Overlapping Chunks Exercise 1

A picture containing timeline

Description automatically generatedThe challenge is inside overlapping\_chunks/exercise1/ . Move to this directory in order to start the challenge.

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

* overlap.c: The source code for the exploitable challenge.
* overlap: The binary for the exploitable challenge.
* start.py: The starting point for the challenge with part of the solution already written.

The goal of the challenge is to use the recently learned ***overlapping chunks*** technique to overwrite a string on the heap. We will work backwards from the goal in order to exploit this. To start the challenge, run python3 start.py. This file has a templatefor solving the challenge with helpful comments.

Figure : Malloc chunk

Exercise Setup

Text

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

At the beginning of the program, three calls to malloc are made of different sizes, which can be seen in *Figure 2*. Because malloc needs to round up to the next grouping of 0x10 on 64-bit, malloc(0x30) returns a chunk of size **0x40** for the variable first. For the same reason, the chunk size of second is **0x100** and the chunk size of third is **0x50**.

Figure : Initial allocations

After the allocations are made, first is written to by a call to *fgets* with a maximum of ***0xa0*** bytes. This write can be any value besides a newline (\n), since this stops the input of the function.

Figure : Writing to the 'first' pointer

Graphical user interface, text

Description automatically generatedThe next part of the program is a few calls to heap specific operations. First, the pointer second is *freed*. Directly after the call to free another call to malloc is made. This time, the chunk returned is of size **0x150**. Naturally, a call to *fgets* follows that writes 0x140 bytes to the pointer.

Figure : Freeing and more allocating

At the end of the program, the data inside *third* is printed out. Earlier in the program, the string *Bill Gates* is copied into this buffer. The **goal** of the exercise is to corrupt this string with the text *Steve Jobs*.

Figure : Final print of the program

The challenge uses GLibC 2.26. Because of the chunk sizes above and the version of LibC in use, we are exploiting the TCache bin. This attack can be used on other bins though.

Vulnerability

In *Figure 3* a call to *fgets* is made with a size of **0xa0** that writes to the pointer first. Because the allocated user memory size is only **0x40** (with a chunk size of 0x50), this creates a *heap* *buffer overflow* within the heap memory; the overflow is by 0x60 bytes. This overflow is large enough to corrupt the metadata of second (chunk metadata, such as *prev\_size* and *size* can be seen in *Figure 1*) but not large enough to corrupt any part of the memory in third.

Text

Description automatically generatedThe vulnerability can be seen by using GDB with a *before & after* style. The *before* is viewing the heap prior to the corruption. The *after* is seeing the following the corruption.

Figure : Chunk setup in memory

To see the *before*, start the program (as shown above). On the initial starting breakpoint, run the command heap in GDB. This shows all of the chunks on the heap, which can be seen in *Figure 6*. Going from top to bottom of the image, we can see chunks of size 0x250, 0x40, 0x100, 0x50. The latter three chunks are the sizes discussed in *Exercise Setup* section above. However, the chunk of 0x250 is for the TCache bin itself, which gets allocated on the first call to malloc on a specific thread. This allocation can be ignored for this challenge.

Text

Description automatically generatedTo see the *after*, change the first TODO in the *start.*py file to be **0x90**; this sets the amount of bytes to be written as 0x90. From the previous starting point, use the continue command to allow the program to continue executing. This will overwrite the *size* of the chunk second with 0x4141414141414141; this is “A” in hex when shown as a byte in ASCII format. When using the heap command again (as seen in *Figure 7*), the size of the second chunk is all “A”s! At this point, we know that we have corrupted the size of the second chunk.

Figure : Chunk setup after corruption

The program only stops because of an intentional **abort** byMalloc. In this case, the program crashes because of a GLibC sanity check realizes that something has gone terribly wrong. But, this could also be a *segfault* as well. When Malloc *aborts*, it sends out nice errors messages on *where* the program crashed at, which are useful for debugging exploits. In this case, the error message is double free or corruption (out), which goes to the line [4284](https://elixir.bootlin.com/glibc/glibc-2.26/source/malloc/malloc.c#L4284) within GLibC Malloc. All the *abort* messages in Malloc are unique; so, simply searching for the error message within Malloc.c will find the crashing section for you. The code snippet Graphical user interface, text, application

Description automatically generatedfor this crash is shown in *Figure 8*.

