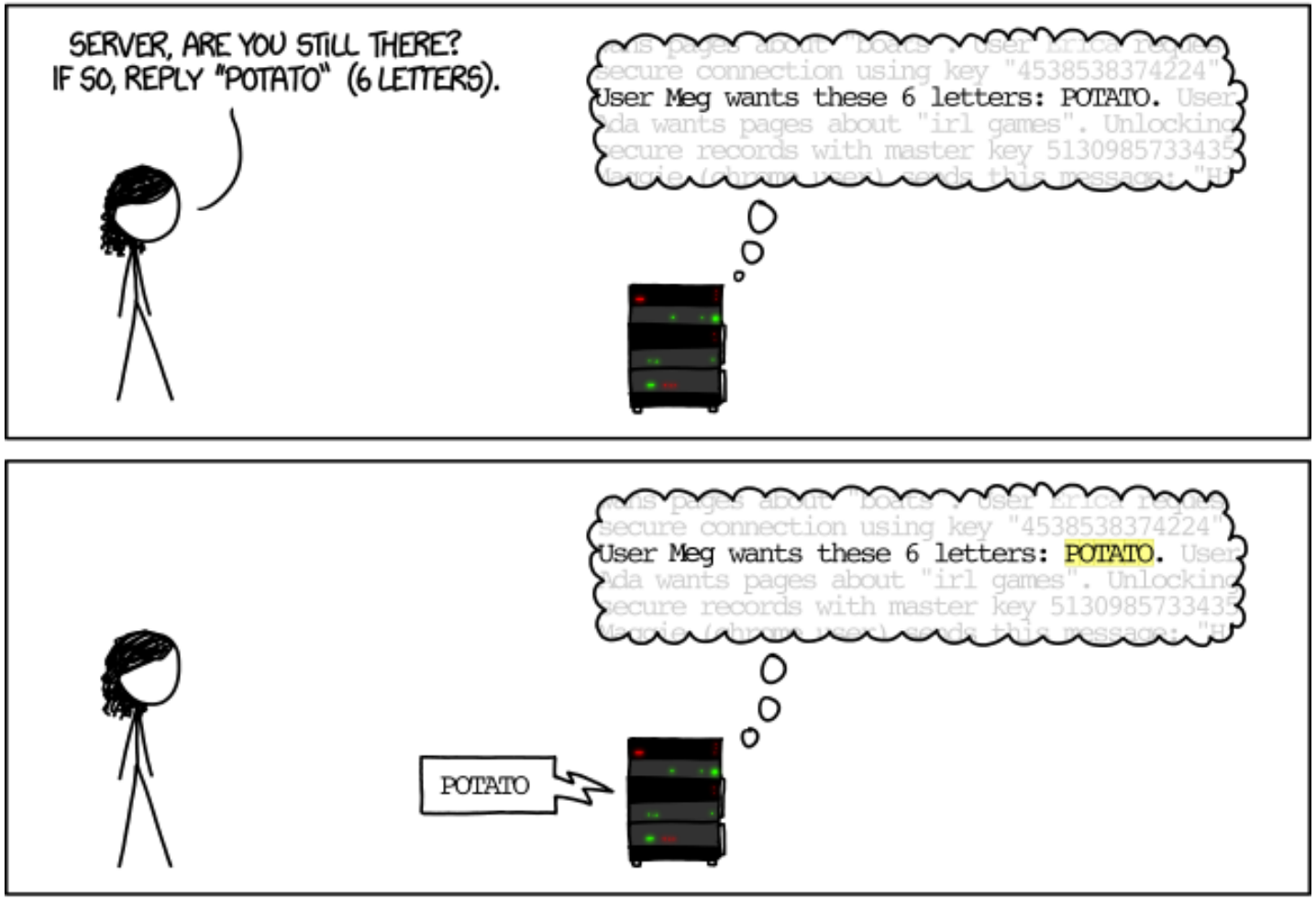
- programs behave more deterministic; a program with uninitialized variables can behave differently each time it's run, which is not nice esp. when debugging
- for security: by reading uninitialized memory, a program can read confidential memory contents left there earlier

