Lab 3

Exploiting buffer overflows with no mitigations enabled

Goal of this lab is to familiarize you with basic buffer overflows, getting comfortable automating exploitation using pwntools package.

Brief Intro

As shown in the slides, buffer overflow happens when programmer does not enforce limits on data being ingested properly. This might happen for various reasons, such as:

* Using function taking input without upper limit
* Specifying limits bigger than buffer can handle
* Using dynamic limits calculation that goes wrong way

In particular, C functions for handling input that you'll see in this workshop will be from the following pool:

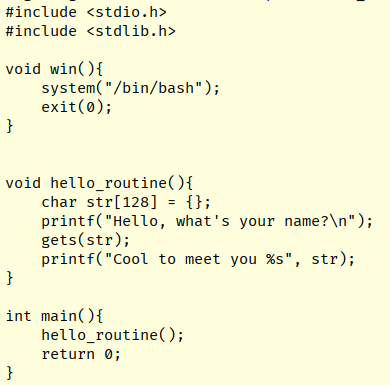
* gets(char\* str) – takes a pointer where input should be saved and reads until newline. No limits enforced.
* fgets(char\* str, int count, FILE\* stream) – reads to \*str pointer at most count – 1 characters.

Lab

The following steps will take you through exploitation of 1.out from "2. BOFs" directory. If you are confident in your understanding, feel free to skip the walkthrough and build the final exploit yourself. Since we are using pwntools, relevant template is provided for your convenience.

Solutions for all labs are provided in relevant "solutions" subdirectory. It is strongly advised you try your best before falling back to them. Ideally, you'd want to solve the challenges yourself and compare with provided solves.

Let's start with inspecting the source code of 1.c



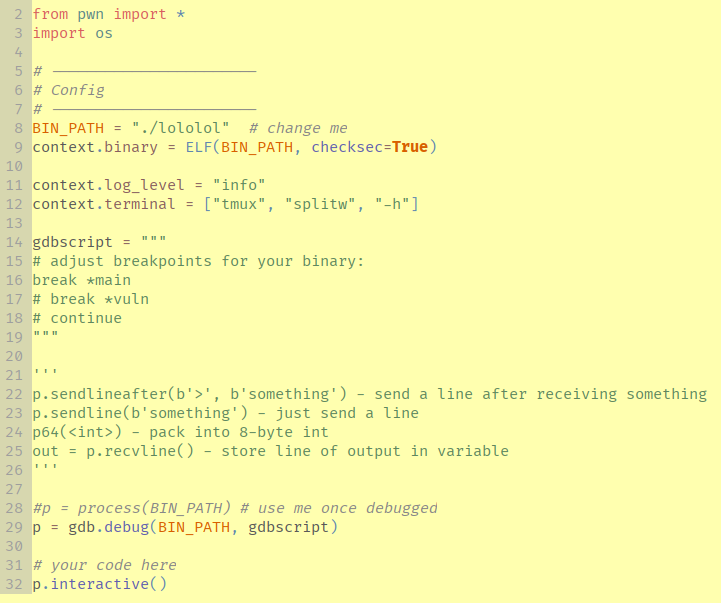
This is an example of ret2win challenge, where we want to redirect the execution to a function that is not called anywhere in the code.

Worth noticing is the gets(str) function, which will copy whatever we send to it without any length limit whatsoever and dump it starting from where str[] array begins.

A good habit to make is to use a program like checksec to assess what protections are in place:



We will use a provided pwntools template to make development quicker and automate exploitation. We'll want to copy the file to not break the template. Then, let's point path towards the target binary.



