TE: Color code: GREEN is correct, YELLOW needs attention or modification, RED RED is incorrect or an error. Some yellow has been added by other editors (e.g. chapter references).

Chapter 4: Analyzing “Badness”

“Sometimes, bad *is* bad.”

Huey Lewis & The News, Sports, Chrysallis Records, 1983

Authors: It feels like this chapter is missing a proper introductory paragraph. What’s the focus and purpose of this chapter? It appears like the purpose of this chapter is to demonstrate data visualization and exploration techniques as they apply to specific types of data encountered in information security. You might pull some of the ideas presented in the Summary to use in the introduction.

You might also introduce the basic “data analysis workflow” pattern:

• downloading (if necessary) new data

• parsing/munging and converting the new data into a data frame

• validating the contents and structure of the new data

• extracting or computing relevant information from the new data source

• creating one or more new columns in our existing data frame

• running new analyses

And then explain that you will be showing how this works through several examples.

Also, the chapter title is not very informative. At the least, it should be “Visualizing ‘Badness’”, because you don’t really do any analysis in this chapter beyond a little correlation.

Also, you might explain what the word “badness” is short for. Eg. “By ‘badness’, we mean….”.

This chapter will continue the exploration of the AlienVault IP Reputation database that began in Chapter 3 along with an investigation into the ZeuS botnet and an analysis of un-fabricated

“un-fabricated” is an awkward term here. Why not just use ‘actual’?

firewall data. It assumes the reader is familiar with the description of the AlienVault data set and has at least followed along with all previous, preliminary analyses. As stated in the introduction of the book, no attempt has been made to incorporate consideration of or conduct analyses on Internet Protocol (IP) version 6 (IPv6) addresses and all the examples found in this chapter will be based on IPv4. Given the slow adoption and migration to IPv6 the plethora of “badness” still on IPv4 networks and the fact that it’s fairly straightforward to extrapolate IPv4 concepts to IP6, this should not be a practical limitation.

The struggle to protect, defend and understand our modern networks begins and ends—more often than not—with the building blocks of the internet: domain names, routes and especially IP addresses. IP addresses are defined in RFC 791, the “*Internet Protocol / DARPA Internet Program / Protocol Specification*” (http://tools.ietf.org/html/rfc791), which has an elegant and succinct way of describing them:

“A *name* indicates what we seek. An *address* indicates where it is. A *route* indicates how to get there.”

For this chapter, we are going to focus on primarily learning more [???] from the IP address. Global entities slice and dice

learning more about what? badness? How about this instead:

“For this chapter, we will focus primarily on visualizing IP address data as a way to understand the characteristics of “badness”.

them for public and private use; devices, systems and applications log them for reference; network management systems test, group, display and report on them; and, security tools make critical decisions based upon them. But, what—exactly—*is* an IP address, what can we learn from them and what part do they play in the quest for finding and mitigating “badness”?

Dissecting The “IP Address”

Some people in information security may just think of IP addresses as the strings used with a ping, nessus, nmap or other commands. But to perform security-oriented analyses of our system and network data, we must fully understand as much as we can about security domain data elements, just as those who perform data analyses in financial, agricultural or bio-medial disciplines must understand the underpinnings of the data elements in those fields. Since IP addresses are one of the most fundamental security domain data elements, let’s dig a bit deeper into them so we can fully integrate them into our analytics endeavors.

Representing IP Addresses

IPv4 addresses are comprised of four bytes, known as *octets*, and we usually come across them in a form called *dotted decimal notation* (e.g. “192.168.1.1”). Practically everyone reading this book understands this representation, if only by sight. This method of representation was briefly introduced in the IETF RFC 1123 in 1989 when they denoted it as “#.#.#.#”, but it was more clearly defined in the IETF’s uniform resource identifier (URI) generic syntax draft (RFC 3986, <http://tools.ietf.org/html/rfc3986>) in 2005. When you come across other security domain elements, you’ll want to do plenty of similar digging to ensure you have all information you need to process them or create complete regular expressions to locate them in unstructured data.

Since we know an 8-bit byte can range in value from 0 through 255, we also know the dotted decimal range is 0.0.0.0 through 255.255.255.255 which is 32-bits and if we count the possible address space, we have a total of 4,294,967,296 possible addresses (the maximum value of a 32-bit integer). This brings us to another method of storing and handling IP addresses: *any IP address can be converted to/from a 32-bit integer value*. This is important because the integer representation saves both space and time and we can calculate some things a bit easier with that representation than the dotted-decimal form. If you are writing or using a tool that only perceives an IP address as a character string or set of character strings, then you are potentially wasting space by trading a 4-byte, 32-bit representation for a (worst case) 15-byte, 120-bit representation. Furthermore, you are also choosing to use less efficient string comparison code versus integer arithmetic and comparison plus bitwise operations to accomplish the same tasks. While this may have little-to-no impact in some scenarios, the repercussions grow significant when dealing with large volumes of IP addresses (and become worse in the IPv6 world) and repeated operations.

type="tip"

Converting IPv4 Addresses To/From 32-bit Integers

To take advantage of integer operations for IPv4 addresses, you need to have some method of converting them to and from dotted decimal notation. IEEE Standard 1003.1 defines the common low-level (e.g. “C”) method of performing this conversion via inet\_addr() and inet\_ntoa() functions (http://pubs.opengroup.org/onlinepubs/009695399/functions/inet\_addr.html). However, these functions are not exposed to R. While it would be possible to write a C library and corresponding R glue module, it’s easier to write the functions in pure R with some help from the bitops package (Listing 4-1).

