Intro to Heap

Nathan Huckleberry

University of Texas at Austin

April 22, 2021

Table of Contents

Table of Contents

What does a heap do

- We ask the heap for n bytes of memory
- It allocates n bytes of memory for us to use
- We eventually give the memory back when we're done

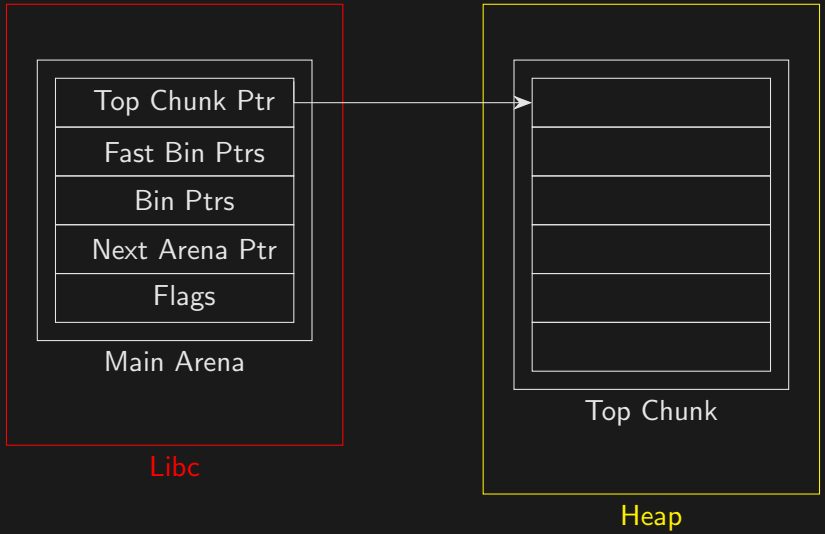
Dynamic Memory

- A program can only request entire pages (contiguous 4096 byte sections) from the kernel
- We want to break up these pages into smaller chunks
- Need some code that manages these chunks for us

Heap Initialization

- The heap will request a large chunk called the top chunk or wilderness chunk
- Subsequent requests from malloc will get a piece of the wilderness chunk
- The heap main arena keeps track of where the top-chunk lives

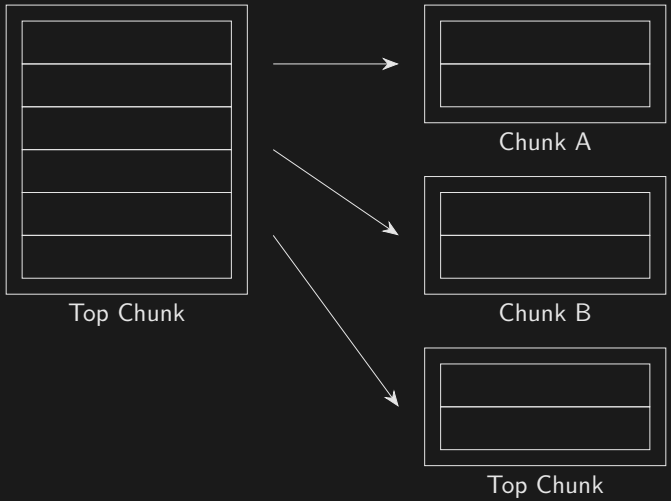
Wilderness Chunk



Allocation

- When the program requests memory, memory is cut off the top chunk
- The top chunk shrinks
- The memory allocated from the top chunk is returned to the program

Allocation



Freeing

- When we free memory, we want it to be made usable for future allocations
- We can only recombine with the top chunk if the freed chunk is next to the top chunk in memory
- Most cases we can't recombine

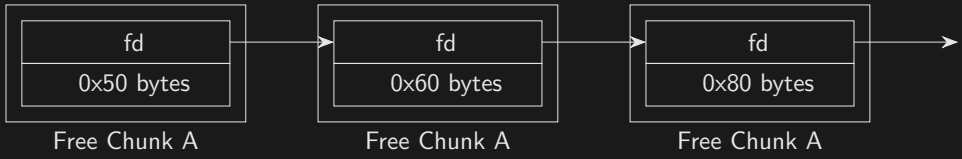
Freeing

- We can't recombine A and the top chunk since B is in the middle

Freeing

- Instead we put it in a list of free chunks
- When we allocate later, we look if there are any suitably sized free chunks in the list before taking from the top chunk
- This free list lookup needs to be extremely efficient

