Intro To Malloc – Exercise 2

The exercise is inside intro\_to\_malloc/exercise2/ . Move to this directory in order to start the challenge.

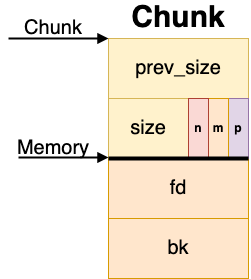
There are a several files in here but only four of them matter:

Figure 1: Malloc chunk

* birdie.c: The source code for the learning exercise.
* birdie: The compiled binary for the challenge.
* start.py: The starting point for the challenge with part of the solution already written.
* solution.py: The solution for the challenge with comments

The goal of the challenge is to use the knowledge of *ordering* in Malloc & Free to puts chunk in the proper location. This is what the mystical *heap feng shui* is all about.

Exercise Setup

In order to understand this challenge, we are going to look through the source code in birdie.c. The VM has *vim* and *Visual Studio Code* installed; use one of these to follow along with the source code.

Text

Description automatically generatedThere are only three options for this challenge. We will dive into each of these individually below:

Figure 2: Exercise options

1. Malloc – Create a chunk
2. Free – Remove a chunk
3. Show All Ptrs – Print all of the pointers that are in the list

The exercise has a list of pointers called ptr\_lst. In order to keep track of where to store the pointers within ptr\_lst, the program has two indexes: begin\_index and last\_index. Pointers are *added* to the back and *removed* from the beginning; this is a First In – First Out (FIFO) data structure, which is also referred to as a *queue*.

*Option 1 – Malloc*

Text

Description automatically generatedText

Description automatically generatedThe program has an array of pointers called ptr\_lst. This list stores all of the pointers that are created from the call to create\_malloc. The creation functionality can be seen in *Figure 3*. Once the pointer has been added to the list, the *end* index of the list is incremented by 1. It should be noted that pointers are added to the back of the list.

Figure 3: Pointer Management on Option 1

Figure 4: Create\_malloc function source code

The first action that happens in create\_malloc (shown in *Figure 4*) is an allocation request of size 0x100; this will return a chunk of size **0x110** into ptr. Following this, a call to Malloc is made that returns a chunk of size 0x120. Finally, the function sets the index (which is the storage location of the pointer in the ptr\_lst) into the second element. As seen in the code in *Figure 3*, this returned pointer gets put into the list of pointers.

The second call to malloc is simply a *filler* to prevent issues in the future with consolidation of chunks that are next to each other in the *unsorted bin*. Only focus on the first call to malloc for the purposes of the ordering in this challenge.

*Option 2 – Free*

Text

Description automatically generatedThe free operation is the opposite of a call to malloc; it *removes* a chunk from the list. The pointer being removed is the memory at the *beginning* of the list, as specified by the begin\_index variable. The call to free\_array simply prints the pointer and *frees* the memory.

Figure 5: Free (remove) option

At the very end to this option, the begin\_index is incremented by one. This moves the back of the list up a single index in order to remove the next item that we want.

A picture containing graphical user interface

Description automatically generatedIf we open GDB we can see this in action. To set this up, start the program then use option 2 (free). At this point, we have freed a single pointer from the ptr\_lst. To demonstrate this, run two commands: print ptr\_lst[0] and unsortedbin, as shown in *Figure 6*.

Figure 6: GDB operation Free

The first commands shows the first pointer in the list of pointers ptr\_lst. In this example, the pointer is ﻿0x55c761cbe010. The second option shows the *unsortedbin*. Because we have only freed one chunk, there is only one chunk in here; this chunks pointer is ﻿0x55c761cbe000. Even though the chunks are off by 0x10, they are the same chunk; one is represented as a userland *memory* pointer (ptr\_lst) and the other is the *chunk* pointer (unsorted bin). This difference can be seen in *Figure 1* above.

*Option 3 – Show All Ptrs*

The final option simply displays all of the pointers in the ptr\_lst, including the chunks that have been removed. This is useful for debugging purposes throughout the exercise but does not contribute to the solution.

The one thing to note is that the *target* of the challenge is explicitly called out on the print statement in order to make the objective clear.

*Environment & Objective*

The challenge environment is running within GLibC 2.23; this version of GLibC Malloc does not have the TCache. Because of this, the chunks of size **0x110** are being allocated are being placed directly into the *unsortedbin* once they are freed. This entire exercise only uses the *unsorted bin*.

Timeline

Description automatically generated with medium confidenceThe objective of the exercise is to get the *pointer* at index 1 in the ptr\_lst to be at index 10 in the ptr\_lst. Since pointers are reused in Malloc, this is a matter of allocating and freeing in the proper *order*. This is the beginning of *heap feng shui*.