Listing 4-1

**library(bitops)** # load the bitops functions

# take an IP address string in dotted octets (e.g. "192.168.0.1")

# and convert it to a 32-bit long integer (e.g. 3232235521)

**ip2long <- function(ip) {**

# convert string into vector of characters

**ips <- unlist(strsplit(ip, '.', fixed=TRUE))**

# set up a function to bit-shift, then "OR" the octets

**arity <- function(x,y) bitOr(bitShiftL(x, 8), y)**

# Reduce applys a funcution cumulatively left to right

**Reduce(arity, as.integer(ips))**

**}**

# take an 32-bit integer IP address (e.g. 3232235521)

# and convert it to a (e.g. "192.168.0.1").

**long2ip <- function(longip) {**

# set up reversing bit manipulation

**arity <- function(nbits) bitAnd(bitShiftR(longip, nbits), 0xFF)**

# Map applys a function to each element of the arguent

# paste converts arguments to character andconcatenates them

**paste(Map(arity, c(24,16,8,0)), sep="", collapse=".")**

**}**

“library(bitops)” generates an error in RStudio: “Error in library(bitops) : there is no package called ‘bitops’”

To avoid this error, the reader must FIRST click the “Install Packages” under the “Packages” tab, in the lower-right sub-window, and type “bitops” in the pop-up window’s text entry box.

You can test out the functionality by reviewing the output from the following test code executed in the R console:

> long2ip(ip2long("192.168.0.0"))

[1] "192.168.0.0"

> long2ip(ip2long("192.168.100.6"))

[1] "192.168.100.6"

These and other IPv4 address functions are in the suda R module found on the companion web site.

NOTE: Python coders can use the pre-existing ipaddr package (<https://code.google.com/p/ipaddr-py/>), which has been incorporated into the Python 3 code base as the ipaddress module.

Segmenting And Grouping IP Addresses

There are a few different reasons we’d want to divide and group IP addresses, internally we may separate hosts by functionality or sensitivity, and routing tables would be overwhelmed if it needed to track each individual IP address. Due to the way TCP/IP was designed and how IPv4 networks are implemented there are numerous ways to segment or group them to make it easier to manage individual networks (i.e. “subnets”) and interoperate in the global internet. The original specification identified top-level *classes* (“A” through “E”) that were nothing more than a list of corresponding bitmasks for consuming consecutive octets which limited the usable range of addresses in each class and put some structure around the suggested use of each class.

Eventually, a more generalized, *classless* method of segmentation was established in RFC 4632 (http://tools.ietf.org/html/rfc4632). Rather than segment on whole octets, you can now specify address ranges in CIDR (Classless Internet Domain Routing) prefix format by appending the number of bits in the mask to a specified IP address. So, “172.16.0.0” (which has a mask of “255.255.0.0”) now becomes “172.16.0.0/16”.

It’s important to understand these points because we’ll want to leverage the groupings to dig into the data and relationships to pull out meaning. But once we understand the CIDR prefix format, we can see how those are grouped and defined as an autonomous system (AS) that are all assigned a numerical identifier known as the autonomous system number (ASN). ASNs have many uses (and associated data), for example, they are used by the border gateway protocol (BGP) for efficient routing of packets across the internet. Because of the relationship between ASN and BGP, it’s also possible to know the adjacent “neighbors” of each ASN. There are many more details regarding autonomous systems that you should dig into if you even only occasionally work with IP addresses in your analyses. To get a feel for the global make up of autonomous systems, you can explore public ASN information at the CIDR Report (http://www.cidr-report.org/as2.0/). But keep reading, as we’ll be looking at “badness” through an ASN lens later in the chapter.

type="tip"

Testing IPv4 Address Membership In a CIDR Block

When performing ASN- and CIDR-based analyses, one task that comes up regularly is the need to determine whether an address falls within a given CIDR range. To do this in R we just expand on our previously defined IPv4 address operations, convert both the IP address in question and the network block address to integers and perform the necessary bitwise operations to see if they do, indeed, line up (Listing 4-2).

Listing 4-2

# take an IP address (string) and a CIDR (string) and

# return whether the given IP address is in the CIDR range

**ip.is.in.cidr <- function(ip, cidr) {**

**long.ip <- ip2long(ip)**

**cidr.parts <- unlist(strsplit(cidr, "/"))**

**cidr.range <- ip2long(cidr.parts[1])**

**cidr.mask <- bitShiftL(bitFlip(0), (32-as.integer(cidr.parts[2])))**

**return(bitAnd(long.ip, cidr.mask) == bitAnd(cidr.range, cidr.mask))**

**}**

**ip.is.in.cidr("10.0.1.15","10.0.1.3/24")**

TRUE

**ip.is.in.cidr("10.0.1.15","10.0.2.255/24")**

FALSE

Your organization most likely uses CIDRs and ASNs internally as well, but these are not the only logical grouping mechanisms of sets of IP addresses. For example, you might have “*workstations*” as a high level grouping that covers all the user-endpoint DHCP-assigned address space or “*printers*” to logically associate all statically assigned single- or multi-function output devices. Servers may be grouped according to function or operating system type (or both). The concept of “*internal*” and “*external*” groupings for nodes may apply even if you use publicly routable addresses across your entire network. When looking for “badness”, do not discount the power of these logical groupings since you may be able to tie characteristics of them (data!) to various indicators you may be looking for. For example, it’s quite reasonable to expect the typical end-user workstation to make attempts to access nodes on the internet. However, the same is probably not true for printers, therefore one of the keys to learning about “badness” is this type of metadata and relationships.

