Android Device Rooting Lab

NAME: JASHWANTH REDDY GANGULA

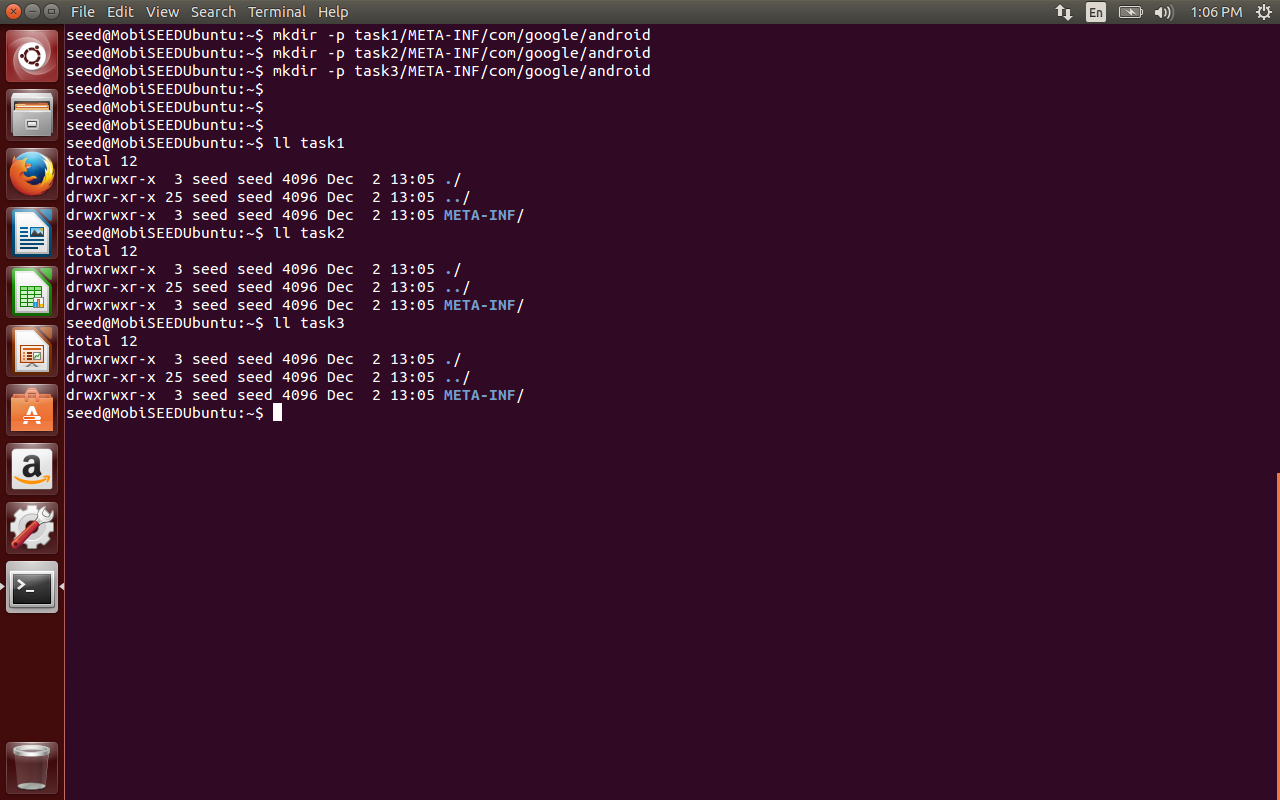
SUID: 646254141

Task 1: Build a simple OTA package

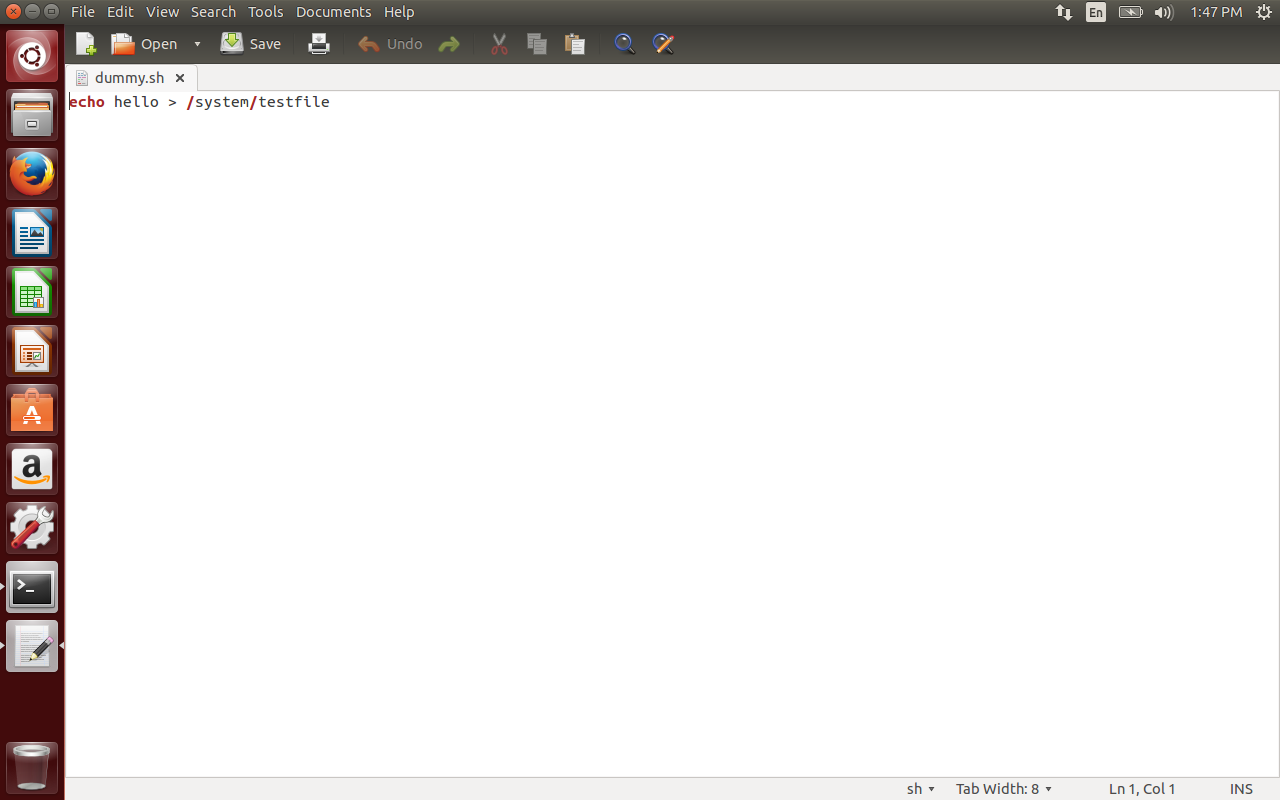
In this task we have to create a file inside /system. But we need root privileges for the file creation. We create dummy.sh script as “echo hello > /system/dummy” , and put it inside /system/etc/init.sh so that when the Linux boots up and conducts system initialization( including starting daemon processes), it runs will root privileges and we insert our dummy.sh inside init.sh.

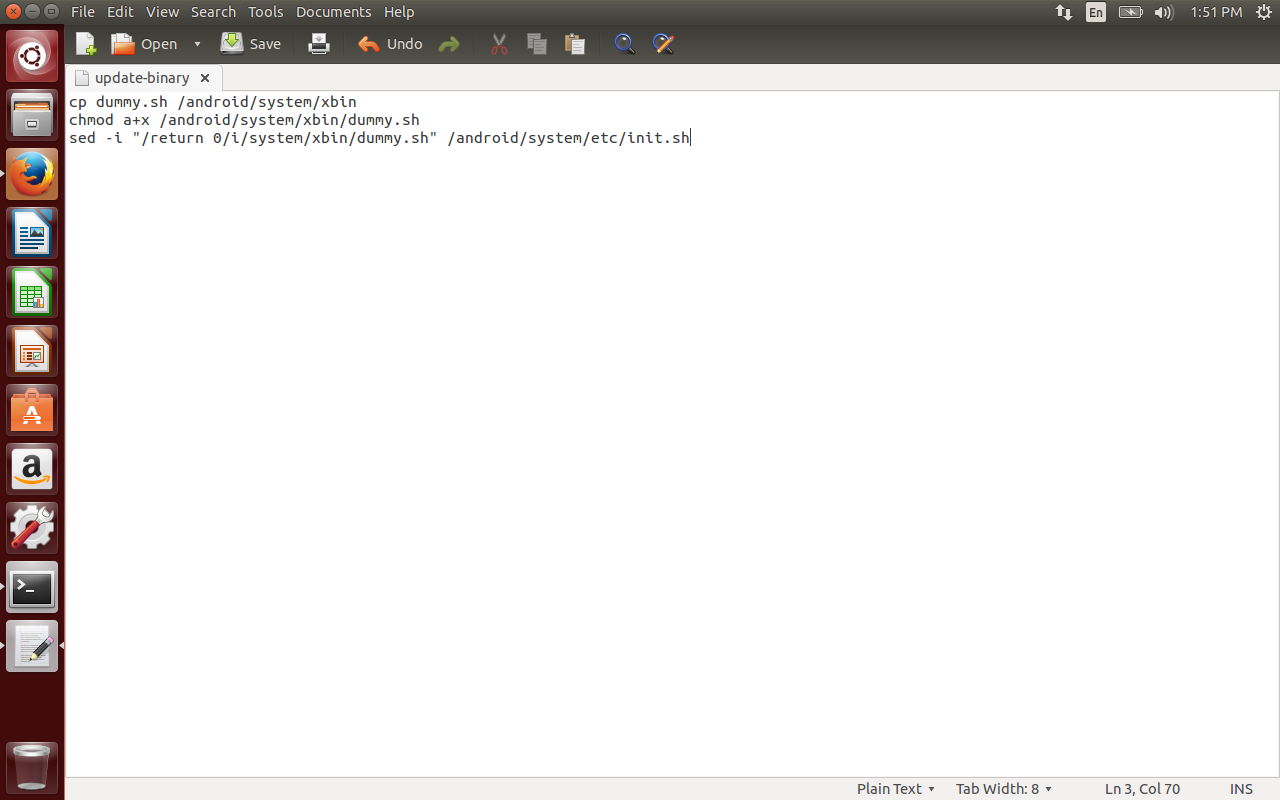
The OTA package searches for text strings in init.sh and adds the dummy.sh before the last line of init.sh file to execute the script.

