TECHNICAL BLOG

JULY 1, 2018 BY NSO RESEARCH GROUP

A Tale of Two Mallocs: On Android libc Allocators – Part 1 – dlmalloc

In this series of three posts, we're going to try to cover a deep dive into the pertinent details of the two Android libc allocators, followed by some thoughts on exploitation in light of those allocators.

All of the information I'll impart is the result of our own research into the allocators in question, including a thorough code review of the implementations of those allocators. That said, much of the information is available online in one form or another. I've yet to encounter a concise but in-depth description of both allocators and the relevant exploitation techniques. Hopefully that's what this presentation will provide.

It's 2018. The days of trivially exploitable stack buffer overflows are over. Modern exploitable vulnerabilities fall into a few meager classes, we'll focus on two of these.

Even at this late stage in the game, memory corruption bugs are still a thing. Chief among these is the good old buffer overflow. Stack cookies have largely neutered the exploitability of stack based memory corruptions, so most modern memory corruption vulnerabilities are in objects and buffers on the heap.

In addition to these heap-based memory corruption vulnerabilities, we have use-after-free vulnerabilities.

This class of bugs is all about heap objects coupled with bad memory management practices.

Together these two classes make up a very large portion of the exploitable bugs we find in modern software.

What these classes of bugs have in common is that they both occur mostly in heap objects. Understanding how the heap works is a critical, often overlooked, step in crafting reliable exploits for these kinds of vulnerabilities.

Other prevalent classes of bugs are type confusions and race conditions. We're not going to focus on those here, because they are not necessarily heap-related.

When we talk about the 'heap', what we usually mean is any and all memory objects which are managed using the libc malloc/free interface. This very simple interface lets us allocate so-called "dynamic memory" for our use, and free it when we are done using it. When we approach the task of exploiting a heap-overflow or a use-after-free, it's not enough to know the semantics of this interface. We need to know what is happening under the hood.

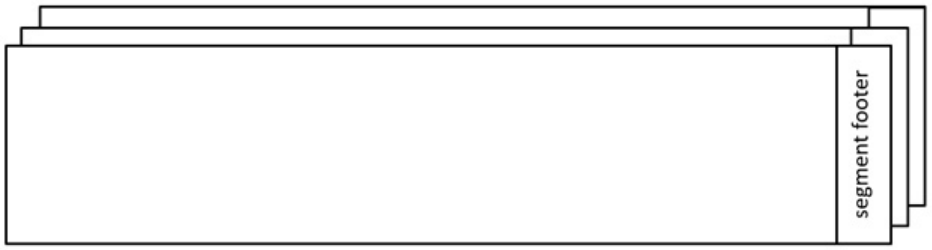
Android uses its own libc implementation, called bionic. When the Android developers came to implement these heap functions, they wisely chose to use an existing, battle tested implementation instead of rolling their own.

dlmalloc

The dynamic memory allocator implementation they chose is called dlmalloc. It's named after its author, Doug Lea. Doug started writing this allocator way back in 1987. It has received many updates and improvements over the years, and was last updated in 2012.

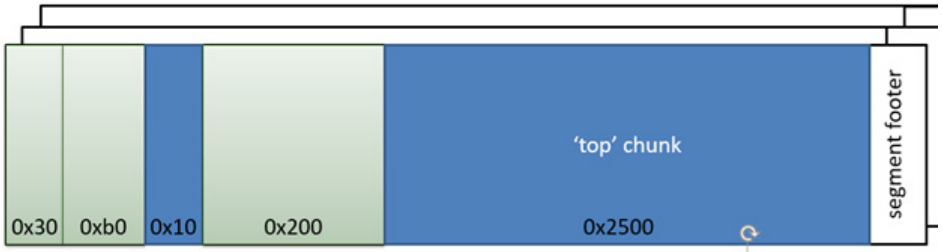
When you call malloc, dlmalloc does a bunch of stuff behind the scenes, and will eventually return a pointer to a block of contiguous memory which you can use in your program. This block is called a '*chunk*', and is guaranteed to be at least as big as the size you requested.

These chunks don't come from nowhere. When dlmalloc needs memory to use for chunks, it requests an allocation from the operating system. Each such system allocation is called a '*segment*'.

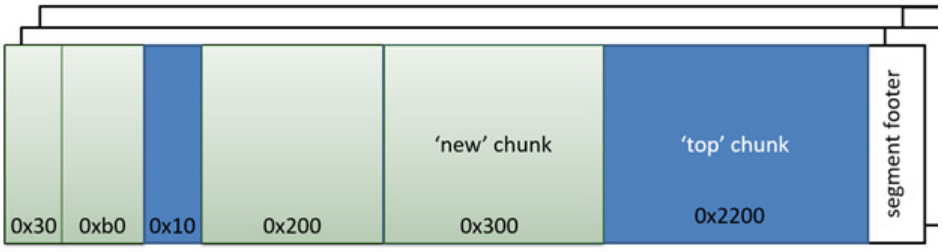


Segments are the base unit of allocation from the OS. dlmalloc keeps a linked list of segments it has allocated from the system, with the pointers stored in the segment's footer. The most recently allocated segment is the 'current' segment. When it needs more system memory, dlmalloc first tries to extend the current segment using sbrk, falling back to mmap-ing a new segment if that doesn't work. Segments can be of different sizes, but are always a multiple of the page size. Segments are not guaranteed to be adjacent to one another in memory, and, in fact, are allocated at random addresses when system-wide ASLR is enabled, as it is on Android. If a new segment happens to be contiguous to an existing segment, the two segments are consolidated into a single larger segment.

The current segment contains the *'top chunk'*, which is the chunk of free space available for immediate allocation of chunks. Here's an example 'current' segment, with in use (allocated) chunks in light green and free (unallocated) chunks in blue.



When dlmalloc needs to allocate a new chunk for a malloc call, it will check if the top chunk is big enough to contain the new chunk, and will carve the new chunk from within the top chunk by splitting it. The first half of the top chunk becomes the new chunk to be returned, and the second half becomes the new 'top chunk'. If the 'top chunk' is not large enough to contain the new chunk, a new segment is allocated from the operating system, and the new chunk is allocated from that new segment.

