Assignment 1 – Antonio Artini - R00147013

Q1).

Part 1 – To find the private keys the commands:

grep -r “PRIVATE” \_Firmware3.bin.extracted

grep -r “PRIVATE” \_Firmware2.bin.extracted

were used. The results lead to a path to the /etc/intercept-server.key files for \_Firmware3.bin.extracted and \_Firmware2.bin.extracted, containing the private keys for both directories. A screenshot of a computer screen

Description automatically generated

A screenshot of a computer screen

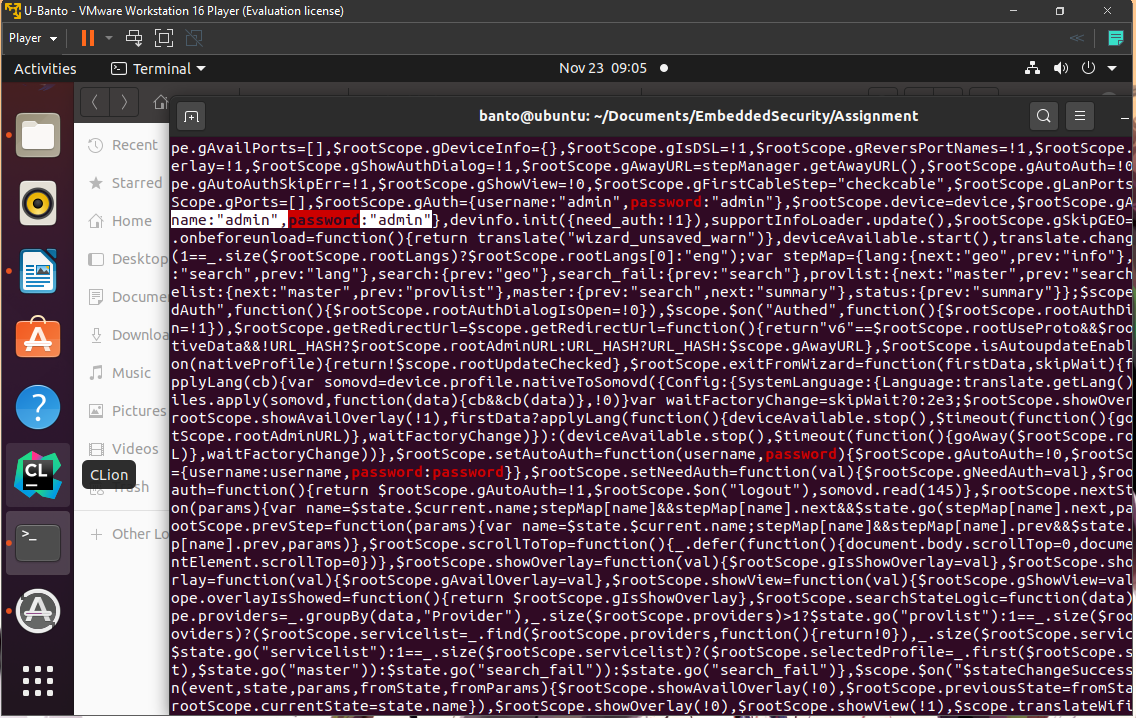
Description automatically generated

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

Part 2 – I used the command grep -r “password” \_Firmware2.bin.extracted to search the extracted Firmware2.bin to look for all the occurrences of password. After scrolling throughout the text, a credentials JSON object can be found containing “admin” as both the login name and password to Firmware2.bin.



Q2).

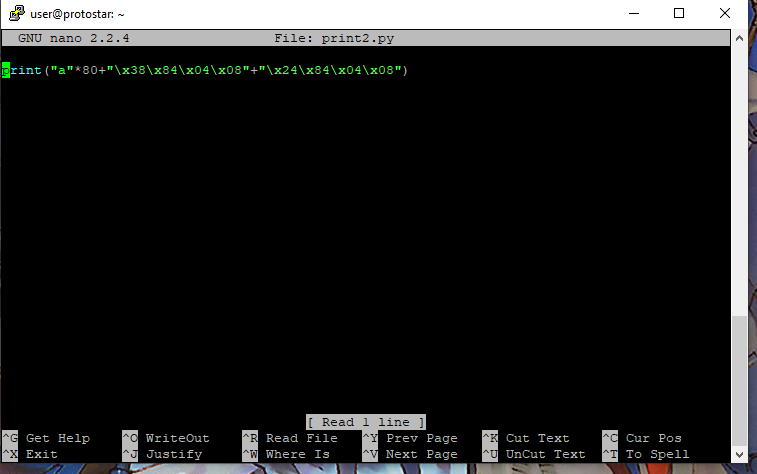
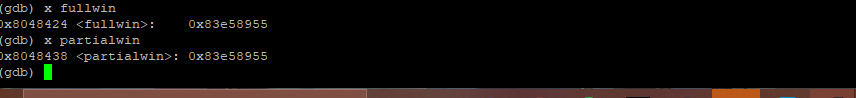
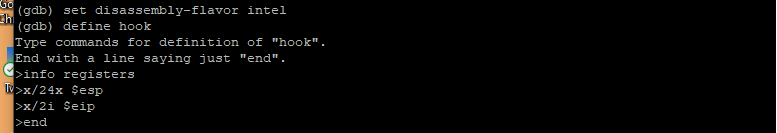
Hook was defined containing a set of commands to be run when debugging the compiled c file.

