Name: Bharath Karumudi
Lab: Android Rooting

Task 1: Build a simple OTA package

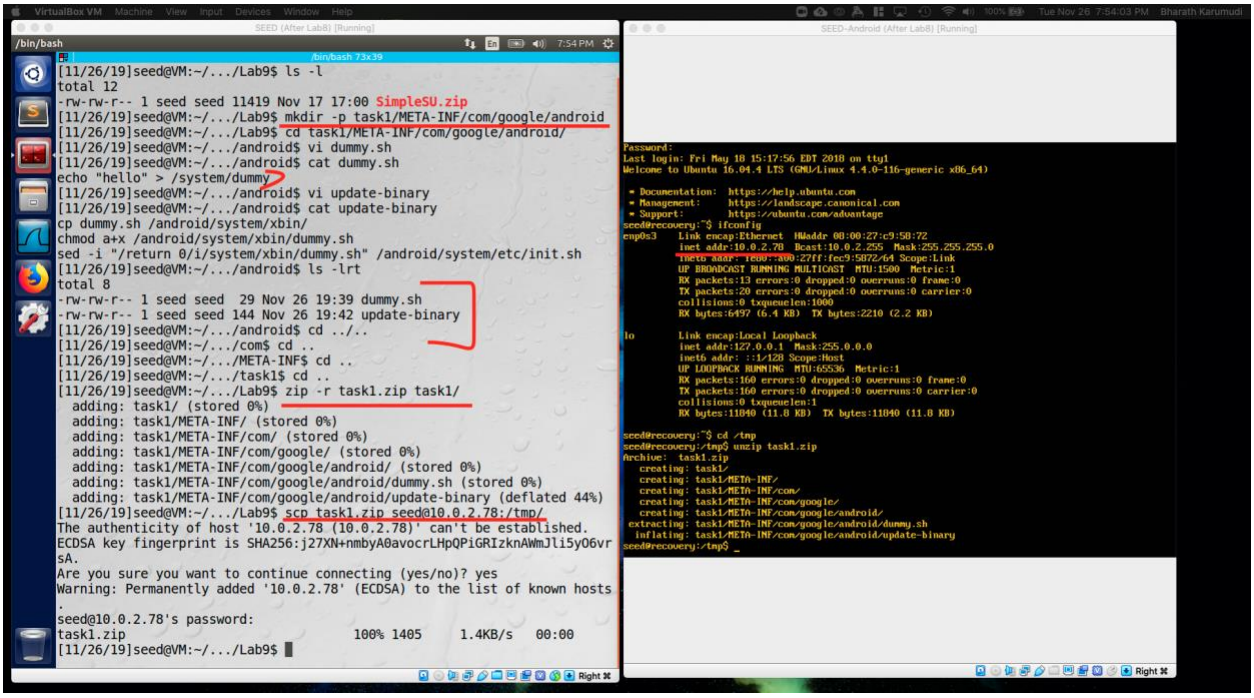


Fig: Building the Package and contents and transferring the OTA to Recovery OS

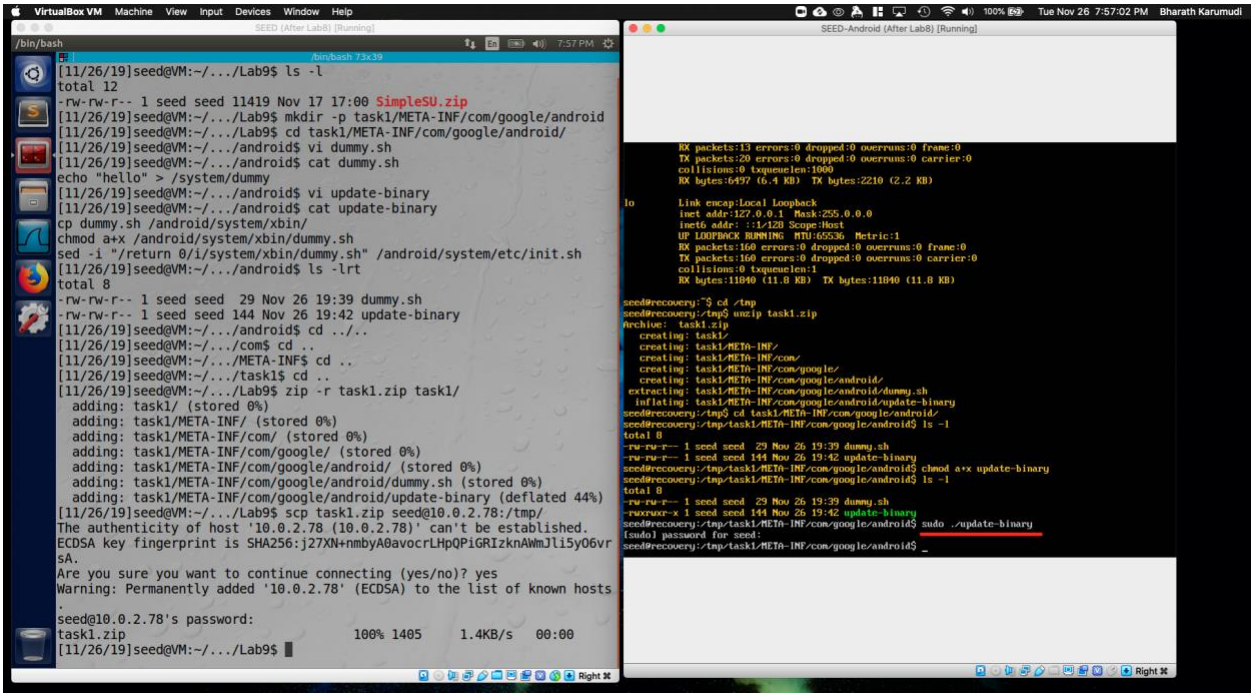


Fig: Executing the OTA script inside Recovery OS

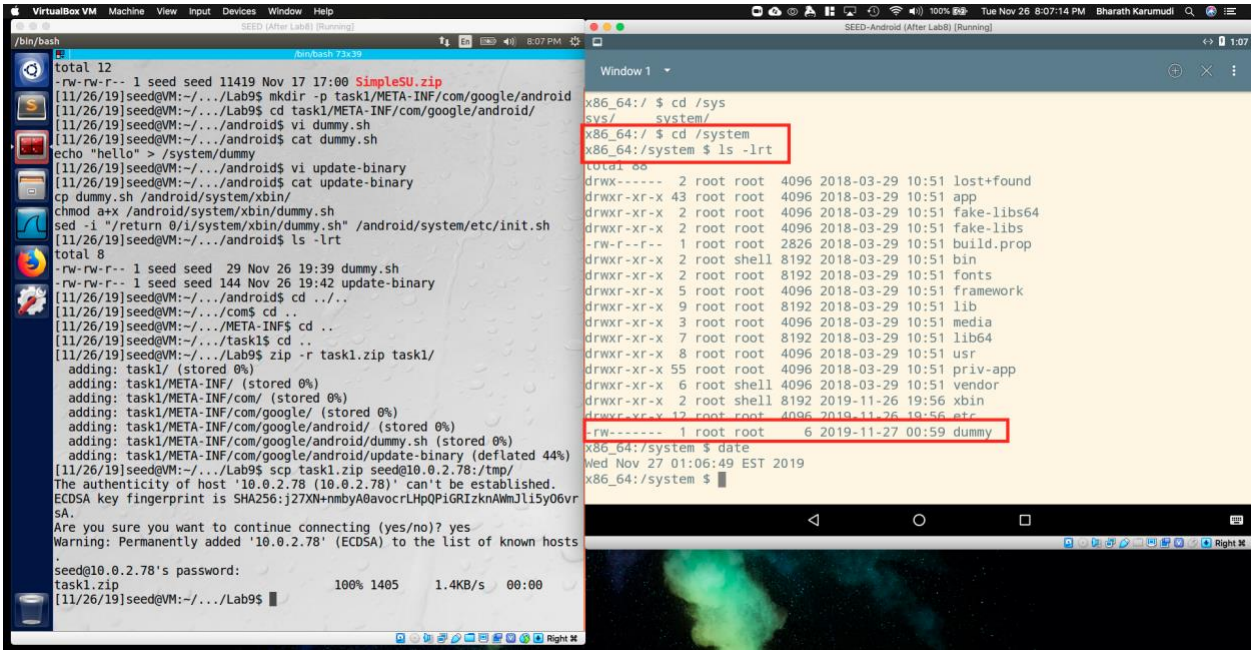


Fig: Terminal Emulator in Android showing “dummy” file

Observation: Created an OTA like package in SEED Ubuntu with required structure and zipped the files and sent to Android Recovery, which is running at 10.0.2.70. Once the zipped package is unzipped and ran the updated binary script and booted to Android and by using terminal emulator, I can see the file “dummy” under system directory.

