Fd Poison Challenge

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

Figure : A TCache chunk in GLibC Malloc

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

* happy.c: The source code for the exploitable challenge.
* happy: The binary for the exploitable challenge.
* start.py: The starting point for the challenge with part of the solution already written.
* version/: The folder that holds the different *stages* of the exploit. This is used in order to easily go to and from newer or older versions of the exploit.

The goal of the challenge is to use all of our knowledge from the workshop to pop a shell on the binary. This is done by using the recently learned ***fd poison*** to a overwrite function pointer and pop a shell. 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 to go from step to step; please read the comments, as they will guide you through the challenge. If you get stuck, use the grab command to go to newer versions of the challenge. For instance, if we want to skip to finding the vulnerability, go to *step 1* run grab stage 1.

Challenge Setup

In order to *pwn* something, we must understand the environment we are working in and the code itself. This first section will walk through the source code and environment of the challenge.

The challenge gives 4 options, excluding exit.

1. Creating a Kid (Allocating memory via malloc)
2. View a Kid (Read the allocated memory)
3. Removing a Kid (Freeing the memory)
4. Editing a Kid (Writing to the memory)

Text

Description automatically generatedAll of these operations act on a kid. Well, what’s a *kid* in this context? As shown in *Figure 2*, a kid is a [*struct*](https://www.geeksforgeeks.org/structures-c/) made up of 3 elements. A *struct* is simply a collection of types used to create a *new* type.

Figure : A kid struct in *happy.c*

First, the struct contains an *age* field that is of type *long long*. A [*long long*](https://en.cppreference.com/w/cpp/language/types) is an integer type (whole numbers) but has an expanded amount of values it can store. On 64-bit systems, a long long is 64 bits. This stores the *age* of a kid throughout the program. Secondly, there is a *string* (list of characters) of size 20 that is used to store the *name* of the kid. Finally, there is a **function pointer** attached to the kid: *print\_info*. Depending on the age of the *kid*, a different print function is used. It should be noted the fields of the struct are representative of *how* the fields are stored in memory. So, *age*, *name* and *print\_info* is the ordering of the structure when stored in memory.

Within the *main* of the program, there are two variables that keep track of the *kids*: all\_kids and counter. All\_kids is an array of pointers to kids. The counter variable keeps track of the *number* of kids and is capped at 10.

Create Kid

Text

Description automatically generatedThe first option is to *create* a kid. As seen in the source code in *Figure 3*, the first operation that happens is a call to malloc with a size of **0x28**. Because of the rounding up and necessary metadata that needs to be added for a chunk, a chunk of size **0x30** is returned.

Figure : Create a kid function

After the allocation, all of the values of the new struct kid are written to. Two calls to *fgets* are made in order to get the kid->name and kid->age. The final field being written is the function pointer at kid->print\_info. If the kid is younger than 10, then the function print\_kid\_younger is used; otherwise, the function print\_older\_kid is used for the function pointer. Finally, the pointer to the dynamically allocated memory is returned from the function.

The *kid pointer* is added to the array of kids. Additionally, the counter variable is *incremented* in order to keep track of the *number* of kids in the program.

Text

Description automatically generatedView Kid

Figure : View kid validation and function pointer call

The second option is to *view* a kid. In order to *view* a specific kid, we must select the *index* of a kid from the all\_kids array, which is 0 indexed. In order to prevent out of bounds access or memory corruption, there is a bounds check on this in both the positive and negative direction. The sanity check (shown in *Figure 4*) validates that the *kid index* is less than the *counter* variable. Additionally, the Text

Description automatically generatedindex must be greater than 0. More on this check later!

After the validation has been done for the *kid index* that the user supplied, the kid->print\_info function is called. Both of the functions access the kid->name and kid->age fields of the struct and print them. Besides this, the two functions have a few textual differences on the display as seen in the *Figure 5*.

Figure : Print functions for a kid

Free Kid

Text

Description automatically generatedThe third option is *removing* or *freeing* a kid. As in viewing a kid we need to access a particular kid from the all\_kids array. In order to prevent out of bounds access or memory corruption, there is a bounds check on this. The sanity check (shown in *Figure 6*) validates that the *kid index* is less than the *counter* variable Additionally, the index must be greater than 0. This is the same as the *View Kid* function.

After the bounds checking on the *index*, the function dump\_kid is called. This function simply calls free on the kid pointer and exits.

