Chapter 3: Learning The “Hello World” Of Security Data Analysis

“from one thing, know ten thousand things”

― Miyamoto Musashi, The Book of Five Rings: Miyamoto Musashi

If you’ve ever tried to learn a new programming language there’s a good chance you started of with a “Hello World” example that quickly introduces basic language structure and code execution. The immediate sense of accomplishment as the syntax is verified by the compiler/interpreter and the familiar two-word output is displayed becomes a catalyst for the notion that, soon, you shall have the ability to bend this new language to your will.

This chapter takes the “Hello World” concept and expands it to a walk-through of a self-contained, introductory security data analysis use case that you will be able to follow along with, execute and take concepts from as you start to perform your own analyses. There are parallel examples in *Python* and *R* to provide a somewhat-agnostic view of the similarities, strengths and differences between both languages in a real life data analysis context. If you’re not familiar with one or both of those languages you should read Chapter 2 first and at least skim some of the external resources referenced there. This is a good place to reinforce the recommendation to use *IPython Notebooks* or *RStudio* for your analyses and exploration as they provide very robust and forgiving environments and each will be far more optimal then saving and executing scripts. Remember, all the source code, sample data and visualizations are on the book’s web site, so no need for transcription, just cut/paste and focus on the flow of and concepts presented in the examples.

Preparing For Analysis

Before jumping into data retrieval and analysis, we need to setup an area where we can organize all our input data, analysis scripts, output (visualizations, reports and/or data) and any supporting documentation. For the purposes of this chapter, we’ll be using the following directory structure:

/book/ch3

|-*R*

|-data

|-docs

|-output

|-python

|-support

|-tmp

Like most elements of programming, there is no one, true way to setup this structure, but you should strive to find one that works for you and stick with it. A great way to do that is to take a lesson from modern web framework builders and use a simple setup script that builds the structure for you:

Sample analysis preparation script

#!/bin/sh

#

# prep: prep analytics directory structure

#

# usage: prep DIRNAME

#

DIR=$1

if [ ! -d "${DIR}" ]; then

mkdir -p ${DIR}/*R* \

${DIR}/data \

${DIR}/docs \

${DIR}/output \

${DIR}/python \

${DIR}/support \

${DIR}/tmp

> ${DIR}/readme.md

ls -lR ${DIR}

else

echo "Directory "${DIR}" already exists"

fi

You now only need to type “prep NAME” whenver you want to start a new project (so, for this project, “prep ch3”). As you develop your own styles and patterns, you can expand this script to include the generation of various templates and even initialization of source code repositories. Once the structure is in place, it’s time to retrieve, explore and analyze some data.

Getting Data

We are living in a silver age of data in information security. The challenge is no longer where to get data from, but what to do with it. And, the kind of information in each data set will drive the type of research you perform.

For this use case, we’ll be working with AlienVault’s IP Reputation Database (<http://labs.alienvault.com/labs/index.php/projects/open-source-ip-reputation-portal/download-ip-reputation-database/>). AlienVault develops OSSIM—an open source security information manager—and a proprietary unified threat management (UTM) product, both of which make use of this freely available data set that contains information on various types of “badness” across the internet. AlienVault provides this data in numerous formats free of charge and the version we’ll be working with is the OSSIM Format (<http://reputation.alienvault.com/reputation.data>) since it provides the richest information of all the available formats.

type="tip"

AlienVault updates their IP reputation data set hourly and produces a companion “revision” file (<http://reputation.alienvault.com/reputation.rev>), enabling you to ensure you are working with the latest data set or keep a history of data sets. There is additional code on the companion web site that shows how to perform this check to see if it’s time to download a new one.

When performing a one-off, exploratory analysis or getting a first look at a data set, it’s acceptable to just do a quick download via browser. If we do that for the AlienVault IP reputation database and examine the first few data elements we can get an idea of the contents and format, which will come in handy when we start to read in and work with the data. Here, we use some simple Linux/UNIX commands to inspect the download:

Performing a quick review of the downloaded data set

$ **head -10 reputation.data** *# look at the first few lines in the file*

222.76.212.189#4#2#Scanning Host#CN#Xiamen#24.479799270,118.08190155#11

222.76.212.185#4#2#Scanning Host#CN#Xiamen#24.479799270,118.08190155#11

222.76.212.186#4#2#Scanning Host#CN#Xiamen#24.479799270,118.08190155#11

5.34.246.67#6#3#Spamming#US##38.0,-97.0#12

178.94.97.176#4#5#Scanning Host#UA#Merefa#49.823001861,36.0507011414#11

66.2.49.232#4#2#Scanning Host#US#Union City#37.59629821,-122.0656966#11

222.76.212.173#4#2#Scanning Host#CN#Xiamen#24.479799270,118.08190155#11

222.76.212.172#4#2#Scanning Host#CN#Xiamen#24.479799270,118.08190155#11

222.76.212.171#4#2#Scanning Host#CN#Xiamen#24.479799270,118.08190155#11

174.142.46.19#6#3#Spamming###24.4797992706,118.08190155#12

$ **wc –l reputation.data** *# see how many total records there are*

258626 reputation.data

For most projects it’s better to get into the habit of retrieving the data source directly from your analysis scripts. If you still prefer to download files manually you should provide some type of comment in your programs that provides details on where the source data comes from and when you retrieved the data for your current analysis to make it easier to repeat the analyses at a later date.