First we create the OTA package with three folders as below.

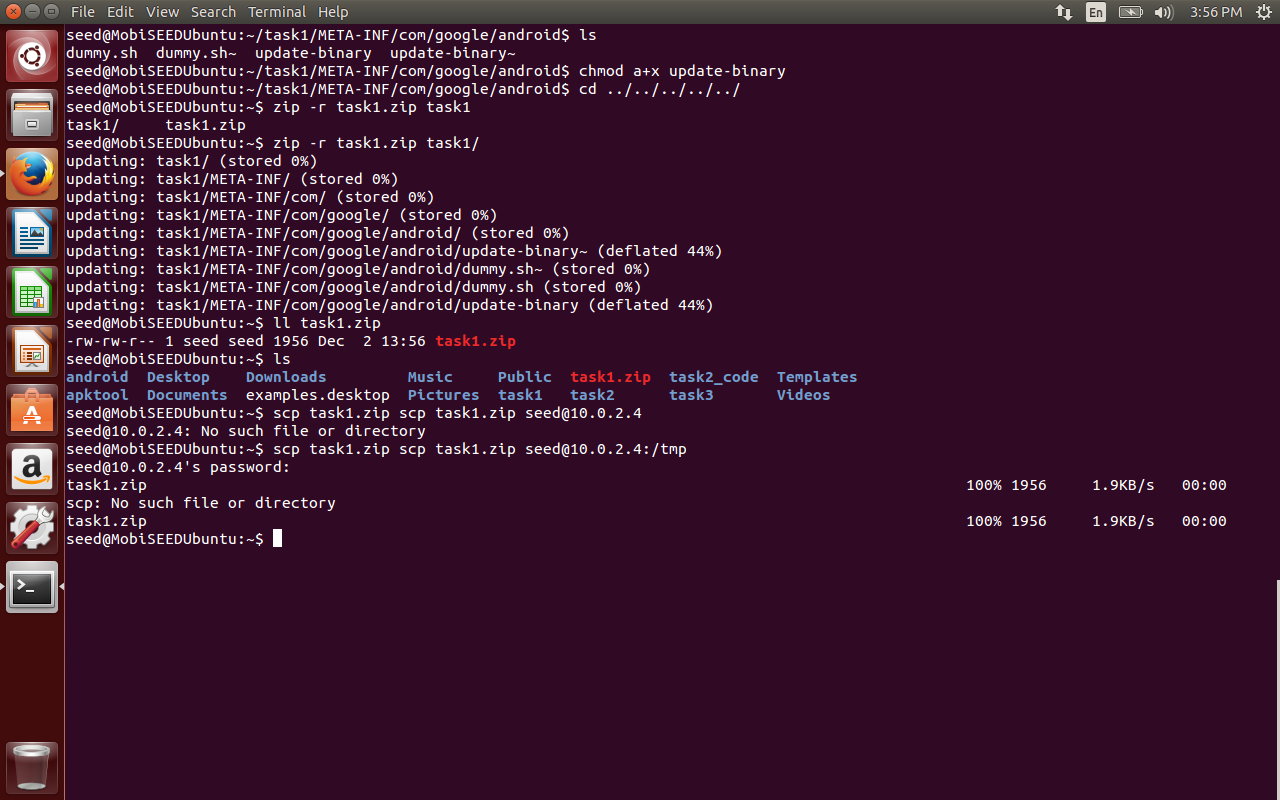


Create a dummy.sh inside task1/META-INF/com/google/android as follows:

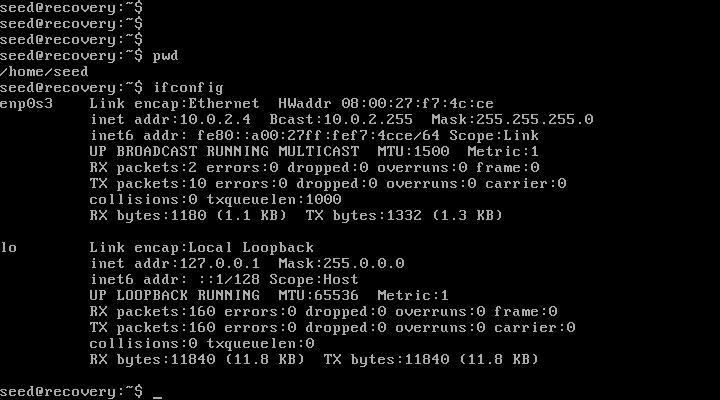


Create update-binary script Create a dummy.sh inside task1/META-INF/com/google/android as follows to place dummy.sh inside init.sh file as below.  


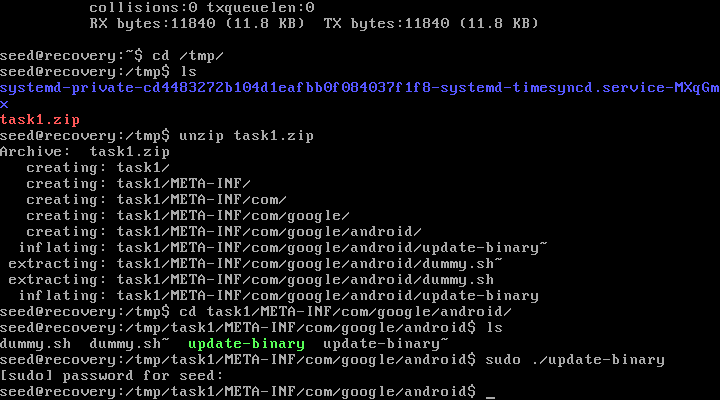
Now make update-binary as executable. Get out of the task1 folder and build the OTA package.   
and send it to the Recovery OS in Android.



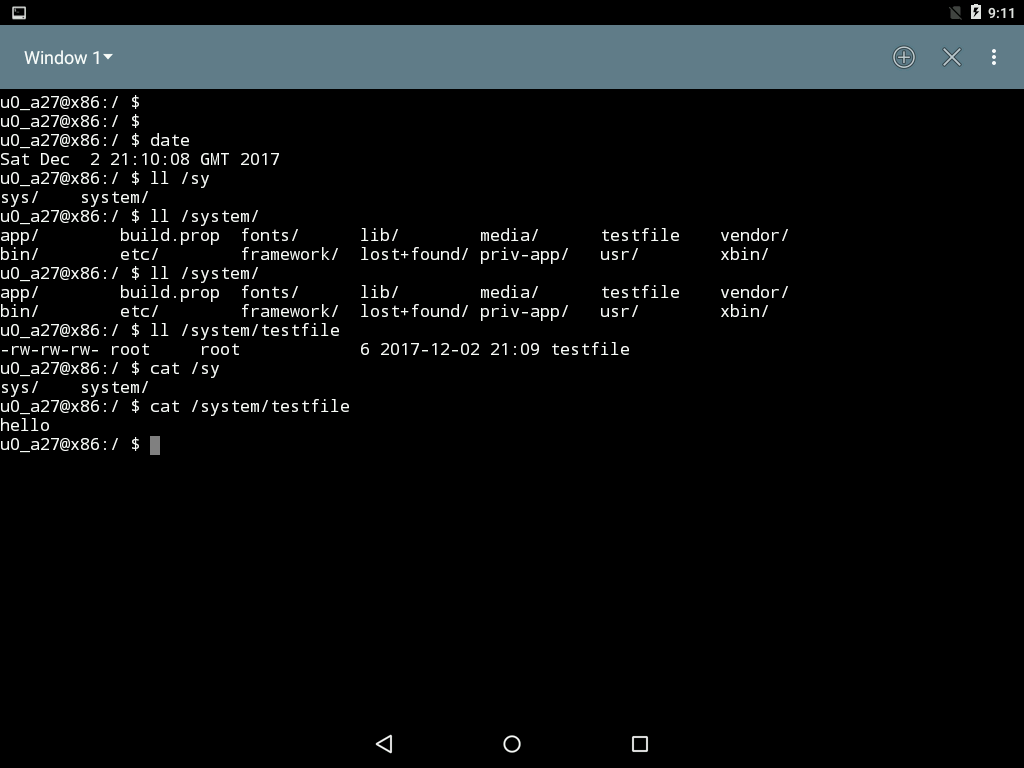
Now I have restarted the Android VM with the recovery OS(Ubuntu). The ip is found to be 10.0.2.4. Now send the OTA package from VM to recovery OS.



