Heap Rules

and their bug classes if they get violated

Do not read or write to a pointer returned by malloc[2] after that pointer has been passed back to free.
⋯→ Can lead to use after free vulnerabilities.

Do not use or leak uninitialized information in a heap allocation.[1]
⋯→ Can lead to information leaks or uninitialized data vulnerabilities.

Do not read or write bytes after the end of an allocation.
⋯→ Can lead to heap overflow and read beyond bounds vulnerabilities.

Do not pass a pointer that originated from malloc[2] to free more than once.
⋯→ Can lead to double free vulnerabilities.

Do not read or write bytes before the beginning of an allocation.
⋯→ Can lead to heap underflow vulnerabilities.

Do not pass a pointer that did not originate from malloc[2] to free.[3]
⋯→ Can lead to invalid free vulnerabilities.

Do not use a pointer returned by malloc[2] before checking if the function returned NULL.
⋯→ Can lead to null-dereference bugs and occasionally arbitrary write vulnerabilities.

1 Except for calloc, which explicitly initializes the allocation by zeroing it.
2 Or malloc-compatible functions including realloc, calloc, and memalign.
3 free(NULL) is allowed and not an invalid-free, but does nothing.

Heap Exploitation Part 1: Understanding the Glibc Heap Implementation

---

azeria-labs.com

Arm Heap Exploitation

Part 1: Understanding the Glibc Heap Implementation

In a previous article, I've discussed an old (but important) category of memory-corruption vulnerability called "stack buffer overflows", and how we, as attackers, can exploit these vulnerabilities to take control of a remote program and make it run our shellcode.

For applications that make use of an exploit mitigation called "stack canaries", it turns out that these stack-buffer-overflow vulnerabilities can be harder for attackers to exploit and often require additional vulnerabilities to exploit them reliably. When developers are using various stack-based exploit mitigations, attackers often instead build their exploits using heap-related vulnerabilities such as *use-after-frees, double-frees,* and *heap-overflows*. These heap-based vulnerabilities are more difficult to understand than their stack-based counterparts because attack techniques against heap-based vulnerabilities can be very dependent on how the internal implementation of the heap allocator actually works.

For this reason, before I write about exploiting heap-based vulnerabilities, I will use the first two parts of this series to talk about how the heap works. This first post will be an introduction into some high-level concepts, and a discussion about how new heap chunks are created. In the next post I will do a deeper dive into the technical implementation of how chunks are freed and recycled.

The way the heap works is very platform and implementation specific; lots of different heap implementations exist. For example, Google Chrome's *PartitionAlloc* is very different to the *jemalloc* heap allocator used in

FreeBSD. The default *glibc* heap implementation in *Linux* is also very different to how the heap works in *Windows*. So for this and the next few posts, I'll be focusing on the *glibc* heap allocator, i.e. how heap allocations work for C/C++ programs running on *Linux* devices by default. This heap is derived from the *ptmalloc* heap implementation, which is itself derived from the much older *dlmalloc* (Doug Lea malloc) memory allocator.

What is the heap, and why do people use it?

First things first: What is the heap, and what is it for?

The *heap* is used by C and C++ programmers to manually allocate new regions of process memory during program execution. Programmers ask the heap manager to allocate these regions of memory via calls to heap functions like *malloc*. These allocated regions of memory, or "allocations", can then be used, modified or referenced by the programmer up until the programmer no longer needs it and returns the allocation back to the heap manager via a call to *free*.

