Medicon Engineering Themes
Volume 6 Issue 4 April 2024
Article Type: Review Article
ISSN: 2834-7218

Dynamic Memory Allocation: Implementation and Misuse

Pratik Prasad Devkota*

Department of Software Engineering, Nepal College of Information Technology, Nepal

***Corresponding Author:** Pratik Prasad Devkota, Department of Software Engineering, Nepal College of Information Technology, Nepal.

Received: March 07, 2024; **Published:** April 12, 2024

Abstract

Heap is a topic of interest among security researchers and exploit developers who are looking for vulnerabilities in modern-day software. The complexity of allocators like the one used by GLIBC incur a broad attack surface. Attackers can exploit an application by misusing the algorithms and metadata used by the memory allocator itself due to various flaws in the implementation. As a result, there are many exploitation techniques related to the heap. However, these vulnerabilities and exploit techniques are more difficult to understand as they can be dependent on the intricate details of how the allocators are implemented. While stack-based attacks have received significant attention and have various mitigations developed, heap-based attacks remain a persistent threat.

This paper provides an analysis of the dynamic memory allocation mechanism used in GLIBC, one of the most widely used C library implementations. We describe the key implementation details of the memory allocator, including the use of segregated free lists, in-band metadata within heap chunks and support for multithreading using arenas. We then demonstrate how vulnerabilities can occur in the heap due to incorrect usage of the memory allocation facilities provided, such as use-after-free, invalid free, heap overflow, and others. To illustrate the severity of these vulnerabilities, we provide an overview of two heap exploit techniques: tcache poisoning and fast bin dup. This paper aims to help developers who want to understand how dynamic memory allocation works under the hood and how to avoid common pitfalls that can lead to serious security vulnerabilities.

Keywords: Heap exploitation; Memory corruption; Low-level vulnerabilities

Abbreviations

Acronym	*Full Form*
GLIBC	*Gnu C Library*
ASLR	*Address Space Layout Randomization*

Introduction

The memory of a process can be divided into different segments or regions, each with its own purpose and characteristics. One of the segments is called the heap. The heap segment is used for storing objects or data structures that are created at runtime. It is often called the region of memory controlled dynamically by the programmer. The process of allocating memory dynamically at runtime is known as *dynamic memory allocation*.

A process does not have a heap segment when it is just loaded into memory. Instead, the extra memory must be requested from the kernel, which allocates chunks of memory and returns a pointer to the program. The programmer is responsible for managing the memory and deallocating it. Since this can be a tedious and challenging task, several libraries exist for managing the heap. These libraries are known as *dynamic memory allocators*.

Dynamic memory allocation provides a great deal of flexibility to developers, but it also poses several challenges. One of the most significant challenges comes from the complexity of the allocators themselves. Since every modern software system relies on those libraries, they need to employ various techniques to ensure efficient memory management. This complexity, however, can also create a broad attack surface on the allocator, making it vulnerable to several types of memory-related attacks. Attackers can exploit vulnerabilities in the allocator itself, compromising the security of the entire program. Knowing how those vulnerabilities occur and how they can be exploited, however, requires a thorough understanding of the lower-level implementation details.

The purpose of this paper is to provide an overview of how a dynamic memory allocator might be implemented and show some of the ways in which it can be misused. We do this by first exploring the implementation details of a common memory allocator used in C/C++ programs on (x64) GNU/Linux. Describing some of the implementation details will require showing code snippets and since the allocator is implemented in C, basic knowledge of the language is assumed. We will then discuss common memory-related vulnerabilities associated with dynamic memory allocation. Finally, we will look at how those vulnerabilities can be exploited by studying two (among many) heap-based exploitation techniques that make clever use of the underlying heap implementation. Through this paper, we hope to shed light on the importance of using safe and secure memory management practices.

