Project phase 2

WPI CS4516   Spring 2019   D term

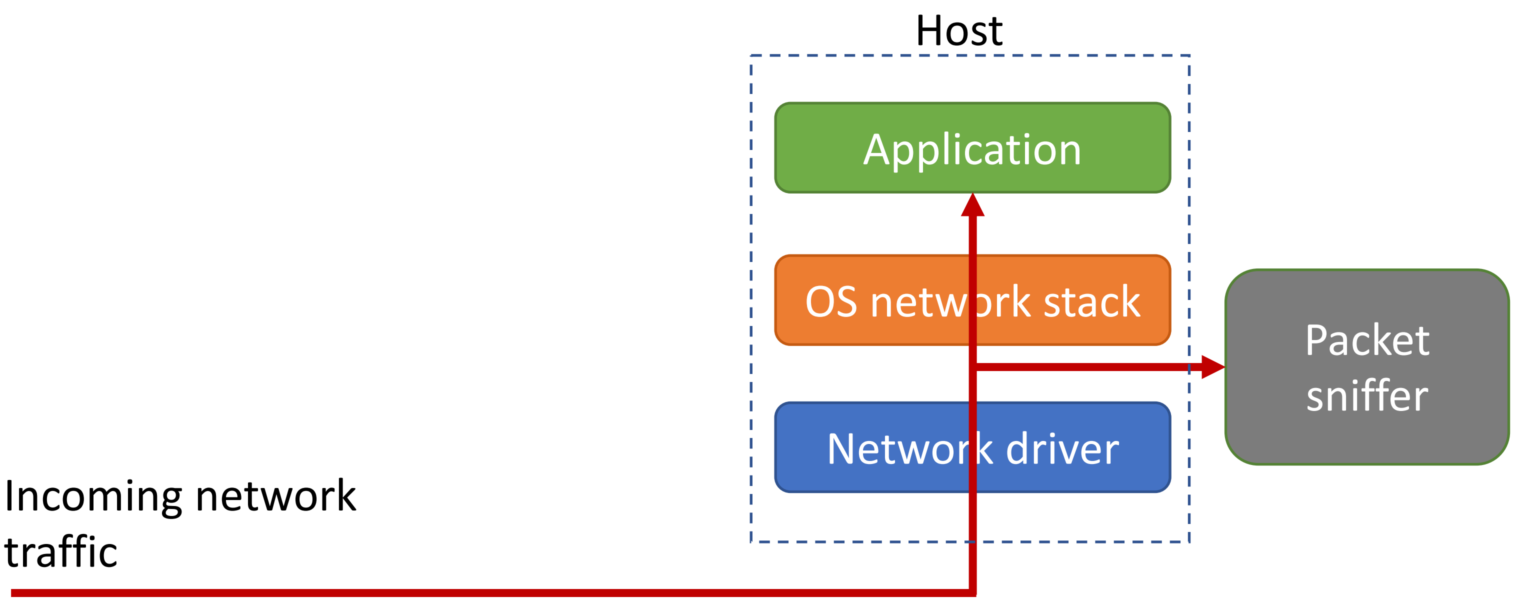
Instructor: Lorenzo De Carli (ldecarli@wpi.edu)

Tutorial summary

1. Packet sniffing
2. Deliverable specifications

Packet sniffing

- Funky term to signify the act of capturing network packets before they reach the OS network stack



Packet sniffing - II

- How does it work?
  - Typically uses some OS-level primitive to get access to packet data (e.g., raw sockets in Linux)
  - Can get more complicated (e.g., necessitates kernel driver in Windows)

Packet sniffing tools

- **Libpcap:** packet sniffing library. Pretty much the industry standard for capturing raw packets
- Consistent packet sniffing API regardless of the implementation
- Binding for lots of languages

Packet sniffing tools - II

- Libpcap offers an API for capturing packets but not much else
- Various tools provide an interface towards libpcap, plus various types of functionality
  - **tcpdump:** granddaddy of packet sniffing tools. Developed (like the original libpcap) at LBL. Command-line interface to libpcap
  - **wireshark:** GUI-based application for packet sniffing and protocol analysis. Performs protocol parsing, session analysis, etc.

Capture filters

- Sniffing all packets transiting through the network driver is typically not very interesting and **overwhelming** for a protocol analyzer
- Oftentimes, before beginning a packet capturing session the operator specifies a **filter** defining **which packets should be captured**
- Filters are specified in Berkeley Packet Filter (BPF) syntax

BPF what?