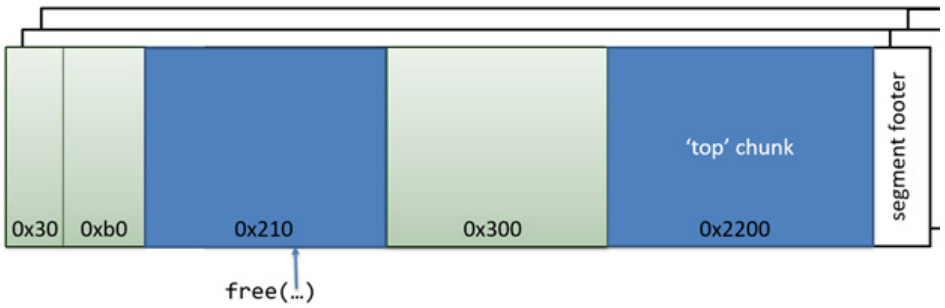


Each chunk has two pointers worth of metadata: in 32bit processes this is 8 bytes. This metadata sits directly before the pointer returned by malloc, i.e. inline before the useable memory. The minimum amount of actual usable memory returned by malloc is two pointers wide.

```
   *chunk->+--------------------------------+
       0x00|size of previous chunk          |
           +--------------------------------+
       0x04|size of chunk   |C_INUSE|P_INUSE|
*user_mem->+--------------------------------+
       0x08|               ...              |
       0x0c|               ...              |
           +--------------------------------+
```

Chunks of different sizes can be allocated one after the other in the segments. Each chunk marks its size and whether it is in use or not, via the C_INUSE flag. It also marks whether the previous chunk in the segment is in use, with the P_INUSE flag, and the previous chunk's size. Because the metadata contains the size of the previous chunk, we can easily walk backwards through the chunks in a segment.

When you call free on a given chunk, the first thing that happens is that dlmalloc checks to see if the preceding chunk is in use. If the preceding chunk is free, dlmalloc will consolidate the two chunks into one larger free chunk.



This means that it is impossible for two consecutive chunks in a segment to both be free. The chunks immediately before and after a free chunk are both guaranteed to be in use.

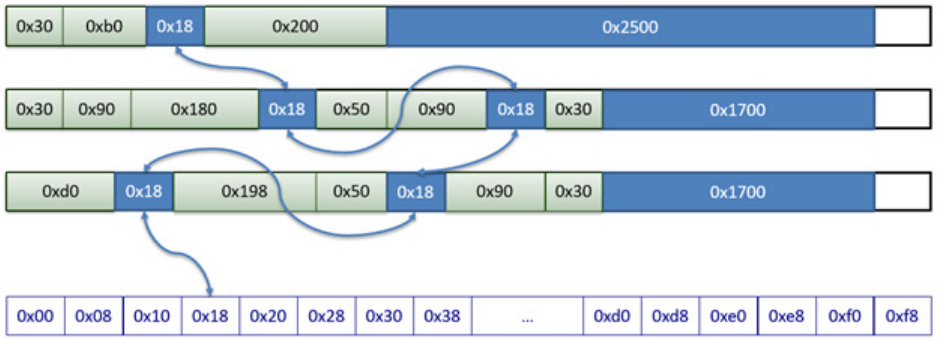
Simple right? Obviously what we've described is a pretty naïve allocator implementation. There's a little more to it. Specifically, what we've described is a system which never reuses freed memory, as it always allocates from the 'top chunk'. So how do we efficiently reuse freed memory?

We need some *bins*.

Bins are used to keep a record of recently freed chunks which can be reused. There are two types of bins: 'small' and 'tree'. *Small bins* are used for chunks smaller than 0x100 bytes. Each small bin contains chunks of the same size. *Tree bins* are used for larger chunks, and contain chunks of a given range of sizes. Small bins are implemented as simple doubly-linked lists, and tree bins are implemented as bitwise digital trees (aka 'tries'),

keyed on chunk size. There are 32 small bins and 32 tree bins.

When a chunk is freed, it undergoes consolidation if needed, and then the consolidated chunk is added to the appropriate bin for its size. The list and tree node pointers are stored within the actual chunk data, which is safe to use for metadata as it is 'free'. This is where the minimum size for a chunk comes from: we need space for previous and next pointers in the free chunk's data.



Here's an example showing a few segments with some in use and free chunks. The 0x18 bin points to the first of the free chunks of size 0x18, and the rest of them are chained together in a doubly-linked-list.

Note that small bins contain chunks of exactly one size. Tree bins contain ranges of chunk sizes.

dlmalloc is a best fit allocator. It will always try to find the free chunk with the smallest size greater or equal to the request size.

During allocation, before looking at the 'top chunk', dlmalloc will first try to find a free chunk in the bins. It first tries to find a chunk which matches the exact size of the allocation request, and then moves upwards through the non-empty bins till it finds the smallest chunk which is larger than the request. If a larger chunk is used, it will be split, and the remainder will be added to the relevant bin to possibly be used for future allocations. Only if no chunk exists in the bins to satisfy the allocation request will the 'top chunk' be used.

Note that the bins are First In First Out. So chunks are allocated in the order that they were freed. This can be an important factor in exploitation.

After looking in the bins for an exact size match, but before going to the 'top chunk', dlmalloc will try to see if the 'designated victim' is large enough to contain the allocation request.

The 'designated victim' is the preferred chunk for servicing small requests that don't have an exact fit. It is the chunk which was most recently split off. It doesn't sit in any bin. Having the 'designated victim' helps to localize allocations to a given memory segment, which can be useful when considering how CPU caches work. Small allocations which don't have an exact fit in the bins will be split off from this chunk.

So for a small allocation, a request size smaller than 0x100 bytes, this is the flow:

- We first calculate the exact size including metadata and padding
- We then look for an exact match in the small bins
- If that fails, we next see if the 'designated victim' is large enough to allocate from
- If the 'designated victim' is too small, we then look for a 'best fit' in the small bins larger than our request size
- If that fails, we look in the tree bins for a 'best fit' match
- Finally, if all else has failed, we look at the top chunk, potentially causing more memory to be allocated from the system.

Larger allocations are a little simpler. We just try to allocate from the tree bins before attempting the 'designated victim' and then the top chunk.

There are no bins for so-called very large allocations, which means anything larger than the MMAP_THRESHOLD, which is 64kb on Android. These allocations don't come from the segments. Instead, each such allocation is mmaped directly.

So that's dlmalloc in a nutshell. Hopefully I've covered all the salient points. There are a couple of things we should note before moving on.

While dlmalloc takes some steps to reduce fragmentation of the heap, particularly the reuse of freed chunks based on bin size, it is still common for smaller free chunks to become trapped between larger consecutive chunks which often remain in use for longer periods in application flow.

dlmalloc is not thread safe. At all. Both malloc and free touch process global data structures and the inline metadata between chunks and inside free chunks. Remember that dlmalloc was designed long before the Age of Parrellism, before every application was multithreaded, before hyper-threading and multi-core processors. To make dlmalloc usable in multi-threaded processes, Doug Lee chose the simplest possible fix: the big lock.

