Android Rooting Lab Report

Mudit Vats
mpvats@syr.edu
11/29/2018

Table of Contents

Overview

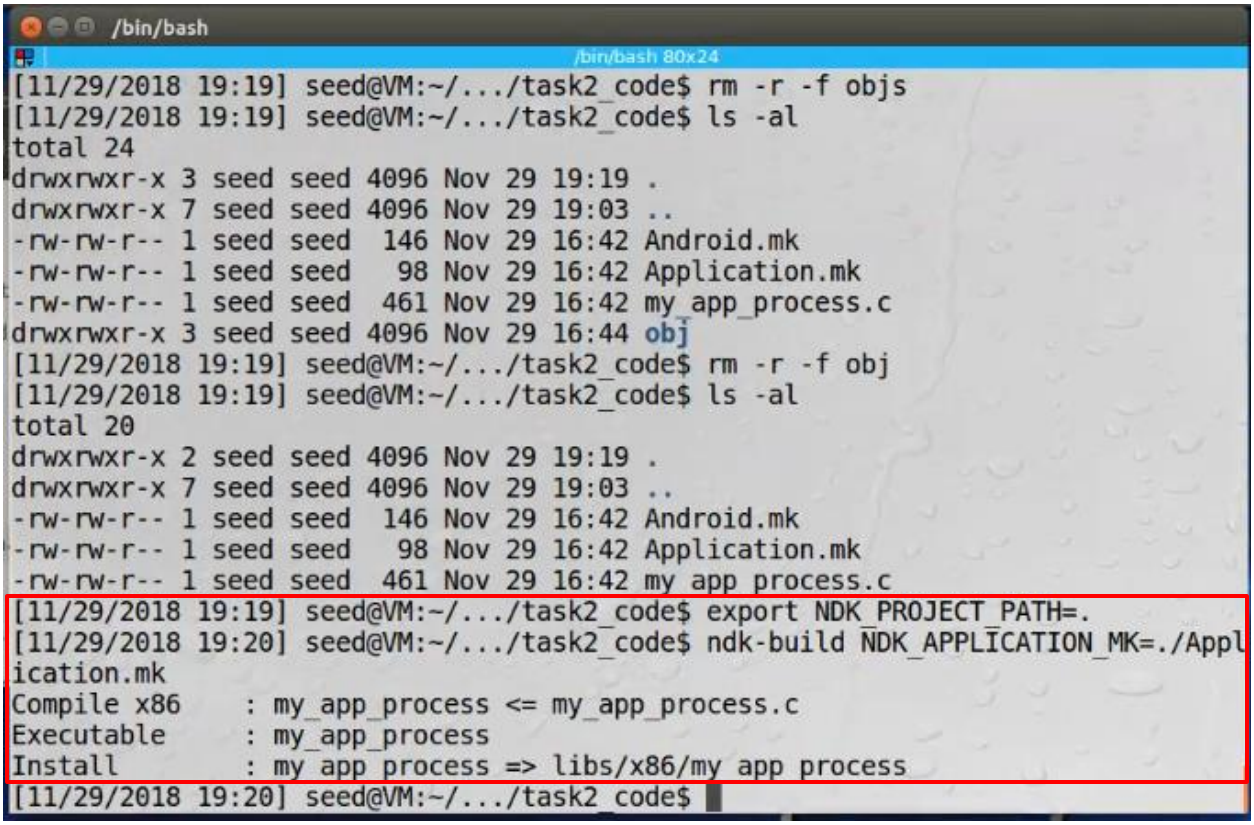
This lab report presents observations and explanations for the tasks described in the Android Rooting Attack Lab.

Please note, per instructions given in our lecture, we are documenting and demonstrating Task 2 and Task 3 only. Task 1 was completed in class.

Task 2: Inject code via app_process

Step1: Compile the code

In the figure below, we see the NDK environment setup and ndk-build, which builds the Android application. We reuse Android.mk, Application.mk and my_app_process.c given to us for this lab.