The following examples show how to perform the data retrieval in both *R* and *Python*. If you are following along with *RStudio* or *IPython*, all code examples assume a working directory of the top level of the project structure (e.g. executing in the “ch3” directory from the prep example). Code blocks are mostly self-contained, but each will expect this first snippet and the one in ‘Reading In Data’ to have been executed in the running *RStudio* or *IPython* session.

R code to download the AlienVault data

# URL for the AlienVault IP Reputation Database (OSSIM format)

# storing the URL in a variable makes it easier to modify later

# if it changes

avURL <- "http://reputation.alienvault.com/reputation.data"

# use relative path for the downloaded data

avRep <- "data/reputation.data"

# using an if{}-wrapped test with download.file() vs read.xxx()

# directly avoids having to re-download a 16MB file every time

# we run the script

if (file.access(avRep)) {

download.file(avURL, avRep)

}

Python code to download the AlienVault data

#!/usr/bin/python

#

# reputation.py

# sample analysis script for AlienVault IP Reputation Database data

# URL for the AlienVault IP Reputation Database (OSSIM format)

# storing the URL in a variable makes it easier to modify later

# if it changes

import urllib

import os.path

avURL = "http://reputation.alienvault.com/reputation.data"

# relative path for the downloaded data

avRep = "data/reputation.data"

# using an if-wrapped test with urllib.urlretrieve() vs direct read

# via panads avoids having to re-download a 16MB file every time we

# run the script

if not os.path.isfile(avRep):

urllib.urlretrieve(avURL, filename=avRep)

The *R* and *Python* code look very similar and follow the same basic structure: using variables whenever possible for URL and filenames plus testing for the existence of the data file before downloading it again. These are good habits to get into and we’ll be underscoring other suggested good practices throughout the rest of the book.

With the IP reputation data in hand, it’s now time to read in the data so we can begin to work with it.

Reading In Data

*R* and *Python* (especially with *pandas*) abstract quite a bit of complexity when it comes to reading and parsing data into structures for processing. *R*’s read.table(), read.csv() and read.delim() and *pandas* read\_csv() will cover nearly all your delimited file reading needs and provide robust configuration options for even the most gnarly input file. Both tools, as we’ll see in later chapters, also provide ways to retrieve data from SQL and “NoSQL” databases, HDFS “big data” setups and even handle unstructured data quite well.

type="general"

The Revolution Will Be [Tab|Comma]–Separated!

Base R and Python’s pandas package both excel at reading in delimited files. While they are also both agnostic when it comes to what that delimiter is, there is a general acceptance in the data science community that it should either be a comma (CSV) or a tab (TSV) character and the majority of the sample data sets available to practice with come in one of those two flavors. This format is thoroughly defined in RFC 1480 (http://www.rfc-editor.org/rfc/rfc4180.txt) and has the following high-level attributes:

* One record per line
* An optional header line
* Header and data rows have fields separated by commas (or tabs)
* Each line should have the same number of fields
* Spaces in fields should be treated as significant

There are a large number of tools in the security domain that can import and export CSV-formatted files and, if you intend to do any work in environments like Hadoop, you will *need* to become familiar with CSV and especially TSV.

Another established format is JSON (JavaScript Object Notation), which has grown to become the preferred way to transport data between servers and browsers. It is also the foundational data format behind many NoSQL database environments/tools. The JSON format is defined in RFC 4627 (http://www.rfc-editor.org/rfc/rfc4627.txt) and has two primary structures:

* A collection of name/value pairs (e.g. “a dictionary”)
* An ordered list of values (e.g. an “array”)

JSON enables richer and more complex data representation than CSV/TSV and is rapidly superseding another popular, structured format—the Extensible Markup Language (XML)—as the preferred *data exchange* representation since it is syntactically less verbose, much easier to parse and (usually) more readable. XML has and will continue to excel at document representation, but you should strongly consider using JSON for your structured data processing needs.

From our cursory examination of the downloaded file, we can see the AlienVault data has a fairly straightforward record format with eight primary fields using a “#” as the field separator/delimiter.

222.76.212.189#4#2#Scanning Host#CN#Xiamen#24.479799270,118.08190155#11

We should also notice that the reputation data file lacks the optional header, each example code segment assigns more meaningful column names manually. This is a completely optional step, but it will help avoid confusion as you expand your analyses and, as we’ll see in later chapters, help build consistency across data frames if you bring in additional data sets.

The consistency in the record format makes the consumption of the data equally as straightforward in each language.

