Understanding the heap by breaking it

A case study of the heap as a persistent data structure through non-traditional exploitation techniques

Abstract:

Traditional exploitation techniques of overwriting heap metadata has been discussed ad-nauseum, however due to this common perspective the flexibility in abuse of the heap is commonly overlooked. This paper examines a flaw that was found in several popular implementations of the GSS-API as a method for elaborating upon the true beauty of data structure exploitation. This paper focuses on the dynamic memory management implementation provided by the GNU C library, particularly ptmalloc2 and presents methods for evading certain sanity checks in the library along with previously unpublished methods for obtaining control.

Outline:

0. The heap, what is it?

  0.1 How the GNU C library implements it
  0.2 Heap data structures
  0.3 Implementation of heap operations
  0.4 Putting it all together

1. Double free()'s

  1.1 What is a double free()
  1.2 Traditional double free() exploitation
  1.3 Oops, it's not 1996 anymore or why that technique doesn't work anymore

2. The example

  2.1 Code overview
  2.2 Vulnerability 0 – array of pointers double free
  2.3 Vulnerability 1 – double free of user-data
  2.4 Goals – Write-what-where

3. The effects of a multi-threaded environment

  3.1 thread safety in GNU libc's allocator

DRAFT

Justin N. Ferguson
IOActive

3.2 what mutual exclusions don't provide
3.3 GNU libc's double free() protection
3.4 Abusing the system with this knowledge

4. Six million ways

4.1 Exploitation method 0: triple free of vulnerability 1 with fastbin's (not exploitable in this instance – previously unpublished method)
4.2 Exploitation method 1: double free of vulnerability 1 where thread X invalidates thread Y's heap reference (exploitable)
4.3 Expansion on method 1, setuid()/capabilities(7), threads and the heap (using an unpriv'd thread to screw a priv'd one [linuxthreads specific])
4.4 Exploitation method 2: ptr = (pointer+offset) = pointer ??, double free of vulnerability 0 where multiple pointers point to the same place (should be exploitable)
4.5 Exploitation method 3: double free of vulnerability 0 where the backwards link is overwritten (exploitable??)
4.6 I'm drawing a blank, but I'm sure there are more methods I came up with

5. Extra slides in case I run short
(esoteric stuff in case I run short in the presentation)

5.1 __dso_handle abuse
5.2 ???

Content

0.0 The heap, what is it?

The heap is a global data structure that provides dynamically allocated memory storage that provides an 'exists until free' scope. It provides a compliment to the stack in that it allows an application to allocate space for variables at run time that can exist outside the scope of the currently executing function.

0.1 How the GNU C library implements it

The GNU libc library provides an implementation of dynamic memory allocation via malloc()/free()/realloc()/et cetera as specified the the ISO/IEC C99 standard. This implementation is provided by ptmalloc2 which was written by Wolfram Gloger and is based on dlmalloc which was written by Doug Lea.

For those familiar with dlmalloc the only significant differences are that multiple arenas/heaps are provided and multi-threaded applications are supported in a 'mostly safe' manner. For those unfamiliar with dlmalloc/ptmalloc2 the heap is

DRAFT

somewhat of a misnomer in that an application can have multiple heap, but to simplify a heap is a section of linear memory that is either memory mapped or obtained via sbrk(). Chunks are allocated either from the chunk of memory known as the 'top chunk' (or sometimes referred to as the wilderness chunk) and it is treated in a special manner in that it is the only chunk of memory that can grow by requesting more memory from the system. New chunks are initially allocated from this top chunk, but as memory use progresses chunks are usually obtained by searching through a series of lists containing various chunks of free memory, there are multiple variations on this list, some a doubly linked and sorted whereas others contain only single links and are all the same size. These free lists are referred to as bin's within the implementation and thus this terminology is continued throughout this paper. In some circumstances, where no chunk fits an allocation request then it is possible to split an already existing chunk, this leaves a smaller sub-chunk that is referred to as the last remainder chunk, and is generally also treated special.

Allocated chunks in the same arena are maintained in the same linear space and are navigated by size, meaning to find the next chunk of allocated memory the pointer to the current chunk plus its size are summed together and then aligned to find the next chunk. The allocated chunks of memory have the characteristic that they may not directly border another allocated chunk, which implies that all allocated blocks of memory either border the top chunk or a free chunk.

Conceptually the heap can be visualized as one or more sections of linear memory that contain any number of free and allocated blocks of memory interspersed with each other. This provides an interesting possibility as a write to a heap under invalid circumstances can overwrite metadata in a deallocated chunk of memory, of if the write occurs to a dangling pointer, to an allocated chunk of memory. An arena/heap is maintained via a series of data structures which are described below in the next section.

0.2 Heap data structures

0.2.0 heap and arena data structures

The glibc heap is implemented via the following data structures, in order of their appearance on a heap. The first is the heap_info

```
typedef struct _heap_info {
        mstate                  ar_ptr; /* Arena for this heap. */
        struct _heap_info *     prev;   /* Previous heap. */
        size_t                  size;   /* Current size in bytes. */
        char                    pad[-5 * SIZE_SZ & MALLOC_ALIGN_MASK];
} heap_info;
```

mstate ar_ptr: a pointer to the heaps arena, this implies a one-to-

DRAFT

one relationship between heaps and arenas (jf: dc)

struct _heap_info *ptr	a pointer to the previous heap_info structure, the heap_info structures are maintained in a circular singular linked list (jf: dc)
size_t size	size of the current heap
char pad[…]	used for padding to ensure proper alignment

The heap_info structure is the first structure in a given heap, by implication you would think it to be an incredibly important structure however when reviewing operations it seems almost superfluous. The structure defines the most basic information necessary for heap operations. Most importantly it specifies the size of the heap and provides a pointer to the arena data structure.

The next structure that is more frequently used is the malloc_state structure, or mstate.

```
struct malloc_state {
        mutex_t                 mutex;  /* Serialize access.  */
        int                     flags;  /* Flags (formerly in max_fast).  */

#if THREAD_STATS
        /* Statistics for locking.  Only used if THREAD_STATS is defined.  */
        long                    stat_lock_direct, stat_lock_loop, stat_lock_wait;
#endif

        mfastbinptr             fastbins[NFASTBINS]; /* Fastbins */
        mchunkptr               top;
        mchunkptr               last_remainder;
        mchunkptr               bins[NBINS * 2];
        unsigned int            binmap[BINMAPSIZE];     /* Bitmap of bins */
        struct malloc_state *next;                      /* Linked list */
        INTERNAL_SIZE_T         system_mem;
        INTERNAL_SIZE_T         max_system_mem;
};
```

mutex_t mutex:	Used to ensure synchronized access to the various data structures employed by the ptmalloc implementation, it is normally obtained prior to calling the _int_XXX() function, for instance when you call free(), you're actually calling public_free() (jf: fix caps), which among other things locks the mutex and calls _int_free()

DRAFT

int flags:	Used to represent the various characteristics of the current arena, for instance if there are fastbin chunks or not, if memory is non-contiguous, et cetera.
long stat_lock_direct: long stat_lock_loop: long stat_lock_wait:	Used to provide various locking statistics.
mfastbinptr fastbins[…]:	This array is the array of fastbin's which is used as a bin for housing chunks that allocated and free()'d, their operations are quicker in large part due to less operations being performed on them. An in-depth look fastbin's are discussed below/later.
mchunkptr top:	The top is a special chunk of memory, that borders the end of available memory, is not in any bin, represents the total unallocated space for a given arena and from subsystem initialization always exists. Memory allocation semantics are discussed later, along with the description of the malloc_chunk/mchunkptr data structure
mchunkptr last_remainder:	Used when a small request for a chunk of memory that does not fit exactly into any given chunk of memory. It is the remainder of the space left when a chunk is split to accommodate a small allocation request.
mchunkptr bins[…]:	The bins array is similar to the fastbins array in that it operates as a list of free chunks of memory, however it is used for larger 'normal' chunks; bins are described in detail below.
unsigned int binmap[…]:	Used as a one-level index to help speed up the process of determining if a given bin is definitely empty. This helps speed up traversals by allowing the allocator to skip over confirmed empty bins.

DRAFT

struct malloc_state *next: Points to the next area, circular singly linked list (jf)

INTERNAL_SIZE_T system_mem:
INTERNAL_SIZE_T max_system_mem: Used to track the amount of memory currently allocated by the system; the INTERNAL_SIZE_T macro is defined as size_t on most platforms.

Now that we have some concept of the higher-layer abstractions, we can begin to delve into the next subject, which in turn we will describe the mchunkptr, binning and the representation of allocated and free chunks of memory. There is really is no good documentation that I could find which covers the layout of these structures, the closest being the two 'original' documents that were published in Phrack 59 (jf: references) and of course the ptmalloc2 source code itself.

0.2.1 Chunks of memory

The chunks that are employed by ptmalloc2 are of the same physical structure regardless of whether they are a fast-chunk or a normal chunk and regardless of whether they are allocated or not, however their representation is different depending on the state of the chunk in question (this is a key point to understanding double free() exploitation), think of it being something akin to a union. A chunk of memory is represented by the following structure. The malloc_chunk/mchunkptr structure is as follow:

```
struct malloc_chunk {

        INTERNAL_SIZE_T         prev_size;
        INTERNAL_SIZE_T         size;
        struct malloc_chunk *   fd;
        struct malloc_chunk *   bk;
}
```

INTERNAL_SIZE_T prev_size: The size of the chunk previous to the current chunk; only used by the implementation if the previous chunk is free.

INTERNAL_SIZE_T size: The size of the current chunk, used to traverse allocated chunks.