Freeing



- An implementation like above may require enumeration of the entire list for large allocations
- Not efficient enough

Freeing Efficiency

- Chunks are bucketed based on size
- For each size chunk, there is a separate free list
- These are called bins

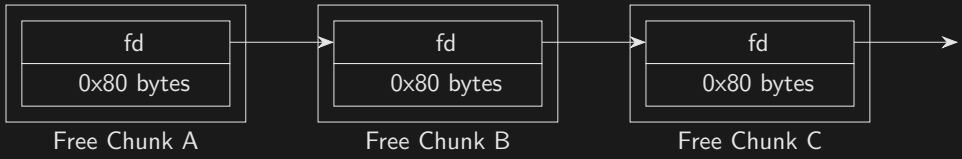
Fast Bins

- Bins for small sized chunks are called fast bins (0x10 - 0xb0)
- They have simplified handling since we expect small allocations to be freed/allocated very often
- Fast bins are the simplest for CTF challenges

Fast Bins

- Fast bins are a singly linked list of free chunks all of the same size
- Allocation is made for small chunk size
  - Round the allocation size up to nearest fastbin size
  - Return first element in fastbin

Fast Bins



- Every chunk in the list is the same size, so we don't need to search

Other Bin Types

- There are other types of bins, namely small bins, large bins and unsorted bins
- These bins are double linked, circular linked lists
- The head/tail are in the arena struct

Table of Contents

Chunk Layout

- In glibc the pointer to the next chunk in the free list is called fd
- The pointer to the previous chunk in the free list is called bk
- Each chunk is allocated with an additional 8 bytes specifying the size

Chunk Layout

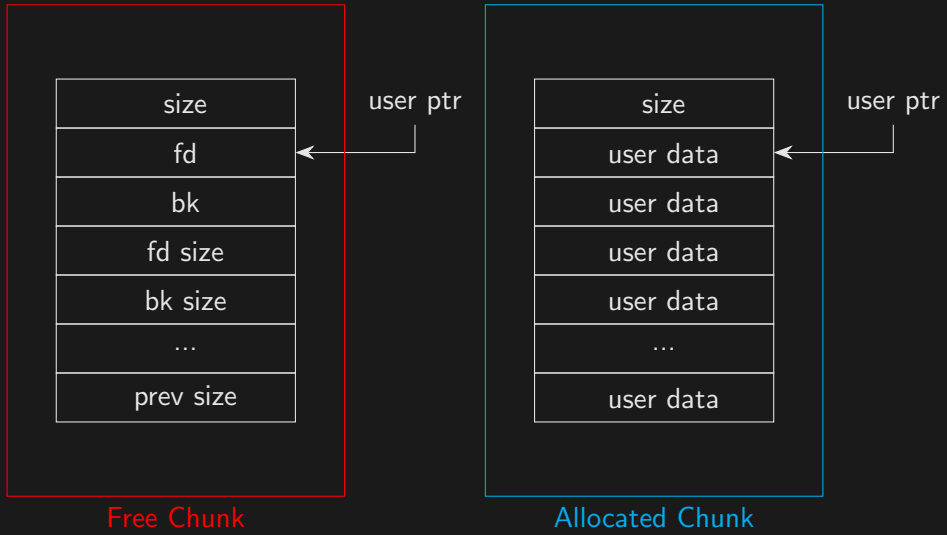
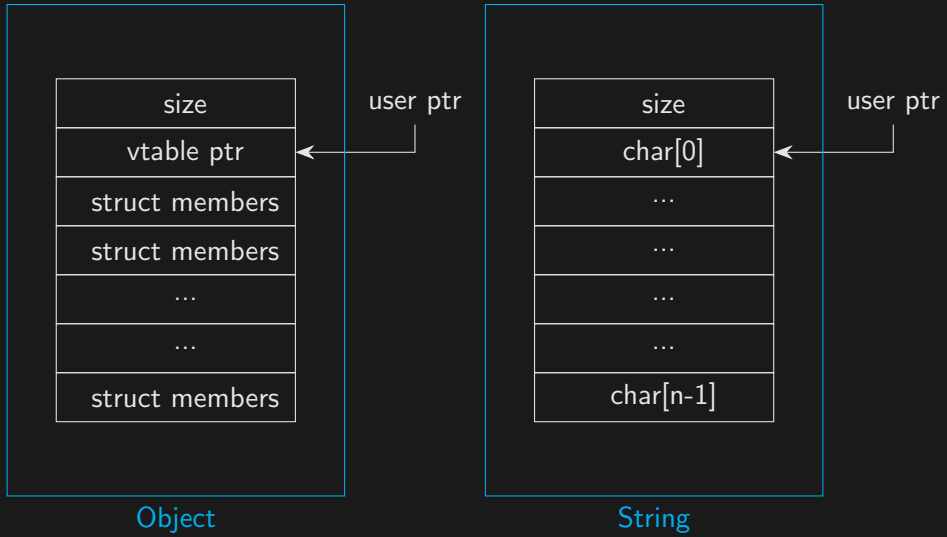


