Fd Poison Challenge 1

The challenge is inside fd\_poison/challenge1/ . Move to this directory in order to start the challenge.

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

* fake\_fd.c: The source code for the exploitable challenge
* fake\_fd: 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 ***fd poison*** attack to return the address of important\_string to the final usage of malloc. We will works backwards from the goal in order to exploit this.

Vulnerability

At the beginning of the program two calls to malloc are made that return two 0x30 sized chunks. The first pointer is data and the second is data2 . After the allocations, both of the pointers are freed via a call to free. When the chunks are freed, they are put into the TCache 0x30 bin because of the size and the version of LibC being used. The code for this can be seen below:

// Allocate and free two chunks of size 0x30

char\* data = malloc(0x20);

char\* data2 = malloc(0x20);

free(data2);

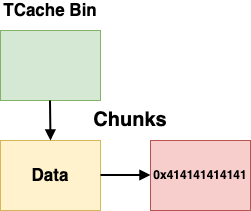
free(data);

In order to see functionality in memory, run the bins command directly on startup of the application in *gdb*. The output of this is shown below:



After the allocations and freeing occurs, the *freed* pointer data is written to. Because this chunk has already been freed and is written to anyway, this creates a ***use after free*** (UAF) vulnerability.

This can be easily seen if the program is paused directly after the call to *fgets* but prior to the third call to malloc. Run the bins command again to show the corrupted heap:



Notice the 0xffffffffffffffff as the second item in the bin, which is not a valid chunk and the value that is being written. This concludes the section discussing the vulnerability.

A graphical visualization can be seen in figure 3 showing the exploit in action.

Fake Fd Pointer

Figure 3: Corrupt TCache Bin chunk Fd Pointer

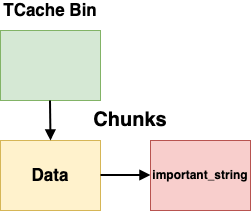
With the vulnerability discussed above, it shows a *uaf* on a freed TCache bin chunk. This *uaf* starts on the fd pointer of the chunk, which is perfect for the *fd poison* attack. The goal of this attack is to place a pointer to our target location as the fd pointer. In this case, our target location is the important\_string variable.

Figure 3 clearly shows that we have control over the *fd pointer* of data. Instead of writing a bunch of A’s, we need to write the address of important\_string over the top of the old *fd* pointer. In order to handle write this properly, the bytes must be packed in little endian form, which is easily done using *pwntools*.

The address of important\_string is 0x601080. If we pack and write this data over the *fd* pointer, this works perfectly. There is an easier way to get the address via pwntools, which is used in the code below for modularity reasons. The code for the section is shown below. Additionally, a visual representation of this can be seen in Figure 4 below.

fake\_fd\_location = elf.symbols['important\_string']

p.sendlineafter("Data:", p64(fake\_fd\_location) )



Allocating the Fake Chunk

Figure 4: Data Fd Pointer pointing to important\_string address.

// Allocate new chunk

malloc(0x20);

printf("Enter victim: ");

char\* victim = malloc(0x20);

fgets(victim, 0x20, stdin);Box and whisker chart

Description automatically generated with medium confidence

The chunk being in the TCache bin is not enough for success, as it is still in the allocator. So, we need to get the chunk out! To get the chunk out, we need to make ***two*** calls to malloc.

Figure 5: One call to malloc has been made.

The TCache bin is FIFO (first in first out). This means that the chunk at the front of the list will be removed first. As seen in Figure 4, there is a chunk between our fake chunk: data. To get the important\_string, this chunk must be removed first. This happens on the first call to malloc shown in the code above.

The *second* call to malloc will bring back our fake chunk into the victim variable. This fake chunk points to the memory address of important\_string. With the final call to fgets, we will overwrite the string, important\_string with whatever value whe want. Finally, when the string is printed, the pwnage is complete. For this section, you only need to write something fun here; nothing technical needs to be done.

If we look at *gdb* after we have written the value, the original string “*You can’t touch this”* has been replaced with *“DEADBEEF”*. 

The code for this was completed in the *start.py* file. However, for completeness, we will add it here.

value = "DEADBEEF"

p.sendlineafter("victim:", value)

After adding in this code to perform the write, the program should print out *“DEADBEEF”*. This can be seen in the image below.   


Mangling Challenge Add-on

In GLibC 2.32, a security mechanism for singly linked list pointers was added: pointer mangling (encryption, mangling, encoding, etc.). The original version of the challenge, the address of the *fd* pointer is trivial to alter and put at a different location. This is because no security protections were added onto singly linked list. The extension to this challenge requires that the pointer mangling protection be bypassed. The only difference in the solution is ***mangling*** the *fd pointer* prior to it being used.

The formula for the encryption is as follows:

.

The *Fd Pointer* of the TCache chunk is XOR’ed with the location in memory where the pointer is being stored at. Prior to doing the XOR, the *storage location* of the pointer is shifted over by 12 bits. ASLR is essentially used as the encryption key for the *fd pointer*. If the address of the heap is not known, then the pointer cannot be mangled.

The *shift* is done because the first twelve bits of an address are always the same. Hence, a relative overwrite was trivially possible on the first 12 bits, bypassing the protection entirely.

The *decryption* is the same except that the *Fd Pointer* is now the *Mangled Pointer* from the *encryption* process.

If the program (on LibC 2.32+) is started and paused directly after the two call to *free*, you’ll notice that the *second* pointer looks completely non-sensical; this is because the pointer has been mangled prior to being added to the *fd pointer* slot. This can be shown in a *gdb* snapshot below:



If we simply add a non-mangled pointer, the program will attempt to decrypt/unmangle the value. This will surely end up not working out; it will either *abort* for alignment reasons or *segfault* because the address does not exist.

Because of the unmangle step, we need to *mangle* the fake *fd pointer* ourselves! The challenge does not have ASLR turned on, meaning that the *storage location* is a static value. The address for this pointer is ﻿0x6022a0 . Additionally, the address of important\_string is ﻿0x601080. These are the two values that we need in order to mangle/encrypt the pointer. Figure 6 shows the formula and result of the mangling process after the *storage location* has been shifted 12 bits.

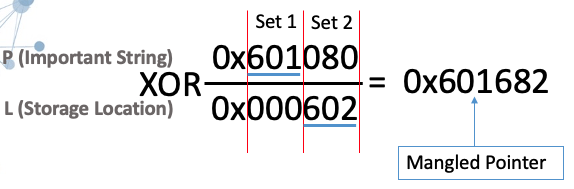


Figure 6: Fd Poison Mangling Pointer process

The Python script has a function for *mangling* and *unmangling* pointers. By calling this function with the proper parameters, the *fd pointer* will be set properly. The code below uses the correct parameters to *mangle* the pointer and write our fake *fd pointer*. The first parameter is the *fd pointer* and the second parameter is the *storage location*.

mangled\_ptr = encode\_ptr(0x601080, 0x6022a0)

p.sendlineafter("Data:", p64(mangled\_ptr))

Diagram

Description automatically generatedAfter the mangling process, the new *fd pointer* is 0x601682. Using this as the new *fd pointer* address will result in the final allocation of *malloc* unmangling the pointer and returning important\_string after the two calls to *malloc*. The diagrams shown above after writing the *fd pointer* are the same as the original challenge, besides the fact that the pointer is *mangled*. This difference can be seen in Figure 7.

Figure 7: TCache Bin after mangling Fd Pointer