**CIS-449 Intro to Software Security**

**With Professor Dr. Anys Bacha**

**Assignment 2**

**Student: Demetrius Johnson**

**13 March 2023**

**Due: 15 March 2023 at 9am**

# **Table of Contents**

[**Table of Contents** 2](#_Toc129740403)

[Abstract 4](#_Toc129740404)

[Methodology (Code/Commands/Results) 4](#_Toc129740405)

[TASK 1: Manipulating Environment Variables 4](#_Toc129740406)

[Use printenv or env command to print out the environment variables. If you are interested in some particular environment variables, such as PWD, you can use: 4](#_Toc129740407)

[Use export and unset to set or unset environment variables. It should be noted that these two commands are not separate programs; they are two of the Bash’s internal commands (you will not be able to find them outside of Bash). 9](#_Toc129740408)

[TASK 2: Passing Environment Variables from Parent Process to Child Process 10](#_Toc129740409)

[Manual of fork() function: 10](#_Toc129740410)

[Step 1 – child case 11](#_Toc129740411)

[Step 2 – parent case 14](#_Toc129740412)

[Step 3 – differences between child and parent case (and conclusion) 15](#_Toc129740413)

[TASK 3: Environment Variables and execve() 15](#_Toc129740414)

[Step 1 16](#_Toc129740415)

[Step 2 17](#_Toc129740416)

[Step 3 (conclusion about how execve() program works) 18](#_Toc129740417)

[TASK 4: Environment Variables and system() 19](#_Toc129740418)

[TASK 5: Environment Variable and Set-UID Programs 22](#_Toc129740419)

[Step 1 – write a program to print environment variables of current process 22](#_Toc129740420)

[Step 2 – compile program and change ownership to root 23](#_Toc129740421)

[Step 3 – Run the setUID program 23](#_Toc129740422)

[TASK 6: The PATH Environment Variable and Set-UID Programs 24](#_Toc129740423)

[Special note: 29](#_Toc129740424)

[TASK 7: The LD PRELOAD Environment Variable and Set-UID Programs 29](#_Toc129740425)

[Step 1 create a shared library object and myprog which calls that library through an environment variable 29](#_Toc129740426)

[Step 2 – myprog under various conditions 31](#_Toc129740427)

[Step 3 32](#_Toc129740428)

[TASK 8: Invoking External Programs Using system() versus execve() 33](#_Toc129740429)

[Step 1 – exploit the system() call 33](#_Toc129740430)

[Step 2 – use execve() to stop exploit 33](#_Toc129740431)

[TASK 9: Capability Leaking 35](#_Toc129740432)

[Conclusion 38](#_Toc129740433)

[Supplemental Material: 38](#_Toc129740434)

[How to SSH into VirtualBox VM from another local machine on the same network as the local machine hosting the VM 38](#_Toc129740435)

# Abstract

In this lab, we explore how the setUID mechanism and environment variables can affect a system, specifically the Linux kernel. Whenever a setUID program is called that is root owned, we need to be careful since a user might be able to change environment variables in order to gain root access. This happens because all environment variables contain paths to programs that a process should look into when it calls a program. These environment variable paths can be easily manipulated by an outside user to exploit a privileged program if that program does not have proper protections in place. For example, it is good to use a function such as execve() instead of system(), since execve() will not fork to a shell to invoke a program, but it will instead overwrite the current parent with the called process data. You also need to look out for capability leaking, since when a privileged programs finishes execution, although it might deescalate its privilege before forking to a child process, if privileged files that were opened were not close, that child will inherit those open files and have access to them.

# Methodology (Code/Commands/Results)

## TASK 1: Manipulating Environment Variables

### Use printenv or env command to print out the environment variables. If you are interested in some particular environment variables, such as PWD, you can use:

* + "**printenv PWD**" or "**env |grep PWD**".

