Heap Overflow on Metadata

Software Security 2019/2020

Andrea Lanzi

Heap vs Stack

Stack

- **Fixed memory allocations** known at compile time
- Local variables, return addresses, function args.
- Fast and **automatic**, done by the compiler.

Heap

- **Dynamic memory** allocations at runtime
- Objects, big buffers, structs, persistence, larger things
- Slower and **Manual**, Done by the programmer.

Heap Overview

- The heap is pool of memory used for dynamic allocations at runtime:
    - **malloc()** grabs memory on the heap
    - **free()** releases memory on the heap

Heap usage example

```c
#include <string.h>

#include <stdlib.h>

#include <stdio.h>



int main(int argc, char *argv[])

{

 char *buf1 = malloc(128);

 char *buf2 = malloc(256);



 read(fileno(stdin), buf1, 200);



 free(buf2);

 free(buf1);

}
```

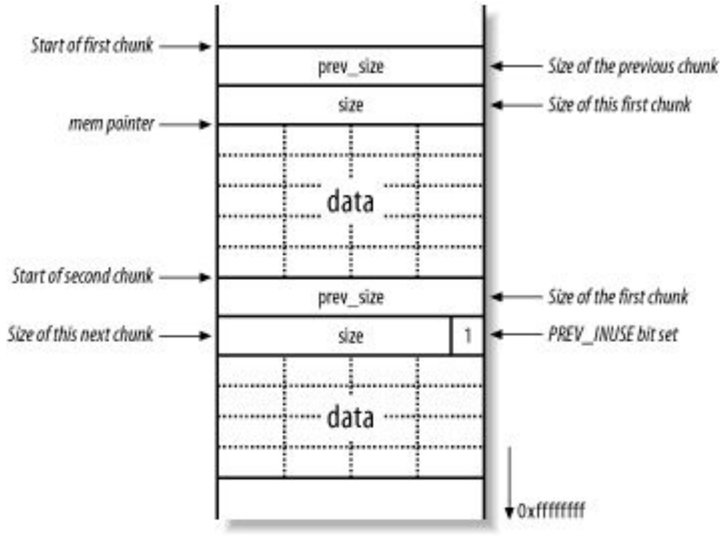
Heap Chunk

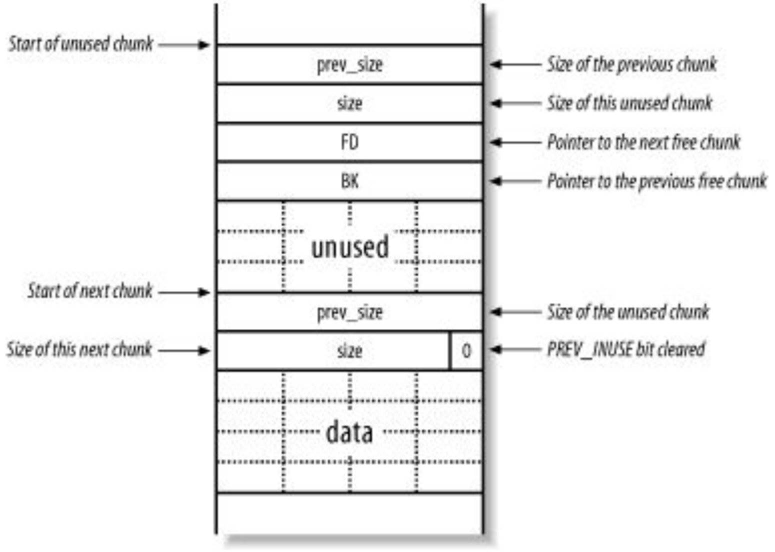
```
struct malloc_chunk {
 INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
 INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */


 struct malloc_chunk* fd;         /* double links -- used only if free. */
 struct malloc_chunk* bk;


 /* Only used for large blocks: pointer to next larger size.  */
 struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
 struct malloc_chunk* bk_nextsize;
};
```

Heap Chunk


Start of first chunk
prev_size
Size of the previous chunk
size
Size of this first chunk
mem pointer
data
Start of second chunk
prev_size
Size of the first chunk
Size of this next chunk
size
1
PREV_INUSE bit set
data
0xffffffff


Start of unused chunk
prev_size
Size of the previous chunk
size
Size of this unused chunk
FD
Pointer to the next free chunk
BK
Pointer to the previous free chunk
unused
Start of next chunk
prev_size
Size of the unused chunk
Size of this next chunk
size
0
PREV_INUSE bit cleared
data

Heap Memory Allocation

- If a program were to call **malloc(256)**, **malloc(512)**, and finally **malloc(1024)**, the memory layout of the heap is as follows:

```
Meta-data of chunk created by malloc(256)
The 256 bytes of memory return by malloc
-----------------------------------------
Meta-data of chunk created by malloc(512)
The 512 bytes of memory return by malloc
-----------------------------------------
Meta-data of chunk created by malloc(1024)
The 1024 bytes of memory return by malloc
 -----------------------------------------
Meta-data of the top chunk
```