R code to read in the AlienVault data

# read in the IP reputation db into a data frame

# this data file has no header, so set header=FALSE

av <- read.csv(avRep,sep="#", header=FALSE)

# assign more readable column names since we didn’t pick

# any up from the header

colnames(av) <- c("IP", "Reliability", "Risk", "Type",

"Country", "Locale", "Coords", "x")

# get an R overview of the data frame with str()

**str(av)**

'data.frame': 258626 obs. of 8 variables:

[AU/ED: Please change the tab(s) in the above line to spaces.]

[AU/ED: Please change the tab(s) in the above line to spaces.]

$ IP : Factor w/ 258626 levels "1.0.232.167",...

$ Reliability: int 4 4 4 6 4 4 4 4 4 6 ...

$ Risk : int 2 2 2 3 5 2 2 2 2 3 ...

$ Type : Factor w/ 34 levels "APT;Malware Domain",...

$ Country : Factor w/ 153 levels "","A1","A2","AE",...

$ Locale : Factor w/ 2573 levels "","Aachen","Aarhus",...

$ Coords : Factor w/ 3140 levels ...

$ x : Factor w/ 34 levels "1;6","11","11;12",...

# take a quick look at the first few rows of data

**head(av)**

IP Reliability Risk Type Country Locale

1 222.76.212.189 4 2 Scanning Host CN Xiamen

2 222.76.212.185 4 2 Scanning Host CN Xiamen

3 222.76.212.186 4 2 Scanning Host CN Xiamen

4 5.34.246.67 6 3 Spamming US

5 178.94.97.176 4 5 Scanning Host UA Merefa

6 66.2.49.232 4 2 Scanning Host US Union City

Coords x

1 24.4797992706,118.08190155 11

2 24.4797992706,118.08190155 11

3 24.4797992706,118.08190155 11

4 38.0,-97.0 12

5 49.8230018616,36.0507011414 11

6 37.5962982178,-122.065696716 11

Python code to read in the AlienVault data

import *pandas* as pd

# read in the data into a *pandas* data frame

av = pd.read\_csv(avRep,sep="#")

# make smarter column names

av.columns = ["IP","Reliability","Risk","Type","Country",

"Locale","Coords","x"]

**print(av)** *# take a quick look at the data structure*

<class 'pandas.core.frame.DataFrame'>

Int64Index: 258625 entries, 0 to 258624

Data columns (total 8 columns):

IP 258625 non-null values

Reliability 258625 non-null values

Risk 258625 non-null values

Type 258625 non-null values

Country 248570 non-null values

Locale 184555 non-null values

Coords 258625 non-null values

x 258625 non-null values

dtypes: int64(2), object(6)

**av.head()** *# take a look at the first 10 rows*

IP Reliability Risk Type Country Locale

0 222.76.212.185 4 2 Scanning Host CN Xiamen

1 222.76.212.186 4 2 Scanning Host CN Xiamen

2 5.34.246.67 6 3 Spamming US NaN

3 178.94.97.176 4 5 Scanning Host UA Merefa

4 66.2.49.232 4 2 Scanning Host US Union City

5 222.76.212.173 4 2 Scanning Host CN Xiamen

6 222.76.212.172 4 2 Scanning Host CN Xiamen

7 222.76.212.171 4 2 Scanning Host CN Xiamen

8 174.142.46.19 6 3 Spamming NaN NaN

9 66.2.49.244 4 2 Scanning Host US Union City

Coords x

0 24.4797992706,118.08190155 11

1 24.4797992706,118.08190155 11

2 38.0,-97.0 12

3 49.8230018616,36.0507011414 11

4 37.5962982178,-122.065696716 11

5 24.4797992706,118.08190155 11

6 24.4797992706,118.08190155 11

7 24.4797992706,118.08190155 11

8 24.4797992706,118.08190155 12

9 37.5962982178,-122.065696716 11

*IPython* *Notebook*s also have a useful set of functions to output data to a more viewer-friendly HTML format:

IPython code to display head() as an HTML table

from *IPython*.display import HTML

HTML(av.head(10).to\_html())

Figure 3-1 *IPython* HTML head() Output [793725c03f001.png]

Exploring Data

Now that we have a general idea of the variables and how they look, it’s now time to bring your security domain expertise into the mix to explore and discover what is interesting about the data and enable us to form good questions to ask and answer. Despite having almost 260,000 records, we have many tools at our disposal to help get a feel for what it contains.

But before we go any deeper into the data, there are some tidbits of information we know about the data and let’s recap that here:

* Reliability, Risk and “x” are integers
* IP, Type, Country, Locale and Coords are character strings
* The IP address is stored in the dotted-quad notation, not hostnames or decimal format
* Each record is associated with a unique IP address, so there are 258,626 IP addresses (in this download)
* Each IP address has been pre-geo-located into the latitude and longitude pair in the Coords field, but they are in a single field separated by a comma. We will have to parse that further if we want to use that field.