Manual for **env** program (notice it is a program which will inherit the exported environment variables from its parent process):

Text

Description automatically generated

Manual for **printenv** program (notice it is a program which will inherit the exported environment variables from its parent process):

Text

Description automatically generated

Using **printenv:**

Text

Description automatically generated

Using **env:**

Text

Description automatically generated

As shown, both **env** and **printenv** printed the same output 🡪 they are both child processes of the Bash shell program, and they print out their current environment variables, which they derived (inherited) from the variables marked for export in the parent (Bash shell) process.

Now using **printenv PWD** and piping env output to grep **env | grep PWD** to print out a specific environment variable:

Graphical user interface, text

Description automatically generated

### Use export and unset to set or unset environment variables. It should be noted that these two commands are not separate programs; they are two of the Bash’s internal commands (you will not be able to find them outside of Bash).

Here, I create an environment variable named **TEST** for the current Bash shell process that is running – I store the root path **/** in that variable; notice, before I export it, it exists only in the parent (Bash shell) process and will not be output as part of the environment variables of **env** since it was not marked as an export variable in the parent process (Bash shell); that is why **env** child process does not show **TEST** as part of its environment variables. After marking it for export, then you notice **env** program does inherit **TEST** environment variable and outputs it accordingly:

Text

Description automatically generated

Notice again it shows up when you run **printenv** since it is exported:

Text

Description automatically generated

Now, when I issue **unset** on **TEST** exported environment variable, it is deleted, thus **env** will not show **TEST** since it did not inherit it:

Text

Description automatically generated

## TASK 2: Passing Environment Variables from Parent Process to Child Process

### Manual of fork() function:

Text

Description automatically generated

Text

Description automatically generated

Notice the **PID of the child process** is returned in the PARENT, and **0** is returned in the CHILD.

Remember: Fork system call is used for creating a new process, which is called child process, which**runs concurrently** with the process that makes the fork () call (parent process). After a new child process is created, both processes will execute the **next instruction** following the fork () system call.

(source: <https://www.geeksforgeeks.org/fork-system-call/>)

### Step 1 – child case

Here, I have compiled myprintenv.c using gcc compiler; it results in a binary called **a.out**. Then, I run **a.out** and redirect its output to a file called **file.** I note that in the code, **extern environ** 2D char array is declared (extern variables are global variables that can be declared multiple times but can only be initialized only once); thus it takes on the values of the originally declared **environ** variable, which contains all the strings of all the environment variable names and associated values. I notice that this part of the code,

Text

Description automatically generated

Actually creates a child process, and both parent and child will begin execution at switch(value), which is the next instruction after childPid = fork().

Text

Description automatically generated

Here is the output of **file**:

Text

Description automatically generated

I notice all environment variables were printed out, and I notice an environment variable called **‘\_’** is assigned path **./a.out,** which is current directory and then executable **a.out.**

### Step 2 – parent case

I used **vi** editor and commented out the code of the child case and uncommented the **printenv**() code in the parent case:

Text

Description automatically generated

Then save the file and recompile it to another executable, and then run that executable (**a.out2**) and redirect it to another file called **file2**:

Text

Description automatically generated

Now, here is the output of my **file2:**

Text

Description automatically generated

I notice that it appears that the output for the parent condition is exactly the same as environment variables printed out when the child condition was satisfied in step 1.

### Step 3 – differences between child and parent case (and conclusion)

Text

Description automatically generated

After running the **diff** command to compare my two files, I notice that they are in fact *identical* (except of course for the executable name from which the file was generated with the redirect operator.

## TASK 3: Environment Variables and execve()

*The function execve() calls a system call to load a new command and execute it; this function never returns. No new process is created; instead, the calling process’s text, data, bss, and stack are overwritten by that of the program loaded. Essentially, execve() runs the new program inside the calling process.*

### Step 1

*Please compile and run the following program, and describe your observation. This program simply executes a program called /usr/bin/env, which prints out the environment variables of the current process.*

Here, I compiled **myenv.c** and named the executable **a.out3**, then I ran **a.out3** and I notice that it did not output anything, thus **execve()** does not pass its environment variables to the process it calls, namely the **env** process, thus **env** has no environment variables to print out and when all of the results are passed up the chain back to the bash shell, there is an empty string to print.