Ideally the recovery OS should automatically update the changes, but since we are running Ubuntu as our recovery OS, we do it manually here. (The recovery OS updates packages only from trusted vendors by verifying signatures etc.) But in our case recovery OS installed bypasses all the verifications and we are able to update changes as per our wish. Now we unzip the OTA package and execute the update-binary script as below and reboot the machine to observe the testfile created inside the system.



Now in the below image we see a testfile created in the /system of android.



Task 2: Inject code via app process

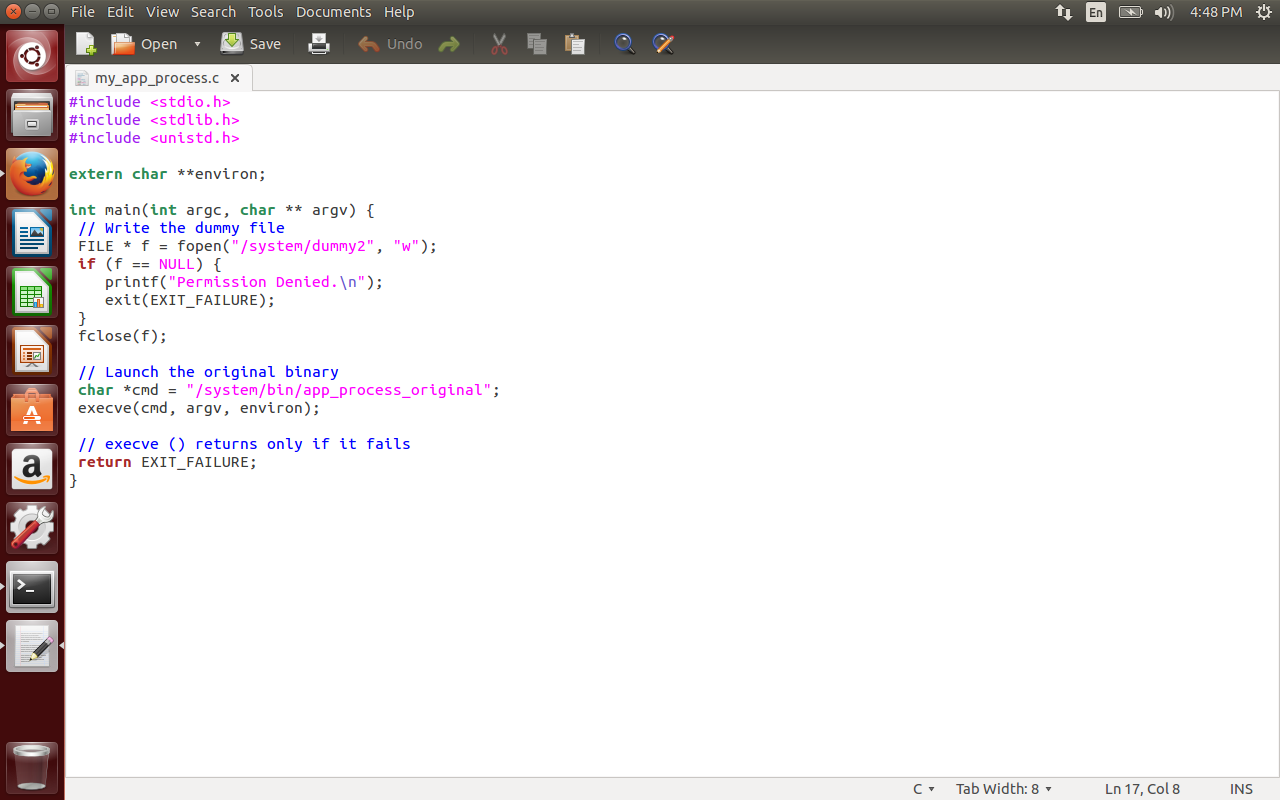
In the previous task, the code is injected into initialization script during Linux OS bootup time. In this task we would like to inject the malicious code during android bootstrap time

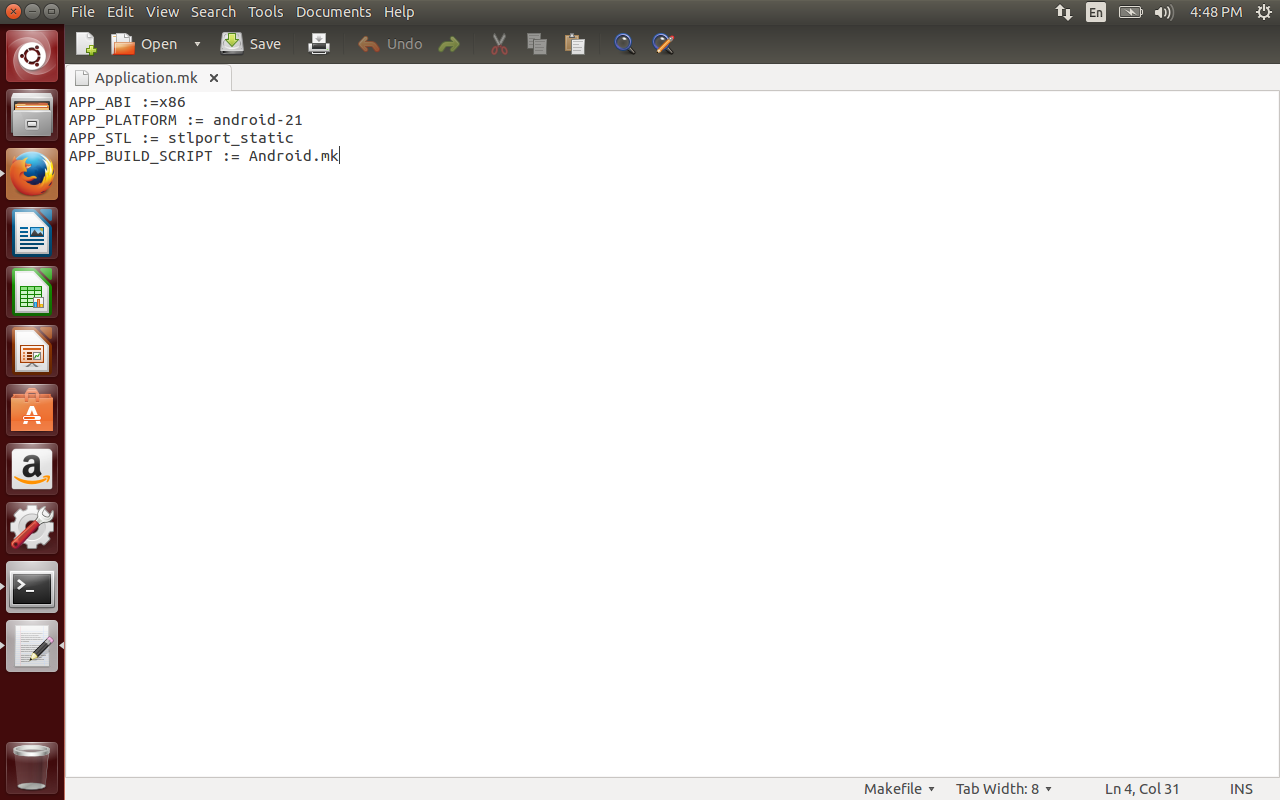
Android runs its bootstrap on the top of linux, once the linux part is initialized. When the android bootstrap runs, it runs a process called app\_process using the root privilege. This starts the zygote daemon, which runs the Dalvik Virtual Machine runtime and the server applications etc. The zygote is the parent of all the process. Our goal is to modify app process, so in addition to launch the Zygote daemon, it also runs something of our choice. Similar to the previous task, we want to put a dummy file (dummy2) in the /system folder to demonstrate that we can run our program with the root privilege.

