Lecture 7 : Heap Overflow

Alexandre Bartel

2019

Previously...

Previously... in Lecture 1 (Introduction)

- Software Development Life-cycle
- Vulnerability Life-cycle
- Vulnerability Disclosure

Previously... in Lecture 2 (Buffer Overflow)

- A buffer on the stack
- Return address on the stack
- Overwrite return address
- Jump to shellcode on the stack

Previously... in Lecture 3 (ROP)

- NX bit (stack non-executable)
- Gadgets in already loaded code
- Chain gadgets (addresses of gadgets and data on the stack)
- Only data on the stack

Previously... in Lecture 4 (ASLR)

- Randomize code segment at program start
- Breaks gadget chains
- Bypass with information leak (e.g, vulnerability)

Previously... in Lecture 6 (CFI)

- Mecanism to allow only "intended" paths
- Binary instrumentation to add IDs
- Indirect jumps, call, returns check if ID of "destination" is correct
- Pure software implementation have 20% overhead

Heap Overflow [1] [2]

[1] Designer, Solar. "JPEG COM marker processing vulnerability in Netscape browsers.", 2000
[2] Anonymous, "Once upon a free()", Phrack Magazine 57(9), 2001

Heap Overflow

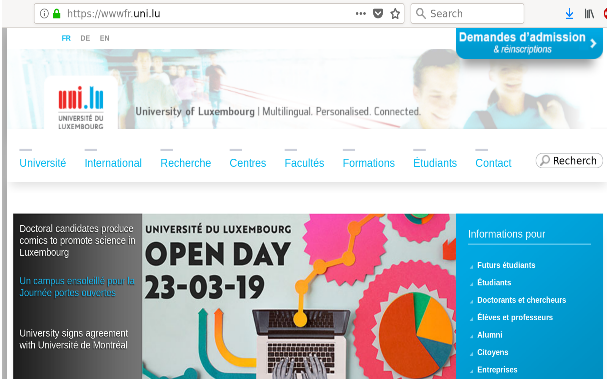
- We have seen buffer overflow on the stack
- In the case of heap overflow the buffer is on the **heap**

Why the Heap?

- Variables can be **global** (static): present in memory for the entire life of the process
- Variables can be **local**: present in memory from the moment the function is entered to the moment the function returns
- It is often the case that variables have *a lifecycle over multiple function but not the whole program life*
- Furthermore, the allocation size might *not be constant* (so, not known at compile time)

Why the Heap? Ex1: units in RTS

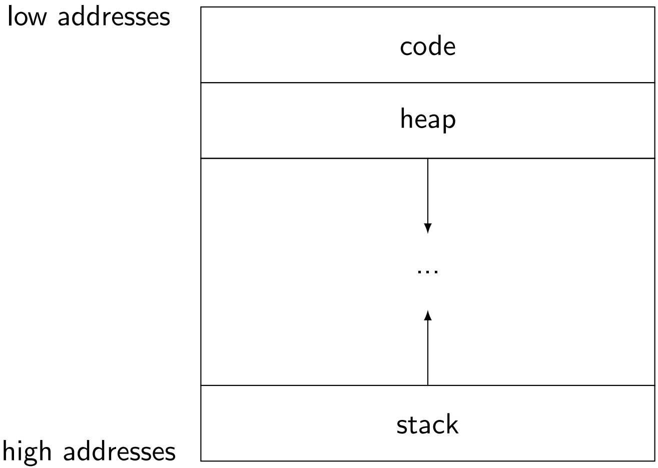
Why the Heap? Ex2: images in web browsers



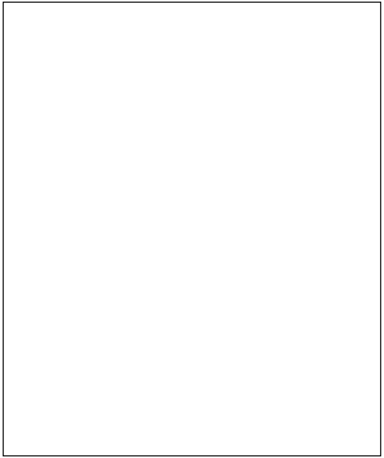
What is the Heap?

- A flexible storage zone in memory
- Memory blocks from this storage zone can be allocated and freed

What is the Heap? (cont)

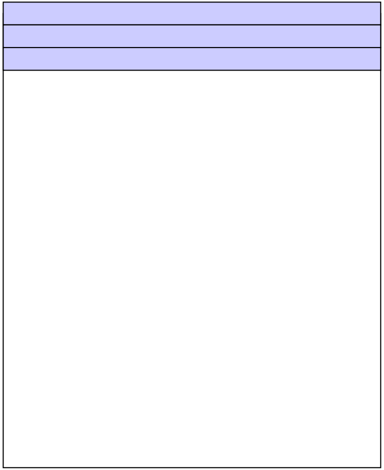


What is the Heap?



