Heap Exploitation

Stuart Nevans Locke

Note:

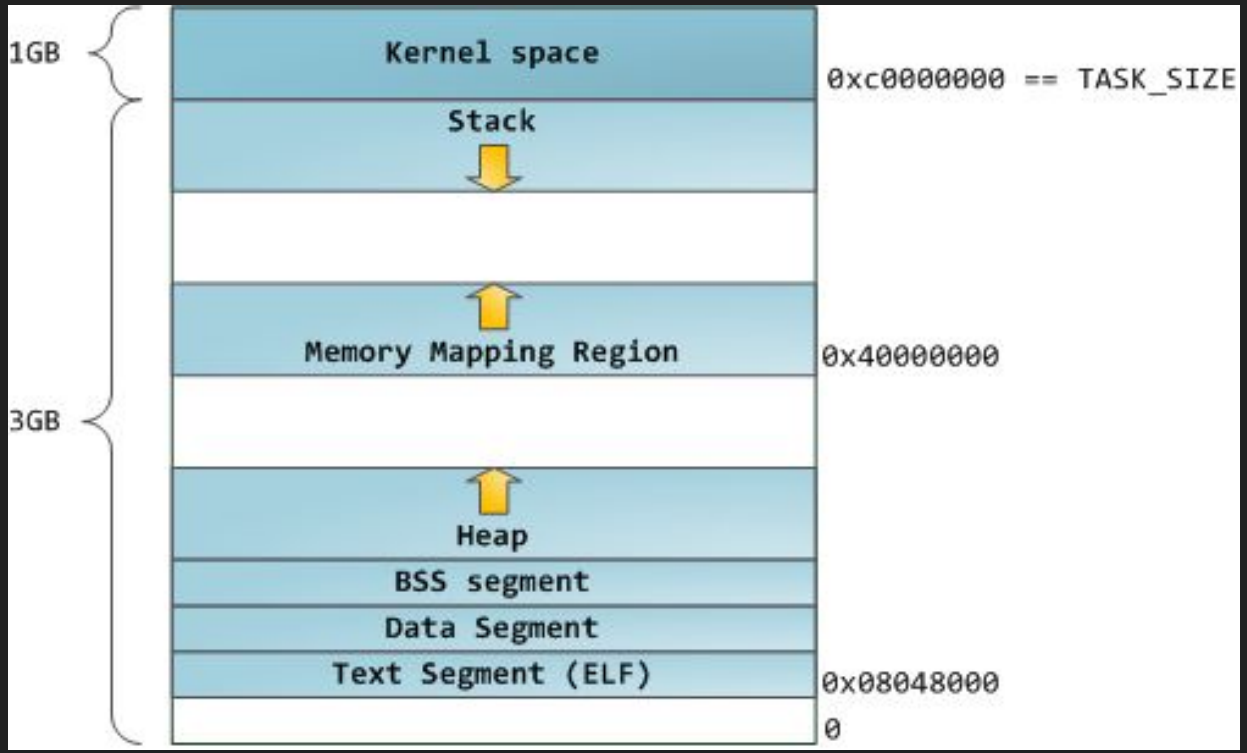
Parts of this presentation are adapted from RPISEC's MBE Heap lecture.

Overview

- Heap Background
  - What is the Heap
  - Heap Chunks
    - What’s in one
- Heap Exploitation
  - Use After Free
  - Double Free
  - Heap Overflow

Heap

- Used for dynamic memory
  - Large data structures
  - Variable size data
- void * malloc(size_t size)
  - Allocates memory on the heap
- void free(void * ptr)
  - Deallocate memory on the heap


1GB
3GB
Kernel space
0xc0000000 == TASK_SIZE
Stack
Memory Mapping Region
0x40000000
Heap
BSS segment
Data Segment
Text Segment (ELF)
0x08048000
0

