Heap Metadata

Leon Chou
OSIRIS Lab Hack Night

OSIRIS

Overview

- What is Heap Metadata?
- Why do we care about Heap Metadata?
- Heap Bug Classes
- How can we put it all together?
- Workshop: Null-Byte Overwrite(Also known as unsafe-unlink)
- Heap Exploitation Checklist

Heap Exploitation Checklist

How to exploit the heap (step-by-step):

1) Look for a way to corrupt the heap
2) Get either an overlapping chunk or control of a freed chunk
3) Use that corrupted chunk to get an arbitrary pointer
4) Arbitrary read with that chunk to leak information
5) Arbitrary write with that chunk to get control of program execution

Storing on the Heap

This is an example of how a classic C program would use the heap:

```
typedef struct
{
    int field1;
    char* field2;
} SomeStruct;

int main()
{
    SomeStruct* myObject = (SomeStruct*)malloc(sizeof(SomeStruct));
    if(myObject != NULL)
    {
        myObject->field1 = 1234;
        myObject->field2 = “Hello World!”;
        do_stuff(myObject);
        free(myObject);
    }
    return 0;
}
```

What does the heap look like?

- The heap itself exists as a very large array in memory
- It is initialized by your program's first call to malloc
- Gets chopped up into small pieces and dished back out to your program.

