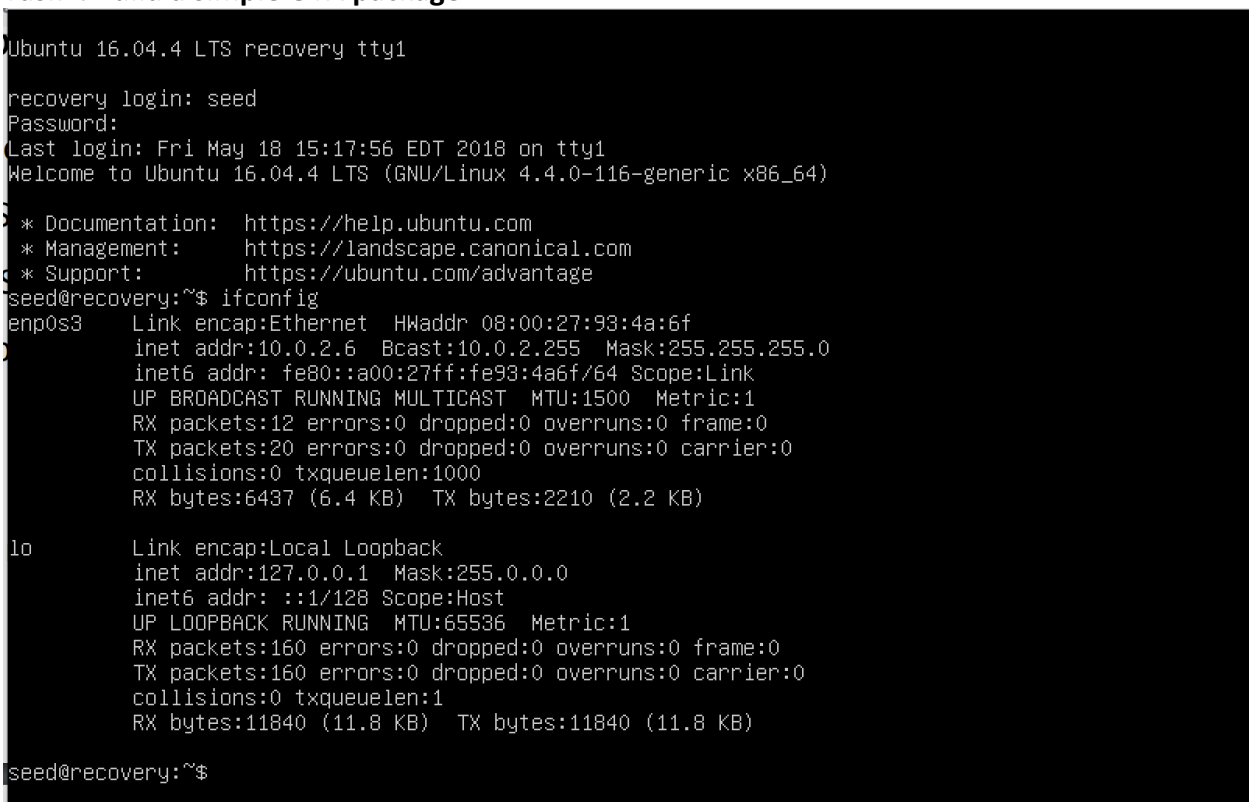
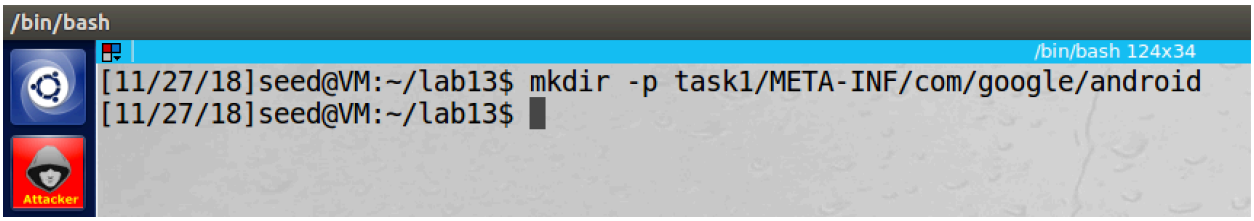
CSE643 Lab13
Yishi Lu
11/30/2018

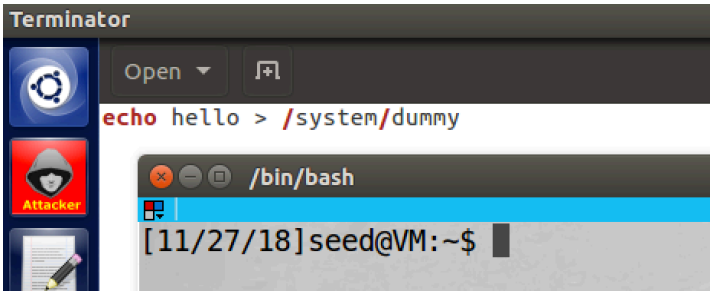
Task1: Build a simple OTA package



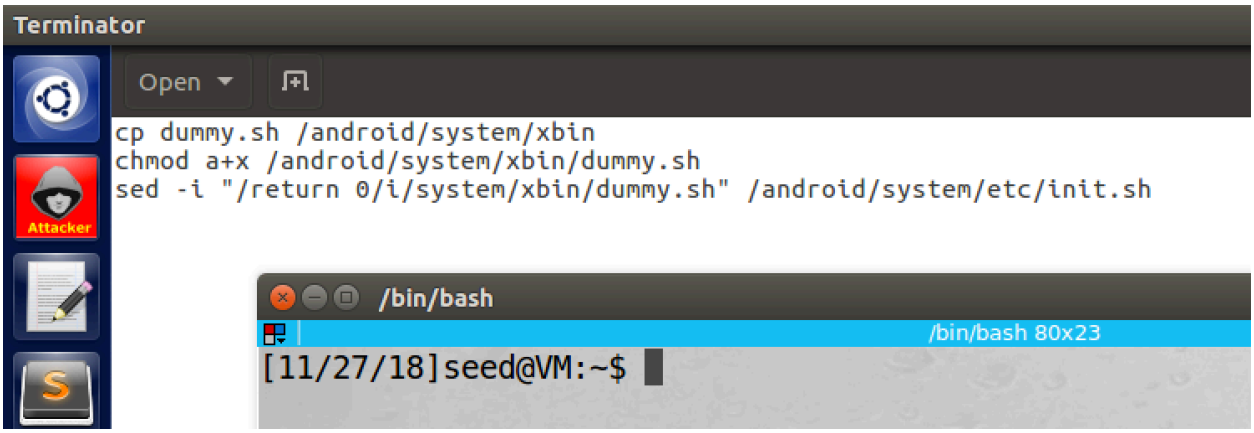
screenshot1, we login to recovery OS of the Android VM