Heap Facts

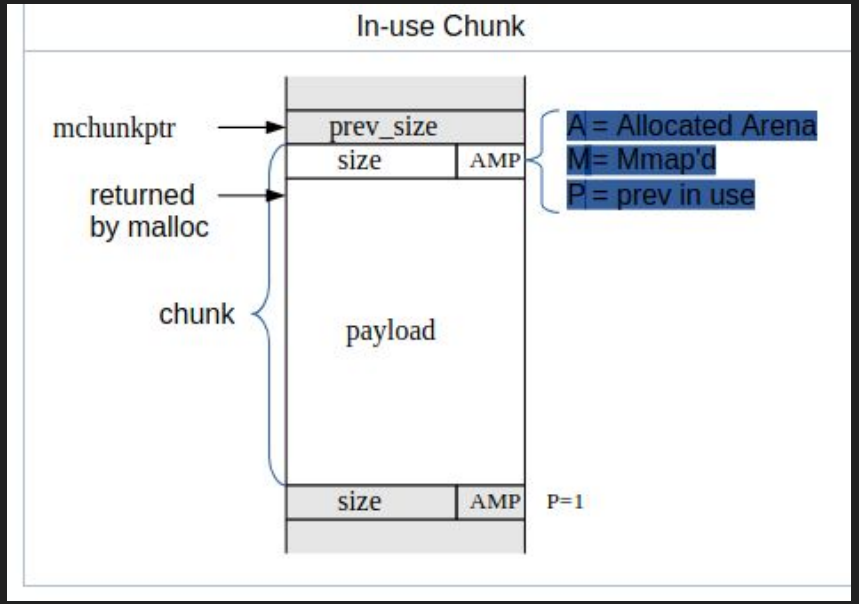
- Slower than stack allocation
- Other functions:
  - Calloc -- malloc, but 0 initialize
    - void * calloc(size_t nmemb, size_t size);
  - Realloc - Allocate new region of requested size with the data originally contained
    - void * realloc(void * ptr, size_t size);
- C++
  - New, delete
- Different implementations are common
  - dlmalloc
  - **ptmalloc -** libc derived from this
  - jemalloc
  - More!

Heap Structure

- How much space do you think this takes?
  - My computer, 64 bit
- malloc(0)
  - 32
- malloc(4)
  - 32
- malloc(16)
  - 32
- malloc(32)
  - 48