To run the exploit, execute

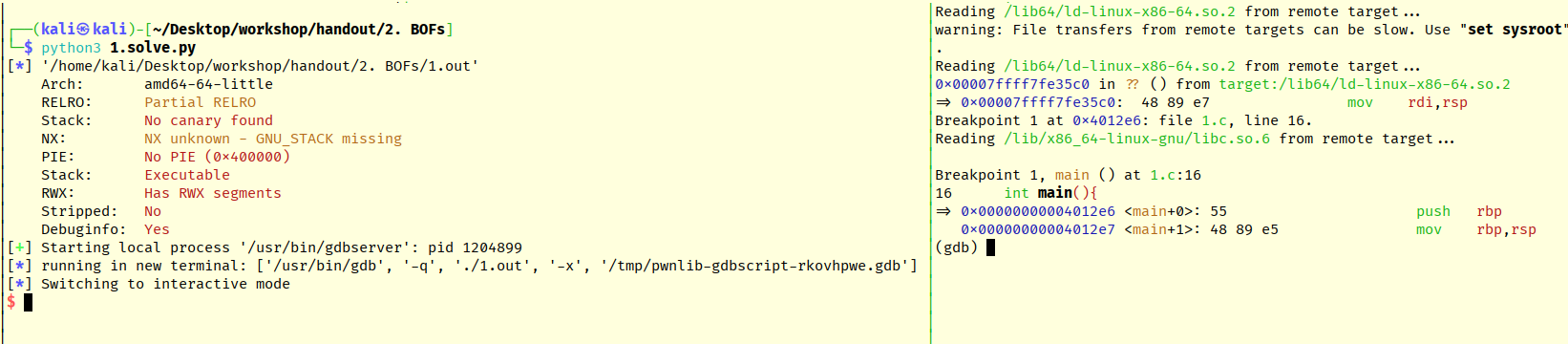
python3 <copied file>

This will launch gdb together with our binary and use tmux to split the terminal window. If you are not familiar with tmux, for our purposes you'll only need to know a shortcut to switch between panes:

Ctrl-B <let go>; arrow-left/arrow-right

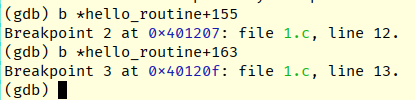
Panes can be also resized by holding ctrl while pressing arrows after letting go ctrl-b.

Our debugging session will execute all commands in "gdbscript" variable, which is where you can freely customize what's going on. By default, it will break on main to give us control.

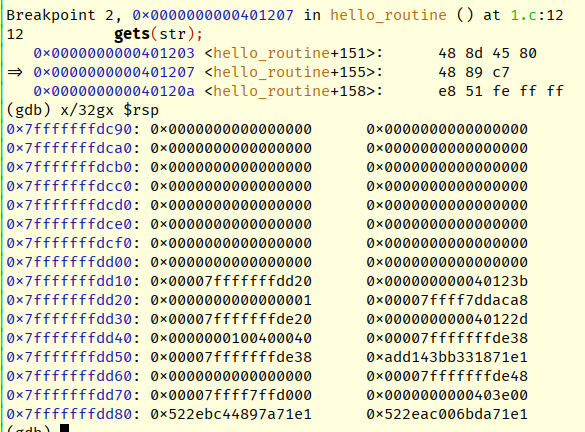


Remembering the slides, we'll want to explore how stack layout looks like before and after gets() call. To do so, we set relevant breakpoints and hit continue:

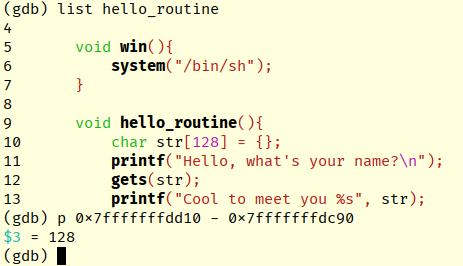




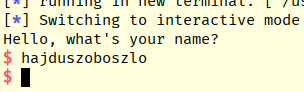
Immediately, we'll hit the breakpoint, lets explore first 32 elements on the stack:



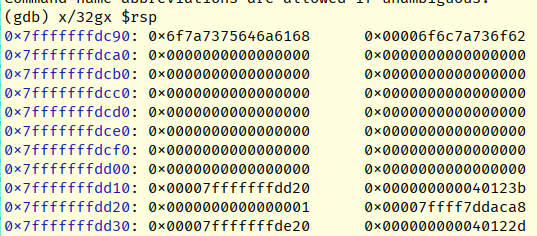
We can see lots of zeroes forming our buffer which is followed up by saved RBP and a saved instruction pointer. To confirm the size of a buffer, we can use print to do some math on the addresses:



Now let's hit continue again and provide some input to the program:

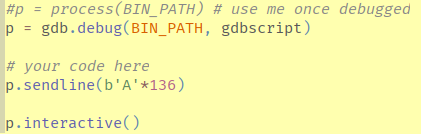


After hitting the breakpoint again, we'll see our buffer partially filled up:

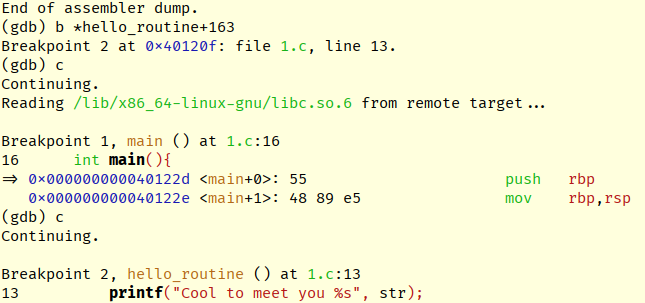




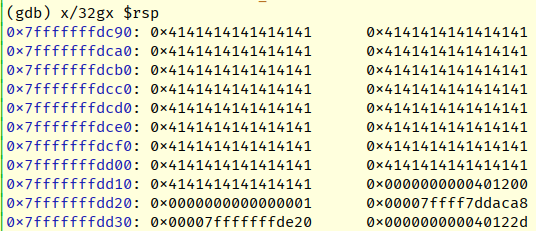
Great, lets continue the execution to the finish and edit our exploit. We'll now send a little bit over the size of our buffer, 136 times letter A:



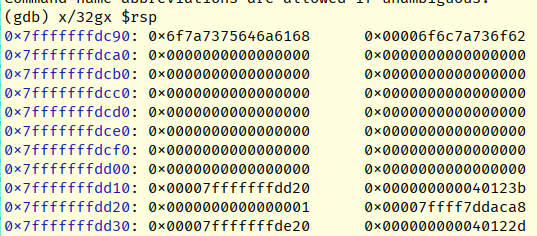
We now run the exploit again, remembering to set a breakpoint at hello+136 and continuing twice:



Looking into the contents of stack, we can clearly observe overwrite now. Old RBP is nowhere to be seen and keen eye will notice least significant byte in saved instruction pointer zeroed:

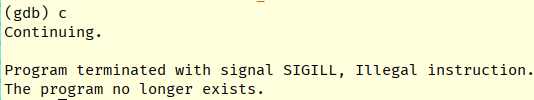


Original layout for reference:

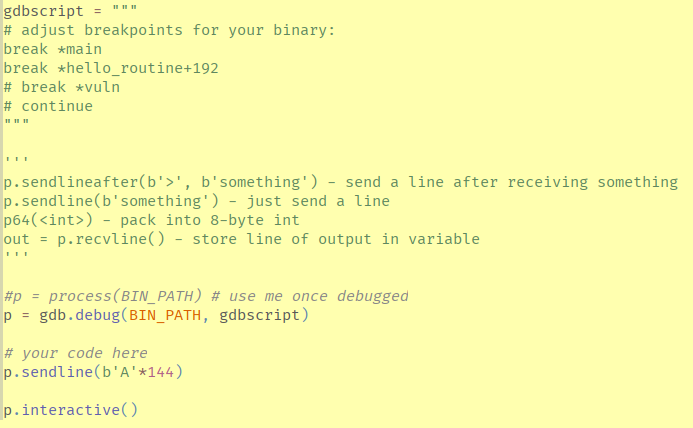


What happened is that gets fetches input up to the newline sign and then substitutes it with a nullbyte to form a proper null-delimited string. As a side-effect, this zeroed part of the return address, despite us providing exactly 136 bytes of input.

