Kyle Oak
CTF Write Ups

Challenge 00:
In this first challenge we open it in radare2 to get an idea of what is going on.
First thing we see is our stack size of 0x40
We see that system is called, and it takes an input via gets to the address rbp-0x40
Then there is a cmp between 0xfacade and rbp-0x4, and if they are equal we get our flag
So all we have to do is write the difference in padding, and then 0xfacade, and we will perform a buffer overflow.

Challenge 01:
This challenge is very similar to the first, only this time the size of the stack is 0x60 and there is an fgets
We know that fgets will only take in the designated amount of data, so we need to use the fgets
Once we're through that we can send a new payload to the gets method the same way, to perform another buffer overflow attack
So the only changes we make to the code are the amount of padding and adding in a second p.sendline before the payload is sent.

Challenge 02:
This challenge has a vuln and a win function that appears to never be called
There are no cmp's like the last two challenges so we know we are not comparing to a fixed value 0xfacade again
There is a gets in the vuln function which means we can look to overflow the stack there
We overflow vuln in order for change the return address to jump to the win function when we reach ret
So we buffer in 0x44 + 0xc worth of padding, and then add onto the payload our win function address
This once again gives us a buffer overflow attack, but instead of having a cmp, we changed the return address to call the win function

Challenge 03:
Since PIE is enabled we can write shellcode to the stack. The stack location is given to us, so we use re.findall to the resp, and get our padding length. Since gets does not have a limit we can use that to receive our shellcode. So we have our padding, then the return address, then our shellcode just to keep it simpler than subtracting the shellcode length from the padding.

Challenge 04:
This is another buffer overflow problem. In radare we see in the main that vuln is called. We can then see that our var_8h is used by call rdx under the vuln function right after another fgets Now all we have to do is find the address of the win function and the offset, and use a buffer overflow attack. Once we know the offset is 74 we just send the 74 bits of padding followed by the address of the win function. I had trouble trying to figure out how to get the offset for this one,

which is still slightly confusing to me, but once I got that it was smooth sailing. Eventually someone suggested using re.findall() to find resp.

Challenge 05:
Again another buffer overflow problem
In this one vuln leaks us the address to main, which we use p.recv() to hold that info. We then calculate the offset of the win function from the main, which in this case is 0x13, or 19, which we then subtract from the main address. Once we have that it's the normal padding+address payload creation into buffer overflow. Same as the last one my main problem was getting the offset, this time it was a little harder.

Challenge 06:
Once again we use shellcode to perform a buffer overflow. In this one we take in the address with p.recv() and then send our asm shellcode with shellcraft.sh(). Once our shellcode is in we just send in the padding with the leaked address and we're done. Shellcode problems have been a little rough for me, their definitely my least favorite

Challenge 07:
Compared to the other shellcode problems this one seemed a lot easier, maybe it's because it was my third one. We just have to send a line to pass the first fgets and then send in our shellcode to the second.

Challenge 08:
This is our first write-what-where problem. We subtracted the target's address from puts' address and divided by 8 to get our where, and then our what is just the address of the win function. We make that into a payload and send it.

Challenge 09:
I couldn't make much of this one on radare and was kind of stumped. I didn't really know where to go except I saw things being XOR'd and compared until I looked at the writeups. I couldn't find much until the command xor(binary.string(binary.sym.key),"\x30") from the writeup. I do however now understand that when the xor'd value is equal to the key, that's when we get our shell.

Challenge 10:
This is like the older buffer overflow problems in the vuln function. We can see a cmp to 0xdeadbeef, so we just find the offset, set the padding, input deadbeef, and send the payload.

Challenge 11:
I had to go and review the printf vulnerabilities class notes, along with rewatching the lecture. Once we are able to find the offset, it just ends up being a right what where. Once we found the offset was 6 all that was left was to construct the payload like in the class notes and send it.

Challenge 12:
This one was the same as 11 except we got the main address from a leak. We just use re.findall and then the steps are the same as before with the same offset. While I felt I understood the core concepts behind this one, I don't think I would've been able to write out the exploit without the help of the write up.

Challenge 13:
This is another pretty straightforward buffer overflow problem, we find the padding, and then we just add on the address of the systemFunc function on the end.

Challenge 14:
I could not get this one, I saw a lot of my classmates were also struggling with this one so I left it till last. I definitely need to freshen up on the ROP Chain stuff and didn't really get anywhere without write ups so no exploit for this one.

Challenge 15:
I looked through the write ups for this one but I couldn't recreate something of my own accord. Like I said on some of the previous problems the shellcode problems are my weakest area, I definitely need some work.

Challenge 16:
I was confused by this one, I saw the XOR again but I couldn't figure it out. After looking at the writeups I am still confused on this one, the exploit seems too simple as it just sends the sym.key. I tried to do stuff like challenge 9 but nothing worked except the send command in the writeup. After asking the class they said there were no tricks, this is just straight forward. So we just needed to send the string of the key.

Challenge 17:
This one was my favorite, I always thought it was funny how the random function in C was the same if the time stayed the same, but I enjoyed it. I had to look up how to import the C stuff like in the writeup. We just use the fact that rand will give the same output if we use the same time, and then we're done.