Lab 9
A20453991

Lab Task 1: Build a simple OTA package

Created a dummy.sh file that contains the shown command. This is placed in the android folder

```
varungunda@VarunPC:~/Documents/VarunIllinoisTech/Spring 2020/System and Network Security/Lab 8/ota/META-INF/com/goog
le/android$ cat dummy.sh
echo hello > /system/dummy
```

update-binary script is as shown below. It copies dummy.sh to android/system/xbin, changes it mode and we use sed to change the “return 0” in init.sh to system/xbin/dummy.sh so that dummy.sh runs at the end of init.sh with the root privilege.

update-binary × dummy.sh

ome ▸ varungunda ▸ Documents ▸ VarunIllinoisTech ▸ Spring 2020 ▸ System and Network Security ▸ Lab 8 ▸ ota ▸ META-INF ▸ com ▸ google

```
1  cp ./dummy.sh /android/system/xbin
2  chmod a+x /android/system/xbin/dummmy.sh
3  sed -i "/return 0/i/system/xbin/dummy.sh" /android/system/etc/init.sh
```

Now building the ota pacakge as shown below using zip command. Unzipped the package to show the package contents

```
varungunda@VarunPC:~/Documents/VarunIllinoisTech/Spring 2020/System and Network Security/Lab 8/ota$ unzip -l my_ota.
zip
Archive:  my_ota.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  2020-03-04 17:39   META-INF/
        0  2020-03-04 17:39   META-INF/com/
        0  2020-03-04 17:39   META-INF/com/google/
        0  2020-03-04 17:42   META-INF/com/google/android/
      138  2020-03-04 17:48   META-INF/com/google/android/update-binary
       27  2020-03-04 17:42   META-INF/com/google/android/dummy.sh
---------                     -------
      165                     6 files
```

Now, we find the ip address of the Android VM from recovery OS to transfer this package to that machine

```
root@recovery:~# ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:49:ca:9c
          inet addr:192.168.43.222  Bcast:192.168.43.255  Mask:255.255.255.0
          inet6 addr: 2600:380:644c:bde6:a00:27ff:fe49:ca9c/64 Scope:Global
          inet6 addr: fe80::a00:27ff:fe49:ca9c/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:15 errors:0 dropped:0 overruns:0 frame:0
          TX packets:21 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2068 (2.0 KB)  TX bytes:2504 (2.5 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:160 errors:0 dropped:0 overruns:0 frame:0
          TX packets:160 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:11840 (11.8 KB)  TX bytes:11840 (11.8 KB)
```

```
varungunda@VarunPC:~/Documents/VarunIllinoisTech/Spring 2020/System and Network Security/Lab 8/ota$ ping 192.168.43.
222
PING 192.168.43.222 (192.168.43.222) 56(84) bytes of data.
64 bytes from 192.168.43.222: icmp_seq=1 ttl=64 time=0.647 ms
64 bytes from 192.168.43.222: icmp_seq=2 ttl=64 time=0.367 ms
64 bytes from 192.168.43.222: icmp_seq=3 ttl=64 time=0.312 ms
```

```
varungunda@VarunPC:~/Documents/VarunIllinoisTech/Spring 2020/System and Network Security/Lab 8/ota$ scp my_ota.zip
seed@192.168.43.222:/tmp
The authenticity of host '192.168.43.222 (192.168.43.222)' 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 '192.168.43.222' (ECDSA) to the list of known hosts.
seed@192.168.43.222's password:
```

Unzipping the package in the recovery OS and running update-binary script as shown below

```
root@recovery:/tmp# unzip my_ota.zip
Archive:  my_ota.zip
   creating: META-INF/
   creating: META-INF/com/
   creating: META-INF/com/google/
   creating: META-INF/com/google/android/
  inflating: META-INF/com/google/android/update-binary
 extracting: META-INF/com/google/android/dummy.sh
```

```
root@recovery:/tmp/META-INF/com/google/android# cat update-binary
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/init.sh
root@recovery:/tmp/META-INF/com/google/android# ./update-binary
root@recovery:/tmp/META-INF/com/google/android#
```

Now on logging into Android VM, we can see the dummy file created in /system folder.