Every single malloc or free call locks a global mutex on entry and unlocks at function exit. This makes dlmalloc usable with threads, but has a major performance impact. Essentially all allocator operations are serialized. This is ok on lightly multi-threaded processes, but can be a significant drag on more complex applications

The poor multithreading performance of dlmalloc is one of the main reasons that the bionic developers decided to switch to a more modern heap implementation.

That wraps up the discussion of dlmalloc. Read the next post in this series to find out about jemalloc, the more modern Android libc allocator.

TECHNICAL BLOG

JULY 1, 2018 BY NSO RESEARCH GROUP

A Tale of Two Mallocs: On Android libc Allocators – Part 2 – jemalloc

In the first post of this series, we discussed why it is important to understand the inner workings of the libc heap allocator, and did a deep dive into the original Android libc allocator: dlmalloc. In this post, we'll examine the allocator which replaced dlmalloc as Android's allocator.

The new allocator selected by the bionic developers is called jemalloc. It is named after its implementer Jason Evans. Jason started implementing jemalloc in 2005. It was then added to FreeBSD's libc to become that platform's default allocator. In 2007, the Firefox Mozilla project adopted the stand-alone version of jemalloc as their primary allocator. Since 2009, jemalloc has been used extensively in Facebook's backend servers. It's currently maintained by a team at Facebook.

During May 2014, jemalloc was added to the bionic source tree for Android 5.0.0. dlmalloc continued to be the default, but it was possible to select this new heap implementation using a board config flag. In July 2014, jemalloc was made the default.

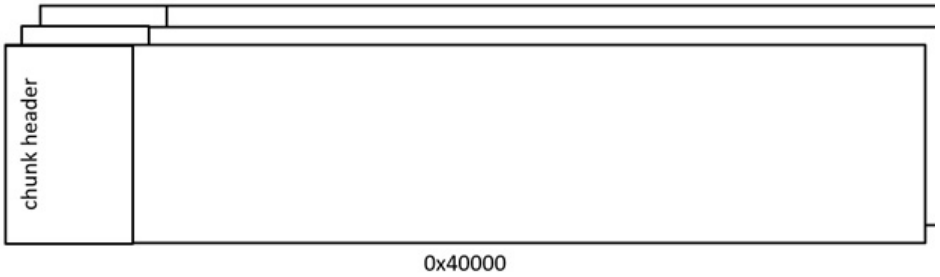
In an ideal world, every vendor of an Android phone would have made the transition to jemalloc with their Android 5.0.0 ROMS. Unfortunately, many vendors chose to remain with the tried and tested dlmalloc heap until later versions of Android. In fact I've seen dlmalloc being used on both lollipop and marshmallow devices. The fact that there is no clear line in time separating the two implementations means that you cannot really know a priori whether a given Android 5 or 6 device is using dlmalloc or jemalloc.

jemalloc was designed from the ground up to be highly-performant in symmetric-multi-processing environments. It has many features which are geared towards increasing efficiency and locality in multi-threaded apps, while reducing overall fragmentation.

The first important concept in jemalloc's implementation is the arena. Each thread is assigned to a given arena, and it allocates and frees only from that arena. Each arena is completely separate from other arenas, and most importantly they have separate mutexs guarding their data structures. This means that you can actually perform allocator operations in parallel so long as the threads involved are assigned to different arenas.

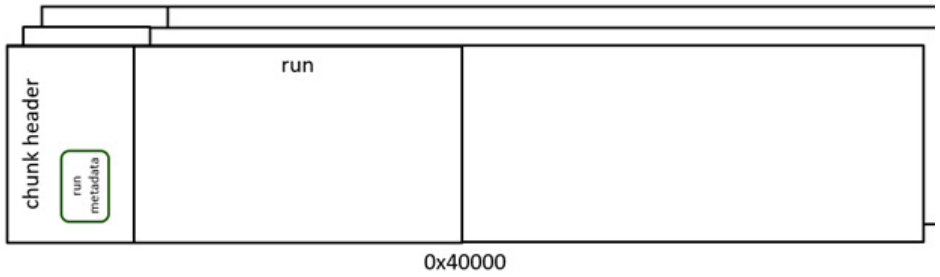
In general jemalloc usage, there should be slightly more arenas then there are hardware cores. For some reason, on android, this is not the case. Instead there are exactly two arenas.

Threads are assigned to arenas in a round-robin fashion, which should ensure that the arenas have a more or less equal number of threads.

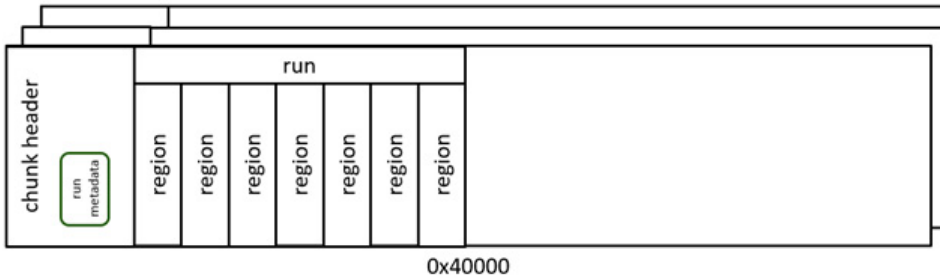


In jemalloc, memory is allocated from the operating system using mmap. Each mmap operation allocates a chunk. jemalloc chunks roughly correlate to dlmalloc segments. Chunks are all of the same size, 256k bytes on Android versions up to 7.0.0. From 7.0.0, chunks are 512kB for 32-bit processes and 2MB for 64-bit processes. Each chunk belongs to a specific arena. There is a chunk header containing metadata for this chunk, specifically including a pagemap which defines which pages are associated with which runs.

For any jemalloc managed address, the relevant chunk header can easily be found by simply rounding down the address to the chunk size. This means that we have o(1) lookups of metadata in most situations.



A run is an area of contiguous memory, located in a chunk. Each run contains a fixed number of regions of a specific size. Different size classes have different numbers of regions. Runs are always a multiple of the page size. Run metadata is stored in the chunk header for the chunk which contains them. In other words, the metadata is out-of-band. Each run has a bitmap which indicates the state of each region in the run. A region can either be in-use or free.



Regions are the smallest unit of the jemalloc system. These are analogous to dlmalloc chunks, except that regions do not carry any metadata at all. Instead each region belongs to a run of regions of the same size. The run stores the metadata for all its regions in the chunk's header. The region address is the return value from a malloc call, and should be the argument to free.

jemalloc is, at its core, a bucket allocator. Each arena has a set of logical bins, each of which services a specific size class. Allocations are made from the bin with the smallest size large enough to fit the allocation request. On Android, there are 39 bins. By having a carefully selected and limited list of bin sizes, with small steps between them, fragmentation can be decreased.