When we have quantitative variables (which is a fancy way to say “numbers representing a quantity”), a good first exploratory step is to look at the basic *descriptive statistics* on the variables. These are comprised of:

* *minimum* and *maximum* values; taking the difference of these will give us the *range* (*range* = *max* - *min*)
* *median* (the value at the middle of the data set)
* *1st* and *3rd quartiles* (the 25th and 75th percentiles, or you could think of it as the median value of the first and last halves of the data, respectively)
* *mean* (sum of all values divided by the number of count)

You may see the min, max, median and quartiles referred to as the *five number summary* of a data set (as developed by Tukey), and both languages have built-in functions to calculate them, along with the mean. We’ll look at the summary on our two primary numeric columns: *Reliability* and *Risk*.

R code to look at the central tendency of Reliabilty and Risk

**summary(av$Reliability)**

Min. 1st Qu. Median Mean 3rd Qu. Max.

1.000 2.000 2.000 2.798 4.000 10.000

**summary(av$Risk)**

Min. 1st Qu. Median Mean 3rd Qu. Max.

1.000 2.000 2.000 2.221 2.000 7.000

Python code to look at the central tendency of Reliabilty and Risk

**av['Reliability'].describe()**

count 258625.000000

*mean 2.798036*

std 1.130419

*min 1.000000*

25% 2.000000

50% 2.000000

75% 4.000000

*max 10.000000*

**av['Risk'].describe()**

count 258625.000000

*mean 2.221363*

std 0.531572

*min 1.000000*

25% 2.000000

50% 2.000000

75% 2.000000

*max 7.000000*

From an examination of the above results, we make a note that the *Reliability* column spreads across the *documented* potential range of [1…10] but the *Risk* column—which AlienVault says has a documented potential range of [1…10]—only has a spread of [1…7]. We can also see that both *Risk* and Reliability appear to heavily gravitate towards a value of “2”

We can now dig a bit deeper and use the fact that the “*Reliability*”, “*Risk*”, “*Type*” and “*Country*” fields are actually the equivalent of categorical data: i.e. they enable slicing the data set into groups. Even though we just treated “Reliability” and “Risk” as numbers they actually are categorical, meaning each entry will be assigned an integer and a value of “4” is not necessarily twice that of “2”, in other words, the number is more a label then a measurement. Categorical data may also be referred to as nominal values, factors or in some cases, qualitative variables. Within *R*, the difference between the two will automatically be handled by the summary()function, and it will display the count for each category. This will not work on the quantitative variables though, in order to get a count of those, we could use the table() command if there are not too many unique values in the variable. Within *Python*, we will create a short function that leverages *pandas* to convert a data frame column (which is just an array) into a very appropriately named Categorical object which we can tweak a bit to give us similar helpful output.

R exploratory code for AlienVault data

**table(av$Reliability)**

1 2 3 4 5 6 7 8 9 10

5612 149117 10892 87040 7 4758 297 21 686 196

**table(av$Risk)**

1 2 3 4 5 6 7

39 213852 33719 9588 1328 90 10

# summary sorts by the counts by default

# maxsum sets how many factors to display

**summary(av$Type, maxsum=10)**

Scanning Host Malware Domain

234180 9274

Malware IP Malicious Host

6470 3770

Spamming C&C

3487 610

Scanning Host;Malicious Host Malware Domain;Malware IP

215 173

Malicious Host;Scanning Host (Other)

163 284

**summary(av$Country, maxsum=40)**

CN US TR DE NL RU GB

68583 50387 13958 10055 9953 7931 6346 6293

IN FR TW BR UA RO KR CA

5480 5449 4399 3811 3443 3274 3101 3051

AR MX TH IT HK ES CL AE

3046 3039 2572 2448 2361 1929 1896 1827

JP HU PL VE EG ID RS PK

1811 1636 1610 1589 1452 1378 1323 1309

VN LV NO CZ BG SG IR (Other)

1203 1056 958 928 871 868 866 15136

Python exploratory code for AlienVault data

# factor\_col(col)

#

# helper function to mimic *R*'s "summary()" function

# for *pandas* "columns" (which are really just *Python*

# arrays)

#

def factor\_col(col):

factor = pd.Categorical.from\_array(col)

return pd.value\_counts(factor,sort=True).reindex(factor.levels)

rel\_ct = pd.value\_counts(av['Reliability'])

risk\_ct = pd.value\_counts(av['Risk'])

type\_ct = pd.value\_counts(av['Type'])

country\_ct = pd.value\_counts(av['Country'])

**print factor\_col(av['Reliability'])**

1 5612

2 149117

3 10892

4 87039

5 7

6 4758

7 297

8 21

9 686

10 196

dtype: int64

**print factor\_col(av['Risk'])**

1 39

2 213851

3 33719

4 9588

5 1328

6 90

7 10

dtype: int64

**print factor\_col(av['Type'])**

APT;Malware Domain 1

C&C 610

C&C;Malware Domain 31

C&C;Malware IP 20

C&C;Scanning Host 7

...

Spamming 3487

Spamming;Malware Domain 5

Spamming;Malware IP 4

Spamming;Scanning Host 24

dtype: int64

**print factor\_col(av['Country'])**

A1 267

A2 2

AE 1827

AL 4

AM 6

...

