DOGE

Yet again, executing the binary simply. Firstly the user is asked for the name then the program asks for the number of messages to be sent. Next we send a message and the program returns the same message just inverting each character’s case if applicable. This happens n number of times, where n is nothing but number user entered earlier. Next let us check the name for string formatting vulnerability. Alas, name is never returned back so we cannot utilize it as a data leak.

On the other hand, the response from ‘Doge’ to our messages is vulnerable to formatting.

Let us inspect the source code now, in the function ‘hello\_doge()’ vulnerability exists at the last statement i.e. ‘printf(buff)’

This buffer too cannot be overflown since ‘fgets()’ is only reading as many bytes as much is the size of the buffer. But we do have a good news too. The source code contains a function named ‘getFlag()’, this can give us the required flag. All we need to do now is to execute this function and we have only one attack factor present at our disposal the string formatting vulnerability in hello\_doge(). But since this statement runs in a loop it can be exploited ‘n’ number of times where again ‘n’ is nothing but the number entered by the user earlier. Though it is limited to 10 still it is more than enough.

To execute the required function we need to change the return address to location where getFlag function is located. To do this we will be utilizing the ‘%n’ string formatting. We can get the address of this function from object dump or disassembly of our given binary. It is <0x804878b>.

Now we just need to know where this address has to be saved. To know this we will be inspect in the running stack at the point where vulnerability occurs to find any address which points to a memory block of the stack itself. Using the command:

x/50wx $esp.

We printed 50 consecutive memory blocks of the stack at the point of vulnerability. We find that our input is located at an offset of 7 in the running stack. We can easily find the value of EBP by using the command ‘p $ebp’, which was <0xffffcff4> in my case. The return address is always located at the position <EBP + 4>. Thus the required address would be <0xffffcff8>. By looking at the 50 values we leaked from the stack we can find offset to an address stored in the stack itself that points to some random memory block in the running stack. We found one such address at the offset 14 which was <0xffffcfa4>. These addresses will keep changing on each execution but difference between them will always remain constant, which is [0xffffcff8-0xffffcfa4] equating to <0x54>.

So now we have the address of return address which is [<%14$x>+0x54]. We will be passing the first message to obtain our random address and then add 0x54 to it to obtain location where return address has been stored.

Next message that we pass, will be the payload that will be writing address of getFlag() function, <0x0804878b>, in place of return address. We will be writing 2 bytes at a time to reduce number which we have to write in vulnerable print statement. This is done to avoid overflowing the number itself. Our payload will be of the following form:

<ADDR>< ADDR+2>%<0x0804-0x8>x%<Offset>$hn%<0x878b-0x0804>x%<Offset+1>$hn

Here,

* ADDR is the location where return address is located and to which %n shall write.
* Offset refers to the offset of ADDR from top which is our own buffer. Thus Offset+1 refers to address of next four bytes of our input i.e. ADDR+2
* %hn is nothing but a special form of %n which only writes a 16 bit integer rather than 32 bit.
* 8 has been subtracted from first number since 8 bytes are already written while writing addresses prior to these arguments. Same goes for the next calculation.

After simplification our payload looks something like this:

<ADDR><ADDR+2>%2044x%7hn%32647x%8hn

This argument is passed to the vulnerable print statement will write 0x804878b at the address ADDR.

But still there is one more thing to be careful about. Doge processes the input given to it and changes cases from upper to lower and lower to upper wherever possible. Thus to invert those changes we will the passing a payload in the inverted cases. This changes our payload and finally we have our payload as:

<addr><addr+2>%2044X%7$HN%32647X%8HN

This payload will return flow of execution to getFlag() function which will print to us the contents of flag.txt which is required flag.