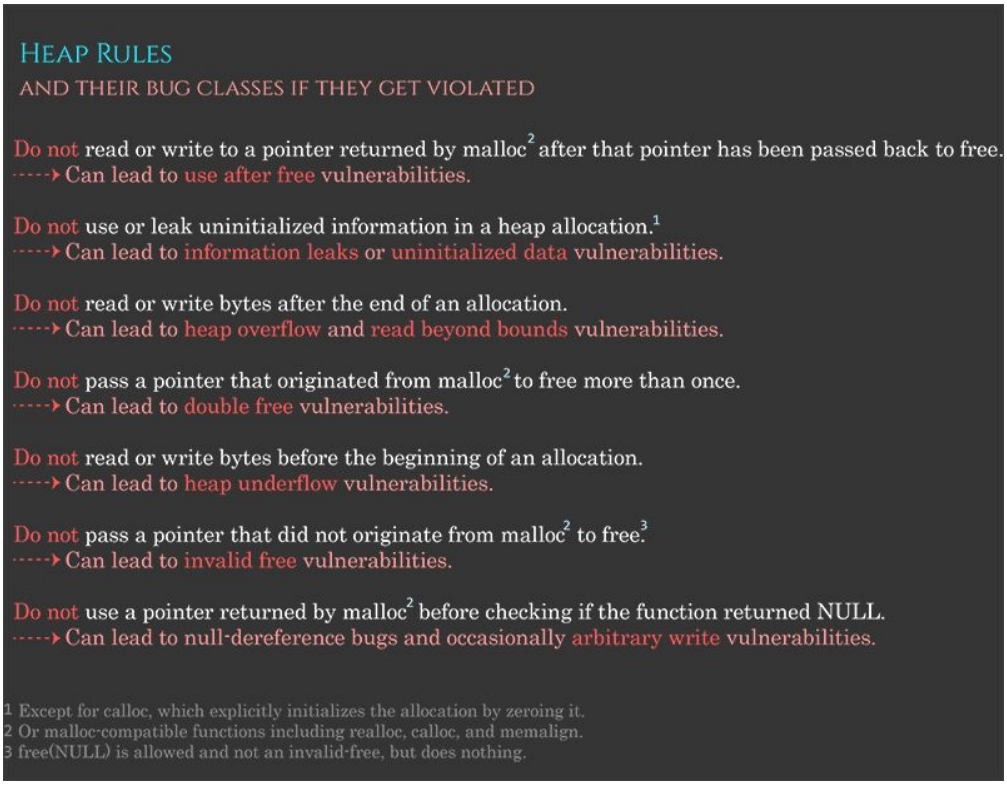
Here is an example of how a C program might allocate, use, and later free a structure on the heap:

```
typedef struct
{
   int field1;
   char* field2;
} SomeStruct;

int main()
{
   SomeStruct* myObject =
(SomeStruct*)malloc(sizeof(SomeStruct));
   if(myObject != NULL)
   {
       myObject->field1 = 1234;
       myObject->field2 = "Hello World!";
       do_stuff(myObject);
free(myObject);
   }
```

```
    return 0;
}
```

So long as the programmer follows a few simple rules, the heap manager ensures that each live allocation won't overlap any other. This feature makes the heap a very useful, highly used, and therefore very performance-sensitive feature of most C and C++ programs.

The following graphic lists some basic rules that programmers must follow when using the heap, alongside some of the vulnerability categories that occur if the programmer violates these rules. In later posts I will discuss all of these heap-related vulnerability classes in more detail, but for now I'll just show how the heap behaves when it is being used correctly.



Of course, *malloc* and *free* aren't the only way C and C++ programmers interact with the heap. C++ developers often instead allocate memory via the C++ operators *new* and *new[]*. These allocations must be released using the corresponding C++ operators *delete* and *delete[]* rather than using *free*. Programmers can also allocate memory via malloc-compatible heap

functions like *calloc*, *realloc* and *memalign*, which, like *malloc*, are eventually released via *free*.

For simplicity I'll initially just discuss *malloc* and *free*. Later we'll see that once we understand these two, most of the other heap functions become very easy to understand.

Here is an example of how a C++ program might allocate, use, and later free a structure on the heap:

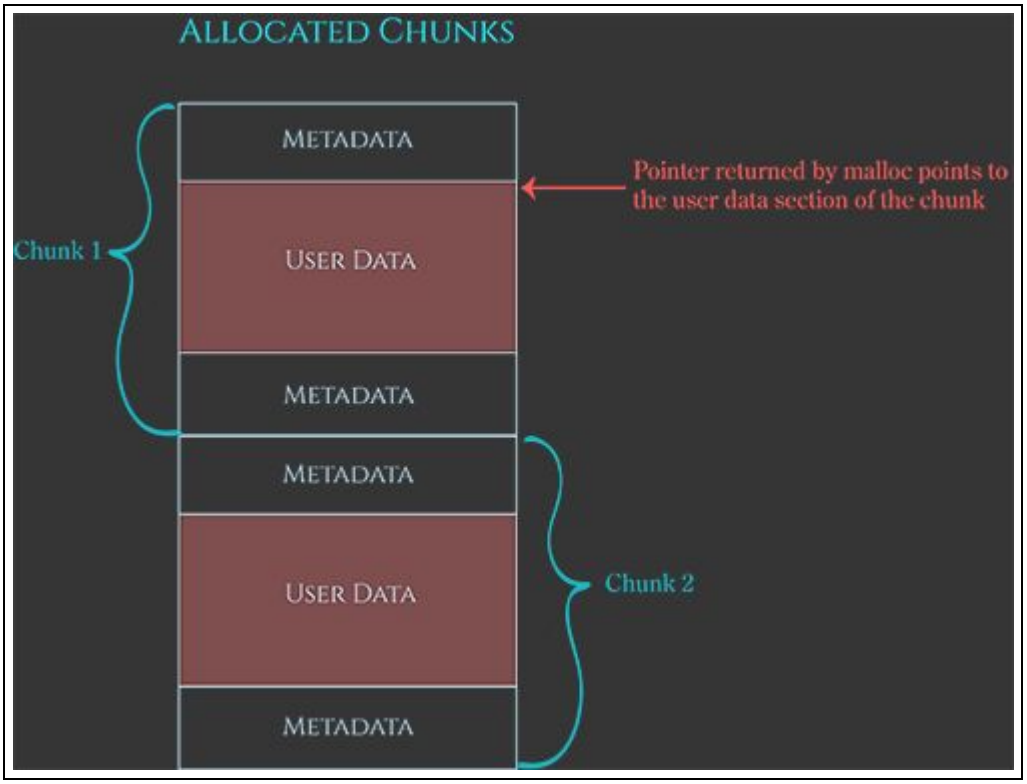
```
class SomeClass
{
public:
   int field1;
   char* field2;
};

int main()
{
   SomeClass* myObject = new SomeClass();
   myObject->field1 = 1234;
   myObject->field2 = "Hello World!";
   do_stuff(myObject);
   delete myObject;
   return 0;
}
```

Memory chunks and the chunk allocation strategies

Suppose a programmer asks for, say, 10 bytes of memory via *malloc*. To service this request, the heap manager needs to do more than just find a random 10 byte region that the programmer can write to. The heap manager also needs to store metadata about the allocation. This metadata is stored alongside the 10-byte region that the programmer can use.

The heap manager also needs to ensure that the allocation will be 8-byte aligned on 32-bit systems, or 16-byte aligned on 64-bit systems. Alignment of allocations doesn't matter if the programmer just wants to store some data like a text string or a byte array, but alignment can have a big impact

on the correctness and performance of programs if the programmer intends to use the allocation to store more complex data structures. Since *malloc* has no way to know what the programmer will store in their allocation, the heap manager must default to making sure all allocations are aligned.



This allocation metadata and alignment-padding bytes is stored alongside the region of memory that *malloc* will give back to the programmer. For this reason, the heap manager internally allocates "chunks" of memory that are slightly bigger than the programmer initially asked for. When the programmer asks for 10 bytes of memory, the heap manager finds or creates a new chunk of memory that is big enough to store the 10-byte space plus the metadata and alignment padding bytes. The heap manager then marks this chunk as "allocated", and returns a pointer to the aligned 10-byte "user data" region inside the chunk, which the programmer sees as the return value of the *malloc* call.

Chunk allocation: basic strategy

So how does the heap manager internally allocate these chunks?

First, let's look at the (heavily simplified) strategy for allocating small chunks of memory, which is the bulk of what the heap manager does. I'll explain each of these steps in a bit more detail below, and once we've done that we can look at the special case of huge allocations.