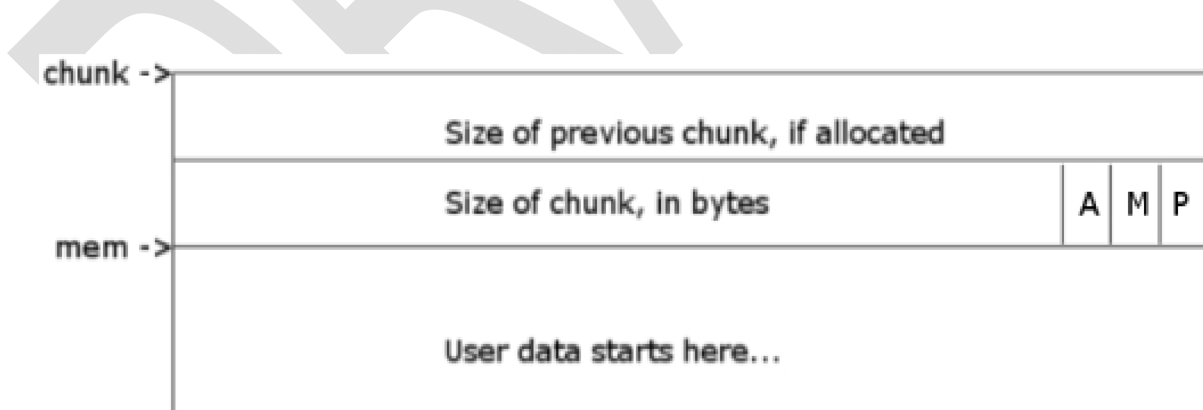
DRAFT

struct malloc_chunk *fd: Pointer to the next chunk in the circular doubly linked free list, if the chunk is currently free.

struct malloc_chunk *bk: Pointer to the previous chunk in the circular doubly linked free list, if the chunk is currently free.

The structure above provides an accurate depiction of a given chunk of memory regardless of its state (only what members are used differs). Even more

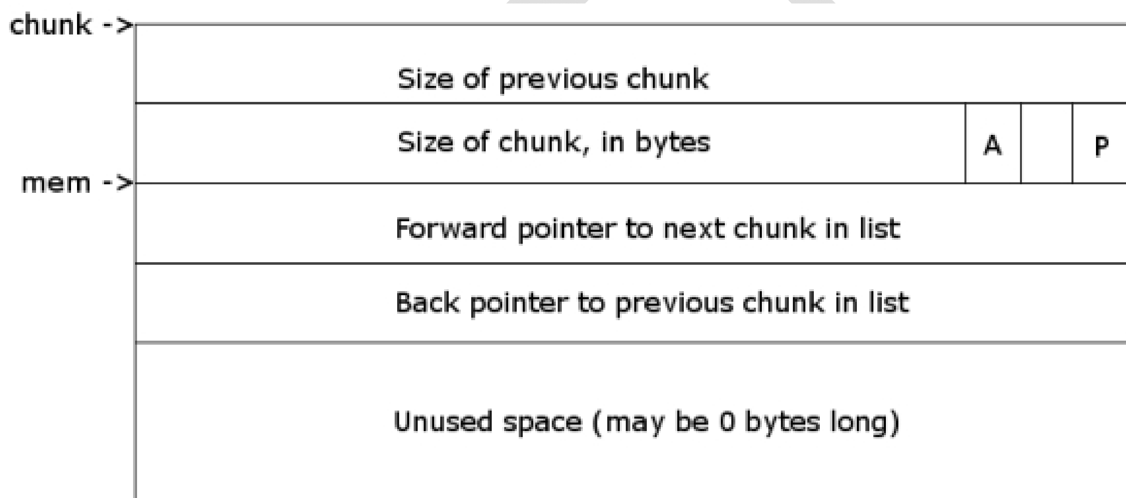
The structure above provides an accurate depiction of a given chunk of memory regardless of its state (only what members are used differs). Even more this subtley implies that regardless of how much memory is requested to be allocated, that there will be extra bytes allocated for metadata, in this case two size_t's and two pointers to struct malloc_chunk, giving us a possibility of 16, 24, or 32 bytes of overhead per chunk. That is to say that we either have four or eight byte size_t's and four or eight byte pointers. Then we have alignment issues, namely that malloc()'d chunks must fall on a boundary of a power of two at least as large as 2 times the sizeof(INTERNAL_SIZE_T), which is suitable for every architecture except for PowerPC32, which is enough of an oddity that it is not addressed here. For the time being, we will assume four byte pointers and size_t's, with a minimum size of sixteen bytes and an alignment of two times the sizeof(size_t). However, despite having the same physical structure, the interpretation of an given chunk changes depending on its state, so an allocate chunk is represented as follows:



The pointer here entitled 'chunk' represents the beginning of a given chunk as represented internally to the various ptmalloc routines, at that pointer you will find the size of the previous chunk if allocated (jf?), followed by the size of the current chunk. It is important to note that because of alignment we can guarantee the lower 3-bits (jf: dc) will always be zero, and therefore they're used as metadata to determine if the current chunk is in a non-main arena, was

allocate via mmap(), and if the previous chunk is in use (A, M and P respectively). Now, finally, we reach the 'mem' pointer, this is the pointer to the beginning of memory returned externally to the user of the API; the pointer returned when you call malloc(). It extends to the end of the allocated memory and then following that we will encounter the next chunk of memory. This introduces the methology employed for traversing allocated blocks of memory. That is to say, they are traversed by size instead of directly through pointers like in the free list.

Now that allocated blocks have been examined, the following represents a chunk of memory that was at one point allocated but has been free()'d. As previously stated, it uses exactly the same data structure, however it's representation is different in use, it is as follows:



Here, the pointer 'chunk' points to the beginning of the block of memory as represented internally to the implementation, the size of the previous chunk is contained there followed by the size of the current chunk, with the third least significant bit to determine if the chink is in a non-main arena, and the least significant bit indicating whether the previous chunk is in use or not. The reader may note that the bit determining if the block was memory mapped or not is not employed in this representation, and that is simply because there are no lists with mmap()'d chunks in them, they are simply unmapped when free()'d. Finally following the size members you have the pointer to the next free chunk in the list and a pointer to the previous free chunk in the list. These are pointed to by the 'mem' pointer in the diagram and it should be noted that this area is where user (of the API) data was previously stored, or rather the linked list pointers exist beginning at the address that was previously returned by malloc() or realloc().

Physically following this chunk, we will either have an allocated chunk of memory or the top chunk; this implies that no two free blocks of memory will border each other because free chunks that border each other will be coalesced into a single larger block. Finally, unlike allocated blocks of memory, free chunks are

traversed via circular linked lists, whereas allocated blocks were traversed by determining their size and doing pointer arithmetic. Furthermore, in an allocated block the pointer returned to the user of the API starts eight bytes after the beginning of the chunk, which in a free block is the start of the metadata used to traverse the linked lists.

0.2.2 Binning

Memory, once it has been free()'d is stored in linked lists called bin's, they are sorted by size to allow for the quickest access of finding a given chunk for retrieval, that is to say that when you free() memory, it doesn't actually get returned to the operating system, but rather gets potentially defragmented and coalesced and stored in a linked list in a bin to be retrieved for an allocation later.

The bin's, if you recall, are arrays of pointers to linked lists. There are essentially two types of bin, a fastbin and a 'normal' bin. Chucks of memory that are considered for use in fastbin's are small (default maximum size is sixty bytes with a configurable maximum of eighty), they are not coalesced with surrounding chunks on free(), they are not sorted and they only have singular linked lists, instead of doubly linked lists. The data structure of the block is still the same as with 'normal' blocks, only their representation and use differs. Furthermore, they are removed in a last in first out (LIFO) manner as opposed to the traditional first in first out (FIFO) method. Because the chunks are not consolidated their access is dramatically quicker than that of a normal chunk, essentially fastbin's trade speed for fragmentation.

There are ten fastbins although this number may vary dependant on your platform, but as we decided earlier to employ a four byte size_ts and pointers, we will have ten fastbins holding chunks ranging from zero (or MINSIZE after metadata) to eighty bytes. In the below diagram, the fastbin number is the index into the array of fastbin's, the 'holds chunk sizes' indicates the range of sizes that the bin will hold, and 'real chunk size' is the actual size (after metadata and alignment) of the block being free()'d.

DRAFT

fastbin #	holds chunk sizes	real chunk size
0	00 - 12	16
1	13 - 20	24
2	21 - 28	32
3	29 - 36	40
4	37 - 44	48
5	45 - 52	56
6	53 - 60	64
7	61 - 68	72
8	69 - 76	80
9	77 - 80	88

Normal sized chunks are split into three categories, the first bin (index 1) is the unsorted bin, it conaints all size chunks that were recently free()'d and are not placed into bin's until malloc()/realloc() has had a chance to take the chunk. Afterwards, they are placed in one of the two other types of bin, small or large. A chunk is considered small if their size is less than 512 bytes. Small chunk bins are not sorted as all chunks in a given bin are of the same size, it may seem redundant to have 'normal' chunks of sizes that would fall into the fastbin ranges, however due to fast-chunks being consolidated under certain circumstances (jf: dc), it is indeed possible to have multiple fast-chunks consolidated into a size that would fit into a small bin. Large chunk's are anything about 512 bytes and less than 128 kilobytes (chunks larger than 128k are serviced directly by mmap()). The large bin's contain chunks sorted by size in the smallest descending order, and are allocated in a least-recently used (FIFO) order. There are 128 bins in all, spaced approximately logarithmic proportions. The spacing between bins is listed below, but the specifics are not detailed in the same fashion as the fastbin's simpy due to the size of the resulting chart:

```
64 bins of spacing size       8
32 bins of spacing size      64
16 bins of spacing size     512
 8 bins of spacing size    4096
 4 bins of spacing size   32768
 2 bins of spacing size  262144
 1 bin  of spacing size what's left
```

Finally, there are two chunks that will never be placed in a bin, and those chunks are the top chunk and the last_reaminder chunk, which are described in detail in the next section.

