Heap Overflows and Data Execution Prevention (DEP)

CS 491/591

Fall 2015

No more code on the stack!

DATA EXECUTION PREVENTION

Defense: What can we do?

1. Get programmers to write better code?
   - Good luck with that…

2. Get programmers to use a safe language?
   - Maybe someday, not today…

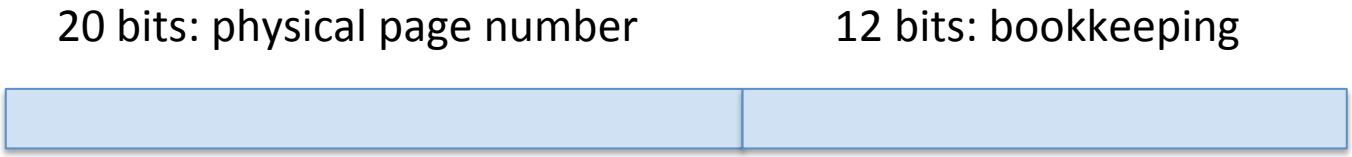
3. Modify the compiler?

4. Modify the OS and hardware?

Data Execution Prevention (DEP) aka NX bit, aka W^X

- Idea: Make data regions non-executable
- Old implementation: Use x86 segmentation
- New implementation: Use new hardware protection features

Page Table Entry Structure



Bookkeeping:

0/1 – Valid? – Is this a valid entry? Or is this entry empty?

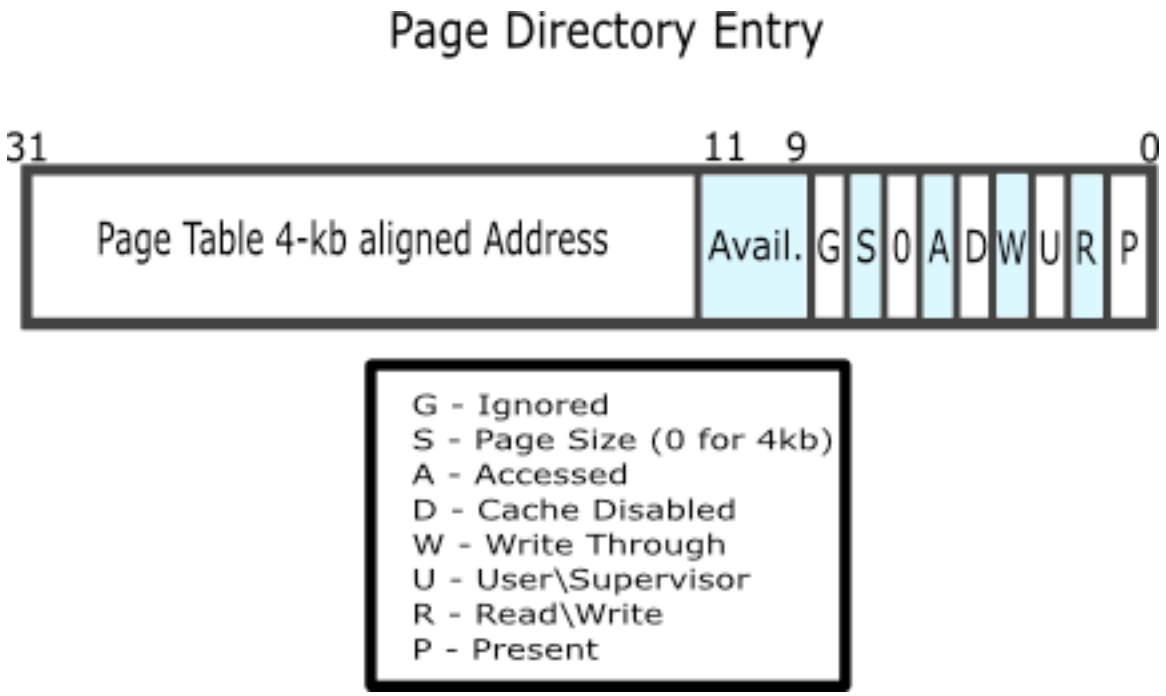
0/1 – "Dirty" bit – Has the page been modified since it was last saved to disk?

0/1 – Privileged? – Allow access to all programs, or only to privileged?

0/1 – Write – Allow writes to this page?

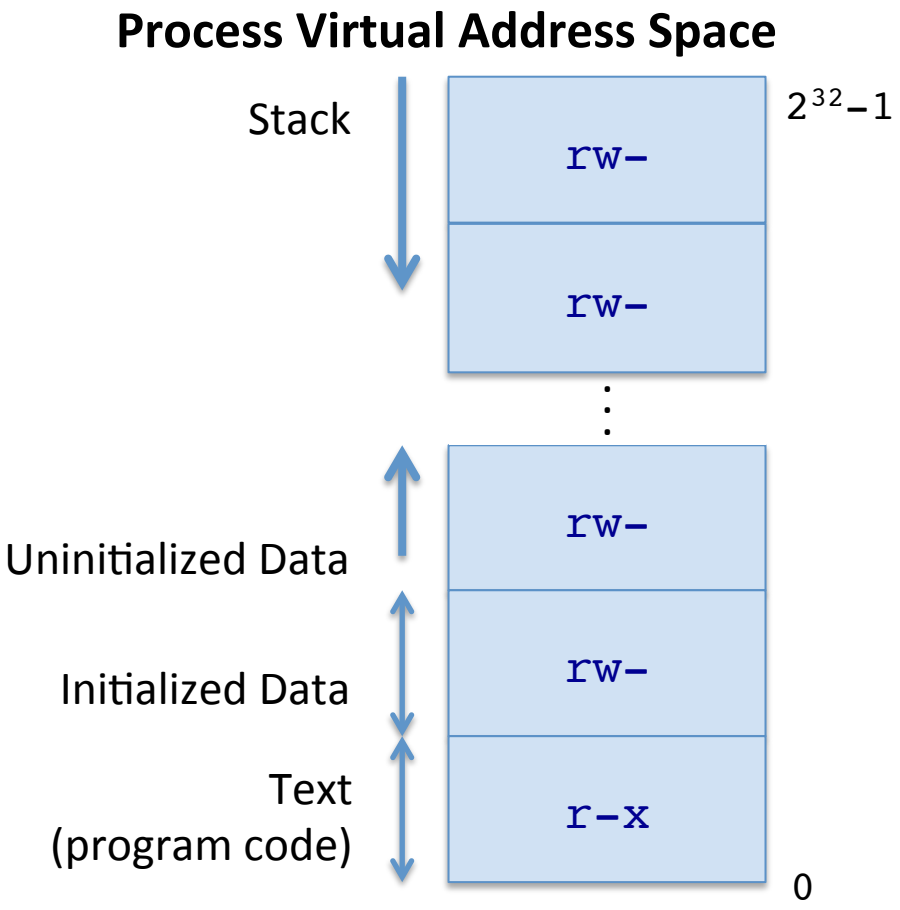
0/1 – Execute – Allow executing this page as code? **New! Data Execution Prevention (DEP)**
(Intel: Execute Disable (ED) AMD: No Execute (NX))

Page Table Entries on x86



Credit: http://wiki.osdev.org/Paging

Data Execution Prevention



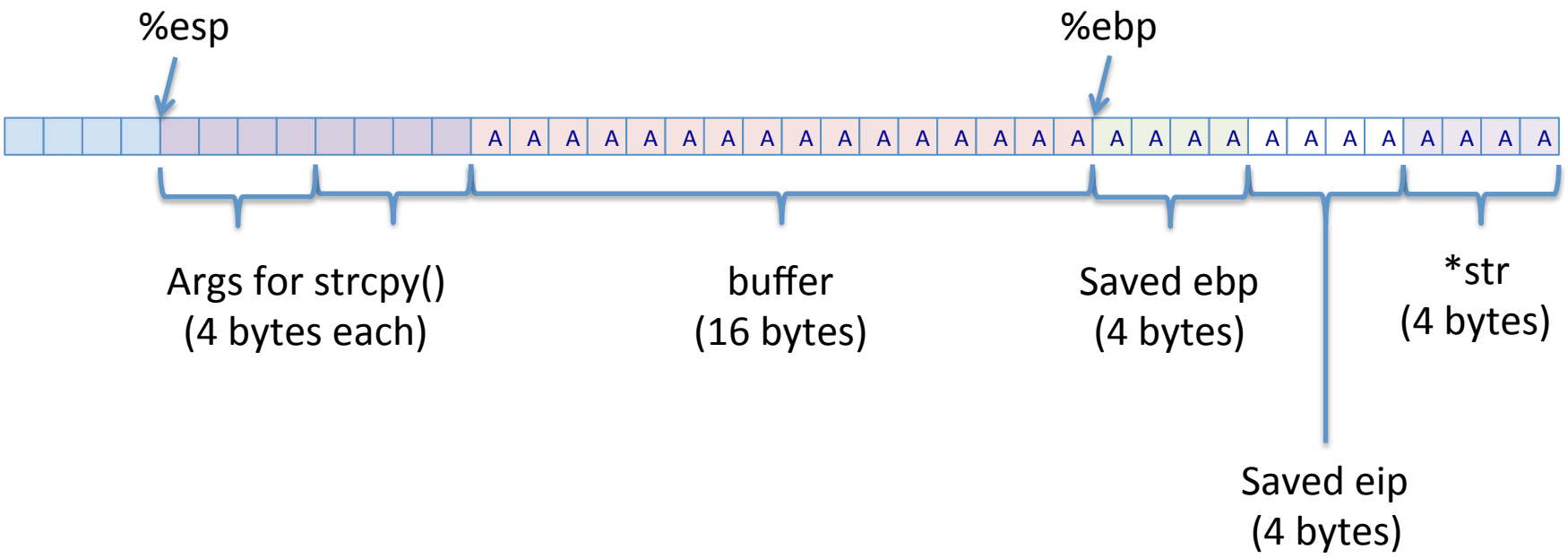
No page has both write (w) and execute (x) permissions

Even if the attacker can inject code, he can't execute it!

