Fix Chunk – Exercise 1

The exercise is inside intro\_to\_malloc/exercise1/ . Move to this directory in order to start the challenge.

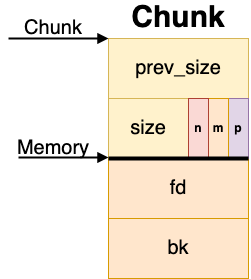
There are a several files in here but only four of them matter:

Figure : Malloc chunk

* fix\_chunk.c: The source code for the exploitable challenge.
* fix\_chunk: The binary for the exploitable 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 recently learned chunk structure information in order to ***fix*** a corrupted chunk. If the chunk is fixed properly, then the end of the program will be reached. By the end of this *exercise* you should have a better understanding of how chunks are structured in malloc.

Challenge Setup

Graphical user interface, text, application

Description automatically generatedIn order to understand this challenge, we are going to look through the source code in fix\_chunk.c. At the beginning of the program, three calls to malloc are made. 0x90, 0x100 and 0x10 are the sizes of the chunks that are requested, respectfully. Because of the need to store the metadata about the chunk, this results in a chunk of size 0xa0, 0x110 and 0x20. However, only the pointer current matters for the purposes of this exercise. The code for this can be seen in *Figure 2*.

Figure : Malloc allocations

In order to see the chunks in action, run the program with python3 start.py. When the program breaks, the chunks above would have already been allocated. In order to see the chunks, run vis\_heap\_chunks inside of GDB. You should notice that there are three chunks of the sizes described above. The image below in *Figure 3* has these values underlined in red. There is an extra **1** for the *prev\_inuse* bit of a chunk, which is why the sizes are 0xa1, 0x111 and 0x21.

A picture containing graphical user interface

Description automatically generated

Figure : vis\_heap\_chunks in exercise 1

Text

Description automatically generatedAdditionally, using the heap command will show the chunks in a more formal and organized way. This output can be seen in *Figure 4*.

Figure : Heap command exercise 1

Text

Description automatically generatedWhen a chunk is returned by malloc it returns the *chunk address* + 0x10. This is because the first *0x10* bytes are the ***size*** and ***prev\_size***, which are always on the chunk and have no reason to be accessible to the user. So, when the chunk is allocated, the allocator *adds* ***0x10*** to the address and returns this. This difference can be seen in *Figure 1* with the difference between a *chunk* (what malloc sees) and *memory* (what the user program sees).

Figure : Overwrite the chunk metadata

In the exercise, the program sets chunk to be current – 0x10. This is because we want to access the *chunk* pointer instead of the *memory* pointer to overwrite the size.

With the pointer set to the *chunk* address, we write to this address. The *prev\_size* and *size* of the chunk can be altered at this location. The source code for this can be seen in *Figure 5*.

Text

Description automatically generatedFinally, after the write occurs, the pointer that we just edited is freed. When this pointer is freed, it is given back to the allocator as the *memory* pointer. This is because *malloc* will convert the pointer from a *memory* pointer to a *chunk* pointer at the beginning of the call to free. If the chunk is properly constructed, then the call to free will succeed. Otherwise, the program will either segfault or abort from built in LibC protections.

Figure : Freeing the modified chunk

With the structure of the program out of the way, we can go into the solution for this challenge.

Solution

Graphical user interface, application

Description automatically generated with medium confidenceAs a quick overview of the challenge:

Figure : Quick synopsis of the code

1. Allocates a chunk via a call to malloc.
2. Writes to the *chunk* pointer, which includes the *prev\_size* and *size*.
3. The chunk is freed via a call to free.

If the chunk is written to properly, the program will finish, denoting that the challenge has been solved! Otherwise, the program will crash. The short version of the code discussed above can be seen in *Figure 7*.

The current *prev\_size* and *size* have been set to **0**’s via a call to memset. So, simply continuing the program without doing anything else would result in a segmentation fault or *LibC Abort*. This is because the ***size*** is invalid in the beginning of the call to free, which results in the error message: free(): invalid pointer: <address>. The code that triggers this error message in LibC can be found at [here](https://elixir.bootlin.com/glibc/glibc-2.23/source/malloc/malloc.c#L3861) at the beginning of the free function. Whenever there is a LibC abort, searching the *Malloc.c* source code is the best way to find *where* and *why* the error occurred at. Searching for a specific error message (without the address), is a good way to find the reason for the crash.

In order to fix the chunk, we need to understand what it originally looked like. The current chunk can be seen in *Figure 4* as the second chunk that is listed. The chunk has a size of **0x110** and has the **prev\_inuse** bit set. This makes sense because the previous chunk is in use (denoted as previous).

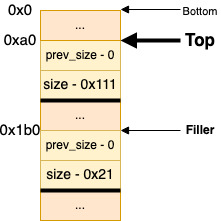
Because the previous chunk is in use, the *prev\_size* field is not used for this chunk. To verify this, the *purple* chunk (second chunk) in *Figure 4* is the current chunk; this shows the raw memory of the chunk. The *prev\_size* field is set to all 0’s in the image, which indicates that the field is not in use.

Figure : fake chunk implemented

If we put all of this together, we write all 0’s for **8 bytes** for the *prev\_size*. Then, write the size of the chunk, along with the *prev\_inuse* bit, which is **0x111**. This can be seen in *Figure 8*.

A screenshot of a computer screen

Description automatically generated with low confidenceBecause we are writing bytes instead of numbers, we will need to use *pwntools* p64 in order to encode this memory properly. This function takes a number and *packs* it (converts the integer to a little-endian string).

In the start.py file, edit the only TODO in the entire file. The *0x0* should be replaced **0x110**.This is because of the sized discussed above (shown in *Figure 7)*. Additionally, the *prev\_inuse* bit is already set for you, so only **0x110** needs to be written.

Once the size has been written properly, the program should finish with no problems. At this point, the exercise has been solved!