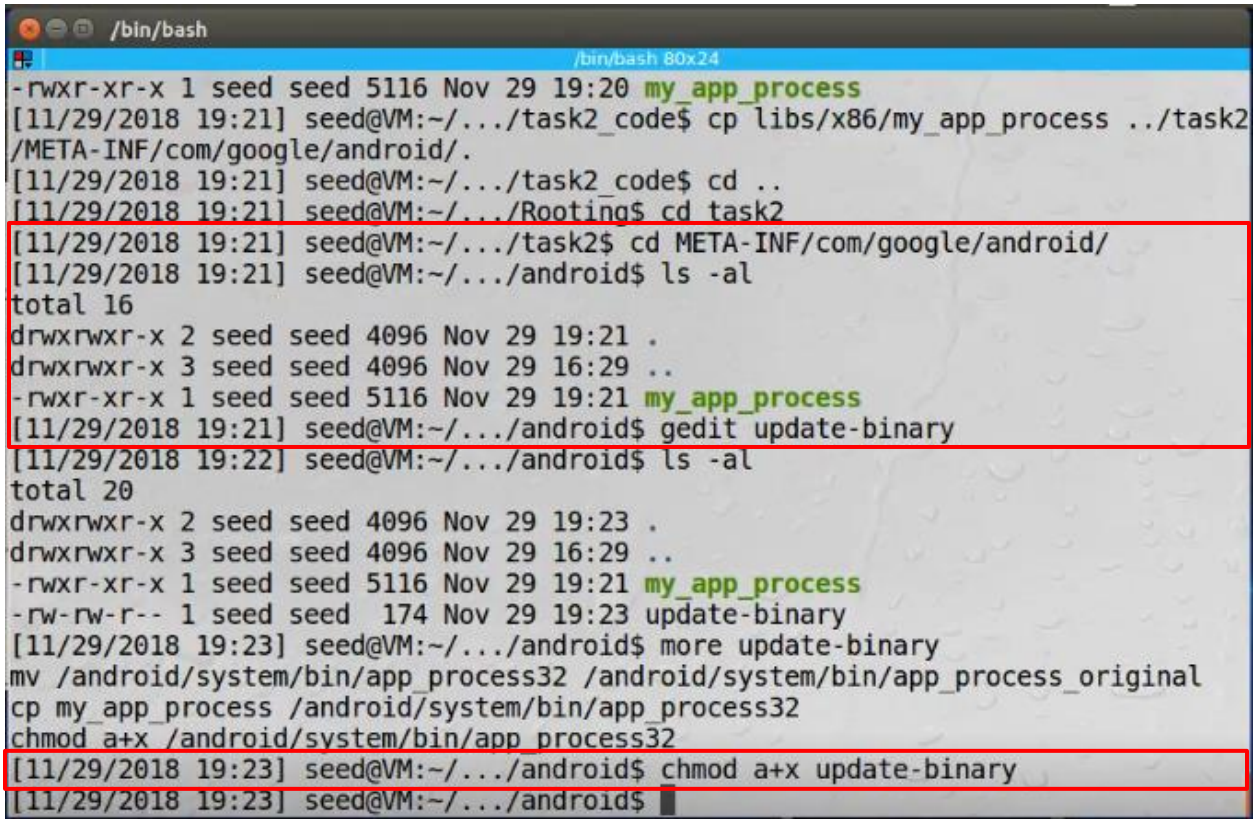


Step 2: Write the update script and build OTA package

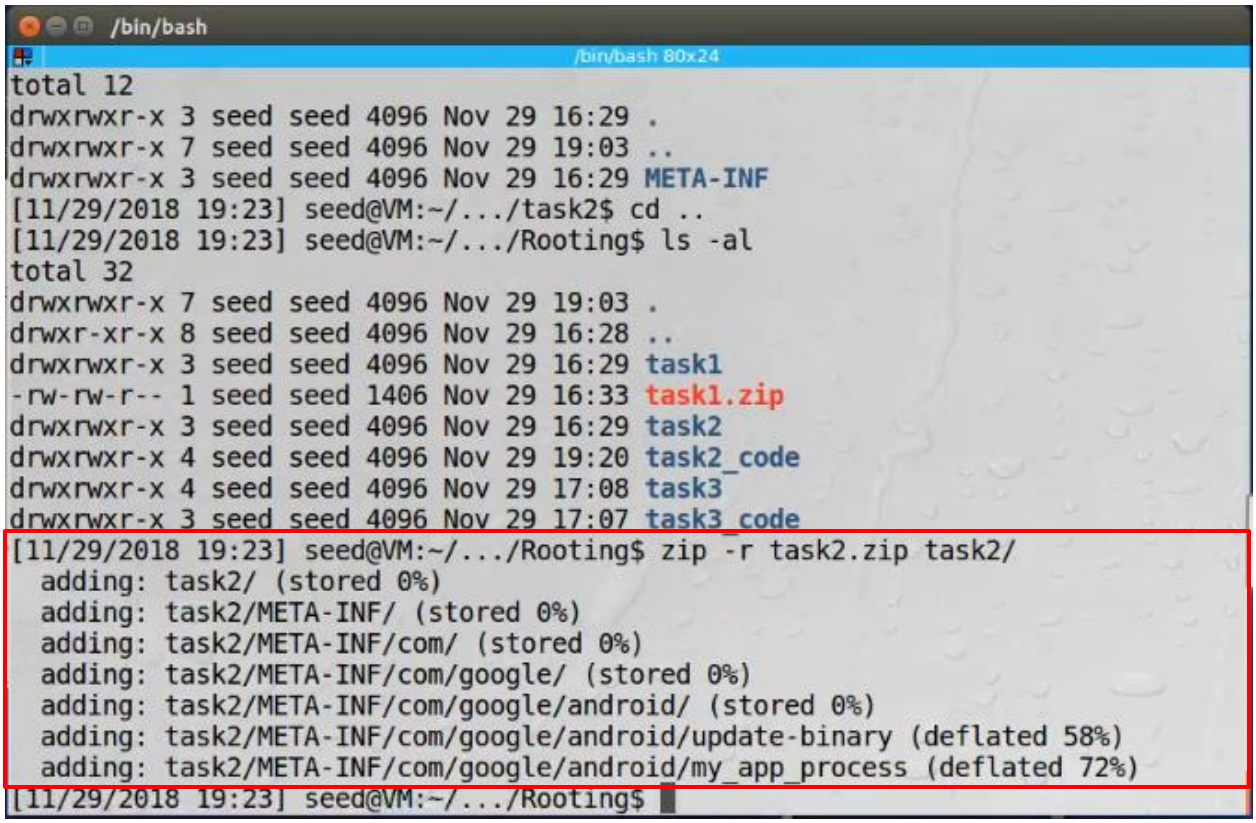
In the figure below, we see the contents of update-binary file which will be executed to issue the OTA update.

In the figure below, my_app_process and update-binary are copied to the /task2/META-INF/com/google/android directory.

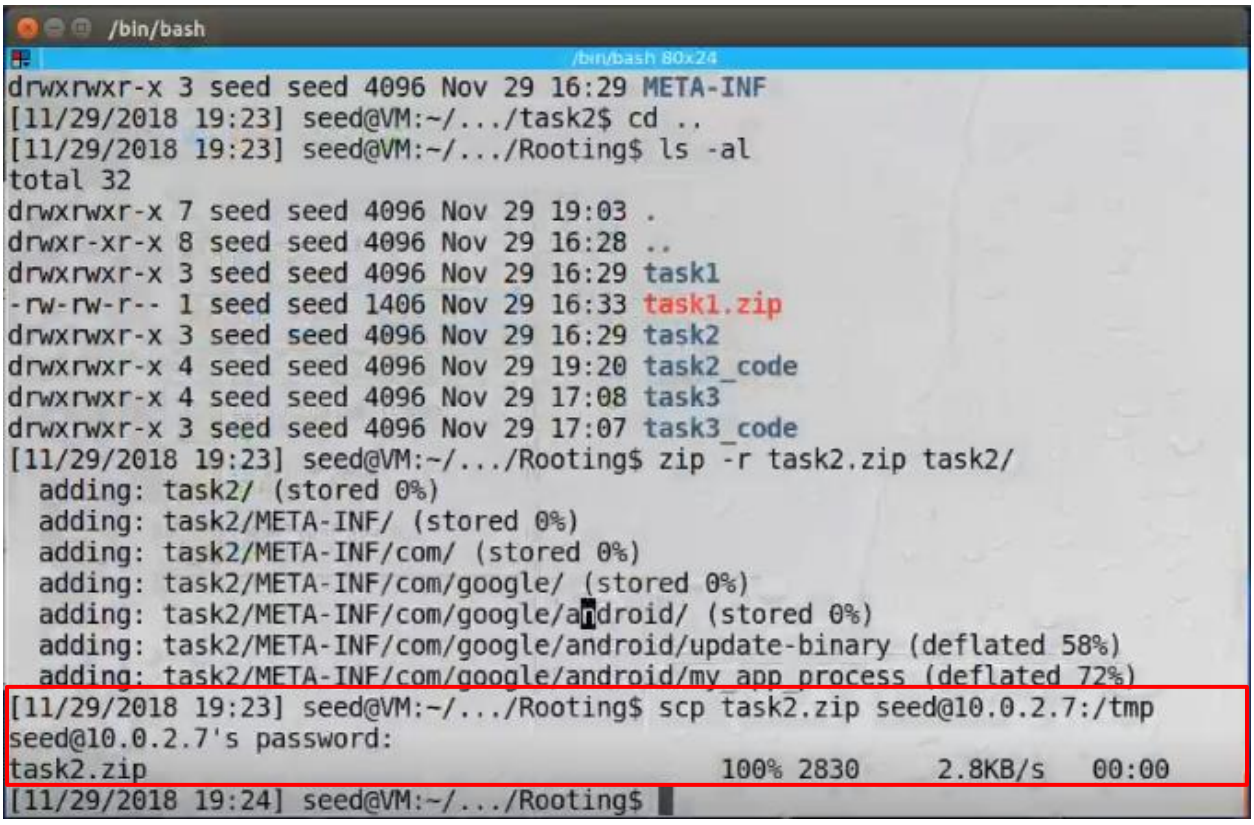
We also set the permissions of update-binary to be executable for all.



