# Proactive Computer Security

## Write What Where - Heap Buffer Overflows

Morten Shearman Kirkegaard
<moki@fabletech.com>

2014-06-02

FableTech

A *Write-What-Where condition* is a vulnerability which gives the attacker the ability to **write** a **value** (what) of the attackers choosing at an **address** (where) of the attackers choosing.

FableTech

```
struct information {
    char real_name[64];
    ...
};

struct login {
    char login_name[64];
    struct information *info;
};

struct login *l = ...;
strcpy(l->login_name, XXX_login_name);
strcpy(l->info->real_name, XXX_real_name);
```

FableTech

If the attacker can write a value at a location of her choosing, she has won.

# Question

What should we overwrite?

We can overwrite function pointers (e.g. GOT entries) or program specific global variables (e.g. set is_authenticated = true).

Dynamically allocated variables are stored on the heap.

If one heap variable is a buffer, and we can trick the program to write outside this buffer, we might be able to take control of the process.

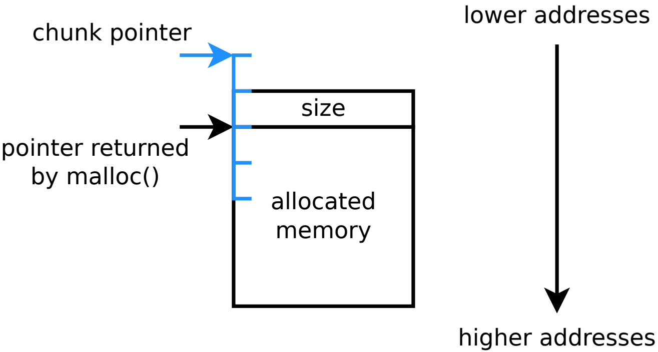
This is a relatively new subject. The first exploit was published in 2000 by Solar Designer.

Since then, priorities of heap managers have gone from "speed" to "resilience to errors, speed."

The heap manager has some internal data structures to keep track of allocations.

These data structures are often stored on the heap itself, next to the allocated memory.

Internally dlmalloc uses a *malloc_chunk* structure for each allocation. It starts 8 bytes (on 32-bit systems) before the allocation returned to the program.

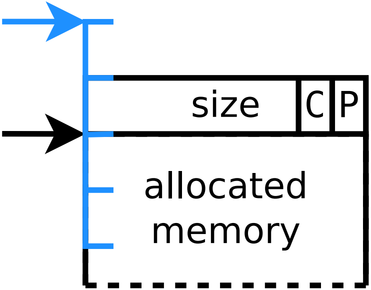


Since all allocations are 8-byte aligned, the lower 3 bits of the size are always 0.

Two of them are used for flags:

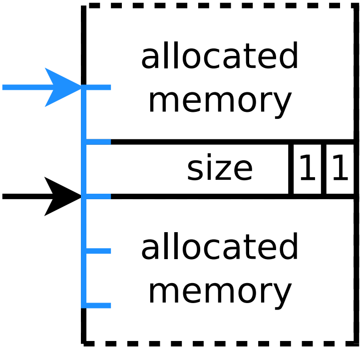
PINUSE_BIT is set if the previous chunk is in use.

CINUSE_BIT is set if the current chunk is in use.



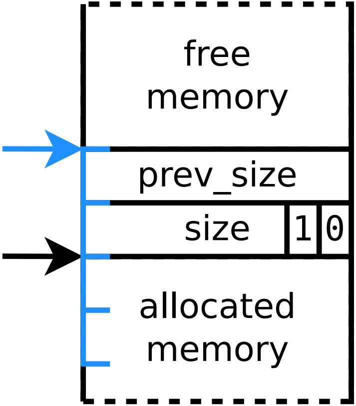

FableTech

If the chunk before the current one is allocated (the PINUSE_BIT is set), the first field of the chunk header belongs to the previous chunk, and must not be accessed.




FableTech

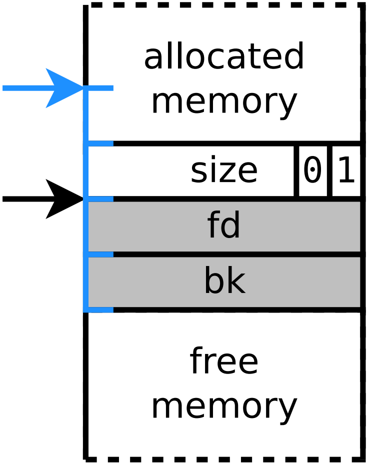
If the chunk before the current one is free (the PINUSE_BIT is clear), the first field of the chunk header contains the size of the previous chunk.



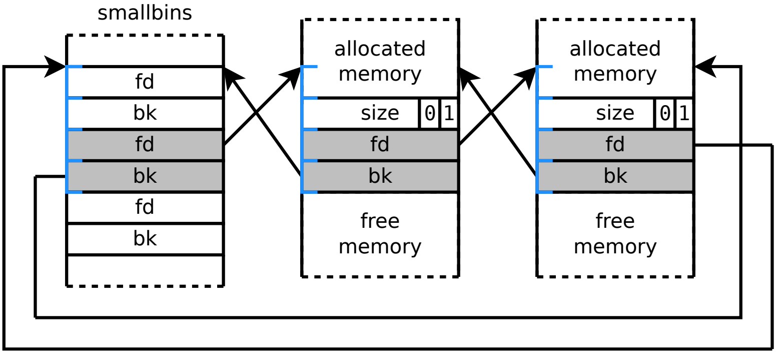

FableTech

If a chunk (of less than 256 bytes) is free, the first 8 bytes are used as pointers for a doubly-linked list of free chunks of that size.