Locating IP Addresses

Going down in detail, IP addresses map to individual devices that (usually) have unique media access control (MAC) addresses. It’s a fairly straightforward process to identify the switch and port of a node on your local network. With the proper metadata, you can create logical groupings based on this physical information and tie additional attributes to it, such as where—organizationally-geographically speaking—the node lives. By tying this information to an IP address, you won’t have to wait until a barrage of help desk calls come in to discover that there is something amiss in a particular department.

On a broader scale, there are also ways to tie an IP address that lives on the internet to a geographical location, with varying degrees of accuracy. One of the most popular ways to do so is with the Maxmind GeoIP database and APIs (http://dev.maxmind.com/geoip/), which is also used by the freegeoip project (http://freegeoip.net/). Chapter 5 will go into more detail on both using geolocated data.

Once you know where “badness” physically is, it’s a fairly straightforward process to visualize it on a map. The AlienVault data provides over 250,000 pre-geolocated addresses, but we’ll need to extract the pairs from the Coords field first (Listing 4-3).

Listing 4-3: R code to extract longitude/latitude pairs from AlienVault data

# read in the AlienVault reputation data (see Chapter 3)

**setwd("/suda/chapters")**

**avRep <- "ch03/data/reputation.data"**

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

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

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

# create a vector of lat/long data by splitting on ","

**av.coords.vec <- unlist(strsplit(as.character(av.df$Coords), ","))**

# convert the vector in a 2-column matrix

**av.coords.mat <- matrix(av.coords.vec, ncol=2, byrow=TRUE)**

# project into a data frame

**av.coords.df <- as.data.frame(av.coords.mat)**

# name the columns

**colnames(av.coords.df) <- c("lat","long")**

# convert the characters to numeric values

**av.coords.df$long <- as.double(as.character(av.coords.df$long))**

**av.coords.df$lat <- as.double(as.character(av.coords.df$lat))**

setwd("/suda/chapters") – generates an error in RStudio because this is not the same directory structure set up in Chapter 2 and used in Chapter 3.

It should be “~/book”. When I change that statement to setwd("~/book"), it works.

With the latitude and longitude coordinates in hand, you could avoid R or Python code altogether and use Google Maps to visualize the locations…if you felt like grabbing a caffeinated beverage while it uploads to Google Fusion Tables (<http://tables.googlelabs.com/>) and renders. Even though Google has considerably sped up their online mapping API, the resultant—albeit, handsome—map would end up being partially obscured with map markers. For example, mapping the AlienVault data set with Google Maps (Figure 4.1) produces a result that makes it seem like malicious hosts have consumed Japan. Rather than rely solely on Google, you can use the mapping functions in R to accomplish a similar task (Figure 4.2) with far greater precision. (You can find the code that generated Figure 4.2 on the companion web site

I don’t see this code in the R code file for Ch04.

and further examples of geographical mapping and analysis in Chapter 5.)

Figure 4.1 Google Fusion Table + Google Maps Chart Of AlienVault Malicious Nodes [793725c04f01.png]

Figure 4.2 R ‘ggplot/maps’ Package Dot-plot Map Of AlienVault Malicious Nodes [793725c04f02.png]

The ability to associate an IP address with a physical location and display it on a map has inherent utility (which will become even more apparent in Chapter 5) since it’s one thing to read the destinations of your internet users and quite another to “see” it on a map, especially if you’re trying to communicate the groupings versus just analyze them. Yet, there can be more to the end result of these geographical than a pretty picture.

Augmenting IP Address Data

In an analyst’s dream world, every data set we are asked to crunch through would be error-free and have all the attributes necessary for thorough and robust analyses. Sadly, information security is no different from other disciplines when it comes to imperfect data sets and highly distributed referential data or just a plethora of potential metadata sources. This *can* pose challenges to effective data analyses, but it is usually possible to find and use the data you need.

Even though we have geographic information in our AlienVault data set, the internet has, as we’ve indicated, both physical and logical groupings. It might be interesting to see how this data looks through a different lens, and for this example we’ll augment our data set (Listing 4-3) with additional data from the IANA IPv4 Address Space Registry (https://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xml). This data is a very high level grouping of IPv4 address space registry allocations and it should be emphasized that most of the registrants are not responsible for the malicious activity of individual nodes. So, while we cannot use this information to cast blame, it will give us one view of where badness is clustered, setting up possible, additional investigations.

type="tip"

On their web page under the heading “Alternative Formats”, IANA provides a handy link to the CSV version of the IPv4 address space allocations as well as a link to the traditional annotated text file. If you run the example code, you may see some strange behavior at times due to the CSV file being incomplete. It seems there is an automated process that converts a source of the IP table into the various formats and stops processing when the first octet hits three digits. You can either practice your data munging skills and convert the fixed-width version in the text file to CSV or use the version of the CSV that’s on our companion web site if you encounter any issues.