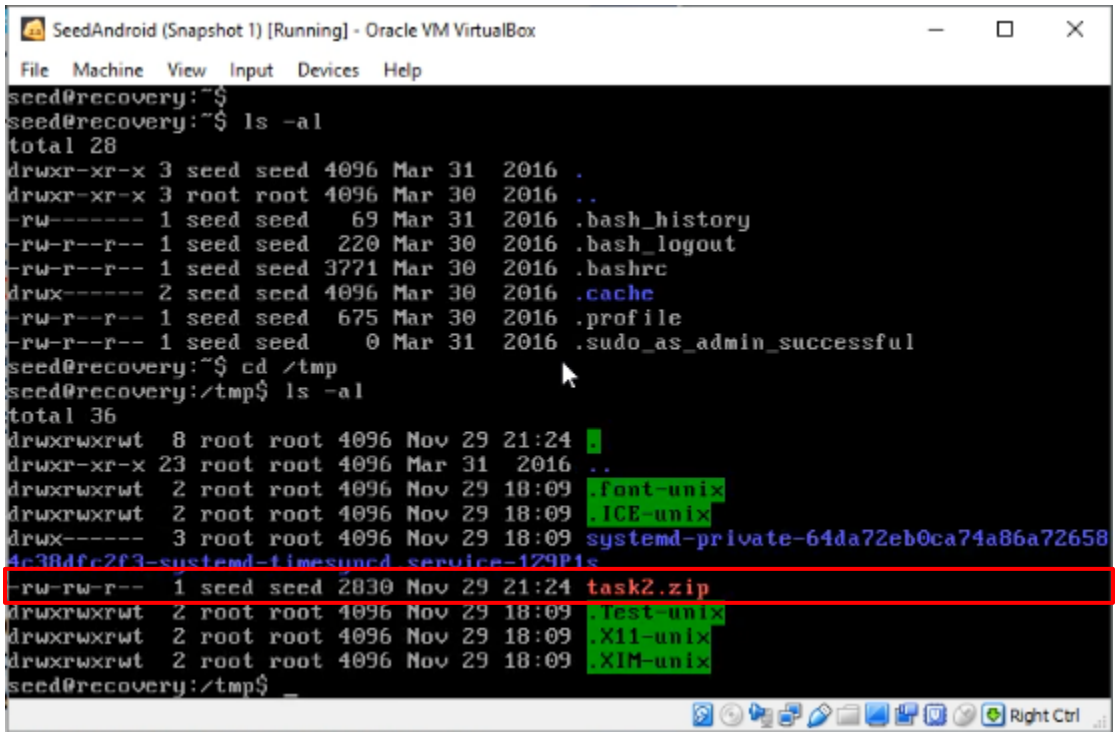
In the figure below, we see the OTA package being built which is simply a zip file.



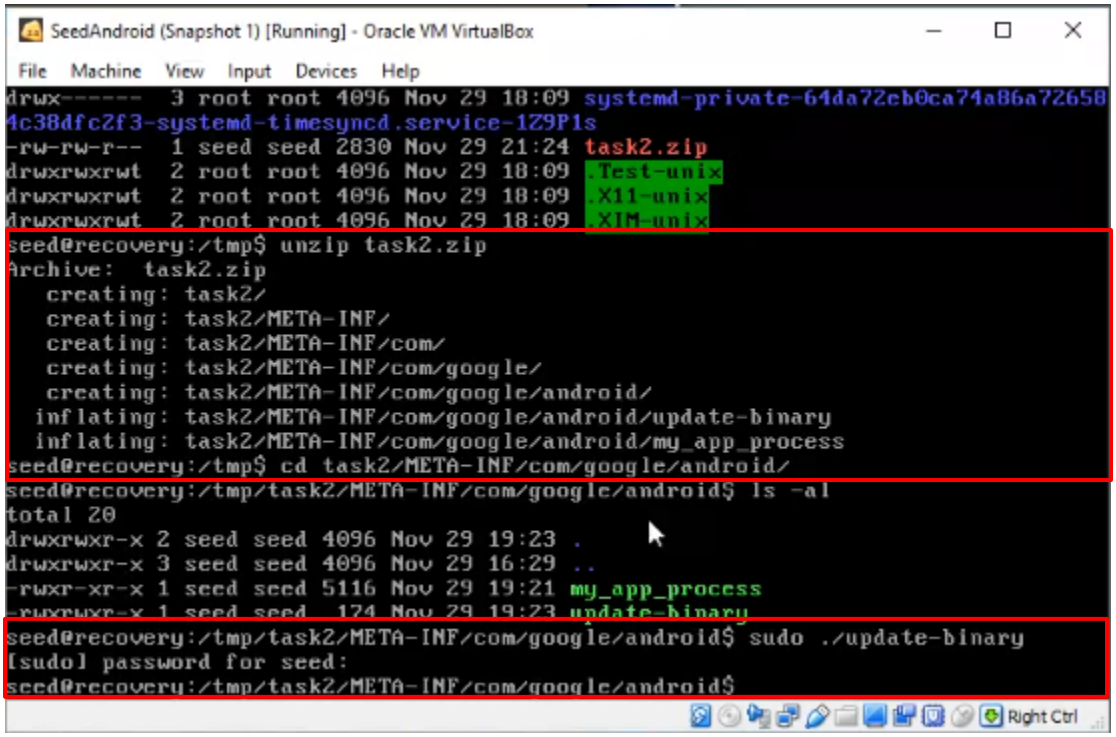
In the figure below, we see the scp to the Recovery OS on the Android VM. The recovery OS's IP address is 10.0.2.7. The task2.zip file gets placed into the /tmp directory.



In the figure below, we see the directory listing of /tmp which shows the successful scp'ing of the task2.zip file.

