MONO: A MONITORING SYSTEM FOR MOBILE NETWORK FORENSICS

A Master’s Thesis
Submitted to the Faculty of the
Escola Tècnica d’Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
by
Guillaume CANAL

In partial fulfillment
of the requirements for the double degree of
TELECOMMUNICATIONS ENGINEERING
IMT Atlantique (Télécom-Bretagne) / UPC (ETSETB)

Advisor: Juan HERNÁNDEZ SERRANO

Barcelona July 2017
Title of the thesis: MONO: A MONITORING SYSTEM FOR MOBILE NETWORK FORENSICS

Author: Guillaume CANAL

Advisor: Juan HERNÁNDEZ SERRANO

Abstract:

We present in this report a study and implementation of a monitoring system for Android applications.

We have first analyzed the requirements of the system; then, we have defined a set of specifications; and, finally, we have proposed a technical solution.

The proposed system is called Mono and is designed to capture traffic and retrieve information of interest for a forensics analyst about a given application (e.g. files accessed, etc.). It is composed of three elements:

- an Android application implementing a VPN client
- a VPN server with a public IP address that can capture, decrypt (under certain conditions) and store packets. It also runs a web server
- a web interface to examine in real-time the traffic of the VPN tunnel.

This first version of the Mono system, is available on GitHub under the MIT license (https://github.com/gcanal/Mono)

Figure 1: VPN-based web packet capturer
Acknowledgements

Gracias a mi tutor Juan Hernández Serrano, por sus consejos sabios, su apoyo moral y su ayuda sobre todo a la hora de corregir los capítulos de esta tesis. Pero no le echen la culpa cuando encuentren un error, ya que me encargué personalmente que este documento so sea perfecto.

Y puesto que uno no tiene muchas veces la ocasión de escribir una tesis de master en su vida, aprovecho la ocasión para saludar y dar las gracias a mis compañeros mexicanos Luis, Sinaí, Daniel, Julian, Osamu San, Pameli y Brian

Je remercie également ces personnes qui m’ont directement aidé : ma Maman, mon Frère, Anne-Marie et Patrick.
## Revision History and Approval Record

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9 June 2017</td>
<td>Document Creation</td>
</tr>
<tr>
<td>1</td>
<td>27 June 2017</td>
<td>Document Revision</td>
</tr>
</tbody>
</table>

**Written by:**

<table>
<thead>
<tr>
<th>Date</th>
<th>Name</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 June 2017</td>
<td>Guillaume CANAL</td>
<td>Project Author</td>
</tr>
</tbody>
</table>

**Reviewed and approved by:**

<table>
<thead>
<tr>
<th>Date</th>
<th>Name</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>27 June 2017</td>
<td>Juan HERNÁNDEZ SERRANO</td>
<td>Project Supervisor</td>
</tr>
</tbody>
</table>
# Contents

1 Introduction 15

2 State of the Art 17
   2.1 Analysis of Android Packet Capturers (Summary) 17
   2.2 Analysis of Android Packet Capturers (Details) 18
   2.3 Techniques of Packet Capturing 22

3 Technical Solution 25
   3.1 Technical Solution 1 25
   3.2 Technical Solution 2 26
   3.3 Comparison of the solutions 27

4 Android Application 29
   4.1 Presentation of OpenVPN for Android 29
   4.2 Setup of the VPN Tunnel 29

5 Monitoring Server 33
   5.1 Technology Analysis 33
   5.2 Packet Capturer 33
   5.3 SSL/TLS Traffic Decoder 37

6 Web Application 41
   6.1 Presentation 41
   6.2 Angular JS 42
   6.3 Application Architecture 43
   6.4 Dataflow 46
   6.5 Packet Display 49

7 Security Concerns 53
   7.1 Security Issues 53
   7.2 Protecting Against Structured Query Language (SQL) Injection 53

8 Conclusion and Future Developments 57

Bibliography 61

Glossary 65
# List of Figures

1. VPN-based web packet capturer .................................................. 3
2.1 Classical use of a VPN service (ref. [31]) .................................. 23
2.2 OpenVPN tun interface (from ref. [11]) ....................................... 23
3.1 Technical solution 1 - Use case 1 ............................................. 26
3.2 Technical solution 1 - Use case 2 ............................................. 26
3.3 Technical solution 2 ................................................................. 27
5.1 GUI of the IPV4 conversations .................................................. 36
5.2 How mitmproxy works (from http://docs.mitmproxy.org [1]) .......... 38
6.1 Web user interface ................................................................. 42
6.2 Data Flow of our client-side application ..................................... 44
6.3 Session information ............................................................... 46
6.4 Change session information ..................................................... 46
6.5 GUI of the Datatable JS library .................................................. 51
List of Tables

2.1 Packet capture applications for Android ........................................... 18
3.1 Packet capture applications for Android ........................................... 27
8.1 Mono application vs Project requirements ........................................ 58
Chapter 1

Introduction

Digital Forensics can be defined as the use of scientific methods to find evidences in digital sources such as computers, tablets, mobile phones, databases, Network Interface Controllers (NICs), smart devices, etc. These evidences may be used to “support or refute a hypothesis”[27] in a public or a private investigation (about criminal activities, intrusions, etc.).

Network forensics is a “sub-branch of digital forensics relating to the monitoring and analysis of computer networks” [29].

ANFORA (ANálisis FORense Avanzado) is a Spanish research project conducted by the Information Security Group of the UPC that is aimed at the innovation in digital forensics. Among its fields of research is the creation and improvement of tools and techniques to ease the work of analysts in digital and network forensics.

In the context of this research project, we present this work that addresses the needs of automation and better user experience in network forensics analysis.

The goal of this work is to create a monitoring system for mobile applications; a tool to assist forensics analysts in detecting and analyzing data connections performed from/to a mobile device. We are specifically interested in:

- data flows from/to the different mobile applications,
- files that have been accessed (in read or write mode) during the session.

The proposed monitoring system should be started and stopped by the so-called operator. The period during which the system remains active is called a session.

In the following, we define the desired functionalities for the system. A priority is assigned to each functionality, being $P_1$ the highest priority.

- ($P_1$) List IP packets with their content.
- ($P_2$) Download selected packets for further analysis with Wireshark.
- ($P_3$) List IP, UDP and TCP conversations.
- ($P_4$) Enable search by keyword in packet header and payload.
- ($P_5$) Decrypt SSL/TLS traffic, whenever possible.
- ($P_6$) List information of each IP address, the mobile device connects to (name resolution, geographic location, appearances in abuse databases, ...)
- ($P_7$) List files accessed (read and write mode) during the session by each application (only possible with rooted/jailbroken device).

There are however some constraints that have been identified:
• (C\textsubscript{1}) **External storage**: The recorded information must not be stored on the device itself, but on another one with enough free disk space.

• (C\textsubscript{2}) **Real time**: The information (e.g. packets) must be made available while recording. The user must be able to perform live operations (inspect packets, see current connections, search in payloads)

• (C\textsubscript{3}) **Support for non-rooted or jailbroken devices**: The functionalities should work, to the extent possible, on non-rooted (Android) or non-jailbroken (iOS) devices.

• (C\textsubscript{4}) **Forensics friendly**: The interface should be intuitive and easy to use.

• (C\textsubscript{5}) **Android approach**: Due to time constraints, this project is only initially developed for Android devices.

The rest of this document will be structured as follows.

After carrying out research on the current **state-of-the-art** technologies in chapter 2, we will describe the process of elaborating a **technical solution** in chapter 3. In this chapter, we will establish the architecture of the Mono system based on the list of constraints. A VPN

Following chapters 4, 5 and 6 will explain how we implemented a first version of the Mono system with the most desired features, each chapter corresponding to a given component of this system.

Chapter 4 will be devoted to the **Android device**. We will see how to set up a VPN tunnel by using the application *OpenVPN for Android*.

In chapter 5, we will expose the two main components of the **monitoring server**: a packet capturer working with the scapy library coded in python 3 [16] and a traffic decoder using the mitmproxy scripting Application Programming Interface (API) also coded in python 3 [23].

Chapter 6 is the most substantial of this report and details the Mono user interface: a **Web application** programmed with the Angular JS JavaScript framework. We will introduce this framework and explain the software architecture of this web application.

In chapter 7, we will evoke security threats and their possible countermeasures.

Finally, in chapter 8 we will present the conclusions of this work and the potential future work.
Chapter 2

State of the Art

The objective of this section is to evaluate if there are existing software that fulfill the specifications of our system, to get insights on the technology they used and, if possible, to reuse their code in our application (provided that the code is open-source).

Our core functionality being the acquisition of packets sent and received by the device, we are specifically interested in open-source Android packet capturers. In section 2.1, we will assess if they respect the constraints specified in the project requirements.

Section 2.2, will provide more details about each software.

Finally, in section 2.3, we will describe the two main methods to capture packets that we found during our investigation.