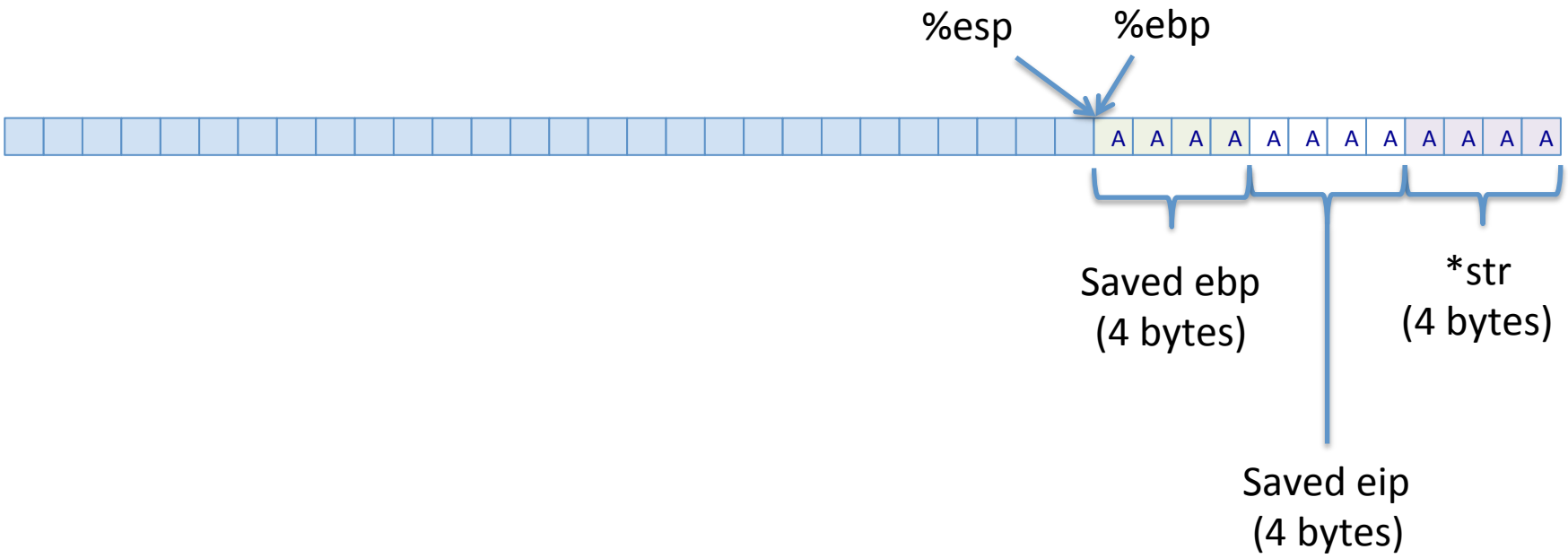
Remember Example 2?

```
#include <string.h>

void function(char *str) {
  char buffer[16];
  strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for(i=0; i < 255; i++)
    large_string[i]='A';

  function(large_string);
}
```

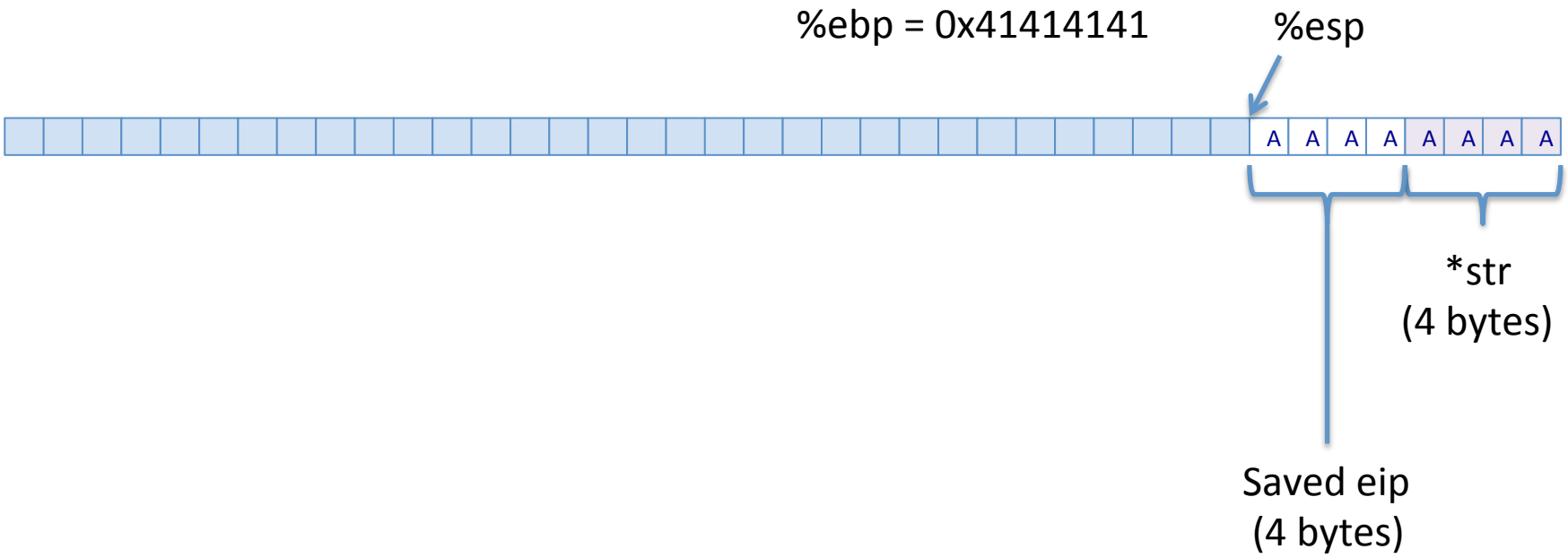
Example 2 – After strcpy()



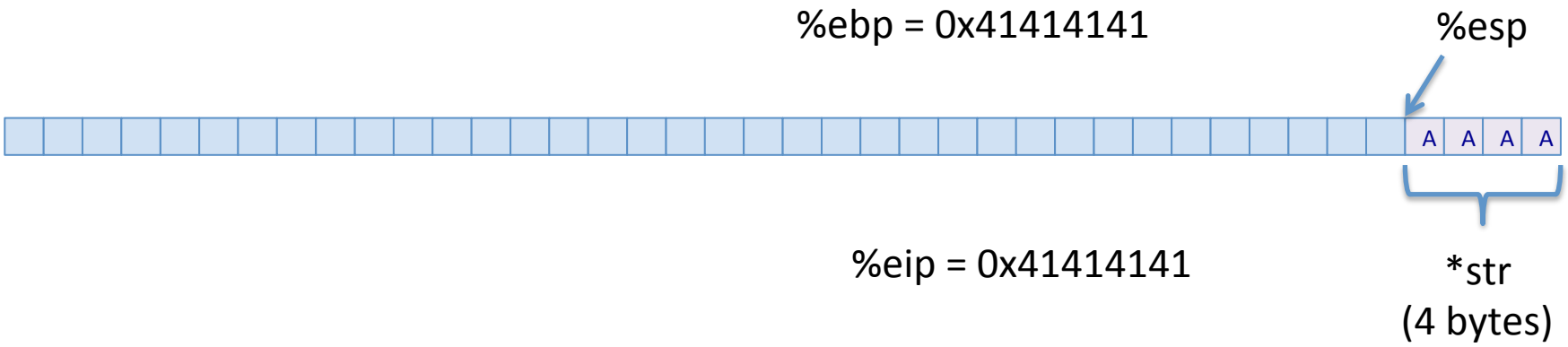
Example 2 – **leave** (1)



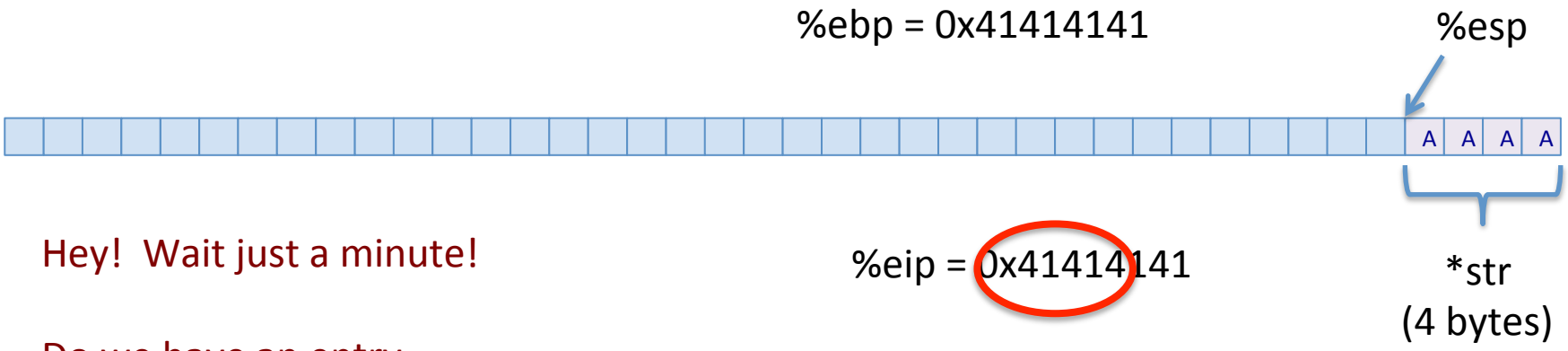
Example 2 – **leave** (2)