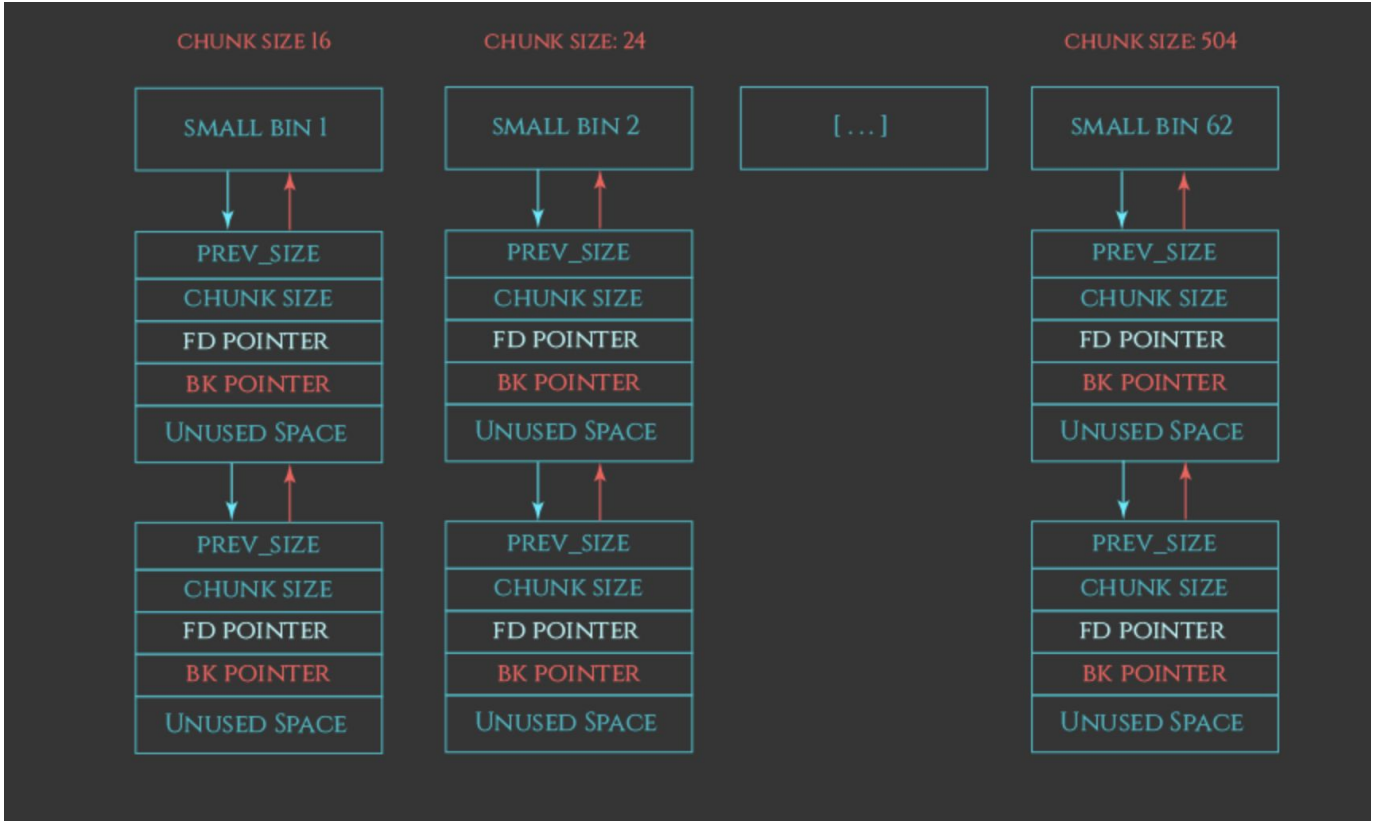
Binning / Recycling Chunks

- Places freed chunks into “bins”
  - Array of linked lists

- Fastbin
- Smallbin
- Largebin

Smallbins

This shows how smallbins are organized



Managing the Heap

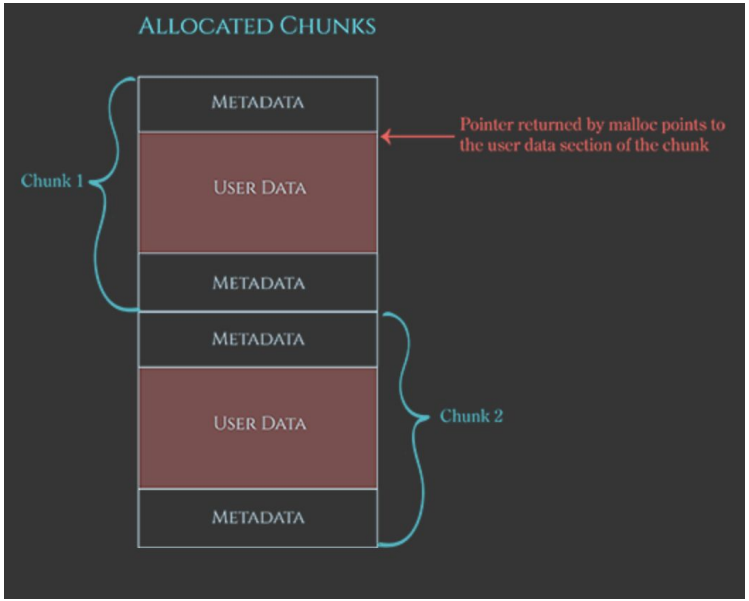
- Heap manager needs to keep track of chunks
  - Lots of info needed for bins

Solution: Heap Metadata

- Job
  - Keeping track of chunks
- Importance
  - Used Heavily -> Efficient
  - Procedural
  - Dynamically allocated memory is key in interacting with user
    - Important that it is used securely
  - Easy to misuse in large, structure heavy programs