Note that on dlmalloc, bins are used only as free lists. On jemalloc, bins are used for ALL allocations.

index	addr	size	runcur
0	0xb48803e8	0x8	0x95f01490
1	0xb4880490	0x10	0x94400968
2	0xb4880538	0x18	0x982c0620
3	0xb48805e0	0x20	0x944414e4
4	0xb4880688	0x28	0x94340b0c
5	0xb4880730	0x30	0x94440818
6	0xb48807d8	0x38	0xada40cb0
7	0xb4880880	0x40	0x9444143c
8	0xb4880928	0x50	0x942c10a0
9	0xb48809d0	0x60	0x944402d8
10	0xb4880a78	0x70	0x96e40b60
11	0xb4880b20	0x80	0x944009bc
12	0xb4880bc8	0xa0	0x98c802d8
13	0xb4880c70	0xc0	0x94440ff8
14	0xb4880d18	0xe0	0x944804d0
15	0xb4880dc0	0x100	0x94401538
16	0xb4880e68	0x140	0x98c010a0
17	0xb4880f10	0x180	0x96841490
18	0xb4880fb8	0x1c0	0xb40801dc
19	0xb4881060	0x200	0x9448071c
20	0xb4881108	0x280	0x96840bb4
21	0xb48811b0	0x300	0x944802d8
22	0xb4881258	0x380	0x95c00cb0
23	0xb4881300	-	-
24	0xb48813a8	0x500	0x942c0134
25	0xb4881450	0x600	0x98ac0134
26	0xb48814f8	0x700	0x98f40818
27	0xb48815a0	0x800	0x944413e8
28	0xb4881648	0xa00	0x942c0b0c
29	0xb48816f0	0xc00	0x94480fa4
30	0xb4881798	0xe00	0x942c0e54
31	0xb4881840	0x1000	0x96901538
32	0xb48818e8	0x1400	0x944812ec
33	0xb4881990	0x1800	0x960011f0
34	0xb4881a38	-	-
35	0xb4881ae0	0x2000	0x96900ff8
36	0xb4881b88	-	-
37	0xb4881c30	0x3000	0x98ac0cb0
38	0xb4881cd8	0x3800	0x95fc0ff8

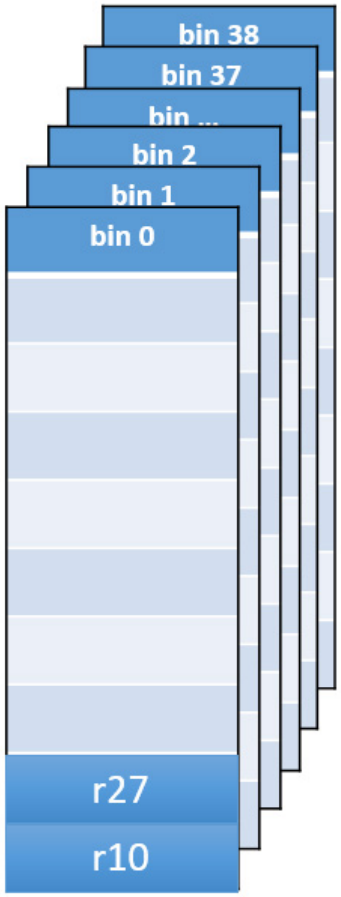
Bin metadata is stored in the arena structure. Each run is associated with a specific bin. Each bin has a ‘current’ run which points to the non-full run from which it is currently allocating. Here you can see the 39 bins for this arena, with their metadata addresses, size classes and current run pointers.

If a run becomes full during an allocation, jemalloc will check if there are any non-full runs for this bin in the arena. If more than one non-full runs exist, the one with the lowest address will be selected and set as the ‘current’ run. If no non-full runs are available in this bin, a new run will be created in either an existing chunk or in a new chunk, and that run will be set as the ‘current’ run of the bin.

Arenas keep track of their non-full runs and available chunk space using a set of red-black trees. Finding a non-full run or available space for a new run is thus at most an O(log(n)) operation.

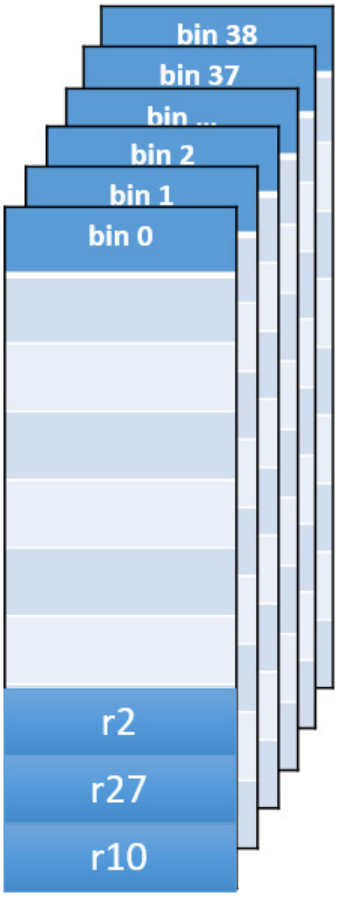
jemalloc reduces lock contention in a few ways, thereby improving multi-threaded performance. Firstly, each arena has its own locks, so operations on different arenas do not contend for locks. Secondly, the critical time is very short. The lock only needs to be held when allocating new runs to a given bin, or when flipping the in-use bit of a region in a run. These mechanisms already make jemalloc significantly more thread-friendly than dlmalloc. However, Jason didn’t stop there. He also implemented thread specific caches.

For each thread, for each bin there is a tcache. The tcache is a list of recently freed regions for the specific bin and thread.



When allocating, jemalloc first looks to see if there is a region in the tcache for the required size's bin before going to the 'current' run for that bin. If so, it uses that region.

When freeing a region, jemalloc pushes the region onto the tcache for the relevant bin. The tcache is LIFO.