Figure : Malloc abort call

The validation in *Figure 8* is being done on the chunk size of the *second* chunk being freed. While this call to free is taking place, the address of the *nextchunk* is validated to not be beyond the end of the heap. Because the *nextchunk* pointer is calculated based upon the **size** of the chunk being freed (*current chunk address + size of chunk = nextchunk*), this causes the error path to be taken because the size of the chunk 0x4141414141414141 is much larger than the address of the heap. Hooray! This confirms our suspicion that we corrupted the size of the chunk by both the chunk being corrupted via the heap command, the memory addresses and the *abort*.

Size Corruption

In order to use the *overlapping chunks* attack, we need to *overlap* chunks together. The best way to do this is to *extend* one of the chunks to be over the top of another chunk by *editing the size*. We have the ability to corrupt the *size* field of a chunk, as discussed in the previous section. There are two main problems to solve with corrupting the size:

* Amount of bytes prior to the overwrite
* The *size* to use for the chunk

In the example above, we overwrote the *size* of the chunk with a chunk of A’s. But we want to overwrite this with a controlled size value. Although we could simply *spray* the *chunk size* to overwrite it without knowing the proper offset, it helps to understand the memory layout for the exploit later.

The *memory* with the buffer overflow vulnerability is the first chunk, which has a chunk size of 0x40, as seen in *Figure 9*. Because the writing starts from 0x10, a buffer overflow occurs at the write of the 0x30th byte. The *size* field (as shown in *Figure 1*) is at an offset of *0x8*. So, in order to overwrite the *size* field of the second chunk, we must write *0x30 + 0x8* or *0x38* bytes of filler. Now, we can properly overwrite the *size* field of the second chunk. At this point, we know *where* to overwrite but not *what* to overwrite.

Figure : Chunk sizes for the first three chunks

Diagram

Description automatically generatedThe *what* to overwrite depends on the rest of the program. Even though we can corrupt the size of the chunk to put it into a *bin*, if we do not have an allocation to take the chunk of the specific size, it is useless. The only other direct call to malloc is the pointer to fourth. This allocation uses a *size* of **0x150** at the end of the program. Because this is the only other call to malloc and it is larger than the chunk we are corrupting (0x100 – second chunk as seen in *Figure 8*), this is the perfect chunk size to use! Because this new allocation is of size 0x150 and the second chunk is only 0x100 in size, this results in a **0x50** buffer overflow into the third chunk. The result of this *fake chunk* overlap can be seen in *Figure 10*.

Figure : Faked chunk size

**0x150**

The modifications to *start.py* are shown below as well:

fake\_chunk = b"A" \* 0x38 # Filler bytes prior to size

fake\_chunk += p64(0x150) # Fake chunk size

Freeing & Allocating the Chunk

Above discusses the *theory* behind overwriting the size of the chunk and how it should work. The code above simply *overwrites* the chunk size of second but never *allocates* it. This section goes over the allocation and debugging steps for this.

The fake size needs to be ***0x150*** because it is the only memory allocation left in the program. Additionally, 0x150 is larger than the original chunk, which creates an *overlap* between the two chunks. However, the chunk is still *allocated* in the second pointer. This is an issue because developers do not use the *chunk size* to determine the size of the chunk; this chunk size is only used by GLibC Malloc and is specific to this allocator.

Text

Description automatically generatedIn order to get this chunk used, we need to free it first, which can be seen in *Figure 4.* Once the chunk (second) is freed, it will be put into the **0x150** bin. The TCache step only validates that the chunk size is within the range of the TCache Bin (0x20-0x410) and does no other validations about the chun­k. Because 0x150is within the range of a TCache Bin, the chunk is inserted into the TCache Bin once it is freed!