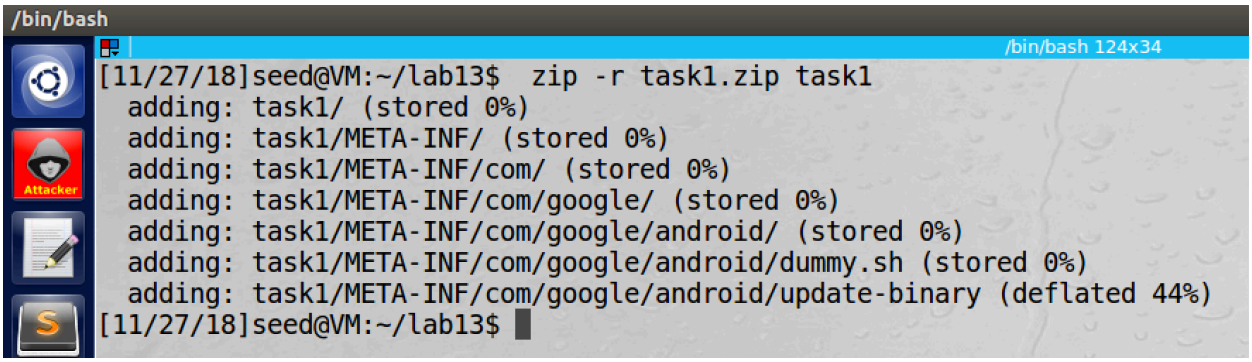
screenshot2, we create the structure of the OTA package



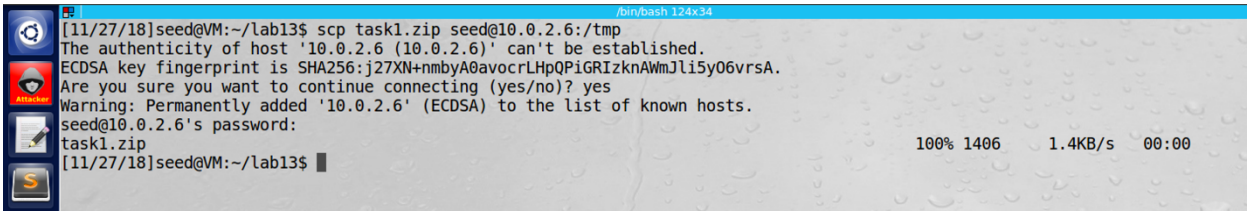
screenshot3, we create dummy.sh file, and we add above content to it



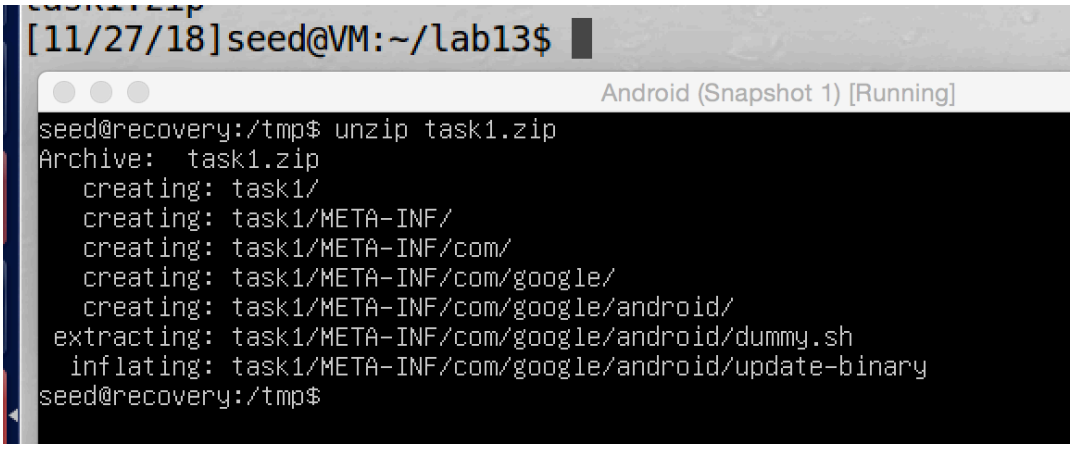
screenshot4, we create update-binary file, and we add above content to it. And we also run chmod a+x to make it executable (for update-binary files in later tasks, we also run this command, so we do not repeat again later)



screenshot5, we compress the OTA package into zip file



screenshot6, we send the OTA package to the Android recovery OS



screenshot7, on the recovery OS, we unzip the OTA package

screenshot8, we run the update-binary file, then we reboot the Android VM

screenshot9, we connect Android VM from the Ubuntu VM by adb command. In the system folder, we see dummy file is created. So our attack is successful

Observation and Explanation:

In this lab, we will build a OTA package, and this package contains our code. Once we install the package in the recovery OS, a dummy file will be added to the /system folder of the Android OS. We divide this task into three steps.

The first step is to write the update script. We create two files; one is called dummy.sh. This file contains our code, we want it to be run when Android boots up with root privilege, and then it will create a dummy file under system folder. Another file is the update-binary file; in this file, we add shell command of copying dummy.sh to /system/xbin folder on Android OS, and then we make dummy.sh executable. Finally, we insert a command in file init.sh. This file

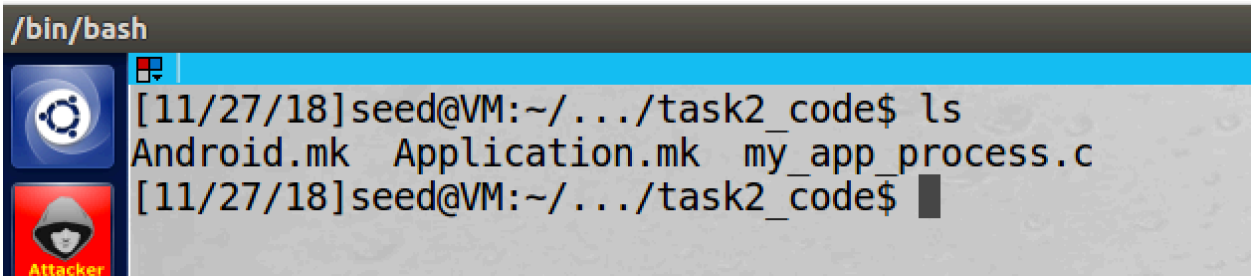
will be run with root privilege when Android system booting. So if we place command to run dummy.sh in this file before it finish, dummy.sh can be run with root privilege.

The second step is to build the OTA package. The structure of the OTA package is shown in screenshot2, and we place dummy.sh and update-binary files all in folder android.