The data frame foundational data structure in R and pandas makes it very straightforward to reference and incorporate new data into our analyses, and your own projects will follow something close to this basic data analysis workflow pattern:

* downloading (if necessary) new data
* parsing/munging and converting the new data into a data frame
* validating the contents and structure of the new data
* extracting or computing relevant information from the new data source
* creating one or more new columns in our existing data frame
* running new analyses

For this example, we process the IANA data to see which registry allocations have the most malicious nodes.

Listing 4-4: R code to incorporate IANA IPv4 Allocations

setwd("/suda/chapters")

# retrieve IANA prefix list

ianaURL <- "http://www.iana.org/assignments/\

ipv4-address-space/ipv4-address-space.csv"

ianaData <- "ch04/data/ipv4-address-space.csv"

if (file.access(ianaData)) {

download.file(ianaURL, ianaData)

}

# read in the IANA table

iana <- read.csv(ianaData)

# clean up the iana prefix since it uses the old/BSD-

# number formatting (i.e. allows leading zeroes and

# we do not need to know the CIDR component.

iana$Prefix <- sub("^(00|0)", "", iana$Prefix, perl=TRUE)

iana$Prefix <- sub("/8$", "", iana$Prefix, perl=TRUE)

# re-read the existing AlienVault data

avRep <- "ch03/data/reputation.data"

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

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

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

# extract just the prefix from the AlienVault list

av.IP.prefix <- sapply(strsplit(as.character(av$IP), '.',

fixed=TRUE), "[", 1)

av$Designation <- sapply(av.IP.prefix, function(ip) {

iana[iana$Prefix == ip, ]$Designation

})

# summarize, order & review the findings

summary(factor(av$Designation))

Administered by AFRINIC Administered by APNIC

322 2615

Administered by ARIN Administered by RIPE NCC

17974 5893

AFRINIC APNIC

1896 93776

ARIN AT&T Bell Laboratories

42358 24

Digital Equipment Corporation Hewlett-Packard Company

1 3

LACNIC Level 3 Communications, Inc.

18914 31

PSINet, Inc. RIPE NCC

30 74789

setwd("/suda/chapters") – generates an error in RStudio because this is not the same directory structure set up in Chapter 2 and used in Chapter 3.

It should be “~/book”. When I change that statement to setwd("~/book"), it works.

We can do a quick check against the main IANA allocation table to see if this matches overall block assignments (Listing 4-5).

Listing 4-5: R code to extract IANA block assignments & compare with AlienVault groupings

# create a new data frame from the iana designation factors

**iana.df <- data.frame(table(iana$Designation))**

**colnames(iana.df) <- c("Registry", "IANA.Block.Count")**

# make a data frame of the counts of the av iana

# designation factor

**tmp.df <- data.frame(table(factor(av.df$Designation)))**

**colnames(tmp.df) <- c("Registry", "AlienVault.IANA.Count")**

# merge (join) the data frames on the "reg" column

**combined.df <- merge(iana.df, tmp.df)**

**print(combined.df[with(combined.df, order(-IANA.Block.Count)),],**

**row.names=FALSE)**

Registry IANA.Block.Count AlienVault.IANA.Count

APNIC 45 93776

**Administered by ARIN 44 17974**

ARIN 36 42358

**RIPE NCC 35 74789**

**LACNIC 9 18914**

Administered by APNIC 6 2615

Administered by RIPE NCC 4 5893

AFRINIC 4 1896

Administered by AFRINIC 2 322

Level 3 Communications, Inc. 2 31

AT&T Bell Laboratories 1 24

Digital Equipment Corporation 1 1

Hewlett-Packard Company 1 3

PSINet, Inc. 1 30

the RED highlighted code generated an error:

Error in `colnames<-`(`\*tmp\*`, value = c("Registry", "AlienVault.IANA.Count" :

'names' attribute [2] must be the same length as the vector [1]

Therefore, **I could not test** the YELLOW highlighted code above or below on this and the next few pages.

Then plot the data (Listing 4-6) to generate the chart in Figure 4.3.

Listing 4-6: R code to plot IANA Charts

# flatten the data frame by making one entry per “count” type

# versus having the counts in individual columns

**library(reshape)**

**library(ggplot2)**

**melted.df <- melt(combined.df)**

# plot the new melted data frame values

**gg <- ggplot(data=melted.df, aes(x=Registry, y=value))**

**gg <- gg + geom\_bar(aes(fill=variable), stat="identity") # using bars**

# and creating two charts, side-by-side based on the two different

# count types; note we’ve changed the default behavior of facet\_wrap()

# by letting it auto-adjust the y-axis scale. If we hadn’t, it would

# have resulted in ggplot applying the larger of the two scales and

# making the base IANA plot look almost blank.

**gg <- gg + facet\_wrap(~variable, scales="free\_y")**

# make a better label for the y axis

**gg <- gg + labs(y="Count")**

# rotate the x-axis labels and remove the legend

**gg + theme(axis.text.x = element\_text(angle = 90, hjust = 1),**

**legend.position = "none")**

Figure 4.3 R Bar Charts Comparing IANA Block Allocations [793725c04f03.eps]

There is some variation, but overall the larger blocks contribute the majority of malicious hosts. We’ve highlighted “RIPE NCC”, “Administered by ARIN” and “LACNIC” in the text/console output since “RIPE NCC” has a significantly larger number of malicious hosts than it’s allocation block count might imply (nearly double that of it’s very close neighbor “ARIN”) and “LACNIC” and “Administered by ARIN” both have a similar number of malicious hosts yet have different allocation block counts. Even with these discrepancies, can we make a more confident statement regarding the comparison between the number of malicious hosts in the /8’s managed by a registrar and the number of /8’s managed by a registrar?