0.2.3 Top chunk

The top chunk is the chunk that borders the end of available memory, it is used when there are no adequate chunks in the bin's, where there is no chunks that can be consolidated to fit the request, when the last_remainder chunk won't fit and so on. It is used as a last resort to provide memory to allocation requests. The top chunk can grow (i.e. when an allocated block of memory is being free()'d border the top chunk then it is consolidated into it), and it can shrink (when used for allocations). In short, the top chunk is just like any other chunk except it has a specified place in memory, it always has its previous in use flag set and generally the code always treats it as if it always exists. For more information about the top chunk/wilderness chunk, please see the still useful albeit somewhat dated paper 'Exploiting the Wilderness Chunk' by Phantasmal Phantasmagoria (jf: ref)

0.2.4 last_remainder

The last remainder block is another special case, it like the top chunk will never be found directly in a bin, however it can be handed out and then eventually find itself back in a bin, just never at the same time that it is the last_remainder chunk. The last_remainder chunk is the result of allocation requests for small chunks that have no exact fit in any of the bins. If the request is larger than the last_remainder chunk but smaller than a block in a bin, then the chunk is split again. The last_remainder chunk is the result of having to split a larger chunk into two, one part of it is handed out from the allocation, and the other because the last_remainder chunk.

Where the last_remainder chunk comes into play for us is if while attempting to massage the heap into a specific state, one of our blocks of memory is split, aside from this the block is not terribly important for our purposes.

0.3 Implementation of heap operations

Any heap implementation has a few basic operations that need to be able to be performed, an operation to create a heap, to resize a heap, to remove a heap. An operation to allocate a chunk of memory, delete a chunk of memory and resize a chunk of memory. In this section we will review the process of doing all of these operations in a high-level fashion.

DRAFT

0.3.1 Heap initialization

Heap initialization occurs the first time a request to allocate memory is called, the interface to create heaps, first one or otherwise is not publicly exported and thus cannot be explicitly accomplished, but rather implicitly by attempting to allocate memory. This step is almost always realized prior to the developers first call to malloc() or realloc() due to process/application initialization done by the step (by the time you call one of the allocation functions its already been called several times by libc code run during your process' creation).

None the less however, upon entry into public_mALLOc() (the mangled name of the public interface as exported in glibc (jf: dc)). The first thing done is a function pointer is assigned to the global variable __malloc_hook, which is in turn a function pointer that is globally initialized to the function malloc_hook_ini().

In malloc_hook_ini(), the variable __malloc_hook() has its value changed so that it points to NULL and ptmalloc_init() is called. Inside of ptmalloc_init(), a global variable __malloc_initialized has its value checked to determine if initialization has already taken place or not and then is set to zero indicating initialization is in progress, then the function ptmalloc_init_minimal() is called which initializes some global configuration values to their defaults, next __pthread_initialize() is potentially called to initialize the POSIX threads interface, followed by having the rest of the debugging callback variables initialized to their default values. The mutex for the main_arena is initialized and its pointer to the next arena is initialized to point to itself (circular singly linked list). Next a thread specific key is created and the arena_key is tied to the main_arena and an thread_atfork() handler is created to deal with mutex's in children processes in certain conditions. Finally, the supported configuration options via environment variables (i.e. MALLOC_MMAP_THRESHOLD_, MALLOC_MMAP_MAX_, MALLOC_CHECK_, et cetera). Finally __malloc_initialized is set to one, ptmalloc_init() is returned and malloc_hook_ini() returns by calling public_mALLOc() again with the size argument that was passed to it.

Upon returning to public_mALLOc() an initial heap has still not been allocated which brings us to the next section, creating a heap,

0.3.2 Create a heap

The creation of a heap is not controlled in a manner familiar to Microsoft Windows developers (ala HeapCreate()), but rather implicitly by calling an allocation function during certain conditions. The first condition is that no heap exists and creation takes place immediately following subsystem initialization as described in the previous section.

When the first allocation occurs, after initialization a call to arena_get() is made, which arena_get() is actually a macro that first retrieves a pointer to the last arena locked by the current thread by retrieving the thread specific data (TSD) for the arena_key, if the returned pointer is not equal to NULL, then it will attempt

DRAFT

to lock the mutex by calling mutex_trylock(), which will be defined to be an alias to pthread_mutex_trylock(). In this instance the returned pointer should be NULL (jf: dc), but supposing it wasn't then interesting behavior occurs if it cannot obtain the lock, it will call arena_get2(), which under specific circumstances will create a new arena.

Because this is a first time call to one of the allocation routines, we will end up in arena_get2(), which will first check the pointer passed in for the arena, if it is NULL it will return a pointer to the main_arena variable. Otherwise it will attempt to lock all of the mutexes in the global circularly linked list of arena's. If that should fail then we attempt to lock the list_lock mutex. If that fails then a blocking call to mutex_lock() is made and once it is obtained the walk of the global circularly linked list of arena's is walked again attempting to lock their mutexes.

Supposing even that failed, then while we're still holding the list_lock mutex, a call to _int_new_arena() is made. Once inside of _int_new_arena() a call to new_heap() is made, attempting to allocate the size of the allocation plus the size of the heap_info structure plus the size of the malloc_state structure plus the size of MALLOC_ALIGNMENT. Inside of new_heap() there is a sanity checks, if the size plus the size of padding for the top chunk is less than the minimum heap size, if the size plus the size of padding to the top chunk is less than or equal to the maximum heap size, if the two summed are larger than the maximum heap size and so on. Following the checks, several calls to mmap() will be potentially be made in an attempt to get a properly aligned chunk of memory. Assuming one of these succeeds then mprotect() is called making the section read and write only, At which point the heap pointer will be assigned to the new section of memory, and the heap_info structures member size will be initialized to the heap size and the pointer will be returned. If failure occurred in the process, NULL is returned.

If the previous heap allocation failed, then another attempt is made allocating the minimum size, which if it fails then the entire venture is considered a wash and NULL is returned. Assuming one of the calls to new_heap() succeeded, then the arena pointer is initialized and a call to malloc_init_state() is made which initializes the circular linked lists for normal size bins, initializes the top chunk and so on. Finally, upon returning back into _int_new_arena(), various size elements are incremented or set and the alignment of the top chunk is checked and fixed if necessary and concluding in returning a pointer to the newly created arena.

Next the arena_key for the thread is set for the new arena, the arena's mutex is initialized and locked and the new arena is put on the global linked list of arena's. A call to atomic_write_barrier() is made, however on IA-32 architecture it is defined to a blank __asm__() call. The main_arena has its next element assigned to the new arena and a pointer to the new area is returned. This pointer is returned back into public_mALLOc() and we have finally created and initialized a new heap and arena, leading us to our next subsection, allocating a chunk of memory.

DRAFT

0.3.3 Allocate a block

After obtaining an arena pointer via arena_get(), assuming it succeeded then a call to _int_malloc() is made passing in the size of the request and the arena pointer. Once inside, the size is rounded up if necessary to account for alignment, at this point one of four execution paths are available, the size of the chunk is checked to see if it falls into the fastbin size, then the small normal bin size, and large normal bin size and finally if it's too large for both of them. The three paths are discussed separately as only one is actually possible per call to malloc().

0.3.3.0 Allocate a fastbin chunk

if the chunk falls within the fastbin chunk size then the index for that size of fastbin is obtained and the fastbin pointer is initialized by obtaining using that index as an offset into the fastbin array in the current arena. Following this a check is made to ensure that returned pointer is not NULL and the 'victim' variable which will eventually be returned as the allocated chunk is assigned to the first block of memory in the linked list of fastbin's for that size. A security check is made to ensure that a chunk of this size should actually be in this particular fastbin. Assuming the check passes then the pointer to the array of fastbin's is assigned to the next chunk in the list. A call to check_remalloced_chunk() is made, however in almost all circumstances this is compiled out due to the function name being a macro that is defined to do_check_remalloced_chunk() _only_ if MALLOC_DEBUG is defined (and as such will not be considered in this paper). Then a pointer to the portion of memory that will be returned to the user is obtained and if previously requested a perturb byte is set there and finally the pointer to user memory is returned to public_mALLOC().

0.3.3.1 Allocate a small normal bin chunk

A request is within the small normal bin chunk size if it is larger than the maximum fastbin chunk size and smaller than 512 bytes. If it falls within this range, then the index for the requested size is obtained and used as an index into the arena's normal bin array obtaining a pointer to the correct bin. Then the victim variable is assigned to the last chunk in the bin, if this pointer does not point to the bin itself (indicating there are no chunks on the list), then the victim pointer is checked to ensure its not NULL. If it is a call to malloc_consolidate() is made.

malloc_consolidate() is a function that is described as a 'variant of free()' in the comments. It checks to see if there are any fastbin chunks that have been free()'d and if so obtains a pointer to the unsorted chunk bin and the largest fastbin list. A pointer is initialized to this fastbin chunk if the chunk doesn't point to NULL and the original pointer is set to point to NULL. Following this the next chunk on the list is obtained, and if the previous

DRAFT

chunk is marked as not being in use then it is consolidated into the current chunk by adding the size of the previous chunk with the size of the current chunk and then modifying the pointer for the current chunk so that it points to the previous one. The chunk is then unlink()'d (The unlink() process will be described in detail later on in the paper). Then the next chunk is checked to see if it is the top chunk, and if not it is consolidated into the current chunk and unlink()'d. It is then added onto the unsorted chunk list. If it is the top chunk, then it is consolidated into the top chunk. This process repeats for every block of memory for this fastbin index, and then for every fastbin list, eventually consolidating all of the fastbin chunks and placing them on the unsorted list and finally returning.

If the victim pointer was NULL and malloc_consolidate() was called, then the allocation operation continues for large normal sized chunks and is detailed in section 0.3.3.2. If the pointer was not NULL, then victim's back link is obtained and the bin's back link is assigned to it, the victim chunk has its previous in use bit set, and the victim chunks forward link is assigned to bin. The pointer to user memory is obtained, the perturb byte is potentially set and the pointer to user memory is returned to public_mALLOc().

0.3.3.2 Allocate a large normal bin chunk

A request that falls outside the range of fastbin's chunks and small normal chunks range potentially falls into the large normal bin. The first step done if the chunk fell outside of the small normal bin chunk size is that the index of the bin for that size is obtained and if there are fastbin chunks in the arena then a call to malloc_consolidate() is made, if the size was in the small normal bin chunk size and malloc_consolidate() was called in the attempt to allocate it, then this entire step is skipped due to fastbin chunks already having been consolidated. The allocator enters a for(;;) loop and points victim to the unsorted chunk index in a while loop that will continue until a chunk if found (jf: dc) or until the list has been traversed. A pointer to the victim's backwards link is obtained and a check against the victims size is made to ensure that its not smaller then the smallest possible chunk size. The size of the chunk is then obtained.