If the free block is 256 bytes or more, it is kept in a tree of free chunks. We will ignore this.



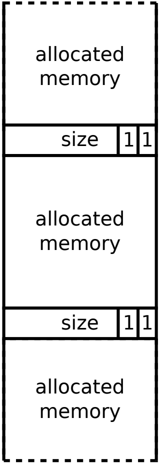

FableTech


smallbins
fd
bk
fd
bk
fd
bk
allocated memory
size
0
1
fd
bk
free memory
allocated memory
size
0
1
fd
bk
free memory

FableTech

If a chunk is freed, and both surrounding chunks are allocated, the freed chunk is simply put in the doubly-linked list of the given `smallbin`.

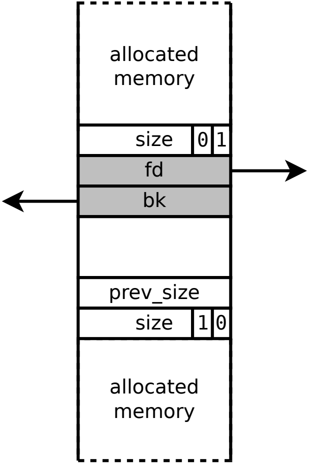
The C bit is cleared in the header, the P bit is cleared in the next header, and the prev_size is set to the size of the free chunk.




FableTech

If a chunk is freed, and both surrounding chunks are allocated, the freed chunk is simply put in the doubly-linked list of the given `smallbin`.

The C bit is cleared in the header, the P bit is cleared in the next header, and the prev_size is set to the size of the free chunk.

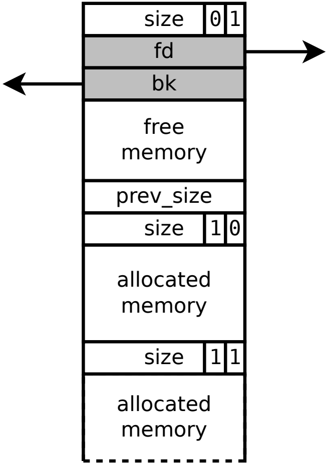



FableTech

If a chunk is freed, and the previous chunk is free, the two are coalesced into one larger chunk.

This is done by unlinking the previous chunk from its free-list, and linking the new — larger — chunk into another free-list.

The P bit is cleared in the next header, and the size and new prev_size are set to the total size of the free chunk.

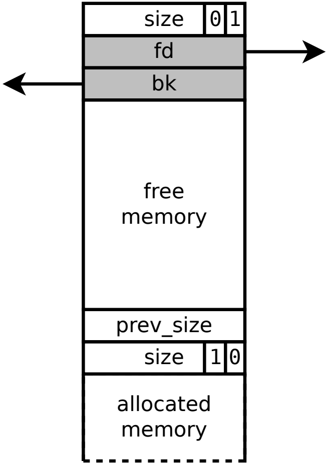



FableTech

If a chunk is freed, and the previous chunk is free, the two are coalesced into one larger chunk.

This is done by unlinking the previous chunk from its free-list, and linking the new — larger — chunk into another free-list.

The P bit is cleared in the next header, and the size and new prev_size are set to the total size of the free chunk.

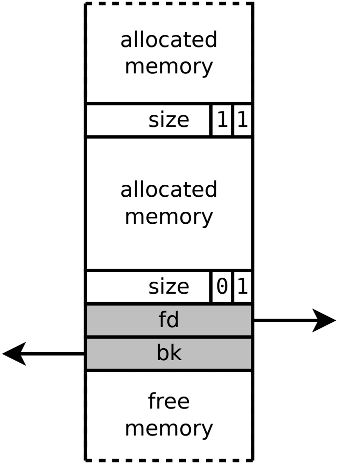



FableTech

If a chunk is freed, and the next chunk is free, the two are coalesced into one larger chunk.

This is done by unlinking the next chunk from its free-list, and linking the new — larger — chunk into another free-list.

The C bit is cleared in the header, and the size and prev_size are set to the total size of the free chunk.

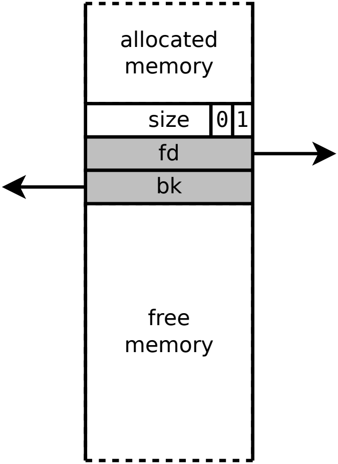



FableTech

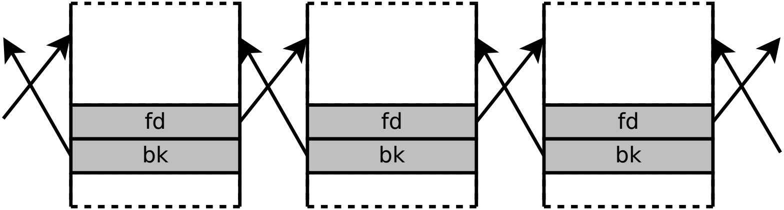
If a chunk is freed, and the next chunk is free, the two are coalesced into one larger chunk.