Figure : Free a kid source code

Text

Description automatically generatedEdit Kid

Figure : Edit kid function source code

The final operation is *editing* or *modifying* a kid. As with the previous two operations, there is validation done on the index of the specific kid being modified. This access check is the same as *free* (*Figure 4)* and *view (Figure 6)*. If the kid is valid, the function edit\_kid is called with a pointer to the *kid* as a parameter. The source code for this function can be seen in *Figure 7.*

The parameter into this function is a *pointer* to a *kid*. Next, two calls to *fgets* are made in order to get the kid->name and kid->age. This function is similar to the *create* function with the main difference being that the pointer is based in as a variable instead of being created for the first time.

Miscellaneous

All of the code within the basic functionality is explained above. Besides the code showed above, there is a function named win() that runs pops an auto shell by running system(“/bin/bash”) for us. Once control flow of the program has been gained, we can trivially pop a shell on this binary.

The program is running within GLibC 2.26; this is first version that includes the TCache in it. Additionally, the binary has a variety of protections to deal with. Of particular note, the binary has *Full RELRO*, the *Nx* bit enabled and ASLR is turned on for the challenge. This concludes the section on the basic setup and challenge analysis.

Vulnerability - Stage 1

In order to get to the exploitation step, we must find an issue with memory handling that allows us to cause memory corruption. When looking at *heap* specific vulnerability classes, it is important to look out for calls to free. **Data access** (reading and writing) after a call to free is a great way to cause memory corruption in most programs.

In the functionality for *freeing a kid* (shown in *Figure 6)*, notice that the pointer stored inside the all\_kids array is never NULLed out (set to 0x0) after being freed. Because the pointer is still around (and was just freed), there is a chance for a *use after free* (UAF) vulnerability within the application. But, what about the bounds check?

All of the bound’s checks are done properly for not going out of bounds on the array both reading and writing. However, this is only keeping track of the *number* of kids being stored. Because kids can be *freed* in any order, the *counter* is insufficient for preventing UAFs. As an example, create one kid then free the kid at index 0. There is a *use after free* on index 0 now.

To demonstrate this more clearly, we will use the following order of operations:

* Create a kid – Index 0 in all\_kids array
* Free the kid at Index 0 – Creates a UAF on index 0
* Edit kid 0 – Writes over the top a freed chunk

The *pwntools* code to do this is shown below (stage1.py):

# Create three kids

create\_kid("Kid#1", 5) # Index 0 – Counter is now 1

free\_kid(0) # Free the kid. Decrement counter to 1

edit\_kid(0, "Mr. Hackerman", 0x42424242) # UAF

To demonstrate this in GDB, use the code above or run python3 stage1.py in the VM. Once the above code has ran, we have triggered the use after free vulnerability and write over the top of the free chunk. In order to see this, run tcachebins and p all\_kids. Now, in *Figure 8,* we can see that the chunk within TCache Bin 0x30 is 0x860260. The first pointer of all\_kids is also 0x860260. To further prove the point, the TCache bin chunk points to 0x42424242, which is not a valid chunk and happens to be the value of the *age* that we wrote in the previous step. Because the *fd* pointer was set to 0x42424242 and the two pointers being the same, we have a clear *use after free* vulnerability.

Graphical user interface

Description automatically generated with medium confidence

Figure 8: Use after free on kid

The use after free vulnerability is caused by the failure to NULL out the pointers and the continuing access of these pointers. Because this pointer handling is done incorrectly all over the entire application, *viewing*, *editing* and *freeing* all are suspectable to being used on a freed pointer.

The *viewing* allows for an *information leak* (next section). The *editing* allows for the corruption of a currently *freed* chunk or editing a chunk after it has been reallocated. The ability to free a chunk again gives a *double free* vulnerability. Even though the double free vulnerability exists in the program, it is simpler to use the use after free because it takes less steps to exploit. But, using the double free is still a valid option for this binary if the *use after free* did not exist.

Information Leak – Stage 2

The presence in memory corruption bugs is a known and terrible reality of low-level code. Developing *secure* low-level code is difficult to do! As a result, mitigations have been put into place to make exploitation *harder*. The full list of mitigations is plentiful, from Nx to the Shadow stack. For the purposes of this challenge, we will only be focusing on *Address Space Layout Randomization* (ASLR) and *Position Independent executable* (PIE).