Example 2 – **ret**



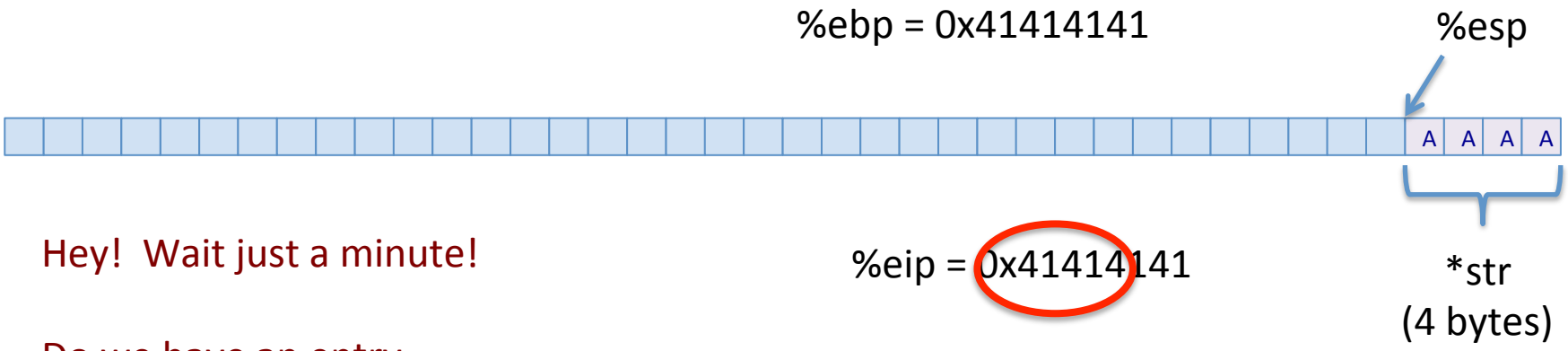
Example 2 – **ret**



Hey! Wait just a minute!

Do we have an entry
in our page table for
virtual page # 0x41414 ?

Example 2 – **ret**

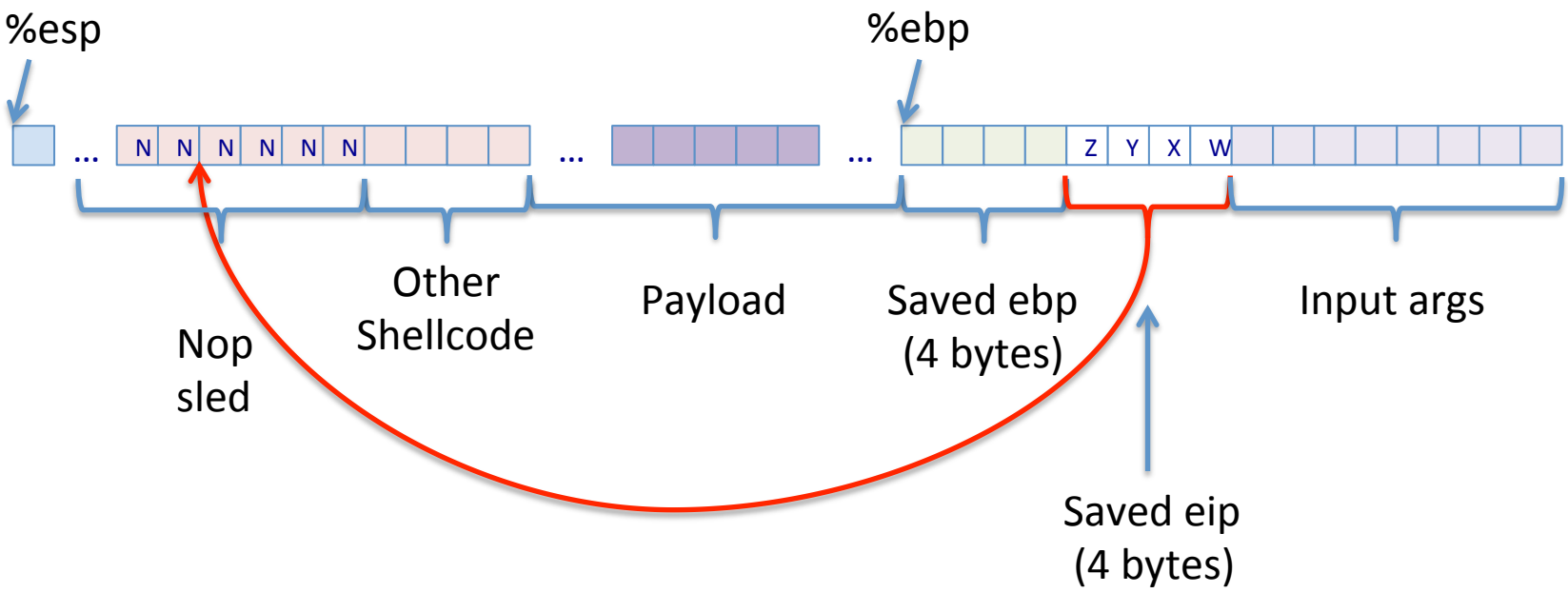


Hey! Wait just a minute!

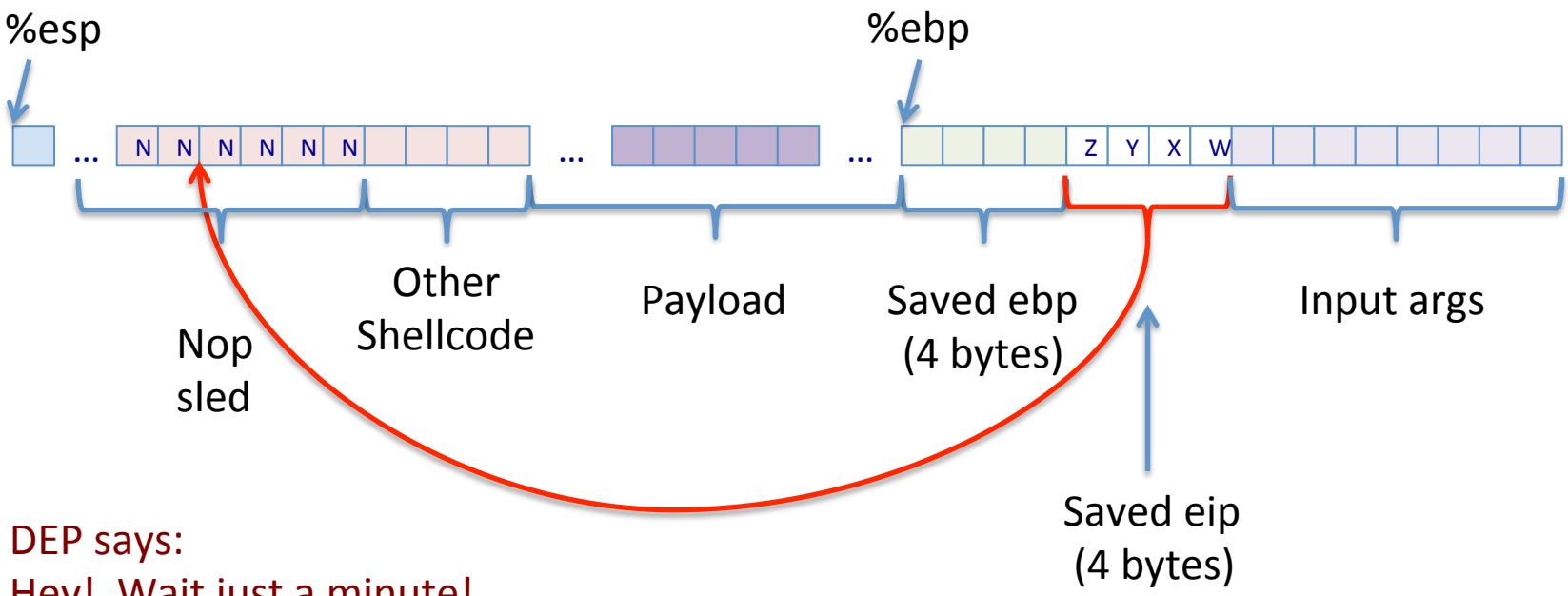
Do we have an entry
in our page table for
virtual page # 0x41414 ?

MMU says "No" → Segmentation Fault!

Stack Buffer Overflow



Stack Buffer Overflow with DEP



DEP says:
Hey! Wait just a minute!

Do we have execute permissions in our page table for the virtual page with WXYZ? (the stack)

MMU says "No" → Segmentation Fault!

