Unlink – Exercise 2

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

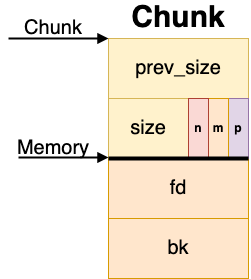
There are a several files in here but only four of them matter:

Figure 1: Malloc chunk

* unlink.c: The source code for the exploitable challenge
* unlink: 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 **unlink** exploit technique in order to solve the challenge. We will use the *unlink* technique in order to overwrite a function pointer to jump to our custom shellcode. By the end of this exercise*,* you should have a better understanding of how the unlink exploit technique works and how consolidation works. This exercise is the same as the previous one, but with a twist to make it slightly harder.

Challenge Setup

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

Graphical user interface, text

Description automatically generatedAt the beginning of the program, three calls to malloc are made; all of which return chunks of size **0x90**. The names are intentional and are used as they are named.

Figure 2: Allocating the chunks

Graphical user interface, text

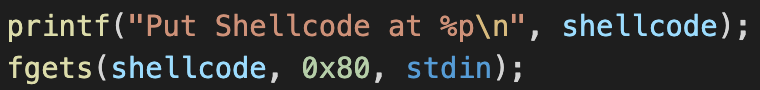
Description automatically generatedThe variable shellcode is controllable by the user. This is where you can store the shellcode for the challenge. Although the *heap* is normally non-executable (NX), the heap was made executable for this exercise. The purpose of the *unlink* technique is to gain a *WRITE-WHAT-WHERE* primitive. Because of this, the best place to write to is a *function pointer* and point this to either the beginning of a ROP chain or shellcode.

Figure 3: Writing to the unlink variable

Figure 4: Writing the shellcode

Text

Description automatically generatedThe next section of code writes to the chunk unlink\_chunk with a size of **0x90**. The code for this can be seen in *Figure 3*. Remember, the chunk unlink\_chunk is only 0x80 in size

Figure 5: Free the victim and print

To end the program, the third chunk (free\_victim) is freed. Then, a call to *puts* is made in order to print out a string.

Besides what is shown in the binary, there are a few environment setups that are important to note.

* This challenge uses GLibC 2.23. The most notable thing about this is that the TCache is not in GLibC 2.23.
* The security checks for the *unlink* macro have been removed
* The heap was made *executable* for this exercise
* The function *winner()* can be called in order to pop a shell

Vulnerability

In *Figure 3* a call to fgets is made that writes to the memory location of unlink\_chunk. The allocation for this chunk is only **0x80** bytes (0x90 sized chunk in total, including the metadata), as shown in *Figure 2*. Because we are writing from the *memory* pointer with only 0x80 bytes being expected, we have a 0x10 (16) byte *heap buffer overflow* on this chunk.

Directly above this chunk is free\_victim. As shown in *Figure 1*, the first two fields (0x8 bytes each) are at the bottom of the chunk. Because the overflow is only by 0x10 bytes, the *prev\_size* and *size* of free\_vctim are the only location we can write to.

Text

Description automatically generatedTo demonstrate this vulnerability, let’s jump into GDB. On the VM, within the *unlink/exercise2* directory, run the command ﻿run ./unlink 2.23\_unlink gdb. This will load the unlink binary into GDB with the proper version of LibC for the exercise. After the first breakpoint (at main) run the command continue to go on with the program. This will cause an automated breakpoint to occur at the call to *fgets* in *Figure 3*.

Figure 6: Chunks prior to buffer overflow

Text

Description automatically generatedThe call to *fgets* shows a write of **0x90** bytes. By running the command x/2gx unlink\_chunk – 0x10, we can see the chunk. Notice, that the second group of 8 bytes (the size) is only **0x91**, which can be seen in *Figure 6*. Because a write starts offset at 0x10 (memory vs. chunk pointer) this demonstrates there is a **0x10** byte buffer overflow. The second command (x/2gx free\_victim – 0x10) is the memory directly after the chunk unlink\_chunk. Notice that the values of the chunk, shown as the second entry of values in *Figure 6*. Run continue or c to keep on with the exercise.