Explanation: Created the directory structure, task1/META-INF/com/google/android/ and under the android directory, placed a “dummy.sh” which creates a dummy file under /system and update-binary which copies the dummy.sh to /android/system/xbin/ and also updates the init.sh script to run the dummy.sh
Once all the files are placed

1. zipped the package as task1.zip
2. Booted into Android Recovery OS and got its IP address as 10.0.2.70
3. scp the task1.zip to Android Recovery OS (10.0.2.70) under /tmp
4. Inside the Android recovery OS, extracted the zip and executed the update-binary script
5. Booted to Android and with the help of Terminal Emulator App, navigated to /system and can see the dummy file.

The update-binary script copied the dummy.sh script to /android/system/xbin/ in recovery OS which is equivalent to /system/xbin/ in Android OS. The update script updated the init.sh to trigger our dummy.sh during boot process. During boot process, root id will be in effective and thus with root id, the init.sh triggers which in turn triggers our dummy.sh script.

Thus, the dummy file was created successfully under /system directory in Android which is root owned.

Task 2: Inject code via app process

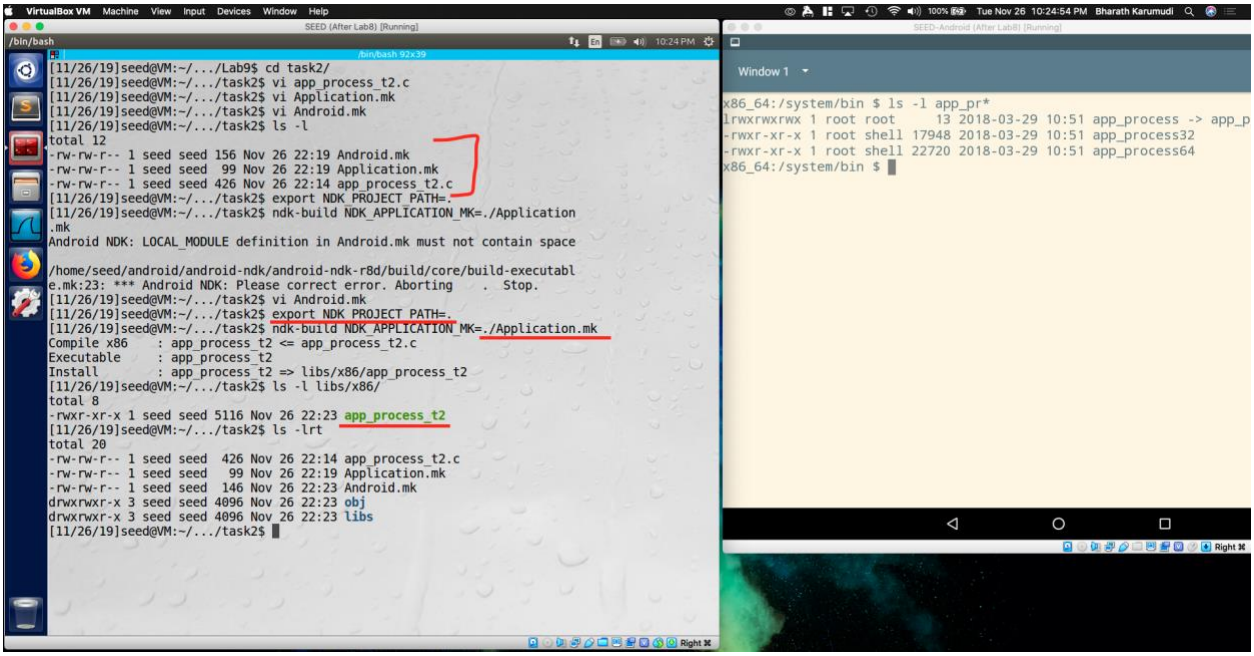


Fig: Compiling the app_process_t2.c to Android Native

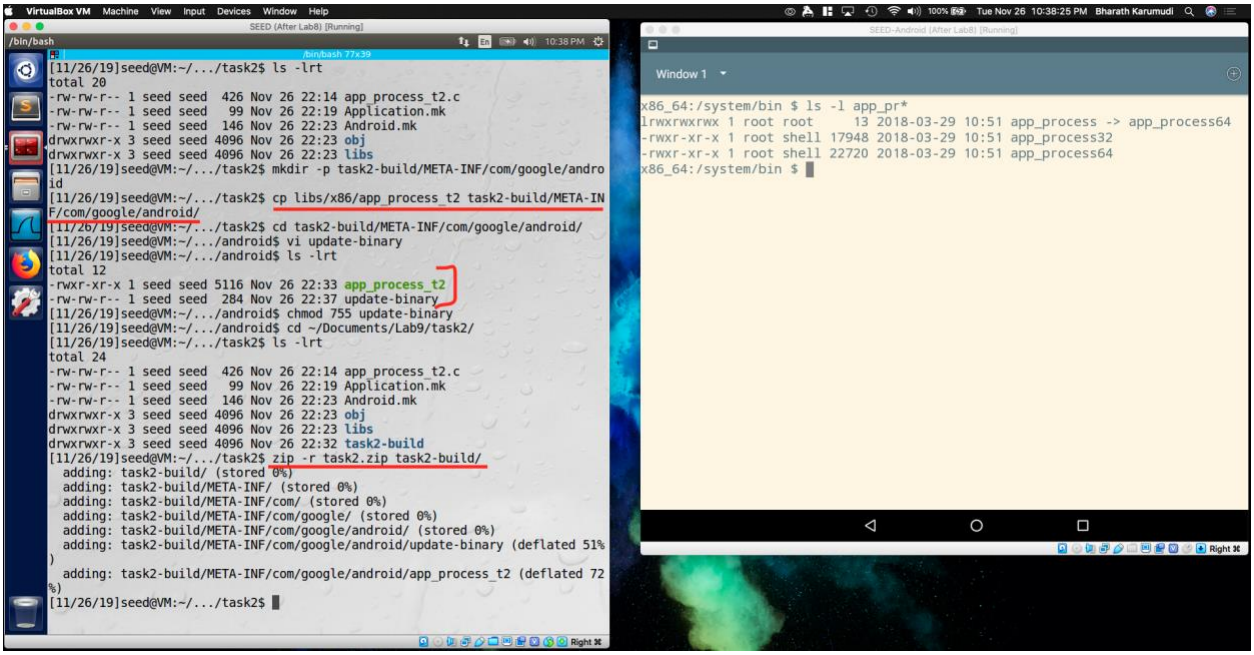


Fig: Building the OTA package

With app_process32 (optional)