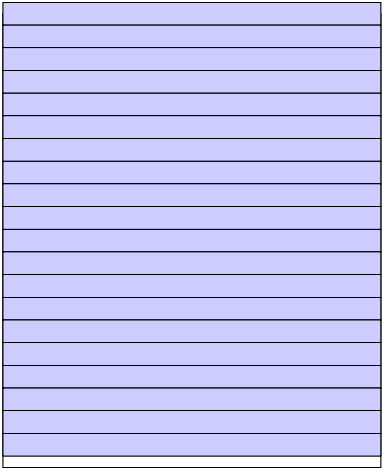
- Empty heap
- Fixed size of n bytes
- How chunks are allocated (where?, what additinal information?, etc) is handled by the heap management implementation

What is the Heap?



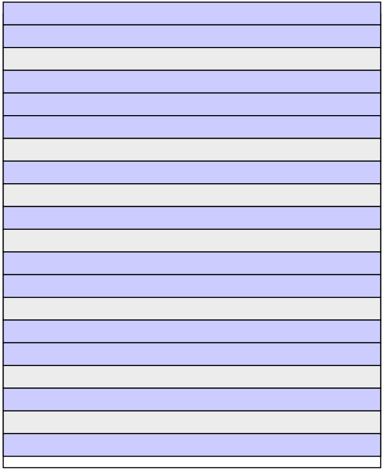
- Allocated 3 chunks
- To simplify, all chunks have the same size

Heap Full



- Problem 1: the heap is full
- Solution: increase the total heap space

Heap Fragmentation



- Problem 2: the heap is fragmented
- Allocating a big chunk on the heap would require to update the heap size **even if the total space available is big enough**
- Because the available space is scattered all around the heap, the allocation cannot happen
- Solution: link free chunks between them and join them when possible
- It keeps the number of reusable blocks low and their size as big as possible

Required Characteristics for a Heap Implementation

- Stable
- Performant
- Avoid fragmentation
- Low overhead (for metadata on blocks, etc.)

List of Heap Implementations

Algorithm	Operating System
BSD kingsley	4.4BSD, AIX (compatibility), Ultrix
BSD phk	BSDI, FreeBSD, OpenBSD
GNU Lib C (Doug Lea)	Hurd, Linux ←
System V AT&T	Solaris, IRIX
Yorktown	AIX (default)
RtlHeap*	Microsoft Windows *

Disclaimer

Disclaimer

We will not look at GNU C library's implementation of the heap management in detail. Only some features necessary to understand heap overflow attack will be explained.

What is the Heap? (cont)

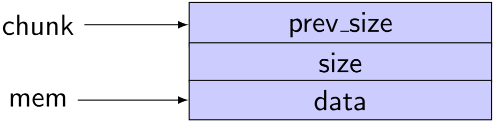
```c
#include <stdlib.h>
#include <stdio.h>
#include <inttypes.h>

int main(int argc, char** argv) {
  uint8_t * byte_array = malloc( 100 );
  // [...]
  printf("0x%016" PRIXPTR " \n", byte_array);
  free(byte_array);
  return 0;
}
```

- In C [1], malloc allocates *n* bytes on the heap
- malloc returns the address of the allocated memory block
- free takes as parameter p (a pointer to a memory block allocated by malloc) and frees this memory block

[1] In Java and other languages, memory on the heap is allocated at every object creation (new Object()) and is not freed explicitly in the code but implicitly by a garbage collector when there is no more reference to the object.

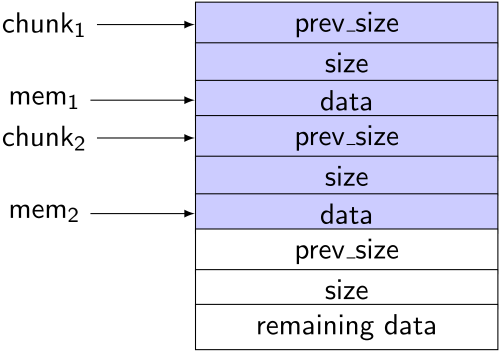
Chunk



- GNU C library keep information about memory blocs in **chunks**
- mem is the pointer returned by malloc
- size is the size of the data*