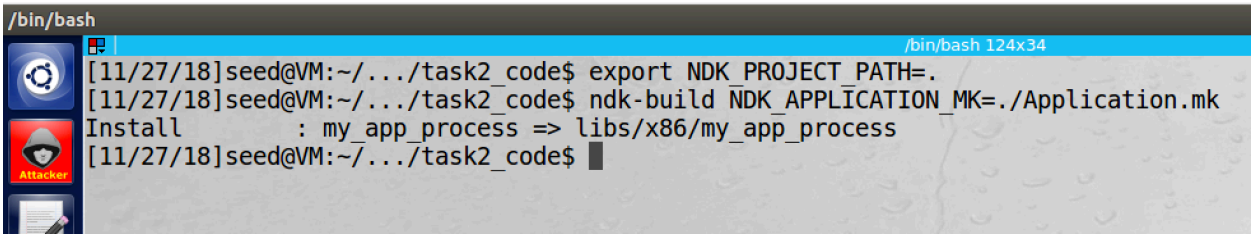
The last step is to run the OTA package on the recovery OS, then after we rebooting to Android system, the dummy file should be added in folder /system.

As screenshot1 shows, we first login to recover OS. Then we create the OTA package (screenshot2). As screenshot 3 and 4 show, we build dummy.sh and update-binary files, and we put them under /android. Then we zip the OTA package and send it to the recovery OS (screenshot 5 and 6). On the recovery OS, we unzip the OTA package, and we run the update-binary file (screenshot7 and 8). Then we reboot the Android OS. To check if our attack is successful or not, we go back to the Ubuntu VM, and we connect to the Android OS by adb command; and then we run a shell program on Android by command adb shell. As screenshot9 shows, the dummy file is added under /system. So our attack is successful.

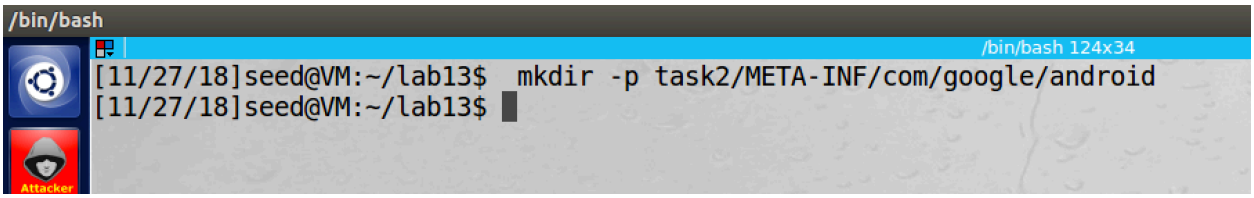
Task2: Inject Code via app-process



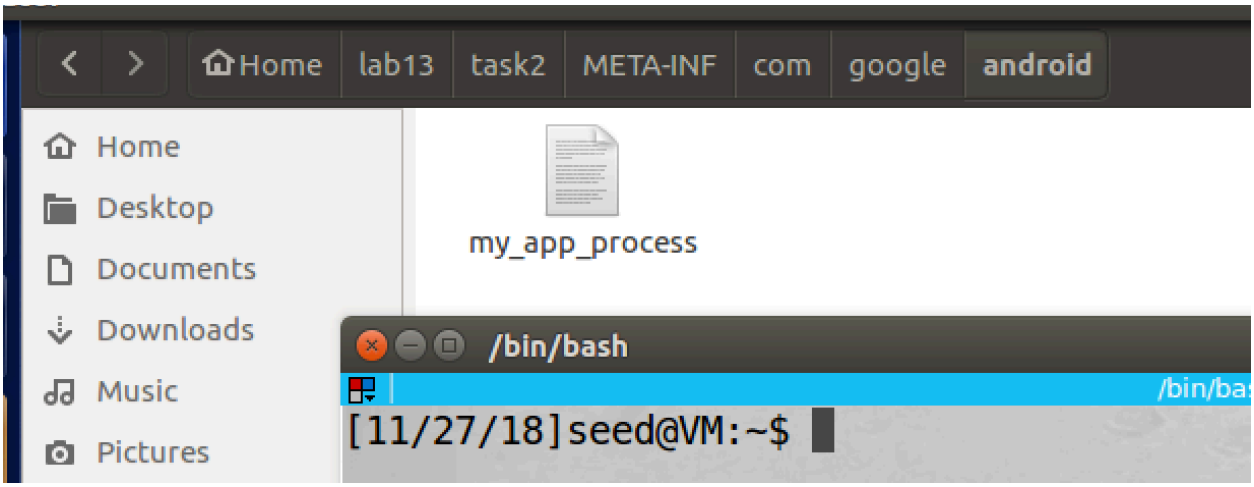
screenshot1, we copy code Android.k, Application.mk, and my_app_process.c from lab description



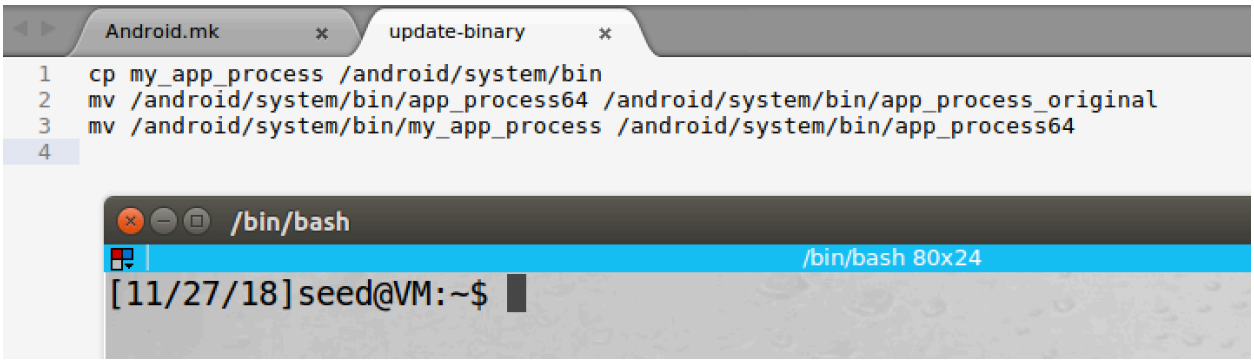
screenshot2, we compile our program by NDK successfully