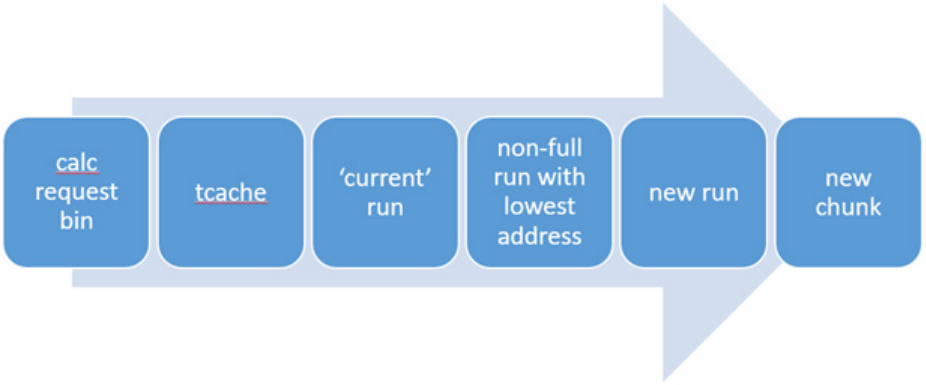


Regions which are currently held in tcaches do not have their in-use bit set to free. Instead they are considered by the greater jemalloc system to be in-use. This saves on locks, as it is only necessary to grab a lock when updating global data structures. Thread specific data structures are by definition safe from other threads, and thus in many cases jemalloc allocations will not grab locks at all.

If jemalloc tries to allocate a region of a given size, and the thread's tcache for that bin size is empty, a pre-fill event will occur. When prefilling, jemalloc will lock the arena mutex, 'allocate' a number of regions for this bin from the 'current' run, marking their bits in the run's bitmaps as in-use, push these regions onto the thread's tcache and release the lock. This ensures that there are always a 'sane' number of regions in a tcache, and significantly improves locality, as a given thread will allocate regions of the same size from mostly contiguous memory.

Each tcache has a maximum number of regions which it can contain. For small bins this is 8, and for larger bins this is 20. When we reach this maximum a flush event occurs. At a flush event, jemalloc takes the oldest half of the tcache's regions and really frees them. I.e. it grabs the lock and marks the region's bits as free. At this point they are free to be allocated by other threads.

In addition, jemalloc implements a 'garbage collection' mechanism. Essentially, jemalloc counts each allocation and free event. When that count reaches a certain threshold, a so-called 'hard event' occurs. Each 'hard event', jemalloc looks at a specific bin across all threads, and clears out three-quarters of the regions from the tcaches for that bin. During the next 'hard event' the next bin will be targeted for cleanup. This is another way that regions can be removed from tcaches and returned to general availability.



So when allocating on jemalloc, we observe the following flow.

- We first calculate the bin for our request size
- We then look in the tcache of the current thread for the calculated bin
- If the tcache is empty, we prefill from the bin's 'current' run
- When the current run is exhausted, we prefill from the non-full run with the lowest address
- If there are not enough regions in the existing non-full runs, a new run will be allocated in a chunk which has available space
- If no space is available in a chunk, a new chunk is allocated from the system, and a new run is allocated in that chunk and is used to prefill the tcache.

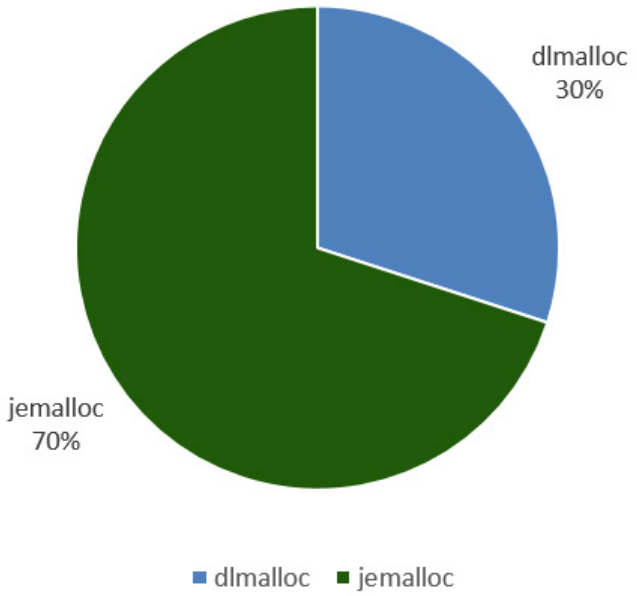
So now we've covered the essential details of jemalloc.

	dlmalloc	jemalloc
allocator type	best fit	bucket
inline metadata	✓	✗
user allocations	chunk	region
allocation from system	segment	chunk
fixed size chunks/regions	✗	✓
adjacent allocations same size	✗	✓
fine grained mutexes	✗	✓
thread-specific free lists	✗	✓
garbage collection	✗	✓

Let's compare some of the important properties of dlmalloc and jemalloc.

- dlmalloc is a best-fit allocator while jemalloc is a bucket allocator
- dlmalloc uses in-line metadata
- user allocations on dlmalloc are called chunks, in jemalloc they're called regions
- dlmalloc allocates variable sized segments from the system, while jemalloc allocates fixed-sized chunks
- jemalloc always allocates from fixed size regions. dlmalloc chunks can be arbitrary 8-byte aligned sizes,
- In dlmalloc, adjacent allocations are usually not the same size. In jemalloc they are.
- dlmalloc only has the big lock, jemalloc has fine grained mutexes which reduce lock contention
- jemalloc has thread specific free lists (aka tcaches) to further increase multithreading performance
- jemalloc has a garbage collection mechanism which helps to clean up tcaches.

- Recently freed chunks or regions on dlmalloc are reused in a FIFO fashion, while on jemalloc they are reused LIFO.



In our estimation, we believe that the current distribution of devices in use is about 70% jemalloc and 30% dlmalloc. This is largely due to the fact that most people update their phones relatively frequently, skewing the distribution towards the more modern jemalloc based systems. Even though the bulk of devices are on jemalloc, it is still necessary to exploit dlmalloc devices in order to get good real-world coverage. For example, certain geographical regions are more likely to have dlmalloc based phones.

TECHNICAL BLOG

JULY 1, 2018 BY NSO RESEARCH GROUP

A Tale of Two Mallocs: On Android libc Allocators – Part 3 – exploitation

In the two previous posts of this series, we've discussed how the Android libc allocators work. In this last post of the series, we can try to determine what we need to do in order to exploit a heap memory corruption or use-after-free, in light of these allocators.

Exploiting these kinds of bugs is all about precise positioning of heap objects. We want to force certain objects to be allocated in specific locations in the heap, in order to form useful adjacencies for memory corruption, or reuse of a desired location for a use-after-free.

The operations performed in order to force the allocator to allocate the objects we want in the positions we want is known as heap shaping or heap 'feng shui'. We essentially need to take advantage of our understanding of the inner workings of the allocator in order to allocate desired objects in the locations or with the adjacencies we desire.

One of the things which can really make a huge difference when implementing heap shaping for an exploit is having a way to visualize the heap: to see the various allocations in the context of the heap.