The Glibc Heap Implementation

Because dynamic memory allocation is so commonly used, there are a number of memory allocators designed and used for different purposes. Some of them are:

- ***dlmalloc***: A general purpose allocator created by Doug Lea[1]; most of the later implementations are based on *dlmalloc*.
- ***ptmalloc***: Wolfram Gloger[2] Posix Thread aware allocator; extended *dlmalloc* for use with multiple threads.
- ***jemalloc***: A general-purpose *malloc* implementation focusing on less fragmentation and concurrency; used in projects like FreeBSD, Firefox, Android, etc.[3]
- ***kmalloc***: A *malloc*-like routine for the Linux kernel; from Linux's slab allocator[4].

In this article, we'll focus on the GLIBC version of *malloc* which is based on the *ptmalloc* allocator. Since glibc is one of the most common libraries in GNU/Linux systems, it must meet many requirements imposed by applications. So, covering the entirety of its implementation will result in this article getting significantly hefty. Furthermore, the code base changes frequently as new features are added and security vulnerabilities are patched, which makes it very difficult to cover this material. Therefore, only a small portion of the implementation will be covered. It should be assumed that *malloc* refers to the *glibc's malloc* from now on.

When a *malloc* service is requested, glibc utilises two different system calls to request memory from the kernel:

brk/sbrk

These system calls are used to change the end of the program *break*. When ASLR is not enabled, *break* refers to the end of the program data/bss segment. When ASLR is turned on, the *break* is placed at a random offset from the end of the program data/bss segment. *brk* sets the end of break to be the value specified in its argument whereas *sbrk* returns a pointer to the program break after adding the given offset value. Calling *sbrk* with an argument of 0 would thus give the current program break. On Linux, *sbrk* is internally just a wrapper around the *brk* system call.

mmap()

This system call allocates a memory page into the process' address space. While this system call is primarily used for memory mapped I/O, giving the special *MAP_ANONYMOUS* flag makes it allocate a memory page for the process. The limitation it poses is that the allocations need to be page-aligned and thus the sizes will be a multiple of 4096 bytes.

The allocator chooses between *brk/sbrk* and *mmap* depending upon the requested size. By default, if the requested size is greater than 128 KiB, then *malloc* uses *mmap*(*Sourceware.org Git - Glibc.git/tree*).

Arenas

malloc also has support for threading which improves the allocator's (as well as the application by extension) performance since *malloc* doesn't need to block a thread from using the heap when another thread is trying to do it at the same time. It does this by implementing the concept of arenas. An arena is a set of data structures consisting of linked lists formed from free chunks (called bins) and separate heap segments (top chunks). Each thread has its own arena and allocates memory from that arena's bins. This means spawning each new thread will create a new arena for that thread. Since this can cause overhead for applications using a lot of threads, there's a limit as to how many arenas can be created. As defined in the *glibc* source code, the maximum number of arenas is 8 times the number of cores on a 64-bit machine.

When the number of threads exceeds the arena limit, two threads will have to share an arena. In such a scenario, *malloc* will try to lock a shared arena when a thread tries to access it and if the lock fails, the thread will be blocked until the same arena is free.

The arena used by the main thread is called the *main arena* and is created and expanded using *sbrk*. For other threads, the allocator uses secondary arenas which are just arenas specific to a thread. These secondary arenas emulate the behaviour of the main arena using "sub-heaps" created via *mmap*. In the context of the allocator, a heap is the chunk of memory requested by the allocator from the kernel which later serves as the top chunk. The main arena will use *mmap* to expand its heap only if the program break can no longer be expanded.

The behaviour of the sub-heaps is slightly different because they use *mmap* to allocate every subheap. When a sub-heap is created, the allocator supplies a special flag called *PROT_NONE* to *mmap* signalling the kernel to only reserve a virtual address range for the process rather than allocate memory on physical RAM. Just like how *sbrk* was used to expand the initial heap for the main thread, the sub heaps are "expanded" by changing the pages of the previously *mmaped* regions from *PROT_NONE* to *PROT_READ|PROT_WRITE* (giving read and write permissions) using the *mprotect* system call. This is when the kernel would attach physical memory to those pages. The sub-heaps are slowly grown until the initially *mmaped* region is full, in which case the (secondary) arena just allocates another sub-heap.