Figure 7: Buffer overflow into 'free\_victim' chunk

Once the input for the *fgets* with the buffer overflow appears, send 0x90 characters. Once this is done, another automated breakpoint should be hit on the binary. Now, run the command x/2gx free\_victim – 0x10 again in order to see if the buffer overflow occurred. As we can see in *Figure 7*, the output has changed! Instead of having a proper *prev\_size* and *size*, we have overwritten these values with A’s instead (0x41 is A in ASCII). At this point, we have demonstrated we have a 0x10 byte buffer overflow vulnerability.

Forcing an Unlink

There is a *buffer overflow* vulnerability in this binary, as described above. Now, it is time to exploit it using the *unlink* technique.

Figure 8: Consolidate chunks backwards source code

Text

Description automatically generatedThe *unlink* process takes place whenever a chunk within a *doubly linked list* (unsorted bin, smallbin, largebin) is being removed from the middle of a linked list. This normally occurs when two chunks need to be *consolidated* (combined) in order to make one chunk; this is done to prevent massive fragmentation within the allocator. In the case of this exercise, unlink\_chunk and free\_victim are directly adjacent to one another. The problem is that only the free\_victim is being freed. This means that the unlink macro will A screenshot of a cell phone screen with text

Description automatically generatednever be triggered; can we force this to happen? Yes we can!

As shown in *Figure 8*, the chunk being freed is checked to see if the chunk below itself is free. This is done by checking the *prev\_inuse* bit on a chunk, which is the first bit of the chunks size. To trick the allocator to performing the consolidation to use *unlink,* all we have to do is set the value of the ***prev\_inuse* bit to 0** on free\_victim.

The buffer overflow above is the perfect candidate for this! We control size of the chunk, which means we can control the *prev\_inuse* bitof the chunk. If we make the consolidation happen on this chunk, we can simply write the *fd* pointer and the *bk* pointer within the data we control. As far as the allocator is knows, unlink\_chunk would be *free*!

The original size of the free\_victim is 0x91; let’s just keep the size the same as before, to ensure all of the sanity checks are passed for the size. The only difference we want to make for this is removing the *prev\_inuse* bit. By setting the size to **0x90** instead of 0x90, we can do just that. This can be seen in *Figure 9*.

Figure 9: 'unlink\_chunk' and corrupted 'free\_victim'

The other value we need to worry about is the *prev\_size*. In *Figure 8* above, the *prev\_size* is used to calculate the location of the chunk directly below it. In order to get the *fd* and *bk* pointers for the unlink attack setup properly, we need to line this up properly. Because the chunk below is 0x90 in *size*, let’s just write the *prev\_size* as 0x90 as well. Now, the first values being written to the chunk will be fake *fd* and *bk* pointers in the unlink attack.

The value of the *prev\_size* can go to anything with data we control. But, doing it to line up with the regular chunk is the simplest way to do it. The below code will properly set the *prev\_size* and *size* of the chunk free\_victim. The next step is setting up the *fd* and *bk* properly for the WRITE-WHAT-WHERE exploit.

fake\_chunk = p64(0x414243444546) # fd (TODO): Function pointer address (WHERE)

fake\_chunk += p64(0x414141414141) # bk(TODO): Shellcode address (WHAT)

fake\_chunk += b"C" \* 0x70 # Filler prior to chunk 'free\_victim'

fake\_chunk += p64(0x90) # Prev\_size of 'free\_victim'

fake\_chunk += p64(0x90) # Size of 'free\_victim'. Unset prev\_inuse bit

Unlink Exploit Triggering

In the previous step we forced the backwards consolidation to happen on a chunk that is actually not free. The reason we did this was to use the *unlink* exploit in order to achieve an arbitrary write primitive. It should be noted that the rest of this is exactly the same as the previous exercise.

Text

Description automatically generatedIn order to see the backwards consolidation being triggered (*Figure 8*), put a breakpoint at the *unlink* macro. This can be done using the command ﻿  
b unlink. If we run the program with this breakpoint set, you will notice that the breakpoint is hit when freeing free\_victim. Additionally, if you print *fwd* and *bck* (print fwd and print bck) you will notice that the pointer is the same value that is written into unlink\_chunk ; this can be seen in *Figure 10*. This confirms that the consolidation is happening as expected.