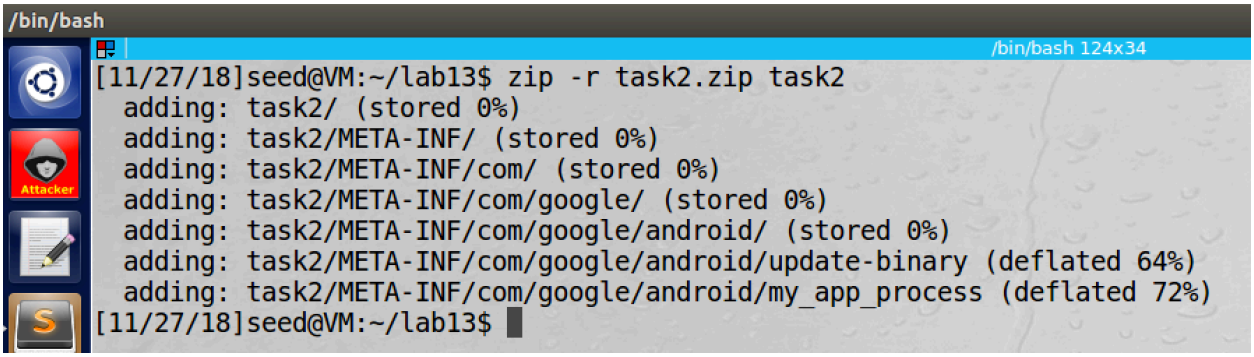
screensho3, we create OTA package structure, it is similar as the one in task1



screenshot4, we put the compiled file in folder /android of OTA package

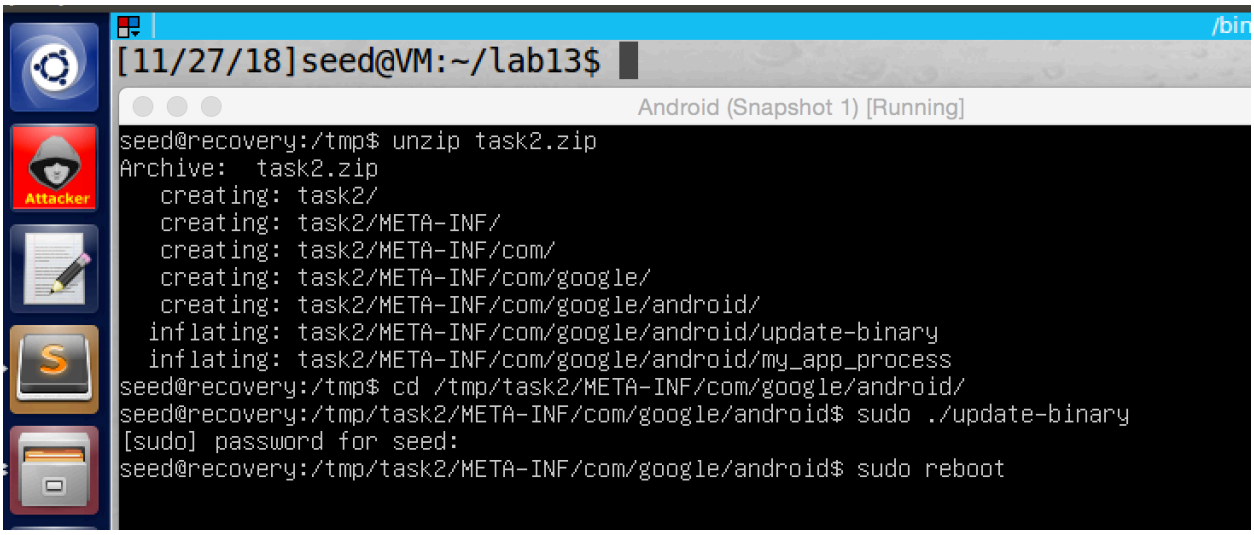


screenshot5, we create the update-binary file in the /android of OTA package

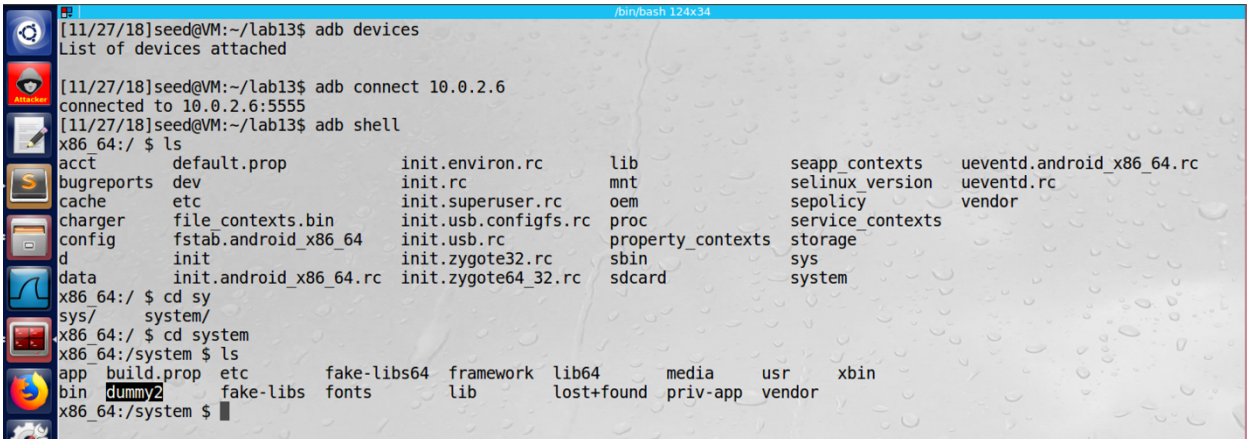


screenshot6, we zip the task2 OTA package

screenshot7, we send it to the recovery OS



screenshot8, on the recovery OS, we unzip the OTA package, and we run the update-binary file, then we reboot the VM to enter android OS



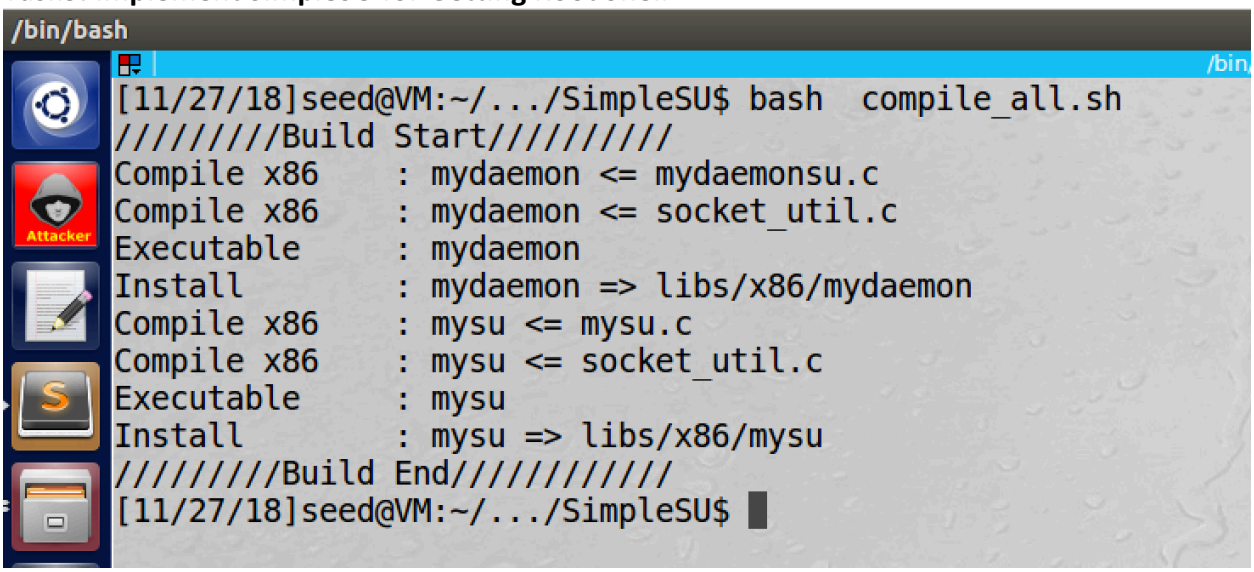
screenshot9, we still use adb command to access the system file of the android OS, and we see dummy2 is added in the /system folder, so our attack is successful