This is done by unlinking the next chunk from its free-list, and linking the new — larger — chunk into another free-list.

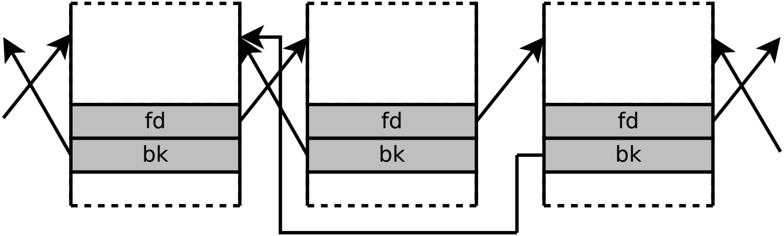
The C bit is cleared in the header, and the size and prev_size are set to the total size of the free chunk.



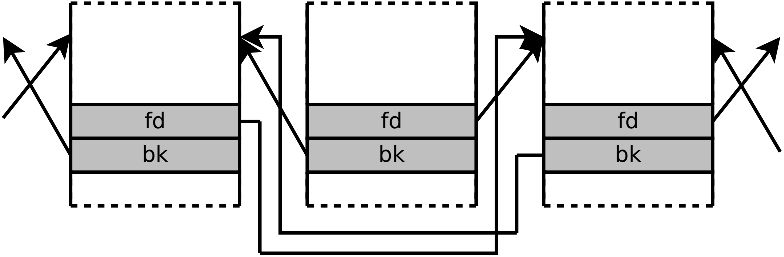

FableTech



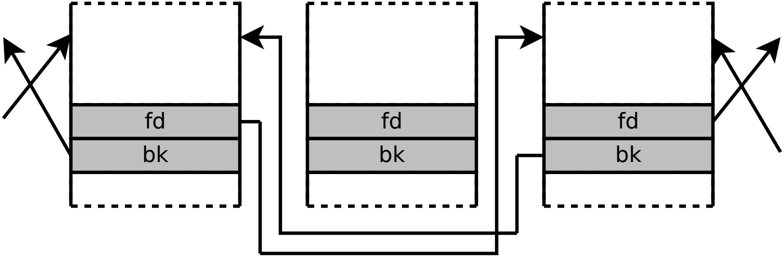

FableTech




FableTech




FableTech


fd
bk
fd
bk
fd
bk

Unlinking is the most exploited part of dlmalloc and most other heap managers.

```
mchunkptr F = P->fd;
mchunkptr B = P->bk;
...
F->bk = B;
B->fd = F;
```

We have two pointers, F and B. At address (F + 12) the heap manager writes B, and at address (B + 8) it writes F. If we control F and B, we control the target process.

```
0x0804c485 <free+991>:  mov    eax,DWORD PTR [ebp-0x14]
0x0804c488 <free+994>:  mov    eax,DWORD PTR [eax+0x8]
0x0804c48b <free+997>:  mov    DWORD PTR [ebp-0x5c],eax
0x0804c48e <free+1000>: mov    eax,DWORD PTR [ebp-0x14]
0x0804c491 <free+1003>: mov    eax,DWORD PTR [eax+0xc]
0x0804c494 <free+1006>: mov    DWORD PTR [ebp-0x60],eax
...
0x804c4cc <free+1062>:  mov    eax,DWORD PTR [ebp-0x5c]
0x804c4cf <free+1065>:  mov    edx,DWORD PTR [ebp-0x60]
0x804c4d2 <free+1068>:  mov    DWORD PTR [eax+0xc],edx
0x804c4d5 <free+1071>:  mov    eax,DWORD PTR [ebp-0x60]
0x804c4d8 <free+1074>:  mov    edx,DWORD PTR [ebp-0x5c]
0x804c4db <free+1077>:  mov    DWORD PTR [eax+0x8],edx
```

We have a function pointer at `0x11111111`
and our shellcode at `0x22222222`.

If we force `fd` to `0x11111105` and `bk` to `0x22222222` the unlink will do the following:

Set `*(fd + 0xC)` to `bk`;
`*(0x11111105 + 0xC)` to `0x22222222`;
`*0x11111111` to `0x22222222`.

Set `*(bk + 0x8)` to `fd`;
`*(0x22222222 + 0x8)` to `0x11111111`;
`*0x2222222A` to `0x11111111`.

FableTech

We now have what's known as *an arbitrary 4-byte memory overwrite*.

The first overwrite will change the function pointer, so it points at our shellcode. This is good.

The second overwrite will change part of the shellcode to point to the function pointer. This is annoying.