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

Task 2: Inject code via app process

Modified the Android.mk file to include file name and module name.

b4rnd01r.c × Application.mk ×

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

b4rnd01r.c × Android.mk × Applic

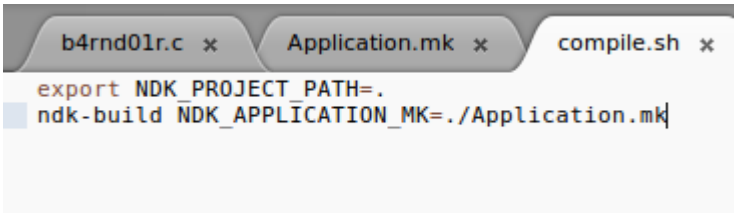
```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := my_app_process
LOCAL_SRC_FILES := my_app_process.c
include $(BUILD_EXECUTABLE)
```

my_app_process.c is copied as shown here.

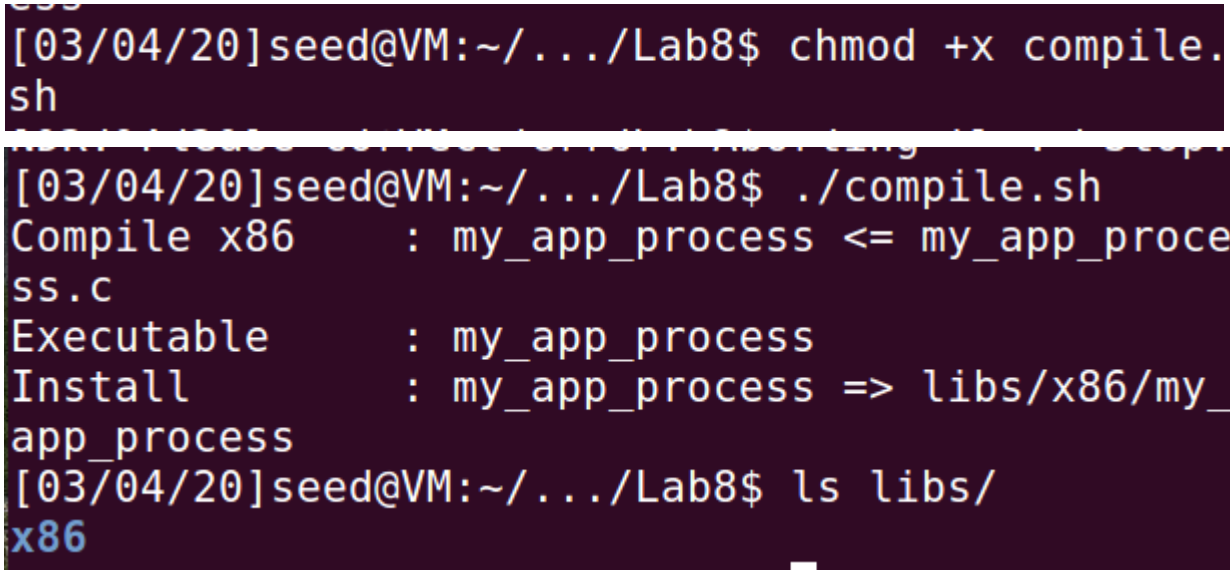
b4rnd01r.c × Android.mk × my_app_process.c ×

```
#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;
}
```

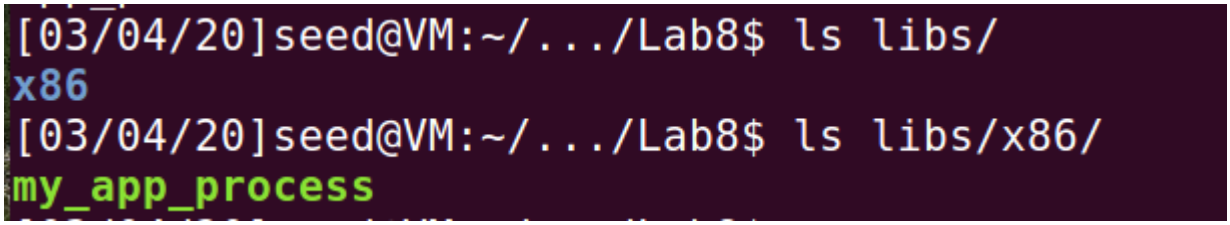
Creating compile.sh file to compile using NDK.