Smashing more than just the stack

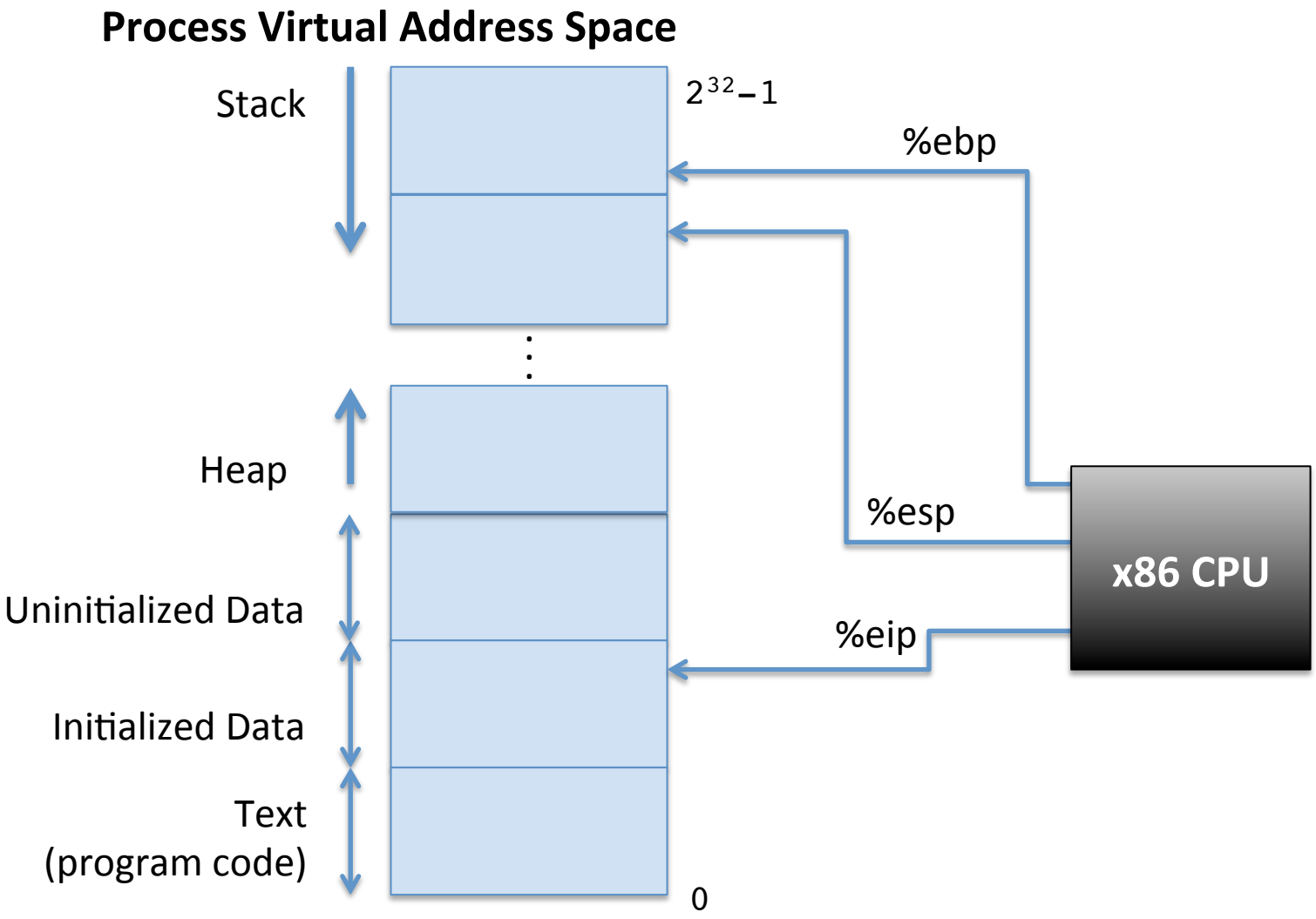
HEAP OVERFLOWS

General Heap Overflow Problems

- Background
  - What is the heap?
  - Why is it a valuable target?
- Simple heap buffer overflow attacks
  - Changing filenames
  - Changing integers
  - Changing function pointers

What is the heap?

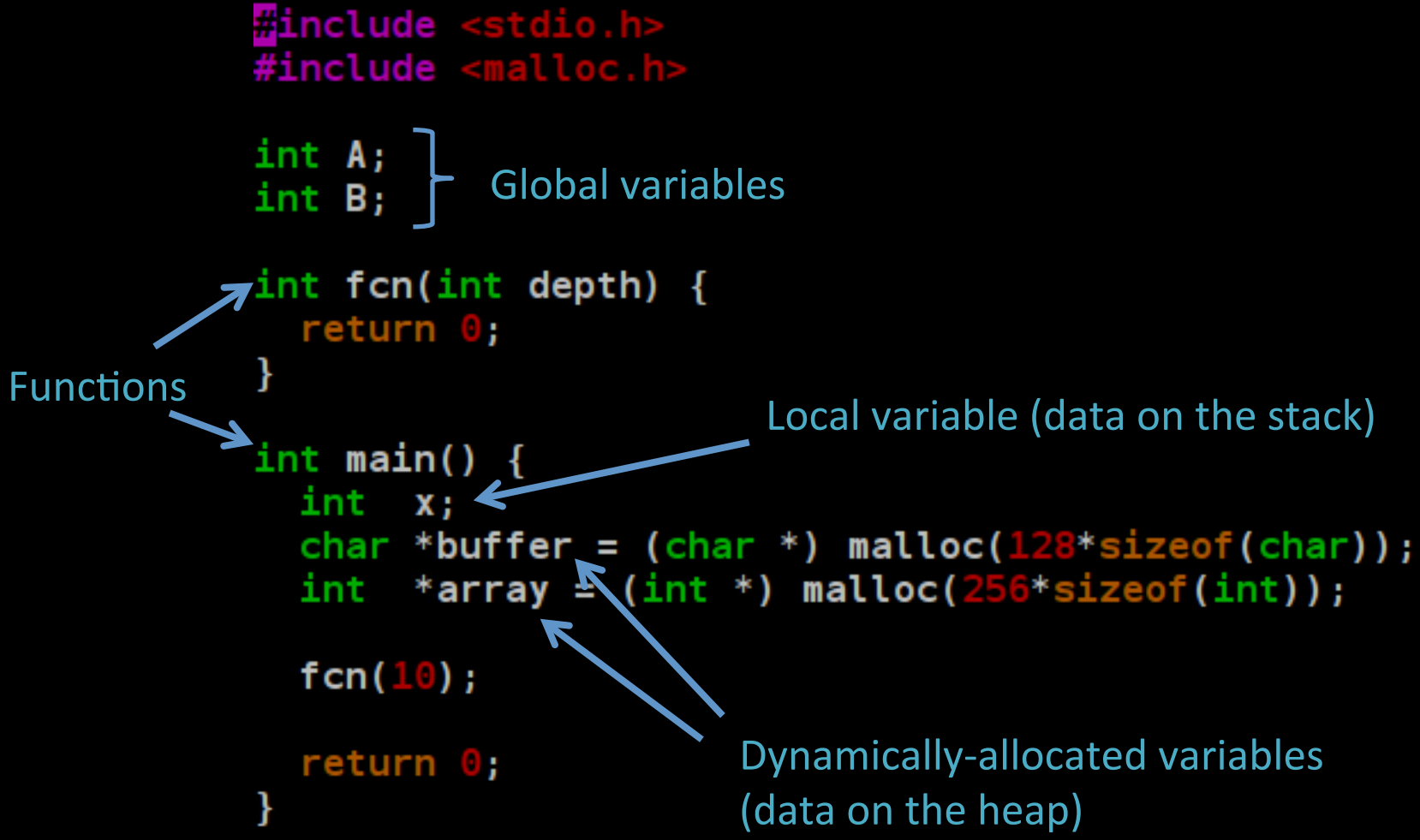
In-Memory Layout of a Process



Example Program

```c
#include <stdio.h>
#include <malloc.h>

int A;
int B;

int fcn(int depth) {
  return 0;
}

int main() {
  int  x;
  char *buffer = (char *) malloc(128*sizeof(char));
  int  *array = (int *) malloc(256*sizeof(int));

  fcn(10);

  return 0;
}
```