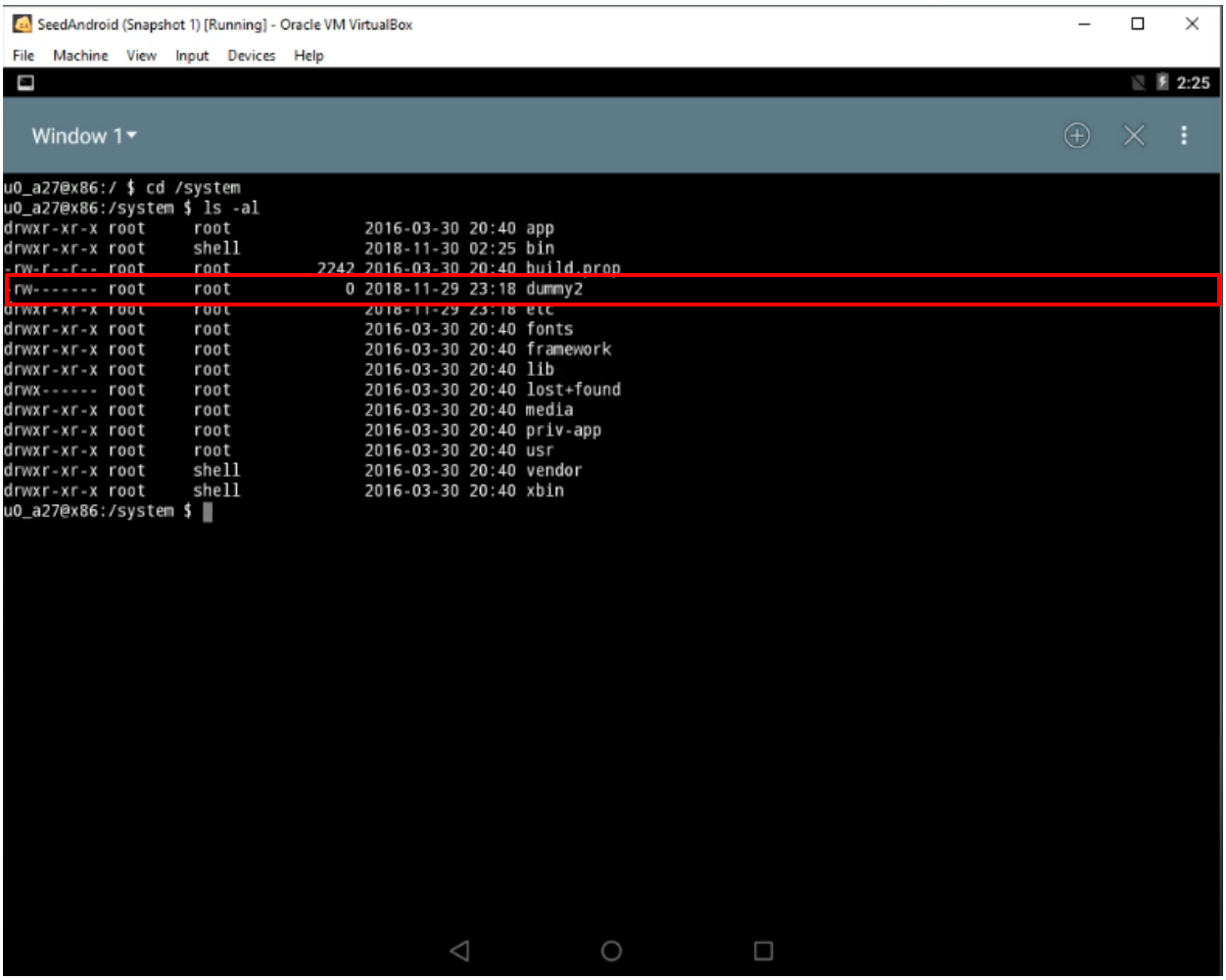


In the figure below, we see the decompression of task2.zip and execution of the update-binary. The ./update-binary executes without issue.



We now reboot into the Android OS. If our exercise is successful, we will see the dummy2 file in the /tmp directory.

In the figure below, we see that that the dummy2 file is present in the /system directory.



Observations / Explanation

In this task, we build an application (my_app_process) which runs during the Android boot sequence. This execution occurs as part of the zygote process; i.e. it executes app_process. Our update-binary, renames the current app_process32 to app_process32_orginal and copies our my_app_process to app_process32. Since zygote calls "app_process" which is a symbolic link to app_process32, it will call our rename my_app_process instead.

Our my_app_process does two things:

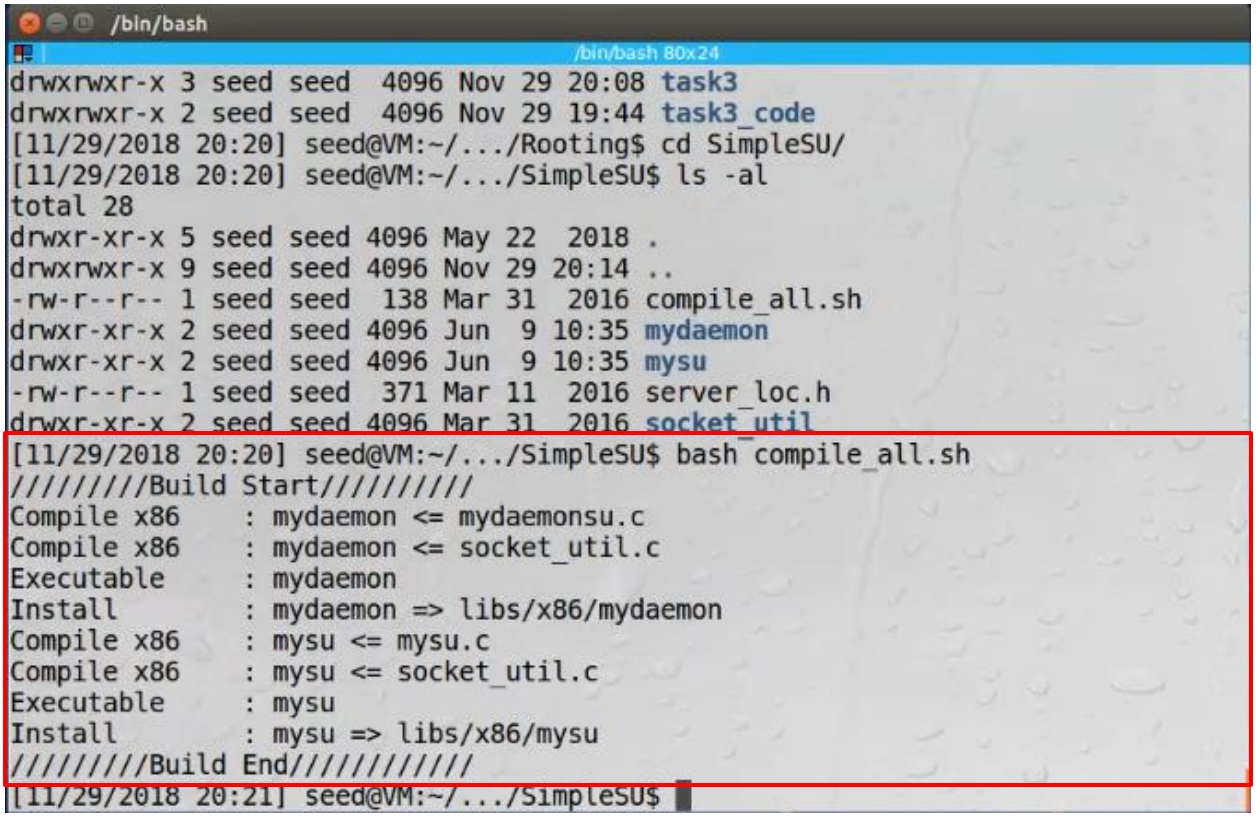
1. It creates a new file /system/dumm2.
2. It launch the /system/bin/app_process_original.

By doing this, we demonstrate the ability to do something "interesting" before the execution of the original app_process. While our my_app_process only creates a /system/dummy2 file, it shows the steps which can used by a hacker to do something more malicious. The main point is that we were able to create our own app_process32 to stealthily write a file as root and still call the original app_process32 thus successfully injecting code into the boot sequence without the end-user realizing it.