For this we need tools. These tools need not be especially complex. We don't really need a fancy GUI to visualize regions or chunks. A simple tool which will allow us to inspect the heap state for a target process during exploit development will make an incredible difference.

As it happens, argp and vats presented a tool for visualizing the jemalloc heap at last year's infiltrate. They later released the tool, which they call shadow, on github with support for both firefox's standalone jemalloc and Android's libc jemalloc. I highly recommend viewing their talk and using their tool. Shadow is a debugger plugin which allows you to view the various jemalloc structures in a textual table format. It really makes a big difference in understanding how the heap is behaving during exploit development.

Despite the fact that it is more than 30 years old, we were unable to find a similar visualization tool for dlmalloc at the time we were working on it. So we wrote one.

We've release shade, a dlmalloc visualization tool, on github at https://github.com/s1341/shade. The tool has an interface very similar to that of shadow.

You can request info about a given chunk using its address. This tells you the size of the chunk, its status and the segment to which it belongs.

```
(gdb) dlinfo 0x7c8ee548
[dl] Info about 0x7c8ee548
[dl] chunk: 0x7c8ee540
[dl] size: 0x00000020
[dl] usable size: 0x00000018
[dl] status: in use
[dl] segment base: 0x7c8e3000
```

You can get a quick picture of the chunks before and after your chunk using the dlaround command. This shows a table of chunks centered around your chunk, including their addresses, sizes and statuses.

```
(gdb) dlaround 0x7c8ee548
data address    size    status
--------------------------------
0x7c8ee260      0xc8    free
0x7c8ee328      0x20    in use
0x7c8ee348      0xf0    free
0x7c8ee438      0x20    in use
0x7c8ee458      0xf0    free
0x7c8ee548      0x20    in use
0x7c8ee568      0xd8    in use
0x7c8ee640      0x20    in use
0x7c8ee660      0xa0    free
0x7c8ee700      0x20    in use
0x7c8ee720      0xa0    free
```

Once you know a segment base address, you can use the dlsegment command to view all the chunks in that segment.

```
(gdb) dlsegment 0x7c8e3000
data address    size      status
--------------------------------
0x7c8e3008      0x18      in use
0x7c8e3020      0x18      in use
0x7c8e3038      0x10      in use
0x7c8e3048      0x10      in use
0x7c8e3058      0x10      in use
0x7c8e3068      0x80      in use
0x7c8e30e8      0x28      in use
0x7c8e3110      0x20      in use
0x7c8e3130      0x10      in use
0x7c8e3140      0x2b0     in use
0x7c8e33f0      0x18      in use
0x7c8e3408      0x18      in use
0x7c8e3420      0x80      in use
0x7c8e34a0      0x10      in use
0x7c8e34b0      0x18      in use
0x7c8e34c8      0x18      in use
0x7c8e34e0      0x2b0     in use
0x7c8e3790      0x18      in use
```

shade currently works only in gdb with Android's libc dlmalloc in 32bit ARM processes. This is mostly because that has been what we've needed it for. Pull requests are, of course, more than welcome!

While preparing for this presentation, I found that the excellent ncc group released their own tool for visualizing dlmalloc heaps about 6 months ago. While I have not used it actively, it seems like a great tool.

Let's get back to exploitation. For the purposes of this discussion, let's assume that you have discovered a 0-day heap buffer overflow in application X. It's time to exploit it.

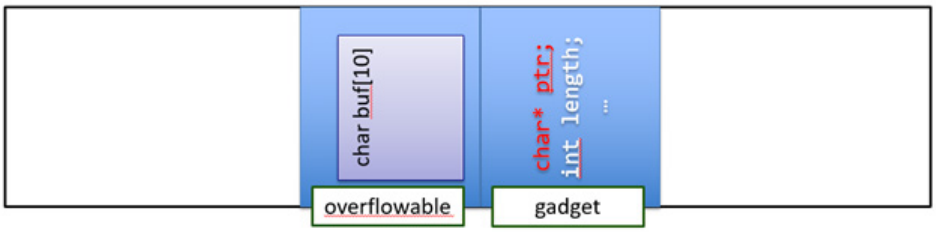
Modern exploitation is hard. The Android system is one of the most heavily hardened platforms out there. Android devices implement a whole host of mitigations to make exploiting vulnerabilities and gaining control of devices more difficult. Things like Address Space Layout Randomization, selinux and process sandboxing are designed to make end-to-end exploitation as painful as possible.

We can, however, overcome some of these mitigations by the generous application of persistence, creativity and luck. We usually end up breaking the exploitation down into a few steps, each of which uses a gadget of some sort.

Gadgets are specific exploitations of your vulnerability set to achieve one or more functionalities which can be chained together to create an end-to-end exploit. We usually need one or more gadgets to actually gain something useful. The gadgets might come from different ways of exploiting a single vulnerability, or from different vulnerabilities.

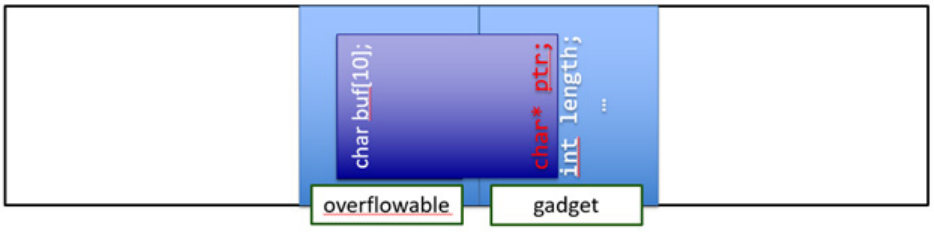
We usually need a relative read or write gadget to overcome ASLR, followed by arbitrary read/write or execute gadgets to gain code execution. Once we have code execution, we can try to escape sandboxing or to evade selinux.

How do we go about creating a gadget from a heap buffer overflow?




```
void gadget_read(gadget* g, char* out){
    memcpy(out, g->ptr, g->length);
}
```

We do this with adjacency. What we want to do is to position the object containing the overflowable buffer just before another object, which we'll call the gadget object]. We need to be able to trigger operations on that gadget object, which will, for example, use a pointer inside the object's data structure to perform a read operation.




```
void gadget_read(gadget* g, char* out){
    memcpy(out, g->ptr, g->length);
}
```

We can then cause the overflow, overwriting the pointer in the gadget object's data structure with a pointer of our choice. We then trigger the read operation to read data at our desired pointer.

Using adjacency, we have created an arbitrary read gadget. We can use similar techniques to create write and execute gadgets. The key is to have a useful operation, which we can repurpose as a gadget by modifying object data through our overflow.