Example Program



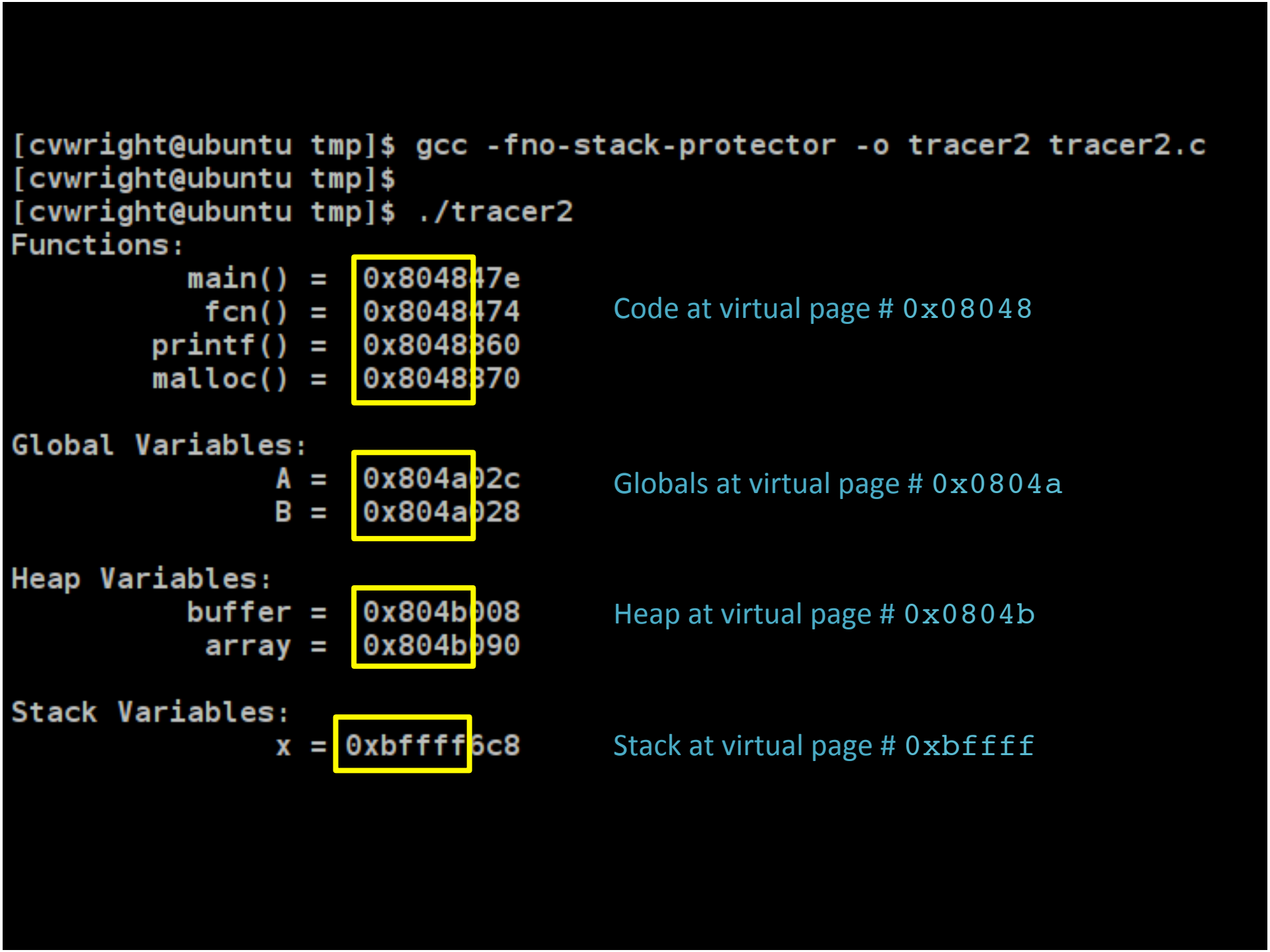
Let's add some instrumentation to help us see what's going on

```c
int main() {
  int  x;
  char *buffer = (char *) malloc(128*sizeof(char));
  int  *array = (int *) malloc(256*sizeof(int));
  void *main_ptr = main;
  void *fcn_ptr = fcn;
  void *x_ptr = &x;
  void *printf_ptr = printf;
  void *malloc_ptr = malloc;
  void *A_ptr = &A;
  void *B_ptr = &B;

  printf("Functions:\n");
  printf("\t  main() = %10p\n", main_ptr);
  printf("\t   fcn() = %10p\n", fcn_ptr);
  printf("\tprintf() = %10p\n", printf_ptr);
  printf("\tmalloc() = %10p\n", malloc_ptr);
  printf("\n");
  printf("Global Variables:\n");
  printf("\t       A = %10p\n", A_ptr);
  printf("\t       B = %10p\n", B_ptr);
  printf("\n");
  printf("Heap Variables:\n");
  printf("\t  buffer = %10p\n", buffer);
  printf("\t   array = %10p\n", array);
  printf("\n");
  printf("Stack Variables:\n");
  printf("\t       x = %10p\n", x_ptr);
  printf("\n\n");

  fcn(10);

  return 0;
}
```

Get addresses of variables in memory

Print addresses in hex

```
[cvwright@ubuntu tmp]$ gcc -fno-stack-protector -o tracer2 tracer2.c
[cvwright@ubuntu tmp]$
[cvwright@ubuntu tmp]$ ./tracer2
Functions:
          main() =  0x804847e
           fcn() =  0x8048474
        printf() =  0x8048360
        malloc() =  0x8048370

Global Variables:
               A =  0x804a02c
               B =  0x804a028

Heap Variables:
          buffer =  0x804b008
           array =  0x804b090

Stack Variables:
               x = 0xbffff6c8
```

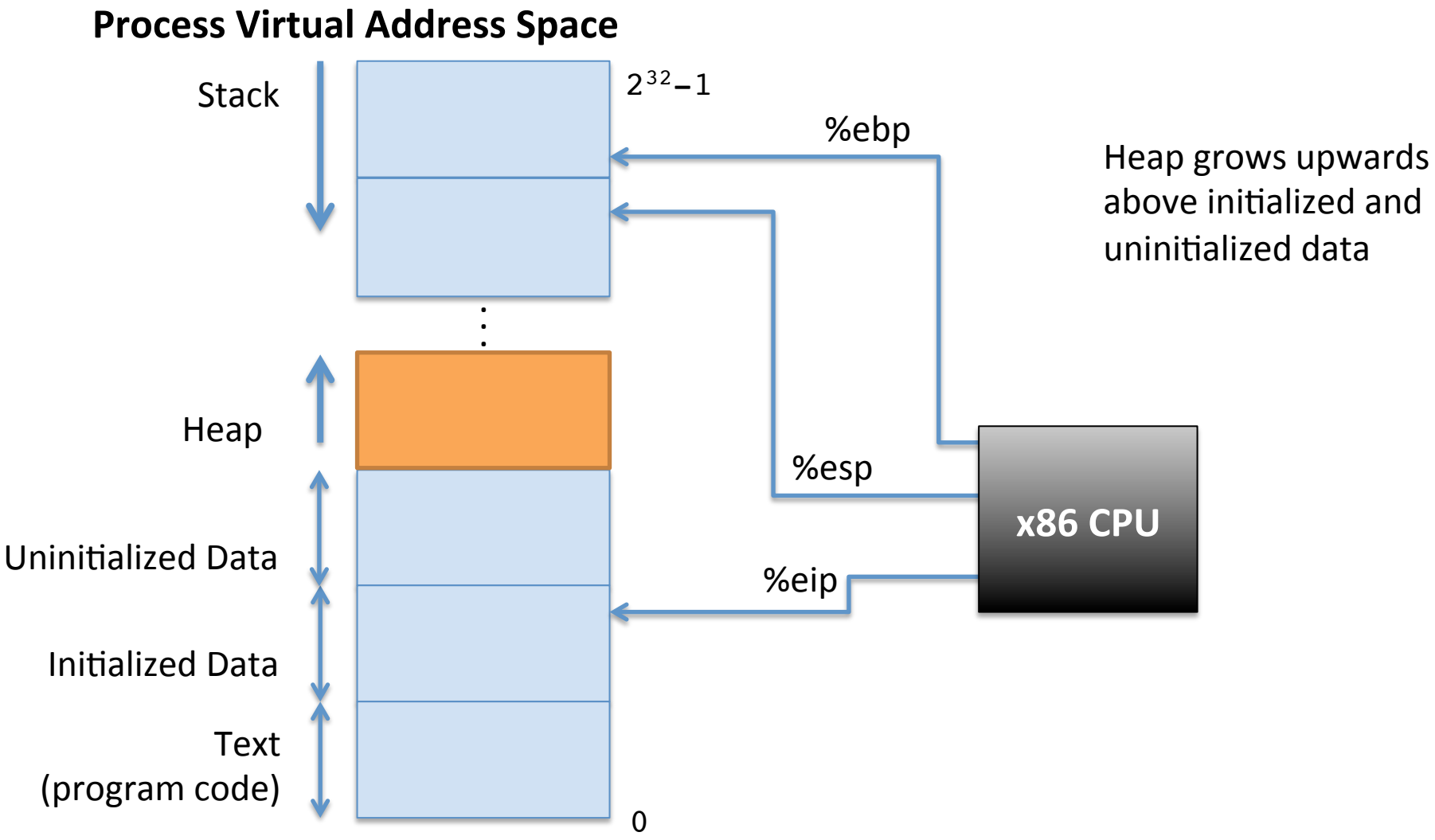

[cvwright@ubuntu tmp]$ gcc -fno-stack-protector -o tracer2 tracer2.c
[cvwright@ubuntu tmp]$
[cvwright@ubuntu tmp]$ ./tracer2
Functions:
main() = 0x804847e
fcn() = 0x8048474
printf() = 0x8048360
malloc() = 0x8048370
Global Variables:
A = 0x804a02c
B = 0x804a028
Heap Variables:
buffer = 0x804b008
array = 0x804b090
Stack Variables:
x = 0xbffff6c8
Code at virtual page # 0x08048
Globals at virtual page # 0x0804a
Heap at virtual page # 0x0804b
Stack at virtual page # 0xbffff

