House of Force – Exercise 1

The exercise is inside house\_of\_force/exercise1/. Move to this directory in order to start the challenge.

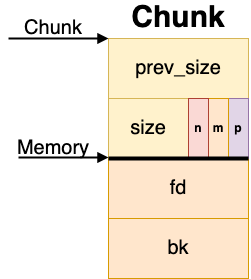
There are a several files in here but only four of them matter:

Figure 1: Malloc chunk

* shining\_moments.c: The source code for the exploitable challenge
* shining\_moments: 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 recently learned **house of force** exploit technique in order to solve the challenge. We will use this technique in order to overwrite the \_\_malloc\_hook or puts GOT entry in order to pop a shell.

Challenge Setup

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

Text

Description automatically generatedAt the beginning of the program, a call to freedome is made. This calls sole purpose is to leak addresses that are useful for the exploit down the road. Besides this, this function can be ignored.

Figure 2: String declarations

Text

Description automatically generatedNow, the bulk of the main program begins. It starts by defining three strings (pointer to chars) are declared, each with a specific label in purpose (note the comments). This can be shown in *Figure 2*. These are the only chunks created throughout the program that matter to us.

Figure 3 (Step 1): Use after free on 'first'

From there, our first call to malloc is made and freed directly after. Since this is the only chunk on the heap and the size is quite large, this is put back into the top\_chunk. This freed chunk is written to directly afterwards, which creates a *use after free* vulnerability on the chunk first. The write occurs at *first-*0x10. This allows us to write to the *prev\_size* and *size* of the chunk directly, as seen in *Figure 1*. All this code can be seen in *Figure 3*. There will be more on this later though!

A screenshot of a computer

Description automatically generated with medium confidenceAfter the initial allocation, the program asks us to choose a size for the allocation. The function strtoull converts our string input into an unsigned long long. This is then passed in for the allocation of a buffer. All of this code can be seen in *Figure 4*.

Figure 4 (Step 2): Choosing the size of the allocation

Text

Description automatically generatedThe next section allocates a chunk of size 0x20 via malloc and writes to it with a call to fgets.

Figure 5 (Step 3): Final controlled allocation with write

Finally, a call to puts and malloc(0x0) are made to end the program.

Prior to starting the exercise, a few things should be noted:

* The challenge is running within GLibC 2.23. This means that the TCache does not exist and a plethora of techniques are unpatched.
* Calls to *free* after using this technique put the program into a major risk of crashing.

Vulnerability

The vulnerability is mentioned above: a *use after free* (UAF) in the handling of the chunk first.

Since chunks are given back to the user with a 0x10 offset (referenced to as the *memory* pointer in *Figure 1*), this puts a UAF at the *fd* pointer on the top\_chunk. The top chunk only uses the size of the chunk though, making this UAF not very useful. In order to overwrite the top chunk, the write occurs at first – 0x10 as a result.

Graphical user interface, text

Description automatically generatedCausing this to crash is only possible when pairing it with a bad allocation size. By giving a bunch of As and a large number, this will crash the program. For instance, starting with “AAAAAAAAAAAAAAAA” as the input string and 0x111111111 as the input size will crash the program. But why?

Figure 6: Removing memory from the top chunk

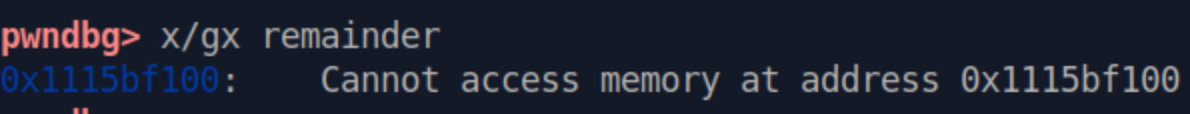
The program crashes at line 3801 in GLibC Malloc 2.23 at the set\_head instruction, as shown in *Figure 6.* This instruction is intending to set the size of the new top chunk address, after we shrank it for our new chunk. However, if we read the remainder variable, this memory address does not exist! This can be seen in *Figure 7*.

Figure 7: Remainder pointer is invalid

This indicates that we have corrupted the top\_chunk into a state where it thinks that it is larger than it really is! This is the perfect scenario to use the *house of force*.

When deciding to use the top chunk, there is an if statement that validates that the top\_chunk is large enough to handle the allocation. If the chunk is large enough, it handles the allocation by taking part of the top chunk. The *reason* for the top chunk being in this corrupted state will be discussed below.

Text

Description automatically generatedCorrupting the Top Chunk (Step 1)

Figure 8: Shrinking the top chunk

We have a vulnerability that allows us to overwrite the top chunk size. But, why does this matter? To understand *why* this is impactful, we will read the source code for GLibC 2.23 Malloc.

The source code in *Figure 8* is the code for using the top chunk inside of \_int\_malloc. nb is the allocation request size from the user, the size is the size of the top chunk. On line *3794*, the top chunk size is checked to see if it is large enough to fit the requested memory from the user. If so, it goes into the *if* statement.

Diagram

Description automatically generatedA picture containing diagram

Description automatically generatedOn line *3796*, the remainder size of the top chunk is calculated. Then, the chunk\_at\_offset macro takes the victim (top chunk) and adds the allocation request size to it. Now, this address is the location of the top chunk *after* it has been shrank. Finally, line *3798* sets the top chunk to be this new address.

Figure 9: Top chunk prior to shrink

Figure 10: Top Chunk after shrink

The code after this is the set\_head function shown in *Figure 6* that changes the size of the top chunk. Eventually, the memory removed from the top chunk is given back to the user as usable memory. A visual representation can be seen in *Figure 9* and *Figure 10* for the chunk shrinking functionality.