Table of Contents

Overlapping Allocations

- Cause the heap to allocate two different structs at the same memory
- Use these allocations to corrupt the structs
  - Overwrite a function pointer in the struct
  - Cause a stack overflow by corrupting in a creative way

Overlapping Allocations



Forged Chunk

- Cause the heap to allocate a struct outside of the heap region
- Overwrite GOT entries or return address directly

Table of Contents

Overlapping Allocation UAF

- We're allowed to write to the chunk *after* it's been freed
- The next time we call malloc with the chunk size, we get the same memory
- Overlapping allocation comes for free

Overlapping Allocation UAF

- Allocate a chunk of size 0x80
- We get chunk A
- We now have two allocations of chunk A

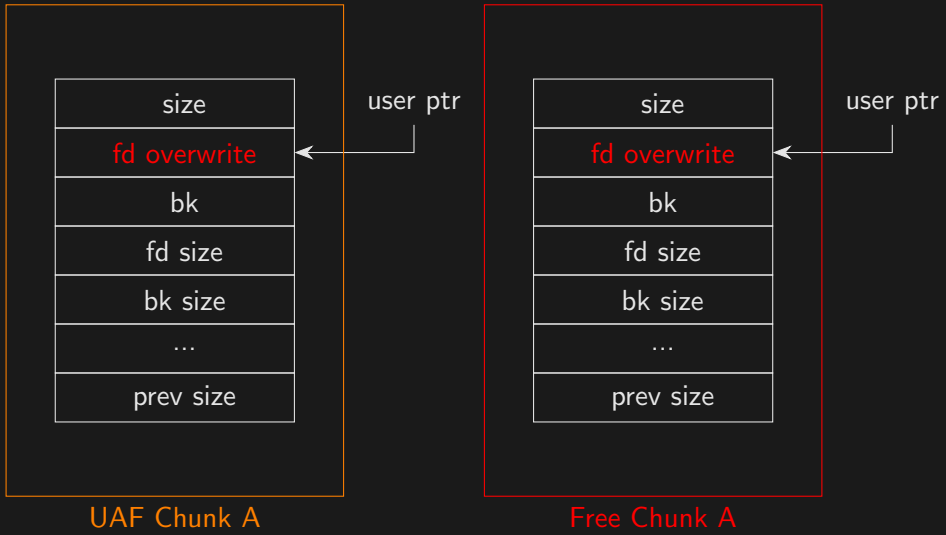
Overlapping Allocation UAF

- Allocate a chunk of size 0x80
- We get chunk A
- We now have two allocations of chunk A

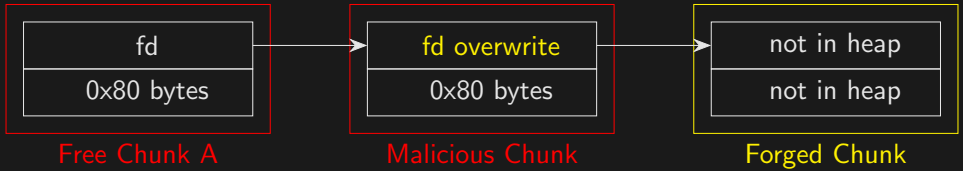
Forged Chunk UAF

- By writing to the freed chunk we can mess with linked list pointers

Forged Chunk UAF



Fast Bins UAF



- After allocating 3 chunks of size 0x80, the heap will return a chunk at an attacker controlled address
- We can then use this chunk to overwrite arbitrary memory (GOT, return address, malloc hook, etc)