More instrumentation for function calls

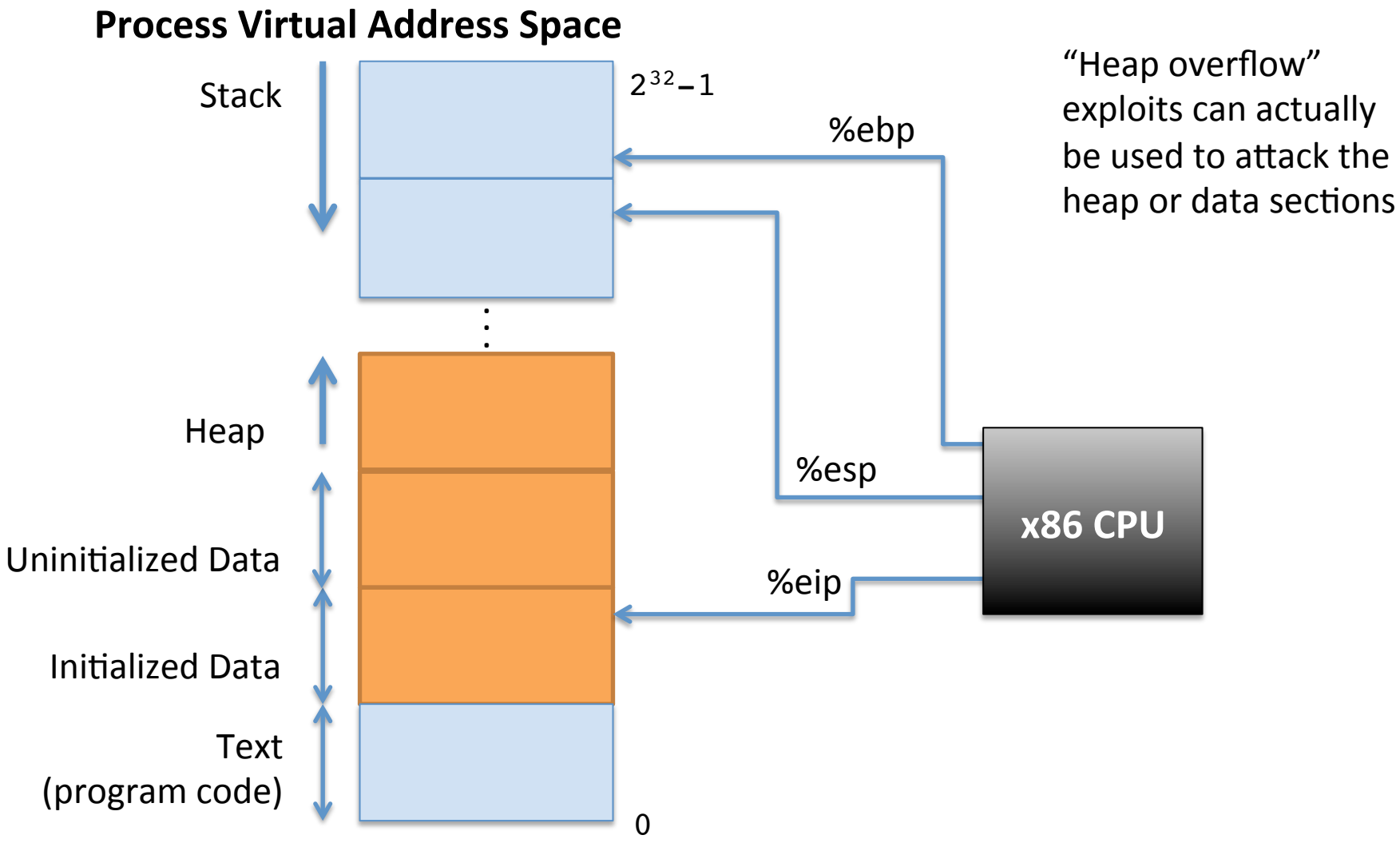
```
int fcn(int arg) {
  int rc;
  char buf[5];
  char *stuff = (char *) malloc(16*sizeof(char));

  printf("depth = %2d   ", arg);
  printf("arg = %10p   ", &arg);
  printf("rc = %10p   ", &rc);
  printf("buf = %10p   ", buf);
  printf("stuff = %10p\n", stuff);

  if(arg < 10)
    rc = fcn(arg+1);
  else
    rc = 0;
  free(stuff);
  return rc;
}
```


Global Variables:
A = 0x804a034
B = 0x804a030
Heap Variables:
buffer = 0x804b008
array = 0x804b090
Stack Variables:
x = 0xbffff6c4
Stack grows "downwards"
Heap grows "upwards"
depth = 0 arg = 0xbffff6b0 rc = 0xbffff698 buf = 0xbffff693 stuff = 0x804b498
depth = 1 arg = 0xbffff680 rc = 0xbffff668 buf = 0xbffff663 stuff = 0x804b4b0
depth = 2 arg = 0xbffff650 rc = 0xbffff638 buf = 0xbffff633 stuff = 0x804b4c8
depth = 3 arg = 0xbffff620 rc = 0xbffff608 buf = 0xbffff603 stuff = 0x804b4e0
depth = 4 arg = 0xbffff5f0 rc = 0xbffff5d8 buf = 0xbffff5d3 stuff = 0x804b4f8
depth = 5 arg = 0xbffff5c0 rc = 0xbffff5a8 buf = 0xbffff5a3 stuff = 0x804b510
depth = 6 arg = 0xbffff590 rc = 0xbffff578 buf = 0xbffff573 stuff = 0x804b528
depth = 7 arg = 0xbffff560 rc = 0xbffff548 buf = 0xbffff543 stuff = 0x804b540
depth = 8 arg = 0xbffff530 rc = 0xbffff518 buf = 0xbffff513 stuff = 0x804b558
depth = 9 arg = 0xbffff500 rc = 0xbffff4e8 buf = 0xbffff4e3 stuff = 0x804b570
depth = 10 arg = 0xbffff4d0 rc = 0xbffff4b8 buf = 0xbffff4b3 stuff = 0x804b588

In-Memory Layout of a Process



In-Memory Layout of a Process



"Heap overflow" exploits can actually be used to attack the heap or data sections

Why is the heap a valuable target?

Interviewer*: Well, can you... blow up the world?*

The Tick*: Egad! I hope not! That's where I keep all my stuff!*

From *The Tick* animated series (1994)

Why is the heap a valuable target?

- That's where our programs keep (most of) their stuff
  - Global variables (really in the data sections)
  - Large, global data structures
  - Any data with dynamic size

Simple Heap-Based Overflows

- Insecure use of the heap allows modification of other heap data
  - Overflow into a string → Change filename
  - Overflow into an int → Change user id
  - Overflow into function pointer → Change program's control flow

```
Global Variables:
               A =  0x804a034
               B =  0x804a030

Heap Variables:
          buffer =  0x804b008
           array =  0x804b090

Stack Variables:
               x = 0xbffff6c4

depth =  0   arg = 0xbffff6b0   rc = 0xbffff698   buf = 0xbffff693   stuff =  0x804b498
depth =  1   arg = 0xbffff680   rc = 0xbffff668   buf = 0xbffff663   stuff =  0x804b4b0
depth =  2   arg = 0xbffff650   rc = 0xbffff638   buf = 0xbffff633   stuff =  0x804b4c8
depth =  3   arg = 0xbffff620   rc = 0xbffff608   buf = 0xbffff603   stuff =  0x804b4e0
depth =  4   arg = 0xbffff5f0   rc = 0xbffff5d8   buf = 0xbffff5d3   stuff =  0x804b4f8
depth =  5   arg = 0xbffff5c0   rc = 0xbffff5a8   buf = 0xbffff5a3   stuff =  0x804b510
depth =  6   arg = 0xbffff590   rc = 0xbffff578   buf = 0xbffff573   stuff =  0x804b528
depth =  7   arg = 0xbffff560   rc = 0xbffff548   buf = 0xbffff543   stuff =  0x804b540
depth =  8   arg = 0xbffff530   rc = 0xbffff518   buf = 0xbffff513   stuff =  0x804b558
depth =  9   arg = 0xbffff500   rc = 0xbffff4e8   buf = 0xbffff4e3   stuff =  0x804b570
depth = 10   arg = 0xbffff4d0   rc = 0xbffff4b8   buf = 0xbffff4b3   stuff =  0x804b588
```

Stack grows "downwards"

Heap grows "upwards"

Why does the heap address increase by 0x18 ?

More instrumentation for function calls

```
int fcn(int arg) {
  int rc;
  char buf[5];
  char *stuff = (char *) malloc(16*sizeof(char));

  printf("depth = %2d    ", arg);
  printf("arg = %10p    ", &arg);
  printf("rc = %10p    ", &rc);
  printf("buf = %10p    ", buf);
  printf("stuff = %10p\n", stuff);

  if(arg < 10)
    rc = fcn(arg+1);
  else
    rc = 0;
  free(stuff);
  return rc;
}
```

After all, we only asked for 16 (0x10) bytes

Example Heap Overflow

```c
#include <stdio.h>
#include <malloc.h>

int main(int argc, char* argv[])
{
  char *username = (char *) malloc(16*sizeof(char));
  char *filename = (char *) malloc(16*sizeof(char));

  sprintf(filename, "%s.txt", argv[1]);
  if(!strcmp(filename, "flag.txt")){
    puts("FLAG ACCESS DENIED");
    return 1;
  }

  printf("Enter username for access to file [%s]\n", filename);
  scanf("%s", username);
```

Attacking malloc() and free()

- How does malloc() work?
  - Chunks
  - Doubly-linked lists
- Abusing free() – the unlink() attack

Malloc

- Malloc – An interface between the low-level POSIX OS interface and high-level C code

- POSIX – brk and sbrk system calls
  - Ask the OS to map more virtual memory at the top of the heap
  - Page-level granularity (e.g. 4KB at a time)
  - Managing already-allocated memory is up to the application itself

Malloc

- Malloc's (and free's) job
  - Keep track of heap memory provided by the OS
  - Ask for more when we run out of space
  - Find the best location to satisfy each request from the application code