[ASLR](https://blog.morphisec.com/aslr-what-it-is-and-what-it-isnt/) is used to *randomize* data sections of the program. This includes the *heap*, *stack* and a few other sections. By randomizing these sections of memory, it is no longer trivial to reference pointers and other parts of data. [PIE](https://access.redhat.com/blogs/766093/posts/1975793) is used to *randomize* code sections. Historically, it was used so that libraries could be loaded anywhere in memory. Recently, it has been used as a security protection because it randomizes the *code*, *bss* and a few other sections. With these protections in place, we now have *another* thing to worry about: we need to bypass the randomization.

For this challenge, we only need to worry about the *ASLR* randomization. In order to bypass it, there are a few options:

* Information leak to break ASLRs randomness. Once we know *one* address, we can use it to figure out the *base* *address* of the different sections.
* Relative overwrites.
* Brute forcing.

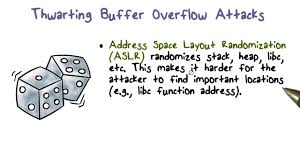
Relative overwrites require precision and tends to be limited in impact. Brute forcing would work but consistently. In order to make this exploit consistent, we will use the *information leak* technique. As mentioned above with the *use after free* vulnerability, we have the ability to *read a kid* while they are free! Considering we need to leak memory, this looks like the perfect option.

Figure : ASLR picture

The next question you are going to ask is *why is the leak necessary?* In exploitation, you commonly find the memory corruption method and see what you need in order to make it possible. In the next section, you will discover *why* this is required as well. For now, just know that it is necessary.

Table

Description automatically generatedIn order to get the information leak, we need to align our data properly with other important values. For instance, the value of *fd* in a chunk and the *key* field both can contain pointers to heap memory; the chunk can be seen in *Figure 1* at the top of the article. These are good areas to try to get a heap leak from.

Within the *kid struct*, there are two fields that align with chunk metadata fields. As shown in Figure 10, kid->age (which is 8 bytes in length) lines up with the fd field of a chunk. Additionally, the beginning of the kid->name field lines up with the *key* field of a chunk. For this challenge, we will use the *fd* field, as it will point to a heap memory location! Because the version of LibC being used is 2.26, there is no *key* field. On the 2.32 version of this challenge, the *key* field can also be used for an information leak as well.

Figure : Allocated Kid struct and Free chunk

The target is the malloc\_chunk->fd. However, it is not as simple as freeing the chunk and leaking the heap memory address; we must dive into how the allocator work for ordering of chunks! The TCache bins, which is where the free chunks of size *0x30* go, is a singly linked list with that is *last in first out* (LIFO). This means that if a chunk is added to the list, it is at the *top* of the linked list. Additionally, the *end* of the linked list of pointers ends with a **NULL** pointer (unless the bin is full with 7 items in it).

Because we are only going to use 2 chunks (minimalist completion) in the bin, this means the leak must happen on the *first* chunk in the bin. Hence, the chunk with the accessible *use after free* must be at the top of the linked list. With two chunks in the TCache bin and a *use after free* on the first chunk in the linked list, the accessible chunk will have a pointer to the other freed chunk. The source code for making this work is shown below:

# Create three kids

create\_kid("Kid#1", 5) # Index 0 - Counter is 1

create\_kid("Kid#2", 6) # Index 1 - Counter is 2

# Make TCache Bin look like '0->1'

free\_kid(1)

free\_kid(0) # On top of the 0x30 TCache Bin

The idea of the code above is to make the TCache bin look like **0->1** with a *use after free* on the chunk at index 0. If we tried to get the leak from index 1, it would not work because the end of the linked list is set to *NULL*.

TCache pointers lined up for a memory leakIn order to demonstrate the impact of this, we will once again use GDB. After adding the code above to trigger the impactful *use after free*, pause at this point of the program. The easiest way to do this is to put pause() within the Python code; this will stop the program and make it easy to see what is happening in GDB.