In order to achieve this adjacency, we need to shape the heap such that our exploitable object is allocated just

before our gadget object.

Note that on dlmalloc the two objects can be of completely different sizes, as dlmalloc allows allocations of various sizes to be contiguous. For jemalloc, however, the overflow and gadget objects must be of the same size class, so that they will be allocated from the same bin, in the same run. This can greatly increase the complexity of exploitation on jemalloc, as it is necessary to find gadget objects which not only provide useful functions, but which fit in the right bin.

If you find an object which will work for jemalloc, it may work for dlmalloc too, but the reverse is not true.

So how do we go about performing the shaping necessary to get our objects allocated as desired? Remember that our interface to the allocator is very simple. We essentially only have the allocate and free operations. We need to use those to perform our shaping. What we need are some useful allocation primitives.

We need some way to cause the target application to perform allocations and frees. Usually we'll look for some discrete, easily triggerable, functionality, such as processing of a particular kind of network packet, or usage of a particular kind of object, which performs allocations or frees as a side-effect of its normal operation.

It's important that the primitive perform the allocation or free operation we want, at the time that we want it, so we can use it to gain control of the heap. A given exploitation might require more than one primitive.

The ideal primitive will allow us to allocate an arbitrary size, fill it with attacker controlled data and free it at a later stage with another application trigger.

Although the ideal primitive is the holy grail, we will often have to make do with lesser primitives. For example, a primitive which only allocates at a specific size, or one which allocates memory we cannot reliably free later. Some of these lesser primitives are more useful on jemalloc, some on dlmalloc.

The primitives you'll find in your target app really depend on the specific details of the target app. They might be the result of sending certain types of packets to the target, performing operations in a scripting context such as javascript, or creating and freeing objects through higher level, more abstract, operations.

Finding good primitives is a bit of an art form. It is often necessary to reverse a lot of code before useful primitives can be found. So what should one look for?

You can start by looking for raw mallocs. These are probably the simplest primitives. A function which performs a raw malloc, either with a size you can control or of a fixed size can be a useful primitive. Sometimes the allocation will be for a multiple of a structure size.

Another good source of primitives are c++ classes which are allocated with the new operator. In most cases this new translates directly to a malloc for the size of the class. If you can easily trigger allocation and freeing of classes of interesting sizes, you have yourself an allocation primitive. These can also serve as excellent gadget objects.

Reallocs can also be a good source of primitives. They are often used with zero-length arrays in C structs in order to create variable length 'contained' arrays of sub objects. realloc is essentially the same as malloc.

If your target process uses the C++ standard template library, a good place to look for allocation primitives is in std::vectors. The growth of those vectors causes a 'new' behind the scenes which is essentially just a malloc. As the vector grows, it might fit into the size bin you are interested in. It may be necessary to add objects iteratively until the vector grows to the correct size.

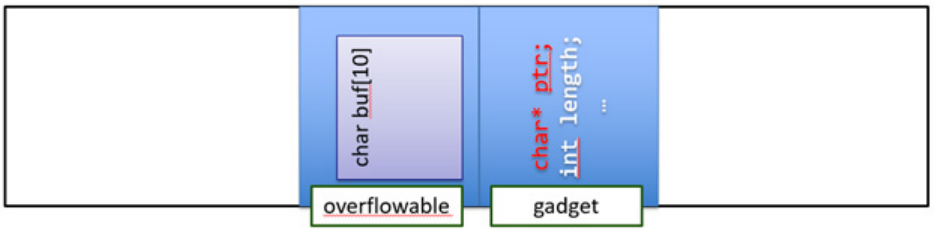
std::string also uses malloc to allocate backing stores for the strings it contains. If you can allocate a std::string from a raw buffer, the size of which you control, you have an excellent primitive. Note however that std::strings are often passed by value, which causes their data to be copied into new allocations. This could be a source of 'allocation noise' which you might want to avoid.

In general, it's necessary to think creatively about the accessible operations in your target in order to find the allocation primitives you need.

Let's walk through an example exploitation with notes on some issues you might encounter on each of the allocators.

First, let's define our asssumptions:

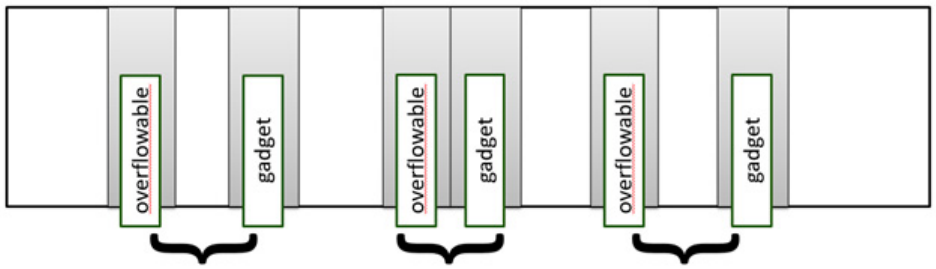
- Let's say we have discovered a buffer overflow in an object of size 0x4e0. We are able to overflow an arbitrary number of bytes with controlled data, starting from the overflowable buffer.
- Let's also say that we have discovered a useful gadget object, with size 0x450. This means this object will be in the same size class as the overflowable object.
- Let's also assume that we have discovered an allocation primitive which allows us to allocate at an arbitrary size
- We've also got the ability to free any of the allocations we make with the allocation primitive
- Finally, by performing an operation on the gadget object, we are able to determine if it has been overflowed or not.



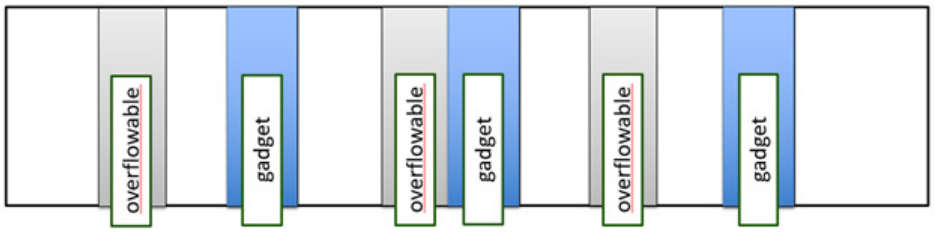
We need to cause the overflowable object to be allocated immediately before the gadget object, using the allocation and free primitives to shape the heap.

The assumptions we've made are good assumptions. Without a gadget object in the same size class as the overflowable object, it is almost impossible to exploit a buffer overflow on jemalloc. Without an arbitrary size allocation primitive, it is very difficult to exploit on dlmalloc.