Heap Chunks

The heap is divided into a series of chunks, which are blocks of memory managed by the allocator. These chunks are simply the pointers returned when we do *malloc*. When the program frees a heap chunk, the allocator adds it back to its free list and may perform various operations to try to optimise the heap such as merging adjacent free chunks or splitting large chunks into smaller ones. Those operations are known as "heap consolidation."

When a *malloc* call is invoked, the allocator will request for a large chunk of memory (much larger than the requested size) from the kernel. Then it will hand off small portions (depending upon the requested size) of the large chunk to the application. This large chunk is known as the "top chunk" or "the wilderness".

Most *malloc* implementations have the behaviour of storing management information within the chunks itself. Each heap chunk has a header field that contains metadata about the chunk. This metadata is known as "boundary tags" described in the original *dlmalloc* implementation itself.

The code listing below shows the metadata that the allocator inserts into a heap chunk. The first two fields - *mchunk_prev_size* and *mchunk_size* - are always present in a heap chunk whereas the remaining ones are only inserted when the chunk is freed. The data type *INTERNAL_SIZE_T* will just evaluate to *size_t* in our case[5]. The *mchunk_size* field contains the size of the chunk (size requested by the application) plus the size of the header (to store additional metadata) whereas the *mchunk_prev_size* field contains the size of the previous chunk. The size information allows chunks to be traversed in either forward or backward direction thereby enabling fast coalescing of adjacent free chunks. Other fields are as described in the comments.

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

    /* double links -- used only if free.
     */
    struct malloc_chunk* phd;
    struct malloc_chunk* bk;

    /* The following fields are only
     * used (if free) for large blocks
     */

    // pointer to the next larger size.
    struct malloc_chunk* fd_nextsize;
    // pointer to previous larger size
    struct malloc_chunk* bk_nextsize;
};