The simplified chunk-allocation strategy for small chunks is this:

1. If there is a previously-freed chunk of memory, and that chunk is big enough to service the request, the heap manager will use that freed chunk for the new allocation.
2. Otherwise, if there is available space at the top of the heap, the heap manager will allocate a new chunk out of that available space and use that.
3. Otherwise, the heap manager will ask the kernel to add new memory to the end of the heap, and then allocates a new chunk from this newly allocated space.
4. If all these strategies fail, the allocation can't be serviced, and *malloc* returns NULL.

Allocating from free'd chunks

Conceptually, allocating a previously-freed chunk is very simple. As memory gets passed back to *free*, the heap manager tracks these freed chunks in a series of different linked lists called "bins". When an allocation request is made, the heap manager searches those bins for a free chunk that's big enough to service the request. If it finds one, it can remove that chunk from the bin, mark it as "allocated", and then return a pointer to the "user data" region of that chunk to the programmer as the return value of *malloc*.

For performance reasons, there are several different types of bins, i.e. fast bins, the unsorted bin, small bins, large bins, and the per-thread tcache. I'll discuss these different types of bins in more detail in the next part of this series.

Allocating from the top of the heap

If there are no free chunks available that can service the allocation request, the heap manager must instead construct a new chunk from scratch. To do this, the heap manager first looks at the free space at the end of the heap (sometimes called the “top chunk” or “remainder chunk”) to see if there is enough space there. If there is, the heap manager manufactures a new chunk out of this free space.

Asking the kernel for more memory at the top of the heap

Once the free space at the top of the heap is used up, the heap manager will have to ask the kernel to add more memory to the end of the heap.

On the initial heap, the heap manager asks the kernel to allocate more memory at the end of the heap by calling *sbrk*. On most Linux-based systems this function internally uses a system call called "brk". This system call has a pretty confusing name–it originally meant "change the program break location", which is a complicated way of saying it adds more memory to the region just after where the program gets loaded into memory. Since this is where the heap manager creates the initial heap, the effect of this system call is to allocate more memory at the end of the program's initial heap.

Eventually, expanding the heap with *sbrk* will fail–the heap will eventually grow so large that expanding it further would cause it to collide with other things in the process' address space, such as memory mappings, shared libraries, or a thread's stack region. Once the heap reaches this point, the heap manager will resort to attaching new non-contiguous memory to the initial program heap using calls to *mmap*.

If *mmap* also fails, then the process simply can't allocate any more memory, and *malloc* returns NULL.

Off-heap allocations via MMAP

Very large allocation requests* get special treatment in the heap manager. These large chunks are allocated off-heap using a direct call to *mmap*, and this fact is marked using a flag in the chunk metadata. When these huge allocations are later returned to the heap manager via a call to *free*, the heap manager releases the entire *mmap*ed region back to the system via *munmap*.

*By default this threshold is 128KB up to 512KB on 32-bit systems and 32MB on 64-bit systems, however this threshold can also dynamically increase if the heap manager detects that these large allocations are being used transiently.

Arenas

On multithreaded applications, the heap manager needs to defend internal heap data structures from race conditions that could cause the program to crash. Prior to *ptmalloc2*, the heap manager did this by simply using a global mutex before every heap operation to ensure that only one thread could interact with the heap at any given time.

Although this strategy works, the heap allocator is so high-usage and performance sensitive that this led to significant performance problems on applications that use lots of threads. In response to this, the *ptmalloc2* heap allocator introduced the concept of "arenas". Each arena is essentially an entirely different heap that manages its own chunk allocation and free bins completely separately. Each arena still serializes access to its own internal data structures with a mutex, but threads can safely perform heap operations without stalling each other so long as they are interacting with different arenas.

The initial ("main") arena of the program only contains the heap we've already seen, and for single-threaded applications this is the only arena that the heap manager will ever use. However, as new threads join the process, the heap manager allocates and attaches secondary arenas to each new thread in an attempt to reduce the chance that the thread will have to wait around waiting for other thread when it attempts to perform heap operations like *malloc* and *free*.

With each new thread that joins the process, the heap manager tries to find an arena that no other thread is using and attaches the arena to that thread. Once all available arenas are in use by other threads, the heap manager creates a new one, up to the maximum number of arenas of 2x *cpu-cores* in 32-bit processes and 8x *cpu-cores* in 64-bit processes. Once that limit is finally reached, the heap manager gives up and multiple threads will have to share an arena and run the risk that performing heap operations will require one of those threads to wait for the other.