If the chunk was in the small normal bin range and the last remainder chunk is the only one on the list, then the last_remainder chunk is split to account for this allocation, and the new last_remainder chunk is set to its new size and put back on the unsorted chunk bin. The victim chunk becomes the excess of the last_remainder chunk and has it's flags set, potentially a perturb byte set and the pointer to the user memory is obtained and returned to the user.

Otherwise, the victim chunk (jf: dc) is removed from the unsorted chunks list. If it's size is exactly equal to the requested size, then it has its flags set

DRAFT

and potentially the perturb byte is set. The pointer to user memory is obtained and finally retuned to the user.

Otherwise, if it is in the small normal bin range, the unsorted victim chunk is placed on its appropriate list. If it is not in the small normal bin range, then the unsorted chunk is placed back onto the large normal bin list sorted by descending size order. Regardless of size, after the unsorted chunk is placed into a bin, the has it's bitmap marked to show that the list is definitely not empty (jf: dc) and the addition to the bin's linked list is finalized.

If the requested size does not fall into the small normal bin range, then the bin for the index found earlier is obtained and a check is done to see if the bin is empty or if the largest chunk in the bin is too small. If the bin is not empty or the largest chunk is larger than the bin is searched through until a chunk that is large enough to accommodate the request is found, assigning victim to the next chunk in victim's list each time, ultimately leaving victim pointing to a chunk that is large enough to fulfill the request. The excess memory is found by subtracting the size of the chunk with the request and the victim chunk is unlink()'d from the list.

If the remaining size is less than the minimum size of a chunk, then the in use bit is set and the excess is not split. Following that if the chunk is not in the main arena then its NON_MAIN_ARENA bit is set. If the remaining excess is large enough to be a chunk, then it is split and placed back in the unsorted bin. In both cases, a chunk has been found in the current bin and a pointer to the user memory is obtained, a perturb byte is potentially set and the pointer to user memory is returned to public_mALLOc().

If the previous check to see if the bin was empty or if it's largest chunk was too small yields that the bin was indeed empty or all of its blocks of memory too small, then the bin index is incremented and a search of the rest of the bins is performed. The process of this search is similar enough to the previous search through the current bin in that it looks at a given bin that isn't marked as empty in the bitmap, a chunk is potentially found and potentially split, or a chunk is not found and the next bin is searched. If none of the bin's contain a suitable chunk, then the loop terminates and the top chunk is used.

First the victim pointer is pointed at the top chunk and the size variable is assigned to the top chunks size. If the top chunk is large enough, then it is split and reassigned to the split chunk, with the other portion having its user portion of the chunk returned to the user after potentially have its perturb byte set. If the top chunk is not large enough and the fastbin chunks have not been consolidated a call to malloc_consolidate() is made and the for(;;) loop earlier is repeated. If the top chunk is not large enough and the fastbin chunks have been consolidated then a call to

DRAFT

sYSMALLOc() is made, bringing us to the next subsection about large chunks.

0.3.3.3 Allocate a large chunk

If an allocation request is made and all of the above methods fail to find an adequate chunk on the free lists or by splitting the top chunk, then we end up in the sYSMALLOc() function, which will attempt to either just mmap() a new region for the chunk or extend the top chunk.

If the requested size is less than the maximum mmap() threshold and we do not already have too many memory mapped sections, then a the size is padded to include the allocators overhead and a new section of memory is obtained via mmap(). If this succeeds, then its alignment is potentially fixed and the pointer to the user portion of the memory is returned to public_mALLOc().

If the call to mmap() fails, there are already too many memory mapped segment or if the size is larger than the mmap() threshold, then the allocator attempts to extend the top chunk. Upon entering this section of the code base, the old pointer to top, the size of the old top, and the end of top are all saved. A few checks via assert() are performed, such as if this is not the first time this has occurred that the old top size be at least the size of the minimum chunk (jf: dc MINSIZE) and that it has its previous in use bit set, that the old top size is not large enough for the request and that all of the fastbin chunks have been consolidated.

Assuming all of those checks pass, then if the current arena is not the arena pointer, then the first step is attempting to grow the current heap which is accomplished by a call to grow_heap(). grow_heap() can be used to grow or shrink a given heap, as determined by its second argument which is a signed size argument. If the size argument ('diff') is positive, then it is properly aligned and a new_size variable is assigned to the size of the current heap summed with the diff variable. A check is done to ensure that the new size is at least as big as the current one, and assuming that condition is true then a call to mprotect() is made attempting to set the permissions to the end of the current heap plus the requested extension size to be read and writeable, implying that this will map the requested memory. If the extension requested is negative, then a the new size is again summed with the current heap size and the diff variable, and assuming it passes a similar check as made above for an extension, then a call to mmap() is made, remapping that space with no permissions to it.

If all went well, the heap size is assigned the value of the new_size variable and the grow_heap() function returns 0, otherwise it returns -1 or -2 depending on the error encountered.

If the resize of the current heap succeeded, then the arena's system_mem variable is assigned the value of its new size subtracted from the old heaps size and the global variable arena_mem is incremented by the new size subtracted from the old size.

If the resize failed, then a new heap is requested by calling new_heap() which was detailed above in section 0.3.2. If this call fails, or if the arena is indeed the main_arena, then the size variable is adjusted to add in padding and potentially modified if for instance the area's memory is contiguous. If the size requested is not negative, then a call to MORECORE() is made for the adjusted size. MORECORE() defaults to sbrk() which extends the data segment of the process.

If this call does not fail, then a potential callback is made by calling the __after_morecore_hook(), this is a variable that cannot be assigned through the ptmalloc2 implementation and as such is not explored here.

If the call to MORECORE() failed, and the platform has mmap(), then the top is extended via mmap(). If the size requested is less than MMAP_AS_MORECORESIZE (1024*1024), then the size is assigned to the value of MMAP_AS_MORECORESIZE. A check is then performed to ensure that the size variable has not wrapped around past zero, a call to mmap() is made. If this call did not fail then the arena has its flags set to show that it is non-contiguous.

Next, if the call to MORECORE or mmap() resulted in an extension, then the top chunk has its size extended as well. Then a check is performed testing that the arena is still contiguous, that the old size is not zero and base returned by MORECORE() is less than the old end of the top chunk, if this is all true then something horrible happened and our previous space is no longer valid and an assert(0) is called. Otherwise, then MORECORE() didn't extend the memory and it is still there, so adjustments are made.

(jf: fix comments copy/paste) The adjustments include things such as, if the first time through or noncontiguous, we need to call sbrk() just to find out where the end of memory lies, The need to ensure that all returned chunks from malloc will meet MALLOC_ALIGNMENT, If there was an intervening foreign sbrk(), we need to adjust sbrk() request size to account for fact that we will not be able to combine new space with existing space in old_top and so on.

Finally, after all of the adjustments and potential paths for extension occur, if one of them succeeded, then the newly resized top chunk is split and the requested allocation is made from the top chunk. Finally, (yes *finally*), a pointer to the user memory is returned to _int_malloc().

If none of the above extension attempts succeeded, then the function returns NULL to _int_malloc(). Interestingly enough, upon return from

DRAFT

sYSMALLOc(), the return value is not checked and if a series of perturb bytes are written, they could potentially be written to NULL causing the using application to crash. Either way, the resulting pointer returned from sYSMALLOc() is returned to public_mALLOc().

If for some reason, the allocation failed and NULL was returned to public_mALLOc(), and the arena used was not the main_arena, then mutex to the current arena is unlocked and the mutex to the main_arena is locked, and another call into _int_malloc() is called with the main_arena this time and then the main_arena's mutex is unlocked. If the arena in question was indeed the main_arena, then a call is made to arena_get2(), afterwards unlocking the main_arena mutex. If that call succeeds, then a call is made back into _int_malloc() again with the new arena, followed by an unlock of its mutex.

Finally, if the allocation resulting from the first possible call to _int_malloc() succeeded, then its mutex is unlocked. Finally a few sanity checks are made via assert() and the pointer it returned to the caller of public_mALLOc() and we are finally done with the internals of chunk allocation.

0.3.4 Resize an already existing chunk

Resizing an existing chunk is a slightly more straight forward process than allocating one, as the entire subsystem is most likely initialized and there is quite simply less to do, or rather parts of the call are serviced by calling _int_malloc(), which makes it simpler for us due to that process having been already described.

Upon entry into public_rEALLOc(), the __realloc_hook is checked and if not NULL it is called. Next if the size of the reallocation is zero and the pointer is not equal to NULL, then public_fREe() is called on the chunk. If the pointer to the chunk to be resized is NULL, then a call to public_mALLOc() is made, returning its return value to the caller.

Having the standard things out of the way, the current size of the chunk and the pointer to the beginning of the chunk as viewed internally is obtained. A few sanity checks are made, ensuring that the pointer has not wrapped around the address space and that the chunk is properly aligned.   If the chunk was previously allocated via mmap() and the system has mremap(), then the chunk is remapped to the new size. If that succeeds then the pointer is returned to the user and we're done.

If that call did not succeed, then a call to public_mALLOc() is made for the new size, if that call succeeds the data in the old chunk is copied over to the new chunk and the old one is unmapped. Finally, either the new chunk is returned to the caller, or NULL is returned indicating failure.

DRAFT

If the chunk was not memory mapped, then the arena for the chunk is obtained via a call to arena_for_chunk() which simply checks to see if the chunk is in a non-main arena, if its not then the main_arena is returned, otherwise the heap for the chunk is determined and its pointer to the arena returned.

Upon receiving a pointer to the arena for the chunk it's mutex is locked, and upon successfully obtaining it _int_realloc() is called with the arguments of the old pointer, the arena and the size requested.

Once inside _int_realloc(), the internal pointer to the chunk is obtained along with its old size. The alignment of the chunk is once again checked along with ensuring that the old size of the chunk is at least as large as the minimum chunk size. Another check to ensure that the chunk is not mmap()'d is made, and a pointer to the next chunk is obtained.

A few basic sanity checks are performed upon the next block of memory, including that its size is at least as big as the minimum chunk size and that its size is not larger than the arena's recorded amount of memory.