typedef struct malloc_chunk* mchunkptr;
```

The allocator ensures chunk sizes are always 16-byte aligned (e.g., *0x10*, *0x20*, etc. in hexadecimal) for performance reasons, which means the least significant nibble is always zero. The least significant three bits are thus used to hold additional information in the form of a bit-field:

PREV_INUSE (P)

It's the first bit in the bitfield, so its value is *0x01*. This bit is set when the previous chunk is still in use which indicates that it shouldn't be used for coalescing. If this bit isn't set and the *current* chunk is freed, then the allocator can coalesce the current chunk with the previous one (by using the *prev_size* field) in order to create a larger free chunk. Thus, by checking the next chunk's *PREV_INUSE* bit, we can check if the current chunk is allocated or not.

IS_MMAPPED (M)

It's the second bit in the bitfield, hence the value *0x02*. This bit is set if the chunk was obtained through the *mmap* system call. The other two bits are ignored in that case. Chunks obtained through *mmap* neither belong to an arena nor are adjacent to a free chunk(Kapil). When such a chunk is freed, it's immediately returned to the operating system via the *munmap* system call.

NON_MAIN_ARENA (A)

It's the third bit in the bitfield, so its value is *0x04*. This bit is set if the chunk doesn't belong to the main arena. When such a chunk is freed, then the allocator has to search each arena to find the appropriate sub-heap. If this flag isn't set, then the allocator can save some operations since it knows the chunk belongs to the main arena.

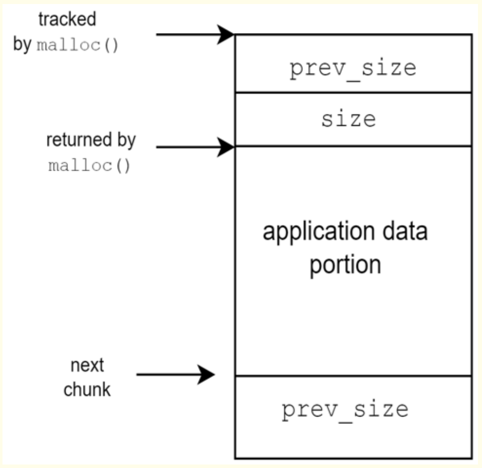

Figure 1: Heap chunk when allocated.

Figure 1 shows a basic layout of a heap chunk when it's allocated (i.e., used by the application). The figure shows that the address returned by *malloc* is different from the actual chunk address that the allocator tracks. The space between those two is used to store the two size fields; which is usually 16-bytes. As an example, suppose we requested a 16-byte (*0x10* in hexadecimal) chunk then the actual size of the chunk tracked by the allocator would be 32-bytes (*0x20*). Interestingly, if the current chunk is in use then the allocator can decide to let the application use the next chunk's *prev_size* field in order to save some memory.

Once we free a chunk, the application-data portion of the chunk is then used to store the *fd/bk* pointers (which point to the next and previous free chunk) as well as the respective size information (i.e., *fd_nextsize*/*bk_nextsize*).

If the current chunk's *PREV_INUSE* flag isn't set (indicating that the previous chunk is free), then the current chunk will be merged backwards to form a bigger chunk. Similarly, if the next chunk is also free then the current chunk is merged forwards. If after merging, the chunks fall into the top chunk, then they aren't put into any bins. Otherwise, the final chunk is placed into the appropriate "bin" in the form of linked lists.

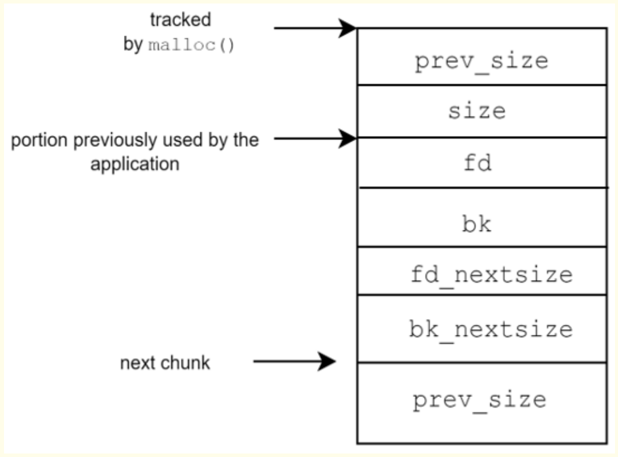

Figure 2: Heap chunk when freed.

Bins

In order to improve performance, *glibc* maintains several linked lists called "bins" used to keep track of free chunks. There are 5 types of bins in total:

Small Bins

These store chunks with sizes less than 1024 bytes. There are altogether 62 small bins each storing freed chunks of a particular size (16, 24, etc.). Each bin is a doubly-linked list and follows the First-In First-Out (FIFO) principle for insertion/deletion. They're faster than large bins but slower than fast bins.

Large Bins

These store chunks with sizes over 1024 bytes. The 63 large bins are similar to small bins except they store chunks within a size range sorted in descending order of their sizes. They're also designed to not overlap with the sizes of the small bins.

Unsorted Bin

The unsorted bin (only 1 in number) is used to store chunks when they're freed. Instead of immediately placing a chunk into the appropriate bin, the allocator places a merged chunk into this bin so that if another *malloc* request of similarly sized chunk is made, the chunks from the unsorted bin can be used immediately without going through the overhead of looking them up in the other bins. The allocator periodically "consolidates"[6] the heap by merging the chunks with adjacent free chunks and placing the resulting chunks into the unsorted bin.

Fast Bins

These bins, altogether 10 in number, are an optimization layer upon the previous bins designed to purpose fast de-allocation/allocation requests. To improve performance, they're intentionally not coalesced (done by *not* setting the *P* bit of the next chunk) with its neighbours. Because of this, they're also stored as singly-linked lists since the back pointers are only useful for coalescing. However, avoiding coalescing can result in fragmentation so the allocator periodically merges adjacent chunks and places them into the unsorted bin. Unlike the previous bins, they're processed in Last-in First-out (LIFO) manner.

Tcache Bins

These bins are an optimization layer upon the fast bins themselves and are specific to each thread. The chunks in the *tcache* contain less metadata, thus saving memory. Additionally, the *tcache* bin is the first priority for a freed chunk to go to which makes it the primary functionality that applications will interact with "behind the scenes". For these reasons, the *tcache* will be described in more depth later.

The allocator maintains an array of these bins in its *malloc_state* structure which holds all of the internal state. There are 128 bins in total as defined by the NBINS constant and the array is defined as such:

```
// inside struct malloc_state

/* Fastbins */
mfastbinptr fastbinsY[NFASTBINS];

/* Normal bins */
mchunkptr bins[NBINS * 2 - 2];
```

Note how the fast bins are tracked separately since they are an optimization layer on top of the three (small, large and unsorted) bins. The size of the bins array is defined as *NBINS*2 - 2* because firstly, *bins[0]* isn't used and the other lists are doubly-linked with each bin header containing two pointers: head and the tail of the list. Also, *bins[1]* is the unsorted bin; *bins[2]* up-to *bins[63]* are the small bins whereas *bins[64]* up-to *bins[126]* are the large bins.

Thread Local Cache

Starting with version 2.26, *glibc* introduced the Thread Local Cache (*tcache*, in short) data structure to make frequent allocations/deallocations more performant. There are altogether 64 singly-linked lists comprising the *tcache* bins with sizes ranging 24--1032. The idea is to have a caching layer to save frequently used (i.e., allocated and/or deallocated) chunks of memory inside a thread, hence called *thread-local* cache. The *tcache* is a small singly-linked list of free chunks that can be used by the thread without locking. It's similar to the fast bins, but the *next* pointers point to the application-data area rather than the chunk header. In other words, the metadata of a *tcache* node is inserted within the first two quad-words of the application-data area of the chunk after it's been freed. The *tcache* is used to improve the performance of *malloc* and *free* calls by reducing the number of times the allocator has to search the free list for a suitable heap chunk or add a freed chunk back to the free list.

```
#define TCACHE_MAX_BINS 64

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

static __thread tcache_perthread_struct *tcache = NULL;
```

Listing above shows that there are two parts in the *tcache* data structure. Each thread will hold a *tcache* variable which is a pointer to an instance of the *tcache_perthread_struct* and is unique to each thread. The *tcache_perthread_struct* maintains a list header for different size allocations and each of those headers go into the *entries* field. In other words, each element in the *entries* array is a pointer to the head of the linked list holding *tcache_entry* nodes. Each index in the counts array of the *tcache_perthread_struct* will hold the number of nodes belonging to each entries element.

A *tcache_entry* node contains only the *next* and *key* fields. This is similar to the chunks in fast bins where the *next* pointer is used to link *tcache_entry* nodes together in the *tcache*. The *key* field is a randomly generated value equal to *tcache_key*. The *tcache_key* mechanism is used so that a chunk may not be freed more than once.

Listing below shows the definition of the *tcache_entry* data structure. Each *tcache_entry* will be assigned to the corresponding entries index based on its size, with a range of 16-bytes: the *tcache_entry* with a size between 0 to 24[7] will be assigned to *entries[0]*, the nodes with size of 25 to 40 to *entries[1]* and so on. It is to be noted that even a request for zero bytes (i.e., calling *malloc* with an argument of *0*) allocates at least 16 bytes.

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

/* Process-wide key to try and catch a double-free in the
 * same thread.
 */
static uintptr_t tcache_key;
```

Following is roughly the strategy followed by the allocator when a *malloc* request is made:

- Search the bins in the arena looking for a chunk that fits the requested size.
- If not found, create a new chunk from the top chunk of the arena.
- Otherwise, request the kernel to supply more memory for the arena then allocate a new chunk from it to service the request.
- If all of the above failed, then return *NULL*.

