Chapter 4: Analyzing “Badness”

“Sometimes, bad *is* bad.”

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

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 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 an IP address. 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.”

Global entities slice and dice them for public and private use; devices, systems and applications log them for future 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 and what part do they play in the quest for finding and mitigating “badness”?

Dissecting The “IP Address”

Unless you write operating system level code for a living or create and/or contribute to network-level tools such as tcpdump (<http://www.tcpdump.org/>) or Wireshark (<http://www.wireshark.org/>) on a regular basis, you may not have given IP addresses more of thought than the strings you use with a ping, nessus, nmap or other commands. To perform security-oriented analyses of your system and network data, you must fully understand as much as you 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, yet the “standard” way to denote this now-common string really wasn’t defined until the IETF’s uniform resource identifier (URI) generic syntax draft (<http://tools.ietf.org/html/rfc3986>) was published (apart from a *brief* mention in RFC 1123 – <http://tools.ietf.org/html/rfc1123> – as “#.#.#.#”). Why dwell on this? We’ll answer a question with a question: “*are leading zeroes in the octet numbers legal?*” If you’re going to write code to parse, slice, dice and analyze IP addresses, that’s an important answer to know. Here is the “official” representation format (in grammar notation):

IPv4address = d8 "." d8 "." d8 "." d8

d8 = DIGIT ; 0-9

/ %x31-39 DIGIT ; 10-99

/ "1" 2DIGIT ; 100-199

/ "2" %x30-34 DIGIT ; 200-249

/ "25" %x30-35 ; 250-255

Even if you’re not familiar with formal grammars, it’s not too difficult to see that this one says an IPv4 address is four 1-3 digit numbers separated by three periods. Given the fact that the standard explicitly calls out that leading zeroes are permitted in the IPv6 grammar section, there is an implicit “no leading zeroes, except in the case of the octet being ‘0’” understanding. When you come across other security domain elements, you’ll want to do plenty of digging to ensure you have all information you need to process them.

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 for a total of 4,294,967,296 possible addresses (the maximum value of a 32-bit integer). This is an important fact to remember: *any IP address can be converted to/from a 32-bit integer value*. The integer representation saves both space and time. 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

The only way to take advantage of integer operations for IPv4 addresses is 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 and safer to write the functions in pure R with some help from the bitops package. The code below requires the bitops package, so you’ll need to install that first:

install.packages("bitops")

then, proceed to add the following functions to your R session:

**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) {**

# Reduce applys a funcution cumulatively on the arguments

# from left to right. bit-shift, then "OR" the octets

**Reduce(function(x,y) {**

**bitOr(bitShiftL(x,8),y)**

**}, sapply(strsplit(ip,".",fixed=TRUE),function(o) {**

**as.integer(o)**

**}))**

**}**

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

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

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

# paste converts arguments to character andconcatenates them

# Map applys a function to each element of the arguent

# in this case, it reverses the operations of ip2long

**paste(Map(function(nbits) {**

**bitAnd(bitShiftR(longip,nbits),0xFF)**

**}, c(24,16,8,0)), sep="",collapse=".")**

**}**

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

Due to the way TCP/IP was designed and how IPv4 networks are implemented there are numerous ways to segment/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 “255.255.255.0”) now becomes “172.16.0.0/16”.

A managed collection of these prefixes is defined as an autonomous system (AS), has a numerical assignment (ASN) and is used by the border gateway protocol (BGP) for efficient routing of packets across the internet (so, 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 sytems, you can explore public ASN information at the CIDR Report (http://www.cidr-report.org/as2.0/) and, 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.

# 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 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 getting a handle on “badness” is this type of metadata.

Locating IP Addresses

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, say, in the “northeast regional sales 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/). Chapter 5 will go into more detail on both using geolocated data and the expected level of accuracy from geolocation services.