If the resize request is smaller than the current chunk size, then the new pointer and size are assigned the values of the old values and it will be split later on in the function. Otherwise, if the next chunk is the top chunk, and the top chunk is larger than the new size and the old chunk size summed, then the new chunk is expanded forward into the top chunk, and the pointer to the useable portion of the chunk is returned to public_rEALLOc(). If the next chunk is not the top chunk, and the next chunks size is large enough to accommodate the request, and the next chunk is marked as not in use, then it is unlinked and used as the resized chunk of memory.

If none of the above conditions will satisfy the resize request, then a call to _int_malloc() is made allocating a new chunk of memory, if the call fails then the failure is propagated up to public_rEALLOc(), otherwise the internal pointer to the chunk is obtained. If this new chunk is equal to the next chunk of the old one, then the new size is adjusted to include the old one and the new pointer is assigned to that of the old one, avoiding a copy of the contents of the old chunk.

Otherwise, if the contents to copy are less than 39 bytes (or 72 if we have eight byte pointers), then the contents are manually copied into the new chunk, otherwise a call to MALLOC_COPY() is made, which is most likely a macro for memcpy(). Finally, the old chunk is deallocated via _int_free() and the useable portion of the chunk is returned to public_rEALLOc().

Finally, if the old chunk is larger than the request new size, then it is split and the old chunk is free()'d and the new pointer to the split chunk is returned to public_rEALLOc().

DRAFT

Upon returning to public_rEALLOc(), the arena's mutex is unlocked, a few sanity checks are performed (the same ones as described at the end of chunk allocation), and the pointer to the new memory is returned to the user of the API.

0.3.5 Free a chunk

The process of free()'ing a chunk is also somewhat straight forward, although it should be mentioned that I will skip over most of the security checks in this routine due to them being covered later in the paper. Upon entry into the public_fREe() function, any debugging hooks that were initialized are called. If the pointer passed in to free() is NULL then the routine immediately returns. Otherwise the internal point to the chunk is obtained and it is checked to see if it was allocated via mmap(), if it is then it is simply unmapped and the routine returns.

Otherwise, the arena for the chunk is obtained via a call to arena_for_chunk(), which was described previously in the discussion of resizing an already allocated chunk. Once the arena is found, its mutex is locked and a call to _int_free() is made passing in a pointer to the arena and the chunk to free().

Once inside _int_free(), the internal pointer and the chunks size is obtained. The usual check to make sure it didn't wrap and that the chunk is properly aligned is made along with a check to make sure the chunk is at least as large as the minimum chunk size.

Next, if the chunk is in the fastbin chunk size range, then the flag indicating that the arena has free fastbin chunks, if there are perturb bytes they are unset, and the chunk of memory is manually linked into the given fastbin linked list for that size.

Otherwise if the chunk is not in the fastbin chunk range and it isn't memory mapped, then the next chunk and the next chunks size is obtained. If a perturb byte was used then it is unset. If the previous chunk is marked as not being in use, then it is coalesced into the current chunk by increasing the size of the current chunk by the size of the previous one's and then unlink()'ing it. Then if the next chunk is not the top chunk and it's marked as not being in use, then is is coalesced in the same fashion as the previous one. Then the potentially coalesced chunk is linked into the unsorted list manually, which provides a potential vector for attack as described in (jf: ref malloc malleificarium). Otherwise, if the next chunk is the top chunk, then the block of memory is coalesced into the top chunk.

Following the above steps, if the size of the chunk being free()'d exceeds 65536 bytes, then triggers possible consolidation of the fastbin chunks (if they exist) via malloc_consolidate(), and if the arena of the chunk is the main_arena, and the size of the top chunk is larger than or equal to the

DRAFT

configurable trim threshold, a call to sYSTRIm() is made, which is described by the comments as being the inverse of sYSMALLOc().

Once inside of sYSTRIm(), the size in excess is calculated and if it's larger than zero bytes, an attempt to give it back to the system is made. However if the end of the arena's top chunk is not equal to the end of the current data segment as returned by a call to MORECORE() (sbrk() called with a NULL pointer returns the end of the current data segment) then no attempts to trim the heap are made due to potentially interfering with external calls to sbrk().

Assuming that is not the case, then the excess memory is released by calling MORECORE() with the size –extra. Following this, the __after_morecore_hook() is potentially called, and the new end of the data segment is obtained and checked to see if MORECORE() failed. If it did not then the amount of memory released is calculated by subtracting the new address from the previous end of the data segment and if that size is not equal to zero then the top chunk is adjusted to account for its new size and sYSTRIM() returns one, in any other conditions sYSTRIM() returns zero indicating that excess memory (if applicable) was not trimmed.

Upon return back into _int_free(), the address of the current heap is obtained, the heap that is obtained has its arena pointer checked against the current arena pointer via assert() and if everything is okay then a call to heap_trim() is made.

heap_trim() is a function that performs a similar set of operations as sYSTRIm(), except that it operates on heaps rather than the top chunk with the purpose being that the heap in question might go away entirely or at least shrank. (jf: fill out details if you have the time) Upon returning back to _int_free(), the final condition is checked and that is if the chunk was allocated with mmap(), if it was the chunk is simply unmapped. This may seem redundant due to checks done in public_fREe() until you realize that other internal functions call _int_free() themselves (while already holding the lock for the arena).

Finally, _int_free() returns to public_fREe(), the mutex for the arena is unlocked and we are done.

0.3.6 Delete a heap

Heap deletion is probably the simplest task described, it is actually implemented as a macro and only referenced once in the code base (jf: dc). Quite simply a check is made to determine if the heap in question is equal to the aligned_heap_area variable, if so the variable is set to NULL and in either case the heap is unmapped via munmap().

DRAFT

This concludes all of the necessary steps for the deletion of a heap, and thankfully finally concludes our discussion of heap operations.

0.4 Putting it all together

Now, reviewing everything we find that the heap is a misnomer, that multiple heaps can exist per process and that they are initialized implicitly by attempting to allocate memory. New heaps can be created if one does not already exist or if the locks to one cannot be obtained. We have found that fastbin chunks have the least operations performed on them and many operations, in particular an allocation of a normal bin chunk can cause them to be coalesced into larger chunks. We have determined that the allocator tries incredibly hard to accommodate any request for memory given to it, and that in extreme and unusual circumstances it is possible to cause a NULL pointer to be written to.

We have found that it's possible for a heap to be deleted and potentially folded into another heap (jf: dc), and that having a top chunk that is rather large can cause it to get trimmed, we have also noted that an application that calls sbrk() itself to allocate some memory will keep this trimming from happening.

Finally, we have learned that the heap is simply a persistent data structure that comprises of contiguous or non-contiguous memory that often exists in the current data segment.

(jf: more here about putting it all together)

DRAFT

1.0 Double free()'s

1.1 What is a double free()?

A double free() is quite simply where a chunk of memory that was previously allocated by one of the allocation routines that is later free()'d more than once, typically twice thus the title. Multiple free bugs are interesting because they give us insight into how a given heap operation works, that is to say unlike a buffer overflow, which is never valid, a multiple free vulnerability occurs as a result of valid instructions being executed in an invalid manner. That is to say that they are invalid because of their time in space, for example given the following example:

```
void *ptr = malloc(size);

if (NULL != ptr) {
```

```
free(ptr);        /* a */
free(ptr);        /* b */
```

When we examine this example, we have the first free() which has been labeled 'a' and the second that has been labeled 'b'. Either one is valid given that the other one does not exist, and you can even switch them around so that 'b' comes first and then 'a' and it is still invalid. What really makes them interesting to the author is primarily that it's somewhat rare in the realm of computer science that executing the exact same instruction twice has such disasterous results, well maybe that is not 100% true, but you surely get the point.

The second thing that makes them fairly interesting is that despite their being fairly well known there has been very little said about them, or exploiting them. In fact, the two seminal papers on heap exploitation published in Phrack magazine in 2001, Vudo malloc tricks and Once upon a free() (jf: ref), make no mention of the problem at all. Later in 2003, a paper entitled Advanced Doug Lea's malloc exploits again published in Phrack magazine (jf: ref) makes a total of four references to double free() vulnerabilities, once in reference to a specific vulnerability (mentioned in passing), once as an example of an operation that can 'corrupt malloc's internal structures' , once in a small chart referencing the flow from vulnerability to their 'aa4bmo' primitive, and finally the most informative reference:

> 'For example, in a double free() vulnerability scenario, we know the second free() call (trying to free already freed memory), will probably crash the process. Depending on the heap layout evolution between the first free() and the second free(), the portion of memory being freed twice may: have not changed, have been reallocated several times, have been coalesced with other chunks or have been overwritten and freed.'

Moving on we have the paper The Malloc Maleficarum (jf: ref) posted to the Bugtraq mailing list in 2005 which makes one reference to double free(), which is actually part of glibc included in part of a code snippet from the library. Then in another brilliant piece of work, The Shellcoders Handbook (ref), they are first referenced as follows:

> *'Also included within the heap overflow biological order are double free() bugs, which are not discussed in this chapter. You can read more about double free() bugs in chapter 16'*

Then however, when you skip to chapter 16 you find:
(jf: look it up when you get home)

In fact, in review only two sources have even a somewhat decent explanation, one full out tells the details of exploitation, and the other hints

DRAFT

at it. In a book sure that should be on your shelf (if it's not already), The Art of Software Security Assessment (jf: ref), published in 2007 (!) the following reference is made:

> *'There is also a threat if memory isn't reused between successive calls to free() because the memory block could be entered into free-block list twice. Later in the program, the same memory block could be returned from an allocation request twice, and the program might attempt to store two different objects at the same location, possibly allowing arbitrary code to run. The second example is less common these days because most memory management libraries (namely, Windows and GNU libc implementations) have updated their memory allocators to ensure that a block passed to free() is already in use; if it's not, the memory allocators don't do anything. However, some OSs have allocators that don't protect against a double free attack; so bugs of this nature are still considered serious.'*

This provides the best reference to exploiting multiple free() vulnerabilities in a professionally published work that the author of this paper is aware of, and it is only a hint at how one goes about exploiting it, and furthermore it essentially tells you that on GNU libc implementations it's no longer possible to exploit. This is the basis of this paper, what protections the glibc implementation has, what protections it affords, where it can be bypassed and when it cannot be bypassed. Furthermore, the hope is that by examining the heap through the looking glass expands your focus to look at more complex bugs that arise from application and data structure state, at writing exploits that manage to massage the application and/or data structures into a certain state and at generally recognizing they're beautiful complexity. To say this another way, as security mechanisms progress, and as traditional style overflows become more and more rare, bugs dependent on state become more important, and as the mechanisms become stronger the ability to work the application into a given state becomes absolutely necessary.