Memory Heap Allocation

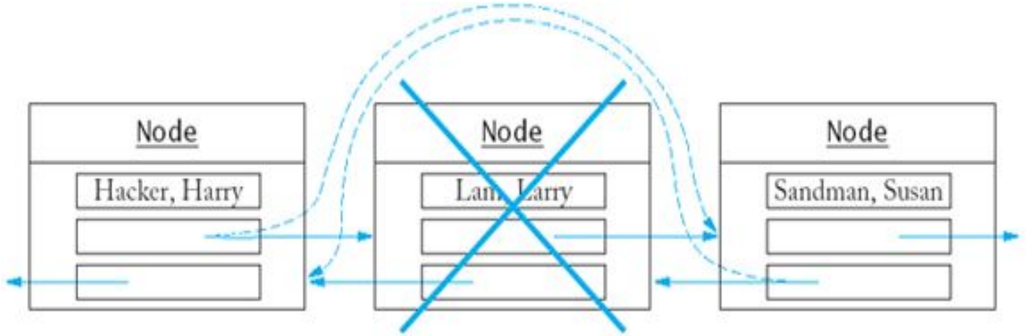
- **top chunk** represents the remaining available memory on the heap.
- When a new memory request (malloc) is made, the **top chunk** is split into two: the first part becomes the requested chunk, and the second part is the new the top chunk (so the "top chunk" shrunk in size)

Heap Deallocation

- When a chunk is freed, the least significant bit of the *size* field in the meta data of the next chunk must be **cleared.**
- Additionally, the **prev_size** field of this next chunk will be set to the size of the chunk we are freeing.
- There are actually **multiple lists of free chunks**. Each list contains free chunks of a specific size.
- When a chunk is freed it checks whether the chunk before it has already been freed. **In case the previous chunk is not in use, it's coalesced** with the chunk being freed.
- When a memory allocation request is made, it f**irst searches for a free chunk that has the same size** (or a bit larger), and will reuse that memory. Only if no appropriate free chunk was found will the top chunk be used.

Deallocation of free chunk and exploit

```
void unlink(malloc_chunk *P, malloc_chunk *BK,
malloc_chunk *FD)
{
        FD = P->fd;
        BK = P->bk;
        FD->bk = BK;
        BK->fd = FD;
}
```



ATTACK This allows us to write an arbitrary value to an arbitrary location

```
...
FD->bk = return address;
FD = P->fd;
BK = P->bk = address of the buffer (injection vector);
FD->bk = BK;
...
```

More Recent Techniques

```
void unlink(malloc_chunk *P, malloc_chunk *BK,
malloc_chunk *FD)
{
        FD = P->fd;
        BK = P->bk;
        if (__builtin_expect (FD->bk != P || BK->fd != P,
0))
                malloc_printerr(check_action,"corrupted
double-linked list",P);
        else {
                FD->bk = BK;
                BK->fd = FD;
        }
}
```

- Unfortunately the technique explained above is no longer possible against newer versions of glibc. The unlink function has been hardened.
- These techniques are called:
    - The House of Prime.
    - The House of Mind.
    - **The House of Force**.
    - The House of Lore.
    - The House of Spirit.
    - The House of Chaos.

House of Force a vulnerable Program

```c
int main(int argc, char *argv[])
{
      char *buf1, *buf2, *buf3;
      if (argc != 4) return;


      buf1 = malloc(256);
      strcpy(buf1, argv[1]);
      buf2 = malloc(strtoul(argv[2], NULL, 16));
      buf3 = malloc(256);
      strcpy(buf3, argv[3]);
      free(buf3);
      free(buf2);
      free(buf1);
      return 0;
}
```

House of Force conditions: Requires that we can overwrite the top chunk, that there is one malloc call with a user controllable size, and finally requires another call to malloc.

House of Force: Exploitation technique

- The av->top variable always points to the top chunk. **The goal is to overwrite av->top with a user controllable value.** During a call to malloc this variable is used to get a reference to the top chunk (in case no other chunks could fulfill the request).
- This means that if we control the value of av->top, and we can force a call to malloc which uses the top chunk, we control where the next chunk will be allocated. Consequently **we can write arbitrary bytes to any address.**
- We want to assure that any request (of arbitrary large size) will use the top chunk. To accomplish this we abuse the overflow in the program to overwrite the metadata of the top chunk. First we write 256 bytes to fill up the allocated space, and we finally overwrite the size with the largest possible (unsigned) integer.

House of Force: Exploitation technique

```
 static void* _int_malloc(mstate av, size_t bytes)
 ....
 top = av->top;
  size = chunksize(top);
  if ((unsigned long)(size) >= (unsigned long)(bytes +
MINSIZE))
  {
    remainder_size = size - nb;
    remainder = chunk_at_offset(victim, nb);
    av->top = remainder;
...
```

define chunk_at_offset(p, s) ((mchunkptr)(((char*)(p)) + (s)))
Av_top + bytes_to_allocate = Address in memory
Bytes_to_allocate = Address in memory - Av_top ;

By writing in memory at any arbitrary address we can execute any arbitrary code.