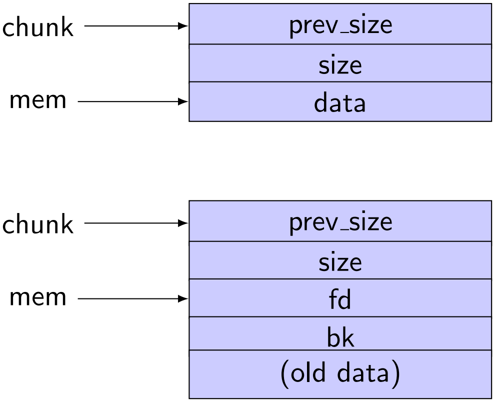
- $*$ the lowest three bits are not used for the size. The lowest bit (PREV_INUSE) indicates if the previous block is used or not.
- prev_size contains the length of the chunk before if it has been freed. Otherwise, it is part of the last bytes of the data section of the previous chunk.

Chunks



- Two chunks in the example
- There is always the "top" chunk to represent the remaining space of the heap
- The chunk "before" the first one does not exist, but is considered to be allocated. Thus, the first chunk's PREV_INUSE is always set to true.

Algorithm: free(chunk)



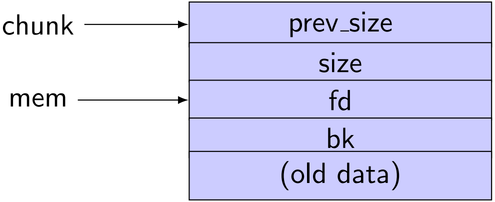
- If a chunk is no longer needed it is marked an unallocated. When this happens, specific pointer values *fd* (forward) and *bk* (backward) are added in the chunk's data section.
- These pointers point to a double linked list of unconsolidated blocks of free memory.
- At every free operation, this list is checked to potentially merge unconsolidated blocks

Algorithm: free(chunk)

- If previous chunk is "free": unlink previous chunk, add previous chunk size to current chunk size, change chunk pointer to point to previous chunk
- If next chunk is "free": unlink next chunk, add next chunk size to current chunk size.

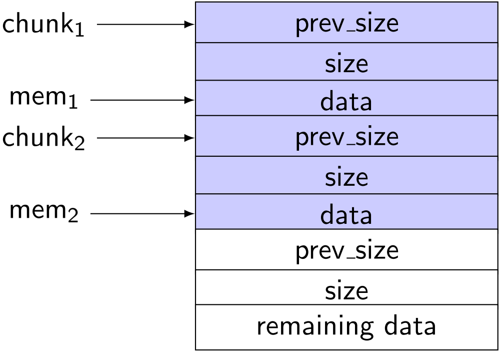
Algorithm: free(chunk)

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



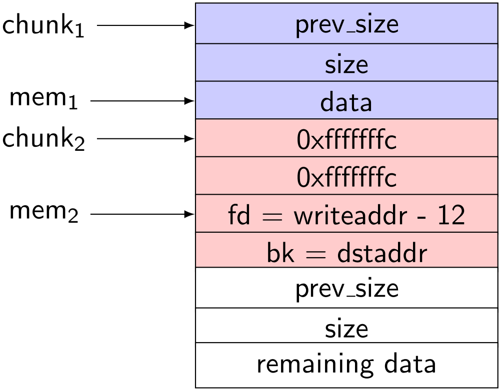
- When consolidating the current block forward, unlink is called with a pointer to a free chunk, P, and two temporary variables, BK and FD
- Unlink is equivalent to the following
- *(next->fd + 12) = next->bk
- *(next->bk + 8) = next->fd
- It results that the next chunk is not part of the double linked list anymore.

Attack



- Suppose there is an overflow in a copy operation do the data section of chunk_1.
- The attacker can control what data is written to the next chunk
- When free() is called on the first chunk, the attacker (by overflowing the "correct" values) can force the consolidation with chunk_2
- The attacker can "fake" that the next block is unused, and "fake" values for fd and fk

Attack



- 0xfffffffc values (negative values) are used to pass some checks in the implementation but also to avoid zero bytes
- *(next->fd + 12) = next->bk
- *(next->bk + 8) = next->fd

Attack

- *(next->fd + 12) = next->bk
- *(next->bk + 8) = next->fd
- **Write anywhere primitive!**
- Warning: bytes at next->bk + 8 will be erased!

Conclusion

- A heap overflow attack depends on the heap management implementation
- The attacker has a "write anywhere" primitive to redirect the control flow
- Recent implementation perform checks to ensure consistency, so this "unlink" attack does not work anymore

Question?