Observation and Explanation:

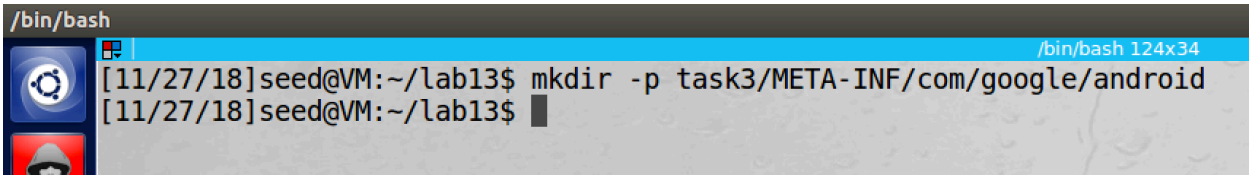
In this task, we perform same attack as task1, but this time we use app_process approach. When android OS is booting, a sequence of procedures will be run. There are three main phases. First, the kernel phase, in this phase, Android kernel will be loaded to initialize the system. The next phase is the Init process. It is the first user-space process, and it running with root privilege. The third phase is the Zygote process. In this phase, a daemon called Zygote will be run, and it will execute app_process binary, and it also run with root privilege. This is our attack point in this task. Our goal is to modify app_process, so our code can be run with root privilege. So we first need to create the app_process binary, we copy the code from lab description. To compile the app_process program, we need to use NDK. To use NDK, we need to create two files, Application.mk and Android.mk, we also get them from lab description (screenshot1). After we compile the app_process program, we can find the compiled binary file in the folder /lib/x86 (screenshot2). Then we need to build the OTA package and update-binary

file. OTA package has same structure as the one task1, so we do not repeat here (screenshot3). The update-binary file is different, in this task, we need to do two things; first we need to move our app_process file into /android/system/bin folder. Then we need to change the name of the original app_process file to other name (app_process_original), then we also change the name of our app_process file to app_process64 (screnshot5). After we contracture the update-binary file, we still put it in the folder /android of OTA package; moreover, we put our app_process file in the same folder as well (screenshot4). Now, we have everything we need. Then we follow step in task1 to install the OTA package in the recover OS (screenshot6 and 7). After we run the update-binary file on the recovery OS, we rebooting the Android OS (screensho8). And we also use adb command on Ubuntu VM to check the /system folder of the Android OS, and we see dummy2 is added in the /system; thus our attack is successful (screenshot9).

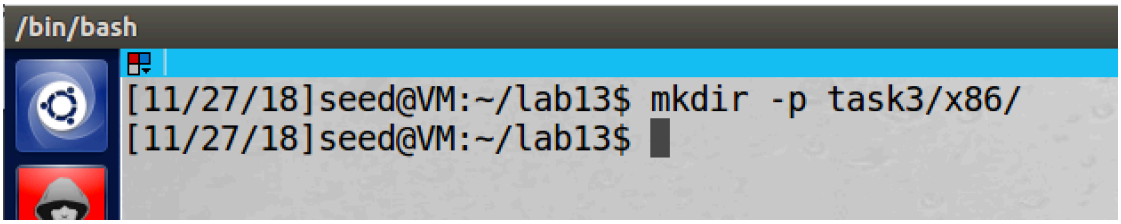
Task3: Implement SimpleSU for Getting Root Shell



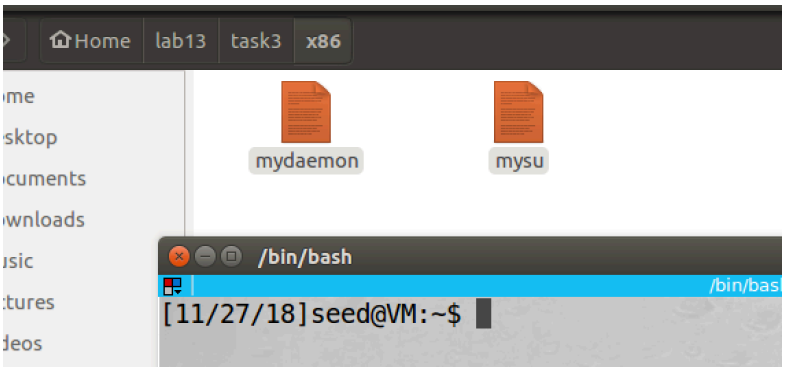
screenshot1, we get the SimpleSU file form lab website, then we run bash compile_all.sh to compile the program, and then we get two compiled binary files mydaemon and mysu.



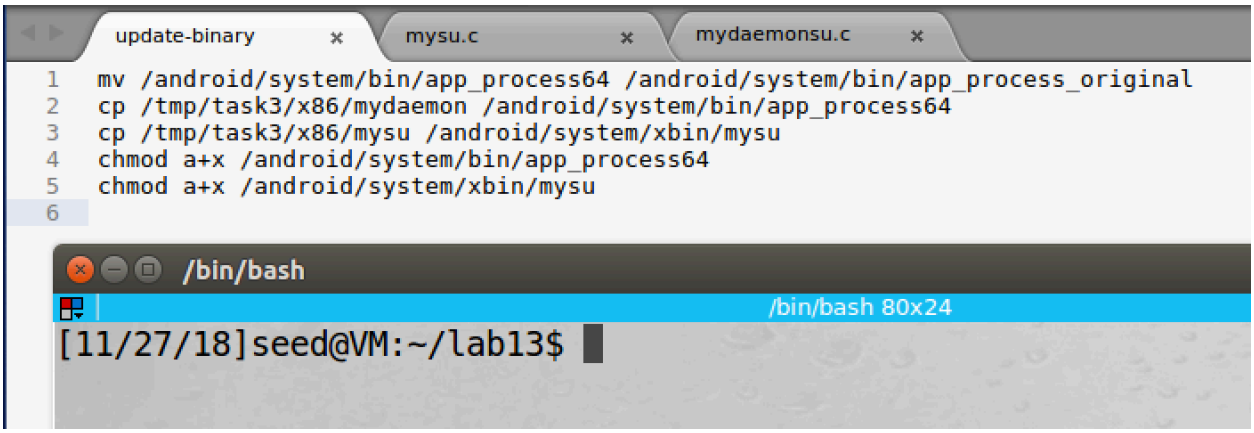
screenshot2, we create OTA structure for task3