Running compile.sh script as shown below:



This creates a /libs directory in the current folder and its contents are as shown below:



```
[03/04/20]seed@VM:~/.../Lab8$ mkdir task2
[03/04/20]seed@VM:~/.../Lab8$ mv META-INF/ task
2/
[03/04/20]seed@VM:~/.../Lab8$ cd task2/
[03/04/20]seed@VM:~/.../task2$ cd ../
[03/04/20]seed@VM:~/.../Lab8$ mv libs/x86/my_ap
p_process task2/META-INF/com/google/android/
```

Creating a new directory task2 and copying all the contents to it and zipping it as shown below.

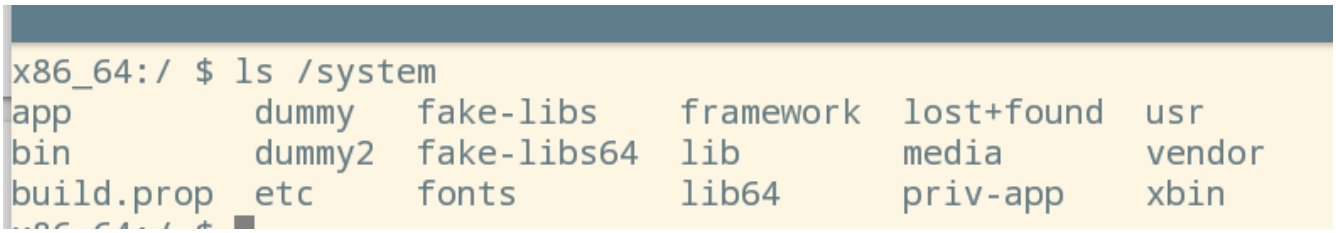
```
[03/04/20]seed@VM:~/.../Lab8$ zip -r task2.zip
task2/
  adding: task2/ (stored 0%)
  adding: task2/META-INF/ (stored 0%)
  adding: task2/META-INF/com/ (stored 0%)
  adding: task2/META-INF/com/google/ (stored 0%
)
  adding: task2/META-INF/com/google/android/ (s
tored 0%)
  adding: task2/META-INF/com/google/android/my_
app_process (deflated 72%)
```

Transferred this package to Android VM and unzipping it on recovery OS as shown below.

```
root@recovery:/tmp# unzip task2_1.zip
Archive:  task2_1.zip
   creating: task2/
   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/update-binary-1
  inflating: task2/META-INF/com/google/android/my_app_process
```

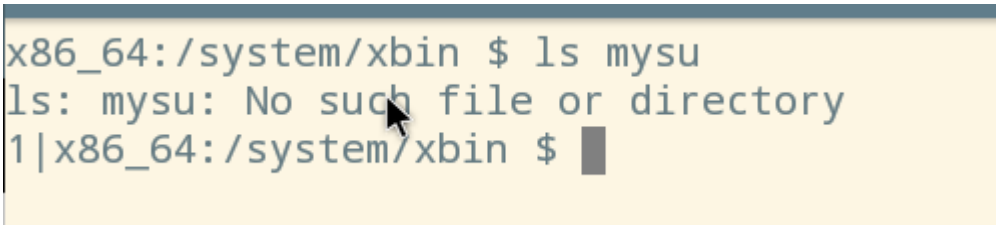
```
root@recovery:/tmp/task2/META-INF/com/google/android# ls
my_app_process  update-binary  update-binary-1
root@recovery:/tmp/task2/META-INF/com/google/android# chmod +x update-binary
root@recovery:/tmp/task2/META-INF/com/google/android# ./update-binary
```

On running update binary as shown above and then starting Android VM, we can see dummy2 file getting created in /system folder



Task 3: Implement SimpleSU for Getting Root Shell

As shown here, initially there is no mysu file in /*system/*xbin directory.