But wait a minute! How do these secondary arenas even work? Before we saw that the main heap is located just after where the program is loaded into

memory and is expanded using the *brk* system call, but this can't also be true for secondary arenas!

The answer is that these secondary arenas emulate the behavior of the main heap using one or more "subheaps" created using *mmap* and *mprotect*.

Subheaps

Sub-heaps work mostly the same way as the initial program heap, with two main differences. Recall that the initial heap is located immediately after where the program is loaded into memory, and is dynamically expanded by *sbrk*. By contrast, each subheap is positioned into memory using *mmap*, and the heap manager manually emulates growing the subheap using *mprotect*.

When the heap manager wants to create a subheap, it first asks the kernel to reserve a region of memory that the subheap can grow into by calling *mmap**. Reserving this region does not directly allocate memory into the subheap; it just asks the kernel to refrain from allocating things like thread stacks, mmap regions and other allocations inside this region.

*By default, the maximum size of a subheap–and therefore the region of memory reserved for the subheap to grow into–is 1MB on 32-bit processes and 64MB on 64-bit systems.

This is done by asking *mmap* for pages that are marked *PROT_NONE*, which acts as a hint to the kernel that it only needs to reserve the address range for the region; it doesn't yet need the kernel to attach memory to it.

Where the initial heap grew using *sbrk*, the heap manager emulates "growing" the subheap into this reserved address range by manually invoking *mprotect* to change pages in the region from PROT_NONE to PROT_READ | PROT_WRITE. This causes the kernel to attach physical memory to those addresses, in effect causing the subheap to slowly grow until the whole *mmap* region is full. Once the entire subheap is exhausted, the arena just allocates another subheap. This allows the secondary arenas to keep growing almost indefinitely, only eventually failing when the kernel runs out of memory, or when the process runs out of address space.

To recap: the initial ("main") arena contains only the main heap which lives just after the where the program binary is loaded into memory, and is expanded using *sbrk*. This is the only arena that is used for single-threaded applications. On multithreaded applications, new threads are given secondary arenas from which to allocate. Using arenas speeds up the program by reducing the likelihood that a thread will need to wait on a mutex before being able to perform a heap operation. Unlike the main arena, these secondary arenas allocate chunks from one or more *subheaps*, whose location in memory is first established using *mmap*, and which grow by using *mprotect*.

Chunk metadata

Now we finally know all of the different ways that a chunk might get allocated, and that chunks contain not only the "user data" area that will be given to the programmer as the return value of *malloc*, but also metadata. But what does this chunk metadata actually record, and where does it live?

The exact layout of the chunk metadata in memory can be a bit confusing, because the heap manager source code combines metadata at the end of one chunk with the metadata at the start of the next, and several of the metadata fields exist or are used depending on various characteristics of the chunk.

For now, we'll just look at live allocations, which have a single *size_t** header that is positioned just behind the "user data" region given to the programmer. This field, which the source code calls *mchunk_size*, is written to during *malloc*, and later used by *free* to decide how to handle the release of the allocation.

*A size_t value is an 4 byte integer on a 32-bit system and an 8 byte integer on on a 64-bit system.

The *mchunk_size* stores four pieces of information: the chunk size, and three bits called "A", "M", and "P". These can all be stored in the same *size_t* field because chunk sizes are always 8-byte aligned (or 16-byte aligned on 64-bit), and therefore the low three bits of the chunk size are always zero.

The “A” flag is used to tell the heap manager if the chunk belongs to secondary arena, as opposed to the main arena. During free, the heap manager is only given a pointer to the allocation that the programmer wants to free, and the heap manager needs to work out which arena the pointer belongs to. If the A flag is set in the chunk’s metadata, the heap manager must search each arena and see if the pointer lives within any of that arena’s subheaps. If the flag is not set, the heap manager can short-circuit the search because it knows the chunk came from the initial arena.