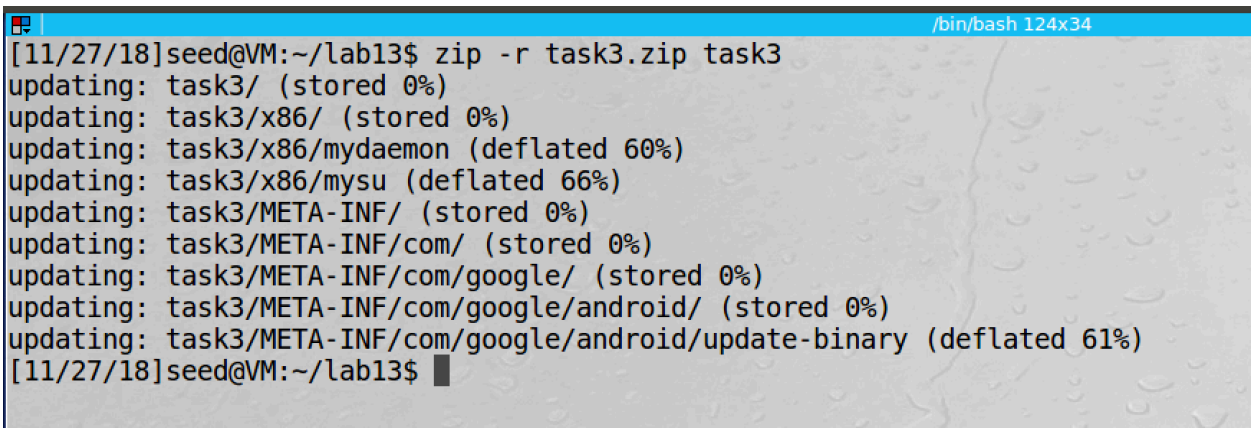
screenshot3, we create /x86 folder under the OTA structure



screenshot4, we move mydaemon and mysu files into /x86 folder

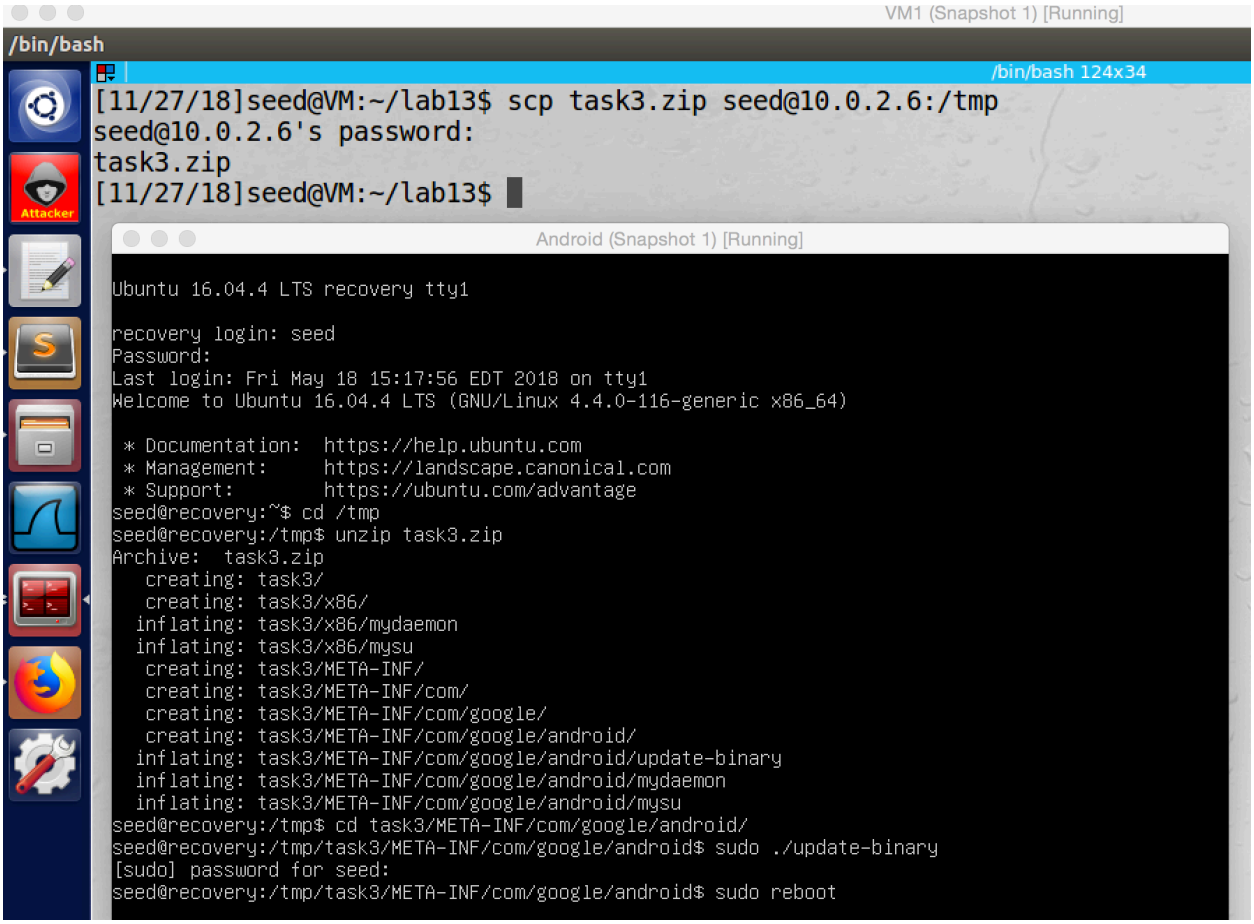


screenshot5, we construct the update-binary file, and we put it in the /android of OTA package