And the following strategy is followed by the allocator when an allocation is freed:

- If the *tcache* isn't full, then store the chunk there.
- If the chunk is small enough, store it in a fast bin.
- If the chunk has the *M* bit set then, release the chunk through *munmap*.
- Coalesce the chunk with adjacent chunks if possible.
- After merging, place the chunk in the unsorted bin provided it isn't now a part of the top chunk. If the chunk is part of the top chunk, then no operations are performed on it.

Heap Misuse

Heap exploitation is a topic that has been seeing active research resulting in a plethora of exploitation techniques to be developed all targeted at the heap. Many of these exploits have either been patched in the later versions of *glibc* or are no longer effective because of code changes in the allocator. Some exploits, though, are still applicable in the latest[8] versions of the library. These techniques are also well-documented (along with proof-of-concept exploit scripts) in repositories like how2heap[9]. However, most exploits require a flaw in the application as well in order to be effective. Following are a few heap-related bug classes that commonly occur in programs:

Use-After-Free (UAF)

This vulnerability occurs when a programmer uses the pointer to a chunk that was previously freed. Since the heap manager uses a free chunk to store various metadata, a UAF bug may allow an attacker to read and possibly even modify those metadata leading to various memory corruption vulnerabilities. Thus, a good practice is to always assign the freed pointer to *NULL* before reusing it. Furthermore, since an allocation may have data previously used by the allocator, it's therefore always recommended to initialise (dynamic) memory using routines like *calloc* instead of *malloc* before using it.

Heap Overflow

Heap overflows are very similar to stack-based overflows but here the corruption can happen with the metadata of the chunks. An attacker can abuse the *in-band* metadata storing behaviour of the heap allocator to perform various kinds of attacks. Heap overflows are just like any other kind of buffer overflow and thus can only be prevented with proper bounds-checking of data and buffers.

Double Free

A double-free vulnerability occurs when a pointer is freed twice. An attacker can use this vulnerability to make two chunks point to the same location. Thus if an attacker has one of the pointers in control, then they can modify the other leading to various kinds of attacks (even code-execution(Kapil).

Invalid Free

This vulnerability occurs when an application passes a pointer that doesn't belong to the heap - or was even returned by *malloc* - as an argument to *free*. The attacker can use this to forge a chunk into arbitrary locations.

Heap Disclosure

Memory disclosure refers to a fault in the program whereby parts of the application's address space are leaked out to the attacker. By using the leaked address, an attacker can find out the base address of various segments of the process' address space thus defeating ASLR(Shacham et al.). A memory disclosure of the heap is referred to as a heap disclosure.

In this section we'll look at how those vulnerabilities can be utilised to achieve useful (in exploitation perspective) primitives.

Tcache Poisoning

The tcache poisoning attack tricks *malloc* into returning a pointer to an arbitrary location by corrupting the *next* pointers of the chunks located in the tcache bin. This attack utilises a UAF as well as a heap disclosure in order to corrupt the heap metadata with valid (albeit attacker controlled) values.

This technique can be used to force *malloc* into returning a pointer to a stack address. The following strategy can be followed in order to do so:

1. Construct or obtain the required stack address and ensure that it's 16-byte aligned (since the allocator checks that).
2. Then utilise the chunk in which the UAF has occurred to overwrite its next pointer with the stack address.
3. Request an allocation which should give us the chunk whose metadata we corrupted, provided we requested an appropriate size.
4. Now if another allocation request is made, then the allocator will give us the fake stack chunk instead of a valid one. This obtained chunk can be used to perform various stack-based exploitation techniques while also bypassing stack-based mitigations like W^X, Stack Canaries, etc.