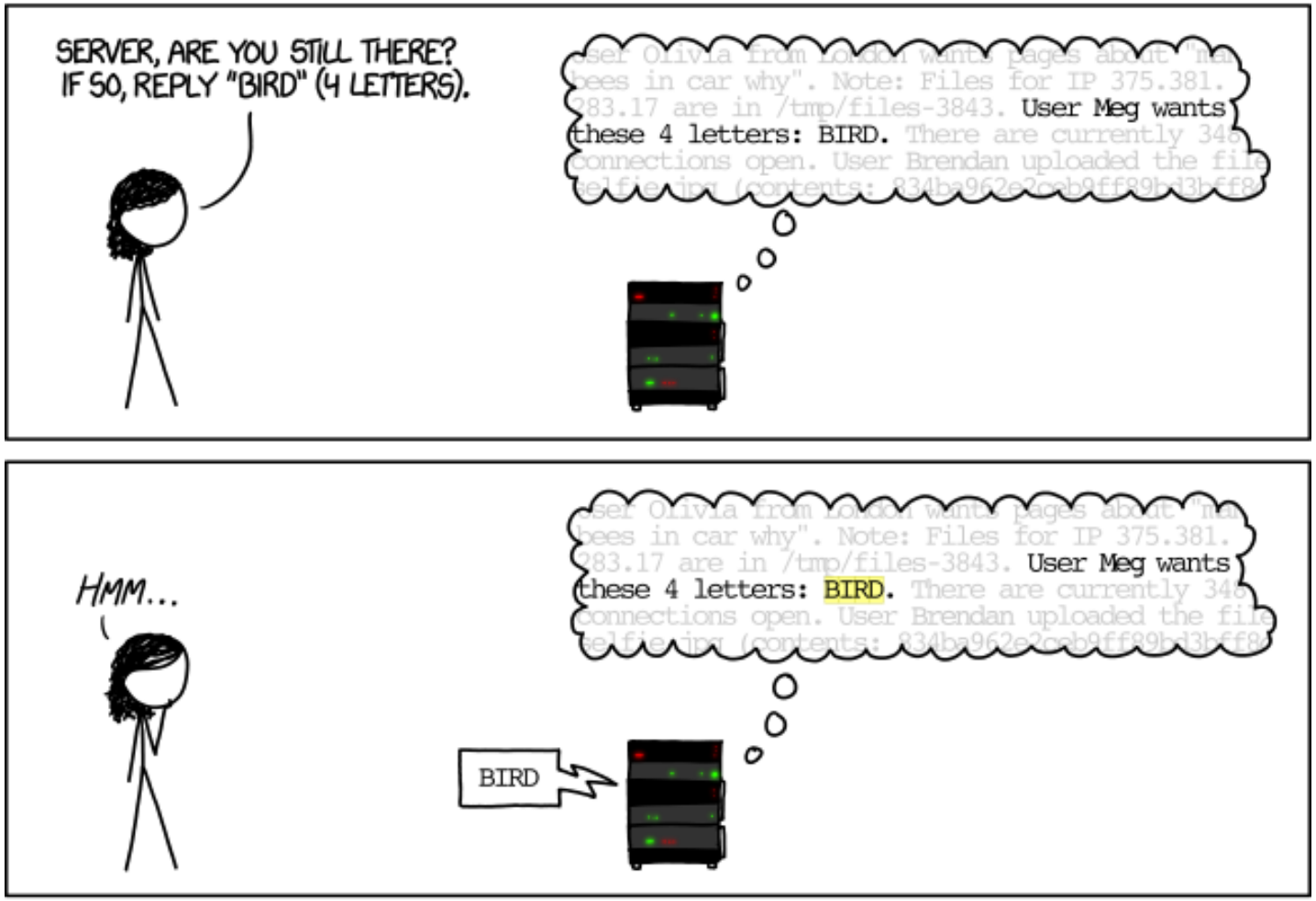
Heartbleed bug

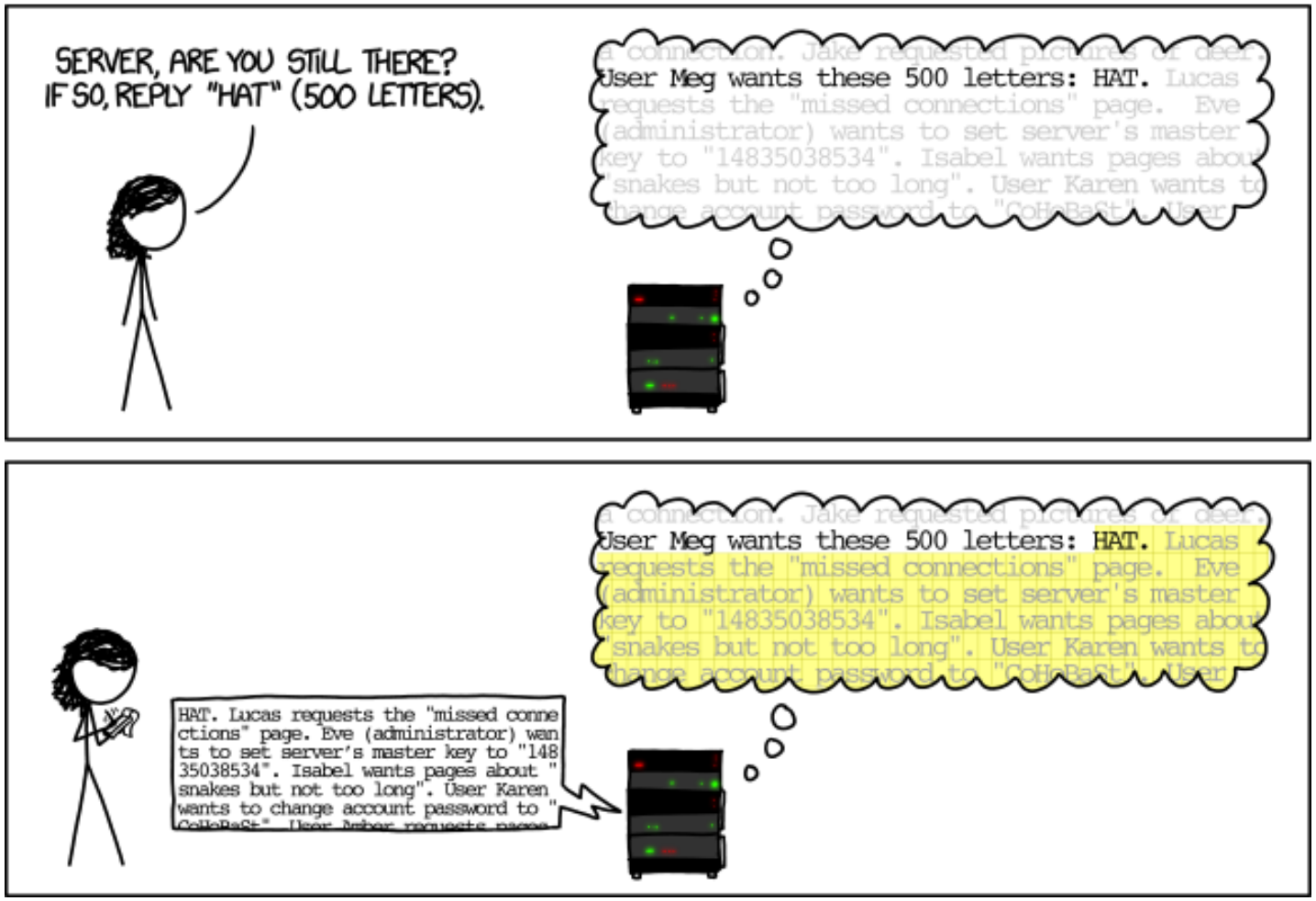
- bug in OpenSSL implemetation of SSL/TLS
- CVE identifier: CVE-2014-0160

- The heartbeat functionality in SSL allows you to check if connection is still alive
- strange input to OpenSSL could make OpenSSL print a large part of the stack, possible containing private keys

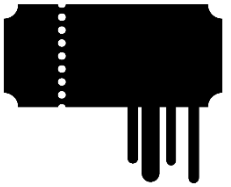
HOW THE HEARTBLEED BUG WORKS:




SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).
these 4 letters: BIRD.
HMM...
these 4 letters: BIRD.
BIRD


SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to
User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to
HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "

Ticketbleed (CVE-2016-9244)



- Bug in TLS implementation of F5 BIG IP firewalls & load balancers
  - revealed Feb 11 2017
- allows a remote attacker to extract up to 31 bytes of memory
- root cause the same as Heartbleed
- ironic that flaw in firewall introduces securiy flaw

[https://arstechnica.com/security/2017/02/newly-discovered-flaw-undermines-https-connections-for-almost-1000-sites]

[https://filippo.io/Ticketbleed]

`calloc`

Memory allocated on the heap with **`malloc`** is typically not initialised

- Many OSs will zero out memory for a new process, but recycling of memory within that process means that malloc-ed memory may contain old junk.
- If OS does not zero out memory for new processes, you can access confidential information left in memory by other processes by malloc-ing large chunks of data!