2.1 Analysis of Android Packet Capturers (Summary)

2.1.1 Respect of Constraints

During the analysis, special care must be taken to ensure that the constraints we were given (besides $C_5$) are respected.

- ($C_1$) **External storage**: This constraint is not respected if the capture is executed directly on the phone. Even if there is an export function to transfer the data on another device, the program might run out of memory during long captures.

- ($C_2$) **Real time**: The program has to allow live packet inspection

- ($C_3$) **Support for non rooted Android devices**: A maximum number of functionalities must work on non rooted devices. For instance, the program must be able to capture all traffic (Wi-Fi and 3G/4G) without needing to root the analyzed device.

- ($C_4$) **Forensics friendly**: The level of detail required by a network forensics analyst require a user-friendly and agile interface which is hard to achieve on a mobile phone. Therefore, we will give priority to systems that allow to read/inspect the results of the device analysis directly from a desktop PC or a laptop.
2.1.2 Summary Table

<table>
<thead>
<tr>
<th>Packet Capture Application</th>
<th>External Storage</th>
<th>Real Time</th>
<th>Non Root</th>
<th>Forensics Friendly</th>
<th>Last Update</th>
<th>Open Source</th>
<th>Uses libpcap</th>
<th>Uses VPN</th>
<th>Android Version</th>
<th>Exportable pcap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android tcpdump [46]</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>18/01/2017</td>
<td>✓</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>SSL Packet Capture [44]</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>19/10/2015</td>
<td>Yes</td>
<td>4.0+</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>tPacketCapture [45]</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>19/10/2015</td>
<td>Yes</td>
<td>4.0+</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SimplePacketCapture [40]</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>29/12/2013</td>
<td>Yes</td>
<td>2.1+</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Android Packet Capture [47]</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>03/07/2016</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>Wicap Sniffer [38]</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>29/07/2016</td>
<td>Yes</td>
<td>4.1+</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Packet Sniffer [37]</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>19/10/2015</td>
<td>Yes</td>
<td>1.6+</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>bitShark [39]</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>19/10/2015</td>
<td>Yes</td>
<td>2.3+</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 2.1: Packet capture applications for Android

Table 2.1 summarizes the characteristics of packet capturers in Android, where we can see two main trends in the technology used:

1. Use a rooted device with the libcap library or the tcpdump binary (which is a compiled version of the libpcap library).
2. Use a VPN on a non-rooted device to capture the packets from all the applications.

Both techniques will be described with more details in section 2.3. The latter one seems the most interesting to us, as it also works with non-rooted devices. Unfortunately, none of the applications using this alternative is open-source, so they could not be forked to develop our system. However, they have been a source of inspiration in order to design the architecture of our project (see section 3.2).

2.2 Analysis of Android Packet Capturers (Details)

In this section, we will present the packet capture applications available for Android as of February 2017. The reader can skip this part without risk of misunderstanding the project as a whole.

2.2.1 Android tcpdump

- Version: 4.9.0 / compiled with libpcap 1.8.1 (January 18, 2017)
- Open source: YES (BSD License)
- Needs root access: YES
- Needs additional Hardware: NO

Android tcpdump is a command line packet capture utility. It is open-source and is extremely similar to unix’s tcpdump application. According to the documentation [46], it has to be run on a terminal:
• either with a terminal emulator application for Android,

• or with the Android Debug Bridge (ADB), via the adb shell command, giving you access to a shell on your Android device from your PC.

Installation of the Android Native Development Kit (NDK) is needed.

2.2.2 SSL Packet Capture

• Version: 1.2.3 (19th October, 2015)

• By: Grey Shirts (greyduchess@gmail.com)


• Open source: NO

• Needs root access: NO

• Presentation video https://www.youtube.com/watch?v=Q-60uuIHI2k

• Android version: 4.0+

The application uses the VPN service added in the API with Android 4.0. As of June 2017, 98.4 % of Android devices uses a version above 4.0 [7]. It is one of the most used packet analyzer for Android because it is free (but not open-source) and does not need a rooted device. It also allows to decrypt some SSL/TLS traffic by performing a simple man in the middle attack. SSL Packet Capture intercepts TLS traffic and generates on the fly certificates with its own public/private key pair firmed by its own authority. Obviously, for the certificates to be accepted by the Android apps, the certificate of the software’s certification authority should be in the Android device. In such a case, the SSL/TLS packets sent by the applications can be decrypted by a man-in-the-middle proxy.

2.2.3 tPacketCapture

• Version: v2.0.1 (5th February, 2015)

• By:taosoftware


• Needs root access: NO

• Open source: NO

• presentation video: https://www.youtube.com/watch?v=PjJbfYfDsX4

• Android version: 4.0+

tPacketCapture also uses the VpnService class provided by Android OS. Captured data are saved as a PCAP file format in the external storage and can then analyzed with a PCAP reader (such as Wireshark). A pro version is also available (9.49 €) and includes some advanced features (such as filtering packets by applications).
2.2.4 SimplePacketCapture

- Version: 0.0.7 (29th December, 2013)
- By: JOJOAGOGOGO (jojoa555@gmail.com)
- Needs root access: YES
- Open source: NO
- Android version: 2.1+

This application captures the packets by directly listening to the network interface of the device. To do this, root privileges are needed. The traffic can be saved to a PCAP file compatible with Wireshark. The log file can be sent by mail.

2.2.5 Android Packet Capture

- Version: xxx
- Last commit: 3 August, 2016
- By: yadavdev
- Google play: NO
- Needs root access: YES
- Open source: YES
- GitHub: https://github.com/yadavdev/android-packet-capture [47]

Captures packet in an Android device and save them in a PCAP file (compatible with Wireshark). It uses a precompiled tcpdump binary. It uses libsuperuser for starting an interactive root shell from where the tcpdump binary runs.

2.2.6 Wicap. Sniffer

- Version: 1.9.1 (29, July 2016)
- By: evbadroid evbadroid@gmail.com (Russian Federation)
- Google play: https://play.google.com/store/apps/details?id=com.evbadroid.wicap [38]
- Needs root access: YES
- Open source: NO
- Android version: 4.1+
- Presentation video: https://youtu.be/GTSWUnt-6e0

This application is also a packet capturer, with a log file that can be saved to the PCAP format. It must be run as rooted and likely uses the libpcap library. It seems nearly as complete as Wireshark and comes with some interesting features:
- filters (by domain, by ip, by application by protocol).
- statistics (percentage of frames with a given protocol)
- It has an interface to create ip packets and send them to a given address (e.g. ICMP packets, HTTP packet with a given user-agent).

Demo version is free and Pro version costs 10.99 €.

### 2.2.7 SniffDroid
- Version: 1.4 (17th December 2013)
- By: SeriousDroid seriousdroid@gmail.com
- Needs root access: YES
- Open source: NO
- Android version: 2.2+

This widget consists in a single button where the user can start/stop the capture of packets. The output file is compatible with wireshark (.pcap format) and can be sent via mail at the end of the capture. Surprisingly enough, it is stated in the description of the application in the Google store that it does not use tcpdump or libcap.

### 2.2.8 Packet Sniffer
- Version: 1.0 (31st May 2010)
- By: Android Arts (AndroidArtsTeam@gmail.com)
- Official website: [https://sites.google.com/site/androidarts/packet-sniffer](https://sites.google.com/site/androidarts/packet-sniffer)
- Needs root access: YES
- Open source: NO
- Android version: 1.6+
- Presentation Video: [https://www.youtube.com/watch?v=7DKNwVzGqP8](https://www.youtube.com/watch?v=7DKNwVzGqP8)

The Application is quite old and was among the very first packet sniffers on the Google play. It must be run as root and needs the previous installation of the tcpdump package. We can choose to save the capture inside a local database or in a local file. It can capture bluetooth packets as well.
2.2.9 bitShark

- Version: 0.9.08.11.2013PS (11th August, 2015)
- By: Blake Hamilton (bitshark.dev@gmail.com)
- Needs root access: YES
- Open source: NO
- Android version: 2.3.3+

BitShark is a paid application (2.27 €), intended to be the equivalent of Wireshark on Android. It allows real time inspection of packets. It has a .pcap viewer, so that off-line inspection is possible as well. It even has the ability to reassemble and save web images from TCP streams.

2.3 Techniques of Packet Capturing

2.3.1 Using Libpcap

Libpcap is an API for capturing network traffic written in C and developed by the tcpdump developers in the Network Research Group at Lawrence Berkeley Laboratory since 1987 [9]. It reads packets directly from the NIC and proposes all the low levels functions needed to inspect them. In Unix-like operating systems, a user (or a program) must have superuser privileges to use it [30].

We found that there are several possible ways to use this library on Android:

- By installing a precompiled tcpdump binary and then building a custom application on top of it (since 2012, the tdpcump group provides a version of their software for Android [46])

- By running the Lil' Debi software. Lil' Debi is an open source application that installs a Debian subsystem on the android device [17]. It allows to run directly on an Android phone a large variety of debian package. For instance, it is possible to run Wireshark with this method.