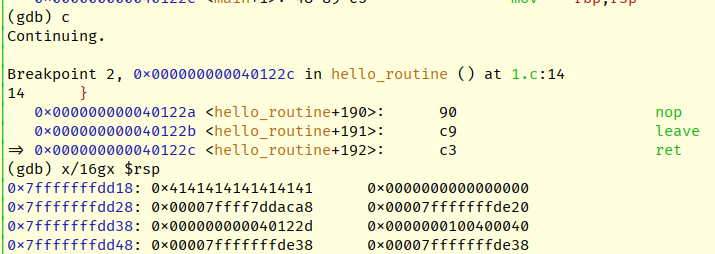
If we now continue executing the program, it will crash due to old RBP being corrupted to impossible address.



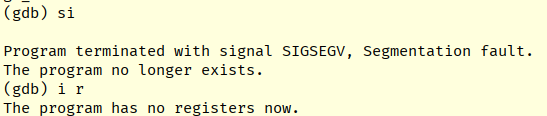
That’s good enough! Using the same logic, lets try overflowing with additional data. This time we'll also set a breakpoint at the RET instruction in the hello\_routine, which is located at +192 offset.



After running and continuing the execution twice, we'll explore the stack again:



We can clearly see, that we've overwritten more data and it looks like the return pointer is ours too! Remember how RET instruction works – it will fetch the address from top of the stack and insert it into RIP. Let's test this by single stepping



Program crashes violently with a different error. Segmentation fault means invalid memory access, which in this case means code tried to execute something it shouldn't.

This confirms our control of RIP and we can finally try to redirect execution to our win() function.

To obtain address of win(), we can use gdb with our target binary loaded. With debugging symbols present it is as simple as printing it:

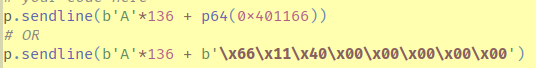


What we need to do is to place this address in our payload precisely, so that it overwrites exactly at the point where return pointer is. Since we've established that 136 bytes overwrite just over RBP, we can append our payload there.

Remember, that values on stack (and all addresses in 64-bit) have to be 8-byte aligned, meaning they have to be multiplies of 8-byte.

What does that mean for us?

We have to pass all addresses in our payloads in accordance with this requirement, we can do so in two ways:

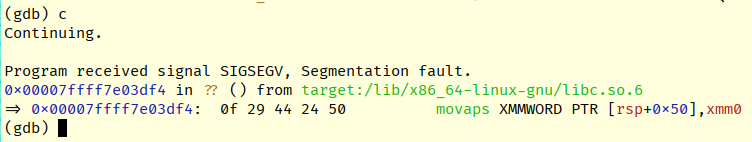


p64() is a small helper function provided by pwntools, which will take our integer and "pack" it into 8-byte little-endian integer.

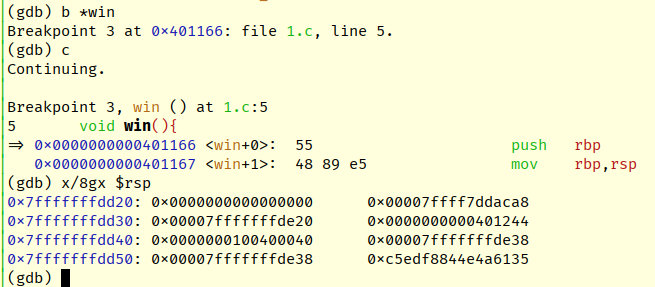
The other way is doing the same work by hand, supplying raw bytes ourselves.

Both are equivalent, using packing functions is obviously suggested for readability.