VN 1203

YE 2

ZA 573

ZM 1

ZW 3

Length: 152, dtype: int64

These numerical tables help us discern the makeup of the data, but a picture has the potential to provide a whole new perspective, often times giving insights that numbers alone cannot reveal. We’ll start with a simple bar chart to get a very quick visual overview of the *Country*, *Reliability* and *Risk* factors.

R code for visualizing portions of AlienVault data

**library(ggplot2)**

# Bar graph of counts (sorted) by Country (top 20)

# get the top 20 countries' names

**country.top20 <- names(summary(av$Country))[1:20]**

# give ggplot a subset of our data (the top 20 countries)

# map the x value to a sorted count of country

**g <- ggplot(data=subset(av,Country %in% country.top20),**

**aes(x=reorder(Country, Country, length)))**

# tell ggplot we want a bar chart

**g <- g + geom\_bar()**

# ensure we have decent labels

**g <- g + labs(title="Country Counts", x="Country")**

# rotate the chart to make this one more readable

**g <- g + coord\_flip()**

# display the image

**print(g)**

# Bar graph of counts by Risk

# note we can call ggplot and add the bar chart in one line

**g <- ggplot(data=av, aes(x=Risk)) + geom\_bar()**

# force an X scale to be just the limits of the data

# and to be discrete vs continuous

**g <- g + scale\_x\_discrete(limits=seq(max(av$Risk))**

**g <- g + labs(title="'Risk' Counts", x="Risk Score")**

**print(g)**

# Bar graph of counts by Reliability

**g <- ggplot(data=av, aes(x=Reliability)) + geom\_bar()**

**g <- g + scale\_x\_discrete(limits=seq(max(av$Reliability)))**

**g <- g + labs(title="'Reliabiity' Counts", x="Reliability Score")**

**print(g)**

Figure 3-2 Country Factor Bar Chart (*R*) [793725c03f002.eps]

Figure 3-3 Reliability Factor Bar Chart (*R*) [793725c03f003.eps]

Figure 3-4 Risk Factor Bar Chart (*R*) [793725c03f004.eps]

Python code for visualizing portions of AlienVault data

# sort by country

**country\_ct = pd.value\_counts(av['Country'])**

# plot the data

**country\_ct[:20].plot(kind='bar', rot=0,**

**title="Summary By Country")**

**factor\_col(av['Reliability']).plot(kind='bar',**

**rot=0,title="Summary By 'Reliability'")**

**factor\_col(av['Risk']).plot(kind='bar', rot=0,**

**title="Summary By 'Risk'")**

**factor\_col(av['Type']).plot(kind='bar', rot=0)**

Figure 3-5 Country Factor Bar Chart (*Python)* [793725c03f005.png]

Figure 3-6 Reliability Factor Bar Chart (*Python*) [793725c03f006.png]

Figure 3-7 Risk Factor Bar Chart (*Python*) [793725c03f007.png]

The *Country* chart shows there are definitely some countries that are contributing more significantly to the number of malicious nodes, and we can go back to numbers for a moment to look at the percentages for the top ten in the list:

R code compare country percentage makeup

# store the top 10 returned by summary() in a vector

**country10 <- summary(av$Country, maxsum=10)**

# now convert to a percentage by dividing by number of rows

**country.perc10 <- country10/nrow(av)**

# and print it

**print(country.perc10)**

**CN US TR DE NL**

**0.26518215 0.19482573 0.05396983 0.03887854 0.03848414 0.03066590**

**RU GB IN (Other)**

**0.02453736 0.02433243 0.02118890 0.30793501**

Python code compare country percentage makeup

# extract the top 10 most prevalent conuntries

top10 = pd.value\_counts(av['Country'])[0:9]

# calculate the % for each of the top 10

top10.astype(float) / len(av['Country'])

CN 0.264421

US 0.193775

TR 0.053935

DE 0.038272

NL 0.030473

RU 0.024371

GB 0.024271

IN 0.021174

FR 0.021023

dtype: float64

Our quick calculations show China and the United States account for almost 46% of the malicious nodes in the list and Russia is just 2.5%. One avenue to explore here would be to see how this compares with various industry reports since we would expect many of these countries to be in the top ten, but the amount some of them are contributing may suggest some bias in the data set. You can also see that almost 4% of the nodes cannot be geo-located. The chapter on “*Mapping Badness*” covers the challenges and pitfalls of IP address geo-location, so we’ll refrain from exploring that further here.

Looking at the “Risk” variable, the level of risk of most of the nodes is, well, *negligible*. There are other elements that stand out with this data though, foremost being that practically no endpoints are in categories 1, 5, 6 or 7, and none in the rest of the defined possible range [8-10]. We should make another note to dig a bit deeper, but there is more than a hint of bias from this perspective.