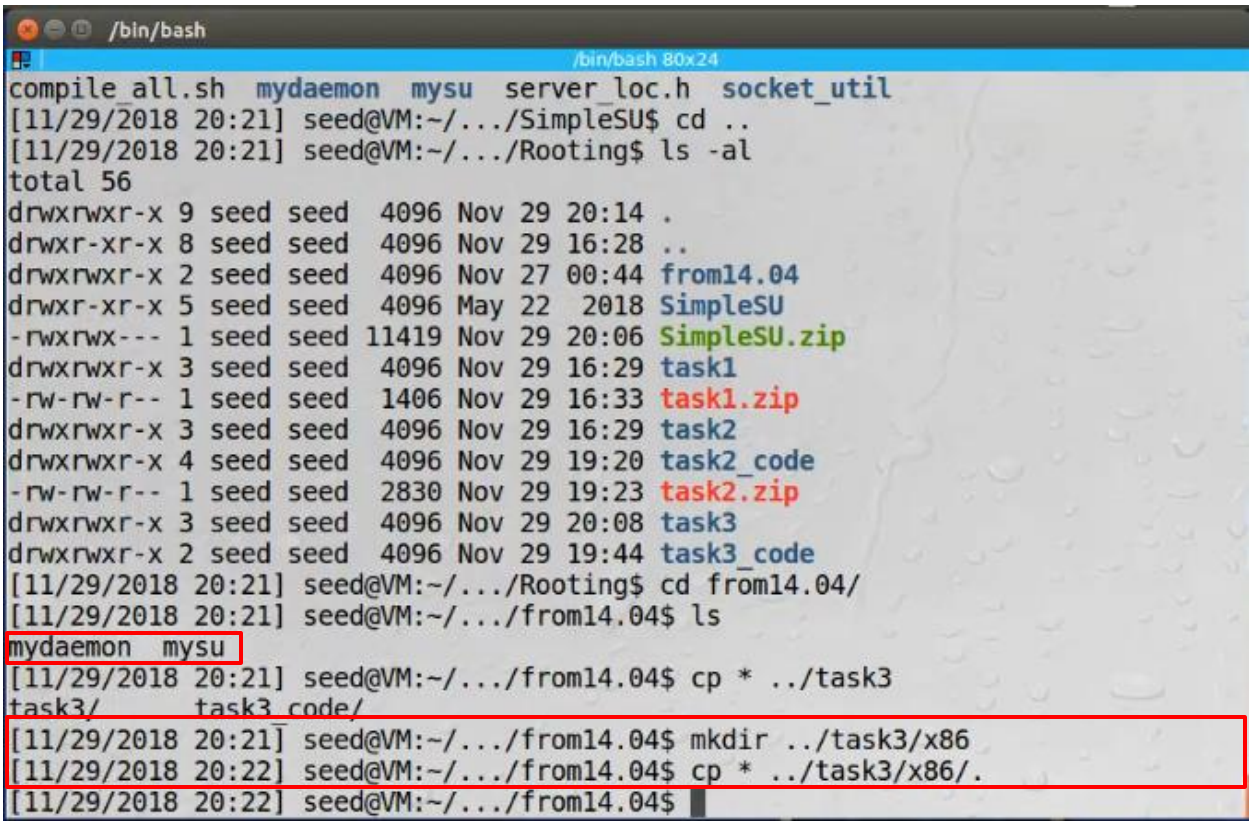
Task 3: Implement SimpleSU for Getting Root Shell

Step1: Compile the code

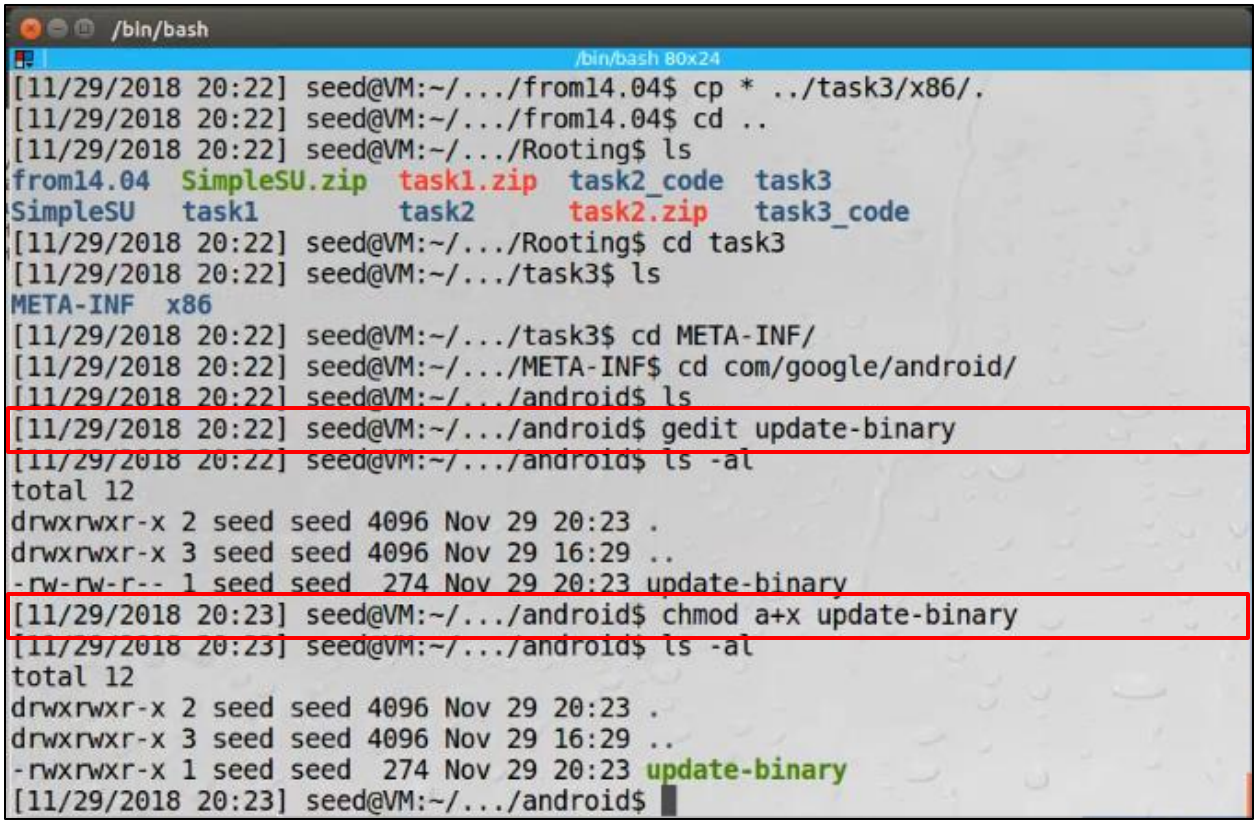
In the figure below, we compile the code using the compile_all.sh script. This sets the appropriate NDK environment variables and executes the NDK builds for the various sub-projects. We reuse the SimpleSU.zip code given to us for this task.