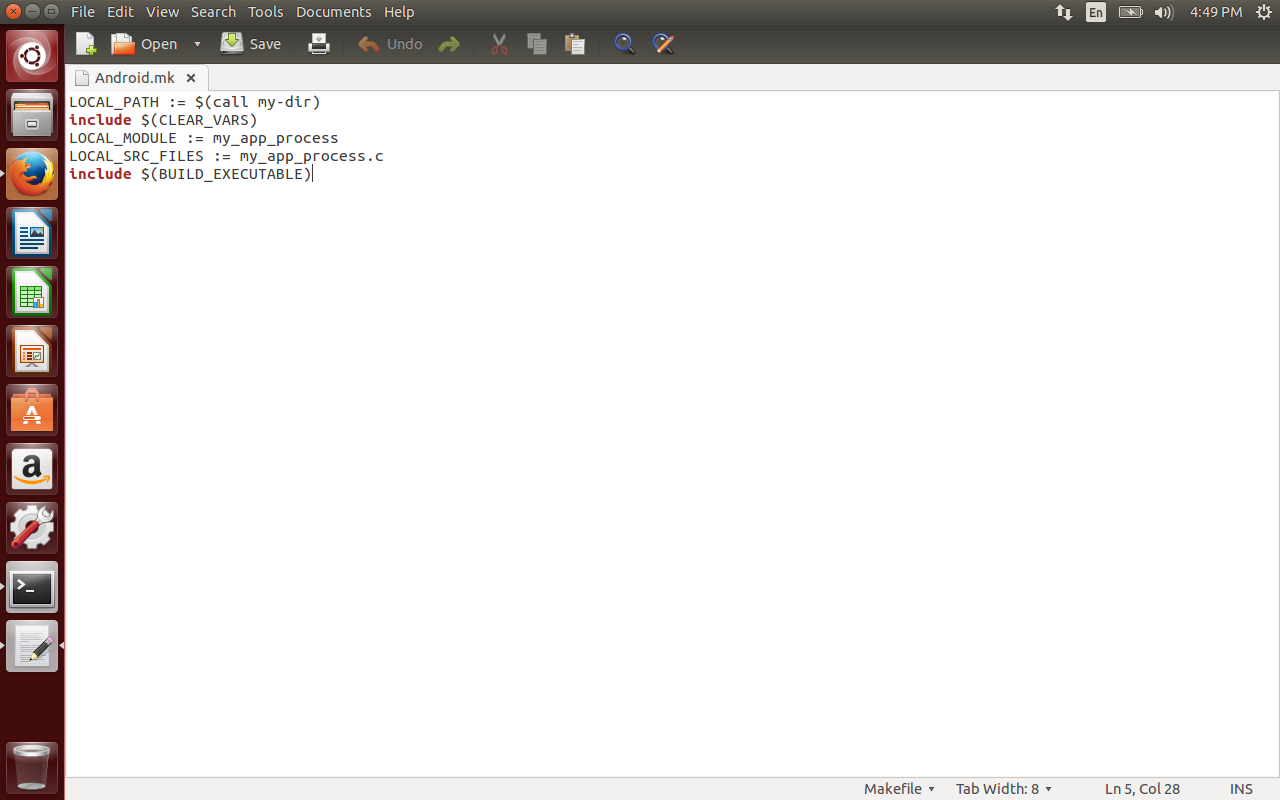
We will rename the original app\_process to app\_process\_original and call our wrapper program as app\_process. In our wrapper app\_process program, we write something to dummy file and then invoke the original app\_process program.

We compile the above program in MobiSeedUbuntu VM as we need NDK (Native Development Kit) in our seed Ubuntu VM. NDK allows us to compile C and C++ programs for Android OS.

Create my\_app\_process.c , Application.mk, Android.mk with the below content inside task2\_code folder.

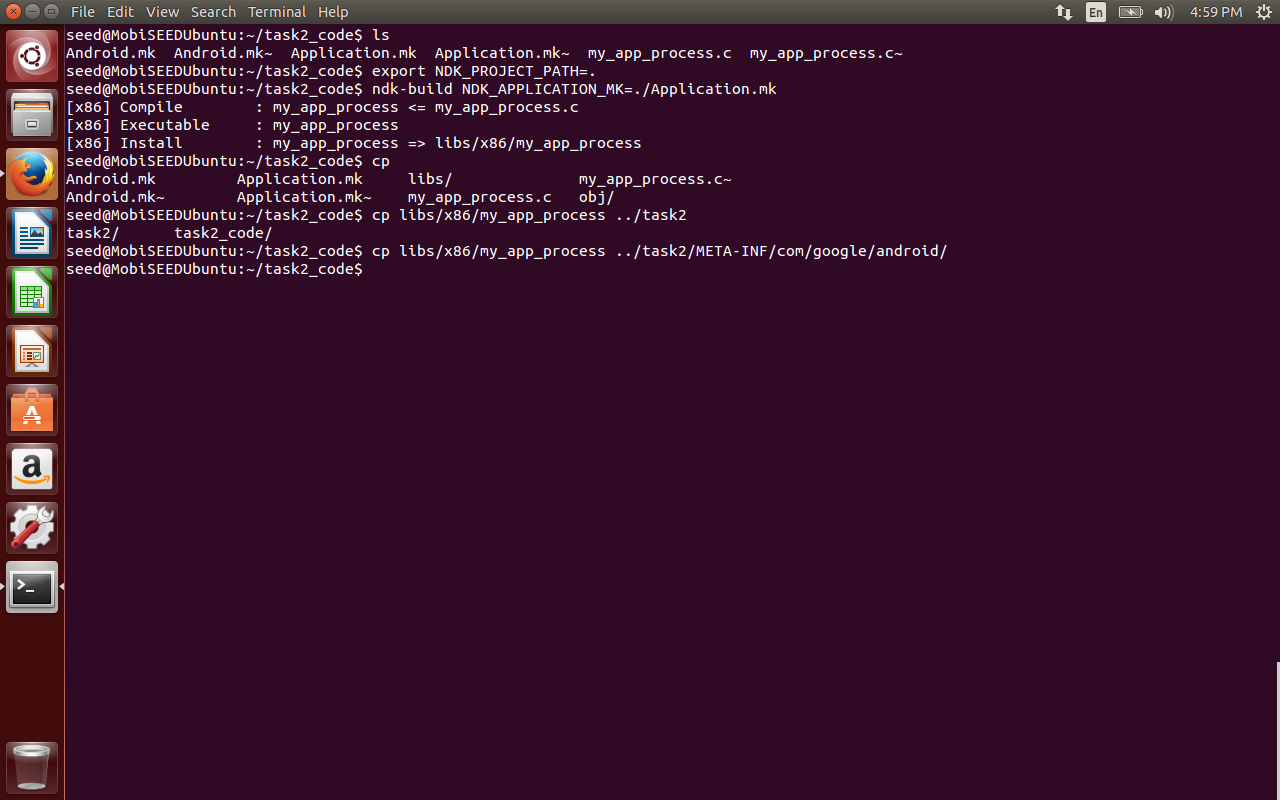






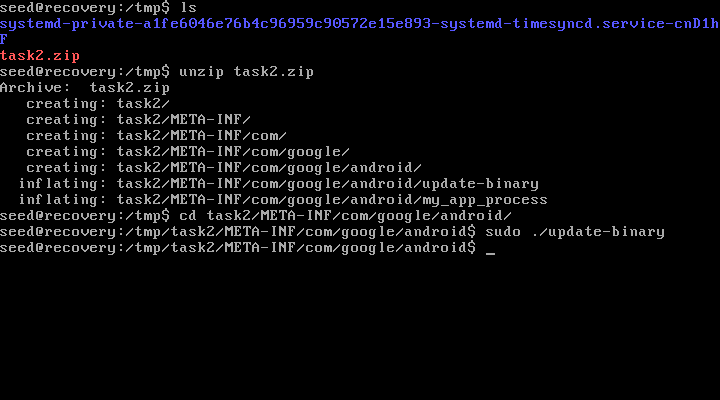
Compile the code using the below command.  
export NDK\_PROJECT\_PATH=.  
ndk-build NDK\_APPLICATION\_MK=./Application.mk

Copy my\_app\_process to task2/META-INF/com/google/android folder.



Now create the update-binary to rename the original app\_process file and run our wrapper app\_process program.

Copy the zip file to recovery OS and the update-binary script.



Observe the dummy2 file created inside the /system folder.



**Task 3: Implement SimpleSU for Getting Root Shell**

When building the OTA package, all the commands that need to be run with root privileges are already decided. Now the goal is to root the device so that the user can run any arbitrary command, they need a shell with root privilege to do that. This attack tries to launch a root shell.   
The approach is to start the root daemon, during the booting process and use this daemon to get the root shell. In this task, we will write such daemon and use it to get the root shell.