The “M” flag is used to indicate that the chunk is a huge allocation that was allocated off-heap via *mmap*. When this allocation is eventually passed back to *free*, the heap manager will return the whole chunk back to the operating system immediately via *munmap* rather than attempting to recycle it. For this reason, freed chunks never have this flag set.

The “P” flag is confusing because it really belongs to the *previous* chunk. It indicates that the previous chunk is a *free* chunk. This means when *this* chunk is freed, it can be safely joined onto the previous chunk to create a much larger free chunk.

In my next post, I’ll talk in more detail about how these free chunks are “coalesced” together, and I’ll discuss how chunks are allocated and recycled using different types of “bins”. After that we’ll look at some different categories of heap vulnerabilities, and how they can be exploited by attackers to remotely take control of vulnerable programs.

There are a couple of nice phrack articles on heaps I would like to share with you in case you want to read more before I publish the next parts.

- Once upon a free()
- Malloc des-maleficarum
- The house of lore
- Advanced Doug Lea’s malloc exploits
- Yet another free() exploitation technique

Looking for a comprehensive training on exploit development?

Sign up for my upcoming Black Hat USA training in Las Vegas, where I will be teaching a 2-day course on "Arm-based IoT Exploit Development" (3-day training at Infiltrate is Sold Out). If your company is interested in private trainings and wants to save around 50% compared to conference trainings, send a message to azerialabs@gmail [.] com.

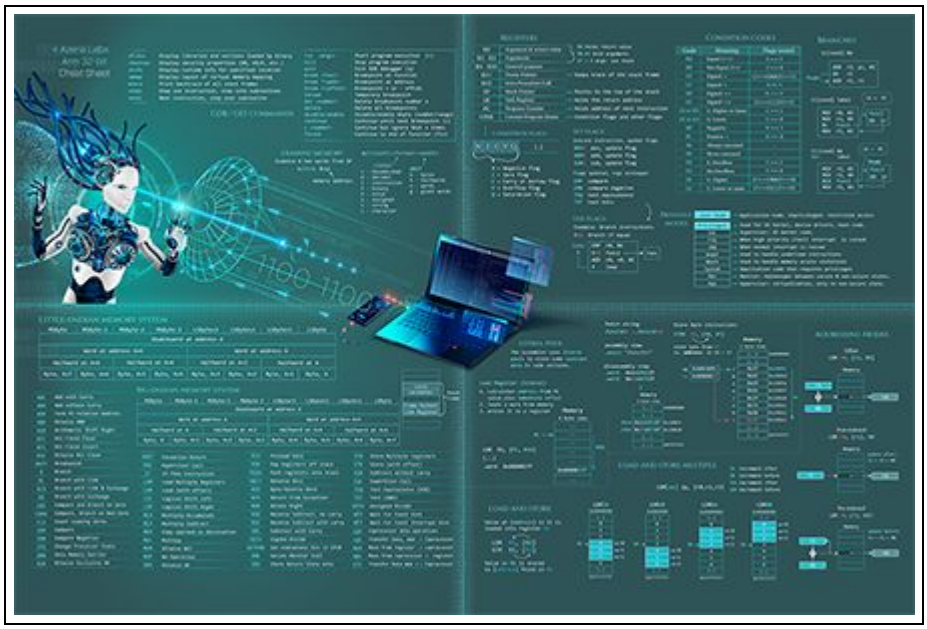
ARM Exploit Development

- Writing ARM Shellcode
- TCP Bind Shell (ARM 32-bit)
- TCP Reverse Shell (ARM 32-bit)
- Process Memory and Memory Corruption
- Stack Overflows (Arm32)
- Return Oriented Programming (Arm32)
- Stack Overflow Challenges
- Process Continuation Shellcode
- Introduction to Glibc Heap (free, bins)
- Heap Exploit Development (Part 1)
- Heap Overflows and iOS Kernel (Part 2)
- Grooming the iOS Kernel Heap (Part 3)

Twitter: @Fox0x01 and @azeria_labs

New ARM Assembly Cheat Sheet

Poster Digital



Source: https://azeria-labs.com/heap-exploitation-part-1-understanding-the-glibc-heap-implementation/