Here is the flow: SHELL calls 🡪A.OUT3 calls 🡪EXECVE calls 🡪ENV

* Where A.OUT3 *inherits* SHELL environment variables;
* EXECVE process *inherits* A.OUT3 environment variables;
* But ENV *does not inherit* EXECVE environment variables.

Thus ENV passes empty string to EXECVE, which passes empty string to A.OUT3, which passes empty string to SHELL.

Text

Description automatically generated

### Step 2

*Change the invocation of execve() in Line ➀ to the following; describe your observation.*

Original:

Text

Description automatically generated

Changed (change NULL to environ in the execve() function):

Text

Description automatically generated

Then I recompiled myenv.c:

Text

Description automatically generated

Now here is the output of the recompiled program:

Text

Description automatically generated

As I notice, it does output the environment variables.

### Step 3 (conclusion about how execve() program works)

So, the extern global variable **environ** contains the list of environment variables of the calling process **a.out4**, which inherited the environment variables from the calling **shell** program. When **execve()** is called, not only is a program name argument provided, but also the second argument which is how the function passes along environment variables the child should inherit, which means that **execve()** will make sure that the child process that it creates inherits all of the values passed into that parameter, namely the strings stored in the **environ** variable that was passed in:



## TASK 4: Environment Variables and system()

*If you look at the implementation of the system() function, you will see that it uses execl() to execute /bin/sh; execl() calls execve(), passing to it the environment variables array. Therefore, using system(), the environment variables of the calling process is passed to the new program /bin/sh. Please compile and run the following program to verify this.*

I used the **touch** command to create a file I call **task5\_system\_call.c (I meant to call it task 4)**, then I edit the file and add the code required for task 5:

Text

Description automatically generated

Text

Description automatically generated

Here, I rename the file from **task\_5\_system\_call.c** to **task4\_system\_call.c** so I do not get confused later in the lab:

Text

Description automatically generated

Now I will compile and run program, then run it:

Graphical user interface, text

Description automatically generated

Text

Description automatically generated

Above, I notice that environment variables did print out as expected since ultimately when system() is called it leads to execl() which calls execve() **AND** passes the environment variables array to it.

## TASK 5: Environment Variable and Set-UID Programs

*Set-UID is an important security mechanism in Unix operating systems. When a Set-UID program runs, it assumes the owner’s privileges. For example, if the program’s owner is root, when anyone runs this program, the program gains the root’s privileges during its execution. Set-UID allows us to do many interesting things, but since it escalates the user’s privilege, it is quite risky. Although the behaviors of Set-UID programs are decided by their program logic, not by users, users can indeed affect the behaviors via environment variables. To understand how Set-UID programs are affected, let us first figure out whether environment variables are inherited by the Set-UID program’s process from the user’s process.*

### Step 1 – write a program to print environment variables of current process

*Write the following program that can print out all the environment variables in the current process.*

I create a file called task5\_printenv.c, then I edit the file and add my code:

Text

Description automatically generated

Text

Description automatically generated

### Step 2 – compile program and change ownership to root

So now, I have changed ownership of the executable to root, then I changed mode to 4755, meaning 100 111 101 101 (set uid ON, set GID OFF, stickybit OFF; owner can RWX, group and others can R and X). Notice executable is red since it is a SET UID program.

Text

Description automatically generated

### Step 3 – Run the setUID program

*In your shell (you need to be in a normal user account, not the root account), use the export command to set the following environment variables (they may have already exist):*

I note from running **printevn** that **PATH** is the only environment variable that is set (for export). For the other two, I will still export them; for the LD\_LIBRARY\_PATH variable, I do not expect it to be exported since I never defined it (if it is defined with a value, then I expect export to work); for my variable (**ANY**\_**NAME**) I will define it to point to root directory so I expect it will export.