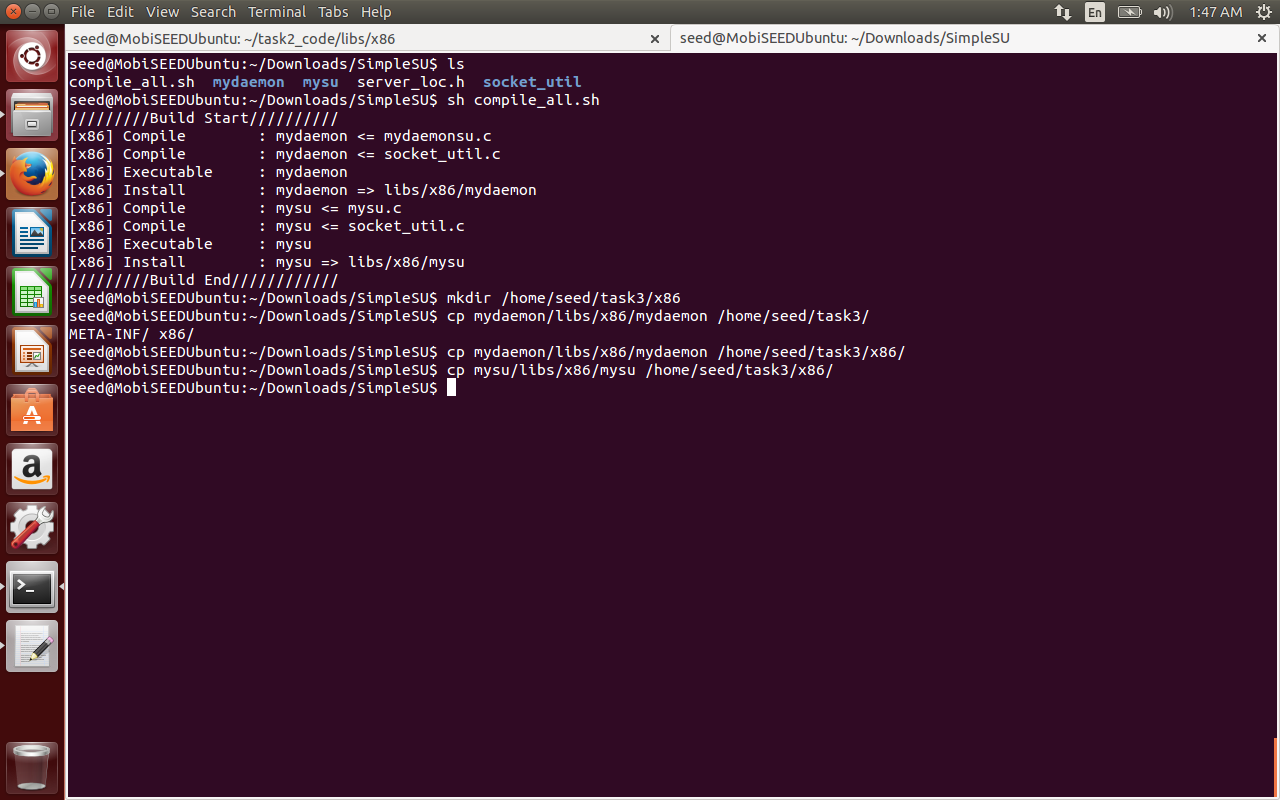
We can use the methods as in the previous tasks to launch the root shell, but the booting of android will be stopped, because shell is a interactive program (and it won’t quit until we give exit command). We can have setuid program where the users runs the shell (bash) program with root privileges. But from Android 4.3 all the setuid programs are removed from android to prevent such attacks.

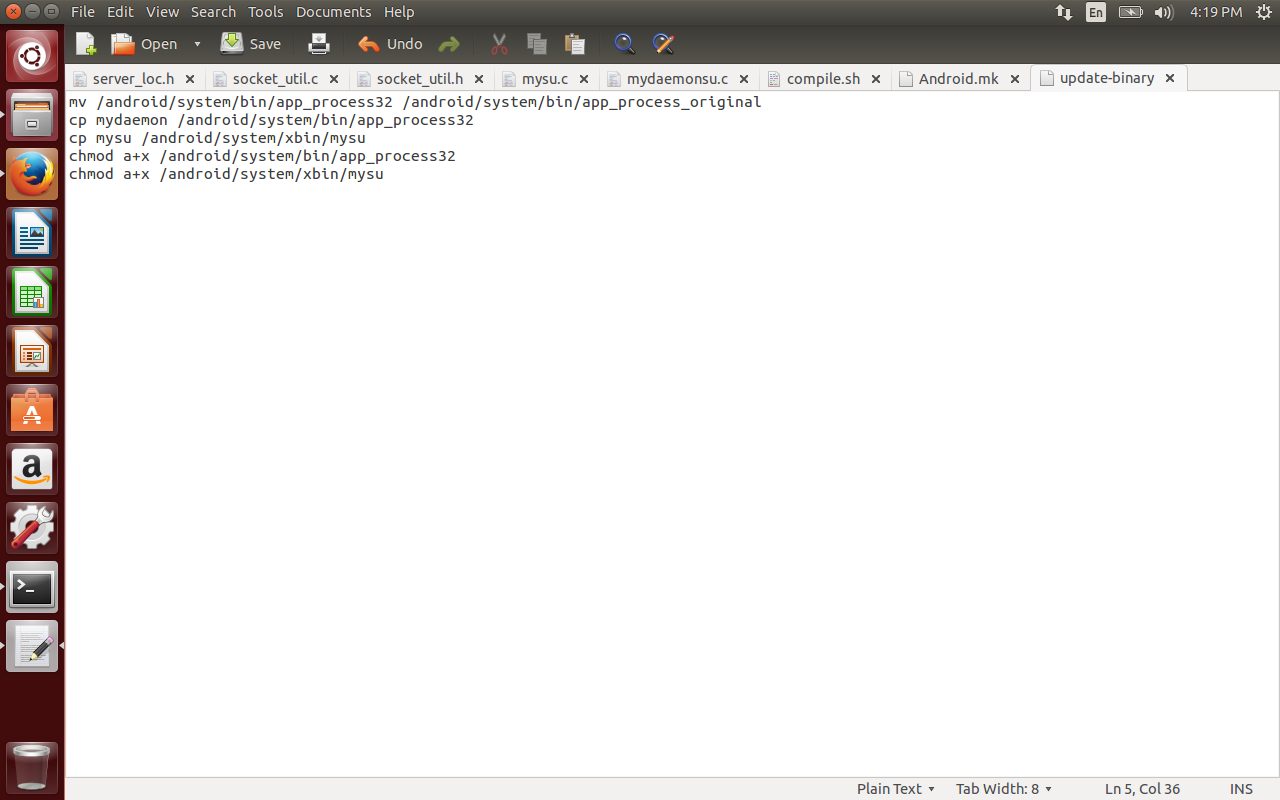
The final approach is to run a root daemon during the booting time. The user or client can use this root daemon to start the shell program as a child of it. The client will send its file descriptors to the child process i.e., shell program and then client interacts with the shell using these standard input , output file descriptors. The standard output of shell program is redirected to standard output of client program, standard input of shell program is redirected to standard input of client program.

The client connects to the server using the Unix Domain Socket.   
Upon receiving the request, the server forks a child process and runs a root shell. The child process inherits all the standard I/O FDs from the parent.