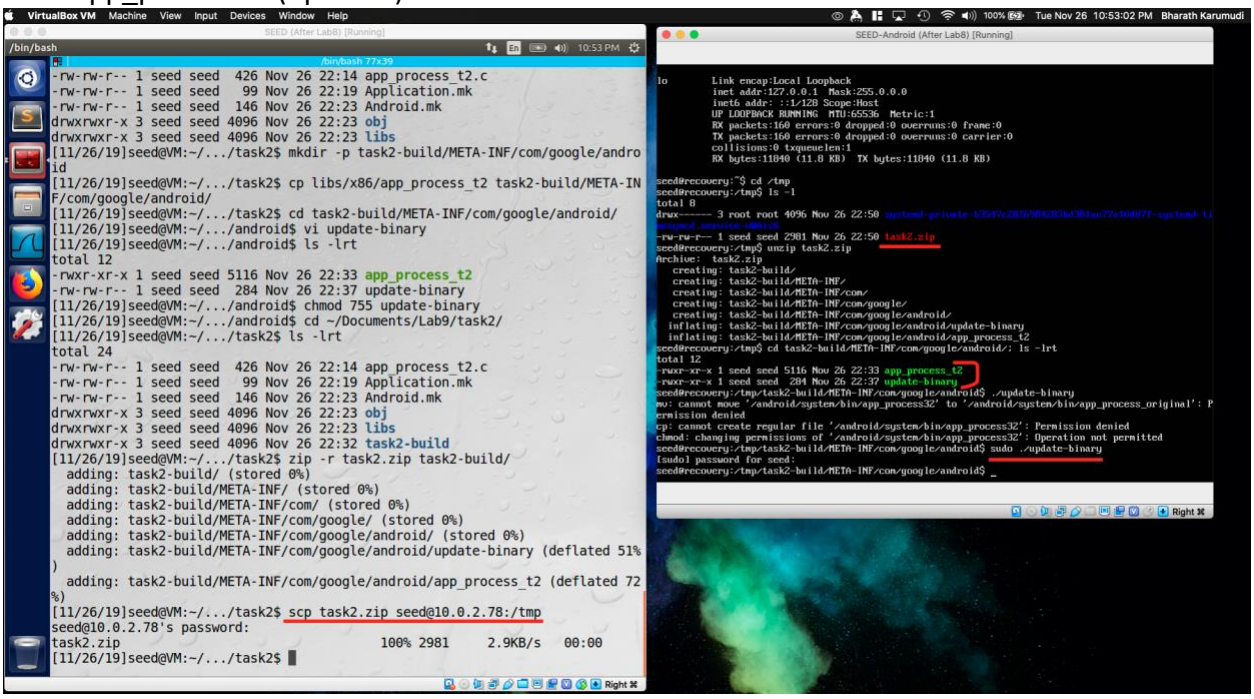


Fig: scp the OTA package to Recovery OS and executing the OTA in Recovery OS

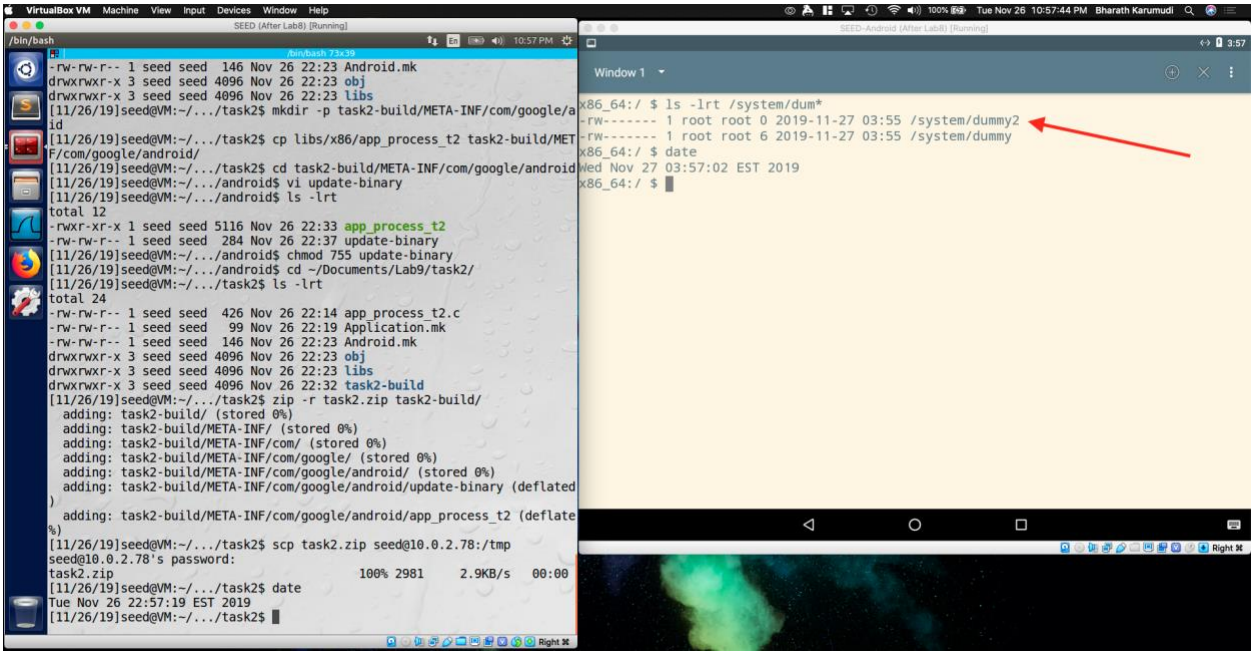


Fig: Process created the dummy2 file

(At this point for some reason, Android not booting, so restored VM to previous snapshot).

With app_process64:

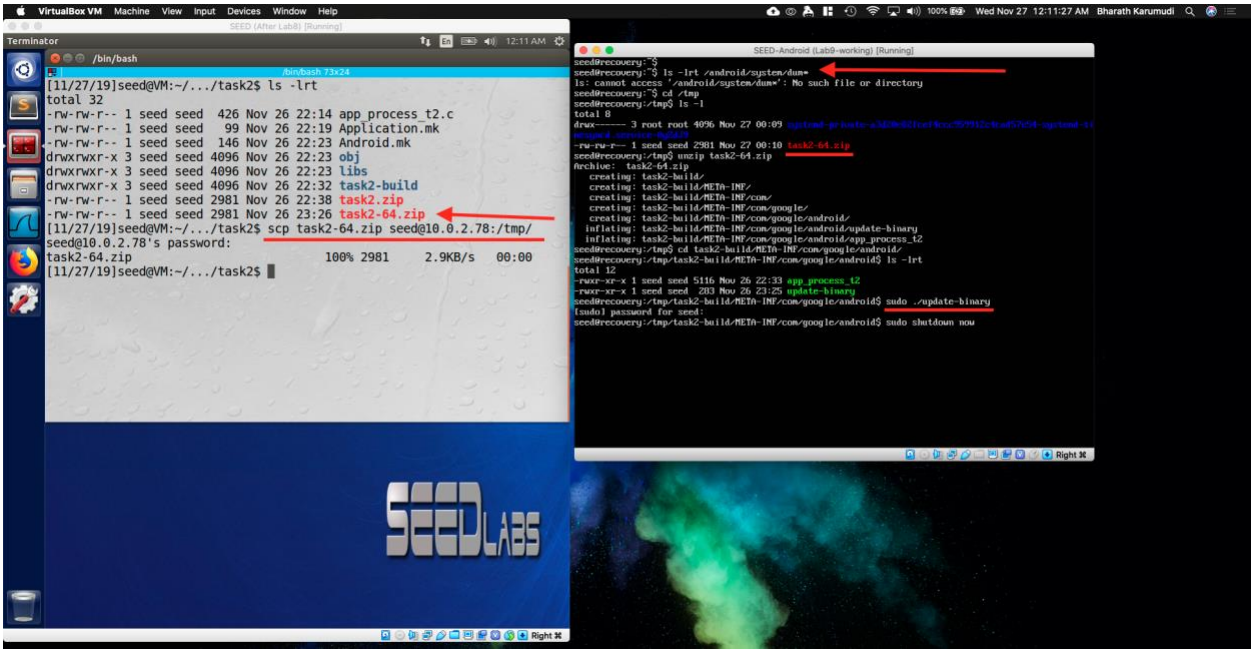


Fig: Building the OTA package and scp to recovery OS and executing the OTA

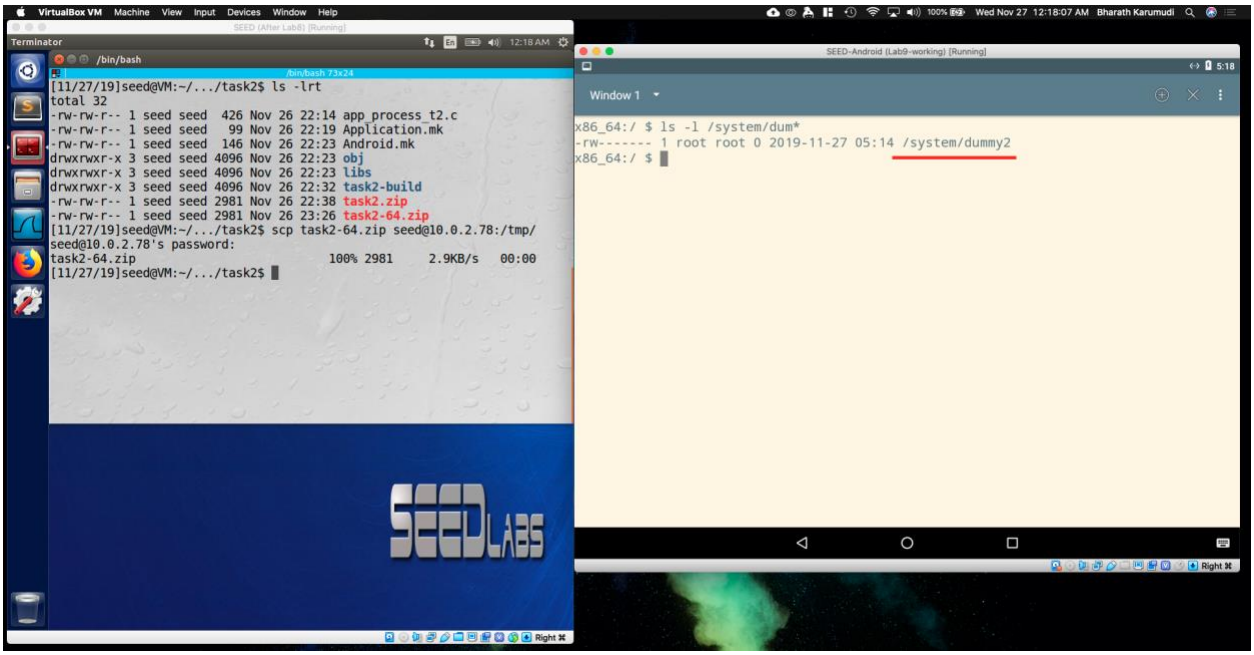


Fig: Process created the dummy2 file in Android

Observation: Created an OTA package that replaces the /system/bin/app_process with our custom app_process and when executed the update_binary process in the Android recovery OS and once booted the Android, I can see dummy2 file created.

Explanation:

1. Created “app_process_t2.c” and compiled the code for Android native with a executable name as “app_process_t2”
2. Created the Android OTA structure: task2/task2-build/META-INF/com/google/android
3. Created the update-binary script under android with instructions to take the backup of original app_process file and then replace the original with our custom process.
4. Built the package and scp to recovery OS as /tmp/task2.zip
5. Inside Recovery OS, unzipped the task2.zip file and executed the update-binary script.
6. Booted to Android and with the help of Terminal Emulator, under /system we can see dummy2 file.

In this attack, we used the Android app_process, during the booting, Android runs this app_process using root privilege which starts the Zygote daemon, which is a parent of all applications. We modified this app_process in such a way, it first creates the /system/dummy2 file and then proceeds to app_process_orignal (which is actual app_process). Our OTA renames the app_process to app_process_original and then places our malicious one as app_process. So during boot process, Android uses our app_process and executed as we instructed.

Thus, the dummy2 file was created by the app_process.