Text

Description automatically generated

Text

Description automatically generated

Now after running my a.out\_task5 program that I compiled earlier, here is the output:

Text

Description automatically generated

…

Text

Description automatically generated

I notice above that only **PATH** and **ANY**\_**NAME** variables were passed to the function, as I expected.

## TASK 6: The PATH Environment Variable and Set-UID Programs

Here are the current paths that **PATH** environment variable is storing

Text

Description automatically generated

Now I will add to the beginning (the first in the hierarchy) of the PATH variable a new path, which will be a path to my user directory (seed):

Text

Description automatically generated

Notice now that path is updated to be itself, but with another path added to the front, namely **/home/seed**, which is the path to my user profile directory. Now that we have that set up, we can now write a program and compile it with the same name that is a commonly used program by a user, for example **ls** or **cd**, so that when a child process is created, it will check the /home/seed directory *first* for any program that is called from that child process.

Now I will compile this program that will call **ls** command using relative path, then change its owner to **root**, and I will exploit this fact to gain root privilege:

Text

Description automatically generated

Graphical user interface, text

Description automatically generated

Now I change owner to root and make it a set UID program:

Text

Description automatically generated with medium confidence

Now I will call my program **a.out\_task6**, and it returns the contents of calling the **ls** program found from one of the paths from the **PATH** environment variable: SHELL calls a.out\_task6 which calls ls program.

Text

Description automatically generated

Now I will write a program and compile it with the name **ls**, and place it in the path I added to the **PATH** environment variable so that when I run my program that calls **ls** using relative path, it will check my directory in /home/seed and run my **ls** program before it checks the remaining paths in the PATH variable; I will make my program the shell program **sh** to try and get root access in a shell:

Text

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

Now I have copied and moved my compiled **ls** program to **/home/seed**, the path I added to **PATH** environment variable:

Graphical user interface, text

Description automatically generated with medium confidence

Now I will attempt to run the program that calls **ls** using relative path:

Text

Description automatically generated

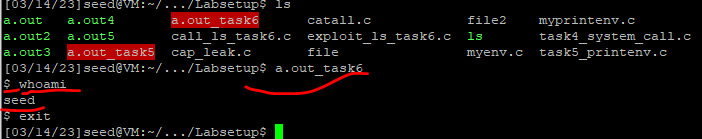
Notice, I did get a shell, this is because effective UID and real UID are the same, as shown above using **whoami** and **id** programs (uid only prints if EUID == RUID).

I know what to do: I must change ownership of the ls program in my **/home/seed** to be owned by **root** and make it a setUID program, so that the program is ran with privileges of the owner (root, as I will set it):

Text

Description automatically generated

Now that my ls program is owned by root and is a setUID program, it should give me root access when I run the setUID program that calls **ls** using relative path:



It still did not work, but I see the special not about sh being linked to **dash** which prevents the attack, so I need to link **sh** to **zsh** program that was written for this lab; then when I ran the program which calls **ls** by relative path, I have root access! It worked!

Text

Description automatically generated

### Special note:

*Note: The system(cmd) function executes the /bin/sh program first, and then asks this shell program to run the cmd command. In Ubuntu 20.04 (and several versions before), /bin/sh is actually a symbolic link pointing to /bin/dash. This shell program has a countermeasure that prevents itself from being executed in a Set-UID process. Basically, if dash detects that it is executed in a Set-UID process, it immediately changes the effective user ID to the process’s real user ID, essentially dropping the privilege. Since our victim program is a Set-UID program, the countermeasure in /bin/dash can prevent our attack. To see how our attack works without such a countermeasure, we will link /bin/sh to another shell that does not have such a countermeasure. We have installed a shell program called zsh in our Ubuntu 20.04 VM. We use the following commands to link /bin/sh to /bin/zsh:*