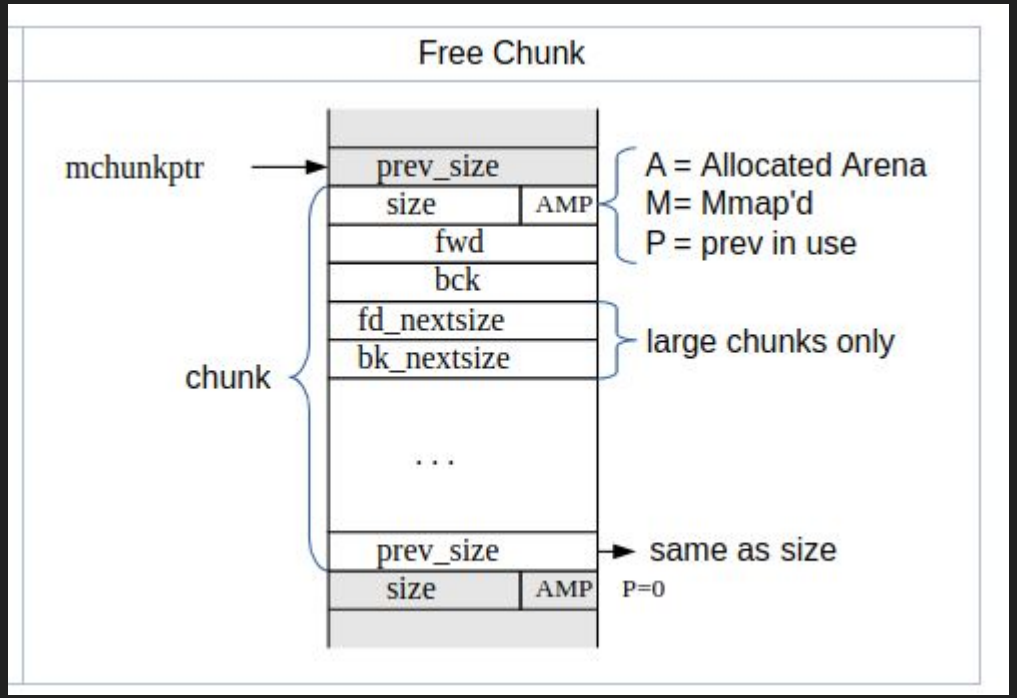
Heap Chunks

- ptmalloc is “chunk-oriented”
  - Every call to malloc() results in a chunk being created, and then returned to the caller
  - Chunks are transparent to the caller, they actually are returned a pointer after the start of the chunk



Heap Chunks - When free

- What happens when you do free(ptr)
- Various metadata gets stored where the user data was before



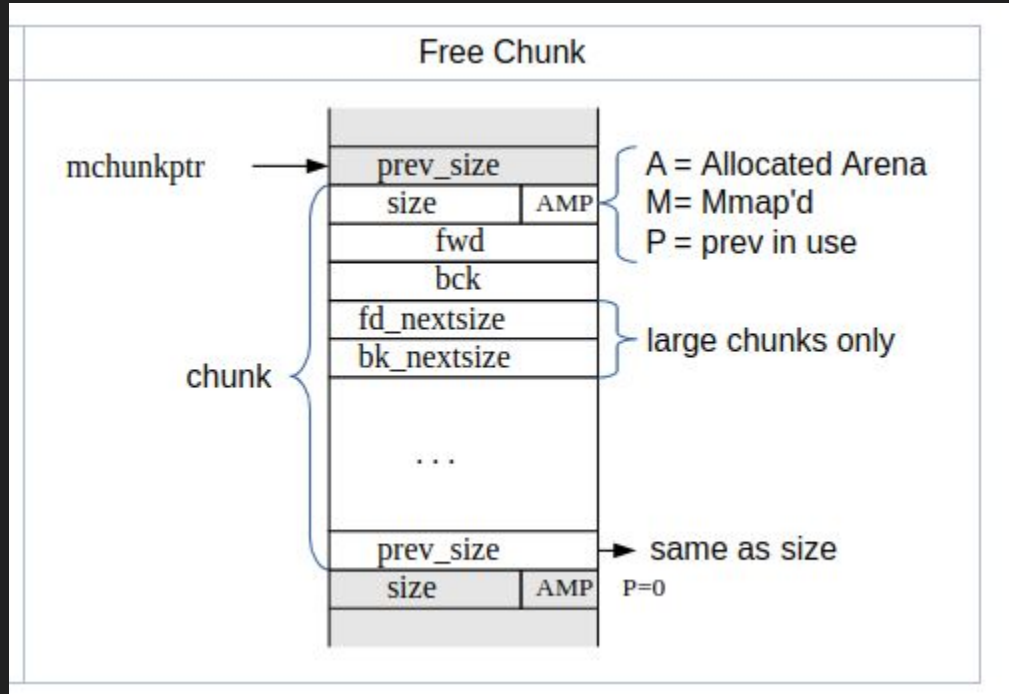
Chunks being weird

Mchunkptr doesn't actually point to the current chunk. It points to the last field of the previous chunk. It's only valid if the previous chunk is free.