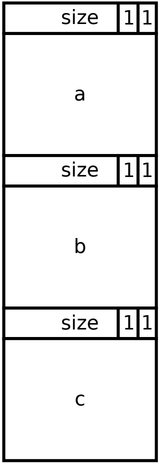
FableTech

If the shellcode is structured like this, the overwrite will not disturb the flow:

```
22222222  90        nop
22222223  90        nop
22222224  90        nop
22222225  90        nop
22222226  90        nop
22222227  90        nop
22222228  EB04      jmp short 0x2222222E
2222222A  41        inc ecx
2222222B  41        inc ecx
2222222C  41        inc ecx
2222222D  41        inc ecx
2222222E  90        nop      ; code here
```

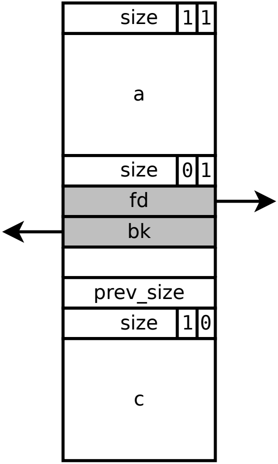
FableTech

```
    char *a, *b, *c;

    a = malloc(20);
    b = malloc(20);
    c = malloc(20);
->
    free(b);

    strcpy(b, "AAAABBBB");

    free(a);
```



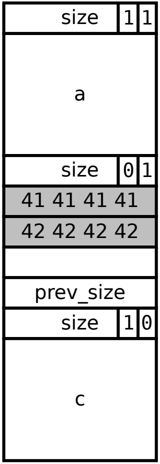

FableTech

```
    char *a, *b, *c;

    a = malloc(20);
    b = malloc(20);
    c = malloc(20);

    free(b);
->
    strcpy(b, "AAAABBBB");

    free(a);
```




FableTech

```
   char *a, *b, *c;

   a = malloc(20);
   b = malloc(20);
   c = malloc(20);

   free(b);

   strcpy(b, "AAAABBBB");
->
   free(a);
```

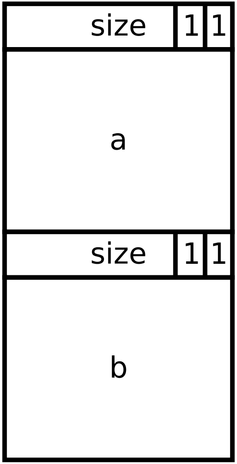



FableTech

```
Program received signal SIGSEGV, Segmentation fault.
0x0804c465 in free ()
(gdb) x/i $eip
0x804c465 <free+1135>:  mov    DWORD PTR [eax+0xc],edx
(gdb) i r eax edx
eax            0x41414141       1094795585
edx            0x42424242       1111638594
```

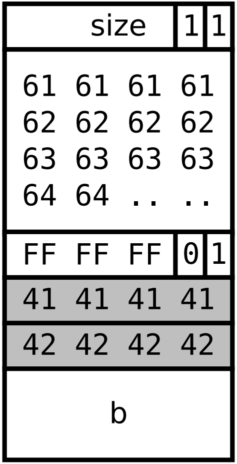
FableTech

```
    char *a, *b;

    a = malloc(20);
    b = malloc(20);
->
    strcpy(a, "aaaabbbbccccdddd"
     "eeee\xFF\xFF\xFF\xF1"
     "AAAABBBB");

    free(a);
```




FableTech

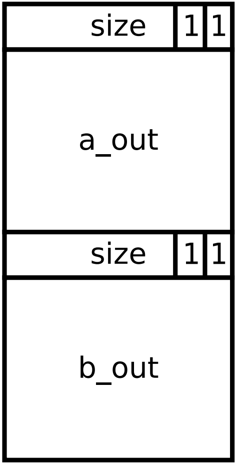
```
    char *a, *b;

    a = malloc(20);
    b = malloc(20);

    strcpy(a, "aaaabbbbccccdddd"
     "eeee\xFF\xFF\xFF\xF1"
     "AAAABBBB");
->
    free(a);
```




FableTech

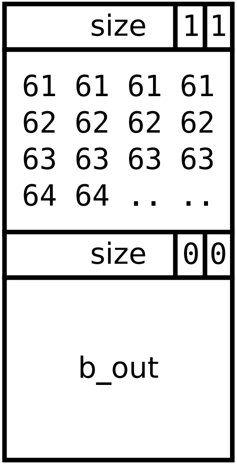
```
Program received signal SIGSEGV, Segmentation fault.
0x0804c465 in free ()
(gdb) x/i $eip
0x804c465 <free+1135>:  mov    DWORD PTR [eax+0xc],edx
(gdb) i r eax edx
eax            0x41414141       1094795585
edx            0x42424242       1111638594
```

```
char *a_in, *a_out;
char *b_in, *b_out;

a_in = "aaaabbbbccccddddeeee";
b_in = "AAAABBBB";

a_out = malloc(strlen(a_in));
b_out = malloc(strlen(b_in));
->

strcpy(a_out, a_in);

strcpy(b_out, b_in);

free(a_out);
```