***$ sudo ln -sf /bin/zsh /bin/sh***

## TASK 7: The LD PRELOAD Environment Variable and Set-UID Programs

*In this task, we study how Set-UID programs deal with some of the environment variables. Several environment variables, including LD PRELOAD, LD LIBRARY PATH, and other LD \* influence the behavior of dynamic loader/linker. A dynamic loader/linker is the part of an operating system (OS) that loads (from persistent storage to RAM) and links the shared libraries needed by an executable at run time. In Linux, ld.so or ld-linux.so, are the dynamic loader/linker (each for different types of binary). Among the environment variables that affect their behaviors, LD LIBRARY PATH and LD PRELOAD are the two that we are concerned in this lab. In Linux, LD LIBRARY PATH is a colon-separated set of directories where libraries should be searched for first, before the standard set of directories. LD PRELOAD specifies a list of additional, user-specified, shared libraries to be loaded before all others. In this task, we will only study LD PRELOAD.*

### Step 1 create a shared library object and myprog which calls that library through an environment variable

1. Write function for dynamic library:

Text

Description automatically generated

Text

Description automatically generated

1. Compile program as dynamic:

Text

Description automatically generated

1. Now create environment variable LD\_PRELOAD and set it equal to current directory, and my .so library (shared object):

Text

Description automatically generated

1. Finally, compile the following program myprog, and in the same directory as the above dynamic link library libmylib.so.1.0.1:

Here is the code for **myprog.c**

Text

Description automatically generated

Here is the code where I create myprog.c file using **touch**, and then I edit it using vi editor, finally I compile the program using **gcc:**

Text

Description automatically generated

### Step 2 – myprog under various conditions

1. Make myprog a regular program, and run it as a normal user.
   * Text

     Description automatically generated
2. Make myprog a Set-UID root program, and run it as a normal user (I did chown **root** and chmod 4755)
   * Text

     Description automatically generated
3. Make myprog a Set-UID root program, export the LD PRELOAD environment variable again in the root account and run it.
   * 
   * 
4. Make myprog a Set-UID user1 program (i.e., the owner is user1, which is another user account), export the LD PRELOAD environment variable again in a different user’s account (not-root user) and run it.

I create a user profile using adduser and name the user user1, then I did a cat on /etc/passwd to see if the user was created:



Text

Description automatically generated

Now I need to set a password for user1:

Text

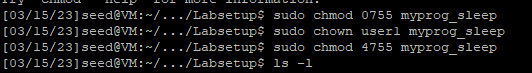
Description automatically generated

Now switch to the user:

Text

Description automatically generated

Now I change permissions to non set UID then change owner user1, then set program back to a setUID program (because it must be done in this order, since when you make a program a setUID program owner needs to be set to who you want it to be first):





Now I set LD\_PRELOAD and run the program as **seed** user and here is the output:

Text

Description automatically generated

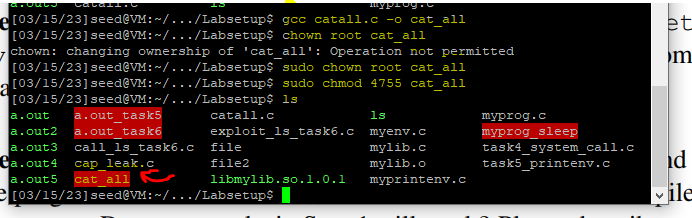
### Step 3

*You should be able to observe different behaviors in the scenarios described above, even though you are running the same program. You need to figure out what causes the difference. Environment variables play a role here. Please design an experiment to figure out the main causes, and explain why the behaviors in Step 2 are different. (Hint: the child process may not inherit the LD \* environment variables).*

For step 2, only when myprog ran as a regular program by a normal user did it print out *I am not sleeping*. This is because