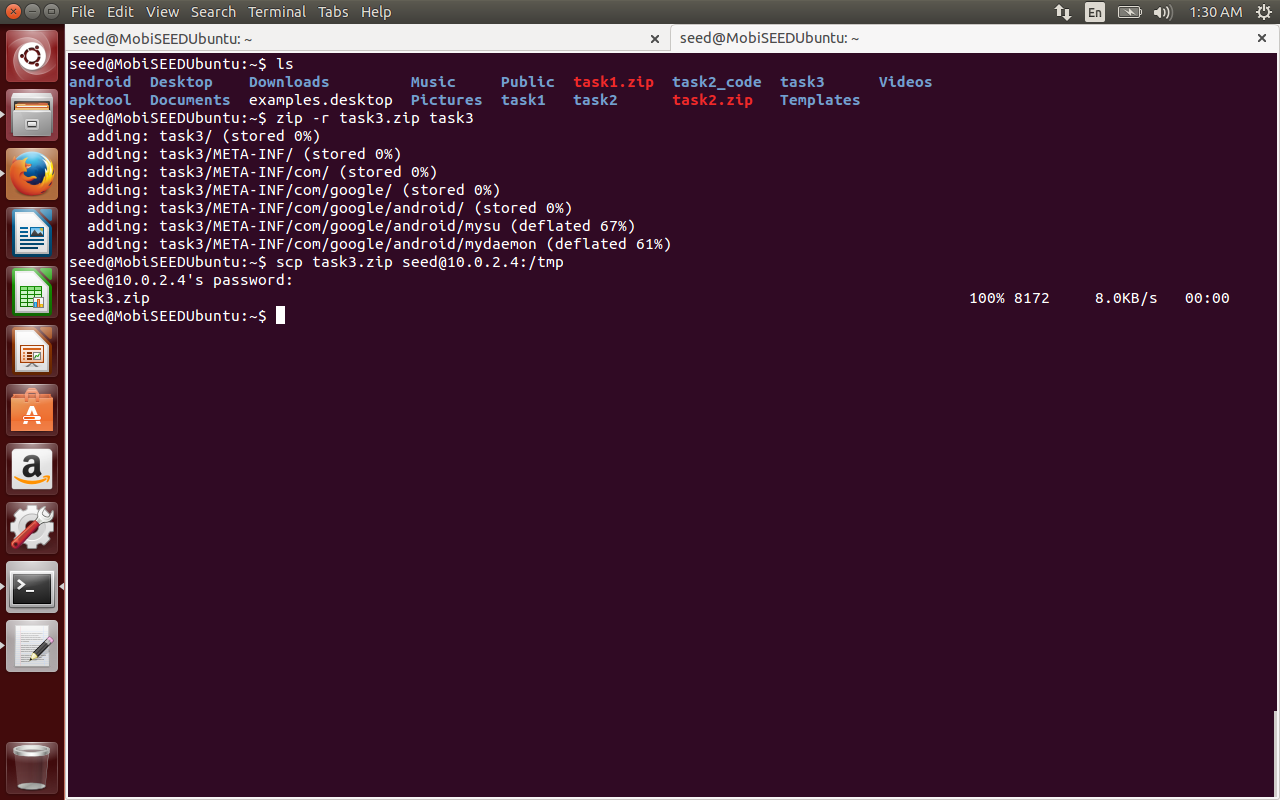
The client sends its FDs 0, 1, and 2 to the server’s child process using the Unix Domain Socket. These FDs will be saved in the table indices 4, 5, and 6, respectively.  
The child process redirects its FDs 0, 1, 2 to the FDs received from the client, resulting in FDs 4, 5, and 6 being used them as the standard input, output, and error devices. Since these three devices are the same as those in the client, essentially, the client process and the server’s child process now share the same I/O devices. Although the client process is still running with a normal user privilege, it has the full control of the server’s child process, which is running with the root privilege.

After compiling both mydaemon and mysu copy the executables to task3/x86 folder as below.

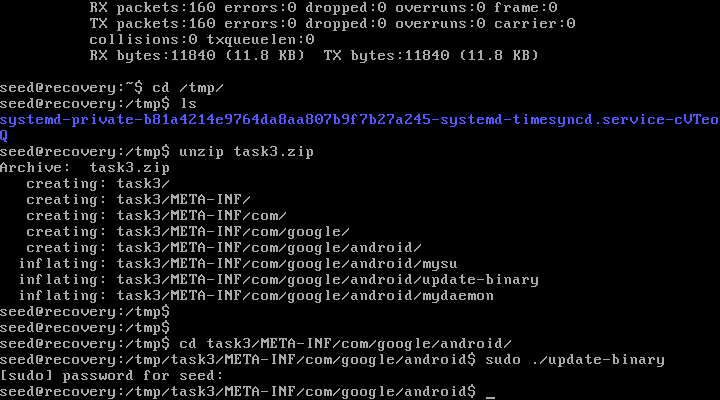


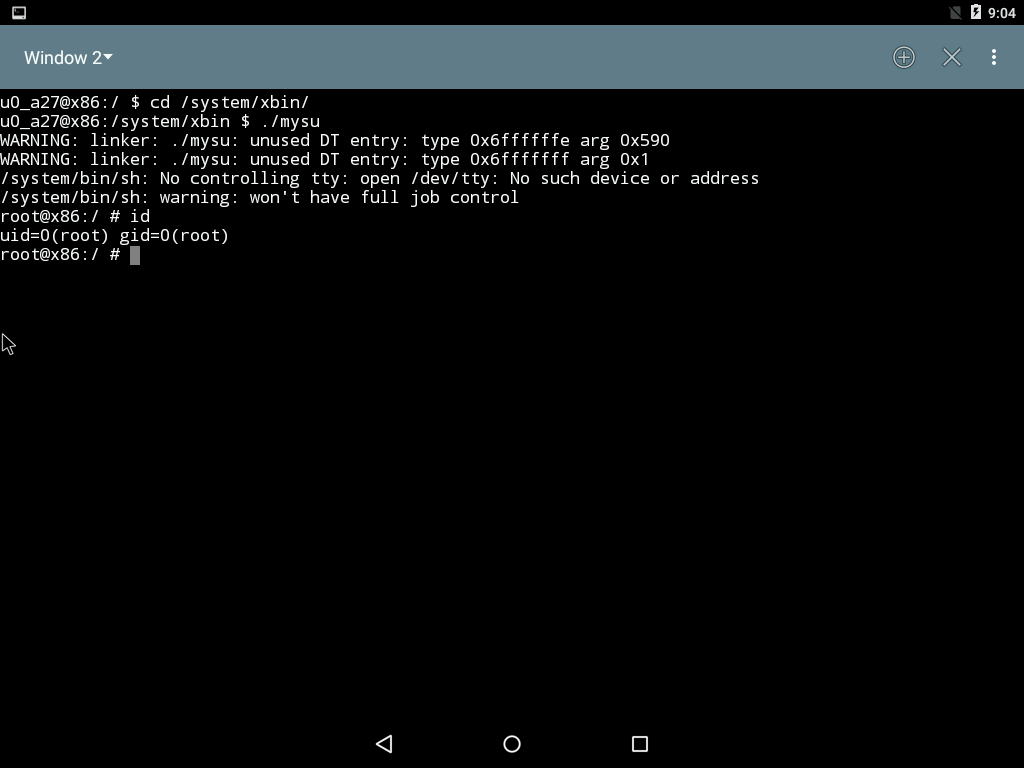


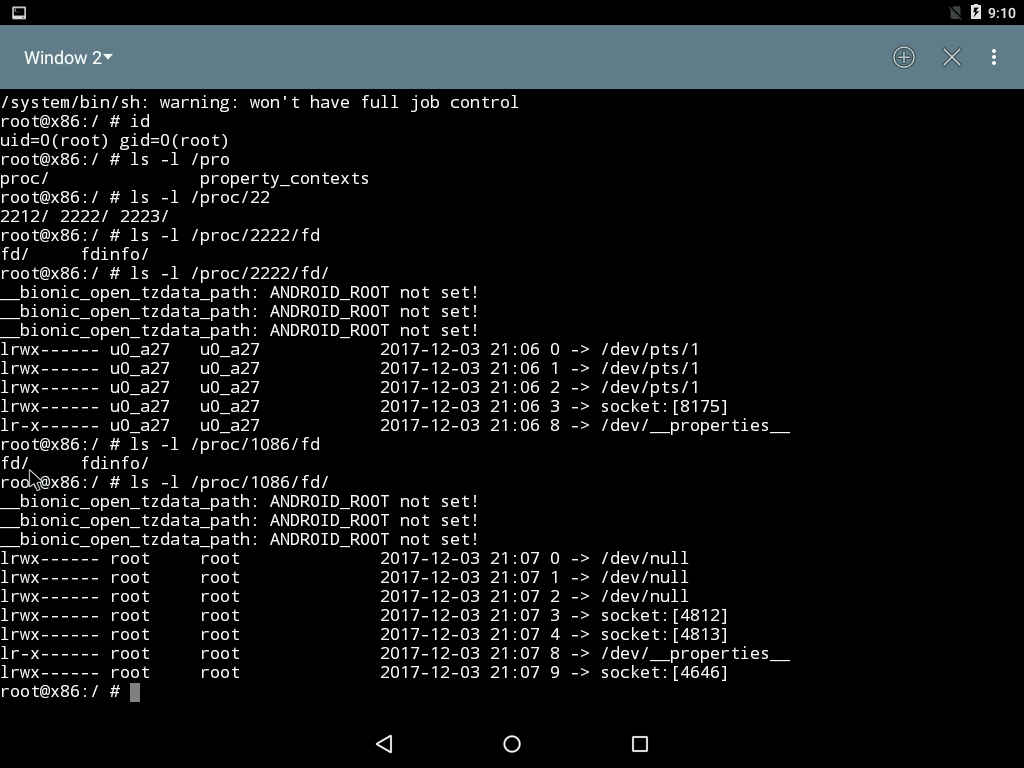
This is the update-script to send both the mydaemon and mysu executables to the android mounted file system.   
Send the zipped folder to recover us of android and run the script manually.