AMP

P - previous chunk in use

(mchunkptr not valid)



In use chunk metadata - from RPISEC

```
mallocing...
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x55c9b51a0670) -------> ... ] - from malloc(0)
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x55c9b51a0690) -------> ... ] - from malloc(4)
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x55c9b51a06b0) -------> ... ] - from malloc(8)
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x55c9b51a06d0) -------> ... ] - from malloc(16)
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x55c9b51a06f0) -------> ... ] - from malloc(24)
[ prev - 0x00000000 ][ size - 0x00000031 ][ data buffer (0x55c9b51a0710) -------> ... ] - from malloc(32)
[ prev - 0x00000000 ][ size - 0x00000051 ][ data buffer (0x55c9b51a0740) -------> ... ] - from malloc(64)
[ prev - 0x00000000 ][ size - 0x00000091 ][ data buffer (0x55c9b51a0790) -------> ... ] - from malloc(128)
[ prev - 0x00000000 ][ size - 0x00000111 ][ data buffer (0x55c9b51a0820) -------> ... ] - from malloc(256)
[ prev - 0x00000000 ][ size - 0x00000211 ][ data buffer (0x55c9b51a0930) -------> ... ] - from malloc(512)
[ prev - 0x00000000 ][ size - 0x00000411 ][ data buffer (0x55c9b51a0b40) -------> ... ] - from malloc(1024)
[ prev - 0x00000000 ][ size - 0x00000811 ][ data buffer (0x55c9b51a0f50) -------> ... ] - from malloc(2048)
[ prev - 0x00000000 ][ size - 0x00001011 ][ data buffer (0x55c9b51a1760) -------> ... ] - from malloc(4096)
[ prev - 0x00000000 ][ size - 0x00002011 ][ data buffer (0x55c9b51a2770) -------> ... ] - from malloc(8192)
[ prev - 0x00000000 ][ size - 0x00004011 ][ data buffer (0x55c9b51a4780) -------> ... ] - from malloc(16384)
```

Free chunk metadata

```
struct malloc_chunk {

  INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk (if free).  */

  INTERNAL_SIZE_T      mchunk_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 Exploitation

- Exploit the application
  - More common in real world
  - Use After Free
  - Application dependent
- Exploit the allocator
  - CTFs love this
  - Heap overflow into metadata
  - Double free
  - Allocator dependent, version dependent
- https://github.com/shellphish/how2heap

Heap Exploitation - Use After Free (UAF)

- Exactly what it sounds like. Using memory after it is freed.
  - Extremely common in modern C code. Hard to spot.
- Use can be read or write
  - Read - can be used for leaks
  - Write - can be used to write to other data structures allocated above yours

```c
int func(){
    char * x = malloc(128);
    free(x);
    *x = 1;
}
```

```c
int func(){
    char * x = malloc(128);
    free(x);
    char * y = malloc(128);
    *x = 1;
}
```

```c
int func(){
    char * x = malloc(128);
    free(x);
    printf("%p\n", x);
}
```

Use After Free

- Exploiting these is very application dependent
  - General flow is freeing your target, allocating another structure to fill where your pointer points to, and then “using” the freed memory to mess with the other data structure.
- Decisions:
  - What structure should I allocate after?
  - Can I manipulate the heap to select a certain structure? (i.e. heap groom)

Heap Overflows

- Same as stack overflows, but on the heap.
- There are generally no “heap cookies/canaries”
- You can either overflow into metadata of other chunks, other chunk’s data, or your own chunk’s data
  - CTFs often like metadata
  - Other chunk’s data often easiest
  - Your own chunk’s data results in more reliable exploits
- The heap often has some randomization, because applications allocate based on what they need, and different runs might allocate different structures on the heap

Heap Overflow

```
int func(char * y){
    char * x = malloc(128);
    memcpy(x, y, 140);
}
```

Double Free

- Free the same memory twice
  - Allocator tries to crash when this happens
- Can result in use after free
- Can also result in the same pointer being returned twice

```
int func(){
    char * x = malloc(128);
    free(x);
    char * y = malloc(128);
    free(x);
}
```

Heap Exploitation - Heap Spraying

- “Spraying” the heap with a ton of data
- For example, allocating a lot of identical structures over and over again.
- This allows you to be more certain about things such as guaranteeing a UAF will be using a specific structure

Questions?