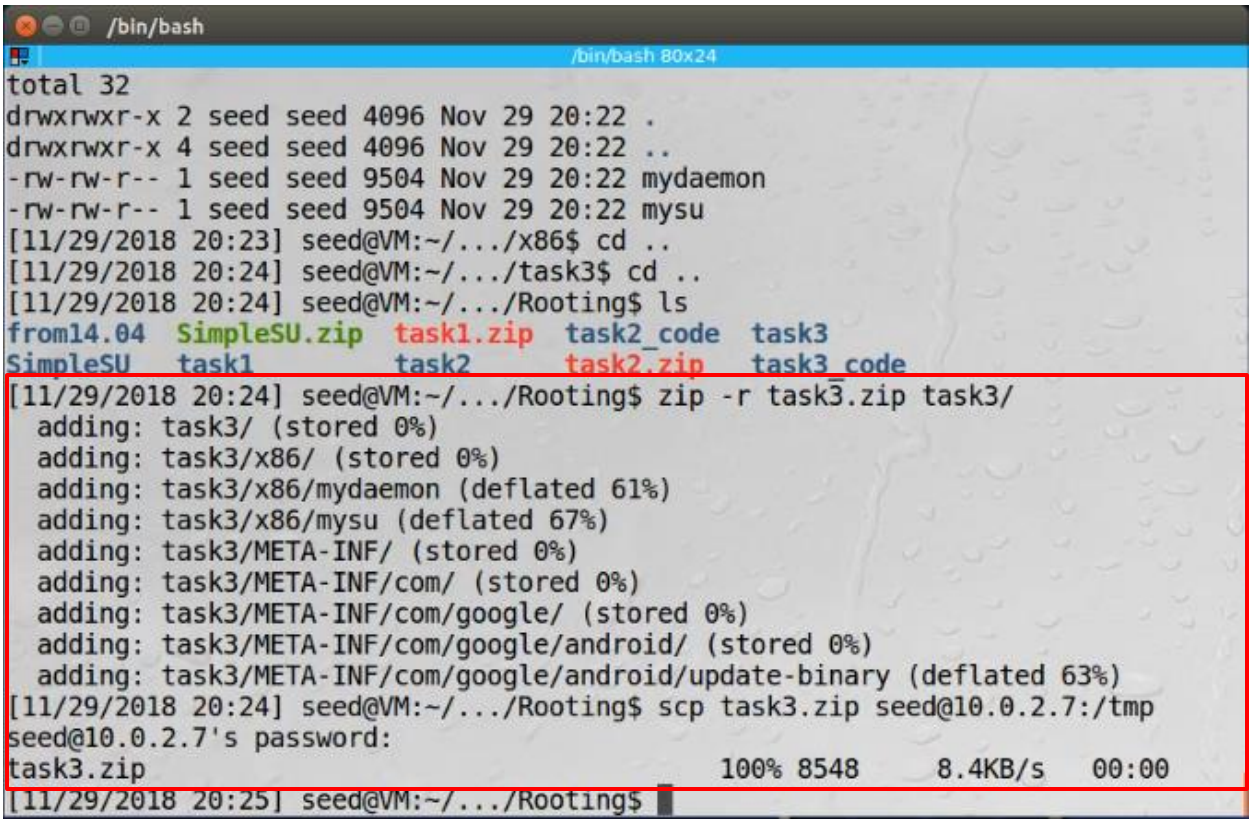
In the figure below, we see that we copy the mydemon and mysu pre-compiled binaries to the OTA's x86 directory. We are using these pre-compiled versions instead of the ones we built in our previous step since our development Ubuntu environment is 16.04 and it cannot produce Android binaries for older Android targets. Still, the step above shows the process on how we would accomplish building mydemon and mysu.



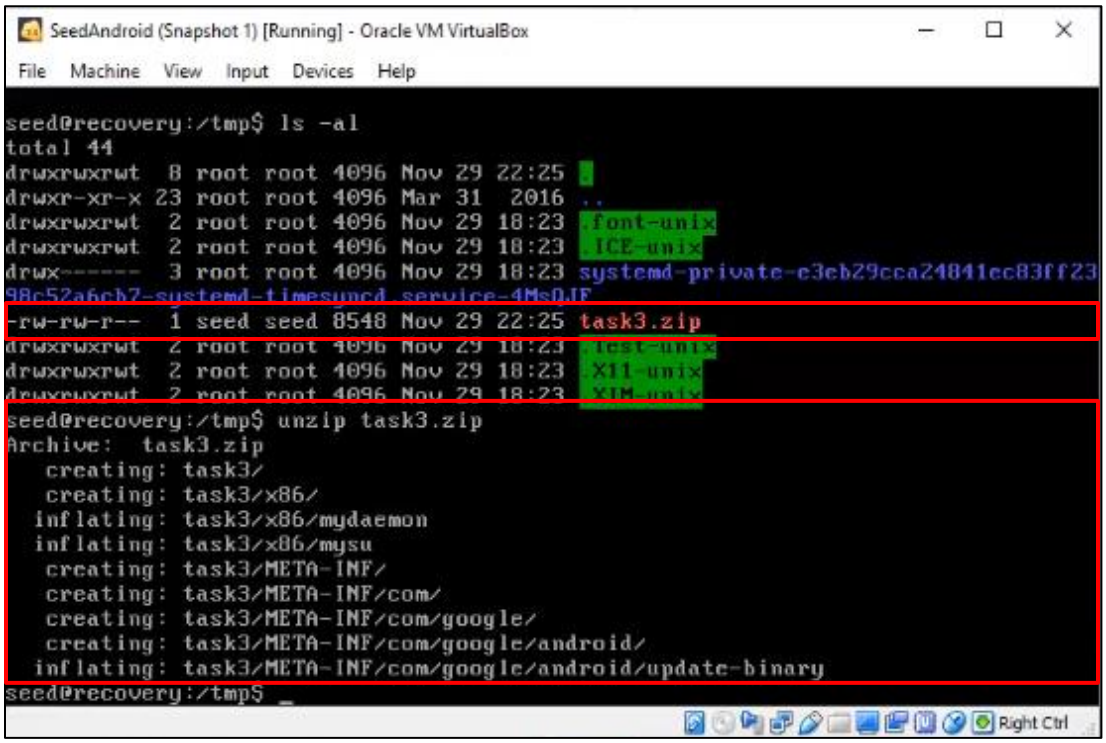
The figures below shows the update-binary script which we will use to execute the OTA update. The first figure shows the edit of the file in the task3/META-INF/com/google/android directory and adding the executable permission to the file. In the figure below that, we show the contents of the update-binary file.



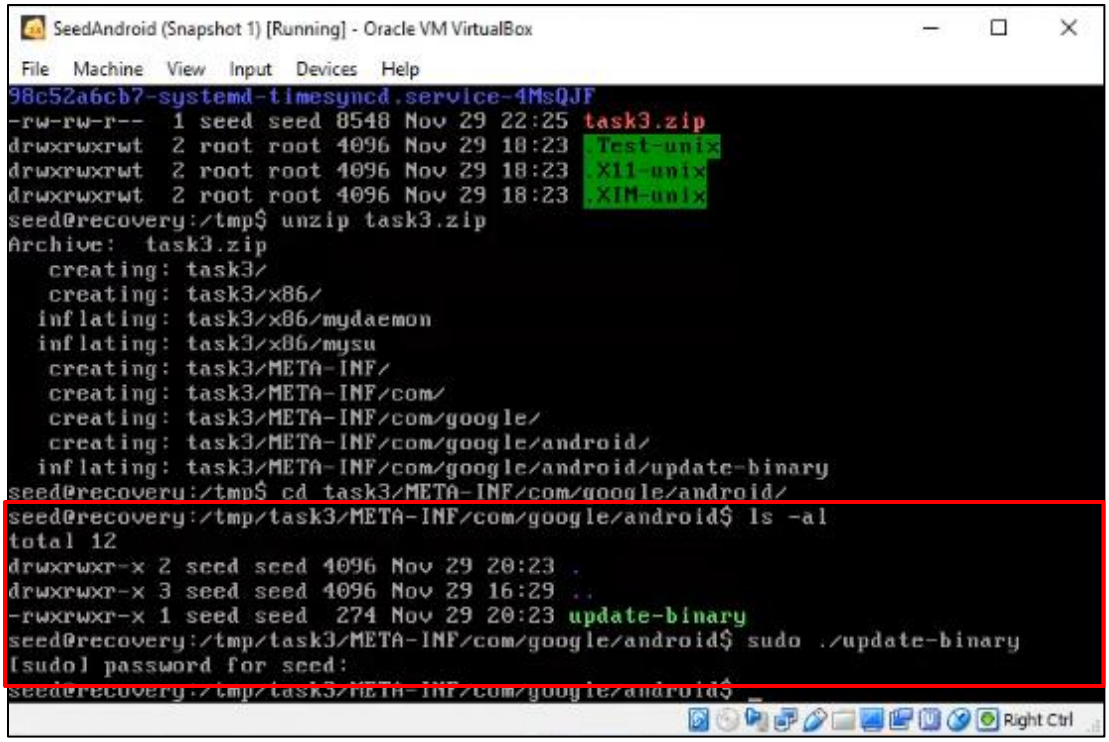
In the figure below, we show the creation of the OTA package which is a zip file of the task3 files. This creates task3.zip. We also wee the scp to the Recover OS whose IP address is 10.0.2.7.