Figure : GDB leak analsysis

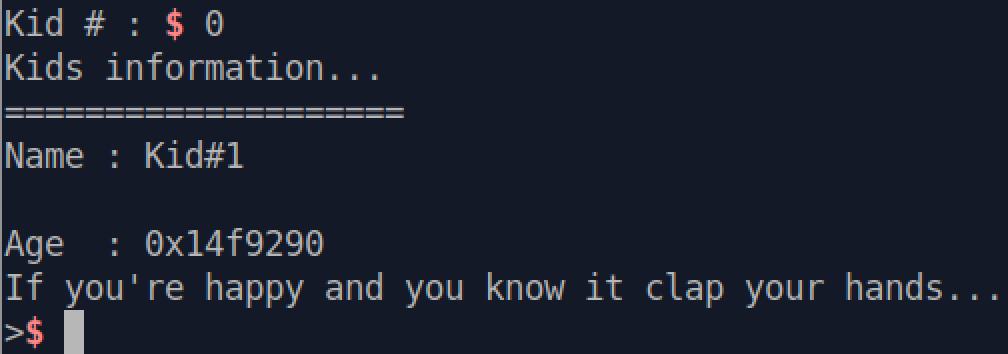
Once the program is paused at the point discussed above, run tcachebins, x/2gx all\_kids[0] and p all\_kids, which can all be seen in *Figure 11*. The first command shows the TCache; notice that there are *two* chunks in the 0x30 TCache. The second command displays a list of *kid pointers* in the all\_kids array, which has a *use after free* on it. To show the leak works, the final command in *Figure 11* shows the memory address of the first pointer in the linked list. Underlined in green, notice that the first entry of the all\_kids array (chunk 0) has a pointer to the *next* chunk in the linked list. Because this field corresponds with the kid->age field, this will leak the *fd* pointer as an integer via the *age* field. The observant hacker will notice that commands *1* and *3* show the same information but in slightly different ways. After the setup above, printing the kid at *index 0* will display the *fd pointer* of the chunk. Boom! There’s a leak that we can use to break ASLR on the heap ☺ This can be seen in *Figure 12*.

Figure : Use after free information leak

Text

Description automatically generatedThe reason the *leak* is so valuable is that this allows us to find the *base* heap address. Because the program is deterministic and starts on our command, the *offset* from the *base* to the *leak* will be the same every single time. The *base* address of the program, while debugging, can be found by running vmmap in GDB. In the bottom row of *Figure 13,* the base address of the heap is shown. Using the *leak* from above, we can subtract the *base* from the *leak* in order to get the *offset*. In this case, the math is 0x14f9290 - 0x14f9000 = 0x290. Now, the next time we get a leak, we can subtract **0x290** from the leak to find the *base address*. For example, 0x14f9290 – 0x290 = 0x14f9000, which is the base address for the heap. Now, we have ASLR defeated for the heap!

Figure : Program memory mapping

To programmatically break ASLR, we can read in the value of the *age* then use the offset math that we did above. The Python pwntools code for this is shown below:

name, age = view\_kid(0) # Read kid with UAF

leak = int(age, 16) # Parse the age

# Find base address of heap

base = leak - 0x290 # Offset math

print("Leaked heap base...", hex(base))

Information leaks are not the *flashy* part of the exploit. However, with modern binary protections, they are a necessary part of the process. In the next section, we will use the *fd poison* technique to overwrite a function pointer. For this to be possible, we had to break ASLR first.

Fd Poison Attack – Stage 3

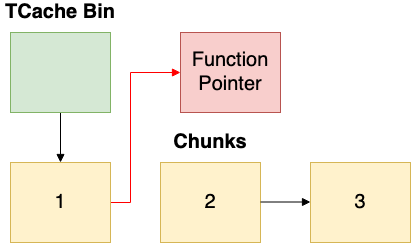
As shown in the *information disclosure* section, the kid->age lines up with the *fd* pointer of the TCache. By writing to the age with the *use after free* bug, which we can be seen in the proof of concept in the *use after free* section (stage1) above, control an *fd* (next) pointer on a TCache bin chunk. Because we control the *fd* pointer of a free TCache chunk, we can use the ***fd poison*** technique to place a chunk to an arbitrary location. As discussed in the *fd poison* module, it is really that simple; the TCache has no sanity checks when being removed from the bin. To launch the attack, overwrite the *fd* pointer of a free TCache chunk, allocate the fake chunk then write to an arbitrary address. For more direct details on the attack, please review the slides and exercises from the *Fd Poison* module and refer to *Figure 14*.