Figure : TCache Bins chunk output

In order to see this, we can put a breakpoint on the fourth malloc, which is directly after the call to free on second. The command to do this is b malloc, after GDB opens up for the exercise. From this stage, we want to get to the call to free. So, continue the program (c). The program should stop after the call to *free* but before the call to malloc in *Figure 4.*

To view the chunk itself, we want to display the TCache Bins. This can be done via a call to tcachebins within GDB. As seen in *Figure 11*, this chunk has a size of 0x150 as we expected and it free within the TCache.

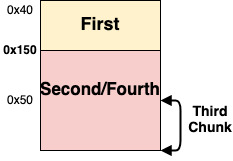
A screenshot of a computer

Description automatically generated with low confidenceNow that the corrupted chunk is within the bin, we need to get it out to cause some havoc. To remove the chunk from the TCache bin, the final allocation of a chunk of size **0x150** is used, which can be seen in *Figure 4*.

Figure : Allocating the corrupted chunk

To know if the proper chunk is allocated, GDB can be used. However, an even easier way is to simply look at the output of the program. The program prints out the *pointers* of the chunks being created throughout. If the chunk was taken out properly, then *Second* and *Fourth* (underlined in red in *Figure 12*) should both have the same pointer. Now, we have a call to *fgets* where the expected chunk size is **0x150**, even though (as shown in *Figure 9* and *Figure 12*) the chunks real size is **0x100**. This creates a dangerous buffer overflow of **0x50** bytes! ☺

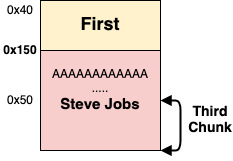
For the Win

The objective of this challenge is to overwrite the string within the chunk third that contains the string *Bill Gates*. In the previous step, we allocated a chunk that overlaps with the chunk third. Using the **0x50** byte buffer overflow, we will overwrite the string with the value of *Steve Jobs*, who is clearly superior. The new question is to figure *how many bytes until the string* from the chunk fourth with the call to *fgets*?

The chunk second/fourth overlaps with the chunk third, as seen in *Figure 12.* The slot for the *second* chunk is only 0x100 in size and should only allow a write of **0xf0.** This is because of the difference between the *chunk* pointer and the *memory* pointer shown in *Figure 1*. The application sees the *memory* pointer, which is 0x10 bytes *more* than the *chunk* pointer, in order to do the write.So, a write of 0xf0 bytes, we come to the beginning of the chunk third.

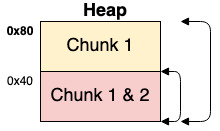
Figure : Overlapping chunk allocated

Finally, we need to account for the *prev\_size* and the *size* of the third chunk as well; this is an additional 0x10 bytes because each of the fields is 0x8 bytes in length. By adding the length from the beginning of the fourth chunk to the string within the third chunk, this is **0x100** bytes in length.

After using 0x100 bytes as the filler, we can successfully write *Steve Jobs* in order to overwrite the string within the third chunk. To confirm this was done properly, the output (from the ending print statement of the string) should say *Steve Jobs* instead of the usual Bill Gates.

The final TODO of the binary is the *amount* of bytes of filler to write. The filler value is *0x100*, as discussed above.

Figure : Overlapping Chunks



Final Words

When looking at the picture of a chunk (such as *Figure 1*), the first attack that comes to mind is to corrupt the size of the chunk to cause havoc. Even though this technique is one of the first attacks that people think about, the technique can be used with relative ease to cause major memory corruption.

Figure : Overlapping chunks

The TCache Bin does not do any size validation checks on chunks and is the most likely bin a chunk will end up in; these two reasons are why it was the primary focus of this module. The fastbin, unsorted, largebin and smallbin all have simple validation checks that can be bypassed by putting bytes in specific locations. The technique is exactly the same beside the *point in time* to corrupt the chunk (while in the free bin or prior) and the validation checks that need to be bypassed.