In short, the good days are mostly gone, and it's time that we start to look at the future, and examining this style of application flaw we can form the foundation of understanding application insecurity in the future.

1.2 Traditional double free() exploitation

Continuing on the theme of the previous subsection, we have to ask 'why has exploitation of this particular type of vulnerability been largely glossed over by an industry defined by it's pedantry?' Is it because the authors of these referenced works did not know or understand the flaws? Is it because the flaws are not important or they were attempting to horde information?

DRAFT

In all cases, the author of this paper believes that the answer is 'No', they very well understood the situation, they understood their importance and they were not trying to horde information, but rather to those who understand the heap and its operations to this detail, the vulnerability and exploitation of it is taken as a given, once you reach one level of comprehension it implies an understanding of all or most of the bugs.

However, in the authors experience this is not true. We work and play in an interesting industry, where if a specific whitepaper has not been written describing all of the details the masses by and large do not experiment and learn on their own. In fact, the general lack of comprehension of exploiting these bugs has surprised even this author. In talking to co-workers and friends, both past and present over the last six months or so it's become incredibly prevalent that the problem is not well understood and often dialogues have gone somewhat akin to the following:

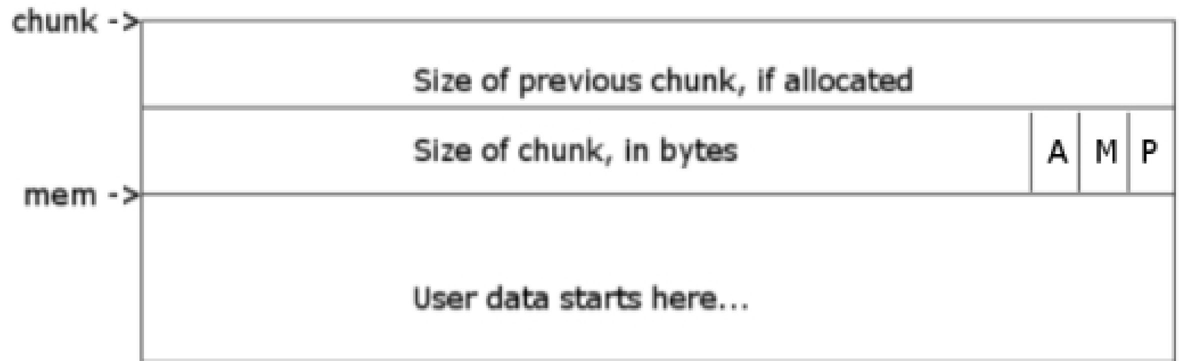
> 'It's quite surprising to find out how few people don't understand even the exploitation of double free() vulnerabilities in heaps from 5 years ago'
>
> 'Yea I know, […], I once had a bug where I needed to get malloc() called instead of free()'
>
> 'That was probably a double free() vulnerability'
>
> 'Oh, yea you're right!'

Furthermore, others who have indeed fully understood them are not entirely sure of the current state of exploitability. None of this is a fault of the people, they're all been fairly bright and generally leaders in their field, but the vulnerability is relatively rare to find and is typically found during the development process.

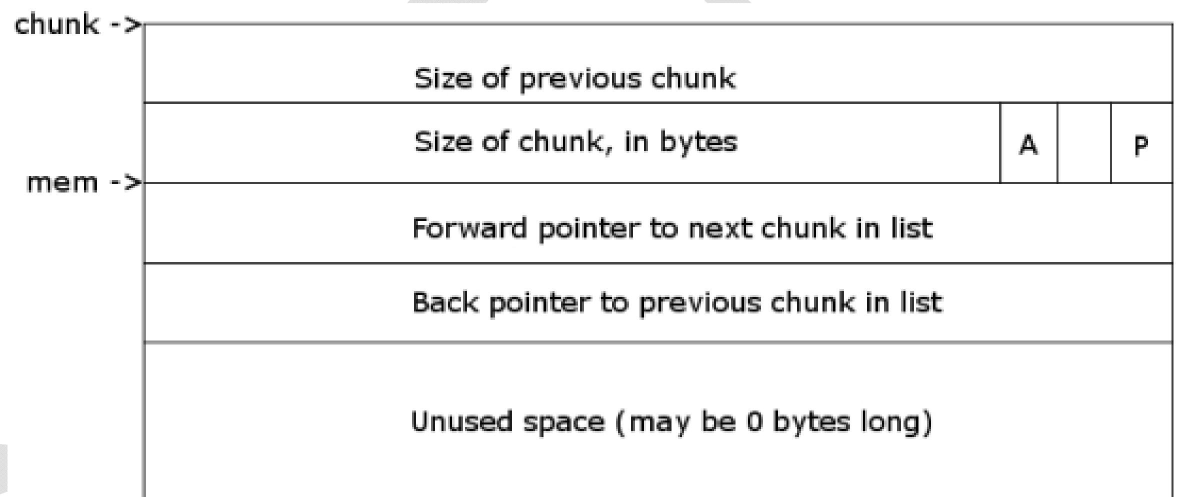
At any rate, at the end of the previous section I mentioned that there were two works that talked in any real detail about multiple free vulnerabilities, one that hinted at its exploitation and one that explained it. I then proceeded to only talk about one of them. The other was a post to the Bugtraq mailing list in 2003 by Igor Dobrovitski (jf: ref) detailing an exploit he had written in CVS servers up to an including version 1.11.4.

The basic summary is that if you recall, a chunk that is currently allocated looks like this:

DRAFT



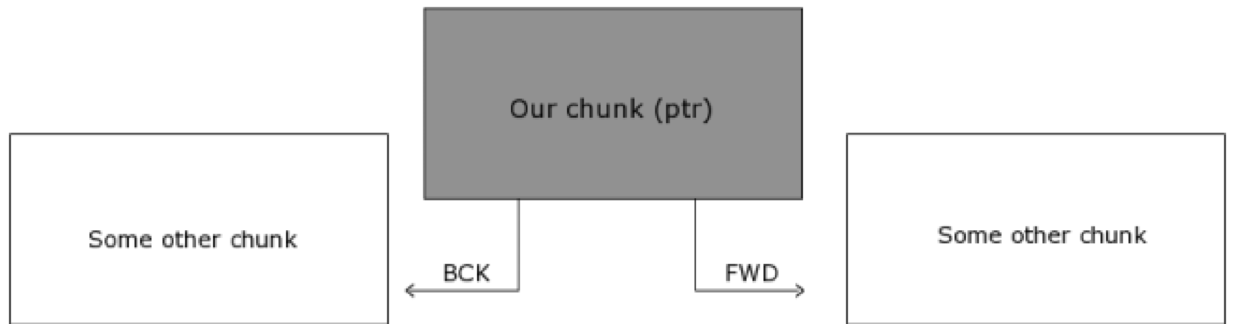
And that a chunk that exists on a free list is represented like this:



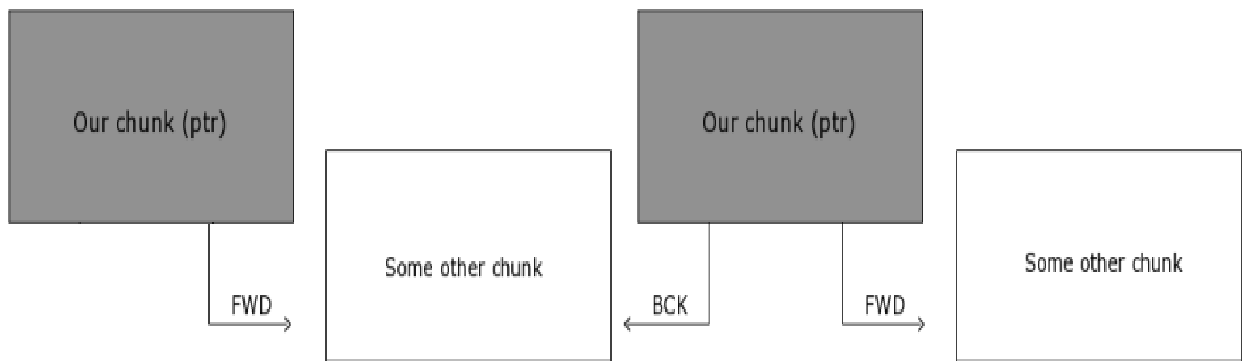
So that in a free chunk of memory, where the pointer returned to the user of the API used to be is now metadata used internally to traverse the free list. So that given our previous code example earlier, where a chunk was allocated and then repeated twice in succession we have the following representation of the free list:

After the first free():

