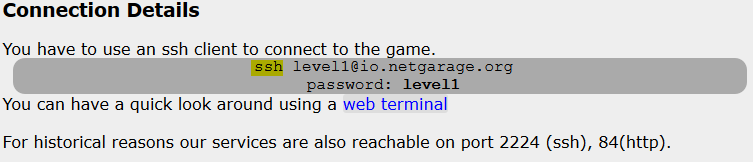
**IO.Netgarage.org Walk-through**

Introduction

IO is the most mature game, but is never the less in continually updated as technology develops. It provides recent radare2 and gdb builds.

Level 1

Initially Level 1 username and the password is given in the Io.netgarage.org web site and what we have to do is just open putty on our PC and ssh to the given address.



Now we are in Level 01.

In each level there is a directory called **levels** and inside that directory there are multiple files related to each level. At the moment we don’t have any permission to open or execute those files. Among those files Level01 is our target file and it doesn’t have any source code. That means we have to use reverse engineering methods to get into Level02.

Using “**file**” command we can identify that this Level01 is an executable file.

Using “**string**” command we can get the string outputs from the file Level01.

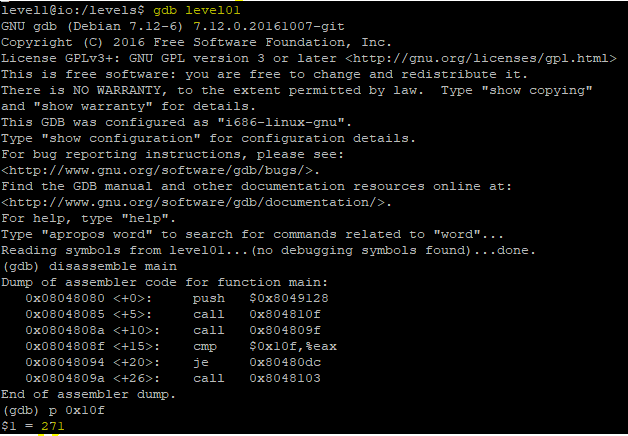
 Policies

Assume that you are the chief information security officer who needs to secure a large organization with multiple departments, and each department requires dedicated compute infrastructure and security policy rules. Design a security policy for the enterprise’s human resources management department. following:

1. Identification and authentication policies - Specifies authorized persons that can have access

We can reveal that program will be asking for a 3-digit passcode.

Let’s try to run a program under “**gdb**” and disassemble the main function.

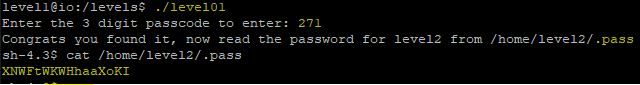


In there we can see that the program compares a fixed value with the value of register “**eax”**

Fixed value is 0x10f and by looking at that we can assume this is a hexadecimal value.

Let’s convert it into decimal value using **“p”** command in gdb.

It gives value as 271 and we can try it with **level01** file.

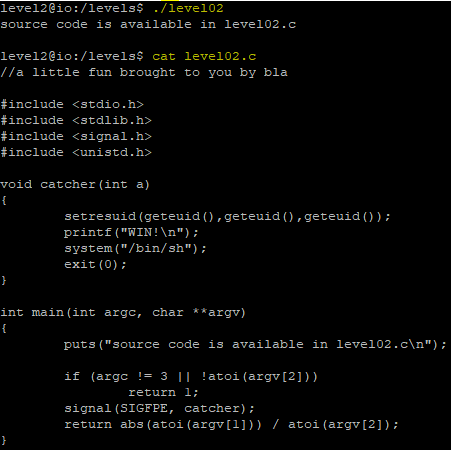


After that step we are free to view “**. pass**” hidden file inside the level2 directory which is located inside the home directory. Using “**cat**” command we can retrieve the password for level2.

Level 2

Now we are in Level2 and inside the “**levels**” directory there is an executable file called level02. Once we execute that file, it will display that “**source code is available in level02.c**”.

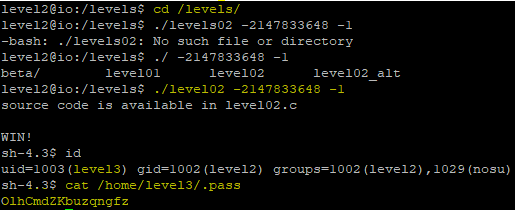
Let’s view the above source code (level02.c).



By reading the source code of level02.c file we can say that,

1. The number of args must be 2, (argv [0] being the caller's name).
2. The two arguments should be numbers
3. The catcher function will be called on the event SIGFPE (launched for example for a division by 0)
4. The return value of the function is argv [1]/argv [2]

We need to execute “**catcher**” function instead of “**SIGFPE**”. For that we need to use an integer value outside of the bound of the integer definition. So most-negative value to be out of range is -2147483648 (sqrt ( -2147483648) = 2147483648). Now we are sending the value – 2147483648 to the abs, the result will be also -2147483648 (because of binary max bound and negative values).