To start the program, **six** calls to create\_malloc are made prior to the user having any control of the binary. This means that there are *six pointers* within ptr\_lst (indexes 0-5) when the user is given control over the program. It should be noted that the *bins* are empty at the start of the program because nothing has been freed yet. A visualize representation for this can be seen in *Figure 7*.

Figure 7: Challenge initial allocation setup

*Unsorted Bin Review*

Diagram

Description automatically generatedThe unsorted bin is a doubly linked list. In GLibC Malloc 2.23, all chunks above size 0x80 are put directly into the unsorted bin once freed. The unsorted bin is unique in the fact that is holds chunks of all sizes.

The unsorted bin adds chunks to the *end* of the list and removes them from the *front*. This is called a *first in – first out* (FIFO) data structure. In the real world, this is similar to a line for ice cream; the people who got in line first will get their ice cream first. In this case, the oldest freed chunk will be allocated the soonest, if there is an exact match.

Figure 8: Unsorted Bin picture

The unsorted bin *consolidates* (combines) chunks that are around itself that are free. For instance, if there is a chunk above a freed chunk, the two chunks will be combined in order to form a chunk with a combined size. In order to avoid this in the challenge, a filler chunk of size *0x120* is created between all of the chunks.

The main note is that the unsorted bin is *first in – first out* (FIFO) for allocations while the TCache and fastbin are *last in – first out* (LIFO) data structures.

*Exercise Solution*

The objective of this exercise is to get the *pointer* within index 1 to be in index 10. In order to do this, we need to allocate and free the pointers in a very specific order. This is a useful skill, as specific chunks will be desired to exploit different heap vulnerability classes in the future. Although this exercise is not an amazing exploit by itself, predicting and controlling where heap chunks will go is a major aspect of *heap feng shui*.

To solve this exercise, we will go through each operation step by step, and the intentional steps will be described as we go along.

A picture containing histogram

Description automatically generatedA picture containing timeline

Description automatically generatedThe first step is allocating to the last possible spot to get the pointer at the chunk at index 1 into index 10. This requires **three** calls to create\_malloc (option 1) in order to do this. The reason for allocating *first* is that once we *free* the chunk at index 1, we are unable to control its placement with the consecutive calls to malloc. The reason for allocating *exactly* **three** chunks is that we need the two spots (indexes 9 and 10) for later when we free *index 0* and *index 1*. This can be seen in *Figure 9*.

Figure 9: After 3 calls to malloc

Figure 10: Index 0 & 1 in the unsorted bin

Text

Description automatically generatedNext, we free **two** chunks, as described in the previous paragraph. If we do this, the front of the list will have the pointer from index 0. This is because it was freed *first*. At the end of the list will be the chunk from index 1, which is our *target*; this will be at the end because it was freed *second*. This can be seen in *Figure 10*. To see this within GDB, run unsortedbin and p ptr\_lst[1]. Notice (in *Figure 11*) that the very end of the unsorted bin and index 1 are the same value, besides the small difference between *chunk* pointer and *memory* pointer.

Figure 11: Unsorted bin and ptr\_lst[1]

Figure 12: Unsorted bin command output

Timeline

Description automatically generated with low confidenceChart

Description automatically generated with low confidenceSo, why did we make the calls to *malloc* and *free* in this particular order? If we do two *more* allocations , the *target* will be placed into index 10! To take this slowly, a single allocation will put the index 0 chunk (in the unsorted bin currently) into *index 9* of the ptr\_lst; this can be seen in *Figure 12*. This brings our *target* chunk (index 1) to the beginning of the list. Now, one final allocation will put our target chunk into index 10. The final graphical image of this can be seen in *Figure 13*.

Figure 13: One more allocation to go

The final solution has the allocation ordering of the following:

* Option 1 (malloc) three times
* Option 2 (free) twice
* Option 1 (malloc) twice

Figure 14: Aligned target chunk with index 10

An alternate solution accomplishes the same in a different order:

* Option 1 (malloc) three times
* Option 2 (free) once
* Option 1 (malloc) once
* Option 2 (free) once 🡨 Freeing the target chunk to the front of the unsorted bin
* Option 1 (malloc) once

*Conclusion*

To master *heap feng shui*, you must master *how* the allocator gives out chunks. The exercise above used a single chunk size in a single bin. With complicated programs, with different allocation sizes in different bins, this becomes more complicated and harder to follow. However, the same principle for the placement of heap chunks into the proper locations still applies.