Malloc

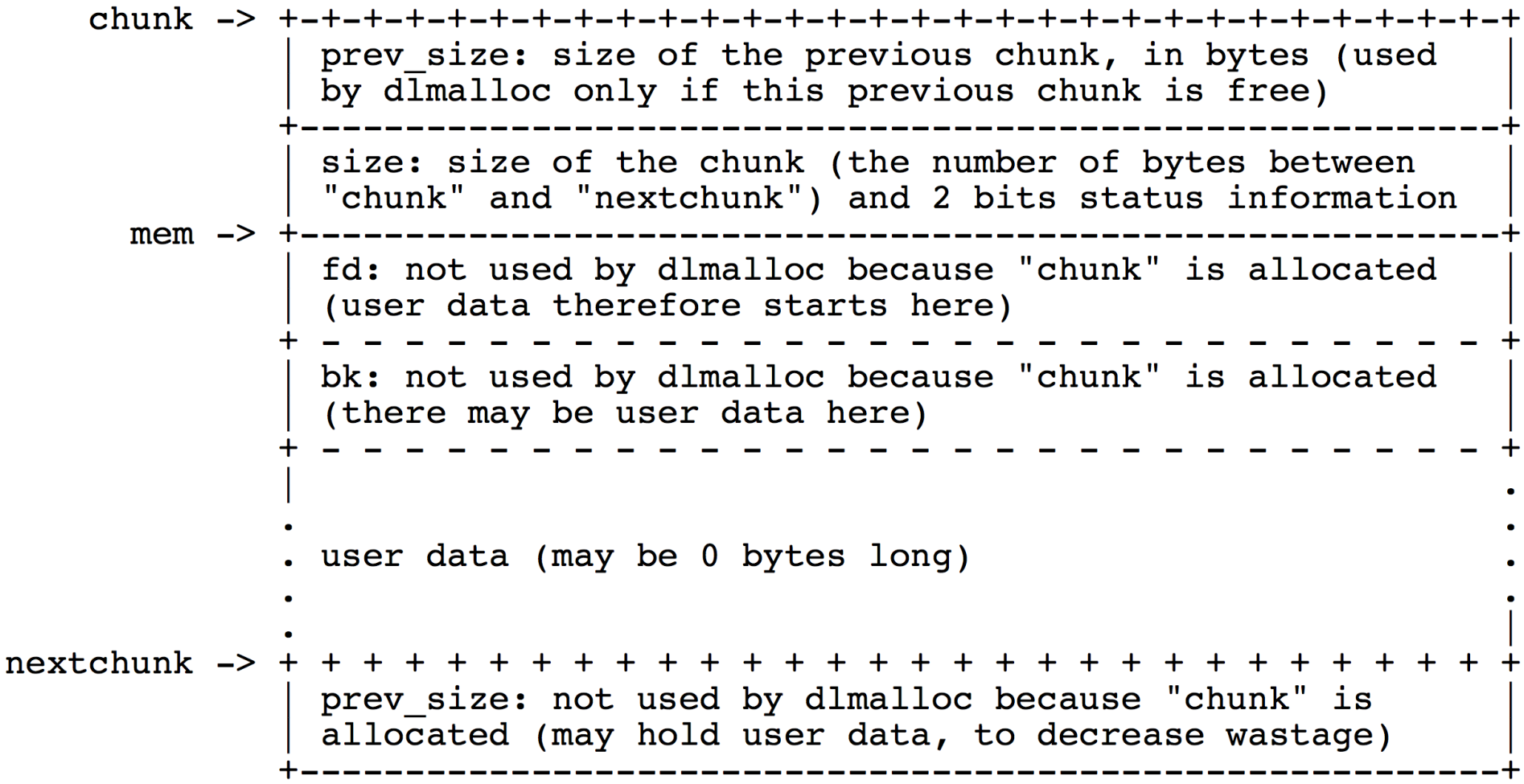
- How does malloc do it?
  - Sort of like a slab allocator from OS class (CS 333)
  - Tracks free regions using linked lists
  - Stores linked list pointers intermixed with data allocated for the application

Malloc Implementation

```
#define INTERNAL_SIZE_T size_t

struct malloc_chunk {
     INTERNAL_SIZE_T prev_size;
     INTERNAL_SIZE_T size;
     struct malloc_chunk * fd;
     struct malloc_chunk * bk;
};
```

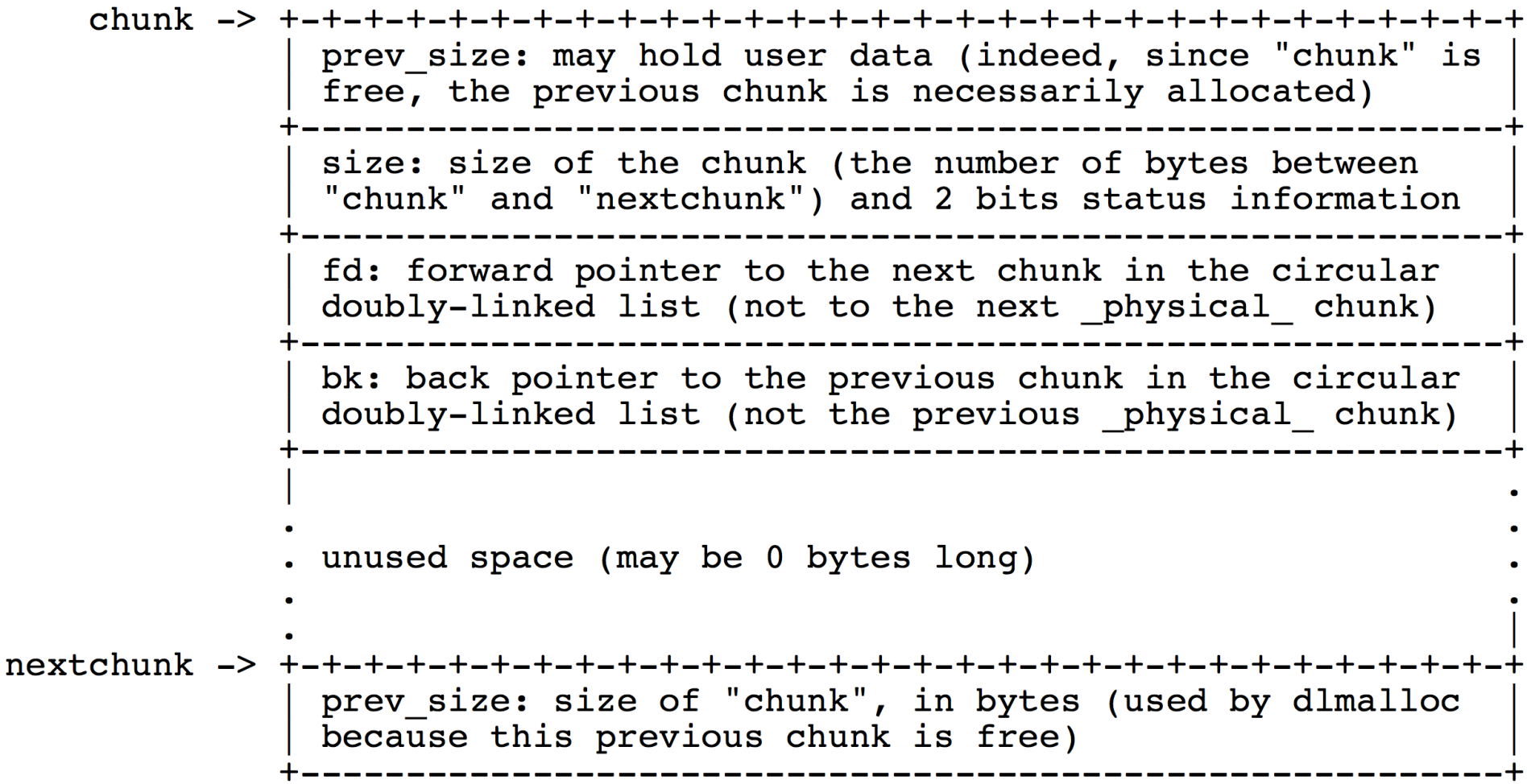
Malloc Implementation

- An allocated chunk looks like this:



Malloc Implementation

- Free chunks are stored in circular doubly-linked lists (described in 3.4.2) and look like this:




Global Variables:
A = 0x804a034
B = 0x804a030
Heap Variables:
buffer = 0x804b008
array = 0x804b090
Stack Variables:
x = 0xbffff6c4
Stack grows "downwards"
Heap grows "upwards"
depth = 0 arg = 0xbffff6b0 rc = 0xbffff698 buf = 0xbffff693 stuff = 0x804b498
depth = 1 arg = 0xbffff680 rc = 0xbffff668 buf = 0xbffff663 stuff = 0x804b4b0
depth = 2 arg = 0xbffff650 rc = 0xbffff638 buf = 0xbffff633 stuff = 0x804b4c8
depth = 3 arg = 0xbffff620 rc = 0xbffff608 buf = 0xbffff603 stuff = 0x804b4e0
depth = 4 arg = 0xbffff5f0 rc = 0xbffff5d8 buf = 0xbffff5d3 stuff = 0x804b4f8
depth = 5 arg = 0xbffff5c0 rc = 0xbffff5a8 buf = 0xbffff5a3 stuff = 0x804b510
depth = 6 arg = 0xbffff590 rc = 0xbffff578 buf = 0xbffff573 stuff = 0x804b528
depth = 7 arg = 0xbffff560 rc = 0xbffff548 buf = 0xbffff543 stuff = 0x804b540
depth = 8 arg = 0xbffff530 rc = 0xbffff518 buf = 0xbffff513 stuff = 0x804b558
depth = 9 arg = 0xbffff500 rc = 0xbffff4e8 buf = 0xbffff4e3 stuff = 0x804b570
depth = 10 arg = 0xbffff4d0 rc = 0xbffff4b8 buf = 0xbffff4b3 stuff = 0x804b588

```
Global Variables:
               A =  0x804a034
               B =  0x804a030

Heap Variables:
          buffer =  0x804b008
           array =  0x804b090

Stack Variables:
               x = 0xbffff6c4

depth =  0   arg = 0xbffff6b0   rc = 0xbffff698   buf = 0xbffff693   stuff =  0x804b498
depth =  1   arg = 0xbffff680   rc = 0xbffff668   buf = 0xbffff663   stuff =  0x804b4b0
depth =  2   arg = 0xbffff650   rc = 0xbffff638   buf = 0xbffff633   stuff =  0x804b4c8
depth =  3   arg = 0xbffff620   rc = 0xbffff608   buf = 0xbffff603   stuff =  0x804b4e0
depth =  4   arg = 0xbffff5f0   rc = 0xbffff5d8   buf = 0xbffff5d3   stuff =  0x804b4f8
depth =  5   arg = 0xbffff5c0   rc = 0xbffff5a8   buf = 0xbffff5a3   stuff =  0x804b510
depth =  6   arg = 0xbffff590   rc = 0xbffff578   buf = 0xbffff573   stuff =  0x804b528
depth =  7   arg = 0xbffff560   rc = 0xbffff548   buf = 0xbffff543   stuff =  0x804b540
depth =  8   arg = 0xbffff530   rc = 0xbffff518   buf = 0xbffff513   stuff =  0x804b558
depth =  9   arg = 0xbffff500   rc = 0xbffff4e8   buf = 0xbffff4e3   stuff =  0x804b570
depth = 10   arg = 0xbffff4d0   rc = 0xbffff4b8   buf = 0xbffff4b3   stuff =  0x804b588
```

Stack grows "downwards"

Heap grows "upwards"

Why does the heap address increase by 0x18 ?

More instrumentation for function calls

```
int fcn(int arg) {
  int rc;
  char buf[5];
  char *stuff = (char *) malloc(16*sizeof(char));

  printf("depth = %2d    ", arg);
  printf("arg = %10p    ", &arg);
  printf("rc = %10p    ", &rc);
  printf("buf = %10p    ", buf);
  printf("stuff = %10p\n", stuff);

  if(arg < 10)
    rc = fcn(arg+1);
  else
    rc = 0;
  free(stuff);
  return rc;
}
```

After all, we only asked for 16 (0x10) bytes

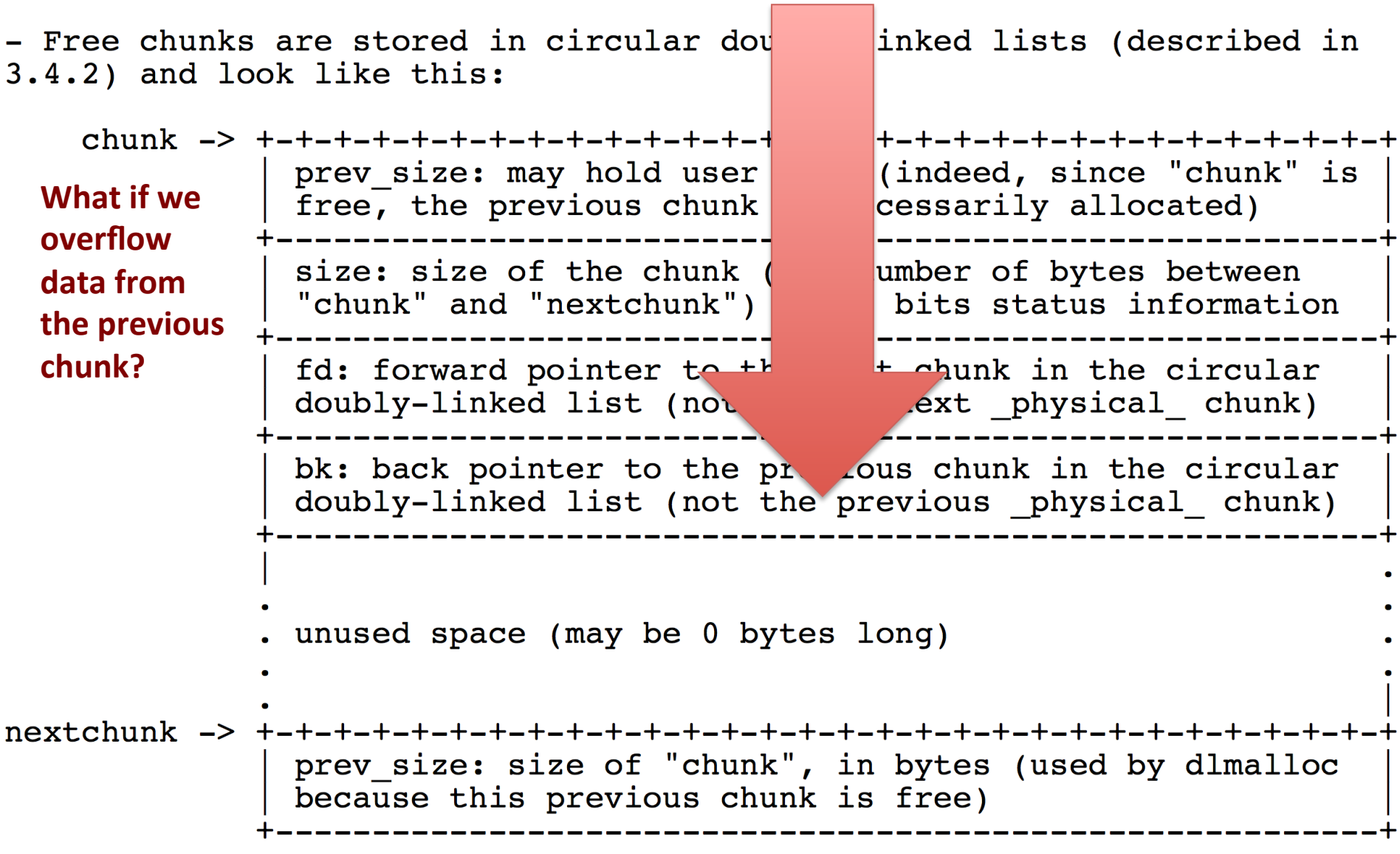
More instrumentation for function calls

```
int fcn(int arg) {
  int rc;
  char buf[5];
  char *stuff = (char *) malloc(16*sizeof(char));

  printf("depth = %2d    ", arg);
  printf("arg = %10p    ", &arg);
  printf("rc = %10p    ", &rc);
  printf("buf = %10p    ", buf);
  printf("stuff = %10p\n", stuff);

  if(arg < 10)
    rc = fcn(arg+1);
  else
    rc = 0;
  free(stuff);
  return rc;
}
```

After all, we only asked for 16 (0x10) bytes

Answer:
Malloc needs at least 8 bytes for its own internal book keeping!

Heap Overflows and Malloc



What if we overflow data from the previous chunk?

The unlink attack

```
#define unlink( P, BK, FD ) {                                  \
    BK = P->bk;                                                \
    FD = P->fd;                                                \
    FD->bk = BK;                                               \
    BK->fd = FD;                                               \
}
```

Basic idea: Overflow the preceding block to overwrite this block's bk and fd fields.

The unlink attack

```
#define unlink( P, BK, FD ) {                                  \
    BK = P->bk;                                                \
    FD = P->fd;                                                \
    FD->bk = BK;                                               \
    BK->fd = FD;                                               \
}
```

Malloc interprets the 4 bytes in fd as an address.

For example, this could be chosen so it's (close to) a function pointer or a saved %eip.

The unlink attack

```
#define unlink( P, BK, FD ) {                                          \
    BK = P->bk;                                                        \
    FD = P->fd;                                                        \
    FD->bk = BK;                                                       \
    BK->fd = FD;                                                       \
}
```

Then unlink writes to memory at a short offset from this address.

Here we cause it to actually overwrite the function pointer or saved eip.

The unlink attack

```
#define unlink( P, BK, FD ) {                                          \
    BK = P->bk;                                                        \
    FD = P->fd;                                                        \
    FD->bk = BK;                                                       \
    BK->fd = FD;                                                       \
}
```

The value that it writes is taken from the 4 bytes of bk.

For our attack, BK could be the address of our shellcode.

The unlink attack

```
#define unlink( P, BK, FD ) {                                          \
    BK = P->bk;                                                        \
    FD = P->fd;                                                        \
    FD->bk = BK;                                                       \
    BK->fd = FD;                                                       \
}
```

One slight complication. 4 bytes of the region at BK get clobbered!

Now we have garbage in our shellcode. What to do?

The unlink attack

```
#define unlink( P, BK, FD ) {                                          \
    BK = P->bk;                                                        \
    FD = P->fd;                                                        \
    FD->bk = BK;                                                       \
    BK->fd = FD;                                                       \
}
```

One slight complication. 4 bytes of the region at BK get clobbered!

Now we have garbage in our shellcode. What to do?

One solution: Shellcode must contain a jmp to "jump over" the 4 bytes that get clobbered.