Network Traffic Analysis

CS4516
Team 18 Cole Winsor & Daniel McDonough
4/29/19

Overview

Over the course of 7 weeks we were tasked to implement a virtual local network, and create tools for traffic analysis on that network. The network was an Android virtual machine connecting to the internet using a TinyCore virtual machine (VM) as a gateway. In order to analyze the traffic, packet sniffing tools were used to log traffic through the network and a python script classified these network traces using a neural network.

Phase 1

Before being able to read, analyze, and classify network traffic, there needs to first be a network in place. The goal of this phase was to create this network by creating two virtual machines, one as an endpoint computer and one as as a gateway to the internet. In this phase, we implemented the endpoint as an Android x86 VM and the gateway as a TinyCore Linux VM. This simulated a user on their phone, and how their network data could be tracked through a gateway. Using Oracle's VirtualBox, we were given an iso file to set up the Android x86 VM and an ova file for TinyCore VM that will act as the gateway. Of course, these files were not the only part of the set up, as there were other factors than needed to be taken into consideration.

When setting up the Android x86 VM, we made sure that it contained version 6 of the Android operating system (OS), 1024 MB of memory, and a 8GB virtual disk that was dynamically allocated. At this point the Android VM did not have a connection to the internet as we still had to implement the gateway, in the TinyCore VM.

TinyCore is a small kernel (11MB) that represents the size and limitations of a gateway in the real world. As a small OS, TinyCore only offers VI as a system to edit and create files. As we have acclimated to our own text and code editors we have to enable SSH on TinyCore to allow the passing of files between the host computer and the VM. As a second level connection to pass files we also enabled git on the TinyCore VM so we could quickly share and manage scripts between each other. As TinyCore is a rapid access memory (RAM) based filesystem, all changes had to be made and saved to run from start up.

In order to enable the TinyCore VM to serve an IP gateway, we had to first configure the eth1 interface. To do this, we had to edit the bootlocal.sh file to run a script that first killed anything on the eth1 interface if existing already. This script then configured the eth1 interface via ifconfig to change the static IP to 192.168.12.1 with a netmask of 255.255.255.0 which is broadcasted over 192.168.12.255. We then enabled IP forwarding by sysctl and changing the option to 1. Then we used udhcpd to configure the eth1 interface that faces outwards toward the Android VM. The start and end of the IP lease block for udhcpd, is a block of 100 addresses from 192.168.12.(100 - 200). Then, udhcpd connects to 255.255.255.0 as a subnet and the router and dns of 192.168.12.1. These should and do match the the eth1 ifconfig interface as the domain is local. We also added a 5 day lease for these port connections. Finally, iptables used masquerading to allow the connection between interfaces eth1 and eth0.

To test that these network configurations were correct, and working, we had to check the VirtualBox internal network configuration. The Android VM adapter should be on the internal network “intnet” and the the gateway VM should be on the same network on its second adapter. Once done, the Android VM was able to connect to the internet through the gateway and immediately started updating its default applications.

Phase 2

At this point be began the live packet capturing process, or packet sniffing. Here we capture network data before they reach the operating system network stack, and right after the data has been received by the network driver. To accomplish this task, we used Python 3.6, Libpcap, and Pyshark. Libpcap, a packet sniffing library, enables binding of other languages (such as python) to allow use of its packet capturing API. This allows us to use Pyshark to capture packets on the TinyCore VM. Of course by installing all of these, the size of the gateway VM increased. These packages were necessary however to enable this task to be done fairly quickly given the time constraints, as Pyshark contained all the features of capturing as well as parsing packets.

The goal of this phase was network traffic. To do this, the traffic was split into bursts and flows, and flows within each burst were analyzed for statistics and printed. A burst is a period of traffic which begins and ends with at least 1 second of no traffic. Within a burst, we define flows as a unique source port and ip address as well as destination port and ip address. If the same source and destination is part of a different burst it is still part of a different flow. While running, the application captures packets from the Andriod VM converting it into its flow components. If there has been less than a second since the last packet, the flow is passed into the current burst to be saved. Otherwise, a new burst is created and the current flow is saved in the new burst. In addition, the previous burst logs it’s statistics on each of its flows including the time of the burst, source address, source port, destination address and port, the protocol used, the number of packets sent and received, and the number of bytes sent and received.