Task 3: Implement SimpleSU for Getting Root Shell

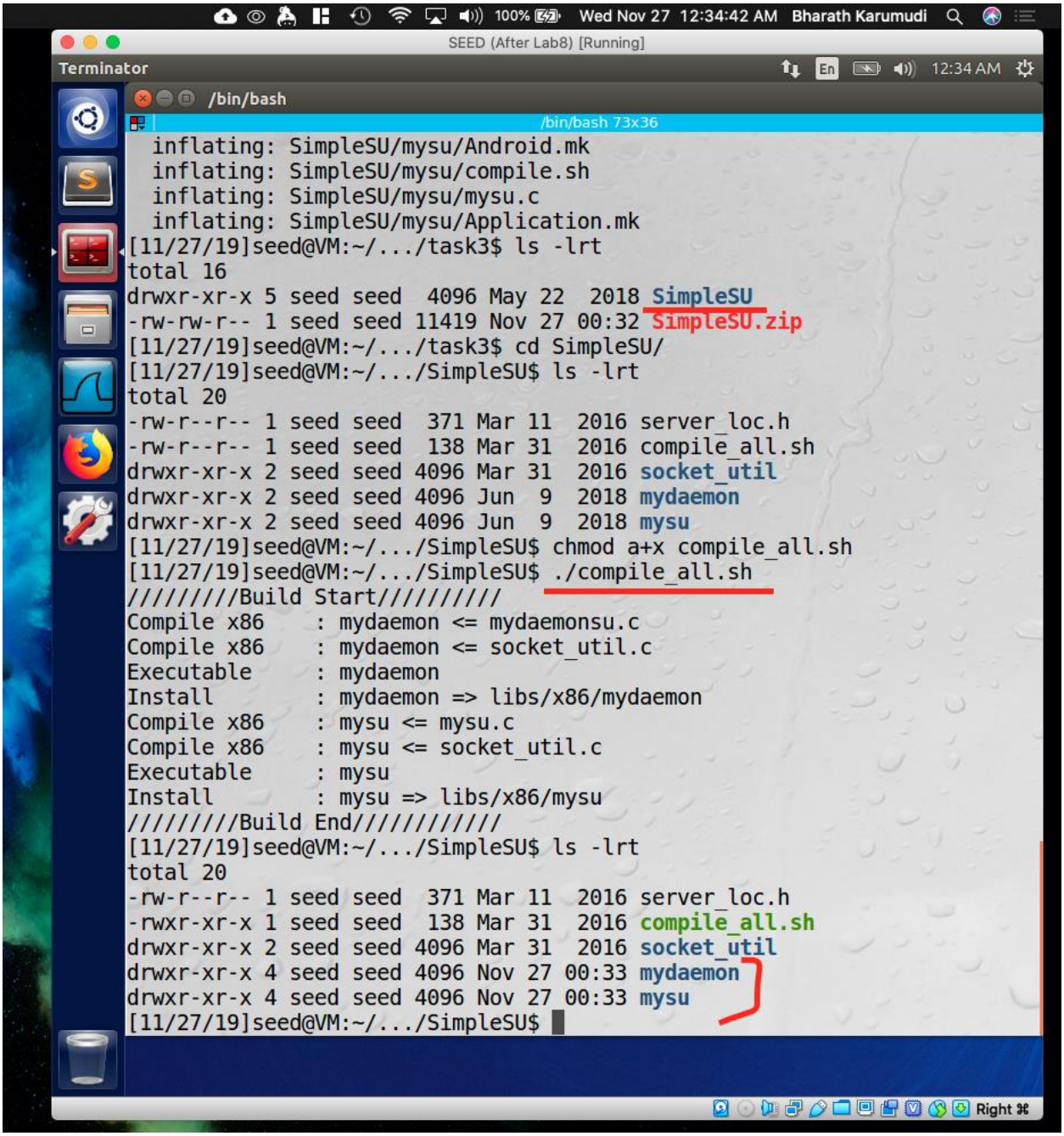


Fig: Compiling the mysu and mydaemon processes to Android Native

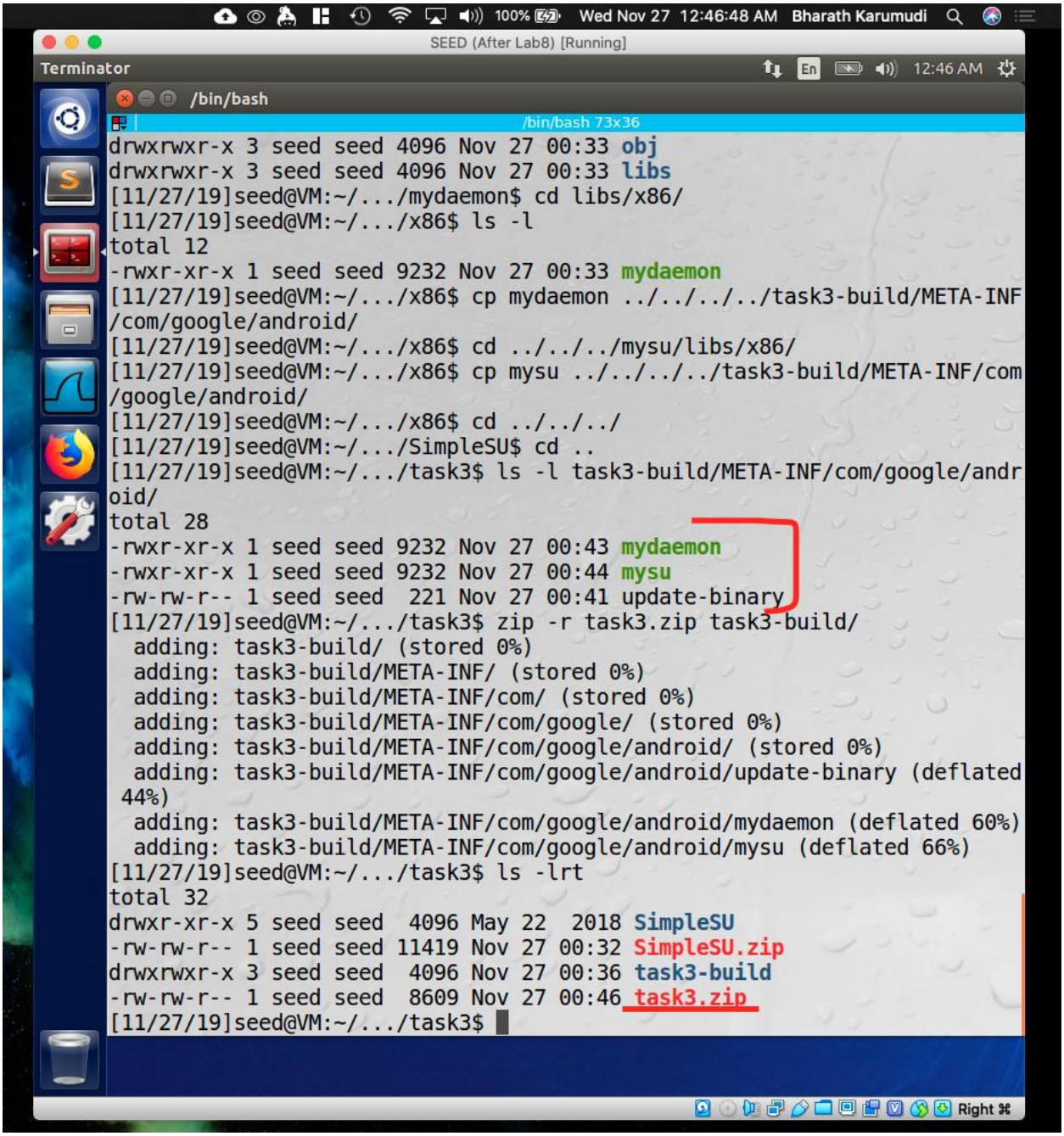


Fig: Building the OTA package

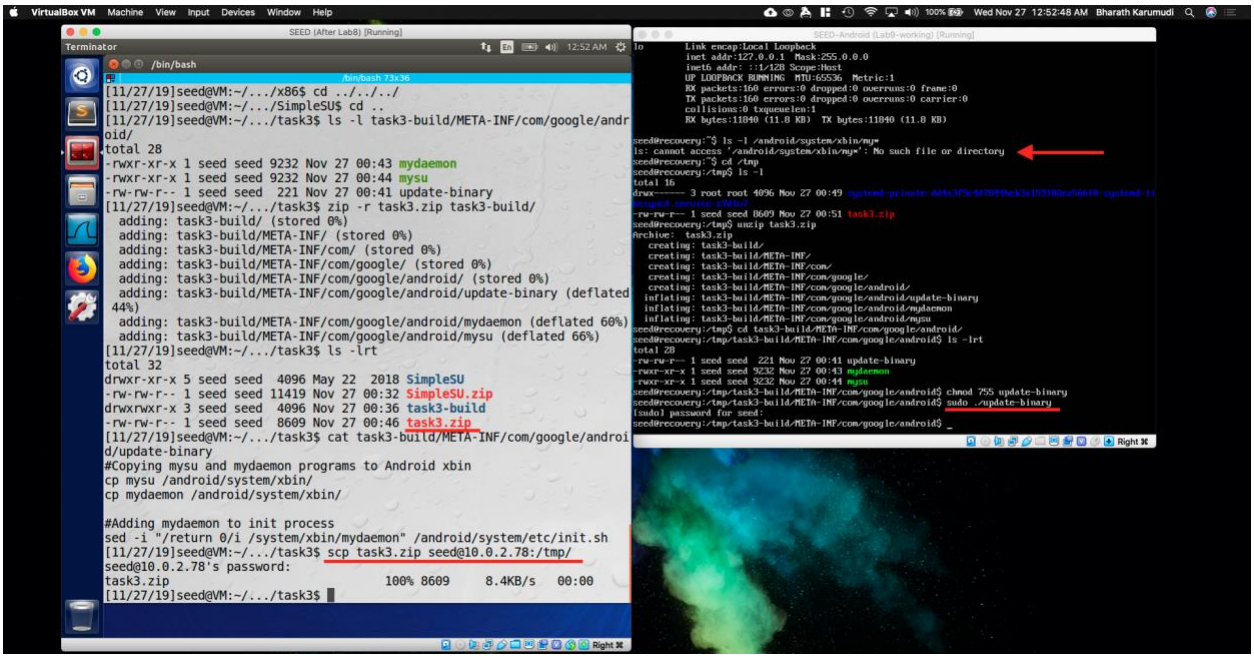


Fig: scp the OTA package to Recovery OS

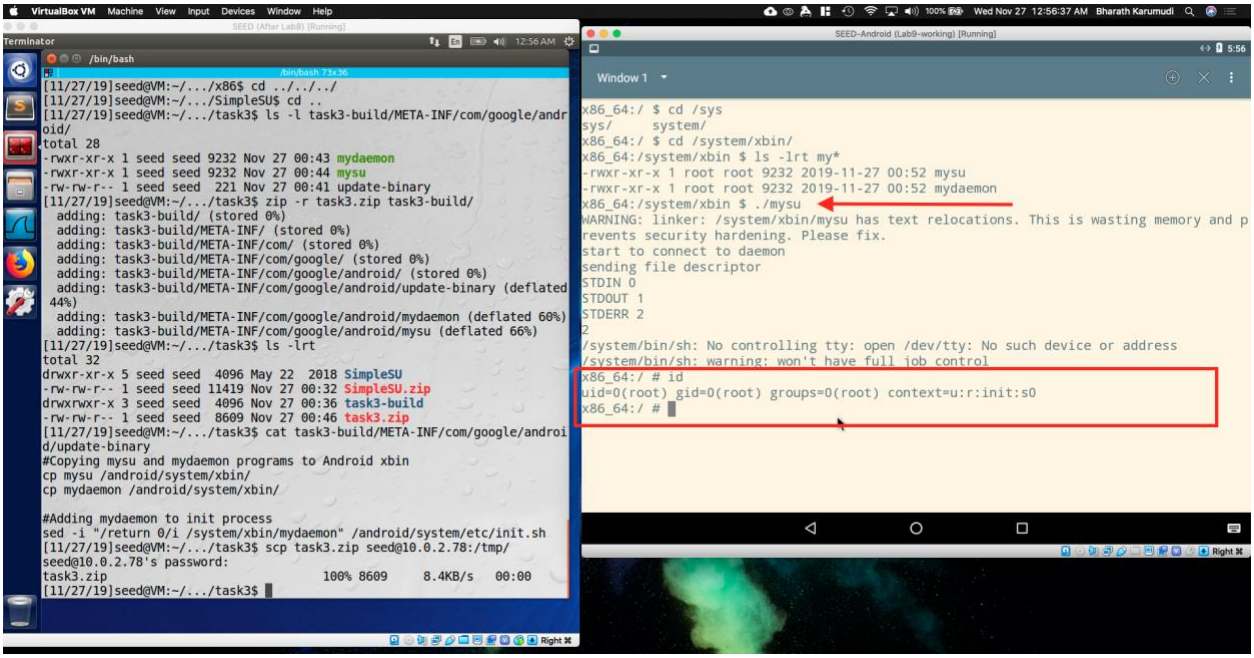


Fig: Executing the mysu program in Android Terminal Emulator

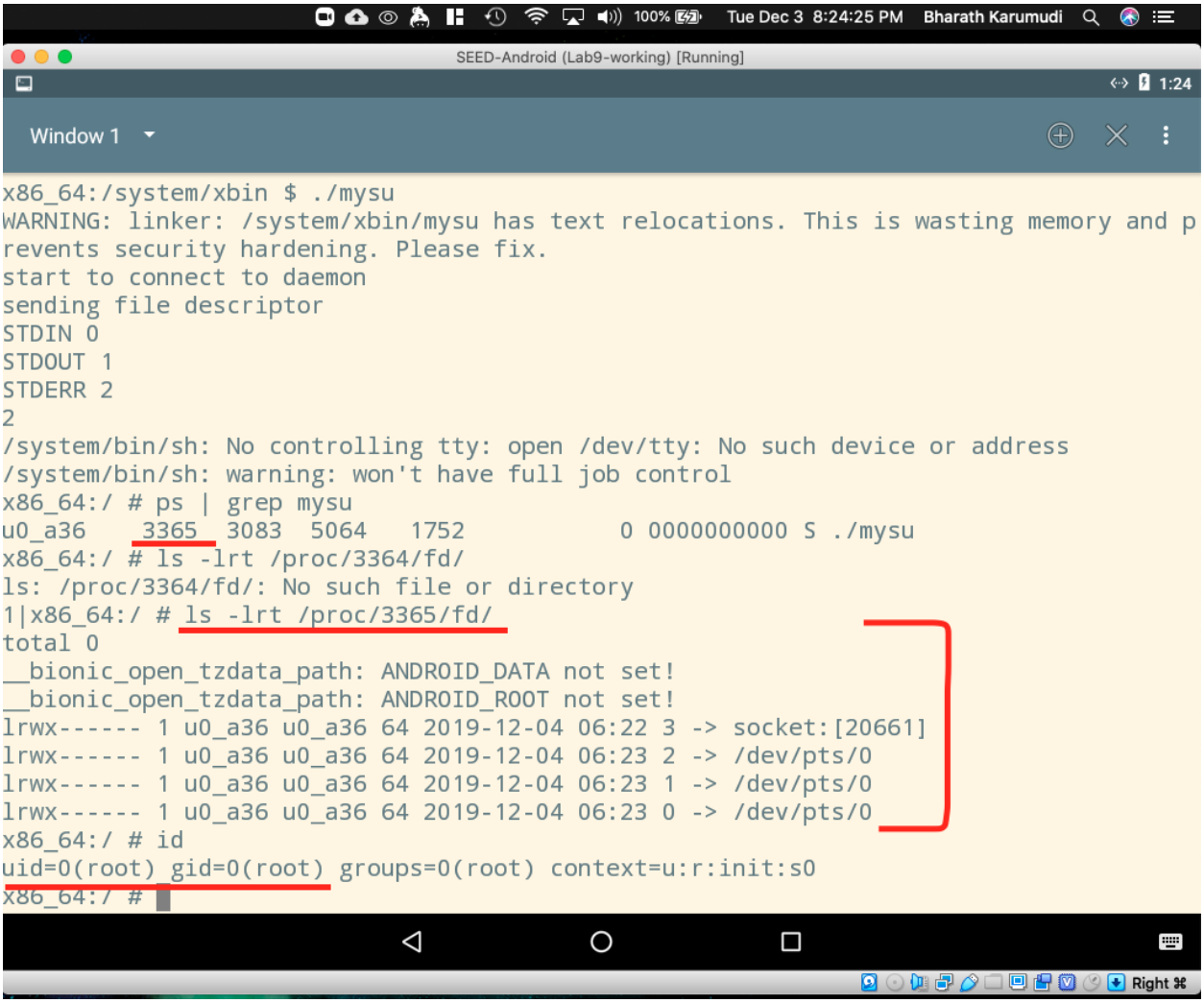


Fig: Showing the mysu file descriptors - /proc/<pid>/fd

Observation: Using the SimpleSU.zip file provided in the lab, compiled all the files and built the OTA package for the Android. Once the update is executed in the recovery OS and booted to Android and in the Terminal Emulator, when typed mysu which is under /system/xbin ; got the root shell.

Explanation:

1. Unzipped the SimpleSU.zip and executed the compile_all.sh script which compiled the programs to Android Native.
2. Created the Android OTA structure “task3/task3-build/META-INF/com/google/android”
3. Copied the mydaemon and mysu to the android directory
4. Created a update-binary script under android directory, which copied daemon and mysu to “/android/system/xbin/” directory.
5. Zipped the package and scp to Android Recovery OS under /tmp/task3.zip
6. Inside the Android recovery OS, unzipped the file and executed the update-binary script.