Regrettably, in both cases, the device needs to be rooted.

2.3.2 Using a VPN

While investigating the Android packet capturers, we found out that the only applications that do not need root privileges use an implementation of the VpnService class [8] of the Android API (added in Level 14).

A VPN is usually used to enable users to send and receive data across shared or public networks as if their devices were directly connected to a private network. It is classically used in the following way (figure 2.1).
On the server of the head office is installed a VPN server. The routers of the regional offices and each of the roaming devices implement a VPN client. The clients relay their traffic to the server in an encrypted tunnel.

There exist various pieces of software to deploy a VPN system. Among them, we chose to work with the openVPN software [12] because it is easy to deploy on Linux devices and there is an open-source Android version, that we could use or embed in our application.

Concretely, the application creates a tap or a tun interface to respectively create a layer 2 or a layer 3 VPN (see Figure 2.2). In our case, a layer 3 VPN is enough to forward IP packets.

The VPN application internally knows — which application emits each packet, and to which application the incoming packets must be forwarded.

This has a key importance insofar as it allows us to capture the network activity of each Android application without the need of being root.
Chapter 3

Technical Solution

In order to fulfill the project’s requirements, we have investigated various possible solutions. We present in the chapter two possible architectures. Each solution is based on a different ruse to capture traffic:

- the first solution is based on a WiFi access point to receive traffic (from the device to be monitored)
- the second one is VPN-based: if a VPN client is installed on the monitored device, then all its traffic is routed to a VPN server where it can be captured.

We will present them in sections 3.1 and 3.2 and finally compare them in section 3.3 to only choose one of them — Solution 2.

3.1 Technical Solution 1

The first technical solution has two modes of operation:

- In the first mode, the mobile device runs the monitoring application and implements a Wi-Fi Access Point (AP) in order to capture traffic from any connected device.
- In the second mode, the same monitoring application is installed in the monitored device to capture its traffic and analyze its files (read and write mode). The device needs to be rooted.

3.1.1 Mode of Operation 1: Analyze Other Devices

With this mode of operation (see Figure 3.1), the application runs on an Android device which is inserted in the Wireless Local Area Network (WLAN) between the gateway and monitored device (which is not necessarily Android). It stores the captured packets in a database and makes them available through a web server so that they can be conveniently displayed on a large screen and further analyzed.

In a nutshell, it behaves as a packet analyzer, such as Wireshark, for other devices.

3.1.2 Mode of Operation 2: Analyze a Rooted Device

In this case (see Figure 3.2) the application runs on the monitored device. It is endowed as previously with a database to store packets and web server to retrieve them from a web browser. But now, since the application is installed on a rooted device, it has access to more information, that we are eager to retrieve such as:
3.2 Technical Solution 2

After we drew inspiration from the state-of-the-art technologies, we came up with this VPN based solution.

It is made of three components:

- an Android application implementing an OpenVPN client, that could send upon a request from the server specific information if the device is rooted (about the file being read/written, the application id associated to a given packet, etc.).

- a monitoring server with a static IP address implementing an VPN server that stores the packets received through its tunnel interface in a database. A Web interface would allow receiving instructions from a client browser and display the packets with the monitoring information sent by the Android application.

- the application associated to each packet,
- the read/write operations of the applications.
Figure 3.3: Technical solution 2

- a web application that displays a graphical interface to — interact with the Android application, interact with the server (e.g. start/stop recording) and see the packets received in real time.

3.3 Comparison of the solutions

Both solutions allow to

- capture traffic without the need of being root
- read comfortably the capture on a wide screen
- provide more details about the internal state of the Android device if it is rooted

With the first solution, we can capture traffic of any device, provided that it uses Wi-Fi.

With the second solution, we can capture traffic of any Android device being able to instantiate a VPN server (Android API level 14 = 98.4 % of active device as of June 2017), including traffic from Wi-Fi and 3G/4G. Moreover, it allows to have external storage.

Table 3.1 shows how both solution respect the constraints defined in the project requirements:

<table>
<thead>
<tr>
<th></th>
<th>Wi-Fi Access Point</th>
<th>VPN Tunnel</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>C2</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>C3</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>C4</td>
<td></td>
<td>✔</td>
</tr>
</tbody>
</table>

Table 3.1: Packet capture applications for Android

We decided to implement the second solution (VPN tunnel) because it respects all the constraints defined in the project requirements. We will present in the following chapters the
three components of its architecture — the Android application, the monitoring server and the client browser.
Chapter 4

Android Application

To elaborate a first working version of our system, the Android application needs to act as a VPN client to route all the IP traffic of the device to a server.

We did not have to develop anything, for there already exist VPN clients for Android. We chose to work with OpenVPN for Android as it is compatible with the openVPN software, which is widely used on Unix/Linux systems.

After a quick presentation of OpenVPN for Android in section 4.1, we will see how to setup the VPN tunnel on both client and server sides in section 4.2.

4.1 Presentation of OpenVPN for Android

OpenVPN for Android, is a port of the OpenVPN software [12] for Android that uses the VPNService of Android API level 14+ (Ice Cream Sandwich) that allows to create a VPN service without root access.

- Version: 0.6.66
- Last update: 14 June, 2017
- By: Arne Schwabe (arne-openvpn@rfc2549.org)
- Needs root access: NO
- Android version: 4.0+
- Open source: YES (GPLv2)
- GitHub: https://github.com/schwabe/ics-openvpn [41]

4.2 Setup of the VPN Tunnel

In this section, we will detail how we configured the VPN server (subsection 4.2.1) and the VPN client on the monitored device (subsection 4.2.2) to set up a tunnel to capture all traffic from the client.

While it is routed through the tunnel, the traffic can not be intercepted insofar as it is encrypted and both sides should be authenticated. We will provide more details about OpenVPN security in subsection 4.2.3.
4.2.1 Configuration of the OpenVPN Server

On a Debian-based server, we install the OpenVPN software from the package repository and set the default configuration in the file `/etc/openvpn/server.conf`

```bash
$ sudo apt-get install openvpn
$ gunzip -c
  /usr/share/doc/openvpn/examples/sample-config-files/server.conf.gz
> /etc/openvpn/server.conf
```

With the default settings, the VPN server is nearly ready to use on port 1194/UDP. Some additional settings shown in Code 4.1 allow to — redirect the client’s gateway through the VPN (the client’s routing table will be changed) and set the client’s DNS server:

```bash
push "redirect-gateway def1 bypass-dhcp"
push "dhcp-option DNS 147.83.2.3" # dns server from the VPN server side
```

Code 4.1: Push Option in `/etc/openvpn/server.conf`

To configure the encryption of the tunnel, we fill in the security parameters:

- "ca": certificate file of the certification authority
- "cert": server’s certificates (contains the public key)
- "key": private key of the VPN server (should be kept secret)
- "dh": Diffie Hellman parameters

Additionally, the machine that runs the OpenVPN server should be configured as a router with a source Network Address Translation (NAT). For example, in Debian-based systems, assuming that eth0 is connected to the Internet, this can be achieved with the following (non persistent) commands:

```bash
$ sudo echo 1 > /proc/sys/net/ipv4/ip_forward
$ sudo iptables -t nat -A POSTROUTING -s 10.8.0.0/8 -o eth0 -j MASQUERADE
```

4.2.2 Configuration of the OpenVPN Client

We configure the OpenVPN client by creating a profile file (.ovpn), that must be loaded into the Android OpenVPN application. The openvpn file contains the following parameters:

- "ca": certificate file of the certification authority (common with the server)
- "cert": client’s certificate (signed by the common certification authority)
- "key": client’s private key.
4.2.3 Security of the VPN Tunnel

OpenVPN uses SSL/TLS for authentication and key exchange. As we've seen in subsection 4.2.1 and 4.2.2, both server and clients have certificates in order to establish a bidirectional authentication (i.e. each side of the connection must present its own certificate) [13].

The packets that go through the tunnel are not encrypted with RSA keypairs. OpenVPN uses precomputed Diffie Hellman (DH) parameters for encryption. The DH algorithm is better suited for encryption because it allows recreating sessions keys much faster than RSA. The session key is regenerated every hour (by default) [14].

4.2.4 Further Reading

Reference [26] proposes more details on how to configure the OpenVPN server.
Chapter 5

Monitoring Server

The monitoring server is installed at the end of a VPN tunnel and receives all the traffic from
the Android device. It is composed of two elements — a packet capturer and a TCP proxy that
performs a Man In The Middle (MITM) attacks to decrypt SSL/TLS traffic.

We will detail how both of these components have been implemented on the server in section
5.2 and 5.3.

But first, in section 5.1, we will present a technology analysis of packet capturers.

5.1 Technology Analysis

As detailed in section 2.3.1 the original library to capture packets on the interface of a Unix
system is libpcap. It is written in C, but it has been ported to many languages. Most of the
libraries that we found can capture packets but also can analyze them if they support protocol
used to format the packet.