Downloaded SimpleSU code and its contents are as shown below. Compiled the code as shown below.

```
[03/05/20]seed@VM:~/.../task3$ ls
SimpleSU
[03/05/20]seed@VM:~/.../task3$ cd SimpleSU/
[03/05/20]seed@VM:~/.../SimpleSU$ ls
compile_all.sh  mysu          socket_util
mydaemon        server_loc.h
[03/05/20]seed@VM:~/.../SimpleSU$ chmod +x comp
ile_all.sh
[03/05/20]seed@VM:~/.../SimpleSU$ ./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/////////////
```

```
[03/05/20]seed@VM:~/.../t3$ cp ../task3/SimpleS
U/mydaemon/libs/x86/mydaemon ./x86/
[03/05/20]seed@VM:~/.../t3$ cp ../task3/SimpleS
U/mysu/libs/x86/mysu ./x86/
[03/05/20]seed@VM:~/.../t3$ ls
META-INF  x86
```

Now we moved the files mysu, mydaemon and update-binary to /android folder and zipped it to create the package. Then transferred this package to recovery OS as shown below.

```
[03/05/20]seed@VM:~/.../Lab8$ zip -r t3.zip t3
  adding: t3/ (stored 0%)
  adding: t3/META-INF/ (stored 0%)
  adding: t3/META-INF/com/ (stored 0%)
  adding: t3/META-INF/com/google/ (stored 0%)
  adding: t3/META-INF/com/google/android/ (stor
ed 0%)
  adding: t3/META-INF/com/google/android/update
-binary (deflated 41%)
  adding: t3/META-INF/com/google/android/mydaem
on (deflated 60%)
  adding: t3/META-INF/com/google/android/mysu (
deflated 66%)
```

```
root@recovery:~# ls /tmp
systemd-private-a17057668c3049feb3f8ac1aef5345f2-systemd-timesyncd.service-cpMWVa  t3.zip
root@recovery:~# cd /tmp
root@recovery:/tmp# unzip t3.zip
Archive:  t3.zip
   creating: t3/
   creating: t3/META-INF/
   creating: t3/META-INF/com/
   creating: t3/META-INF/com/google/
   creating: t3/META-INF/com/google/android/
  inflating: t3/META-INF/com/google/android/update-binary
  inflating: t3/META-INF/com/google/android/mydaemon
  inflating: t3/META-INF/com/google/android/mysu
root@recovery:/tmp# cd t3/META-INF/com/google/android/
root@recovery:/tmp/t3/META-INF/com/google/android# chmod +x update-binary
root@recovery:/tmp/t3/META-INF/com/google/android# ./update-binary
```

Unzipped the package and ran the update-binary. Now on starting android, we can see mysy and mydaemon files in /system /xbin directory

```
x86_64:/system/xbin $ ls my
mydaemon  mysu
```

On running mysu, we can see that we get root privilege.

```
x86_64:/system/xbin $ ./mysu
WARNING: linker: /system/xbin/mysu has text relocations. This is wasting memory and
revents security hardening. Please fix.
start to connect to daemon
sending file descriptor
STDIN 0
STDOUT 1
STDERR 2
2
/system/bin/sh: No controlling tty: open /dev/tty: No such device or address
/system/bin/sh: warning: won't have full job control
x86_64:/ # id
uid=0(root) gid=0(root) groups=0(root) context=u:r:init:s0
```

Q. Server launches the original app process binary

```
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);
    }
}
```

As shown the above execve line does this.

Q.Client sends its Fds

```
int connect_daemon() {
    int connect_daemon()
    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

    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
```

In connect_daemon function of mysu.c, we can see above client sending fds in send_fd lines.

Q. 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);
    }
}
```

We can see fork command above in main function of mydaemonsu.c

Q. Child process receives client's Fds

```
//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("/");
```

We can see in child_process function of mydaemonsu.c, client receivinng the fds in lines recv_fd commands above

Q. Child process redirects its standard I/O Fds

```
//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("/");
```

We can see in child_process function of mydaemonsu.c file in dup2 commands above.

Q. Child process launches a root shell

```c
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);
}
```

As we can see in mysu.c main function, the above execve is launching the root shell.