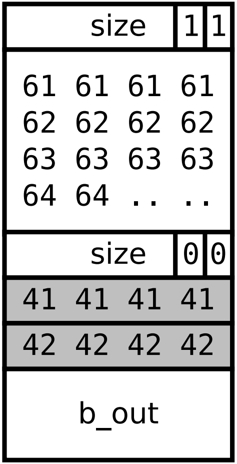

FableTech

```
char *a_in, *a_out;
char *b_in, *b_out;

a_in = "aaaabbbbccccddddeeee";
b_in = "AAAABBBB";

a_out = malloc(strlen(a_in));
b_out = malloc(strlen(b_in));

strcpy(a_out, a_in);
->
strcpy(b_out, b_in);

free(a_out);
```




FableTech

```
char *a_in, *a_out;
char *b_in, *b_out;

a_in = "aaaabbbbccccddddeeee";
b_in = "AAAABBBB";

a_out = malloc(strlen(a_in));
b_out = malloc(strlen(b_in));

strcpy(a_out, a_in);

strcpy(b_out, b_in);
->
free(a_out);
```

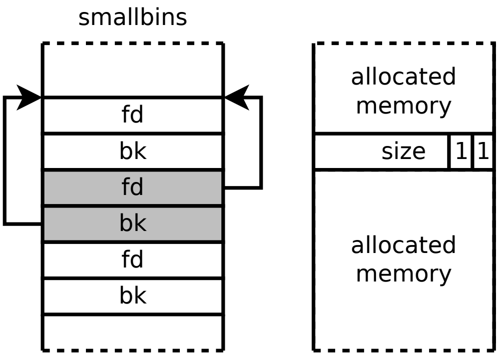



FableTech

```
Program received signal SIGSEGV, Segmentation fault.
0x0804c465 in free ()
(gdb) x/i $eip
0x804c465 <free+1135>:  mov    DWORD PTR [eax+0xc],edx
(gdb) i r eax edx
eax            0x41414141       1094795585
edx            0x42424242       1111638594
```

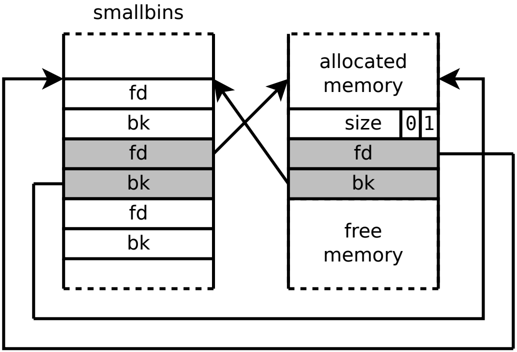
FableTech

```
   p = malloc(20);
->
   free(p);

   free(p);

   q = malloc(20);

   strcpy(q,
    "AAAABBBB");
```



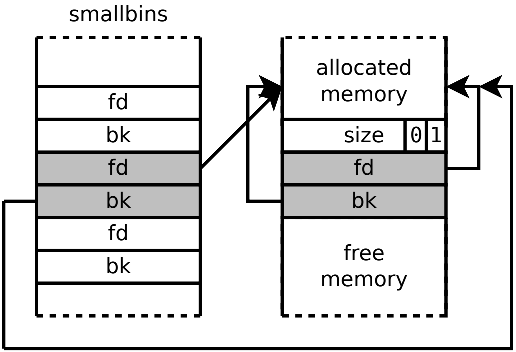

FableTech

```
    p = malloc(20);

    free(p);
->
    free(p);

    q = malloc(20);

    strcpy(q,
     "AAAABBBB");
```



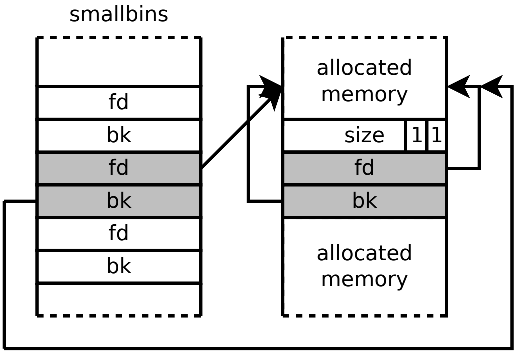

FableTech

```
p = malloc(20);

free(p);

free(p);

->

q = malloc(20);

strcpy(q,
 "AAAABBBB");
```



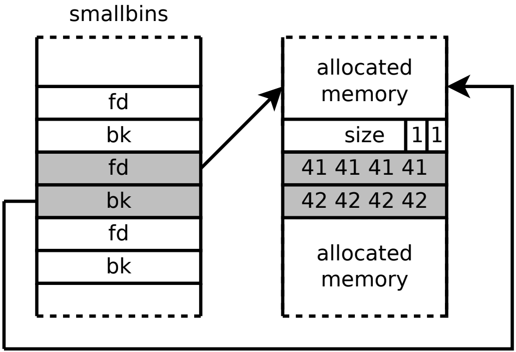

FableTech

```
  p = malloc(20);

  free(p);

  free(p);

  q = malloc(20);
->
  strcpy(q,
   "AAAABBBB");
```




FableTech

```
p = malloc(20);

free(p);

free(p);

q = malloc(20);

strcpy(q,
 "AAAABBBB");
->
```




FableTech

# Coffee Break

FableTech

If you don't know exactly which address your shellcode will be placed at, you can make sure that it does not matter.

Make enough code, which does nothing. Place it in front of the shellcode.

When EIP is set to any address within this area, it will "slide" down to your shellcode.