- node pcap in Node JS: has http and tcp analysis tools [34]
- pcap4j in Java: supports IP, PPP, 802.11, ARP, IPv4, ICPv4, TCP, DNS protocols [33]
- JNet pcap (packet capturer) and jNetStream (packet decoder) in Java: support more than
  a hundred protocols [36]
- pypcap in Python: a wrapper for libpcap (only works as a packet capturer) [32]
- scapy in Python: a packet manipulation program with more than a hundred supported
  protocols (Ethernet, IP, UDP, ICMP, UDP, ARP, DNS, ...) [35]

We decided to use Python with the Scapy library. Indeed the Python language is really
widespread in the realm of hackers. Therefore, if we ever want to add new functionalities to our
monitoring server, chances are high that we could base ourselves on an existing Python library.
It is also commonly used for web applications.

The packets must be stored in a database. We decided to use the MySQL technology,
a classic, with a connector available in Python 2 and 3. Sure enough, they are many other
database systems compatible with Python that might have been equally satisfactory.

5.2 Packet Capturer

The basic functionality of the monitoring server is to capture packets and store them in a database.
For this purpose, we used the Scapy library [20] and then the Scapy3k library [16] when updating
our project for Python 3.
5.2.1 Presentation of Scapy

Scapy is a packet manipulation program. It enables the user to capture, analyze, forge and send network packets [19].

Its main asset is its ability to send packets and receive the corresponding response. It behaves as follows:

1. **Packet creation**: With Scapy, it is easy to create any kind of packet, without having to respect the protocols. Creating unconventional packets is specially interesting for hackers. The example 5.1 shows how to create and ICMP packet with padding (not payload)

2. **Packet transmission**: Scapy sends the packet

3. **Packet response**: Scapy returns a list of packet couples (request, answer) and a list of unmatched packets. This is the task of the `sr` (send/receive) command. Below is an example of `sr1`, an alternative to the `sr` command that only returns one response packet.

```python
# sudo scapy
Welcome to Scapy (3.0.0)
>>> p=sr1(IP(dst="upc.es")/ICMP()/"XXXXXXXXXXX")
Begin emission:
. Finished to send 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
>>> p
<IP version=4 ihl=5 tos=0x0 len=39 id=3387 flags=DF frag=0 ttl=251
  proto=icmp chksum=0xe6ff src=147.83.2.135 dst=10.192.235.0 options=[]
  |<ICMP type=echo-reply code=0 chksum=0xee45 id=0x0 seq=0x0 |
  |<Raw load='XXXXXXXXXXX' |>>>
```

Code 5.1: Command-line example of the Scapy `sr1` method (from ref. [19])

As we can see, scapy has some advanced features, most of which were not used in this project. To implement our packet capturer, we rather used the basic `sniff` method introduced in the next section 5.2.2.

5.2.2 Sniffing

The core of our capture program lies in the following 2 pieces of code.

- In Code 5.2, we start the session by calling the `sniff` method of the Scapy API with 2 arguments — `iface`: the interface to be monitored and `prn`: the callback triggered for each packet received on the interface.

- In Code 5.3 we define the actions to be performed for each packet — store the packet into database and add the packet to the conversations (conversations are detailed in section 5.2.3).
### 5.2.3 Conversations

A conversation is a summary of the packets exchanged between two endpoints. We define three types of conversations: IPV4 conversation, TCP conversations and UDP conversations. It contains the following information:

- $IP_A, IP_B$: IPV4 addresses of endpoints
- $Bytes = Bytes_{A\rightarrow B} + Bytes_{B\rightarrow A}$: Cumulated size of the IP payloads
- $Packets = Packets_{A\rightarrow B} + Packets_{B\rightarrow A}$: Number of packets exchanged between the 2 endpoints
- Duration: Time (in s) elapsed between the first and the last packet recorded
- Relative start: Time (in s) elapsed between the start of the session and the first packet of the conversation recorded

For TCP and UDP conversations, there are two additional fields: $Port_A$ and $Port_B$.

Figure 5.1 shows the summary of all the IP conversations during a session.

Without revealing too many boring implementation details, we confess that conversations are stored in a table of the MySQL database. They are associated to a line in one of the tables $CONVERSATION_{IPV4}$, $CONVERSATION_{UDP}$ and $CONVERSATION_{TCP}$ with a given id.

### 5.2.4 Code Structure and Unit Testing

The library we developed to implement the functionalities of the monitoring server is called **Mono**. The functions defined in this library typically interacts with the database or with the Scapy library. They are thematically organized in files (e.g. mono/session.py, mono_packet.py, ...) and tested within small scenarios in test files (mono/test/test_session.py, mono/test/test_packet.py).

Let us take the example of the function `get_or_create_conv(...)` to retrieve a conversation from the database. It is defined in file `mono_conversation.py` (see Code 5.4).
Let $IP_{src}$ and $IP_{dst}$ be respectively the source and destination ip of the packet; and $IP_a$ and $IP_b$, the IP endpoints of the conversation. An IP conversation matches the packet if:

$$IP_a = IP_{src} \text{ and } IP_b = IP_{dst}$$

or

$$IP_a = IP_{dst} \text{ and } IP_b = IP_{src}$$

A small scenario is written in file `test_conversation.py` (see Code 5.5) to verify that this rule is respected in two cases:

- When the function `get_or_create_conv(...)` is called with the same packet, it should return the same conversation (with the same id).

- If we invert the source and destination IP of the packet, the function must also return the same conversation.

```
#returns a dictionary of the conversation associated to the given packet.
def get_or_create_conv(packet, db, conv_type, id_session):
    #complex functions that ends up making an SQL request to the database
    ...
```

Code 5.4: `get_or_create_conv(...)` (from file `mono/mono_conversation.py`)

Having automated unit tests was a valuable help to spot regression issues while migrating the code from Python 2 to Python 3.
class ConversationsTestCase(unittest.TestCase):
    def test_add_conv_ipv4(self):
        # setup: connect to database
        db = connect(...)
        # test 1: call get_or_create_conv twice with the same packet
        # (it should return the same conversation)
        packet1 = mono_packet.create_ping_packet(...)
        conv1 = mono_conv.get_or_create_conv(packet1, db, 0, 0)
        conv2 = mono_conv.get_or_create_conv(packet1, db, 0, 0)
        self.assertEqual(conv1["id_conversation"], conv2["id_conversation"])
        # test 2: swap src_ip and dest_ip and call get_or_create_conv
        # (it should still return the same conversation)
        src = packet1.getlayer(IP).src
        packet1.getlayer(IP).src = packet1.getlayer(IP).dst
        packet1.getlayer(IP).dst = src
        conv3 = mono_conv.get_or_create_conv(packet1, db, 0, 0)
        self.assertEqual(conv1["id_conversation"], conv3["id_conversation"])

Code 5.5: Unit test (in file mono/test/test_conversation.py)

5.3 SSL/TLS Traffic Decoder

5.3.1 Presentation of mitmproxy

To decrypt SSL/TLS traffic, we decided to use the Python 3 program mitmproxy. It is defined in the Kali package repository as “an SSL-capable man-in-the-middle HTTP proxy. It provides a console interface that allows traffic flows to be inspected and edited on the fly” [3].

Mitmproxy is a TCP and HTTP/HTTPS proxy. “It pretends to be the server to the client, and pretends to be the client to the server” [1].

As most traffic on the Internet does not require the client to be authenticated, the traffic from the server is not encrypted.

To decrypt the traffic from the client, mitmproxy masquerades as the server by sending a fraudulent certificate to the client firmed by its own certification authority, containing its own public key (for which he has the corresponding private key), with the corresponding Common Name (CN) and Subject Alternative Names (SANs).

The major trick is to retrieve the CN and SANs that change according to the IP address or the domain name, which is sent via the Server Name Indication (SNI) protocol. To do this, they must first receive the server’s certificate that contains all these details before sending their own.

As a consequence, the initialization of a connection is performed in 3 steps: 1) start of the TLS handshake with the client until the SNI is received (if any); 2) TLS handshake with the server with the accurate ip address or domain name; 3) end of the TLS handshake with the client (with the accurate CN and SANs).

As pictured in Figure 5.2, the steps of this process are the following:

1. The clients connects to the server

2. mitmproxy intercepts the request and sends back a 200: Connection established to the client

3. The client then initiates the TLS handshake with the SNI (if any)
4. mitmproxy pauses the TLS negotiation with the client and ask the server’s certificate because it has now all the elements to do so.

5. mitmproxy completes the TLS handshake with the server and retrieves its certificate. It can now forge its fake certificate with the correct CN and SANS fields of the server’s certificate.

6. The fake certificate is sent to the client.

7. The conversation can now begin.

---

**Figure 5.2: How mitmproxy works (from http://docs.mitmproxy.org [1])**

Mitmproxy has four modes of operation:

1. **Regular mode**, the client application must be configured to use an HTTP proxy. (for example in a Firefox browser in Preferences → Network → Proxy set the IP and port of the HTTP proxy). Traffic is redirected to the proxy at the HTTP layer.