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:

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")**

# extract the longitude & latitude data from the Coords

# column and create a new data frame by splitting each

# coordinate (at the comma), converting it to a matrix

# and then converting the matrix to a data frame. We

# put it into a data frame since the ggplot() routines

# used for mapping work from data frame sources.

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

**matrix(unlist(strsplit(as.character(av.df$Coords),",")),**

**nrow=nrow(av.df), byrow=TRUE),stringsAsFactors=FALSE)**

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

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

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

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

Figure 4.1 Google Fusion Table/Google Maps Chart Of AlienVault Malicious Nodes [c0401.png]

Figure 4.2 R ‘ggplot/maps’ Package Dot-plot Map Of AlienVault Malicious Nodes [c0402.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, 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 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"

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. 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 pattern:

* downloading (if necessary) new data
* parsing and converting the new data into a data frame
* validating the contents and structure of the new data
* performing any necessary munging of the new data to make it easier to process/incorporate
* 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.

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

**desig <- summary(factor(av$Designation))**

**desig <- desig[order(-desig)]**

**desig**

APNIC **RIPE NCC**

93776 **74789**

ARIN **LACNIC**

42358 **18914**

**Administered by ARIN** Administered by RIPE NCC

**17974** 5893

Administered by APNIC AFRINIC

2615 1896

Administered by AFRINIC Level 3 Communications, Inc.

322 31

PSINet, Inc. AT&T Bell Laboratories

30 24

Hewlett-Packard Company Digital Equipment Corporation

3 1

We can do a quick check against the main IANA allocation table to see if this matches overall block assignments:

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$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)**

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

Registry IANA.Block.Count AlienVault.IANA.Count

6 APNIC 45 93776

**3 Administered by ARIN 44 17974**

7 ARIN 36 42358

**14 RIPE NCC 35 74789**

**11 LACNIC 9 18914**

2 Administered by APNIC 6 2615

4 Administered by RIPE NCC 4 5893

5 AFRINIC 4 1896

1 Administered by AFRINIC 2 322

12 Level 3 Communications, Inc. 2 31

8 AT&T Bell Laboratories 1 24

9 Digital Equipment Corporation 1 1

10 Hewlett-Packard Company 1 3

13 PSINet, Inc. 1 30

# also plot the values (Figure 4.3)

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

# versus having the counts in individual columns

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

# we’ll use ggplot to create a bar chart to make it easier to compare

**library(ggplot2)**

# plot the new melted data frame values

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

# using bars

**geom\_bar(aes(x=Registry, y=value, fill=variable), stat="identity") +**

# 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.

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

# make a better label for the y axis

**labs(y="Count") +**

# rotate the x-axis labels and remove the legend

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

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

Figure 4.3 R Bar Charts Comparing IANA Block Allocations [c0403.png]

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 scientific 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 pattern in the change between variables, they are said to be uncorrelated. It’s a bit easier to visualize using a scatterplot:

Figure 4.4 Scatterplots Showing Correlations [c0404.eps]

It’s easy to determine the upward, downward and lack of correlation properties in these graphs, but real-life scatterplots aren’t always as straightforward to visually analyze. Even when it is possible to visually discern some correlation between variables, it would be useful to have a way to describe the strength of the correlation in a more concrete way than the “meh: negative” of Figure 4.5.

Figure 4.5 A Weak Negative Correlation [c0405.eps]

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 regressions is also the basis for one type of predictive modeling. For the purposes of this chapter, we’ll be using the basic form of correlation.

However, before discussing correlation, we need to review a very related topic: *covariance*. Covariance is similar to correlation in that the purpose behind the calculation is to determine if, what kind (positive or negative) and how much variation exists between two variables. Without resorting to icky equations, the basic idea of covariance (and correlation) is to make rectangles of the (x, y) coordinate pairs we plotted on the scatterplot in Figure 4.4, summing the areas of the “positive” rectangles and subtracting the sum of the areas of the “negative” rectangles (then dividing it all by by n-1). We’ve presented this visually in Figure 4.6. If the areas are the same (middle scatterplot) then the covariance is near 0, otherwise it will be positive or negative. Interpreting the calculated covariance value requires some insight into the data, which is precisely what calculating the correlation does for us! In essence, the correlation calculation assesses the covariance relative to the variances between the two variables, which can be done by dividing the covariance by the product of the standard deviations of the variables. This normalizes the result, (it will now be in the range of -1 to 1 versus aligned with the magnitude of the variables) which makes it much easier to quickly understand the overall strength of the dependence between the variables. For the above examples, the covariance coefficient for each scatterplot is 103.0902, -3.128741, -89.65348, -33.188, respectively. We have no idea just how strong or weak those are without looking at the data, whereas it’s pretty clear to understand from the correlation coefficients: 0.8749181, -0.05405642, -0.8406494, -0.4974314