In the figure below, we see the task3.zip now exists in the /tmp directory. Further, we decompress it.

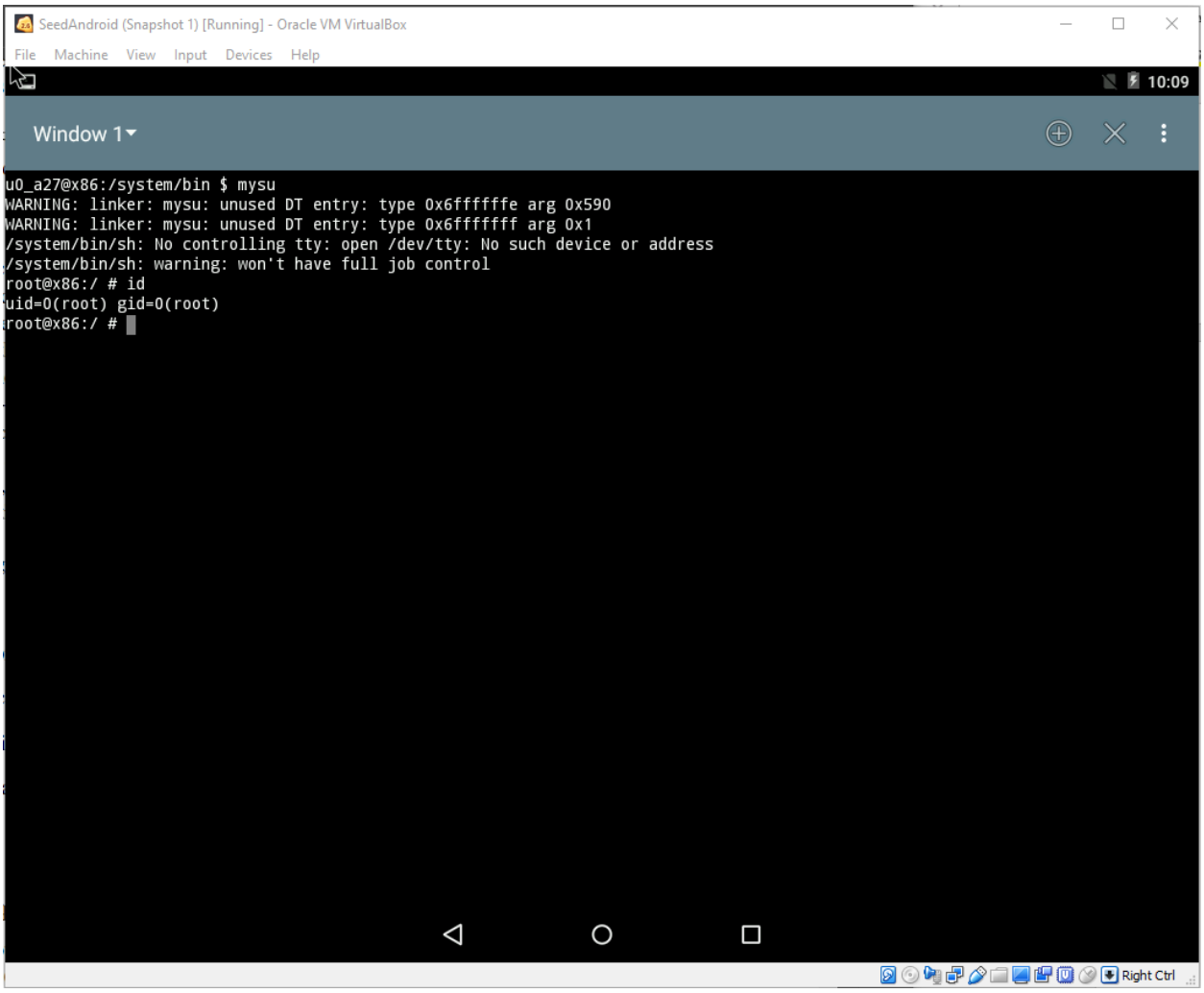


In the figure below, we see the execution of the update-binary script.



We now reboot to the Android OS. In the figure below, we show the execution of the Terminal Emulator and we also execute the "mysu" binary which was copied to the Android OS per the update-binary.

The successful execution of mysu confirms that the OTA update succeeded and since we were able to type "id" and see that our UID is 0, which is root. Also, we know we now have an interactive root shell; i.e. successfully rooting the Android OS.



Observations / Explanation

In this task, we actually built on Task 2 but added a root daemon which is used to create a terminal which a client can connect to. We use the daemon (mydaemon) and client (mysu) that were provided to us. The mydaemon is take the place of app_process32 and responds to client (mysu) connections to then create an interactive terminal. The client passes in its file descriptor to use by the daemon which will redirect STDIN, STDOUT and STDERR to the client's file descriptors. Once the mapping is complete, an interactive root shell is created. The end result, a successful attack which accomplishes the ultimate goal of create a root shell.

Subtask: Code Identification

After completing the task, students need to look at the source code, and indicate where the following actions occur. Filename, function name, and line number need to be provided in the answer.

Action	Filename	Function Name	Line Number

Server launches the original app process binary	mydaemonsu.c	main	255
			argv[0] = APP_PROCESS;
			execve(argv[0], argv, environ);
			where APP_PROCESS is "/system/bin/app_process_original"

Client sends its FDs	mysu.c	connect_daemon	112
			send_fd(socket, STDIN_FILENO);
			send_fd(socket, STDOUT_FILENO);
			send_fd(socket, STDERR_FILENO);

Server forks to a child process	mydaemonsu.c	run_daemon	189 if (0 == fork()) { … } Child process starts handling client connection.