The function **`calloc`** will initialise the memory it allocates, to all zeroes

- downside: this is slower
- upside: This is good for security and for avoiding accidential non-determinism due to missing initialisation in a (buggy) program
- But, in security-sensitive code, you may still want to zero out confidential information in memory yourself before you free it

Stack vs heap allocation

Consider `main() {while (true) { f(); } }`

Difference in behaviour for the two functions `f()` below?

```
void f(){
  int x[300]; x[0]=0;
  for (int i=1; i<300; i++) {x[i] = x[i-1]+i;}
  printf("Result: %i \n", x[299]);
}
```

possible stack overflow
(but unlikely for this non-recursive code)

```
void f(){
  int *x = malloc(300*sizeof(int));
  x[0]=0;
  for (int i=1; i<300; i++) {x[i] = x[i-1]+i;}
  printf("Result: %i \n", x[299]);
}
```

malloc may fail;
possible heap overflow

memory leak!
the memory for x is not freed, so `main` *will* crash when heap memory runs out

Heap allocation – and de-allocation, done right

Correct and *secure* version of function f that uses the heap:

```
void (f){
     int *x = malloc(300*sizeof(int));
     if (x==NULL) { exit(); }
     x[0]=0;
     for (int i=1; i<300; i++) {x[i] = x[i-1]+i;}
     printf("result is %i \n", x[299]);
     free(x); // to avoid memory leaks
  }
```

Moral of the story: heap allocation is more work for the programmer

Btw, the code above is a bit silly: it allocates heap memory that is only used within one procedure; we might as well use the stack.

Heap problems: memory leaks

Memory leaks occur when you forget to use free correctly.

Programs with memory leaks will crash if they run for long enough.

You may also notice programs running slower over time if they leak memory.
Restarting such a program will help, as it will start with a fresh heap

More heap problems: dangling pointers

Never use memory after it has been de-allocated

```
int *x = malloc (1000);
free (x);
...
print("Let's use a dangling pointer %s", x);
```

A pointer to memory that has been de-allocated (freed) is called a dangling pointer. When using dangling pointers, all bets are off...

More heap problems: using free incorrectly

- Never free memory that is not dynamically allocated

```
char *x = ”hello”;
free (x); // error, since ”hello”
          // is statically allocated
```

- Never double free

```
char *x = malloc (1000);
free (x);
...
free (x); // error
```

Memory management trouble: spot the bug

```
int *x = malloc (1000);
int *y = malloc (2000);
y = x;
```

memory leak!
we cannot access the 2000 bytes that y pointed to, and we cannot free them!

Aliasing – spot the bug!

Aliasing can make some of these bugs hard to spot

```
int *x = malloc (1000);
int *y = malloc (2000);
int *z = x;
y = x;
int *w = y;
free (w);
free (z);
```