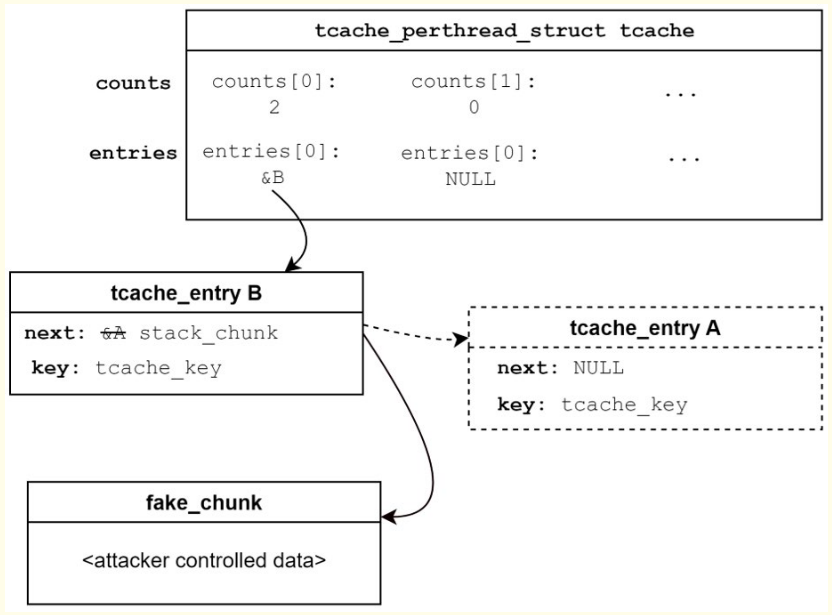

Figure 3: Tcache poisoning attack.

Figure 3 demonstrates what the state of the tcache will look like after the exploit. The figure shows a *tcache_perthread_struct* instance named tcache tracking all of the *tcache_entry* nodes. Before the exploit, the first entry in the tcache had the next field of chunk "*B*" pointing to the chunk "*A*" so the state of the links was: *B->A* (i.e., *B* points to *A*). If the UAF was present in *B*, then overwriting its *next* field would replace the existing link and *next* field point to the malicious chunk named "*fake_chunk*" being controlled by the attacker.

It is to be noted that the value stored at the next pointers aren't the exact addresses of their links but rather a masked value. This value is given by the following equation:

```
new_ptr = (pos >> 12) ^ ptr
```

Here, *new_ptr* is the actual value stored at the *next* fields, *pos* is the address of the next pointers which in the demonstrated example would be *&B->next* (i.e., address of *B->next*), and *ptr* is the actual value we want to write so that would be *&fake_chunk* (the address of the attacker controlled *fake_chunk*) in this case. This mechanism is known as Safe Linking developed to protect the singly-linked lists - including the fast bins - managed by *glibc* (like *tcache*) from tampering(Itkin). This necessitates the knowledge of what the value of *pos* is, which is why we need the heap disclosure.

Fast-bin Dup

The "*fast-bin dup*" is a heap exploit technique that triggers the double-free vulnerability in the fast-bins while also bypassing *glibc's* double-free detection mechanism. The double-free detection mechanism in the fast-bins is a lot simpler (and thus easily by-passed) than the *key* mechanism used in *tcache*. Whenever a chunk is being added to the fast-bin, the heap manager checks if that pointer is the head of the fast-bin list, in which case the application aborts with a "*double free or corruption* (*fasttop*)" message. However, the detection mechanism only works for subsequent *frees* so to bypass it, we can just do the following:

1. Free the first allocation. Let's call it *A*.
2. Then free a different pointer, say, *B* in order to add a different pointer to the head of the fast-bin.
3. Now we can free A again and the double-free detection won't trigger and the fast-bin will have this state: *A->B->A* (i.e. the *fd* pointer of *A* contains the address of *B*, and the *fd* pointer of *B* again contains the address of *A*).

This attack is suitable if the application has a double free vulnerability in a chunk which belongs to the fast bin. This is possible if, for example, the application had allocated some chunks beforehand then it allocated some more chunks and freed them which makes the tcache full, so that the chunks allocated in the beginning will go to the fast bin if freed.