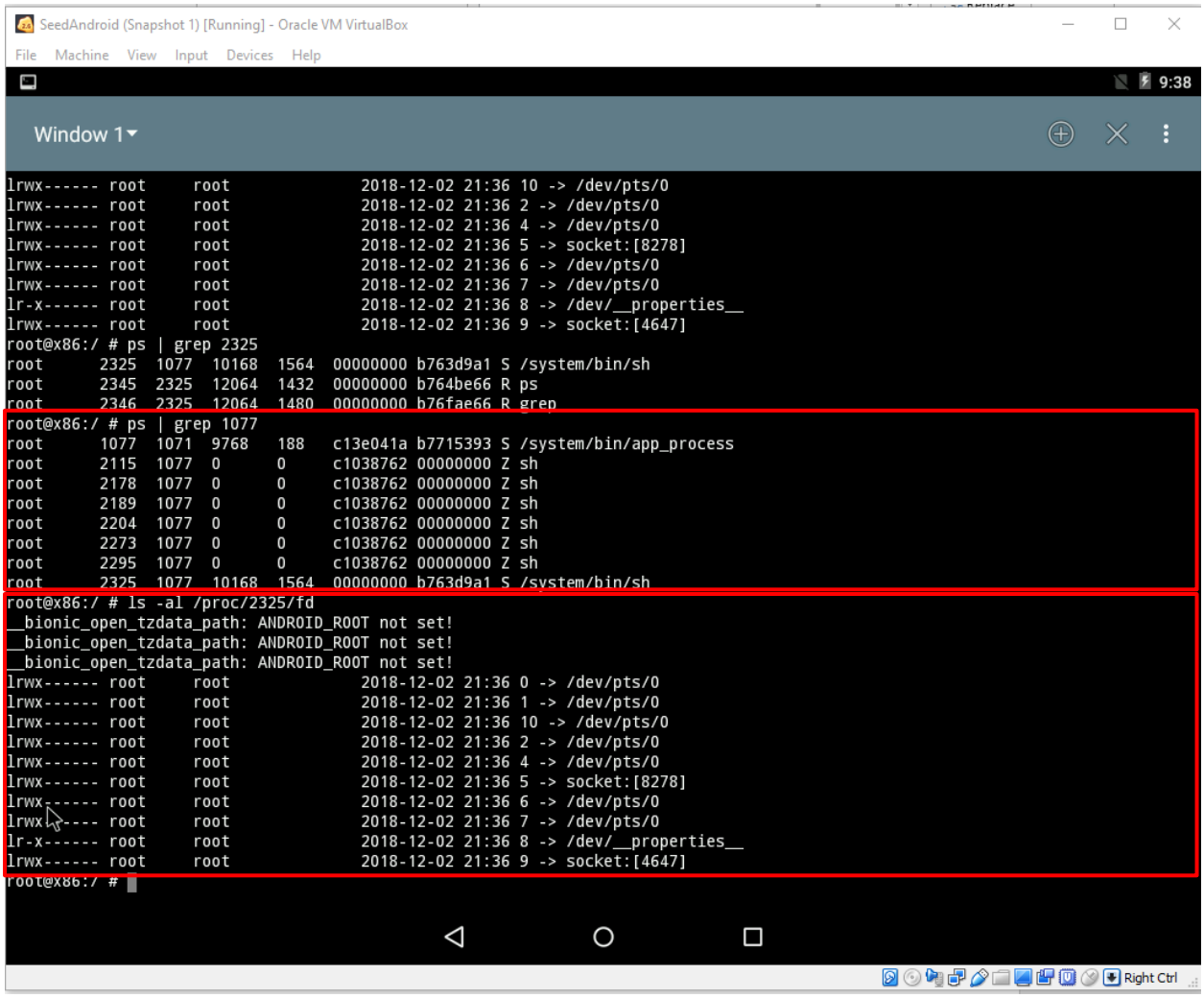
Child process receives client's FDs	mydaemonsu.c	child_process	147
			int client_in = recv_fd(socket);
			int client_out = recv_fd(socket);
			int client_err = recv_fd(socket);

Child process redirects its standard I/O FDs	mydaemonsu.c	child_process	152
			dup2(client_in, STDIN_FILENO);
			dup2(client_out,STDOUT_FILENO);
			dup2(client_err, STDERR_FILENO);

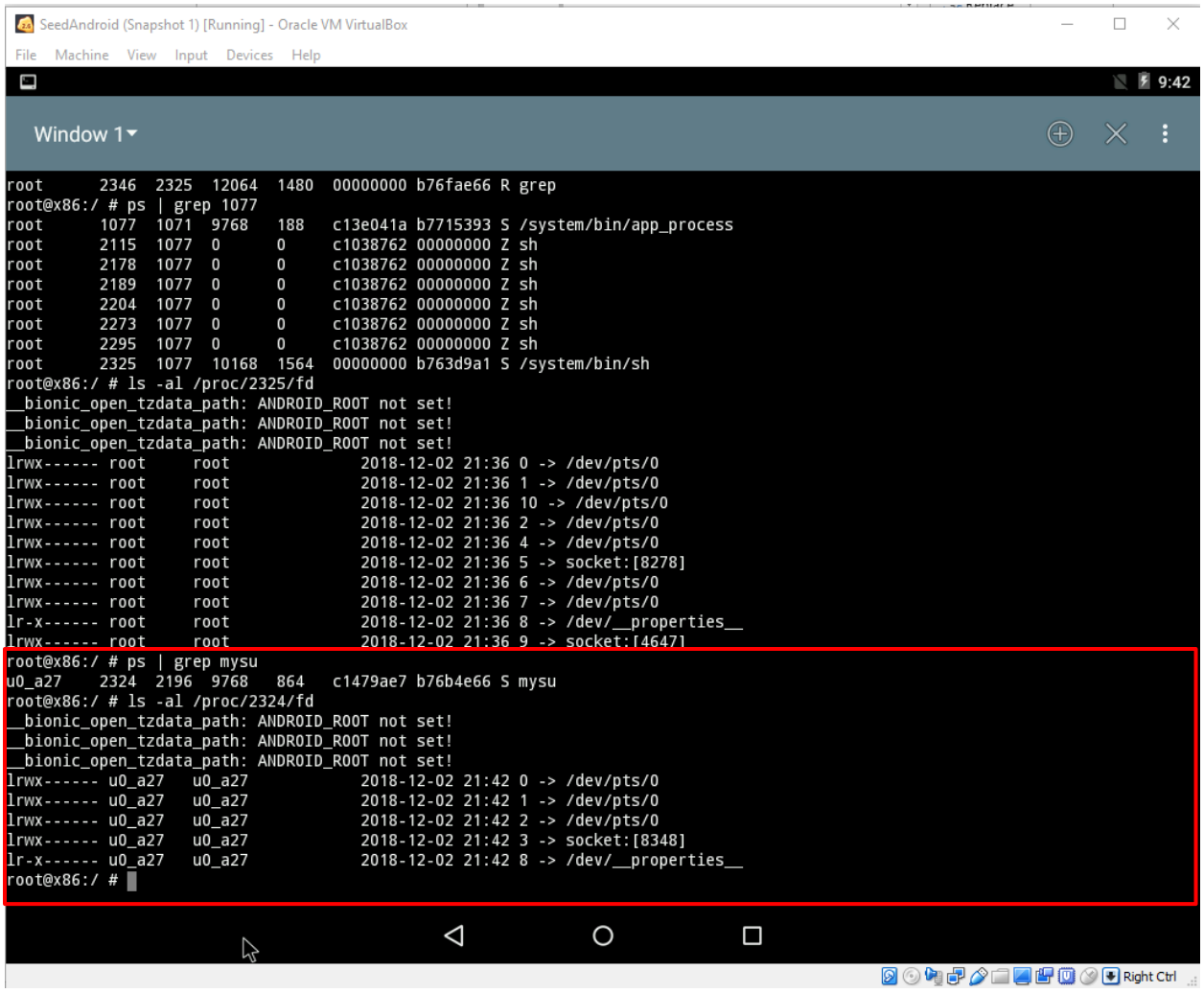
Child process launches a root shell	mydaemonsu.c	child_process	162 execve(shell[0], shell, env);

Subtask: File Descriptor Identification

In the figure below, we see the process details for app_process and the /system/bin/sh root shell. That's the first red block. The second red block, shows the file descriptors associated with the root shell.



In the figure below, we see the file descriptors of the mysu process.



Base on my observations, I believe we have the following file descriptor mappings, from mysu client to root shell:

- STDIN 0 → /dev/pts/0 mapped to STD 4 → /dev/pts/0
- STDOUT 1 → /dev/pts/0 mapped to STD 6 → /dev/pts/0
- STDERR 2 → /dev/pts/0 mapped to STD 7 → /dev/pts/0

Demonstration

The demonstration of this lab is in two parts. One for Task 2 and one for Task 3. They are included in the submission. Please view these files:

1. Vats_Mudit_Task2.mp4
2. Vats_Mudit_Task3.mp4