Association/Correlation, Causation And Security Operations Center Analysts Gone Rogue

Since this chapter contains the first examples where data elements (variables) are grouped together and compared with other variables, this is a good place to mention the concept of *association* or, as you’ll see it referred to more often, *correlation*. Correlation is simply a measurement of the linear relationship between two or more variables. A positive correlation means a change in the value of one variable will also be reflected in the same directional change by the compared variable. Similarly, a negative correlation implies that a change in one variable will mean a change in the other, but in the opposite direction. If there is no consistent linear pattern in the change between variables, they are said to be uncorrelated. When we calculate the correlation value (stats-nerds call it the “r” value or correlation coefficient), we get a value between 1 (perfect positive correlation) and -1 (perfect negative correlation). As r gets closer to zero the linear correlation decreases and at zero, we say there is no correlation between the two values.

Figure 4.4 Scatterplots Showing Correlations [793725c04f04.eps]

I don’t see any scatter plots when viewed as PDF. I only see headings. This may be an artifact of the conversion process.

It’s important to remember that calculating simple correlation like this is a linear comparison. Look at the scatter plot with the parabola (upside down U shape), obviously there is a pattern and some type of relationship, but it’s not a linear correlation, so the calculated r value is very close to 0. Like most elements of statistics (or any complex discipline) there are many methods available to perform various tasks. This is also true for calculating correlation between two variables. Chapter 5 will take a look at topic called *linear regression*, which provides more detailed insight into correlation. Linear regression is also the basis for one type of predictive modeling. For the purposes of this chapter, we’ll be using a basic form of correlation.

type="note"

Correlation Caveats

Believe it or not, there are concrete parallels between statistics and information security. Statisticians use strange symbols and tools to perform their dark art much like malware researchers and network security specialists stare at rows of hexadecimal, octal and binary data to derive meaning. Security researchers also understand which tool to use for the job at hand (i.e. you wouldn’t use netflow data to try to understand detailed payload information in a communication session between two nodes). The same holds true for data scientists and there are, unfortunately, some further considerations to take into account when working with even basic correlation techniques.

This chapter described the Pearson correlation method, which is widely used given that it can work with data on an interval or ratio scale, with no restrictions placed on both variables being the same type. If you have ordinal or ranked data, two other algorithms: Spearman or Kendall’s Tau should be used instead. We aren’t delving into the correlation algorithm subtleties in this book, but you should have a solid understanding of the uses and limits of each before applying correlation in your own analyses.

Finally, correlation is a descriptive statistical measure versus an inferential one, meaning we can only describe the population we are studying and cannot use the outcome to generalize a statement about a larger group or make predictions based on the outcome.

It’s also important to remember that correlation is just showing some existence of a relationship between variables with no implication of *causation*. For example, perhaps a (hypothetical) analyst looked at the relationship between security incidents and the number of security operations staff, and reported, “*There is a strong positive correlation between the number of SOC analysts in an organization and the number of incidents reported*” which is could be misunderstood to imply that SOC analysts cause security incidents. In reality the patterns of data have similar trends with nothing else implied. Perhaps organizations with more incidents hire more SOC analysts, or after hiring more analysts, organizations discover more incidents. Perhaps the two are both a product of something else completely like larger organizations are targeted more and have both more incidents and analysts. When we calculate things like correlation, we have to be careful to keep this in context. People (and especially those looking for a headline) put a lot of faith in mathematically derived answers and “having a number”. That overconfidence may take our results out of context and into some really weird places, “***Researchers suggest we fire SOC analysts to reduce breaches!***” We just have to be careful of how we position our work and that we present our results with an appropriate communication of confidence in the techniques.

For the IANA data, it makes sense that there would be more malicious nodes in larger groups of assigned network blocks. This is an expert opinion that’s based on a cursory observation of data and an intuitive feel for the “right” answer. To make a more statistically backed statement, we should first plot the relationship between the two variables IANA.Block.Count and AlienVault.IANA.Count:

**ggplot(data=combined.df) +**

**geom\_point(aes(x=IANA.Block.Count, y=AlienVault.IANA.Count))**

Figure 4.5 Scatterplot of Malicious Node Counts to Number of /8 Blocks Managed By A Registrar [c0405.eps]

The scatterplot in Figure 4.5 appears to show a positive correlation, but to be sure, we move from eyeballs to keyboards to run a statistical comparison. There are a number of methods available to perform basic pairwise correlation and R provides access to three fundamental algorithms via the built-in cor() function:

**> cor(combined.df$IANA.Block.Count,**

**combined.df$AlienVault.IANA.Count, method="spearman")**

[1] 0.9488598

The value returned by cor() is known as the *correlation coefficient* and, as pointed out earlier, if it falls close to +1, this indicates there is a strong positive linear relationship between the two variables. R’s built-in cor() function offers three methods of correlation. For this example, we applied the Spearman correlation, which produces a rank correlation coefficient and generally more suited to variables that do not have a Normal distribution (you can execute the hist() function on each list to show that) and are better compared by rank.

We now have some statistical backing to help validate the visual pattern and logical (common sense) view that larger blocks of networks will contain more malicious hosts. You could run a similar analysis of your own internal data and, say, see if there’s a relationship between the number of employees in a department and the number of viruses detected. Chapter 5 goes into more detailed methods of testing for a relationship between variables.