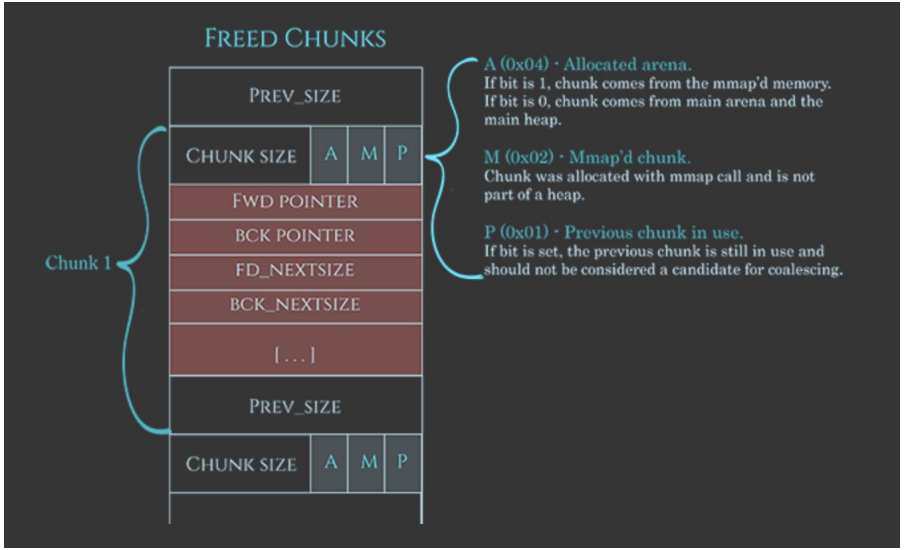
Malloc'd Chunk Metadata

- Metadata
  - Size of chunk



Freed Chunk Metadata

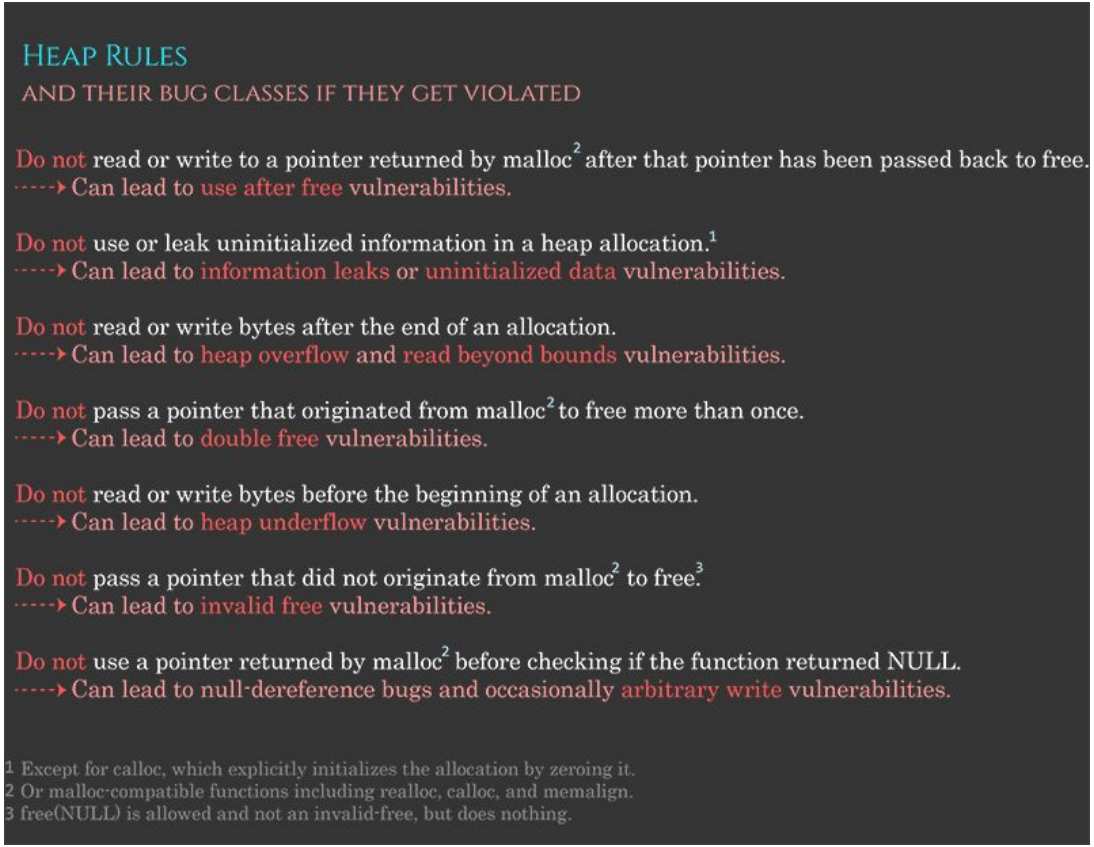
- Metadata
  - prev_size
  - chunk_size
  - forward pointer
  - back pointer



Heap Bug Classes

Bug classes with the Heap are very similar to stack bug classes, but it also introduces some more issues:

This graphic is taken from
https://azeria-labs.com/heap-exploitation-part-1-understanding-the-glibc-heap-implementation/



New Tools

- glibc heap has some debug tools for malloc and free
  - __malloc_hook and __free_hook
    - Pointers to functions that will be passed the arguments to malloc and free

- What happens if you overwrite __malloc_hook or __free_hook?
  - free(ptr to /bin/sh)?

```
void (*hook) (void *, const void *) = atomic_forced_read (__free_hook);
if (__builtin_expect (hook != NULL, 0))
{
    (*hook)(mem, RETURN_ADDRESS (0));
    return;
}
```

How can we put it all together?

1. Get a Libc leak
2. Have arbitrary write
3. Overwrite __malloc_hook

How to get a libc leak from the heap?

- Where do forward and back pointers point?
  - In smallbin chunks → main arena
- Where is main arena?
  - In libc

How to get arbitrary write with the heap?

- Many different ways
  - Corrupt heap metadata
- We'll go over one example today
- Come to the lab to learn more!

How to get control over execution?