DRAFT



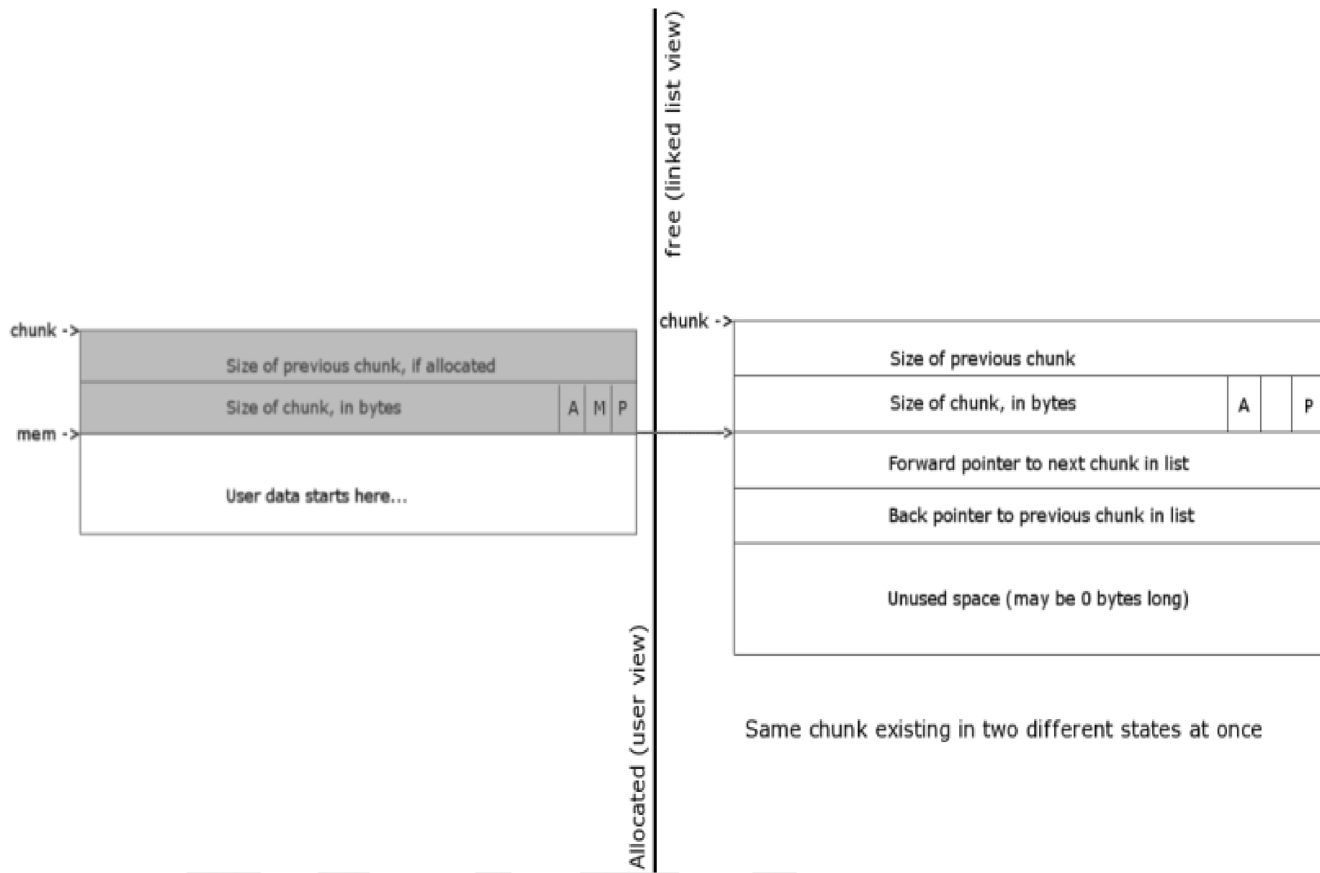
After the second free():



These images are of course somewhat simplified and don't include for instance the forward and backward pointers for the other chunks or the backwards pointer for our second instance of our chunk, these details were left out mostly in order to simplify the picture and put the emphasis on the overall point.

As we can see here, we have the crux of the issue, a pointer to the same block of memory occurs on the linked list twice, what essentially happens then, is that when the next call to malloc() occurs (providing chunk's are not coalesced, that the chunk is not taken from the top chunk and so on), then what essentially happens is this:

DRAFT



Same chunk existing in two different states at once

Hopefully at this point, the problem itself has largely become appearent to you. If not, the problem is that because the same chunk of memory exists in two different states and parts of the chunk are reused that after one of the chunks is reallocated that the pointer returned to the user of the API corresponds to the start of the pointers in the free chunk, so the first eight bytes (or more depending on pointer size) will overwrite the metadata and potentially interfere with internal operations. But what operation?

Well specifically, classic exploitation attacks the unlink() macro (just like pretty much everything else in classic heap exploitation). The unlink() macro was defined in the following manner:

```
#define unlink( P, BK, FD ) { \
        BK = P->bk; \
        FD = P->fd; \
        FD->bk = BK; \
        BK->fd = FD; \
}
```

Basically what is happening here is your basic operation to remove a node in a linked list, P is the pointer to the chunk itself, BK is a pointer to the

backward link stored in the chunk, FD is then of course the forward pointer. So by overwriting the BK and FD pointers, when this macro was called you could overwrite any four bytes with four bytes of your choosing. If you don't understand this concept it will not be further covered here because it's been covered in detail in almost all of the papers listed in the references and originally detailed by Solar Designer in an advisory affecting Netscape (jf: ref).

So the flow of a classic multiple free() exploit was as follows (disregarding any special tricks needed to avoid chunks from being coalesced/et cetera):

0. Get the same chunk of memory free()'d twice

1. Get one of these chunks allocated back

2. Overwrite the two pointers to the forward and backwards link in the chunk on the free list.

3. Get the second instance of the chunk still on the free list allocated so that when the unlink() macro was called our modified pointers are used and an arbitrary address is overwritten.

1.3 Oops, it's not 1996 anymore or why that technique doesn't work anymore

So what's wrong? What's changed and why doesn't this technique work anymore? Quite simply the world became a little more wise and one of the greatest advances in security for linked list based allocators came along and changed the unlink() macro to the following:

```
#define unlink(P, BK, FD) {                                              \
  FD = P->fd;                                                            \
  BK = P->bk;                                                            \
  if (__builtin_expect (FD->bk != P || BK->fd != P, 0))                  \
    malloc_printerr (check_action, "corrupted double-linked list", P);  \
  else {                                                                 \
    FD->bk = BK;                                                         \
    BK->fd = FD;                                                         \
  }                                                                      \
}
```

As you can see, a basic sanity check has been added to verify that the current chunks next chunks previous chunk is equal to the current chunk, and that the current chunks previous chunks next chunk is equal to the current chunk. This check effectively squashes the ability to use unlink() as a method for overwriting arbitrary data. However, like all good things, that was not the end and the paper The Malloc Maleficarum (jf: ref) is the most

DRAFT

up to date, public paper on current exploitation methods of overflows in the glibc heap.

2.0 The example

2.1 Code overview

The vulnerability is the result of multiple error handling checks being performed on functions that call each other that on error will cause a multiple free condition. The vulnerability referenced here is specifically in mod_auth_kerb versions (jf; version) through (jf: version) and appears to be the result of using the asn.1 compiler that at one time shipped with the Heimdal Kerberos implementation (jf: proper name?), furthermore the vulnerability existed in multiple vendors, however for the sake of clarity this is the only one discussed here. It should be noted that current code generated by the asn.1 code is still flawed in multiple areas and any project, organization or person who has made use of the Heimdal asn.1 compiler is encouraged to inspect the generated code to verify that similar vulnerabilities do not exist in their code base.

2.2 Vulnerability 0 – array of pointers double free

```
asn1_NegTokenInit.c:
  70 #define FORW if(e) goto fail; p += l; len -= l; ret += l
  71
  72  int
  73  decode_NegTokenInit(const unsigned char *p, size_t len,
NegTokenInit *data, size_t *size)
  74  {
  [...]
0 105 e = decode_MechTypeList(p, len, (data)->mechTypes, &l);
  106 FORW;
  [...]
  206 fail:
  207 free_NegTokenInit(data);
  208 return e;

asn1_MechTypeList.c:
  37 #define FORW if(e) goto fail; p += l; len -= l; ret += l
  38
  39 int
  40 decode_MechTypeList(const unsigned char *p, size_t len,
MechTypeList *data, size_t *size)
  41 {
  [...]
1 62 e = decode_MechType(p, len, &(data)->val[(data)->len-1], &l);
  63 FORW;
  [...]
70 fail:
71 free_MechTypeList(data);
```

```
 72 return e;
 73 }

asn1_NegTokenInit.c:
 211 void
 212 free_NegTokenInit(NegTokenInit *data)
 213 {
 214 if((data)->mechTypes) {
2 215 free_MechTypeList((data)->mechTypes);
 216 free((data)->mechTypes);
 217 }
asn1_MechTypeList.c:
 75 void
 76 free_MechTypeList(MechTypeList *data)
 77 {
 78 while((data)->len){
 79 free_MechType(&(data)->val[(data)->len-1]);
 80 (data)->len--;
 81 }
4 82 free((data)->val);
 83 }
```

What we have here is at reference number (RN) 0 we see that the function decode_NegTokenInit() calls decode_MechTypeList(), if that routine fails then we have a macro named FORW that expands to a test of the return value and if it is not zero it goes to fail. Then at line 206 of asn1_NegTokenInit.c we see that the label fail equates to a call to free_NegTokenInit() with an argument of data. Next at RN1 we find that inside of decode_MechTypeList() it calls decode_MechType() and upon failure it also executes a goto statement to the label fail, which executes the function free_MechTypeList(). Next at RN2 we see that the function free_NegTokenInit(), the function called upon failure by RN0, which in turn calls calls free_MechTypeList if (data)->mechTypes is not equal to NULL. Finally inside of free_MechTypeList() we see that RN4 that (data)->val is free()'d. The flaw being that first the failure in the function called by RN1 free()'s (data)->val, and then the error is propagated up to decode_MechTypeList() which in turn calls free_NegTokenInit(), which eventually reaches RN4, double free of the variable data->val.