2. **Transparent mode**, no client configuration is required, IP packets are emitted by the client as if there were no proxy. Mitmproxy is installed in a router that sees all the traffic from the client. Traffic is redirected to the server at the TCP layer.

3. **Reverse proxy mode**, mitmproxy receives requests from the internet and forward them to another server.

4. **Upstream proxy mode**, mitmproxy can be configured to forward all its traffic to another HTTP proxy.

The **transparent mode** is the best adapted to our situation.

### 5.3.2 Implementation of the Traffic Decoder

To integrate mitmproxy in our server program, we use the mitmproxy scripting capability whereby a callback function can be called for specific events: reception of a TCP packet, of an HTTP request, of an HTTP response [2]. In our case, we are interested to work at the TCP layer in order to decode as much traffic as possible. To do this, we define the `tcp_message(flow)` callback (see Code 5.6). Each time this function is called, it means that a new TCP packet
has been received. We then simply retrieve the information of this packet and place it in the database.

```python
def tcp_message(flow):
    # get last message of the flow
    lm = flow.messages[-1]
    ...
    # retrieve flow information: ip and ports of endpoints
    ip_server = flow.server_conn.address.host
    ip_client = flow.client_conn.address.host
    port_server = flow.server_conn.address.port
    port_client = flow.client_conn.address.port
    
    # packet summary: ips, ports and payload
    m = {
        "from_client": lm.from_client,
        "packet_length": len(lm.content),
        "ip_src": ip_client if lm.from_client else ip_server,
        "ip_dst": ip_server if lm.from_client else ip_client,
        "port_src": port_client if lm.from_client else port_server,
        "port_dst": port_server if lm.from_client else port_client,
        "payload": lm.content,
    }
    # add tcp decrypted packet to database
    (id_packet, packet) = mono_mitm.mitm_into_db(m, current_id_session, db)
    ...
```

Code 5.6: Definition of the `tcp_message(flow)` function in file `mitm.py`

This code is interpreted only if the mitmproxy process is active which is started and stopped from our web interface (presented in chapter 6). We use the following method (Code 5.7) to launch it in three steps:

- redirect the traffic from the VPN tunnel to the port that mitmproxy will be listening to (8182) with the `iptables` command
- parametrize mitmproxy with the appropriate options — `cadir`: directory containing the certificates; `listen_port`: traffic must be forwarded to this port; `mode`: transparent (selected among the 4 proxy modes regular, transparent, reverse, upstream); `rawtcp`: True to enable the TCP proxy (every tcp packet will be intercepted); `scripts`: path to where the `tcp_message(flow)` callback is defined; `ssl_insecure`: set to True to continue the upstream SSL handshake even if the server’s certificate is invalid; `ssl_version_client` and `ssl_version_server`: set to all to support all the versions of SSL/TLS (downstream and upstream); `tcp_hosts`: set to `['.*']` to intercept all TCP traffic.
- start the mitmproxy program

### 5.3.3 Limitations of the Traffic Decoder

There are situations where the traffic decoder does not work:

- With a custom encryption scheme (not TLS/SSL).
- When the transport protocol is not TCP (for instance TLS over UDP).
def start_mitm():
...

global mitm_master

#add iptable rule
os.system("iptables -A PREROUTING -t nat -s 10.8.0.0/24 -p tcp -j REDIRECT --to-ports 8182")
...

#set mitmproxy options
dump_options = options.Options()
dump_options.merge(
  dict(
    cadir = '~/mitmproxy',
    listen_port = 8182,
    mode = 'transparent',
    rawtcp = True,
    scripts = ['mitm.py'],
    ssl_insecure = True,
    ssl_version_client = 'all',
    ssl_version_server = 'all',
    tcp_hosts = ['.*'],
  )
)

#start mitm process
server = process_options(dump_options)
mitm_master = dump.DumpMaster(dump_options, server)
mitm_master.run()

---

Code 5.7: Definition of the start_mitm() function in file mono/mono_mitm.py

- When an application uses a client certificate.

- With certificate pinning: the operation of pinning consists in associating a host with their expected x509 certificate, public key or certification hierarchy [15]. If these properties change, the client will not trust the certificate.

- With HTTP Public Key Pinning (HPKP): an extension of the HTTP protocol where a server can inform the client of a list of public key hashes associated to period of time during which they are valid [28]. Only the public keys corresponding to the hashes can be used afterwards by the server.

In those cases, any attempt to use mitmproxy will result in a TLS handshake failure which will disrupt the normal traffic of the Android device and prevent the applications to do what they are programmed to do. This is a major problem when trying to analyze them.

However, it is still possible to program mitmproxy to stop intercepting traffic to specific domains as long as they are known in advance.

---

Chapter 5. Monitoring Server
Chapter 6

Web Application

The web application is the major component of the Mono project.

It uses a Python Web framework (Flask) and a client-side JavaScript framework (Angular JS).

Both of these elements will be introduced in section 6.1.

In section 6.2, we will present with more details the Angular JS Framework, so that the reader gets used to its design.

In section 6.3 we will expose the architecture of this web application.

To better understand this architecture, and how components relate one with another, we will introduce the dataflow in section 6.4 and see how a simple user action is handled by the system.

Finally, we will see the packet display mechanism together with the DataTable JavaScript library in section 6.5.

6.1 Presentation

In subsection 6.1.1, we detail the main tools with which we’ve built this Web interface — Flask: a Python Web server and Angular JS: a JavaScript application framework. We will explain why we chose them.

Subsection 6.1.2 unveils the look of the Web interface.

6.1.1 Technology Analysis

The Web server uses the same technology (i.e. the Python language with a MySQL database) as the monitoring server (presented in chapter 5). This way, we can have a unique application on the server which is more maintainable.

Among all the Python web frameworks [18], we distinguish two categories:

- Full-Stack Frameworks: They provide all the elements needed for a Web application (an HTTP server, a storage mechanism such as a database, a template engine, a request dispatcher, an authentication module and an AJAX toolkit). Django is the most famous of them.

- Non Full-Stack Frameworks: They only provide some of the previously cited features with always a base “application server”. The most widely used is Flask.

We have decided to use Flask (0.12.2 Released 2017-05-16) described by its developers [25] as “a microframework for Python based on Werkzeug, Jinja 2 and good intentions”. We chose it because it includes the two elements that we need for our web application — a built-in development server and a request dispatcher. According to the documentation [24], the built-in server is “not suitable for production as it doesn’t scale well and by default serves only one
This does not matter to us, as only one user is supposed to be using the web application at a time.

On the client side, we have decided to use the Angular JS web application framework, because its capacity to reduce development time needs no further proof. What has been specially interesting to us is its aptitude to automatically bind JavaScript variables with HTML elements. Of course, many other JavaScript libraries would have been equally satisfactory. Section 6.2 describes with more details this framework.

### 6.1.2 User Interface

Figure 6.1 shows a capture of the Web interface. On the left hand side is the list of all session with a form to create a new one. On the right hand side we can see:

- **at the top:** the session’s details (name, date of creation and monitored interface) with the main controls — start/stop recording, start/stop MITM (decrypting traffic with a Man In The Middle attack)
- **at the bottom:** various tabs, each displaying a list of packets (all packets, packets captured with the MITM attack, packets specific to a conversation) or a list of conversations (IP, UDP, TCP)

---

**Figure 6.1: Web user interface**

---

### 6.2 Angular JS

Angular JS is a client-side JavaScript framework. It is intended to be used for dynamic web applications.

With Angular JS, the page is not written in pure HTML, but with a specific template language, that is compiled when the page is rendered. This template language allows writing specific directives in the HTML code, to bind HTML objects to JavaScript objects, or GUI events to JavaScript methods, with little lines of code.

The official introduction to Angular JS [5], gives some examples of the essential elements that are needed when writing a web application and that angular JS provide with its templating language:
Data-binding

DOM manipulation

DOM event handling

Grouping of HTML into reusable components

They are explained in the following examples.

Data binding:

1. `<input ng-model="name">
2. `<div>{{name}}</div>

In this example the `ng-model` directive allows to change the content of the JavaScript variable "name" from the HTML input. And the `{{name}}` directive, displays the content of this JavaScript variable. This is called 2-way binding, as the HTML can be updated by the JavaScript and vice-versa.

DOM manipulation:

With the `ng-repeat` directive, we can easily repeat DOM fragments.

1. `<ul>
2. `<li ng-repeat="x in names"> {{ x }} </li> </ul>

With the `ng-show` directive, it is also possible to hide/show DOM elements upon a specific condition:

1. `<div ng-show="a"> <!-- Showed if 'a' is true--> </div>

DOM event handling:

Directly from the HTML, it is possible to attach new behavior to DOM elements. For instance in the following example the method `myFunction()` will be called when the button is clicked, thanks to the `ng-click` directive.