FableTech

A *NOP Slide* a/k/a *NOP Sled* a/k/a *NOP* ... can be as simple as:

```
90               nop
90               nop
90               nop
90               nop
XX XX XX XX XX   SHELLCODE
```

If it must allow overwrites (for unlink exploits), a series of jumps is useful:

```
EB 7E              jmp short +126
EB 7E              jmp short +126
EB 7E              jmp short +126
...
90                 nop
90                 nop
90                 nop
...
XX XX XX XX XX     SHELLCODE
```

FableTech

If you have to guess a heap address of one buffer, and you must hit somewhere in a 1 kilobytes NOP Slide, your odds are small; About 1 in 4000000 on a 32 bit host.

If you have to guess a heap address of one buffer, and you must hit somewhere in a 1 kilobytes NOP Slide, your odds are small; About 1 in 4000000 on a 32 bit host.

If you have to guess the address of one of 1000 NOP Slides of each 1 megabyte, the odds are a lot better; About 1 in 4.

If the target process contains a scripting engine, like JavaScript or ActionScript, it can be used to do Heap Spraying.

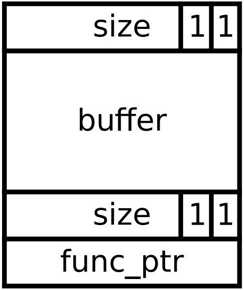
```
var spraySlide = unescape("%u9090%u9090");
while (spraySlide.length*2 < spraySlideSize) {
    spraySlide += spraySlide;
}

var memory = new Array();
for (i=0; i<heapBlocks; i++) {
    memory[i] = spraySlide + payload;
}
```

FableTech

NOP Slides and Heap Spraying are signs of unreliability.

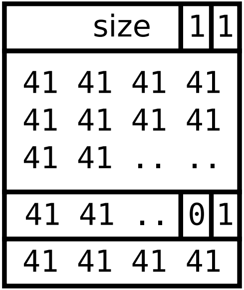
Don't use them in the assignment.

```
    char *buffer;
    void (**p_func_ptr)(void);

    buffer = malloc(20);

    p_func_ptr = malloc(4);
    *p_func_ptr = some_func;
->
    strcpy(buffer,
     "AAAAAAAA...");

    (*p_func_ptr)();
```




FableTech

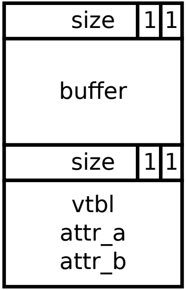
```
    char *buffer;
    void (**p_func_ptr)(void);

    buffer = malloc(20);

    p_func_ptr = malloc(4);
    *p_func_ptr = some_func;

    strcpy(buffer,
     "AAAAAAAA...");
->
    (*p_func_ptr)();
```




FableTech

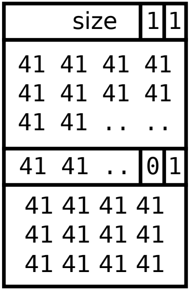
```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

```
class Demo {
public:
 virtual void method_a(void);
 int attr_a;
 int attr_b;
};

char *buffer = new char[16];
Demo *demo = new Demo;
->
strcpy(buffer,
 "AAAAAAAA...");

demo->method_a();
```



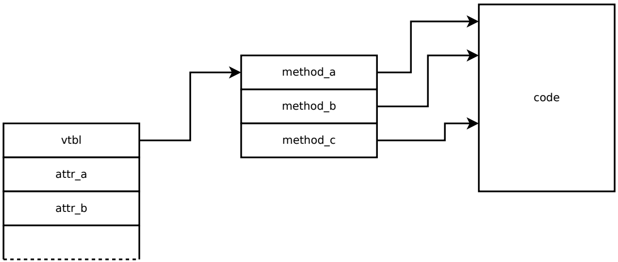

FableTech

```
class Demo {
public:
 virtual void method_a(void);
 int attr_a;
 int attr_b;
};

char *buffer = new char[16];
Demo *demo = new Demo;

strcpy(buffer,
 "AAAAAAAA...");
->
demo->method_a();
```

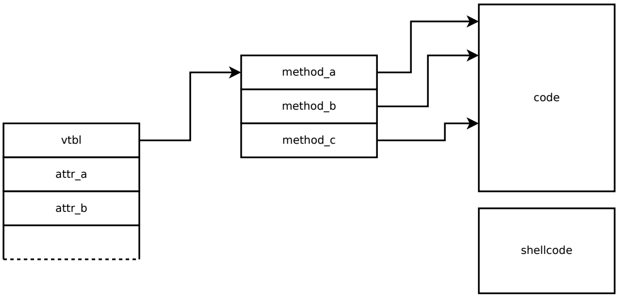