Figure : Fd Poison attack visualization

With the ability to write to any location we want in the program, *where* should we write? To do maximum damage, we want to overwrite a function pointer. Here are the options for this:

* kid->print\_info function pointer on a kid
* \_\_malloc\_hook or \_\_free\_hook
* Global Offset Table (GOT) Entry
* RIP on stack

The latter three are not possible in this challenge. First, we cannot overwrite the GOT entry because the binary is *Full RELRO*; this means that the GOT table is a read only. Second, even though we *could* try to overwrite a saved RIP on the stack, we do not know the location that this is stored in memory because of ASLR. The issue applies for the GLibC Malloc [hook functions](https://www.gnu.org/software/libc/manual/html_node/Hooks-for-Malloc.html) as well the hook is stored in a location that that is randomized by ASLR/PIE. Because we do not know the address of LibC, we cannot overwrite these function pointers, even though they are good targets.

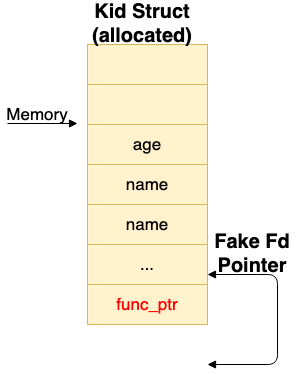
The only suitable target for hijacking the control flow for the binary is the kid->print\_info. Since this function pointer lives on the heap, and we know the *base address* of the heap, the function pointer on the heap is a perfect candidate for overwriting! The reason we wanted to know the heap address above was because we needed it to know where to overwrite the function pointer in this section. This attack is only possible because the address of the heap is *known* from the previous section. Sometimes, it is best to work *backwards* from the initial memory corruption primitive to see what you need to make it work.

Figure : Placement of the Fake Chunk

The best address to set the fake *fd* pointer is directly on top of a kids kid->print\_info. A diagram of this can be seen in *Figure 15.* The TCache fd (next) pointer points to the address of the *next* location of the *fd* pointer; as talked about in the *Intro To Malloc* section, it points to the *memory* pointer of a chunk, not the *beginning* (chunk) pointer. For this exploit, we will be using the kid at index 0. However, any one of the kids could have their function pointer overwritten.

Text

Description automatically generatedTo see the overwrite in action, view the demonstration of the *use after free* section (stage 1); it can be seen in *Figure 8* that the *fd* pointer has been overwritten with a value of 0x42424242. While running the same proof of concept from *stage1.py*, if you create two more kids, the program will crash on the creation of the second kid because 0x42424242 is not a valid memory address. As shown in *Figure 16,* if you run tcachebins and print e at the point of the crash, you will notice that the pointer being acted upon is actually 0x4242424242, even though this is not a valid memory address. However, this means we *do* control the address being written to.

Figure : Corrupted Fd Chunk

The source code for this programmatically (shown below) uses the *heap base* in order to calculate the proper address to overwrite: the *heap* *address* of the index 0 kid struct. Then, the pwntools code will overwrite the *fd* pointer of the kid at index 0 to corrupt the TCache bin. It should be noted that the edit\_kid function is writing the fake address to the *age* field of the kid. This is because the *age* field lines up with the *fd* pointer on a freed chunk.

tcache\_chunk\_offset = 0x250 # Size of the tcache storage chunk

fake\_chunk\_offset = 0x30 # Align 'age' of new fd with 'func\_ptr'

fake\_chunk\_ptr = leak + tcache\_chunk\_offset + fake\_chunk\_offset

edit\_kid(0, "Edit FD", fake\_chunk\_ptr) # Set the fake fd

Graphical user interface

Description automatically generatedThe code above is considered content complete for *stage3*. To see this in action, run the code through *stage3.py* to eventually hit the *interactive* block. At this point, Cntl-C in GDB to pause the program. Run the command tcachebins in order to see the contents of the bin, such as in *Figure 14*. The first pointer of the TCache, which is underlined in red, is the pointer at *index 0* of all\_kids. The pointer underlined in green is the value that we wrote at the end of stage3; the *fd* pointer of the memory at index 0 of the all\_kids array. We know this is in the proper address because pointer underlined in pink is a function pointer to print\_kid\_younger. As we wanted to overwrite kid->print\_info, we are right on track. The next section is about getting this fake chunk out of the bin and overwriting the function pointer.