2.3 Vulnerability 1 – double free of user-data

```
asn1_MechTypeList.c:
 37 #define FORW if(e) goto fail; p += l; len -= l; ret += l
 38
 39 int
 40 decode_MechTypeList(const unsigned char *p, size_t len, MechTypeList *data,
 size_t *size)
 41 {
 […]
0 62 e = decode_MechType(p, len, &(data)->val[(data)->len-1], &l);
 63 FORW;
 […]
```


DRAFT

```
70 fail:
71 free_MechTypeList(data);
72 return e;
73 }
```

asn1_MechType.c:

```
30 #define FORW if(e) goto fail; p += l; len -= l; ret += l
31
32 int
33 decode_MechType(const unsigned char *p, size_t len, MechType *data,
size_t *size)
34 {
1 41 e = decode_oid(p, len, data, &l);
42 FORW;
[...]
45 fail:
46 free_MechType(data);
47 return e;
48 }
```

der_get.c:

```
388 int
389 decode_oid (const unsigned char *p, size_t len,
390             oid *k, size_t *size)
391 {
[...]
2 411     e = der_get_oid (p, slen, k, &l);
[...]
418 }
144 int
145 der_get_oid (const unsigned char *p, size_t len,
146              oid *data, size_t *size)
147 {
[...]
170     if (p[-1] & 0x80) {
3 171         free_oid (data);
172         return ASN1_OVERRUN;
173     }
[...]
178 }
```

asn1_MechTypeList.c:

```
75 void
76 free_MechTypeList(MechTypeList *data)
77 {
78 while((data)->len){
4  79 free_MechType(&(data)->val[(data)->len-1]);
80 (data)->len--;
81 }
82 free((data)->val);
83 }
```

asn1_MechType.c:

```
50 void
```

DRAFT

```
 51 free_MechType(MechType *data)
 52 {
5 53 free_oid(data);
 54 }

der_free.c:
 52 void
 53 free_oid (oid *k)
 54 {
6 55     free(k->components);
 56 }
```

First at RN0 we see that decode_MechTypeList() calls free_MechTypeList() if the call to decode_MechType() fails. Next inside of decode_MechType() at RN1 we find a call to decode_oid(), which calls der_get_oid() at RN2, We find that inside of der_get_oid() at RN3 that if an argument potentially can cause a call to free_oid() (p is a pointer that is user-controlled and incremented in the body of the function). At RN4 we see that free_MechTypeList() will call free_MechType(), which is also called by RN1, and at RN5 free_MechType() calls free_oid() and eventually, when all three functions reach free_oid() they free() k->components and then propagate the error up to the calling function, which in turn errors and calls it's free() function. The end result being that data->components (referred to as k->components locally in free_oid()) is free()'d three times, oops.

2.4 Goals – write-what-where

The goals of exploitation of these bugs are of course what they always are, arbitrary execution of code. However, as the abstract of the paper implies, we are seeking to accomplish this without ever overflowing a single heap buffer. In the first example explored this is not entirely the case but the example was left intact in order to provide an introduction to the 'worst case' scenario that multiple threads can produce while operating in the heap.

That said, the overall goal is to be able to write what we want, where we want, write-what-where.

3.0 The effects of a multi-threaded environment

3.1 Thread safety in GNU libc's allocator

In earlier versions of glibc, specifically versions prior to (jf: version) there was no thread safety and the functions were not async-safe meaning that an interruption during critical sections of code could has disasterous results as specifically noted by Michal Zawelaski (jf: sp?) in his paper Delivering Signals for fun & profit and demonstrated in a vulnerability he discovered and published at the same time. (jf: ref)

DRAFT

Times, and code have changed however and in those days the glibc dynamic memory allocator was simply Doug Lea's, and in current implementation as pointed out previously the implementation employed by glibc is ptmalloc2 and as all neat tricks in this industry, they were eventually fixed.

In ptmalloc2, thread (and other similar environments) safety is provided by two main mutex's or locks (jf: dc), the first being the list_lock which was mentioned earlier that is used during heap and arena creation and the second being a per-arena mutex that is locked prior to entry into the internal routines (those with the titles starting with _int_XXX()). These mutexes are locked shortly after entering the public routines typically when attempting to obtain a pointer to an arena. The protection provided by them is quite simply that no more than one execution context can be operating on critical portions of the code at once and thus the code is 'thread safe'.
[p[[p [p
These two sets of mutexes by and large comprise of the thread safety provided by ptmalloc2, with the second ones being the ones that we are primarily interested in.

3.2 What mutual exclusions don't provide

Mutual exclusions are good, they protect against concurrent access to critical sections of code or data, however they don't protect against assumptions written into the code base. Specifically, assumptions that expect that certain events happen in a specific order. Mutual exclusions protect against a resource from being accessed while they're in a fragile state, but they don't provide any further protection.

In ptmalloc2, we find these types of assumptions by and large in double free() protection, and to be fair these assumptions are not really the fault of the authors as at a certain point you have to throw up your hands and say 'well what did you think was going to happen?'. The problem as a whole exists less because of specific checks, but more as a result of the overall structure of how already free() chunks are stored, namely in linked lists.

In the coming sections we'll explore this idea a little more, and will expand on the idea to a point that its hopefully more clear.

3.3 GNU libc's double free() protection

3.3.1 Normal bin chunks

The first check that we find that normal bin chunks undergo while being free()'d is that it is checked against the top chunk:

DRAFT

```
if (__builtin_expect (p == av->top, 0))
  {
    errstr = "double free or corruption (top)";
    goto errout;
  }
```

In the above code, we simply have the pointer to the current block of memory being deallocated, p, and a pointer to the current arena, av. The check is specifically that the current block is not the arena's top chunk, which is what occurs if the chunk bordered the top chunk when it was free()'d the first time.

The second check that it undergoes is that a check is performed to ensure that the next chunk is past the end of the arena:

```
if (__builtin_expect (contiguous (av)
                && (char *) nextchunk
                >= ((char *) av->top + chunksize(av->top)), 0))
  {
    errstr = "double free or corruption (out)";
    goto errout;
  }
```

Here we have a simple macro that tests first to see if the arena is contiguous, if you recall from our earlier dialogue that the only way for an arena to become non-contiguous is for a rather large chunk to be allocated when there are already too many memory mapped sections and MORECORE() fails so that it calls back on mmap() again. The second part of this check is that the next chunk from the current chunk, aptly named nextchunk in this code is not outside of the arena. This situation can only occur if somehow the heap size was reduced during a previous operation.

The third check is that the next chunk has it's previous in use bit set:

```
if (__builtin_expect (!prev_inuse(nextchunk), 0))
  {
    errstr = "double free or corruption (!prev)";
    goto errout;
  }
```

Finally, that is all of the checks specifically for whether a normal bin chunk is in the middle of a multiple free condition. The checks are few, but they're generally adequate, we'll explore what conditions we need in order to potentially bypass these checks in section 3.4.

3.3.2 Fastbin chunks

DRAFT

We have to love all of the things that occur in the name of efficiency, so many operations that should be performed are not, sacrifices are made and often superior security is one of the first victims of the efficiency battle. With that all said, let's see what checks exist for fastbins. As fate would have it, there is only one:

```
if (__builtin_expect (*fb == p, 0))
  {
    errstr = "double free or corruption (fasttop)";
    goto errout;
  }
```

In the above section of code, we have p, which is the current chunk being free()'d and *fb, which is the pointer to the first chunk in the bin for that size fastbin. If you recall, when a fastbin chunk is free()'d it is not sorted and instead it is placed on the top of the list, so this check simply makes sure that the last chunk free()'d of that size is not the one we're currently releasing.

Fast:
0. the current chunk being free()'d cannot be the first chunk for that bin (i.e. free(a) free(b) free(a) is valid)

Normal:
0. Cannot get coalesced with the top chunk

1. Next chunk cannot be outside the bounds of the arena or we need to get the arena to be non-contiguous

2. Next chunks previous in use must be set

3.4 Abusing the system with this knowledge

So taking what we know, we recognize that we have to make the following conditions be true in order to free() a chunk of memory two or more times. For normal bin's we need to:

0. Our chunk on the first free() cannot get coalesced with the top chunk for that arena

1. If the current arena is contiguous the chunk after our currently free()'d chunk (the next chunk) cannot be outside the bounds of the arena,

2. In the next chunk, the previous in use bit must be set.

And for fastbin chunk's we need to ensure the following:

DRAFT

0. The current chunk being free()'d cannot be the first chunk in it's bin

So examining this list from a top-down perspective, the first thing we must accomplish is that our chunk cannot get coalesced with the top chunk, this is a pretty easy object as we need to ensure our chunk does not border the top chunk, which if we review our prior discussion on the cases that a chunk is coalesced with the top chunk, we see that this only occurs if our next chunk is (a) not currently in use and (b) is bordering the top chunk. In order to ensure this we have a couple options, (jf double check malloc code to ensure everything said here is correct [and next time you're writing about this crap make sure you're not in a Jamaican coffee shop with no inet access, jackass]) First we can just allocate a number of chunks and do our best to ensure that they are held whenever our first chunk is deallocated. This is where the beauty of threads starts to become obvious, under normal circumstances we would have to try to accomplish this is the same context as our data that causes the vulnerability, however threads typically share a heap, so we can make multiple connections to the server and allocate memory, then make another connection to cause our double free(), this is possible and easily accomplished however because of it being somewhat of a race we end up with a heap in a less defined state due to our multiple connections. While creating an exploit for these bugs this idea was considered and eventually tossed aside due to this factor instead preferring to just 'chance it' hoping that we don't end up with our next chunk next to the top. In practice it was fairly rare for one of our chunks to border the top considering the amount of memory that was allocated per request- however it is not a point to be disregarded in different situations. The next option is to ensure our next chunk is allocated or marked in use, this is a bit of a more tricky situation to cause, as allocation can cause any number of events to occur, depending on heap state the chunk can be pulled from one of the free lists, from the top chunk or under the most undesirable circumstances, from another heap. So in order to make this happen what we need to have occur, is that in between the time our first call to free() occurs and the time the second call is made, we need either a large allocation to occur that causes all of the chunks in the free lists to be too small and the top chunk to be extended and our request to be filled from there

4.0 Six million ways

4.1 Exploitation method 0: double free of vulnerability 1 where thread X invalidates thread Y's heap reference (exploitable)

4.2 Expansion on method 0, setuid()/et cetera, threads and the heap (using an unpriv'd thread to screw a priv'd thread [linuxthreads specific])

4.3 Exploitation method 1: triple free of vulnerability 1 with fastbin's (not exploitable in this instance – previously unpublished method)

4.4 Exploitation method 2: ptr = (ptr+offset) = ptr?? Double free of vulnerability 0 where multiple pointers point to the same place (should be exploitable)

4.5 Exploitation method 3: double free of vulnerability 0 where the backwards link is overwritten (exploitable??)

4.6 Anything else?

DRAFT

5.0 Conclusions & Summary

5.1 Summary

5.2 Conclusions

5.3 Thanks

DRAFT