## TASK 8: Invoking External Programs Using system() versus execve()

### Step 1 – exploit the system() call



Yes, we can exploit it by using the command separator, which is a semicolon **;**

So, I can call the function and pass in the filename parameter like this: filename;myothercommand :

Notice I run cat\_all on catall.c as parameter, but pass in another command **/bin/sh** in the string; the filename is invalid, but the next command will still run, as shown; also note what I pass to the function needs to be in “” because otherwise it would run each command from my shell instead of passing the entire string as one input to the catall function:

Text

Description automatically generated

Notice I successfully gained root access, since second command in the string parameter passed to cat\_all is /bin/sh, and it is a setUID program, so that when /bin/sh is called it checks EUID and see that it is root, so I get a root shell. Note /bin/sh is NOT a set UID program; but remember EUID is often what is checked by programs to seed determine who the program runs as/under.

### Step 2 – use execve() to stop exploit

Now I will comment out system and use execve instead:

Text

Description automatically generated

Text

Description automatically generated

Now recompile my code:

Text, chat or text message

Description automatically generated

Try to do the exploit again:

Text

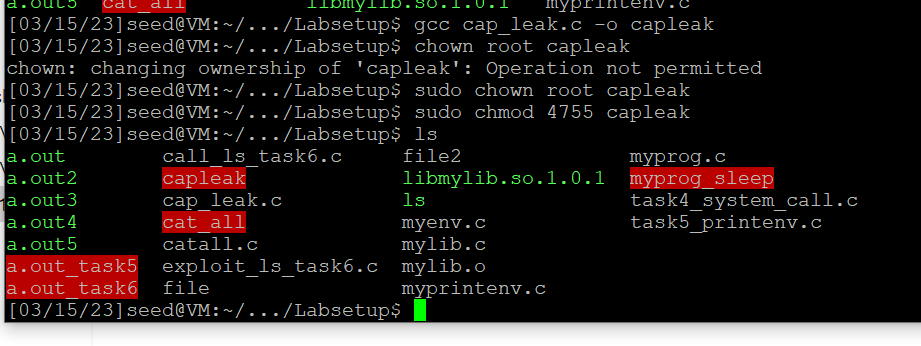
Description automatically generated

*As shown no root access is gained; this is because system(command) merged data and code together, however execve() separates the data (filename) from the code (string parameters passed in), so that when /bin/cat is invoked through a shell, it will actually search for the string “aa;/bin/shell” as the filename since that string is only 1 parameter, not a single string represent several parameters (filename + arguments as command stored).*

## TASK 9: Capability Leaking

*To follow the Principle of Least Privilege, Set-UID programs often permanently relinquish their root privileges if such privileges are not needed anymore. Moreover, sometimes, the program needs to hand over its control to the user; in this case, root privileges must be revoked. The setuid() system call can be used to revoke the privileges. According to the manual, “setuid() sets the effective user ID of the calling process. If the effective UID of the caller is root, the real UID and saved set-user-ID are also set”. Therefore, if a Set-UID program with effective UID 0 calls setuid(n), the process will become a normal process, with all its UIDs being set to n. When revoking the privilege, one of the common mistakes is capability leaking. The process may have gained some privileged capabilities when it was still privileged; when the privilege is downgraded, if the program does not clean up those capabilities, they may still be accessible by the non-privileged process. In other words, although the effective user ID of the process becomes non-privileged, the process is still privileged because it possesses privileged capabilities. Compile the following program, change its owner to root, and make it a Set-UID program. Run the program as a normal user. Can you exploit the capability leaking vulnerability in this program? The goal is to write to the /etc/zzz file as a normal user.*

Compile program and make root as owner and make it a set UID program:



Create the zzz file as per the instruction of cap\_leak and make it so users can only read:

Text

Description automatically generated

Text

Description automatically generated

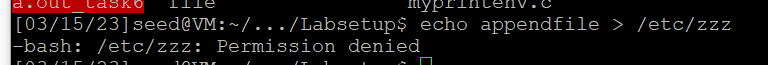
Text

Description automatically generated

Text

Description automatically generated with medium confidence

So notice permission is denied if I try to write (append) to the file:



Now I will show that capleak leaves the privileged file open so that in the child shell process that capleak spawns I can access and write to the file even as a normal user (seed):

Text

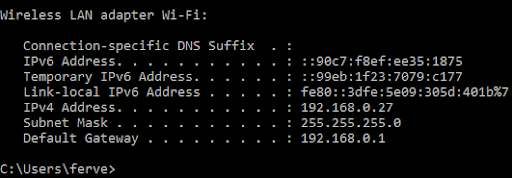
Description automatically generated

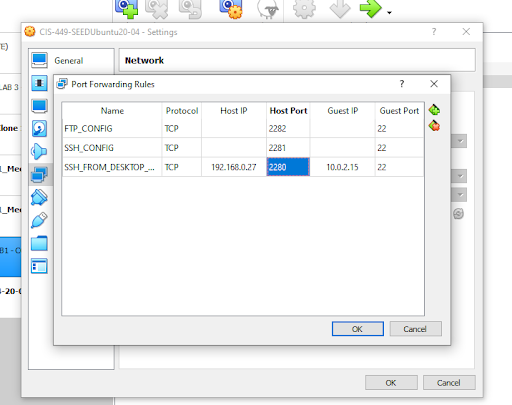
# Conclusion

Overall, I learned about how important it is for a programmer to be aware of privileged programs and to be aware of the environment variables that their program may consume or programs associated with it may consume. Such considerations are important when programming as it can create vulnerability in a system. Also, we need to make sure that we are closing files, since capability leaking can allow a non privileged user to still access an unclosed file in a child process.

# Supplemental Material:

## How to SSH into VirtualBox VM from another local machine on the same network as the local machine hosting the VM

1. You must be on the same local network as the host machine (the machine that is running the VM through Virtual Box) (unless you set up port forwarding on your home router network to the local host machine).
2. You must have IP address of the local host machine.
   * On windows for example, my IP address is 192.168.0.27:
   * 
3. You must have SSH client software (such as PuTTy) on the remote machine (by remote in this case, it means the machine that is not running VirtualBox to host the VM, but still on the same network as the host machine).
4. The VM should be configured as a NAT network (have a network adapter on the machine configured with NAT):
   * Graphical user interface, application

     Description automatically generated
5. Now, go to Port Forwarding and add a new translation rule:
   * 
   * Notice I added the IP address for HOST IP as the local IP address of the host machine that is running the VM through VirtualBox, and I picked 2280 to be the listening port.
   * Then, Guest IP and port is the local IP address of the network adapter of the VM, in this case it is 10.0.2.15 (I used ifconfig command in Linux Ubuntu), and listening port is 22 (for ssh service which uses port 22)
   * Now, we can connect to the VM from another local machine on the same network as the local host machine running VirtualBox.
   * Note: you could leave HOST IP and GUEST IP blank and only fill in the port, this simply means Virtual Box (which is running on the local host) will listen to the loopback address (127.0.0.1) on the local address and forward it to the loopback address of the VM; this is how to SSH into VM from the same local machine that is hosting the VM.
6. Use PuTTY from another machine in the local network (in my case, I used my desktop computer):
   * Graphical user interface, application

     Description automatically generated
   * Notice, seed is the username.
   * Also notice, connection address and port is the address of my laptop (the host machine running the VirtualBox VM) and the port configured in VirtualBox port forwarding settings for my specific VM. Now, my laptop (192.168.0.27) will see these requests from my desktop (some address on same network, i.e. 192.168.0.xx) and then forward them to the virtual machine at socket ip=10.0.2.15 port=22, accordingly.
7. And it worked!:
   * Text

     Description automatically generated