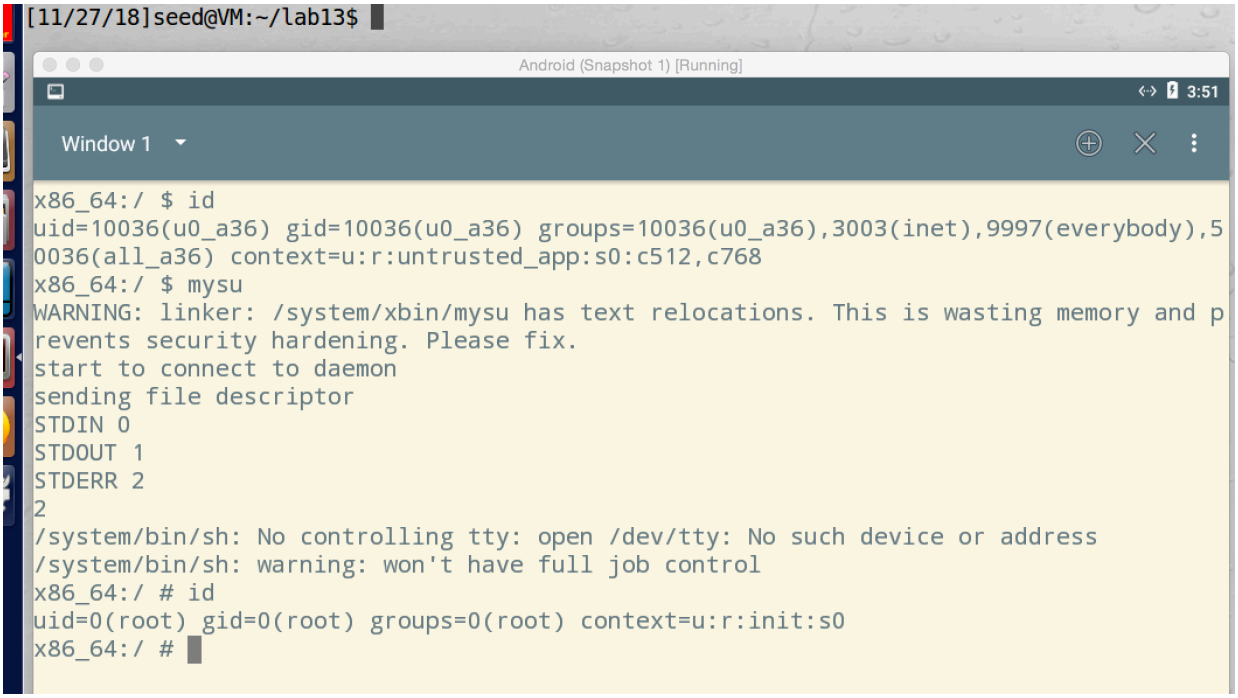


screenshot6, we compress the OTA package

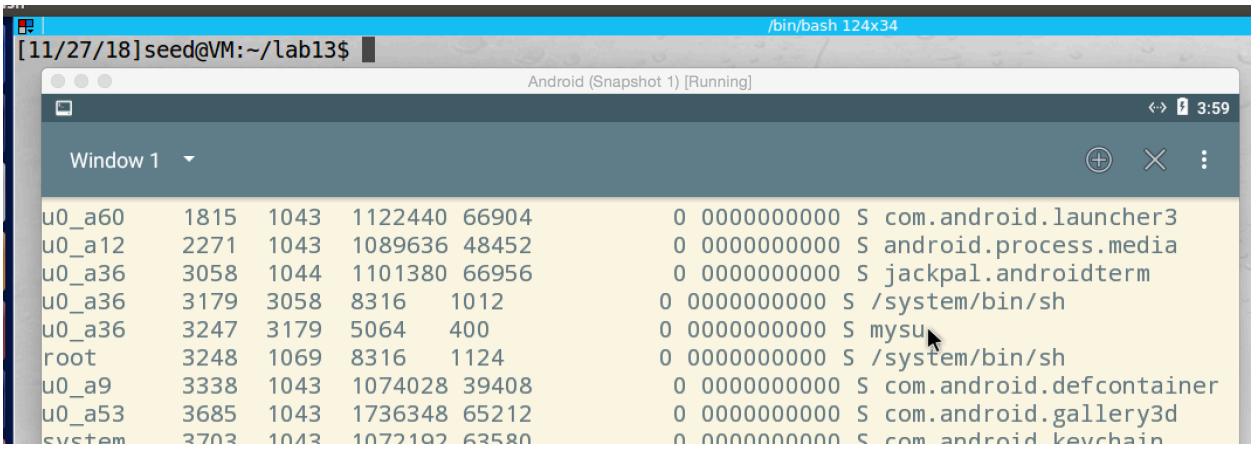
screenshot7, we send it to recovery OS



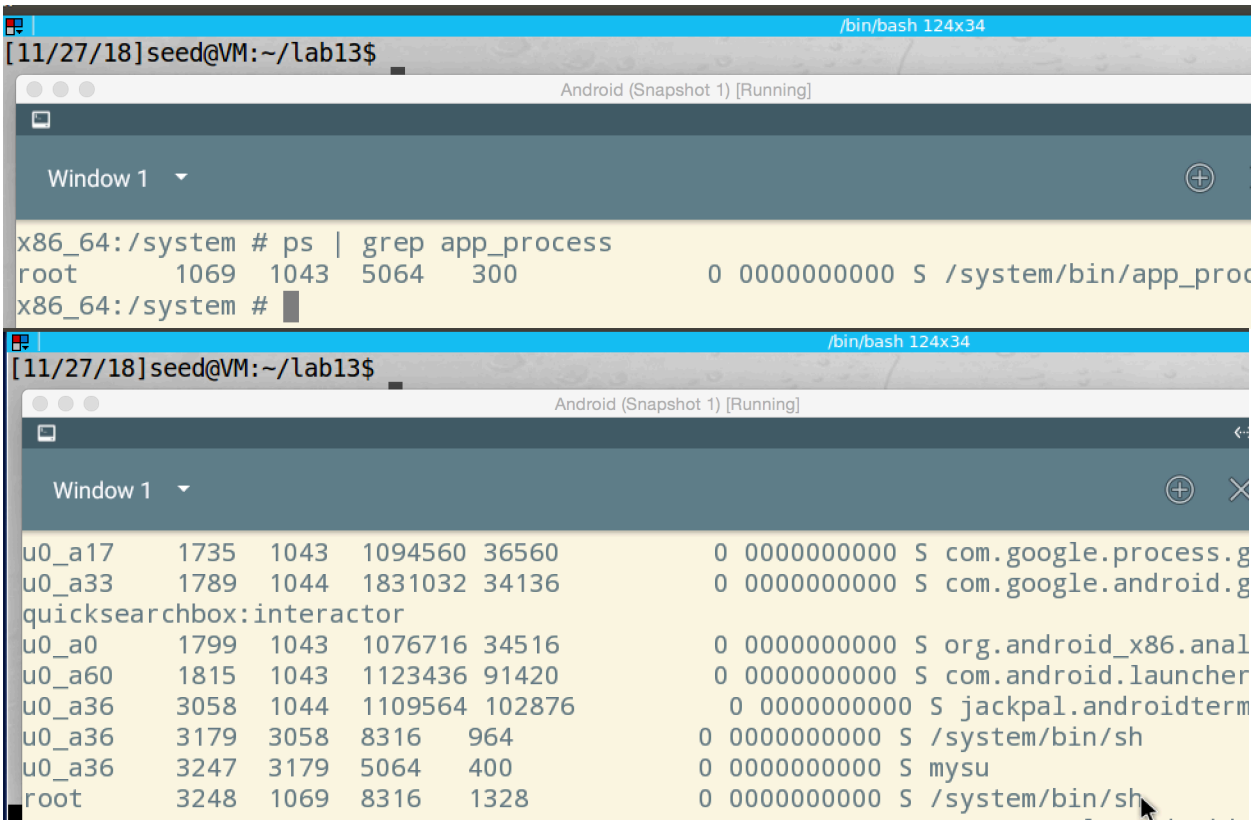
screenshot8, we run the update-binary file on recovery OS, then we reboot the system