Finally, the *Reliability* rating of the nodes also appears to be a bit skewed. The values are mostly clustered in levels 2 and 4 and not many ratings above level 4. The fact that it completely skips a reliability rating of 3 should raise some questions. It could indicate a systemic flaw in the assignment of the rating, or it could be that we have at least two distinct data sets. Either way, that large quantity of 2’s and 4’s and low quantity of 3’s is something we may see if we can determine what’s going on, because it’s just a little odd.

We now have some leads to pursue and a much better idea of the makeup of the key components of the data, which should be plenty of fodder for formulating a practical question.

Asking A Question

Consider the primary use-case for the AlienVault reputation data: importing it into a SEIM or IDP/IDS environment to alert incident response team members or just log/block malicious activity. How would this quick overview of the reputation data influence the configuration of your security technologies to ensure the least number of false positives? Or, possibly more importantly, how valuable is this data set to you given the reliability and risk levels?

We’ll take a slightly more deterministic view of those questions by asking, “*which nodes from the reputation database represent a real threat?”.* There *is* a reason AlienVault included both *Risk* and *Reliability* fields, and we should be able to use these attributes to classify nodes into two categories: 1) the nodes we really care about, and 2) everything else. The definition of “really care about” can be somewhat subjective, but it is unrealistic to believe we can alert on all detected activity by one of these 260,000 nodes. Some form of prioritization triage *must* occur and it is a far better approach to base the outcome on statistical analysis versus a “gut call” or solely on “expert opinion” alone.

A good first step to answering the “which nodes do we really care about?” question is to cross-classify the nodes using the *Risk* and *Reliability* factors. This is more commonly referred to as a *contingency table*, which is a tabular view of the multivariate frequency distribution of specific variables. If that sounds a bit confusing, it will make much more sense when you review the output below. After building a contingency table, we can take both a numeric and graphical look at the results to see where the AlienVault nodes cluster.

R code for risk/reliability contingeny table generation

# compute contingency table for Risk/Reliability factors which

# produces a matrix of counts of rows that have attributes at

# each (x, y) location

**rr.tab <- xtabs(~Risk+Reliability, data=av)**

**ftable(rr.tab) # print table**

# virtually identical output to pandas (below)

# graphical view

**library(lattice)**

# cast the table into a data frame

**rr.df = data.frame(table(av$Risk, av$Reliability))**

# set the column names since table uses "Var1" and "Var2"

**colnames(rr.df) <- c("Risk", "Reliability", "Freq")**

# now create a level plot with readable labels

**levelplot(Freq~Risk\*Reliability, data=rr.df, main="Risk ~ Reliabilty",**

**ylab="Reliability", xlab = "Risk", shrink = c(0.5, 1),**

**col.regions = colorRampPalette(c("#FFFFFF", "#0868AC"))(20))**

Figure 3-8 Risk/Reliability Contingency Table Level Plot (*R*) [793725c03f008.eps]

Python code for risk/reliability contingeny table generation

# compute contingency table for Risk/Reliability factors which

# produces a matrix of counts of rows that have attributes at

# each (x, y) location

**pd.crosstab(av['Risk'], av['Reliability'])**

Reliability 1 2 3 4 5 6 7 8 9 10

Risk

1 0 0 16 7 0 8 8 0 0 0

2 804 149114 3670 57652 4 2084 85 11 345 82

3 2225 3 6668 22168 2 2151 156 7 260 79

4 2129 0 481 6447 0 404 43 2 58 24

5 432 0 55 700 1 103 5 1 20 11

6 19 0 2 60 0 8 0 0 1 0

7 3 0 0 5 0 0 0 0 2 0

# graphical view of contingency table (swapping risk/reliability)

**xtab = pd.crosstab(av['Reliability'], av['Risk'])**

**plt.pcolor(xtab,cmap=cm.Blues)**

**plt.yticks(arange(0.5,len(xtab.index), 1),xtab.index)**

**plt.xticks(arange(0.5,len(xtab.columns), 1),xtab.columns)**

**plt.colorbar()**

Figure 3-9 Risk/Reliability Contingency Table Heatmap (*Python*) [793725c03f009.png]

Figure 3-4 is a level plot and uses both size and color to show quantity whereas Figure 3-5 is a standard heatmap that relies on color alone to show quantity. With both factors combined, we are definitely starting to see bias around [2, 2] in the data set.

Just as a fun aside, we’d like to know if the patterns we are seeing are occurring by chance, or if there is some underlying patter to it. While we could do some fancy-pants statistics here and maybe apply Fisher’s exact test we don’t want to get crazy. What if we made the assumption that every value of Risk and Reliability had an equal chance of occurring (ya know, for fun), what would the level plot look like? We should expect some amount of natural variation both is the systems and the data collection process so some combinations would naturally occur more often then others. But how different would it look from the data we are looking at? In R we can use the sample() command to generate random samples and build a contingency table from randomness. Running this multiple times should produce random tables each time.

