Vuln Classes – Exercise 1

The exercise is inside vuln\_classes/exercise1/ . Move to this directory in order to start the challenge.

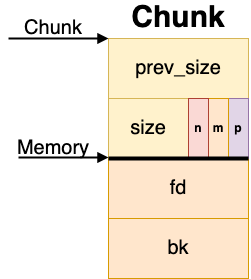
There are a several files in here but only four of them matter:

Figure 1: Malloc chunk

* high\_score.c: The source code for the exploitable challenge.
* high\_score: 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 **use after free** (UAF) vulnerability class in order to solve the challenge. In particular, the user is supposed to get the *high score* on the binary using a UAF. By the end of this *exercise* you should have a better understanding of how the *use after free* vulnerability class works.

Challenge Setup

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

Text

Description automatically generated

Figure 2: Challenge options

First, we will go through all of the options in the exercise, as seen in *Figure 2*.

Graphical user interface, text

Description automatically generated**Option 1:** In order to start the game we need to create a *player*. This can be done with *option 1*. When creating a player, a call to malloc(0x10) is made for the *player* struct (*Figure 3*). This creates a chunk of size **0x20**. The chunk size is 0x20, instead of 0x10, because the *size* and *prev\_size* of a chunk (*Figure 1*) needs to be added to the chunk.

Figure 3: Player struct

After this allocation, the user is asked for a name that can only be 12 bytes. This is stored in player->name , which is a global variable that stores the information of the user. Finally, the player->score is initialized to be 0. The *player* struct is only 16 bytes (0x10) in size but a **0x20** sized allocation is used.

Text

Description automatically generated**Option 2:** The second option is to **free** the player, which is intended to act as a deletion operation. This makes a call to free on the *player* struct allocated from before. Nothing else is done on this operation. This can be seen in *Figure 4.*

Figure 4: Option 2 - Deleting a player

Graphical user interface, text

Description automatically generated**Option 3**: Option number 3 is all about playing the game, which can be seen in *Figure 5.* Prior to playing the game, a call to malloc(0x10) is made for an array of integers called p\_array. This is then passed to the function add\_score in order to be used.

Figure 5: Option 3 - Play the game!

The function that generates the score sets the values of p\_array to be 100, 101, 102 and 103 respectively. After this step, a random number is taken between 0 and 15. Finally, the random number is added with the p\_array value subtracted by 90 on all of the numbers in the *p\_array.* The maximum score is 100 after doing all of these operations.

At the end of the function, the score returned from the function is set into the local variable *new\_score.* In option 4, the recently achieved score can be stored in the *player* struct, if this is the score that player selects. Finally, the p\_array variable is freed.

Graphical user interface, text

Description automatically generated**Option 4:** Set the score of the user. In option 3, the game is played, and the score is stored in a local variable called *new\_score*. However, this score is not used to validate if the user has won the challenge. Instead, option 4 sets *player->score* to the *new\_score* variable. It should be noted that this option is not needed in order to solve the challenge.

Figure 6: Set the stored high score

**Option 5**: Check to see if the user has gotten the high score! The player->score is validated to be higher than the current high score. So, what is the starting high score? 101! However, the maximum possible score is 100, making this game impossible to win without using some other way.

In order to beat the game, the player->score must be larger than 101. This can only be done by using a vulnerability in the program!

Vulnerability

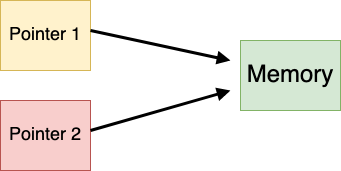
A use after free vulnerability is when a previously freed pointer is being used; hence, the name *use – after – free.* Eventually this can lead to two pointers referencing the same memory or can be used to directly alter parts of the chunk’s metadata, such as the *fd pointer*. The former exploitation method can be seen in *Figure 7.*

Figure 7: Two pointers on the same address

When attempting to find use after frees, it is important to look at calls to *malloc* and *free*. A use after free can only occur if a piece of memory is *freed* but used later. So, finding this vulnerability is all about the heap memory specific operations.