With these details in mind, the initial crash makes much more sense! We corrupted the top chunk size, which meant that the code at line *3794* for validating the size is not valid anymore. Hence, when we move the top chunk by a large value, the set\_head macro to write the chunk size is to invalid memory.

Now, if we can move the top chunk to this random spot in memory, can we control where the top chunk goes? Yes we can! This primitive is achieved by *moving* the top chunk pointer to anywhere we want in memory by controlling the allocation size. By crafting the allocation size just right, we can move the top chunk to an *arbitrary location in memory*. Damn, that’s an amazing primitive!

Diagram

Description automatically generatedWhat’s the actual fake size we should use then for the top chunk? It turns out that our random input (16 A’s) works fine for the input. The first 8 bytes are the *prev\_size*, which the value does not matter for. Then, the *size* of the chunk is the next 8 bytes; this is the only value of the top chunk that we care about. Although 8 A’s works for this input, we want to make this as large as possible! Hence, the ideal fake size is -1 (unsigned this is the same as 0xFFFFFFFFFFFFFFFF ) because this makes the top chunk large enough to go anywhere in memory. For the purposes of this exercise, we will use -1 for the fake chunk size. For the input of the first TODO this should be “\xff” \* 8 because this is being packed into a string. The top chunk overlap with another location (LibC) can be seen in *Figure 11*.

Figure 11: Top chunk size overlapping with LibC

Finding a Good Target

The top chunk is corrupted and we want to move the top chunk to somewhere in memory to compromise the program. What makes a good target though?

This attack is a fairly one-shot primitive. Once you corrupt the heap and put it into this weird state, free has a really hard time functioning. As a result, we need to take control of the program fast after we use this technique.

At the end of the program, there are two seemingly random function calls: malloc(0x0) and puts(“dynamite failed”). These functions are amazing targets for this overwrite though, as they contain beautiful function pointers for us to use.

The first good target is the \_\_malloc\_hook. The malloc family of functions can be *hooked* at runtime. This is done when using *MCheck* for memory corruption or *MTrace* for performance analysis of the heap. But, since these are always in the program, the malloc hook family of function pointers make great targets to attack.

The second option we can do is compromise the call to puts. The binary is compiled without full RELRO. As a result, we could overwrite the global offset table (GOT) entry of puts to gain control over the flow of the program. Although this seems odd since the .bss section is behind the heap, the top chunk can overflow to wrap around to this section of memory. For more information on the GOT, please read the slides and solution for the unlink exercises.

The \_\_malloc\_hook is the route that the proof of concept will take for this, even though the puts GOT overwrite works just the same. The RET pointer on the stack could work as well, depending on if the stack cookies are turned on.

Allocating Close to the Target (Step 2)

Even though this is called an *arbitrary write* primitive, a more accurate representation is a *pointer move* vulnerability. At this point, we are inthe source code in *Figure 4*.

With the corrupted top chunk size that overlaps with everything else in memory, our goal is to move the top chunk pointer to our target, which is the \_\_malloc\_hook. The result of this can be seen in *Figure 12*, with the original in *Figure 11*.

This is done by shrinking the top Diagram

Description automatically generatedchunk via a call to malloc with a specific size that will place our pointer to our target location. But what is this size?

Figure 12: Moving the pointer (step 2)

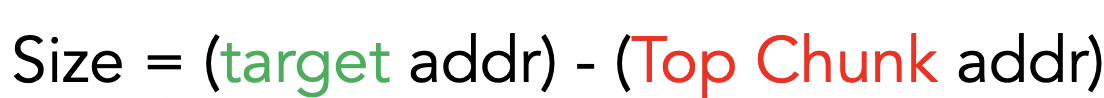
Our goal is to move the top chunk address from the heap to directly before our target in LibC. Our only controllable input is the *allocation request size*. Since the *request size* moves the pointer *request size* (nb) bytes from the original top chunk, we just need to know the ***difference*** between the top chunk and the target address. In the real world, this would require an information leak but this has the leaks done for you.

Figure 13: Allocation size algorithm

In simple terms, we are finding the difference between two points then adding this difference to get to the end from the original point. By subtracting the address of the target location by the current top chunk address, we find the ***difference*** between the two points. Then, when we make the allocation as this ***difference***, our pointer will move from the top chunk address to our target location, which is the \_\_malloc\_hook. The partial formula is shown in *Figure 13*.

But, there is one small problem though: if it is on top of our target, then we cannot overwrite it afterwards. In order to compensate for this, we also need to subtract **-0x20** bytes from the allocation. This is because we need to consider the metadata in the allocation when moving *close to the target* (as 0x10 will be added to the request size) in step 2 and the allocation for the overwriting our target in step 3. Hence, the full formula is Size = Target Address – Top Chunk Address – 0x20

The actual code to perform this in the start.py file is shown below. It should be noted that the variable malloc\_hook holds the address of the \_\_malloc\_hook and the top\_chunk\_loc holds the address of the top chunk. The variable size is then used for the allocation size.

size = malloc\_hook – top\_chunk\_loc

Overwriting the Target (Step 3)

Moving the pointer directly in front of our target is the hard part. Now, we can reap the fruits of the labor above, to hijack the control flow of the program.

The final call to malloc in *Figure 5* will return a chunk directly over the top of the \_\_malloc\_hook function pointer. With the call to fgets, we can write to this address! Since we have an auto-shell with the pop\_shell function, we will write this. When the call to malloc(0x0) is made, our hook will be used, and it will pop a shell for us!

Conclusion

The House of Force is an amazing primitive and is personally one of my favorite heap exploitation techniques. In order to use this exploit, you need to be able to corrupt the top chunk size and control the size of an allocation to hop over larger unmapped sections of memory. May the force be with you Luke.