Mapping Outside the Continents

Calculating and graphing information about “badness” is highly useful and vital to the operation of most, if not all, security technologies we deploy in our organizations. However, as a security data scientist, it’s a good idea to get into the habit of visualizing data to pick up structures or patterns you might not see otherwise. The classic example of this is Anscombe’s quartet:

Figure 4.6 Anscombe’s Quartet [793725c04f06.pdf]

All four data sets have the same statistical description (mean, standard deviation, and they even fit the same linear regression). Yet, when visualized, patterns emerge and we have a much better understanding of the data.

As seen earlier in the chapter, maps can also be powerful tools to communicate information visually, but there are other logical and physical visual representations of IP addresses available, especially if we want to see the interconnectedness of nodes. One very versatile representation is the *graph* structure since it provides both statistical data and has a myriad of options for visual presentation. Do not confuse our use of the term “graph structure” here with producing a graphic or chart. A graph structure is nothing more than a collection of nodes (vertices) and links between nodes (edges). Nodes and edges have inherent attributes, such as a name/label, but also have attributes that are calculated, such as the number of links going into and coming from the node (the degree). In a traditional graph structure, the direction of an edge (in or out) can be specified as well. In fact, as we’ll see in Chapter 10, graphs are becoming so generally useful that there are extremely popular, custom databases that make it very straightforward to store, modify and analyze large graph structures.

Visualizing the ZeuS Botnet

We now want to combine the metadata we can pull from IP addresses and apply the graph structure to it to visualize relationships in “bad” IP addresses. We’ll be mostly focusing on building and visualizing graph structures for the remainder of this chapter and introducing a smidgen of graph-based analysis as well. Previous examples have worked with the AlienVault IP reputation database, but it’s time to switch things up a bit an look at one slice of badness on the internet: the ZeuS botnet. Most security professionals have heard of ZeuS before, but just in case, here’s the description from the abuse.ch ZeuS tracker site (https://zeustracker.abuse.ch/):

ZeuS (also known as *Zbot* / *WSNPoem*) is a crimeware kit, which steals credentials from various online services like social networks, online banking accounts, ftp accounts, email accounts and other (phishing)

Despite some prominent attempts at taking down this botnet, it continues to hum along siphoning credentials. The abuse.ch site provides a handy blocklist (https://zeustracker.abuse.ch/blocklist.php?download=badips) of IP addresses that organizations can use to both identify ZeuS infected nodes and prevent infected systems from communicating with ZeuS C&C servers. To work with the blocklist, we’ll need to get it into R (a task you’re hopefully getting very familiar with by now).

Listing 4-7: R code to read in the ZeuS blocklist

# retrieve ZeuS blocklist

**zeusURL <- "https://zeustracker.abuse.ch/blocklist.php?\**

**download=ipblocklist"**

**zeusData <- "/suda/chapters/ch04/data/zeus.csv"**

**if (file.access(zeusData)) {**

# need to change download method for universal "https" compatibility

**download.file(zeusURL, zeusData, method="curl")**

**}**

# read in the ZeuS table; skip junk; no header; assign colnames

**zeus <- read.table(zeusData, skip=5, header=FALSE, col.names=c("IP"))**

The code in RED is the same working directory error as mentioned above.

The following code works:

setwd(“~/book”)

zeusURL <- "https://zeustracker.abuse.ch/blocklist.php?\

download=ipblocklist"

zeusData <- "ch04/data/zeus.csv"

if (file.access(zeusData)) {

# need to change download method for universal "https" compatibility

download.file(zeusURL, zeusData, method="curl")

}

# read in the ZeuS table; skip junk; no header; assign colnames

zeus <- read.table(zeusData, skip=5, header=FALSE, col.names=c("IP"))

We’ve switched to read.table() (read.csv() is a variant of that function) in Listing 4-7 since we only have one column. This particular data file has no header but it does have five lines at the beginning of the file that are comments and of no use to us. We also use some shorthand by avoiding a separate call to colnames() and embedding the column names right in the read.table() function call.

We’ll start off our “badness” quest by asking “which countries host ZeuS bots?” We could use a geolocation service to get this data, but we’ll take a different approach since we will also require some additional information for the next part of our analysis. The Team Cymru firm provides a number of IP-based lookup services (http://www.team-cymru.org/Services/ip-to-asn.html), including an IP to ASN mapping service that supports bulk queries over port 43 and returns quite a bit of handy information:

* AS number
* BGP prefix
* Country Code
* Registry
* When it was allocated
* AS organization name

The Team Cymru site clearly states that the country code data is only as accurate as the regional registry databases, but we’ve run some comparisons against geolocation databases and, for the purposes of the examples in this chapter, the data is accurate enough. To use this data, we’ll need some helper functions that can be found in the code provided on the book’s web site:

* trim(c): takes a character string and returns the same string with leading and trailing spaces removed
* BulkOrigin(ips): takes a list of IPv4 addresses and returns a detailed list of ASN origins
* BulkPeer(ips): takes a list of IPv4 addresses and returns a detailed list of ASN peers

To build our graph structure, we’ll perform the following steps:

* Lookup the ASN data
* Turn the IP addresses into graph vertices
* Turn the AS origin countries into graph vertices
* Create edges from each IP address to its corresponding AS origin country

It’s surprisingly simple R code (Listing 4-8):