Now unzip task3.zip in the Ubuntu Recovery OS and run the update-script.



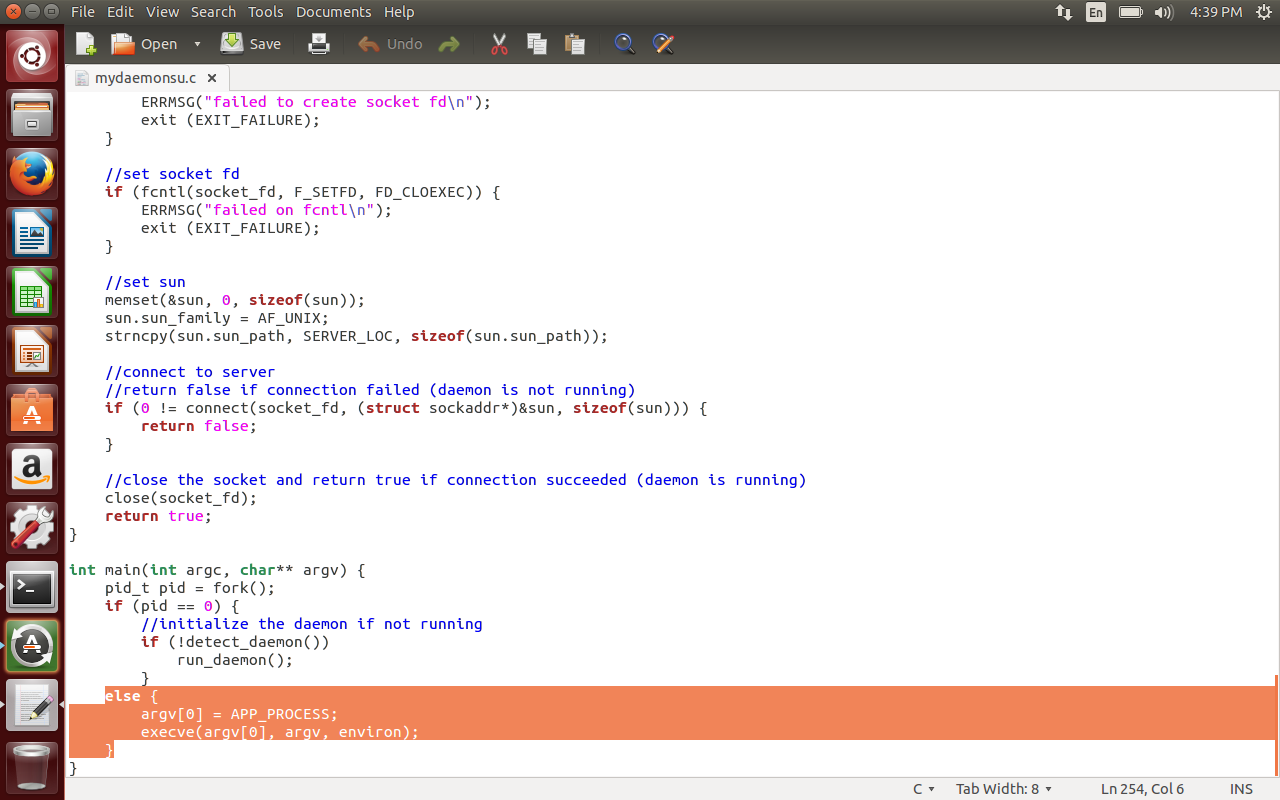




Questions:

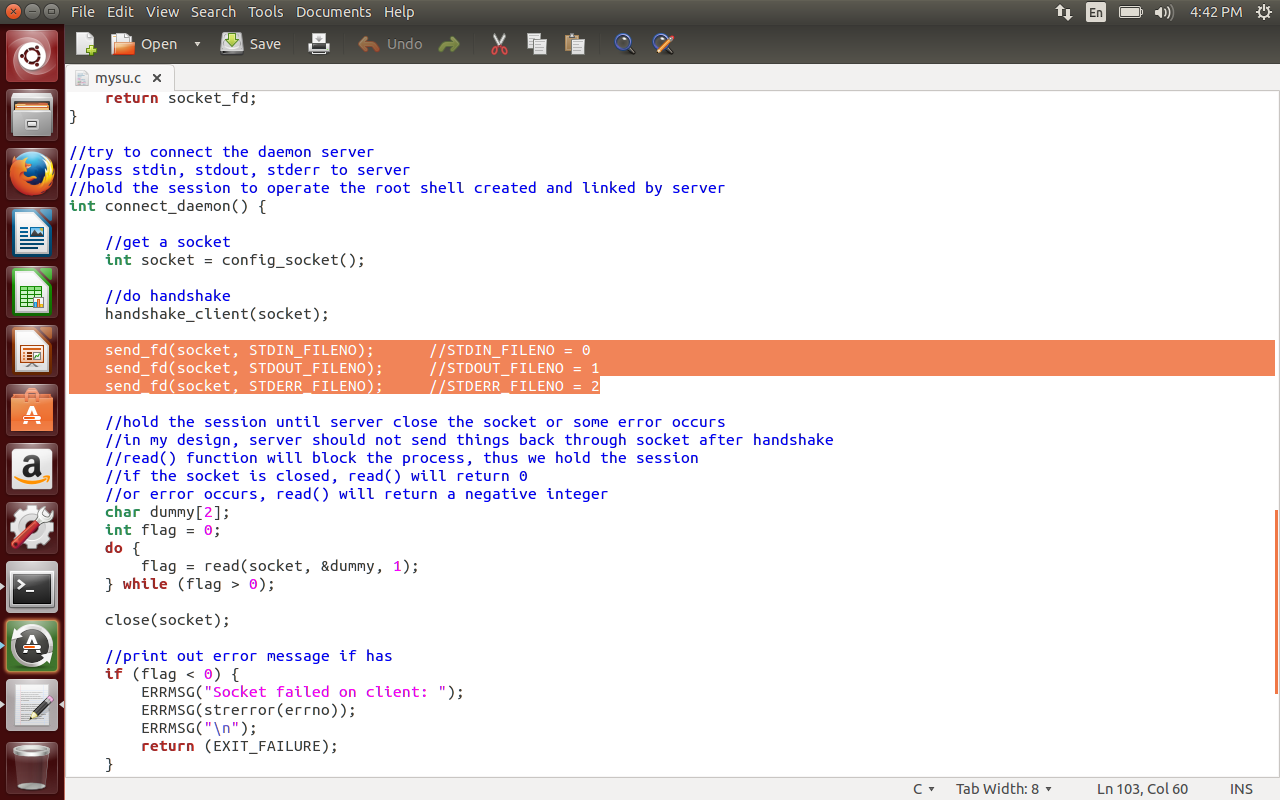
Server launches the original app process binary