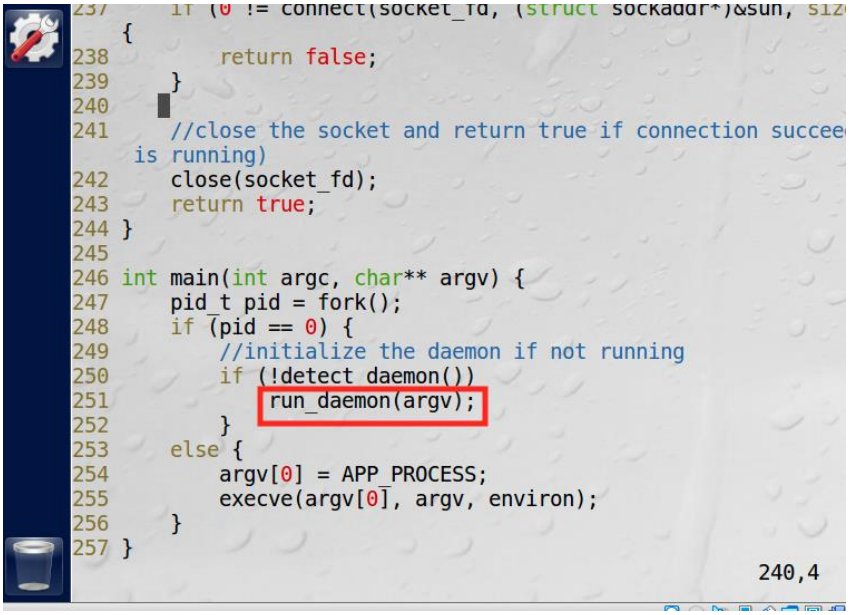
7. Booted back to Android and with Terminal Emulator App, navigated to /system/xbin and when executed the mysu, got the root shell and verified with id command.

We started a root daemon and the client program sends the request to root daemon. The root daemon starts the shell process and return it to the client which has root privileges. But when the shell is created, it inherits the stdin and stdout from its parent, who is root. But a regular user owned client cannot control that. So, we give our client program input and output to the shell process devices. By this, the user will have the control on the shell process.

Thus, rooted the Android OS successfully.

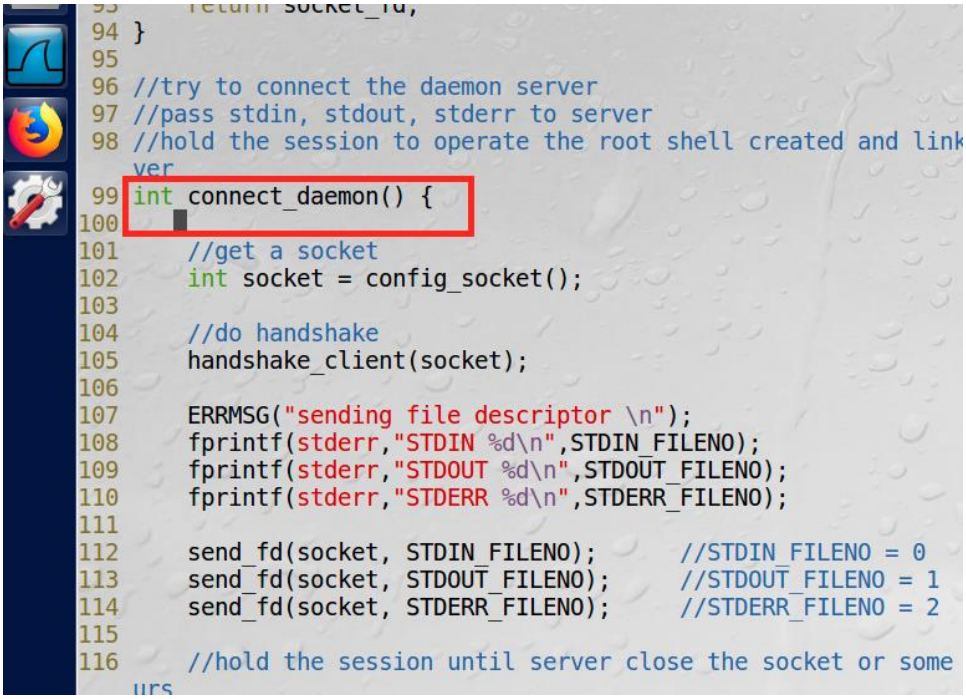
Questions:

1. Server launches the original app process binary



File Name: mydaemonsu.c
Function: main()
Line: 251.

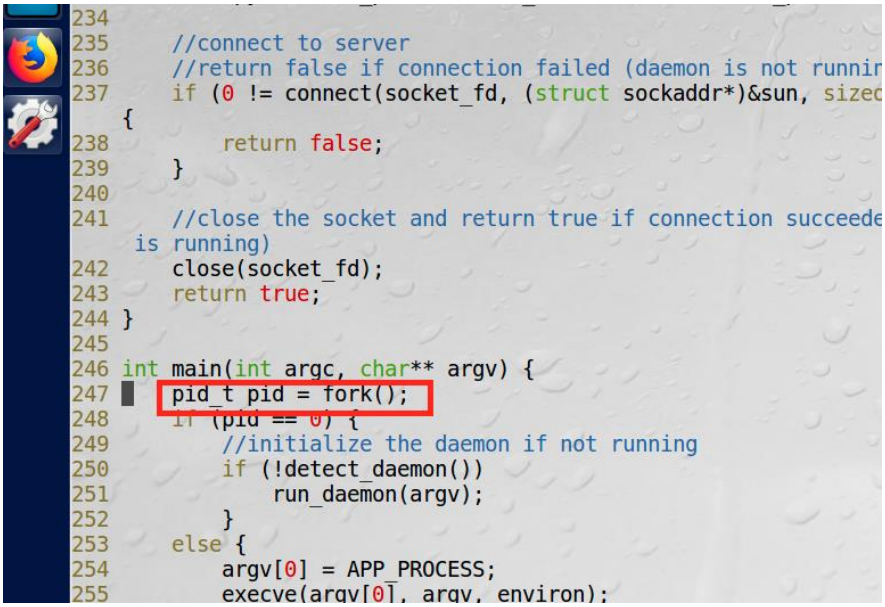
2. Client sends its FDs



```
 94 }
 95
 96 //try to connect the daemon server
 97 //pass stdin, stdout, stderr to server
 98 //hold the session to operate the root shell created and link
    ver
 99 int connect_daemon() {
100
101     //get a socket
102     int socket = config_socket();
103
104     //do handshake
105     handshake_client(socket);
106
107     ERRMSG("sending file descriptor \n");
108     fprintf(stderr,"STDIN %d\n",STDIN_FILENO);
109     fprintf(stderr,"STDOUT %d\n",STDOUT_FILENO);
110     fprintf(stderr,"STDERR %d\n",STDERR_FILENO);
111
112     send_fd(socket, STDIN_FILENO);      //STDIN_FILENO = 0
113     send_fd(socket, STDOUT_FILENO);     //STDOUT_FILENO = 1
114     send_fd(socket, STDERR_FILENO);     //STDERR_FILENO = 2
115
116     //hold the session until server close the socket or some
    urs
```

File Name: mysu.c
Function: connect_daemon()
Line: 112.