double free! **this memory is already de-allocated in the line above**

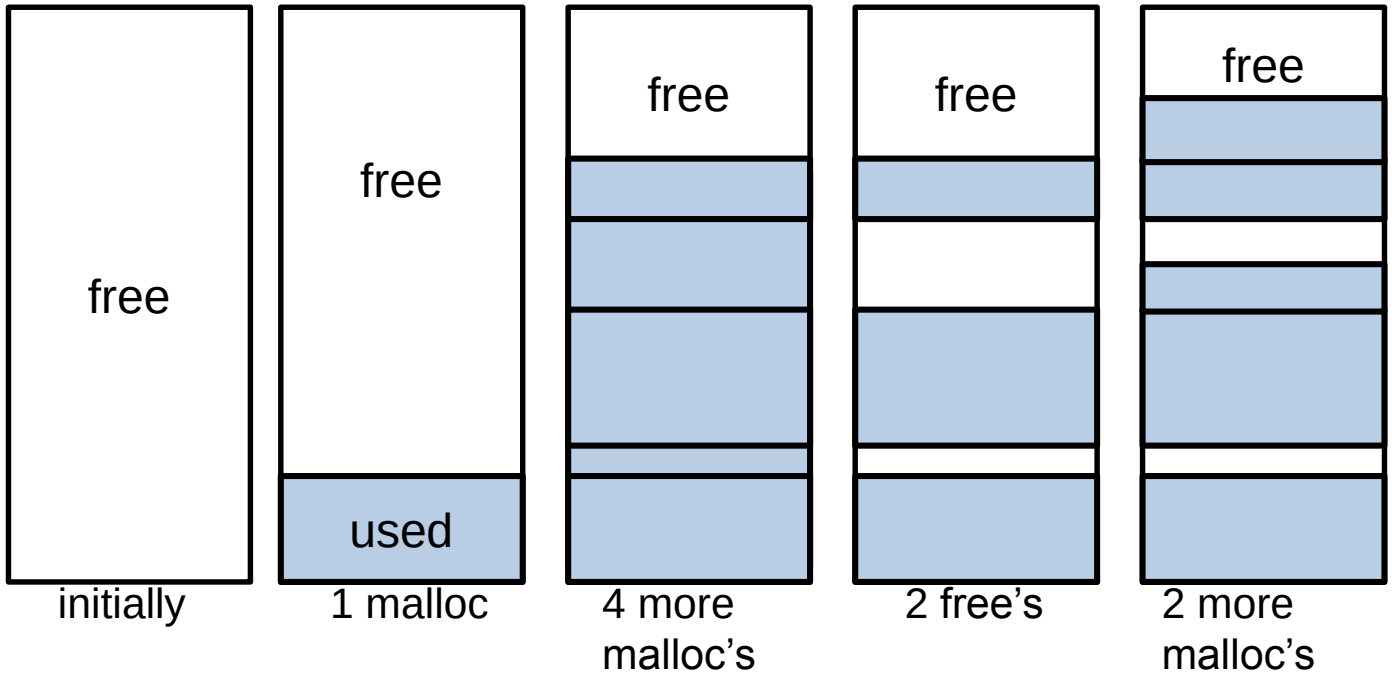
Recall: pointers are called aliases if they both point to same address

Aliasing, together with the fact that **`malloc`** and **`free`** can happen in different places of the program, make dynamic memory management extremely tricky!!

Heap memory management

The implementations of **`malloc`** and **`free`** have to keep track of which parts of heap are still unused.

- Initially, the free memory is one contiguous region.
- As more blocks are malloc-ed and freed, it becomes messier