In the program, there are only two calls to *malloc* and two calls to *free*. Both calls to *malloc* produce a chunk of size **0x20**. One *malloc* and one *free* are for creating and freeing a player. The other two calls are for p\_array.

On option 3, the p\_array pointer being used is allocated and freed, then never used again. The handling of the p\_array variable is handled securely because of this, with no use after free vulnerability; this can be seen in *Figure 5.*

However, if we look at the program, there is a use after free vulnerability lurking! After *freeing* a player, the pointer is never set to NULL or erased (see *Figure 4* for the source code). Because of this, the *player* variable has a *use after free* on all other operations after the call to free. To trigger the UAF, run option 1 (create), then option 2 (free). This will have a pointer to a free chunk still within the *player* variable.

A screenshot of a video game

Description automatically generated with medium confidenceUsing GDB, this *use after free* vulnerability can be spotted. In order to see the vulnerability in GDB we need to trigger the vulnerability; this can be done by using option 1 then option 2. Now, we can use the amazing *pwndbg* in order to see the pointer in the *bins* and within the *player* struct variable. In *Figure 8*, the first command tcachebins shows all chunks within the TCache. The address of this chunk is 0x203a280. The second command print player displays the address of the *player* struct pointer, which is also 0x203a280. Because there is still a reference to this pointer in the *player* variable and this chunk is within the *malloc bins*, we have a clear use after free vulnerability!

Figure 8: Use after free in GDB

Exploiting the Use After Free

When exploiting use after frees, there are two main ways to go about it:

1. Allocate the chunk then abuse the *double* pointer usage
2. Use the chunk metadata directly for attacking

Table

Description automatically generatedBoth attacks work quite well, depending on the situation. In this exercise, we will use the first exploitation type in order to attack the binary.

Figure 9: Two pointers to same memory

After using the vulnerability described in the section above, the *player* pointer is in **both** the TCache 0x20 bin and within the *player* pointer. Besides the create player call, there is only one other call to *malloc*: the p\_array pointer when playing a game.

When playing a game, a call to malloc is made with the same size chunk as the *player* struct. So, if we play a game, then the pointer used for p\_array will be the same chunk as the *player* variable! This is because the chunk is in the TCache **0x20** bin and is there is a stale reference for the *player* variable. This creates a scenario, such as *Figure 7,* where we have two pointers to the same spot in memory.

The overlap can be seen in *Figure 9*. The *player* struct uses the first 12 bytes for the name of the user. The final 4 bytes are for storing the **score** of the user, which is our target for corruption. For the p\_array (as described in the *Challenge Setup* section for Option 3), the values 100, 101, 102 and 103 are stored when calling the *add\_score* function.

Because the value 103 in p\_array lines up with player->score perfectly, the score has been overwritten with 103 when the writing happens within *add\_score.* Finally, it should be noted that a call to free does NOT automatically erase memory. So, when the p\_array is freed, the values 102 and 103 remain!

Text

Description automatically generated with low confidenceThe description above is only using option 3. So, the chain is now below:

Figure 10: Player variable overwritten score

1. Option 1
2. Option 2
3. Option 3

In order to see the exploit in action, run the command sequence above. Then, in GDB, print out the contents of the *player* pointer with print \*player . The score is now set to 103, even though this score should be unobtainable! This can be seen in *Figure 10*.

Text

Description automatically generatedFinal Touches

Figure 11: getting the flag

Now that the player->score has been overwritten with 103, all we need to do is check to see if we obtained the high score. Because 103 is greater than 101, this will pass! This can be done by selecting option 5.

The full sequence of events for the challenge is as follows:

1. Option 1: Create the player
2. Option 2: Delete the player, causing the use after free
3. Option 3: Play the game, which exploits the use after free
4. Option 5: Check to see if you won the game.

There is an additional way to exploit this; using a double free vulnerability. However, that is the next challenge ☺ If you are feeling extra adventurous, come back to this challenge after going through module 3 (Fd Poison). This can be used in order to control the flow of execution to *jump* to simply print the flag out or pop a shell!