Figure 10: Showing the pointer addresses

With the *unlink* macro being triggered and we control the pointers for the fwd and *bck*, this is the perfect candidate for using the *unlink* attack! Using control over the *fd* and the *bk* pointers in our chunk (see *Figure 1*), we can construct a *WRITE-WHAT-WHERE* primitive.

A screenshot of a cell phone

Description automatically generatedIn the *unlink* macro in *Figure 1*1 the address of *P* is the pointer being freed. The value of FD is the fd pointer of P. Then, the value of BK is the bk pointer of P. We control both P->fd and P->bk because the *use after free* takes place on *P.* Since FD=P->fd and BK=P->bk, we control FD and BK as well.

Figure 11: Unlink macro

The third line (FD->bk=BK) is writing to the address of FD (plus some offset) with the value of BK. Additionally, the fourth line is writing to the address of BK with the value of FD. The important thing to note is that we control BOTH FD and BK. This means that we control the *address* being written to and the *value* being written. The control over both the *address* and the *value* means that we have a *WRITE-WHAT-WHERE* primitive!

Where and What to Write

Now that we have a *WRITE-WHAT-WHERE* primitive, what can we do with it? Ideally, we want to get code execution. So, for the *where,* we want to overwrite a function pointer directly. In this challenge, either the RIP on the stack or the *Global Offset Table* (GOT) can be used. For the rest of this write up, we will use the *GOT*, as it is easier and more consistent to use than RIP on the stack.

Diagram

Description automatically generatedIn Linux, binaries can either be *dynamically* or *statically* linked. When a binary is

Figure 12: Symbol resolution in LibC

*dynamically* linked, the libraries are not included within the binary. Although it looks like magic, they are figured out at run time and the usage in the *binary* itself is not overly complicated. When a library function is called, the *resolution* process finds the *address* of this symbol within LibC and writes the address of the symbol back to the *Global Offset Table* *(GOT)*. Because this symbol is effectively a function pointer, we can overwrite the GOT entries in order to get code execution. For more on information on how the *GOT* works, read [this](https://systemoverlord.com/2017/03/19/got-and-plt-for-pwning.html) article.

Table

Description automatically generatedAt the very end of the program (shown in *Figure 5*), there is a call to *puts*. By overwriting the GOT entry for *puts*, we can control the flow of execution! Now that we have the *where* taken care of, we can talk about the *what.*

Figure 13: Chunk setup for exploit

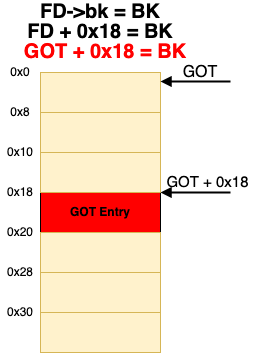
Because we are overwriting a function pointer, we need to overwrite the GOT entry with an executable section of code. Additionally, since *two* writes occur, we cannot simply write the value of *winner()*. Instead, we need to write to a location that is *writable* and *executable*. Luckily for us, the *heap* has been made executable for this challenge. As denoted by the variable name shellcode, it is the perfect place to find a *writable* and *executable* section of memory.

Figure 14: Unlink offset on the address

The outcome of the *where* and *what* can be seen in *Figure 13*. One concept that is not obvious is the *where* that we want to write to. The *unlink* macro, shown in *Figure 11*, contains the code FD->bk=BK. This is writing to the address of *FD* with the struct accessor *bk*. When accessing a struct in C, the accessor is only a small *offset* on the memory that depends on the content in the struct. FD->bk is functionality equivalent to FD + 0x18. Because of this, if we simply try to write to the address of the GOT entry (the target of the write) then we would write to GOT entry + 0x18, which is 0x18 bytes too far. In order to compensate for this, we need to **subtract** 0x18 from the address that we are attempting to write to. This way, the struct accessor is neutralized since FD + 0x18 – 0x18 is the same as FD. This concept can be seen visually in *Figure 14*.