info registers 🡺 Shows the registers

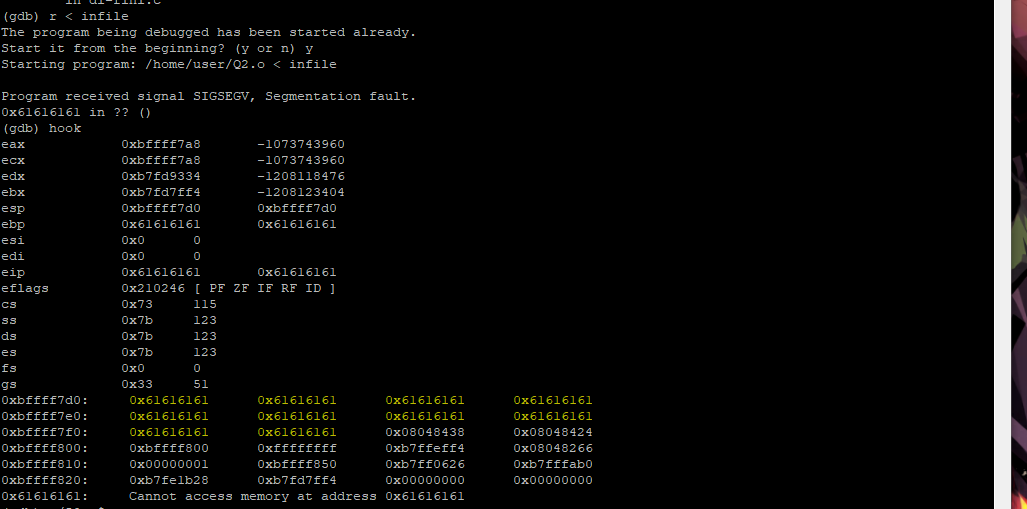
x/24x $esp 🡺 Examines the next 24 words at the top of the stack iin HEX

x/2i $eip 🡺 Shows the next 2 instruction pointers

x fullwin and x partialwin 🡺 Examines the address of a function in memory. In this case we examine fullwin and partialwin in memory.

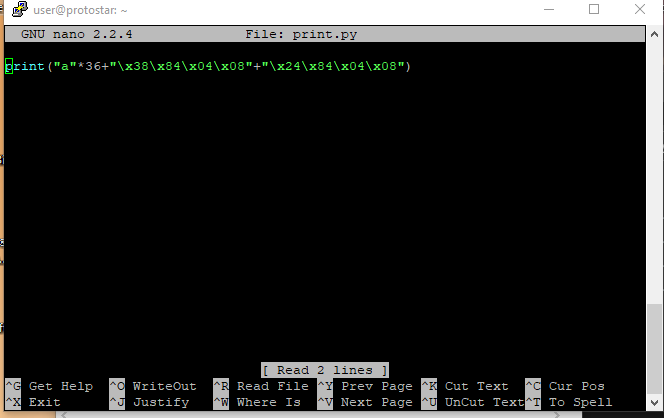


The python script is printing a string containing an overblown amount of a’s to overflow the char buffer in the compiled Question2.c, followed by the addresses of fullwin and partialwin in little endian. To try and figure out how far off the fullwin and partialwin address are from the return address location for main.