Listing 4-8: Building ZeuS blocklist in a graph structure by country

library(igraph)

ips <- as.character(zeus$IP)

# get BGP origin data & peer data;

origin <- BulkOrigin(ips)

g <- graph.empty() # start graphing

# Make IP vertices; IP endpoints are red

g <- g + vertices(ips, size=2, color="red", group=1)

# Make BGP vertices

g <- g + vertices(origin$CC, size=2, color="orange", group=2)

# for each IP address, get the origin AS CC and return

# them as a pair to create the IP->CC edge list

ip.cc.edges <- lapply(ips, function(x) {

iCC <- origin[origin$IP==x, ]$CC

lapply(iCC, function(y){

c(x, y)

})

})

g <- g + edges(unlist(ip.cc.edges)) # build CC->IP edges

# simplify the graph by combining commmon edges

g <- simplify(g, edge.attr.comb=list(weight="sum"))

# delete any standalone vertices (lone wolf ASNs). In "graph" terms

# delete any vertex with a degree of 0

g <- delete.vertices(g, which(degree(g) < 1))

E(g)$arrow.size <- 0 # we hates arrows

# blank out all the IP addresses to focus on ASNs

V(g)[grep("\\.", V(g)$name)]$name <- ""

1) library(igraph) -- Same problem as above. You need to tell readers to “Install Packages” in RStudio before running this code, typing in “igraph”.

Now that we have a graph structure, it’s equally as straightforward to visualize it (Figure 4.7) just by passing the graph structure to the plot() function with some layout and label parameters (Listing 4-9).

Listing 4-9: Visualizing the ZeuS blocklist country cluster graph

# this is a great layout for moderately sized networks. you can

# tweak the "n=10000" if this runs too slowly for you. The more

# iterations, the cleaner the graph will look

**L <- layout.fruchterman.reingold(g, niter=10000, area=30\*vcount(g)^2)**

# plot the graph

**par(bg = 'white', mfrow=c(1,1))**

**plot(g, margin=0, layout=L, vertex.label.dist=0.5,**

**vertex.label.cex=0.75,**

**vertex.label.color="black",**

**vertex.label.family="sans",**

**vertex.label.font=2,**

**main="ZeuS botnet nodes clustered by country")**

Figure 4.7 ZeuS Nodes Clustered By Origin Country [793725c04f07.eps]

If your country code memory is a bit rusty, we can use R to provide a lookup table (Listing 4-10):

Listing 4-10: Country code name translation

# read in country code to name translation table

**zeus.cc <- grep("[A-Z]", V(g)$name, value=TRUE)**

**zeus.cc <- zeus.cc[order(zeus.cc)]**

# read in the country codes data frame

**cc.df <- read.csv("/suda/chapters/ch04/data/countrycode\_data.csv")**

# display cc & name for just the ones from our data set

**print(head(cc.df[cc.df$iso2c %in% zeus.cc, c(7,1)], n=10),**

**row.names=FALSE)**

iso2c country.name

AR ARGENTINA

AU AUSTRALIA

AT AUSTRIA

AZ AZERBAIJAN

BG BULGARIA

CA CANADA

CL CHILE

CN CHINA

CZ CZECH REPUBLIC

DE GERMANY

Directory should be “~/book” instead of “/suda/chapters”

As stated earlier, it’s easier to understand quantities with a simple bar chart or table (Listing 4-11) but the graph tends to add a visual impact that traditional presentation techniques lack (so, it’s probably a good idea to put them both together when presenting your output).

Listing 4-10: Country code name translation

> sort(table(factor(origin$CC)))

AR AT AZ BG CL EE HK IN KG MY RW SA SK TR TW VG AU ES IE IL IT PL SC

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2

SE VN CN CZ LT LV TH FR LU CA NL DE RO GB UA RU US

2 2 3 3 3 3 3 5 5 6 9 10 13 14 18 31 61

From our previous work with ASNs, we know that IPs live both in physical and logical space. Now that we have a graph view of the physical world, let’s take a look at the ZeuS IP addresses in relation to their ASNs of origin and also include ASN peers to truly start to see it as a network.

Listing 4-11: Connected network of ZeuS IPs, ASNs & ASN Peers

g <- graph.empty()

g <- g + vertices(ips, size=2, color="red", group=1)

origin <- BulkOrigin(ips)

peers <- BulkPeer(ips)

# add ASN origin & peer vertices

g <- g + vertices(unique(c(peers$Peer.AS, origin$AS)),

size=2, color="orange", group=2)

# build IP->BGP edge list

ip.edges <- lapply(ips, function(x) {

iAS <- origin[origin$IP==x, ]$AS

lapply(iAS,function(y){

c(x, y)

})

})

bgp.edges <- lapply(

# Team Cymru’s whois service shouldn’t be fed “NA”s

grep("NA",unique(origin$BGP.Prefix),value=TRUE,invert=TRUE),

function(x) {

startAS <- unique(origin[origin$BGP.Prefix==x,]$AS)

lapply(startAS,function(z) {

pAS <- peers[peers$BGP.Prefix==x,]$Peer.AS

lapply(pAS,function(y) {

c(z,y)

})

})

})

g <- g + edges(unlist(ip.edges))

g <- g + edges(unlist(bgp.edges))

g <- delete.vertices(g, which(degree(g) < 1))

g <- simplify(g, edge.attr.comb=list(weight="sum"))

E(g)$arrow.size <- 0

V(g)[grep("\\.", V(g)$name)]$name = ""

L <- layout.fruchterman.reingold(g, niter=10000, area=30\*vcount(g)^2)

par(bg = 'white')

plot(g, margin=0, layout=L, vertex.label.dist=0.5,

vertex.label=NA,

main="ZeuS botnet ASN+Peer Network")

Figure 4.8 ZeuS Nodes Graph With ASNs + Peers [793725c04f08.eps]

By expanding the network with the ASN peers, we can see a cluster of interconnected ASNs that might be worth exploring further, but you’ll need to digest the resources in the “Recommended Reading” section to take that next step.