Finally, we are onto the code for the *unlink*. In order to line up the attack, we need to setup the pointers properly on the chunk. The *fd (WHERE)* pointer needs to have the address of the *GOT* minus the offset. Then, the *bk* (*WHAT*) needs to have the address of the *shellcode*. The code for this can be seen below. The only difference between this code and the previous code snippet is that the *fd* and *bk* are now written in properly.

fake\_chunk = p64(puts\_GOT - 0x18) # fd: Function pointer address (WHERE)

fake\_chunk += p64(heap\_base + 0x30) # bk: Shellcode address (WHAT)

fake\_chunk += b"A" \* 0x70 # Filler prior to chunk 'free\_victim'

fake\_chunk += p64(0x90) # Prev\_size of 'free\_victim'

fake\_chunk += p64(0x90) # Size of 'free\_victim'. Unset prev\_inuse bit

It should be noted that although we choose the FD as the *what* and the BK as the *where* but this attack can be done in either direction. For the sake of simplicity, we just chose one and went with it. Additionally, *two* writes occur with *unlink* operation; one at FD + 0x18 and one at BK + 0x10. So, the content of the location being written to (in this case, the *shellcode*) could become corrupted if you are not careful.

The code changes above are the *only* TODO’s in the program! So, this should be enough to solve the exercise.

Shellcode

The shellcode was written for you but not explained; let’s change that. This shellcode is shown below:

0: 68 f6 11 40 00 push 0x401236

5: c3 ret

The goal of the shellcode is to jump to the *winner* function that pops a shell; the address 0x401236 is the address of the *winner* function. Within x86\_64 assembler, there is no *jmp* or *call* function that accepts an absolute address. For instance, the assembler call 0x401236would be quite nice, if it existed. Additionally, there is no instruction for moving a value directly into RIP. Because of these downsides, we must be creative!

In order to emulate an assembler instruction that *jmps* directly to the code, we are *pushing* the address of the *winner* function onto the stack. Then, the ret instruction takes this address off the stack and sets the RIP (instruction pointer) to this address. Now, the next instruction will be the *winner* function, popping the shell, which demonstrates major pwnage.

If you are trying to debug shellcode on x86\_64, the best instruction is int3 or 0xcc. This instruction will *pause* execution of the *shellcode* and wait for a debugger to continue execution. If trying to simply get to the point of shellcode, this instruction is extremely helpful.

A final note on the shellcode for the *unlink* challenge is to be careful that the shellcode does not get corrupted. A write occurs at FD + 0x18 and BK + 0x10. These writes will BOTH happen, corrupting the shellcode. In order to get around this, write small shellcode (like the one use in this example) or *jmp* over the top of the expected to be corrupted values.

Welcome to the Now

The *unlink* technique is a relic of the past. From the original *unlink* post in 2001, titled [*Once Upon a Free*](http://phrack.org/issues/57/9.html)*,* security protections have been put in place in order to prevent this exact attack from happening.

Figure 15: Unlink doubly linked list protection - [Source](https://elixir.bootlin.com/glibc/glibc-2.23/source/malloc/malloc.c#L1414)

The [protection](https://elixir.bootlin.com/glibc/glibc-2.23/source/malloc/malloc.c#L1414) aims to prevent the arbitrary setting of the *fd* and *bk* pointers on the check. In order to do this, the check validates that the chain for the linked list is valid from *both* directions. This is done by ensuring that P->fd->bk == P and P->bk->fd == P. By adding this protection, the classic *unlink* vulnerability has been patched in modern versions of GLibC.

You may be wondering: “*was everything I just did a waste of time?”* However, this could not be further from the truth! In order to run, we first then to crawl, then walk then run. From this module, you now have practical experience with exploiting the *fd* and *bk* pointers, understanding how the allocator works , *pwndbg* commands and many other important things. Additionally, the ***unsafe unlink*** builds off the *classic unlink* technique. Understanding this technique is crucial for the firm grasp of the *unsafe unlink* technique.

In the demo/unsafe\_unlink directory, there is a proof of concept for the ***unsafe unlink*** technique that is well worth the time to go through.