In our code we use a class based structure and command line interface as a model. Here we had our main BurstLogger class start a live capture session using pyshark. When a new packet is captured, the packet is converted into a Flow object. The Flow class holds the source ip and port, destination ip and port, and protocol information for a given flow. In addition, it has some class functionality, such as the __eq__ and __str__ functions implemented for ease of use. The Burst class contains the information for each burst, including its start time and all of it’s flows. It includes functionality to add flows and get a string representation of the burst itself. In addition, it internally organizes each flow into a FlowStat object which aggregates statistics about each flow.

Phase 3

During this phase we introduced the artificial intelligence (AI) component to analyze and classify flows. Here we used TShark to capture traces of traffic going through the gateway VM. We used a Support Vector Machine (SVM) classifier with the RBF kernel in order to classify 5 different apps that were installed on the Android VM. SVM was chosen as it separates the given N separable classes by the largest possible margin. We believed that based on chosen vectors we would be able to separate and classify packets. For this phase we chose seven of the most prominent features of a packet to be classified. These features were the number of packets sent and received, the size of those packets, the standard deviation on the sizes and the overall number of packets sent both ways.

Of course to classify anything we need data. We obtained 50 pcap files for each for the 5 apps; Youtube, Google News, Weather Channel, Fruit Ninja and the default browser. Each of these apps were installed on the Android VM and had to run at least once to set up correctly. The Weather Channel, Browser and Google News apps specifically had to be set up before recording classifications. The browser's homepage had to be reset to the Wikipedia homepage for consistency between colleague data. The other two apps had some set-up to do, enorder to start the base application. Here, the location was set to Worcester Massachusetts, and a new gmail and google account were created.

In order to compare classifiers, we incorporated a feature to look at different trained classifiers by saving a pkl file that held the pre trained classifier. In our program we added a command line interface with several commands. By default the program expects a Pcap file as a argument to read from and classify it. Otherwise, the program takes options -c, -b, and -t. The first option -c, creates a new classifier save file, overwriting the old classifier. Options -t and -b train and test the classifier respectively.

When collecting network data for each of the apps, the tcpdump packet sniffer was started prior to the start of the decided app. The sniffer was to only detect TCP and UDP protocol packets. The sniffer was only stopped manually after the app fully loaded and there were no print logs for an extended period of time (longer than 1 second). Of course this was not the best method as some apps produced more bursts than others, producing unstandardized data. This produced 250 pcap files, 70% of these for training and 30% for testing. Overall this produced a 60% accurate classifier, via f-score.

Phase 4

In this phase, we incorporated the live packet sniffer from phase 2 and applied it to the classifier from phase 3. We added, the live classification feature to the command interface under -l. With the live classification, all network flows that are passed through the gateway VM are classified as one of the apps as if they were being loaded at startup. It was here that we noticed that the training data for classification for static pcap files were not applicable to live classification. This was because for many of the applications, such as youtube and the browser, the landing pages changed, so the captured traffic was less applicable when looking at the new pages. It was then we redid the data collection of all 250 pcap files, and retrained the classifier with this new data.

During live classification, a new class, LiveClassifier, handles capturing the traffic and dividing it into bursts. While running, each burst is printed like in phase 2, it is printed after the burst has been fully completed. In addition to the information printed out in phase 2, the application also uses the classifier to analyze which application created that traffic and prints this guess. Overall, the object oriented structure of our program made it easy to add this new feature.

Lessons Learned and Limitations

Through this project, we learned how vulnerable our network traffic is. If not encrypted properly, anyone is able to view data sent over and use deep packet inspection to analyze people's internet patterns. Simply from just the timestamp, people can produce classifiers that may accurately depict your data, removing anonymity between data and the user. This is a real wake up call for taking proper measurements for network security.

This project had several limitation that held it back from working properly. One limitation was our classifier. One problem is that when the SVM cannot property classify flows, it takes the best guess at picking one of the five apps rather than classifying it as "unknown". Similarly, we the lack of vector features and training data may have resulted in poor classification. We also feel that the use of SVM may have been a mistake and that random forest may have been a more appropriate choice. In addition, we had many problems with the live classification. Our analysis showed that for our capture files we had roughly 60% accuracy, but when using the live classifier, it is much more inaccurate. We were not able to figure out why this was the happening, but even after regathering data, we were still unable to get the accuracy of live classification up. Overall, we felt that our lack of background with machine learning may have been what held us back the most.