Vuln Classes – Exercise 2

The exercise is inside vuln\_classes/exercise2/ . Move to this directory in order to start the challenge.

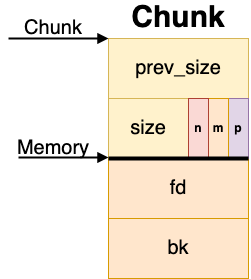
There are a several files in here but only four of them matter:

Figure 1: Malloc chunk

* 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 **double free** 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 *double 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 the options in the exercise, as seen in *Figure 2*. These options are the same as *exercise 1*. The only difference is that there is a *use after free* protection put onto this exercise.

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) for the *player* struct (*Figure 3*) that 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.

Finally, there is a *use after free* (UAF) protection that is set; this variable, is\_player\_active, is set to 1. If the variable is set to 1, then the function *add\_score* (called in Option 3) will work. Otherwise, if the variable is set to 0, then the the call will not work.

**Text

Description automatically generated with medium confidence**

Figure 4: Option 2 - Remove the player

**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.

After the user is freed, the is\_player\_activevariable is set to 0. This is used in order to prevent usage after something has been *freed*. This can be seen in *Figure 4.*

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.

It should be noted that this action can only be performed ***if*** the variable is\_player\_activeset to 1. This prevents a UAF vulnerability from using this option.

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.

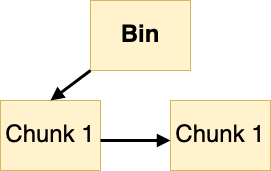
Figure 6: Set the stored high score

**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 order to do this!

Vulnerability

A double free vulnerability is when a pointer is freed multiple times; hence, the *double – 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.*

When attempting to find double frees, it is important to look at calls to *malloc* and *free*. A double free can only occur if a piece of memory is *freed* then *freed* again. So, finding this vulnerability is all about the heap memory specific operations; in particular, calls to free.

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 double free vulnerability lurking! After *freeing* a player, there is not check for the usage of the pointer prior to be freed again. In particular, the pointer is never NULLed out and the is\_player\_activevariable is not validated prior to freeing the variable.

Because of this, the *player* variable has a *double free* vulnerability. To trigger the vulnerability, run option 1 (create), then option 2 (free), option 2 (free) again. This will have a pointer to a free chunk still within the *player* variable. A visual of this can be seen in *Figure TODO.*

A screenshot of a video game

Description automatically generated with medium confidenceUsing GDB, this *double 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 **twice**.

Figure 7: Double free in GDB

Now, we can use the amazing *pwndbg* in order to see the pointer in the *bins*. In *Figure 8*, the command tcachebins shows all chunks within the TCache. The address of the first chunk in the bin is 0x434280. Then, the second chunk in the bin is also 0x434280. Because the same chunk is in the TCache bin twice, this indicates that there is a *double free* vulnerability!

Exploiting the Double Free

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

1. Free the same chunk into the bins ***twice***
2. Allow the chunk to be reallocated, then free the chunk at this point to artificially create a use after free

Both attacks work quite well; however, the second option is more common in real world exploitation because of modern double free protections built in to LibC. In this exercise, we will use the first exploitation type in order to attack the binary.

Diagram

Description automatically generatedAfter using the vulnerability described in the section above, the *player* pointer is in the TCache 0x20 bin **twice**. This can be seen in *Figure 9*.

In order to exploit this *double free* vulnerability, the goal is to allocate *one* chunk into one variable and allocate the other chunk into *another* variable. This creates a classic *use after free* where two pointers are on the same location. Which calls to *malloc* are possible to do this with?

Figure 8: P\_array and player pointers

Table

Description automatically generatedBecause there are only *two* calls to *malloc* in the programming (both of the same size), this is an easy choice! We want to allocate one of the pointers in the *bin* to the playerstruct and the other to p\_array. This way, we create a situation where two pointers are at the same address, such as the UAF exercise. The ending diagram for this can be seen in *Figure 8*.

Figure 9: Overlapping player and p\_array

From this point on, the exploit method is the same as *exercise 1.* 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. 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 p\_array in 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! This setup can be seen in *Figure 9.*

To create the *use after free-like* scenario, the following sequence of operations needs to be taken:

1. Option 1: Create the player
2. Option 2: Delete the player
3. Option 2: Delete the player (**again**)
4. Option 1: Create the player
5. Option 3: Play the game

Text

Description automatically generated with low confidenceIn 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*.

Figure 10: Player variable overwritten score

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
3. Option 2: Delete the player (**again**)
4. Option 1: Create the player
5. Option 3: Play the game
6. Option 5: Check to see if you won the game.