With basic graph network concepts well in hand, we can turn our attention to a more practical application of these functions: visualizing “badness” on *your* network using actual data from a real environment and attempt to *visualize* the answer to the question, “How much badness is attempting to [come into|get out of] my network?”

Visualizing Firewall “Badness”

Examining generic “badness” data has some merit, but it would be more helpful to apply these analysis and visualization techniques to your own organization. To that end, this last example provides a way to use both the AlienVault IP Reputation database and the graphing techniques presented in this chapter to examine what’s happening on a perimeter firewall. Rather than generate some artificial data, we asked for volunteers to provide us with a days’ worth of inbound and outbound summary data (source, destination, port, session count) from their perimeter firewalls. We asked for summary data as the mean of the size of the raw logs from a diversity of volunteers as just over 3GB and we aren’t ready to start working with that amount of data in this chapter. Unfortunately, this summary data doesn’t contain information on whether the traffic was blocked or accepted, but it’s sufficient for an example. We asked

??

We’ve broken the firewall data down into a source list and and destination list to see what we can learn about the types of malicious traffic that is trying to get in and out of the network of one of the volunteers. We’ve also created two new functions, which can be found on the web site:

* graph.cc(ips,av.df): takes in an list of IPv4 addresses and an AlienVault data frame and returns a complete graph network structure of nodes clustered by country code; also (optionally) plots the graph with a summary of malicious traffic types;
* graph.asn(ips,av.df): takes in an list of IPv4 addresses and an AlienVault data frame and returns a complete graph network structure of nodes clustered by ASN; also (optionally) plots the graph with a summary of malicious traffic types.

We’ll start by loading in the destination IP addresses and filtering out everything that isn’t in the AlienVault database, then assess the result and try to get a feel for what type of “badness” to hone in on. Even with the potential bias in the data (as described in Chapter 3), a higher reliability rating should still mean there is a better chance the node is actually “bad”, so we’ll focus on entries with reliability greater than six, which will give us 127 nodes to send to graph.cc() to process and plot.

Listing 4-12: Working With Real Data

avRep <- "/suda/chapters/ch03/data/reputation.data"

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

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

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

# read in list of destination IP addresses siphoned from firewall logs

dest.ips <- read.csv("/suda/chapters/ch04/data/dest.ips",

col.names= c("IP"))

# take a look at the reliability of the IP address entries

# you could also plot a histogram

table(av.df[av.df$IP %in% dest.ips$IP, ]$Reliability)

1 2 3 4 5 6 7 8 9 10

16 828 831 170 1 266 92 2 23 24

# extract only the "bad" ones, designated by presence in alienvault

# database with a reliability greater than 6 since there seems to

# be a trailing off at that point

ips <- as.character(av.df[(av.df$IP %in% dest.ips$IP) &

(av.df$Reliability > 6), ]$IP)

# graph it

g.cc <- graph.cc(ips, av.df)

This code runs, but I get this warning at the end:

Warning message:

In vattrs[[name]][index] <- value :

number of items to replace is not a multiple of replacement length

Figure 4.9 Graph Of Malicious Destination Traffic By Country [793725c04f09.pdf]

The bar chart on the right serves as both a legend for the colors of the graph nodes and also provides a summary of the totals of each classification type. Of note in Figure 4-9 we can see there is some potential C&C traffic and that the United States has the highest number of possible malicious destinations. With graph.cc()’s ASN cousin and the slicing & dicing example in Listing 4-12, you should have enough tools to generate your own views to look at different aspects of the malicious traffic.

In Summary

Hopefully you’ve seen the importance of fully understanding the data elements you wish to analyze and visualize as well as the need to start with a question and iterate through computations and visualizations to work towards an answer. There are plenty of other similar data sets available on the internet to substitute for the ones provided in the most of the examples. Hunting those down (or just using your own firewall data in the last example), working through the sample analyses and formulating your own questions will help to ingrain the pattern of the data analysis workflow in your mind.

There are many ways to look at IP-based “badness” and this chapter was by no means comprehensive. Furthermore, R was not entirely necessary for anything but the visualizations and statistical analyses. Much of the sorting, slicing and dicing could have been performed in a database and—as we’ll see in Chapter 10—that is definitely the place to start when working with larger data sets.

The next chapter will expand on these analyses of “badness”, giving us some new “out of this world” perspectives on botnet data.

I suggest adding one reference to support the “Correlation Caveats”– Pearson, Spearman, and Kendall’s Tau

Recommended Reading

*Mining Graph Data* by Diane J. Cook & Lawrence B. Holder (John Wiley & Sons, Ltd. ISBN: 9780471731900)

*Graphical Models With R* by Højsgaard, Søren, David Edwards, and Steffen Lauritze (Springer, ISBN 978-1461422983)