1. `<button ng-click="myFunction()">Click me</button>

Grouping of HTML into reusable components:

It is possible to include HTML content using the ng-include directive

1. `<div ng-include="'myFile.html'"></div>

6.3 Application Architecture

After the first weeks dedicated to elaborating a proof of concept, we expanded our prototype with a few features (pagination, server side processing, conversations). As a consequence, the number of lines of code started growing and it was time to redesign our client architecture.
Figure 6.2 represents the key components of our application and how they relate within the data-flow. Most of these components are standard elements of the Angular JS library and we will briefly detail their roles below.

### 6.3.1 View and Controller

The view is written in HTML with some Angular JS directives. Each view is related to a controller. We declare the view at the same time as the controller in the HTML:

```html
<body ng-app="monoApp">
  <div ng-controller="monoController as mono">
    <!-- The HTML inside is controlled by the monoController -->
    <!-- We display the content of $scope.greeting -->
    {{greeting}}  <!-- prints ¡Hola guapo! -->
  </div>
</body>
```

**Code 6.1: index.html**

In the Javascript file, we define the controller in the following way:

```javascript
var monoApp = angular.module("monoApp", []);
monoApp.controller("monoController", ['$scope', function($scope){
  var self = this;
  $scope.greeting = '¡Hola guapo!';
}]);
```

**Code 6.2: app.js**

A Controller contains only the logic needed for a single view. They are meant to handle user actions. If these actions entail more complex algorithms, they must not do the job themselves, but rely on third-parties. In order to keep them slim, which is good practice, it is advisable to use services (see section 6.3.3).

Controllers communicate with their corresponding views via the scope object.
6.3.2 Scope

The scope is, “the glue between application controller and the view” [6]. Both controllers and directives have reference to the scope, but not to each other. The controller is therefore view agnostic and can be tested on its own, without the need of a view.

It is advised to consider the scope as read-only in templates and write-only in controllers [10]. We tried to follow this rule in our design.

Scopes are not the model but have reference to it. They gather the data, that are mutually shared by the view and the controller.

When a view is embedded in another one, the scope from the child view inherits from the scope of the parent view. Scopes inherit prototypically; it means that the child object can access all the properties and methods from the parent object, but if a property in the child object is changed, it won’t affect the parent object.

6.3.3 Services

In Angular JS, a service is a function, or object, that is available for an Angular JS application. In our application, the API we use to communicate with the server is stored in various services. For instance, we declare our Session API with a service in the following way:

```javascript
angular.module("monoApp").service('Session', ['Model', function(Model) {
  this.addSession = function(){...};
}]);
```

Code 6.3: excerpt of the file Session.js

It is now available for use in a controller, provided it is declared in its dependencies:

```javascript
monoApp.controller("monoController", ['$scope', 'Session', function($scope, Session) {
  this.addSession = function(){ //triggered by user action
    Session.addSession( ... ); //call to the Session API
  };
}]
```

Code 6.4: Use of the Session service in a controller (app.js)

6.3.4 The Model

Models represent the knowledge, the data presented to the user to interact with. In our architecture, the data are gathered inside the server database. The data needed by the application are downloaded from the server and kept inside a “proxy” model, which is a single object. It is the task of our API functions to keep this client-side model synchronized with the database.

Besides, the controller that want to be informed of the evolution of a variable, need to subscribe to the model. When the model will update a variable, it will fire this event to all controllers (see arrow 6 of figure 6.2).
6.4 Dataflow

We present thereafter an example of how the user interacts with the database to change the name of the current session. In order to illustrate our dataflow, we will follow the eight steps of figure 6.2. Let us start from the GUI. When the user wants to change the current session’s name, he simply clicks on it to open a pop-up input as illustrated on figures 6.3 and 6.4.

![Figure 6.3: Session information](image)

![Figure 6.4: Change session information](image)

**Step 1:**

After confirming the new session name by clicking on the blue tick-button, the view triggers the controller method `mono.updateName()`, thanks to the `onaftersave` attribute.

```html
<div class="col-md-6">
  <a href="#" editable-text="mono.newName" onaftersave="mono.updateName()">{{ current_session.name || "" }}</a>
</div>
```

Code 6.5: excerpt of the file session-top.html

**Step 2:**

The controller’s method presented below clones the current_session object and changes its name attribute. The GUI is not updated yet, because we have to perform the modification on the database first. This is the purpose of the method `updateCurrentSession()` of the `Session` API.
```javascript
this.updateName = function(){
  var sess_clone = angular.extend({}, $scope.current_session);
  sess_clone.name = self.newName;
  Session.updateCurrentSession(sess_clone);
};
```

Code 6.6: updateName method in the monoController (app.js)

**Step 3 and 5:**

The Session API (an angular JS service), is in charge of contacting the server to request the modification of the session via an AJAX call. Only when the server has confirmed that the session has been modified, the (proxy)Model can be updated by calling the method `Model.setCurrentSession()` on line 15.

```javascript
// updates any session
this.updateSession = function(sess, cb){
  $http.post('/awebservice', {method: "update_session", params: {session: sess}},)
  .then(function(response) {
    sess = response.data.session;
    if (cb instanceof Function) cb.call(sess, sess);
  }, function(error) {console.log(error);});
};

// updates the current session
this.updateCurrentSession = function(sess, cb){
  this.updateSession(sess, function(rec_sess){
    new_curr_sess = rec_sess || Model.getCurrentSession();
    Model.setCurrentSession(new_curr_sess);  //
    if (cb instanceof Function) cb.call();
  });
};
```

Code 6.7: Definition of Session.updateCurrentSession method (Session.js)

**Step 4:**

The server is in charge of updating the database and sending the modified session to the client.

At the server side, the first function called is the web service and acts as a dispatcher, to call the right
function in the server’s API for each request. Each request is identified by its ‘method’ parameter. Here the ‘method’ update_session, triggers a call to the mono_session.update_session() (line 8) in the mono_session.py file. Once the session has been modified in database, we reply by sending it back to the client.

```python
@app.route("/awebservice", methods=['GET', 'POST'])
def webservice():
    try:
        ... 
        elif (method == 'update_session'):
            print 'update session'
            sess = params["session"]
            mono_session.update_session(sess, db2) #
            return jsonify(session =
                          mono_session.get_session(sess["id_session"],db2))
```

Code 6.8: Method update_session in the server webservice (server.py)

Code 6.9, shows the SQL query executed by the server when updating a session.

```python
def update_session(session, db):
    try:
        sql = "UPDATE SESSIONS SET name = %s, iface = %s, date = %s WHERE id_session = %s"
        params = (session["name"], session["iface"],session["date"],
                  int(session["id_session"]))
        db.cursor().execute(sql, params) #safe sql querry
        db.commit()
    except Exception as e:
        ...
```

Code 6.9: SQL query executed by the server (mono_session.py)

### Step 6:

Once the Session Service has received the answer from the server, the (proxyModel) is updated by calling the Model.setCurrentSession() method (detailed below) which, in turn, triggers an event to inform all the controllers (listeners of this event).

![Diagram of Step 6](image)
1. `this.setCurrentSession = function(new_curr_session){
2.   this.current_session = new_curr_session;
3.   $rootScope.$broadcast('Model::currentSessionUpdate',
4.     this.current_session);
5.};

Code 6.10: The model is updated and fires an event (Model.js)

**Step 7:**

When the main controller receives the event that the Model has changed, a callback function is called to update its attributes.

1. `$scope.$on('Model::currentSessionUpdate', function(event, curr){
2.   //automatically updates the HTML
3.   $scope.current_session = curr;
4.   //re-initialize forms
5.   self.newIface = $scope.current_session.iface;
6.   self.newName = $scope.current_session.name;
7.   //update current session's name in sessions' list
8.   var res = $scope.sessions.find(function(element){
9.     return element.id_session == curr.id_session;
10.   });
11.   angular.extend(res, curr);
12.});

Code 6.11: The event Model::currentSessionUpdate is received(app.js)

**Step 8:**

When a controller modifies the attributes of the `$scope`, they are automatically changed in the HTML code by the Angular JS library (2 way binding property).

### 6.5 Packet Display

One of the main tasks of the Mono Web application is to display the traffic captured during a session or being recorded while a session is active. In this latter case, the graphical interface should be regularly updated.

We cannot simply display all the packets on a Web page, as it would slow navigation down or eventually freeze the browser page. Therefore we have to use pagination.

In subsection 6.5.1, we will see the possible ways to perform pagination (with Local Storage or server side processing) and choose one among them (server side processing).
In subsection 6.5.2, we will present a way to implement server side processing with Datatables, a JavaScript library.

### 6.5.1 How to paginate:

To avoid running low on memory, we have to implement pagination, with one of the following scenarios:

- **use the Local Storage** of the Javascript engine to store all the packets’ information and display only a small amount of them in a page on the browser.