Using this technique, we can access a pointer that is still on the fast-bin free-list. We can continue on the fast-bin dup technique extending it to trick *malloc* into returning a fake chunk (one that, say, resides on the stack).

Suppose we trigger the double-free vulnerability on two heap chunks (which will go to the fast-bin) *A* and *B* so that the fast-bin has the "*A->B->A*" state. Then, we can create a fake chunk --- one that resides on the stack, and make (say) the *fd* pointer of *A* point to our fake chunk. Creating the fake chunk in this case not only requires its address to be 16-byte aligned but also the *size* field to be valid with the proper bits set. If the size of *A* and *B* were 32 bytes, then the size field of our fake chunk needs to be *0x20* (in hex). Here the *PREV_INUSE* bit shouldn't be set since the previous chunk isn't supposed to be in use.

Then we obtain an allocation to the first chunk in the fast-bin i.e. *A* which gives us an allocation, say, *D* (here the address of *D* is the same as *A*). The free-list will still contain links to *A* but we can modify it through *D*. If we modify *D->fd* to have the masked value (with safe linking in consideration) to point to our fake stack chunk, then retrieving *A* which should put the fake chunk at the head of the fast-bin list. If we perform the final allocation, then we'll receive the *fake_chunk* itself as the new heap chunk. We can then write or read arbitrary data to/from the *fake_chunk*.

Conclusion

In conclusion, this paper has highlighted the potential security vulnerabilities that can arise from the implementation of the dynamic memory allocation mechanism provided in GLIBC, and has provided an overview of two heap exploit techniques. It is clear that memory allocation, a fundamental mechanism in programming, can have significant security implications if not handled correctly. While such vulnerabilities are mostly present in systems-level programming languages like C/C++, it is important for developers of any language to ensure that memory has been safely handled in their applications. With the overview of the two exploitation techniques, we see that libraries like GLIBC too have several vulnerabilities. However, those vulnerabilities aren't immediately exploitable without the application itself having a heap-related vulnerability. Therefore, it is essential to follow best practices for secure memory management to avoid common pitfalls that can lead to serious security vulnerabilities.

Features like garbage collection and other higher-level features can help eliminate memory vulnerabilities, but they introduce overhead that may not be affordable for lower-level programs. However, there are still various tools and features available that developers can use to improve the security of their applications, such as smart pointers and tools like Valgrind[10] for detecting memory-related bugs in programs.

In summary, by gaining a deeper understanding of how dynamic memory allocation works and how to avoid common pitfalls, developers can take the necessary steps to ensure that their applications are not vulnerable to heap-based attacks.

Acknowledgements

I am deeply thankful to Yan Shoshitaishvili and Connor Nelson for creating the pwn.college[11] educational platform, helping students like me to learn core cybersecurity concepts. I thank Birendra Bista for encouraging me to write this paper. I thank Dr. Roshan Chitrakar for mentoring me for this paper. I thank Saurab Tharu for providing various suggestions and pointing out mistakes while writing the paper. Finally, I am thankful to Library Genesis and Alexandra Elbakyan for their commitment to free education and preserving

academic freedom.

References

1. "Background". GitHub, https://github.com/jemalloc/jemalloc/wiki/Background
2. Itkin Eyal. "Safe-Linking - Eliminating a 20 Year-Old Malloc() Exploit Primitive". Check Point Research (2020). https://research.checkpoint.com/2020/safe-linking-eliminating-a-20-year-old-malloc-exploit-primitive/
3. Kapil Creators Dhaval. DhavalKapil/heap-Exploitation.
4. Nethercote Nicholas and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". SIGPLAN Not 42.6 (2007): 89-100.
5. Shacham Hovav., et al. "On the Effectiveness of Address-Space Randomization". Proceedings of the 11th ACM Conference on Computer and Communications Security, Association for Computing Machinery (2004): 298-307.
6. Sourceware.org Git - Glibc.git/tree. http://sourceware.org/git/?p=glibc.git;a=tree

Volume 6 Issue 4 April 2024