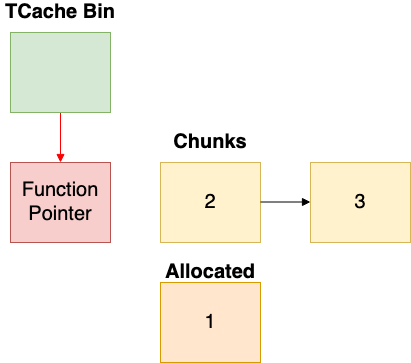
Figure : Fake fd pointer set on function pointer

Overwriting the Function Pointer (final)

A close up of a clock

Description automatically generatedThe code above puts the heap into a state where the fake chunk is in the TCache bin. However, in order to cause damage, we need to get the chunk *allocated*. Since the TCache is LIFO, the allocator will remove the chunks from the top of the linked list first. As shown in *Figure 18*, this will require *two* allocations. The first one is to get the chunk at index 0 of all\_kids out (the chunk labeled ‘1’ in the image). The second allocation will be to get the chunk at the *function* pointer, which can be seen in *Figure 18* by the chunk that says *Function Pointer* in red.

Figure : TCache Bin with the fake chunk

There is only a *single* malloc operation in the entire program. Because of this, adding and removing chunks is as simply as calling *create kid* and *delete kid.* So, a call to *create kid* with any parameters will get this first chunk out. After this call to malloc, as shown in *Figure 19*, we now only have the function pointer (in red) in the bin. The next allocation will remove the *fake chunk* from the bin and cause further memory corruption.

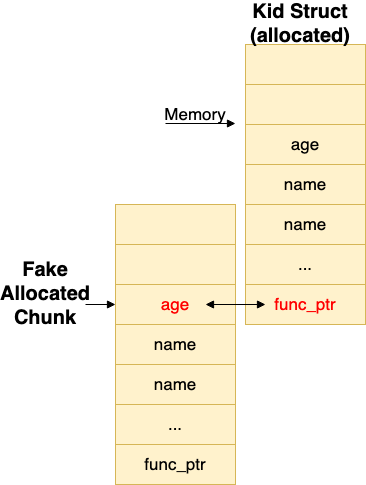
The observant fellow will notice that the TCache Bin count for the 0x30 bin is already at 0, prior to the final allocation. So, if the *count* is already at 0, then how can we take out this new chunk? Prior to GLibC 2.30, the tcache->counts had a [flawed validation check](https://elixir.bootlin.com/glibc/glibc-2.26/source/malloc/malloc.c#L3062) that makes this *count* value useless. In newer versions of GlibC, we must free an additional chunk, prior to the corruption, in order to get the count high enough to take the fake chunk out. In the Pointer Mangling Extension of this challenge on GLibC 2.32, an additional malloc and free must be done in order to get the fake chunk out of the bin. For the purposes of this part of the challenge, this can be ignored though.

Figure : TCache bin with next allocation to be fake chunk

The next call to malloc will allocate over the top of the function pointer at kid->print\_info of the kid at index 0; the visualization of this can be seen in *Figure 20*. But, *what* do we actually want to write? We want to overwrite the *function pointer* with the address of the win()function*.* Because of how the *fake chunk* was setup, the *age* field will be used to overwrite the function pointer, as seen in *Figure 20*. However, depending on the setup in *stage2*, the name (if packed properly) could be used to corrupt the function pointer as well.

Figure 20: Overlapping allocation setup with fake chunk

Text

Description automatically generatedAs described above, the final call to malloc will overwrite the function pointer. When writing the kid->age of our new kid, it will overwrite the function pointer. We will set the *age* on the write to be the address of the winfunction. If after this step we pause execution in GDB and display the function pointer of the kid (p all\_kids[0]->print\_info), it will show the address of the function pointer points to the win function; this can be seen in *Figure 21*.

Figure 21: Overlapping the function pointer

Now, all we have to do is trigger the function pointer by *viewing* the kid at index 0. This will dereference the pointer and call it! Instead of calling one of the *print* functions for the kid, it will pop a shell ☺ Pwnage complete! The source code for the final part of the exploit is below, which does exactly what we discussed above in a *pwntools* scripts. The call p.interactive() allows for you to interact with the shell at the end of the script.

create\_kid("A" \* 8, 1) # Remove the filler chunk

# Overwrite ‘kid’ function pointer with 'win()'

create\_kid("PWNED!", elf.symbols['win'])

p.sendlineafter('>', '2') # 'View kid' option

p.sendlineafter(':', '0') # Trigger 'win()'

p.interactive() # Interact with the shell

Conclusion

The challenge above uses a *use after free* vulnerability to defeat ASLR on the heap then abuses the *fd* pointer of a chunk to write to an arbitrary location. Defeating ASLR via an information disclosure is a necessary part of exploits now-a-days. This is a standard full chain exploit used in a countless amount of *CTF* challenges and many real-world exploits.