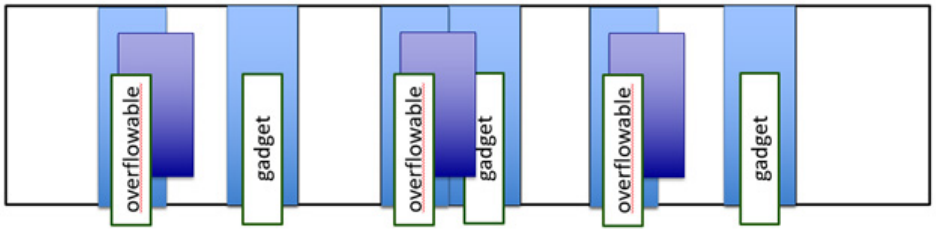
One technique for heap shaping that I've found particularly successful is called the placeholder technique.



The idea is simple. Using our arbitrary allocation primitive, we allocate a bunch of placeholder sets for our target objects, at the sizes of those objects. Each set of placeholders has a placeholder for the overflowable object and a placeholder for the gadget object. We hope that at least one set of placeholders is allocated such that the overflowable placeholder is directly before the gadget placeholder.



Next, we iterate over all of our placeholder sets, and for each set, we free the gadget placeholder and cause a gadget object to be allocated. Hopefully this gadget object will fall into the freed placeholder slot.



Then we once more iterate over each of our placeholder sets, this time freeing the overflowable placeholder, allocating our overflowable object and performing the overflow. We then activate the gadget for each instance of the gadget object, and use the result to determine if we have overflowed this instance or not. Hopefully at least one set of placeholders will have been turned into a working gadget.

This idea is the basis for the placeholder heap shaping technique. This technique can be used on both jemalloc and dlmalloc, but there are various things to look out for on each of the respective allocators.

One thing that can help increase the probability that one or more placeholder sets will have contiguous overflowable and gadget objects is to spray a large number of allocations of the relevant size class at the very beginning of the exploit. This will hopefully cause any best-fit chunks in the dlmalloc free lists or partially used runs in the relevant jemalloc bin to be used up, filling the holes. The placeholder allocations will then most probably be from new dlmalloc segments or new jemalloc runs, with successive allocations for the same size class falling after one another.

One problem you may encounter is that the allocation of the overflowable object, the gadget object or even the allocation primitives performs more than just the target allocation. In fact, it is very common for one or more of these to perform a bunch of smaller or bigger allocations around the actual target allocation we're interested in.

These unwanted allocations shouldn't make any difference when using jemalloc, as there we are concerned only with allocations in our bin.

On dlmalloc, these allocations might fall between the overflowable object and the gadget object, messing with our heap shaping.

```
for i in range(100):
    small_filter_allocs[i] = prim_alloc(0x100)

...

placeholder_sets = allocate_placeholders()

...
o = 0
for ps in placeholder_sets:
    prim_free(ps.gadget)
    for i in range(o, o + 10):
        prim_free(small_filter_allocs[i])
    o += 10
    allocate_gadget()
```

Assuming the gadget object is the one which allocates the unwanted allocations, one way to deal with this issue is to do the following:

- First, at the beginning of the exploit, we allocate a bunch of smaller blocks, let's say 100 blocks of 0x100 bytes each
- We then allocate all the placeholder sets
- Then, for each placeholder set, we first free the gadget placeholder.
- Then we free a few of the small filter allocations. These will be added to the free bins
- Then we allocate the gadget itself, the smaller unwanted allocations will use the filter allocations we just freed, and the gadget itself will fall on the placeholder.

On dlmalloc, freeing a placeholder can cause it to be consolidated with an adjacent free chunk, resulting in it being placed in a larger bin's free list. It then might not be used for the next allocation of the placeholder size. In other words, the placeholder will not be used to service the gadget or overflowable object allocation.

We can solve this quite easily using pinner allocations. The idea is that you simply allocate objects before your first placeholder and after your last placeholder. You never free these. As these pinners are always in use, and as we only ever free one placeholder in a set at a time, the placeholders will not be consolidated when freed.

Once you finally get your objects to be one after the other, there is an additional gotcha on dlmalloc. The metadata for the gadget chunk will be between your overflowable object and the gadget data. You need to overwrite this metadata with the sizes of the first and second objects respectively. Otherwise the allocator will fail when it comes time to free your chunks.

On jemalloc, the metadata is out-of-band, so this is not necessary.

Another thing to watch out for is that your primitive candidates may cause allocations or frees on threads other than the one which allocates your overflowable and gadget objects. In fact, the overflowable and gadget objects might not be allocated on the same thread at all. So your placeholders might be allocated (and freed) by one thread, but the gadget or overflowable object will be allocated on another.

This shouldn't present a problem for dlmalloc, but can be a significant pain on jemalloc.

The basic question is how can we move an allocation from one thread's tcache to another thread.

One way to do this is to use flush events.

We free our desired placeholder region, r15. We then free up to 20 more regions on that same thread. This will fill up the tcache, resulting in a flush. The flush removes the oldest half of the tcache and marks those regions as free, making them available for other threads to allocate. We then allocate on our desired thread to get the region we want with the object content we need.

Note that the desired region might not be first in line on the thread which allocates the desired object. You might need to spray a bunch of allocations on that thread in order to catch the freed placeholder. Filling holes before allocating the placeholders can help prevent this issue.

Getting this right can be really complicated and is going to be very specific to your target, your vulnerability and the gadgets and primitives you find.

One thing to remember is that on jemalloc, the regions for your overflowable and gadget objects will be of the same size – the maximum size of this particular bin. This might be larger than the objects themselves, so when you overflow from your overflowable buffer, you need to pad that overflow with the difference between the overflowable object's size and the bin size before you overflow any gadget object data.

On dlmalloc, the objects are directly after one another, except for the metadata between them.

On jemalloc it is possible that the threads that are involved in your exploitation are not assigned to the same arena. This can be problematic if, for example, you need to allocate contiguous regions in different threads.

Arena selection is supposed to use a round-robin approach, but in reality it will always allocate a new thread to the arena with the least threads assigned. If you can create and destroy threads somehow, before your overflow, you can do the following:

First add a whole bunch of threads to the system, say 30 threads. They will be spread evenly across the two arenas. Now destroy every second thread you created, causing one arena to have 15 threads and the other 0. Now allocate your exploitation threads. Up to 15 of them will be allocated in the same arena.

That's about all I've got for you guys. I hope that I've given you a foundation in understanding how the Android libc allocators work, and that you've heard some tips regarding how to successfully exploit heap vulnerabilities on Android.