After updating our exploit with address overwrite, we execute aaaand… fail:



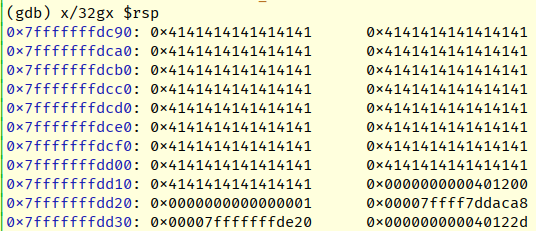
So, it turns out (and has been a major surprise for me too), that we actually do obtain control of RIP and redirect execution correctly but end up with misaligned stack for calling libc functions (such as system):



Because we RETurned into a function (which would be CALLed), there is a different alignment on stack than would be expected by libc functions. Apparently, these require 16-byte alignment.

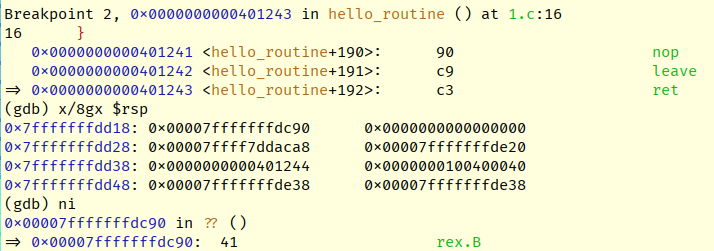
These limitations won't bother us. Since we have full control over the RIP register, we can just force the processor to interpret our input as instructions to be executed (remember – everything is data).

Let's circle back a little and try to do so. First thing we need is to obtain address of where our input is stored. Recall this screenshot:



Let's try to transfer execution here

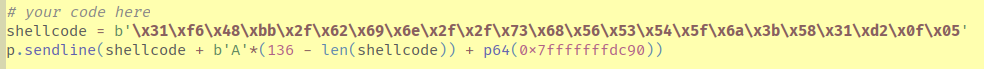




That's perfect, our data is being executed!

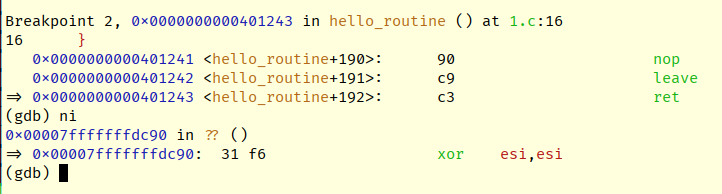
We'll be working on understanding shellcoding in the next chapter, for now let's just use shellcode provided in the shellcode.txt file.

You might be familiar with generating shellcode through msfvenom, this is exactly same thing, we will just use handcrafted one.

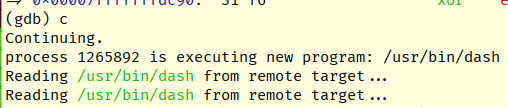


Pay attention to how we have to remember about padding our shellcode to maintain that 136-byte distance from buffer start to saved return address.

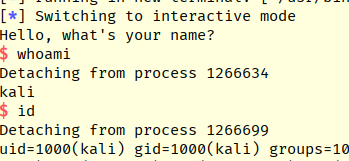
Let's launch the exploit and continue to the RET:



This is definitely different instruction from what we've seen before! Let's continue running:



Uh oh



Congratulations, you've just exploited your first buffer overflow!

Tasks

This lab is more about being getting your hands dirty. It will get confusing once you start doing the work yourself but that is your major learning moment, cherish it!

1. Go along through the walkthrough and reproduce all the steps till you get your first shell.
   1. Some addresses might change and differ from those on screenshots, as long as it works, we're good
   2. Once you get the shellcode on the stack, try examining it as instructions. Can you identify what it does?
      1. x/32i $rsp will be helpful
2. There is another binary in this folder with very similar vulnerability. Can you exploit it?