```
Program received signal SIGSEGV, Segmentation fault.
0x080486ce in main ()
(gdb) x/4i $eip
0x80486ce <main+202>:   mov    edx,DWORD PTR [eax]
0x80486d0 <main+204>:   mov    eax,DWORD PTR [esp+0x18]
0x80486d4 <main+208>:   mov    DWORD PTR [esp],eax
0x80486d7 <main+211>:   call   edx
(gdb) i r eax
eax            0x41414141       1094795585
```

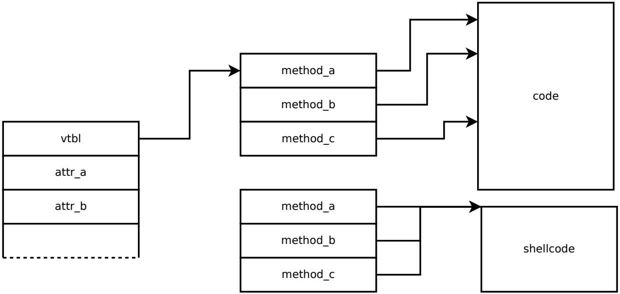
FableTech



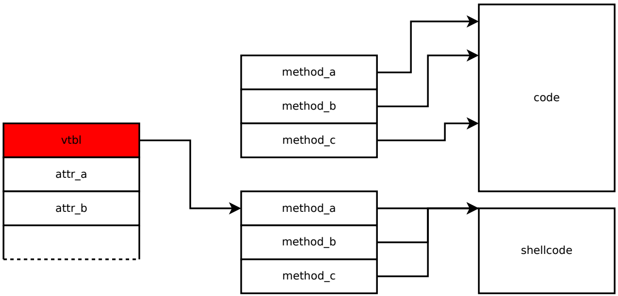

FableTech




FableTech


vtbl
attr_a
attr_b
method_a
method_b
method_c
code
method_a
method_b
method_c
shellcode

FableTech


vtbl
attr_a
attr_b
method_a
method_b
method_c
method_a
method_b
method_c
code
shellcode

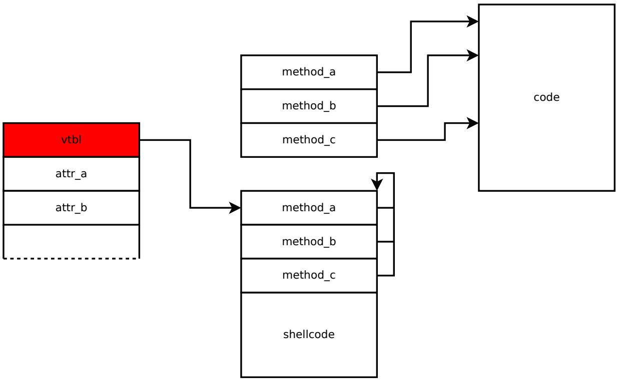
FableTech

The address 0x0C0C0C0C is often used for spraying fake virtual method tables. You only need to spray around 200 MB, the alignment doesn't matter, and the address is valid code — almost NOPs.

```
0C0C0C0C  0C0C        or al,0xc
0C0C0C0E  0C0C        or al,0xc
0C0C0C10  0C0C        or al,0xc
0C0C0C12  0C0C        or al,0xc
```

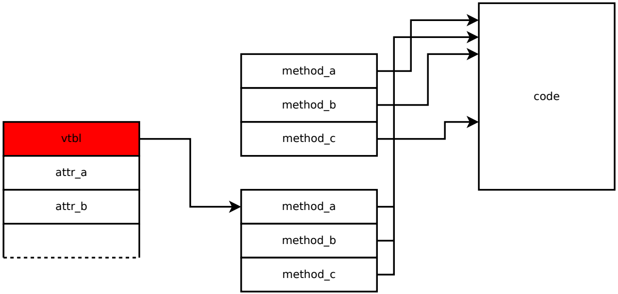
FableTech

NOP Sled and virtual method table in one.




FableTech

With non-executable heap, the method pointers must point to the code segment, making exploitation more challenging.




FableTech

In an exploit for a stack buffer overflow, the attacker can often set up a number of fake stack frames, chaining several *ROP gadgets*. This is often necessary because of ASLR.

In general, a heap buffer overflow "only" gives the attacker EIP control. It is not possible to chain ROP gadgets directly. The very first gadget *must* set up everything needed for chaining.

FableTech

In this example, ESP points to the original stack, and EAX points to a heap buffer the attacker controls.

By copying EAX to ESP, the heap becomes the new stack, allowing the attacker to chain multiple ROP gadgets.

```
EIP ->   mov esp, eax
         ret
```

| | |
|---|---|
| | ... |
| | 7F 88 04 08 |
| | F8 CD FF FF |
| | 00 00 00 00 |
| ESP -> | F4 4F FB F7 |

| | |
|---|---|
| | 29 AD 04 08 |
| | 1E B1 04 08 |
| | 41 41 41 41 |
| EAX -> | C1 BB 04 08 |

**FableTech**

In this example, `ESP` points to the original stack, and `EAX` points to a heap buffer the attacker controls.

By copying `EAX` to `ESP`, the heap becomes the new stack, allowing the attacker to chain multiple ROP gadgets.

| | |
|---|---|
| | mov esp, eax |
| EIP –> | ret |

| | |
|---|---|
| | ... |
| | 7F 88 04 08 |
| | F8 CD FF FF |
| | 00 00 00 00 |
| | F4 4F FB F7 |
| | |
| | 29 AD 04 08 |
| | 1E B1 04 08 |
| | 41 41 41 41 |
| ESP EAX –> | C1 BB 04 08 |

FableTech

Another instruction – `XCHG` – is very useful for making a heap buffer the new stack.