R code to generate baseline “random” sample for contingency table comparison

# generate random samples for risk & reliability and re-run xtab

# starting PRNG from reproducable point

**set.seed(1492)** # as it leads to discovery

# generate 260,000 random samples

**rsk=sample(1:7, 260000, replace=T)**

**rel=sample(1:10, 260000, replace=T)**

# cast table into data frame

**tmp.df = data.frame(table(factor(rsk), factor(rel)))**

**colnames(tmp.df) <- c("Risk", "Reliability", "Freq")**

**levelplot(Freq~Reliability\*Risk, data=tmp.df, main="Risk ~ Reliabilty",**

**ylab="Risk", xlab = "Reliability", shrink = c(0.5, 1),**

**col.regions = colorRampPalette(c("#FFFFFF", "#0868AC"))(20))**

Python code to generate baseline “random” sample for contingency table comparison

# generate random data to show the difference

**data = { 'rsk': randint(1, 7, 260000),**

**'rel': randint(1, 10, 260000) }**

**tmp\_df = pd.DataFrame(data, columns=['rsk', 'rel'])**

# compute crosstab and plot

**xtab = pd.crosstab(tmp\_df['rel'], tmp\_df['rsk'])**

**print xtab # not shown**

# plot

**plt.pcolor(xtab,cmap=cm.Blues)**

**plt.yticks(arange(0.5,len(xtab.index), 1),xtab.index)**

**plt.xticks(arange(0.5,len(xtab.columns), 1),xtab.columns)**

**plt.colorbar()**

Figure 3-10 “Unbiased” Risk/Reliability Contingency Table (*R*) [793725c03f010.eps]

Figure 3-11 “Unbiased” Risk/Reliability Contingency Table (*Python*) [793725c03f011.png]

Figure 3-6 and Figure 3-7 show two things, first, we can make some pretty and colorful random boxes with a few lines of code and second, there is definitely something pulling nodes into the lower *Risk* and *Reliability* categories. It could be because the world just has low risk and reliability or the sampling method or scoring system is introducing the skew.

Let’s turn our attention to the “Type” variable and see if we can’t establish a relationship with the “Risk” and “Reliability” ratings. Looking closely at the “Type” variable, we notice that some entries have more than type assigned to it and they are separate by a semi-colon (there are 215 “Scanning Host;Malicious Host” for example). Since we want to see how those types compare, those with a combination of types shouldn’t be mixed with other types. So rather than try to parse out the nodes with multiple types, we will just reassign all of them into a category of “Multiples” to show that they were assigned more than one type. Then we can create a three-way contingency table and see how that looks. Let’s pull in the *Type* column and see how that impacts the view.

R code to generate a 3-way risk/reliability/type contingency table

# Create a new varible called "simpletype"

# replacing mutiple categories with label of "Multiples"

**av$simpletype <- as.character(av$Type)**

# Group all nodes with mutiple categories into a new category

**av$simpletype[grep(';', av$simpletype)] <- "Multiples"**

# Turn it into a factor again

**av$simpletype <- factor(av$simpletype)**

**rrt.df = data.frame(table(av$Risk, av$Reliability, av$simpletype))**

**colnames(rrt.df) <- c("Risk", "Reliability", "simpletype", "Freq")**

**levelplot(Freq ~ Reliability\*Risk|simpletype, data =rrt.df,**

**main="Risk ~ Reliabilty | Type", ylab = "Risk",**

**xlab = "Reliability", shrink = c(0.5, 1),**

**col.regions = colorRampPalette(c("#FFFFFF","#0868AC"))(20))**

Python code to generate a 3-way risk/reliability/type contingency table

# create new column as a copy of Type column

**av['newtype'] = av['Type']**

# replace multi-Type entries with “Multiples”

**av[av['newtype'].str.contains(";")] = "Multiples"**

# setup new crosstab structures

**typ = av['newtype']**

**rel = av['Reliability']**

**rsk = av['Risk']**

# comput crosstabl making it split on the

# new “type” column

**xtab = pd.crosstab(typ, [ rel, rsk ],**

**rownames=['typ'], colnames=['rel', 'rsk'])**

**print xtab #output not shown**

**xtab.plot(kind='bar',legend=False) #output not shown**

Figure 3-12 3-Way Risk/Reliability/Type Contingency Table (*R*) [793725c03f011.eps]

They say a picture is worth a thousand words, but in this case it’s worth about 234,000 data points in the Scanning Hosts category (about 90% of the entries are classified as scanning hosts). That category is so large and generally low risk that it is overshadowing the rest of the categories. Let’s remove that from the *Type* factors and re-generate the image. This isn’t to say those aren’t important, but remember we are trying to understand which of these entries we really care about, nodes with low risk and reliability ratings are things we don’t want to be woken up from our nap for. We want to peal that away and look at the relationships that may exist underneath the scanning hosts.