- **use server side processing**, whereby all the packets are stored remotely (in the webserver database), and their content is retrieved each time it has to be displayed in a page.

Although, the second option can be considered as less efficient, because it consumes more bandwidth (an Ajax request must be sent for each user action), it overcomes the memory limit problem of the Local Storage (5 MB max). This is why we have decided to choose this solution.

It is usually the rule in web design to unburden the server, and give as much work load as possible to the client, but since in our case, there can only be one client for one server, we rely on the server storage and its computing capabilities.

### 6.5.2 Datatables

In this subsection, we present the DataTables library and then give a more thorough explanation of how it performs server-side processing.

**Presentation**

DataTables is a JavaScript library (a jQuery plugin) to deal with dynamic tables in HTML[21]. It is licensed under the **MIT license** and endowed with some features that are specially interesting to us:

- pagination,
- instant search
- multi-column ordering

To perform these actions, DataTables has two different modes:

- **Client-side processing**: the data to be displayed is given once to the DataTables constructor and then data-processing is performed with JavaScript in the browser.

- **Server-side processing**: “an Ajax request is made for every table redraw, with only the data required for each display returned” [21]. Data-processing is performed by the server.

Figure 6.5 shows the default display of an HTML table made with Datatables.

**Server side processing**

The data that are displayed on the client screen are directly connected to the content of a table in a database. This is of major interest when the data are constantly updated on the server side (while recording packets) and when there is a great amount of data that cannot be displayed at the same time.
On each user action, the Datatable JS library sends to the server an AJAX request with the following list of parameters in Javascript Object Notation (JSON) format:

```json
{
    "draw": 1,
    "columns": [
        {
            "data": "id_packet",
            "name": "ID packet",
            "searchable": true,
            "orderable": true,
            "search": {"value": "", "regex": false}
        },
        {
            "data": "ip_src"...
        },
        {
            "data": "ip_dst"...
        }
    ],
    "order": [{"column": 0, "dir": "asc"}],
    "start": 0,
    "length": 10,
    "search": {
        "value": "",
        "regex": false
    },
    "method": "datatables_packets",
    "id_session": 94,
    "params": {} //
}
```

**Code 6.12: Example of a JSON query from datatables.js**

We specify:
• **columns**: each column of the database table we want to display (id_packet, ip_src, ip_dst, etc.).

• **order**: the column according to which the data must be sorted (here we sort according to the first column, in the ‘asc’ direction).

• **start** and **length**: start point of the data set, and number of records to display.

• **search**: specify if the user has entered a search string. The search will be carried out server side in all columns that have the searchable parameter set to true.

• The other parameters (“**method**”, “**id_session**”, “**params**”) have been added by us to select the desired SQL table and session.

To process this query on the server side, we used a script from the datatables.net website [22] (Python + CGI + MySQL) and adapted it to our software stack model (Python + Flask + MySQL).

The purpose of the server-side script is to translate the Datatables parameters into a MySQL query to fetch the desired table rows.

The JSON parameters in Code 6.12, will be translated to the following MySQL query (Code 6.13):

```
SELECT SQL_CALC_FOUND_ROWS id_packet, ip_src, ip_dst, ...
FROM PACKETS
WHERE id_session = 94
ORDER BY id_conversation_ipv4 asc
LIMIT 0, 10
```

Code 6.13: Translation of the JSON parameters to MySQL query

When a search string is entered, we look for a match in all columns. The following Code 6.14, shows the MySQL query with search["value"]='TCP'.

```
SELECT SQL_CALC_FOUND_ROWS id_packet, ip_src, ip_dst, ...
FROM PACKETS
WHERE id_session = 94 AND ( id_packet LIKE '%TCP%' OR ip_src LIKE '%TCP%',
ORDER BY id_packet asc
LIMIT 0, 10
```

Code 6.14: MySQL query with string search
Chapter 7

Security Concerns

In the design of a Network security tool, as in any software design, it never hurts to evoke security threats together with their possible countermeasures.

In section 7.1, we will present a list of measures that have been taken to guarantee the safety of the development environment, that must also be applied to production environments. In section 7.2, we will focus on the specific issue of SQL injection and see what measures have been taken to circumvent it.

7.1 Security Issues

As a packet capturer, Mono needs to inspect all traffic of an interface and uses the libpcap library (which requires high privileges). As a transparent decrypting proxy, it needs to forward all traffic of an interface to a given port with the iptables command (which requires root privileges). Consequently, the Mono program is run as root. If a hacker were capable to take control of the program, he might be able to execute commands with the same right as of the program, which would compromise the entire system.

During the elaboration of the project, we tried our best to protect the system with the following set of security measures:

- Use https to avoid eavesdropping: the Mono Web application can be configured to use HTTPS. We used TLS certificates firmed by the letsencrypt authority [4].
- Place the server behind a reverse proxy with a strong firewall policy and port forwarding.
- Secure the system against SQL injections: we detail this point in next section 7.2.

We also can think of other measures, that we did not have a chance to implement, that are desirable in the future.

- Set up a user system, so that only authenticated users can access the interface.
- Only allow connections from a given location.

7.2 Protecting Against SQL Injection

In order to implement the server-side processing of the DataTables JavaScript plugin, we based ourselves on a existing piece of program [22], proposed on the official DataTables website.

It turns out that this script is sensitive to SQL injection, because it trusts JSON parameters that come from the client, without checking that they do not contain malicious SQL code.
Every attribute of the JSON query made by the DataTables library (Code 6.12) can be modified by a user. These inputs must be sanitized.

We will present two ways of sanitizing user inputs — with argument escaping (subsection 7.2.1) and with withelisting (subsection 7.2.2).

7.2.1 Argument Escaping

The server program uses the MySQLdb python library. The easiest way protect the SQL queries is to trust the execute(self, query, args) method of the cursor.MySQLCursor class.

In the following example (code 7.1), if the parameters come from user inputs, their potential malicious content will be neutralized with the "cursor.execute()" method.

```python
#open connection to MySQL server
import MySQLdb
from flask import request
db = MySQLdb.connect(host='localhost', user='', password='', db='')
cursor = db.cursor()

#retrieve id_packet from user input
id_packet = request.get_json()['id_packet']

#execute query
sql = "SELECT * FROM packets WHERE id_packet = %s"
cursor.execute(sql, (id_packet,))  #user input is escaped
rows = cursor.fetchall()
```

Code 7.1: Parameters are escaped with the execute method

On the other hand, the following code 7.2 is vulnerable to SQL injections because it performs direct string substitution with the % operator:

```python
#retrieve id_packet from user input
id_packet = request.get_json()['id_packet']

#execute query
sql = "SELECT * FROM packets WHERE id_packet = %s or id = %s"
cursor.execute(sql, (id_packet,))  #this is BAD
rows = cursor.fetchall()
```

Code 7.2: Erroneous use of the cursor.execute() method

**Injection based on 1=1**

Let us examine how vulnerable boths codes are vulnerabilities to SQL injections and assume that an attacker tries an injection by changing the id_packet parameter:

```python
id_packet = "3400 OR 1=1"
```

If we don’t use the cursor.execute() method correctly (Code 7.2), the server executes the following query:
Select all columns from the PACKETS table where either the id_packet is 3400 or it satisfies the condition 1=1. All lines of the table will be returned (the code is injectable).

On the other hand, if the `cursor.execute()` method is used correctly (Code 7.1), the following query will be executed:

```
SELECT * FROM PACKETS WHERE id_packet = '3400 OR 1=1'
```

The injection does not work because parameters are quoted. The MySQL engine will succeed to convert the string '3400 OR 1=1' to the integer 3400 and return only one row (the code is not injectable).

If we notice that the user input has been quoted, we might try to trick the MySQL engine, by closing the quote inside the user input string:

```
id_packet = "3400 OR 1=1; --"
```

The injection will not work with Code 7.1 because strings are escaped. So the following query will be passed to the MySQL engine:

```
SELECT * FROM PACKETS WHERE id_packet = '3400 OR 1=1; --'
```

Here again, the string '3400 OR 1=1; --' will be successfully converted to the integer 3400 and only one row will be returned.

### Injection based on batched SQL Statements

The method `cursor.execute()` is designed to take only one statement, because it makes no guarantees about the state of the cursor afterward. Therefore, injections based on batched SQL statements are not possible.

For instance if we supplied the input:

```
id_packet = "3400; DROP TABLE PACKETS; --"
```

the program will have tried to execute the query:

```
SELECT * FROM PACKETS WHERE id_packet = 3400; DROP TABLE PACKETS
```

and an error would have been thrown ("Commands out of sync; you can’t run this command now").

### 7.2.2 Whitelisting

With the python method `cursor.execute()`, not all strings can be escaped. This is the case when we have dynamical tables and columns names, for instance when we use the datatables Javascript plugin in "server-side-processing" mode (see Code 6.12).