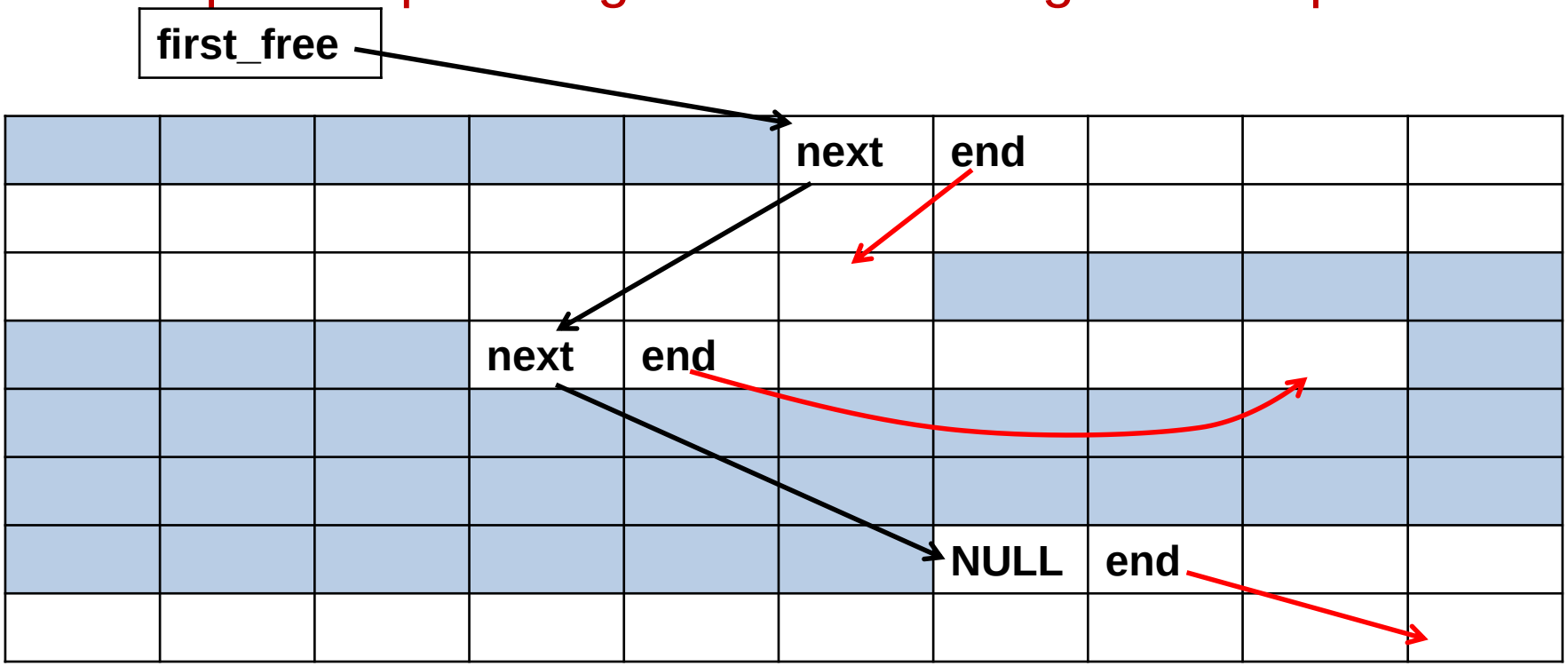


Heap memory management

The implementations of **`malloc`** and **`free`** have to keep track of which parts of heap are still unused.

How would you do this?

Example heap management: recording free heap chunks



Inside each free chunk

- a pointer to the next free chunk
- a pointer to the end of the current free chunk

Not very efficient: real **`malloc`** and **`free`** do more admin for efficiency

Heap memory management

One way is to maintain a free list of all the heap chunks that are unused.

- This info can be recorded on the heap itself, namely in the unused parts of the heap.
- You can also maintain meta-information *in* the used chunks on the heap to help in de-allocation (eg the size of the chunk)
- NB an attacker can try to corrupt any this data!

Padding malloc-ed data to a round number reduces fragmentation.

The programmer can make memory management easier and reduce fragmentation by often allocating chunks of data of the same size.

Malloc-ed data can *not* be moved or shifted on the heap, because this will break pointers to that data!

Garbage collection

In modern programming languages (Java, C#, ...), instead of the programmer having to free dynamic memory, there is a garbage collector which automatically frees memory that is no longer used.

Advantage: much less error-prone

Disadvantage: performance

- Garbage collection is an expensive operation (it involves analysis of the entire heap), so garbage collection brings some overhead.
- Moreover, garbage collection may kick in at unexpected moments, temporarily resulting in a very bad response time.

  Still, there are clever garbage collection schemes suitable for real-time programs.

Recap: stack vs heap

Stack

- variables are allocated and de-allocated automatically
- allocation is much faster than for the heap
- data on the stack can be used without pointers
- data needs have to be known at compile time
- stack space may run out, eg. due to infinite recursions
- max size of the stack usually fixed by OS when program starts

Heap

- (de)allocation has to be done manually by the programmer;
  this is highly error-prone!
- allocation of heap memory slower than for stack memory
- to access data on the heap, you *must* use pointers;
  this is also error-prone!
- more flexible, and must to be used when data needs are not known at compile time
- heap space may run out too, but can grow during the lifetime

Lack of memory protection

Data is typically not initialised when allocated

- except static global variables, memory allocated by **`calloc`**, and possibly fresh heap memory allocated by **`malloc`** the first time it is used

Irrespective of whether we store data on the heap or the stack:

malicious code and *buggy (insecure)* code

can access data *anywhere* on heap or stack, eg

- by doing pointer arithmetic
- by overrunning array bounds

Buggy or insecure code acting on malicious input supplied by an attacker can be used for malicious purposes.

Malicious, buggy or insecure code can be in libraries or in, say, a browser plugin.