- Overwrite __free_hook or __malloc_hook
  - System or magic gadget
- Call free or malloc
  - WIN

Heap Exploitation Checklist

1) Is the program not nulling pointers after freeing them?
2) Is the program not initializing chunks after creating them?
3) Is the program passing incorrect pointers to free()?
4) Is the program overflowing the chunks in some way?
5) Is the program freeing a pointer multiple times?

If the answer to any of these questions is yes, then you have a heap PoC to write!

Heap Exploitation Checklist

During exploitation: (when the heap is unhappy with how you are exploiting)

1) Is top-of-heap coalescing back over my chunks?
2) Am I allocating chunks into the right bins?
3) Am I failing one of glibc's security checks? (this is a good reference for that)

Heap Exploitation Checklist

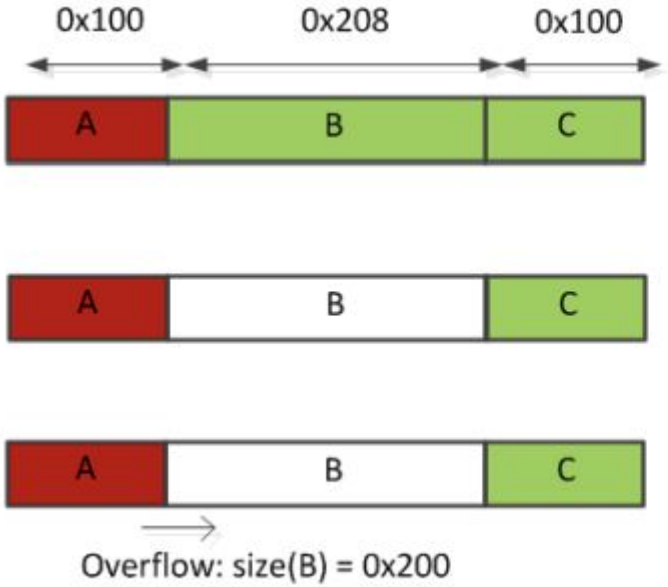
How to exploit the heap (step-by-step):

1) Look for a way to corrupt the heap
2) Get either an overlapping chunk or control of a freed chunk
3) Use that corrupted chunk to get an arbitrary pointer
4) Arbitrary read with that chunk to leak information
5) Arbitrary write with that chunk to get control of program execution

Poison Null-Byte

DEMO

Shrinking free chunks

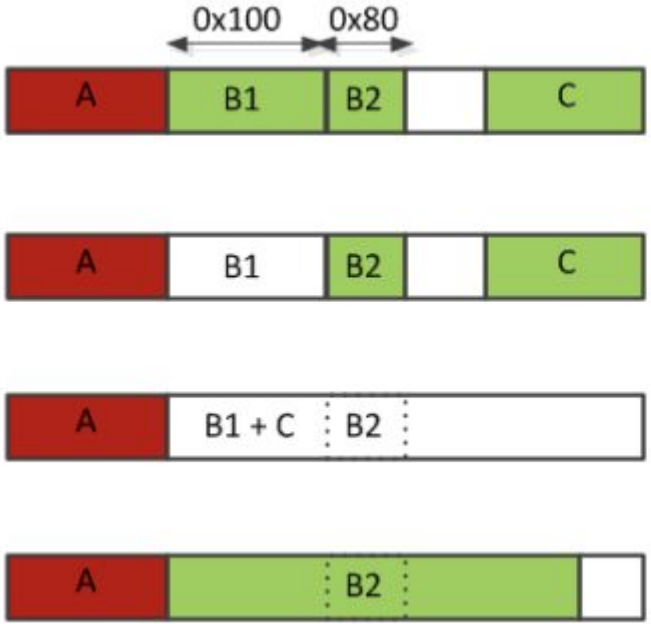


Initial state

B is free

Overflow into B
- Size truncated to 0x200 from 0x208
- Further allocations in that space do not properly update C's "prev_size" field

Shrinking free chunks



Two allocations within the old B chunk
The first is not a fastbin

The beginning of the old B chunk is free

C is freed and merged with the old B, where
a valid non-fastbin free chunk resides

1+ allocations larger than B1's initial size
B2 is overlapped

References

https://azeria-labs.com/heap-exploitation-part-1-understanding-the-glibc-heap-implementation/

https://heap-exploitation.dhavalkapil.com/attacks/

https://www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf

https://github.com/Naetw/CTF-pwn-tips#hijack-hook-function