```
EIP ->    xchg esp, eax
          ret
```

| | |
|---|---|
| | ... |
| | 7F 88 04 08 |
| | F8 CD FF FF |
| | 00 00 00 00 |
| ESP -> | F4 4F FB F7 |

| | |
|---|---|
| | 29 AD 04 08 |
| | 1E B1 04 08 |
| | 41 41 41 41 |
| EAX -> | C1 BB 04 08 |

FableTech

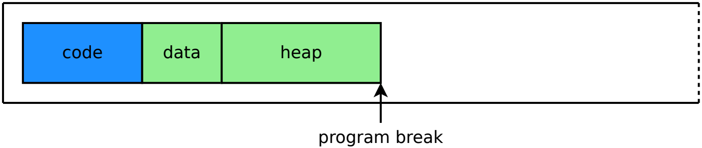
Another instruction – `XCHG` – is very useful for making a heap buffer the new stack.

```
         xchg esp, eax
EIP ->   ret
```

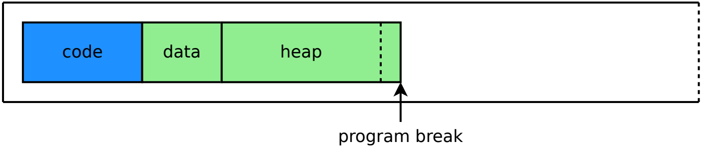
```
          |     ...     |
          | 7F 88 04 08 |
          | F8 CD FF FF |
          | 00 00 00 00 |
EAX ->    | F4 4F FB F7 |

          | 29 AD 04 08 |
          | 1E B1 04 08 |
          | 41 41 41 41 |
ESP ->    | C1 BB 04 08 |
```

FableTech

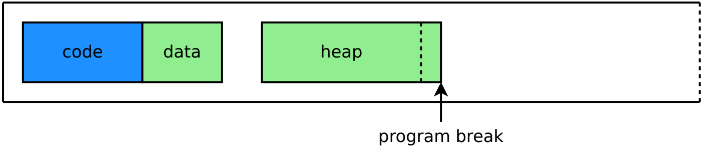
Heap space has traditionally been allocated by moving the *program break*, using the BRK system call.



Heap space has traditionally been allocated by moving the *program break*, using the BRK system call.



With ASLR this gives some entropy, but not much. Only the base address of the heap is randomized.

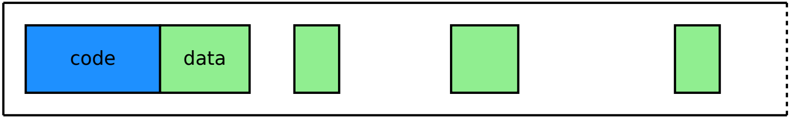



FableTech

OpenBSD's heap manager uses MMAP, rather than BRK, to allocate heap space.

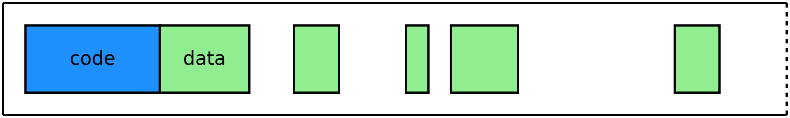
This ensures that addresses are randomized, because of ASLR. It also prevents some overflow, use-after-free, and double-free bugs from being exploitable.

By using the MMAP system call to map new heap pages, a lot more entropy can be added to the heap layout. The price is performance.




FableTech

By using the MMAP system call to map new heap pages, a lot more entropy can be added to the heap layout. The price is performance.




FableTech

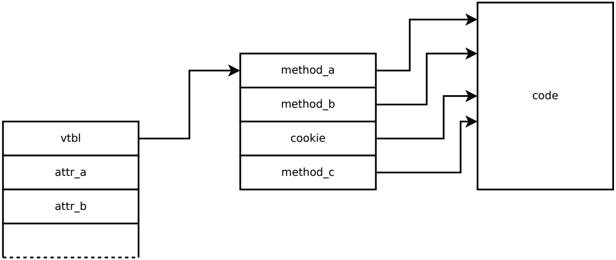
Today all heap managers have consistency checks. In dlmalloc it looks like this:

```
static void do_check_free_chunk(mstate m, mchunkptr p) {
...
assert(!is_inuse(p));
assert(!next_pinuse(p));
...
assert(next->prev_foot == sz);
assert(pinuse(p));
assert(next == m->top || is_inuse(next));
assert(p->fd->bk == p);
assert(p->bk->fd == p);
```

FableTech

Cisco IOS has a special process ("Check heaps") which verifies heap consistency.

According to Cisco's documentation it *[c]hecks the memory every minute. It forces a reload if it finds processor corruption*.


vtbl
attr_a
attr_b
method_a
method_b
cookie
method_c
code

The Windows heap manager has an 8-bit cookie in each chunk header, and will terminate the process, if a cookie is invalid.

FableTech

PHP has its own malloc() wrapper, which does not have *safe unlinking*. This disables the mitigations of the default heap manager — not a smart move.

The *Hardened PHP Project*'s *Suhosin* patch adds this, as well as cookies to the header and footer of each heap chunk.

Remember to fill in the feedback form.

# That's it. Have fun!

FableTech