File: mydaemon.c function name: main(), code: highlighted



Client sends its FDs

File: mysu.c function: connect\_daemon() code: highlighted

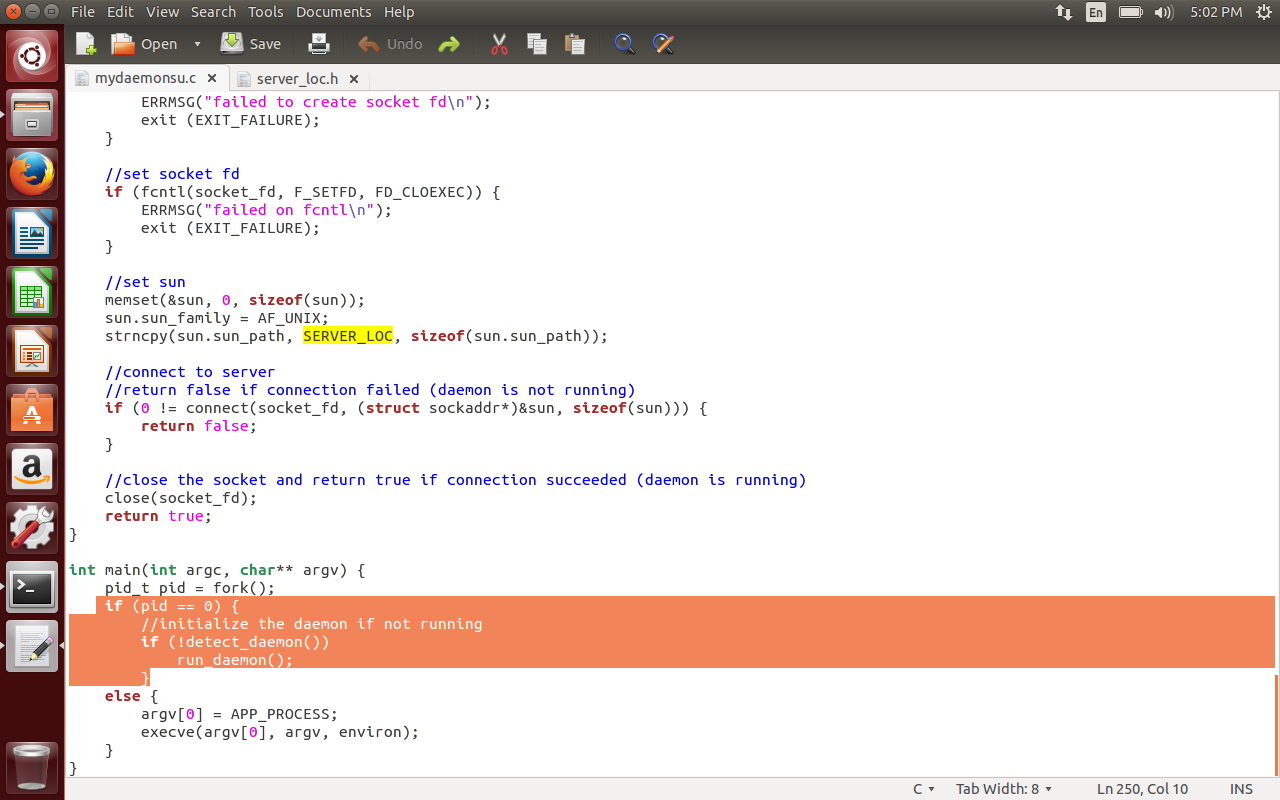


Server forks to a child process

File: mydaemonsu.c Function Name: main(int argc, char\* argv[]), Code: highlighted

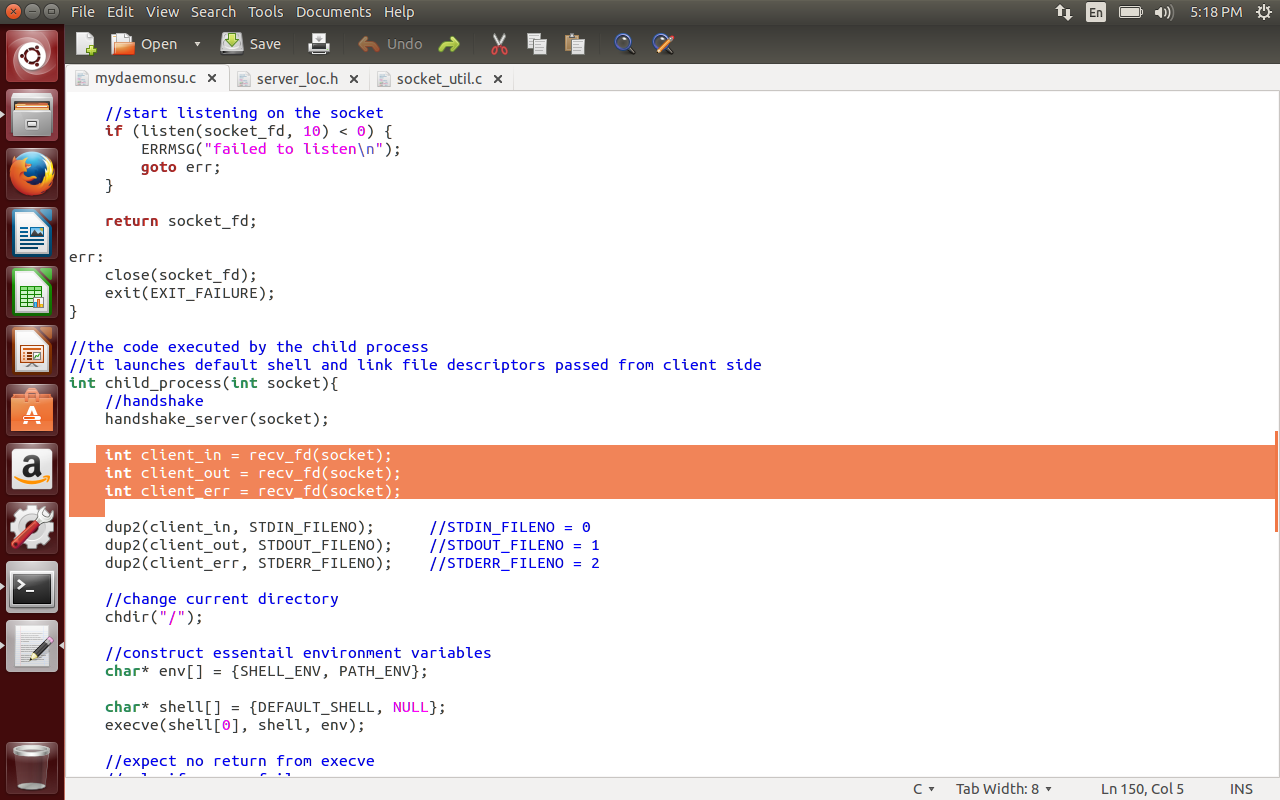
In the run\_daemon() we create a socket and wait for new requests, handle them in the

Child\_process() and then close the connection.



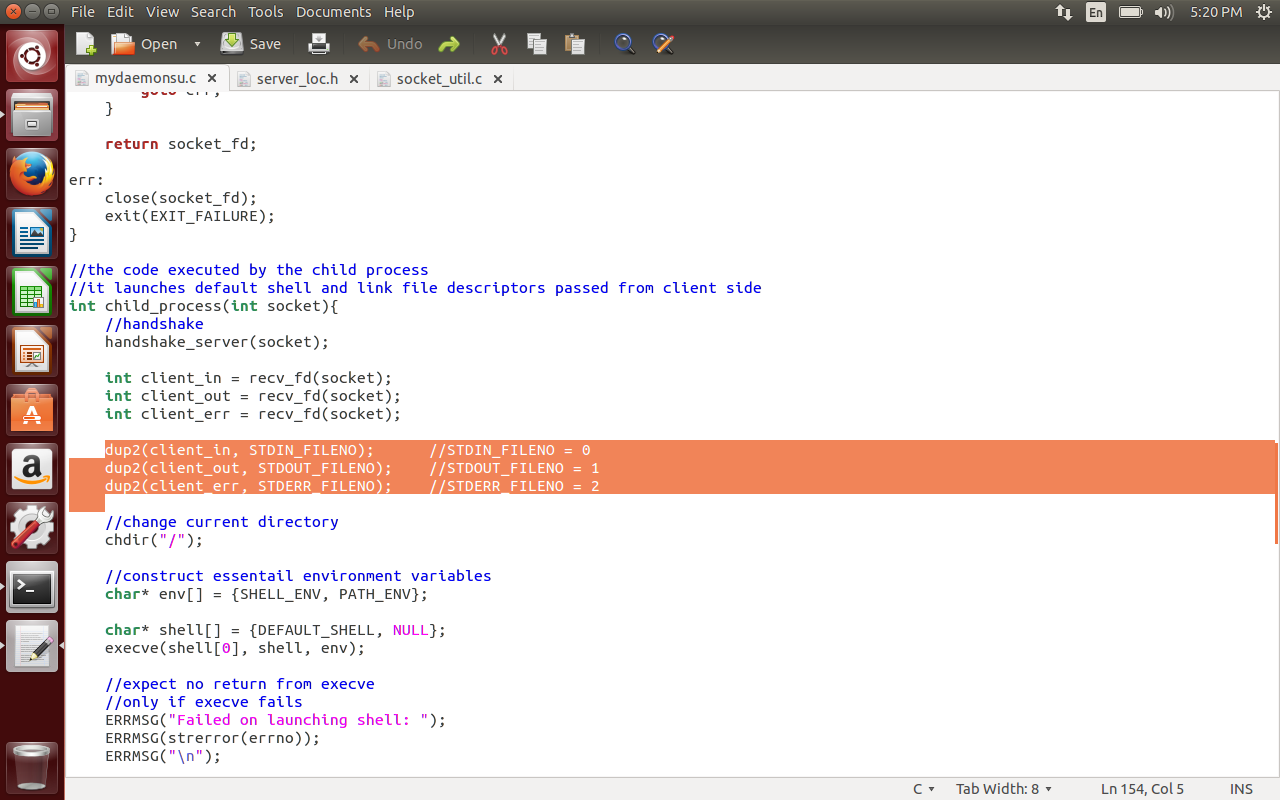
Child process receives client’s FDs

File: mydaemon.c function: child\_process(), code: highlighted



Child process redirects its standard I/O FDs

FileName: mydaemon.c function: child\_process(), code: highlighted



Child process launches a root shell:

File: mydaemon.c function: child\_process(), code: highlighted

Here DEFAULT\_SHELL is /system/bin/sh which is run with root previliges to get the root shell.