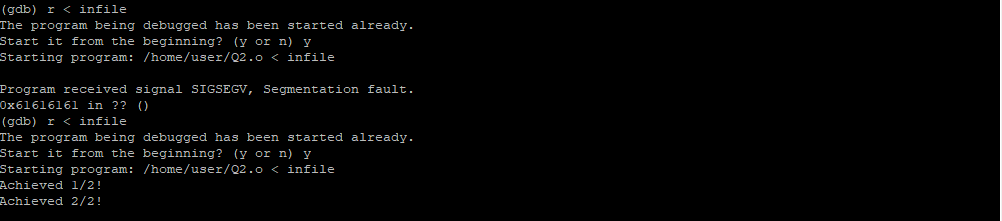


As can be seen in the above snip there are multiple 61’s indicating that the string of “a’s” is overflowing passed the return address of main (where we set a break point at before running the program). The above snip shows the next 24 addresses from the stack pointer. To get the functions to call, we have to remove the a’s until the function pointers are placed just before the first address on the stack above.

The correct location turned out to be the 37th and 38th address after overflowing from the buffer.



After feeding the output of this python script into the compiled c file we can see the two functions being called on execution, printing the contents of fullwin and partialwin.



Q3).

x target 🡺 The memory address of the target function

disas <function> 🡺 Disassembles a function showing the addresses of all the events inside the function.

info frame 🡺 lists the current frame. In the below snip, the command was used to find the start of the base pointer for the vuln function, showing the start of vulns stack frame. Once we have this return address, we need to change where the return address is pointing to, in this case it will be the address of the target function.



Text

Description automatically generated

Text

Description automatically generated

Struct iis used to convert the integer address to a binary string, in the case converting ext\_plt to binary.

The %x format specifier, in the exploit variable, is used to show the x’th value of the printf.

The %n format specifier is used to write a number of printed characters to the stack.

To find out how large the offset (134 in this case) needed to be, the padding had to be tinkered with similar to question2 until the correct location on the stack was found. The command r “$(cat infile)” was used this time, instead of feeding the output of the python script with r < infile, to Question3 on gdb.

A screenshot of a computer screen

Description automatically generated

The below snip shows a bunch blank space indicating that the exploit string was padded with 134,513,579 spaces – this figure overflowing the stack until we reach the correct spot on the stack to call our target function. This large number was found by converting the value of the target functions address (080483c4) to decimal and then making fine adjustments until the address of exit\_plt appears on the print after all the blank spaces.

A screenshot of a computer screen

Description automatically generated

Q4).

1).

Terminator

When an attacker attempts a buffer overflow, they're forced to over-write this canary value, which is set as 0 at the end of a buffer. The program can then detect that the canary has changed if the attacker writes over the value and takes appropriate actions. Any buffer overflow exploits that impact the stack frame will have to write through the canary to get at the return address. In order to keep the canary from changing, an attacker would need to include a 0 in the buffer overflow exploit input in a location that will cause the excess input to be ignored.

Random

A 32-bit number that is generated by another program when the original program is initialized. With this kind of canary, it is difficult for an attacker to know the value of the Canary unless they read it from the stack. However, this does not stop an attacker from manipulating a program on their own computer through a debugger such as gdb. It is still possible to read the canary through a debugger if the position of it is known to the attacker.

Random XOR

This canary is very similar to the Random canary. When this canary is placed on the stack it is the XOR value of a random 32-bit number, generated with the help of part of or all the control data, when the program is initialized. Whenever an attack is made on some part of the control data on the stack, they need to change the value of the canary too. To overcome this canary, an attacker needs to know the original canary, the algorithm used to scramble it and the new canary created from the scramble.

2).

EMSI

The random and terminator canaries are both susceptible to the EMSI vulnerability. This vulnerability is the overwriting of an address pointer to point to the return address. Once this pointer is being pointed to the return address the return becomes overwritten, becoming the return address to the first pointer, bypassing the protection from the canary. The Random XOR is a solution to this vulnerability as it checks against changes to the return address as well as the canary.

3).

They are effective with protecting from buffer overflows as they check for alterations in the stack. The terminator has a problem in which it is the most predictable canary to an attacker. It can be tricked by supplying the return address in a buffer overflow attack with a 0 value. The random canary has the issue in which it is still possible to read from stack by the attacker. Once the attacker knows the position of the canary, they can trick the program into believing that the stack was manipulated. Both also suffer from the EMSI vulnerability where there return addresses point elsewhere to other addresses on the stack. The Random XOR Canary is the most effective to prevent buffer overflow as they are much harder to detect on the stack and checks are made after each function call.

4).

Canaries are not effective against format string exploits as they have a vulnerability with the printf, in particular with reading the stack with the %x format string exploit and writing with the %n format string exploit. Canaries are effective with protecting from attacks with buffer overflows but with printf this can be gotten around.