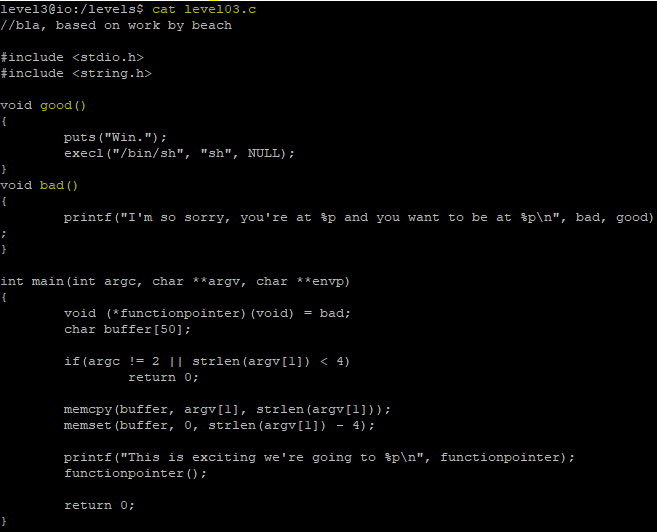


Same as the previous level just **cat** the.pass file inside the **level3** directory which is located inside the **home** directory.

Now we have a password for level3.

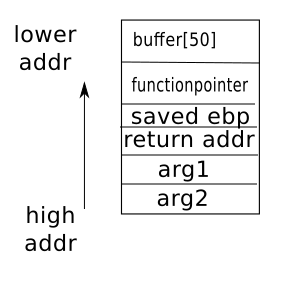
Level 3

In here as well we have to view the source code of **level03.c** file.



Simply we can see that we need to execute the **good** function instead of **bad.** Mean is this program introduce the concept of buffer overflow to overwrite nearby local variable.

Stack Hierarchy.



The variable **functionpointer** is store below the variable **buffer**.

And there is a command called **memcpy.** After referring the man page of memcpy we can see that did not restrict length of argv [1] to be copied to buffer. Hence, by copying more than what the buffer [50] can hold, we start to overwrite the memory below it. This is known as **buffer overflow**.

**Step 01**

Disass main

0x0804855a <+146>: mov eax,DWORD PTR [ebp-0xc]

0x0804855d <+149>: mov DWORD PTR [esp+0x4],eax

0x08048561 <+153>: mov **DWORD PTR [esp],0x80486c0**

0x08048568 <+160>: call 0x80483ac <printf@plt>

0x0804856d <+165>: mov eax,DWORD PTR [ebp-0xc]

By referring the source code, we know that **printf** function will have a string as argument1 and **functionpointer** as argument2.

DWORD PTR [esp], 0x80486c0 means that esp+0x4 address will assigning the value of 0x80486c0.

(DWORD= 4bytes).

Let’s see what is included in memory address 0x80486c0.

gdb-peda$ x/s 0x80486c0

0x80486c0: "This is exciting we're going to %p\n"

Now we can confirm that argument1 definitely contain the string. That means **functionpointer** is located at **esp+0x4**.

0x0804856d <+165>: mov eax,DWORD PTR [ebp-0xc]

0x08048570 <+168>: call eax

From this we know that **functionpointer** is at **ebp-oxc**

**Step 02**

Finding buffer

Same as previous step we can do the same to find the buffer.

0x08048519 <+81>: mov eax,DWORD PTR [ebp+0xc]

0x0804851c <+84>: add eax,0x4

0x0804851f <+87>: mov eax,DWORD PTR [eax]

0x08048521 <+89>: mov DWORD PTR [esp+0x4],eax

0x08048525 <+93>: lea eax,[ebp-0x58]

0x08048528 <+96>: mov DWORD PTR [esp],eax

0x0804852b <+99>: call 0x804838c <memcpy@plt>

Buffer is at ebp-0x58

**Step 03**

Finding the actual buffer size

For this we need to find the distance between **buffer** and **functionpointer.**

0x58-0xc

**76**

So actual size of buffer is 76 bytes.

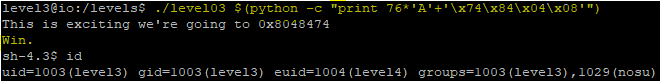
**Step 04**

Finding the address of **good**

Using **print** command, I’m going to find the address of function **good.**

E:\SLIIT\4th year-2nd semester\OHTS\assingment01\level03-02.PNG

Now we know the address of **good** function and we can launch a buffer overflow attack.

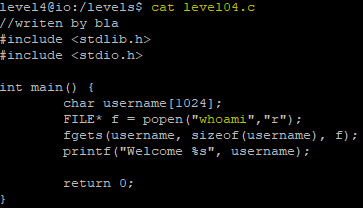


**Step 05**

E:\SLIIT\4th year-2nd semester\OHTS\assingment01\level03-06.PNGRead the password from **.pass** file.

Level 4

Same as the previous levels we can look at the source code of the **level04.c** file for get an idea about the level04.



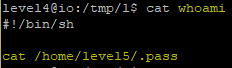
By looking at the deep of the above result we can see that **whoami** file has been called without using an absolute path.

The OS will look at $PATH for list of directories to search the binary for in a left to right order. When it finds a binary that has the requested name, it will execute that binary.



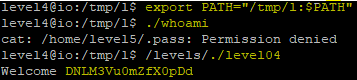
So we can run our own **whoami** program that spawn a shell. As long as our directory comes before the actual location of **whoami**, our binary will be run instead.

Let’s create our own **whoami** program.



Now make a directory called **new** and put above **whoami** file inside that new directory.

Edit **PATH** variable as our new directory.



Compile the program with the same name as **whoami**.

Finally run the **level04** file inside the levels directory for gaining the password for level05.