R code to filter out “Scanning Host” type

# from the existing rrt.df, filter out ‘Scanning Host’

**rrt.df <- subset(rrt.df, simpletype != "Scanning Host")**

**levelplot(Freq ~ Reliability\*Risk|simpletype, data =rrt.df,**

**main="Risk ~ Reliabilty | Type", ylab = "Risk",**

**xlab = "Reliability", shrink = c(0.5, 1),**

**col.regions = colorRampPalette(c("#FFFFFF","#0868AC"))(20))**

Python code to filter out “Scanning Host” type

# filter out all “Scanning Host”s

**rrt\_df = av[av['newtype'] != "Scanning Host"]**

**typ = rrt\_df['newtype']**

**rel = rrt\_df['Reliability']**

**rsk = rrt\_df['Risk']**

**xtab = pd.crosstab(typ, [ rel, rsk ],**

**rownames=['typ'], colnames=['rel', 'rsk'])**

**print xtab** # not shown

**xtab.plot(kind='bar',legend=False)** # not shown

Figure 3-12 3-Way Risk/Reliability/Type Contingency Table without “Scanning Host” (*R*) [793725c03f012.eps]

Now we are getting somewhere. In this graphic, we can see the Malware domain *Type* has risk ratings limited to 2’s and 3’s, and the reliability is focused around 2, but spreads the range of values. We can also start to see the patterns in the other categories as well, but let’s regenerate this again after we remove the Malware domain. Also, it looks like Malware distribution does not seem to be contributing any risk. Let’s filter that out of the *Type*s out as well.

R code to filter out remaining types

**rrt.df = subset(rrt.df,**

**!(simpletype %in% c("Malware distribution",**

**"Malware Domain")))**

**sprintf("Count: %d; Percent: %2.1f%%",**

**sum(rrt.df$Freq),**

**100\*sum(rrt.df$Freq)/nrow(av))**

# this outputs:

# [1] Count: 15171; Percent: 5.9%

**levelplot(Freq ~ Reliability\*Risk|simpletype, data =rrt.df,**

**main="Risk ~ Reliabilty | Type", ylab = "Risk",**

**xlab = "Reliability", shrink = c(0.5, 1),**

**col.regions = colorRampPalette(c("#FFFFFF","#0868AC"))(20))**

Python code to filter out remaining types

**rrt\_df = rrt\_df[rrt\_df['newtype'] != "Malware distribution" ]**

**rrt\_df = rrt\_df[rrt\_df['newtype'] != "Malware Domain" ]**

**typ = rrt\_df['newtype']**

**rel = rrt\_df['Reliability']**

**rsk = rrt\_df['Risk']**

**xtab = pd.crosstab(typ, [ rel, rsk ],**

**rownames=['typ'], colnames=['rel', 'rsk'])**

**print "Count: %d; Percent: %2.1f%%" %**

**(len(rrt\_df), (float(len(rrt\_df)) / len(av)) \* 100)**

# this outputs:

# Count: 15171; Percent: 5.9%

**print xtab # not shown**

**xtab.plot(kind='bar',legend=False**) # not shown

Figure 3-13 3-Way Risk/Reliability/Type Contingency Table — Final (*R*) [793725c03f013.eps]

With this final bit of filtering, we’ve reduced the list to less than 6% of the original and have honed in fairly well on the nodes representing the ones we “really should care about”. Looking at this graphic, we can see none of the Command and Controls hosts are below a risk of 4 (nor above the risk of 5). If we wanted to further reduce the scope, we could filter by various combinations of *Reliability* and/or *Risk*. Perhaps we want to go back to the categories we filtered out too and bring a subset of those back in.

The rather simple parsing and slicing we did here doesn’t show us the important variables, it simply helps understand the relationships and the frequency with which they occur. Just because 90% of the data was scanning hosts, perhaps we only want to filter of those hosts with a risk of 2 or below. Our analysis has merely let us identify a set of nodes we can generate a higher priority alerts on while still capturing the other *Type*s into a lower priority or informational log.

Since AlienVault updates this list hourly, we can create a script to do this filtering before importing new revisions into our security tools and keep track of the percentage of nodes filtered out as a flag for the need to potentially readjust our rules. Furthermore, we should strongly consider re-performing this exploratory analysis on a semi-frequent basis to see whether we need to re-think our perspective on what constitutes nodes we “really should care about”.

In Summary

This chapter introduced the core structure and concepts of data analyses in *Python* and *R*. We incorporated basic statistics, foundational scripting/analysis patterns and introductory visualizations to help us both ask and answer a pertinent question. In addition, each example has demonstrated the similarity of *Python* (with *pandas*) and *R* coding techniques and generated output. The steps presented are just an example of one direction this particular analysis led. Every situation will be different and will require you to pull in different tools and techniques as needed.

In future chapters we will focus mainly on *R* code, with some *Python* sprinkled in on occasion. If you are already familiar with *Python*/*pandas* the previous examples should help you translate between the two languages. If you are new to both *R* and *Python* the standardization of future examples in one language should both help you follow along with less confusion and learn *R* a bit better.

Recommended Reading

*Statistics and Data with R: An applied approach through examples* by Yosef Cohen and Jeremiah Y. Cohen (John Wiley & Sons, Ltd. ISBN: 9780470758052)

*Python for Data Analysis* by Wes McKinney (O’Reilly Media, Inc. ISBN: 9798-1-4493-1979-3)