- The Berkely Packet Filter is a **packet filtering language** and a **kernel-level mechanism** to execute those filters
- Most implementations (including the original one) **compile filters to bytecode** and execute the bytecode in a in-kernel virtual machine
- Modern implementations may use **just-in-time compilation** (JIT) to compile filters to native code in real time
- **Bottom line:** filters are usually very efficient (more than capturing all packets and trying to do the filtering yourself)

Some examples of capture filters

- host 203.0.113.50
- dst host 198.51.100.200
- ether host 00:00:5E:00:53:00
- udp and src port 2005
- Host 203.0.113.50 and udp port 2005

(from https://docs.extrahop.com/7.2/bpf-syntax/)

Let’s have an demo, shall we?

Libpcap and you

- Before doing anything, you'll need to **install libpcap on TinyCore Linux**
- You may also want to install **tcpdump** (to make sure packet sniffing works)
- Then, you'll need to write **packet sniffing code**
- Suggested Python packages:
  - Pcapy (probably the most mature)
  - Pyshark
  - Pypcap

Parsing packets

- Once you have captured some packets, you'll need to make sense of them
- Approaches:
  - Use **dpkt** (or other similar Python packages) which offer primitives for **easily converting a blob of bytes into a structured packet**
  - Use a packet sniffing package like pyshark, which also includes **protocol dissectors**

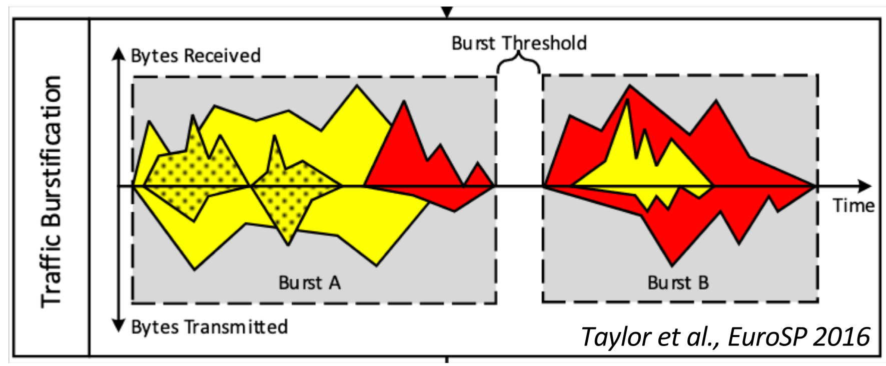
Project phase 2 - goals

Packet analysis

- You must develop code that analyzes flows to/from the Android VM
- Flow is defined as a set of packets between the same *source IP, source port, dest IP, dest port, protocol (TCP/UDP)*
- You will need to compute and print per-flow statistics
  - *Source IP, source port, dest IP, dest port, protocol, #packets sent, #packets received, #bytes sent, #bytes received*

When to log flows

- Your code must print flow statistics every time a *packet burst* concludes
- Packet burst definition:



  A packet burst includes all packets transmitted and received between the end of the previous burst, and a period of time when no packet is transmitted/received for 1s

- In other words, segment the flow of captured packets in sections separated by 1s of silence

Log format & other details

- You must create an executable named `logFlows` in `/home/tc`
- When executed, `logFlows` must capture packets and print a list of flows identified within every burst, with every entry as:

```
<timestamp> <src addr> <dst addr> <src port> <dst port> <proto>\
<#packets sent> <#packets rcvd> <#bytes send> <#bytes rcvd>
```

(Note: one line per flow)

- Logging should continue until the program is terminated

Deadlines

- Previous phase due 3/25
- This phase due 4/1

Spoiler

If you are curious to know where the project is going, read the assigned reading for the lecture on traffic classification:

2016 IEEE European Symposium on Security and Privacy

AppScanner: Automatic Fingerprinting of Smartphone Apps From Encrypted Network Traffic

Vincent F. Taylor*, Riccardo Spolaor†, Mauro Conti† and Ivan Martinovic*

*Department of Computer Science
University of Oxford, Oxford, United Kingdom
{vincent.taylor, ivan.martinovic}@cs.ox.ac.uk

†Department of Mathematics
University of Padua, Padua, Italy
{riccardo.spolaor, conti}@math.unipd.it

***Abstract*—Automatic fingerprinting and identification of smartphone apps is becoming a very attractive data gathering technique for adversaries, network administrators, investigators and marketing agencies. In fact, the list of apps installed on a device can be used to identify vulnerable apps for an attacker to exploit, uncover a victim's use of sensitive apps, assist network planning, and aid marketing. However, app fingerprinting is complicated by the vast number of apps**

Smartphones are well-equipped out of the box, but users regularly download and install add-on applications, called apps, to introduce additional features and functionality. The intense demand for smartphones, and rapid increase in app usage, makes the mobile platform a prime target for any individual or organisation looking to identify the presence of specific apps on users' smartphones, whether for benevolent or malevolent reasons.