For example, the following code will not work:

```python
cols = ["id_packet", "ip_src", "ip_dst"] # from user input
query = "SELECT %s, %s, %s FROM PACKETS WHERE"
params = (cols[0], cols[1], cols[2])
cursor.execute(query, params)
rows = cursor.fetchall()
```
This is because the columns names will be interpreted as strings. The following query will be passed to the MySQL engine (and it does not return the desired data):

```sql
SELECT 'id_packet', 'ip_src', 'ip_dest' FROM PACKETS
```

In this situation, we won’t any other choice but to use string concatenation, but in order to avoid injections, the users input must be checked against a whitelist:

```python
cols_input = ['id_packet', 'ip_src', 'ip_dst']  # from user input
cols = whitelist(cols_input)  # comparison with allowed strings
query="SELECT %s FROM PACKETS" %(', '.join(cols),)
cursor.execute(query)
rows = cursor.fetchall()
```

Code 7.3: Safe SQL query where column names are whitelisted

In Code 7.3, we protect the SQL query by whitelisting the column names. The function used (whitelist(cols_input)) should try to match the inputs, with the list of actual column names.
Chapter 8

Conclusion and Future Developments

We have presented in this report the Mono project, a monitoring system for mobile applications to carry out network forensics investigations.

We laid the cornerstone of this solution by implementing a VPN-based web packet capturer available at https://github.com/gcanal/Mono.

Let us enumerate the current capabilities of the Mono application:

1. **Manage session**: Mono can deal with various sessions (a set of packets recorded during a certain period). It is possible to create and remove a session as well as exploring its recorded packets. From the Web interface, it is possible to change its properties — name, listening interface (but not while it is active). Only one session can be active at the same time.

2. **Capture packet**: From the interface, we can launch the Scapy capture process on the selected interface. Received packets are placed in a MySQL database.

3. **Decrypt SSL/TLS traffic**: Also with a single click, we can launch the mitmproxy process. All traffic from the selected interface, will be redirected to the mitmproxy listening port and decrypted if possible (section 5.3.3 details the limitations of the traffic decoder). The certificate of the mitmproxy's certification authority should first be installed in the monitored device.

4. **Explore conversations**: When a packet is received, it is added to a conversation, whose number of received and bytes are incremented. Even while the traffic is being captured, it is possible to list all IP, UDP and TCP conversations and see their associated packets.

5. **Paginate**: After having adapted the DataTables JavaScript library to our project architecture, the list of packets is directly read from the database, which is automatically divided into pages of adjustable length (10, 20, 50, 100). The list can be ordered according to any column.

6. **Select and download packets and conversations**: It is possible to select individually or collectively packets and conversations. For instance, it is possible to select:
   - all packets of a given UDP conversation,
   - all packets in the session,
   - a given TCP conversation,
   - all TCP conversations.

Once conversations and packets are selected, it is possible to download the corresponding packet list in a PCAP file. When a conversation is selected, all its packets are written into the PCAP file. Packets will only appear once (even if they belong to multiple conversations).
7. **Search**: It is possible to search in packets’ headers (IP address, port number) as well as in their payloads.

At the end of the project, it is time to reconsider the requirements that have been established at the beginning to see if they have been fulfilled. Table 8.1 shows an objective summary of the state of the project. The election of the appropriate architecture allows to respect all projects constraints. But some functionalities are still missing:

- $(\mathcal{P}_6)$ List information of each IP address the mobile device connects to (name resolution, geographic location, appearances in abuse databases, etc.)
- $(\mathcal{P}_7)$ List files accessed (read and write mode) during the session by each application (only possible with rooted/jailbroken devices).

<table>
<thead>
<tr>
<th>External storage</th>
<th>Real time</th>
<th>Non root</th>
<th>Forensics friendly</th>
<th>List packets</th>
<th>Download packets</th>
<th>List IP/UDP/TCP conversations</th>
<th>Search in header and payload</th>
<th>Decryp TLS traffic</th>
<th>IP information</th>
<th>Files accessed</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Table 8.1: Mono application vs Project requirements**

To complete the specified system and fulfill functionality $\mathcal{P}_7$ (**Files accessed**), a new component must be added to the system — a mobile application able to:

- implement an OpenVPN client
- relate sent packets to applications’ id
- explore files accessed by applications in read or write mode

Indeed, the mobile application we’ve used so far (**OpenVPN for Android**) implements the OpenVPN client but cannot be extended to also address the last two items. Therefore, a new mobile application should be created from scratch. This is one of the many possible follow-ups of this project.

Functionality $\mathcal{P}_6$ (**IP information**) has not been implemented either due to time constraints. Along with this initial functionality, during the development of the project we have also identified several others that can improve the Mono system. We present hereafter a non-exhaustive list of important features:

- **User authentication system**: As explained in section 7, the application is quite sensitive and must be run on a public IP server. It is not desirable that anyone can connect to this Web interface.

- **Zip decompression**: Often web pages are compressed. In order to quickly see its content without using Wireshark, we would need to support TCP reassembly and decompress the reassembled files, when we detect an archive (**rar, zip, tar.gz**)
• **File detection**: If an application performs data-leakage, it would be interesting to know which files are being sent and to see their content, specifically media files (images, video). As previously stated, Mono would need to support TCP reassembly.

• **Name resolution**: As multiple domains can be associated to a same IP address, it would be interesting (more meaningful) to perform name resolution by inspecting captured Domain Name System (DNS) queries.

• **Filtering system**: the actual filtering system is quite limited. It is only possible to search with a single string and see if it matches any field of the IP packet (IP, port, etc). It would be profitable to have multiple filters (for instance to look for packets with a given destination IP containing a given string) and also to filter packets within a given time interval.

As the reader will agree, there is still room for improvement in this project. But let us put forward the innovative architecture of this application that respects all the constraints to be conveniently used in a network forensics investigation.

Amid all this gloom, perhaps we can take comfort from this one ray of hope.
Bibliography


Chapter 8. Conclusion and Future Developments

Libpcap Bindings


Chapter 8. Conclusion and Future Developments


Android Applications


Glossary

**ADB**  The Android Debug Bridge (ADB) is command-line tool to communicate with a device (an emulator or a connected Android device) providing access to a unix shell. 19

**AJAX**  Method of exchanging data with a server, and updating parts of a web page without reloading the entire page. 51

**AP**  Access Point. 25

**API**  Application Programming Interface. 16, 22, 27, 29, 34, 45

**CN**  Common Name. 37, 38

**DH**  The Diffie Hellman (DH) algorithm allows to exchange a shared secured key over an insecure channel. It was named after Whitfield Diffie and Martin Hellman. 31

**DNS**  Domain Name System. 59

**HPKP**  HTTP Public Key Pinning. 40

**IP**  Internet Protocol. 42

**JSON**  Javascript Object Notation. 51, 53

**Local Storage**  Feature of the HTML 5 norm allowing web application to store data in the memory of the Web Browser. It is an alternative to cookies that need to be included in every server request, and only offer a small amount of memory. On the other hand with Local Storage, the data does not move from the client machine and storage limit is larger: 5 MB vs 4 096 B for cookies. Ref: [https://www.w3schools.com/html/html5_webstorage.asp](https://www.w3schools.com/html/html5_webstorage.asp). 49, 50

**MITM**  Man In The Middle. 33, 42

**NAT**  Network Address Translation. 30

**NDK**  The Native Development Kit (NDK) is a toolset to implement parts of an application using native-code languages such as C and C++. This can help you reuse code libraries written in those languages. 19

**NIC**  Network Interface Controller. 15, 22

**RSA**  The RSA algorithm is an asymmetric cryptographic system. It was named after Ron Rivest, Adi Shamir, and Leonard Adleman. 31
SAN  The Server Name Indication is an optional field of an X509 certificate to include additional identities to be bound to the subject of the certificate (email addresses, DNS names, IP addresses). These identities are then allowed to present this certificate. 37

SNI  The Server Name Indication is an optional field of the client hello message sent during a TLS handshake that allows the client to indicate the domain name it wants to reach. This way a same server can present multiple certificates on a given IP address with a given port. 37

SQL  Structured Query Language. 9, 52–55

SSL  The Secure Socket Layer is a cryptographic protocol that provides authentication, and confidentiality to internet connections, originally created by Netscape. The versions of this protocol are SSLv1 (never released,) SSLv2 (1994) and SSLv3 (1995). 9, 33, 37, 39, 57

TCP  Transmission Control Protocol. 33, 35, 37–39, 42

TLS  The Transport Layer Security protocol is the evolution of SSLv3. The versions of this protocol are TLSv1.1 (2002) TLSv1.2 (2006) and TLSv1.3 (2014). TLSv1.3 is still a draft. 9, 33, 37–40, 53, 57

UDP  User Datagram Protocol. 35, 39, 42

VPN  Virtual Private Network. 3, 9, 11, 16, 18, 19, 22, 23, 25–27, 29–31, 33, 39, 57

WLAN  Wireless Local Area Network. 25