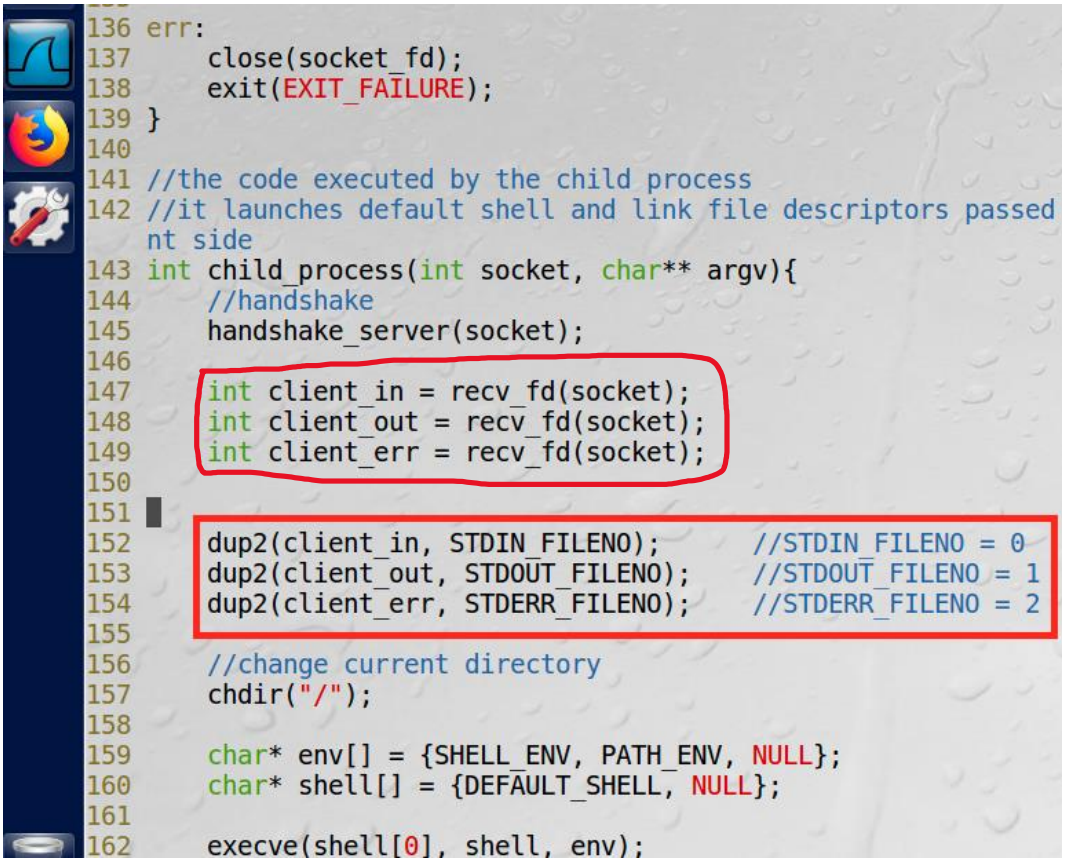
3. Server forks to a child process



```
234
235     //connect to server
236     //return false if connection failed (daemon is not runnir
237     if (0 != connect(socket_fd, (struct sockaddr*)&sun, sizec
    {
238         return false;
239     }
240
241     //close the socket and return true if connection succeede
    is running)
242     close(socket_fd);
243     return true;
244 }
245
246 int main(int argc, char** argv) {
247     pid_t pid = fork();
248     if (pid == 0) {
249         //initialize the daemon if not running
250         if (!detect_daemon())
251             run_daemon(argv);
252         }
253     else {
254         argv[0] = APP_PROCESS;
255         execve(argv[0], argv, environ);
```

File Name: mydaemonsu.c
Function: main()
Line: 247

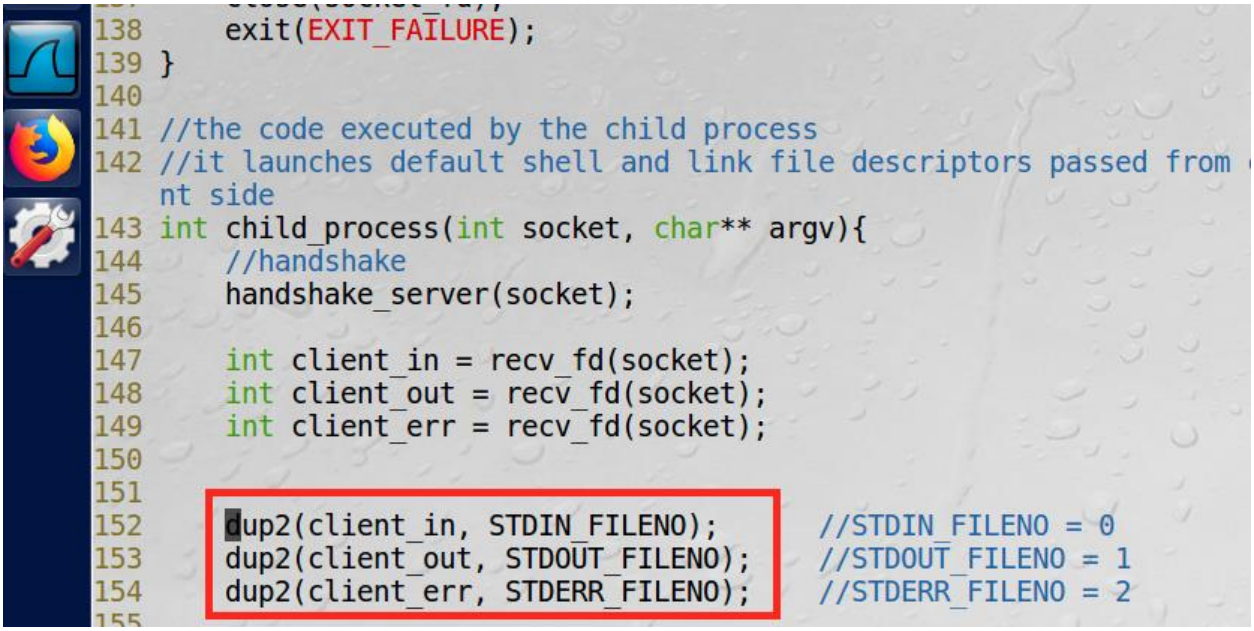
4. Child process receives client's FDs



```
136 err:
137     close(socket_fd);
138     exit(EXIT_FAILURE);
139 }
140
141 //the code executed by the child process
142 //it launches default shell and link file descriptors passed
    nt side
143 int child_process(int socket, char** argv){
144     //handshake
145     handshake_server(socket);
146
147     int client_in = recv_fd(socket);
148     int client_out = recv_fd(socket);
149     int client_err = recv_fd(socket);
150
151
152     dup2(client_in, STDIN_FILENO);      //STDIN_FILENO = 0
153     dup2(client_out, STDOUT_FILENO);    //STDOUT_FILENO = 1
154     dup2(client_err, STDERR_FILENO);    //STDERR_FILENO = 2
155
156     //change current directory
157     chdir("/");
158
159     char* env[] = {SHELL_ENV, PATH_ENV, NULL};
160     char* shell[] = {DEFAULT_SHELL, NULL};
161
162     execve(shell[0], shell, env);
```

File Name: mydaemonsu.c
Function: child_process()
Line: 147

5. Child process redirects its standard I/O FDs



```
138     exit(EXIT_FAILURE);
139 }
140
141 //the code executed by the child process
142 //it launches default shell and link file descriptors passed from
    nt side
143 int child_process(int socket, char** argv){
144     //handshake
145     handshake_server(socket);
146
147     int client_in = recv_fd(socket);
148     int client_out = recv_fd(socket);
149     int client_err = recv_fd(socket);
150
151
152     dup2(client_in, STDIN_FILENO);      //STDIN_FILENO = 0
153     dup2(client_out, STDOUT_FILENO);    //STDOUT_FILENO = 1
154     dup2(client_err, STDERR_FILENO);    //STDERR_FILENO = 2
```

File Name: mydaemonsu.c
Function: child_process()
Line: 152

6. Child process launches a root shell

```
130     close(socket);
131
132     //print out error message if has
133     if (flag < 0) {
134         ERRMSG("Socket failed on client: ");
135         ERRMSG(strerror(errno));
136         ERRMSG("\n");
137         return (EXIT_FAILURE);
138     }
139
140     return (EXIT_SUCCESS);
141 }
142
143 int main(int argc, char** argv) {
144     //if not root
145     //connect to root daemon for root shell
146     if (getuid() != 0 && getgid() != 0) {
147         ERRMSG("start to connect to daemon \n");
148
149         return connect_daemon();
150     }
151     //if root
152     //launch default shell directly
153     char* shell[] = {"/system/bin/sh", NULL};
154     execve(shell[0], shell, NULL);
155     return (EXIT_SUCCESS);
156 }
157
```

File Name: mysu.c
Function: main()
Line: 149.