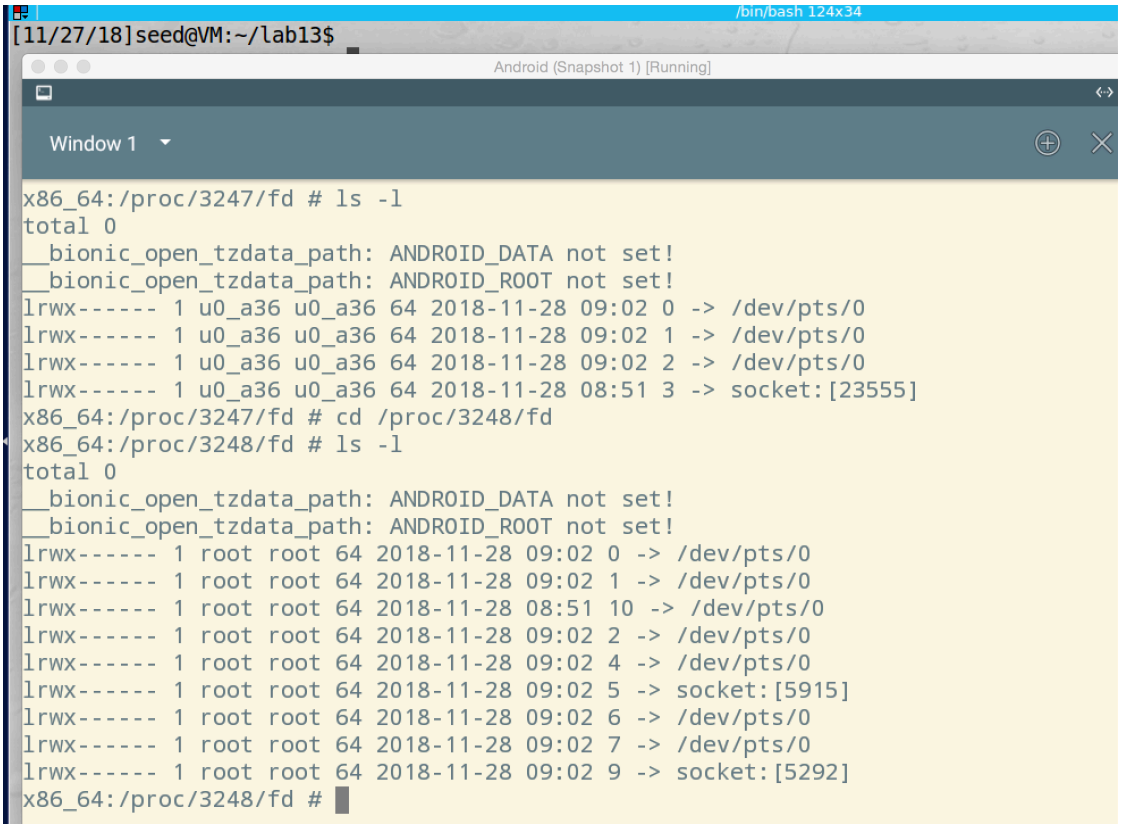
screenshot9, we run mysu on the terminal, and we get a root shell



screenshot10, we get PID of mysu which is 3247



screenshot11, to find the child process PID, we first find server process PID which is 1069. Then we run command ps to find process with PPID 1069, which is 3248, so this is the child process



screenshot12, we get the file descriptor of mysu and the child process

Observation and Explanation:

In this task, we show how to gain a root shell by SimpleSU on Android OS. Because on Android OS, there is no Set-UID program, we cannot gain a root shell by it. So we use another approach, we run a daemon program when the OS is booting, so the daemon program run with root privilege. Then we run a client program to make a request to the daemon program, then the daemon program create a shell process and give the control to us. Then we get a root shell. However, there is a problem which we need to solve. The shell process generated by the daemon program, and it inherits output device, input device, and error device from its parent, so we cannot control these devices. Our solution is to give the input, output, and error device of the client program to the shell process and make these devices also become devices of the shell process. Then user can use these devices to input to the shell process and receive result (print) from shell process. Moreover, in Linux, each process use file descriptors (FDs) 0, 1, 2 to represent input, output, error devices; and it also allow process send its FDs to other process by Unix Domain Socket. So the whole process is following: when the Android OS is booting, a daemon program is running (server). After client program connect to it, the server will fork a shell process, and the shell process inherits input, output, and error devices from the server. Then the client process sends its FDs to the shell process by Unix Domain Socket, and then the shell process uses these FDs as its stdin, stdout, and stderr. Now the client process can have the full control of the shell process. Because the shell process is running with root privilege, the client process actually run with root privilege as well.

As screenshot1 shows, we get the server and client program from lab website, and we compile them to get two binary file. Then we construct the OTA package, and we also add a new folder /x86, we put these two compiled files into this folder (screenshot 2, 3, 4). Then we construct the update-binary file (screenshot5). when this file run, the original app_process file will be renamed to app_process_original. Then we also put the mydaemon file into the /system/bin with name app_process64. We also put mysu file into /system/xbin. And we also set them to be runnable by command chmod a+x. Now we constructed the OTA package, so we compress it, and send it to the recovery OS (screenshot 6, 7). On the recovery OS, we unzip the OTA package, and then we run the update-binary file, then we also reboot the system (screenshot8). Then we enter the Android OS, and we run terminal app. We first run command id to check the user id, which is a normal user. Then we run the client program mysu to connect to the server, and we get a root shell. To prove this, we run command id again, and it shows root (screenshot9). To find the PID of process mysu, we run command ps, and we see its PID is 3247 (screenshot10). We also need to find the child process ID. We know that the server process has name app_process64, so we get its PID; then we run command ps again to find process with PPID 1069. We find it, the child process has PID 3248 (screenshot11). Then we run command ls –l to get their FDs. As secreenshot12 shows, we get all FDs of child process and client process. Their FD 0, 1, 2 point to same place, which is terminal device /dev/pts/0; moreover, FD 4, 6, and 7 of child process are also point to /dev/pts/0. So they share the same stand in, stand out, and stand error devices.

Server launches the original app_process binary:
File: mydaemonsu.c, Function: main(), Line#: 255

Client sends its FDs:
File: mysu.c, Function: connect_daemon(), Line#: 112-114

Server forks to a child process:
File: mydaemonsu.c, Function: run_daemon(), Line#: 189-193

Child process receives client's FDs
File: mydaemonsu.c, Function: child_process(), Line#: 147-149

Child process redirects its standard I/O FDs
File: mydaemonsu.c, Function: child_process(), Line#: 152-154

Child process launches a root shell
File: mydaemonsu.c, Function: child_process (), Line#: 162