Fast Bins UAF

- Overwriting GOT pointers will allow us to take control of the program

In Practice

- Malloc checks to make sure the size matches with the bin we're allocating from
- We need to write *size* somewhere in memory before we can allocate a forged chunk

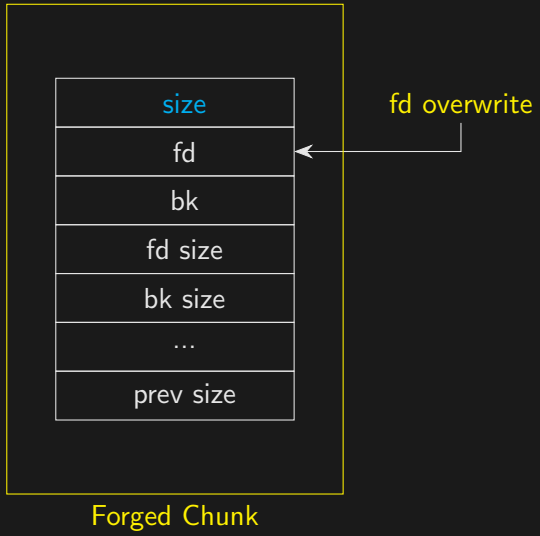


Table of Contents

Double Free

- We're allowed to write to allocated chunks
- We're allowed to free a pointer twice

Double Free

- Freeing the same pointer twice creates a cycle in the free list
- Allocating will give us chunk A and the free list stays the same

Double Free

- Allocate a chunk of size 0x80
- We get free chunk A
- The head of the free list stays free chunk A
- This is enough for an overlapping allocation

Double Free

- We do an fd overwrite as in the UAF case
- Allocating again gives us free chunk A
- Allocating again gives us our forged chunk

In Practice

- The heap has checks to make sure that you're not freeing the head of the free list
- To get around this, we free an additional chunk in between

In Practice

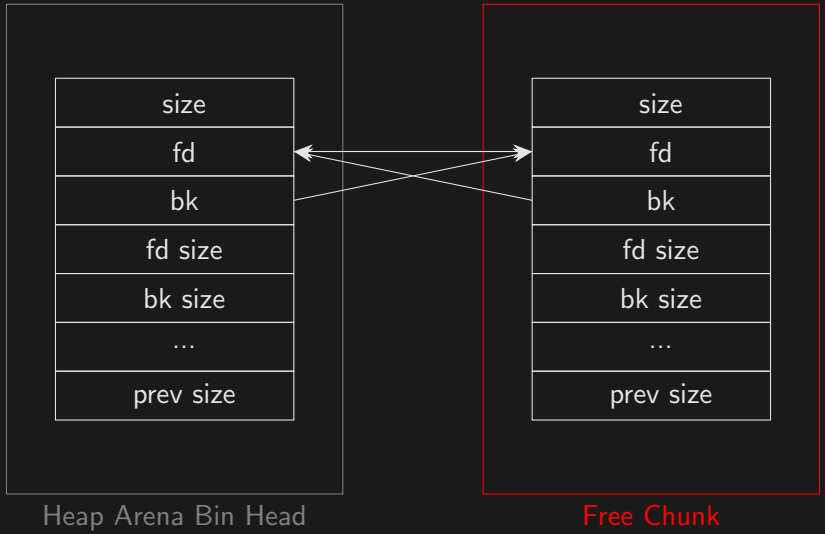
- Free A, Free B, Free A again
- There's still a cycle, it's just length 2 now

Table of Contents

Libc Leaks with UAF

- For allocations larger than fast bins, the free list is doubly linked and circular
- We can use this to leak heap addresses
- We must be aware of the list head in the heap arena
  - A fake chunk in libc so the heap can keep track of the free list

Libc Leaks with UAF



Libc Leaks with UAF

- Printing the fd or bk pointers will give a leaked libc address
- Some address inside the main_arena struct