Figure 4.6 Covariance “Rectangles” [c0406.png]

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 researches 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 important to remember that correlation is just showing the type of linear relationship between variables with no implication of *causation*. For example, after performing an analysis of last year’s security incidents, the SOC manager reported “*there is a strong positive correlation between the number of SOC analysts in an organization and the number of incidents reported*” which, on the surface, implies that SOC analysts cause security incidents. In reality, having more SOC analysts most likely means more indicators/events are reviewed resulting in identifying more security incidents. There is as equal a danger in making assumptions with scientific backing as there are making judgments based solely on simple statistical calculations.

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 4.7 Scatterplot of Malicious Node Counts to Number of /8 Blocks Managed By A Registrar [c0407.png]

Given the large gaps and the right-skew of the variables, we can also do a scatterplot of the log() of each variable.

ggplot(data=combined.df) +

geom\_point(aes(x=log(IANA.Block.Count),

y=log(AlienVault.IANA.Count)))

Figure 4.8 Scatterplot of Malicious Node Counts to Number of /8 Blocks Managed By A Registrar (log) [c0408.png]

Both scatterplots seem to show a linear relationship, 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.

# validte our visal skewness assumption

**library(e1071)**

**skewness(combined.df$IANA.Block.Count)**

0.8633021 # > 0.5 so moderately right skewed

**skewness(combined.df$AlienVault.IANA.Count)**

1.431707 # > 1 so highly right skewed

# use the log of each variable for the correlation **cor(log(combined.df$IANA.Block.Count),**

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

0.9488598

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

**method="spearman")**

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 are not normally distributed (even with the log() transformation, neither variable is normally distributed) 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 4.9 Anscombe’s Quartet [c0408.png]

All four data sets have the same statistical description. 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. 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 # (databases chapter), graphs are becoming so generally useful that there are extremely popular, custom databases that make it very straightforward to store, modify and analyze very large graph structures.

NEED TO INSERT A REFERENCE TO WILEY’S MINING GRAPH DATA BOOK Mining Graph Data Diane J. Cook (Editor), Lawrence B. Holder (Editor) ISBN: 978-0-471-73190-0

We’ll be focusing on building and visualizing graph structures for the remainder of this chapter and introducing a smidgen of graph-based analysis.

# smaller helper function to [l|r]trim blanks

**trim <- function (x) gsub("^\\s+|\\s+$", "", x)**

**BulkOrigin <- function(ip.list,host="v4.whois.cymru.com",port=43) {**

# setup query to bulk whois server

**cmd <- "begin\nverbose\n"**

**ips <- paste(unlist(ip.list), collapse="\n")**

**cmd <- sprintf("%s%s\nend\n",cmd,ips)**

# creat the connection and post the query

**con <- socketConnection(host=host,port=port,blocking=TRUE,open="r+")**

**cat(cmd,file=con)**

**response <- readLines(con)**

**close(con)**

# trim header, split fields and convert results

**response <- response[2:length(response)]**

# split and trim each response line

**response <- laply(response,.fun=function(n) {**

**sapply(strsplit(n,"|",fixed=TRUE),trim)**

**})**

# put the results into a data frame

**response <- adply(response,c(1))**

# ignore the trailing field

**response <- subset(response, select = -c(X1) )**

# assign meaningful column names

**names(response) = c("AS","IP","BGP.Prefix","CC",**

**"Registry","Allocated","AS.Name")**

# return the result

**return(response)**

}

# 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"))**

# we'll be using the IP data quite a bit, so assign it to a variable

**ips <- as.character(zeus$IP)** # we don’t want it as a factor now

# get BGP origin data & peer data; we can use this data beyond

# making pretty graphs, too.

**origin <- BulkOrigin(ips)**

# start graphing

**g <- graph.empty()**

# Make IP vertices; IP endpoints are red

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

# Make BGP vertices

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

# Make IP/BGP edges

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

# them as a pair to create the edge list

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

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

**lapply(iAS,function(y){**

**c(x,y)**

**})**

**})**

# build ASN->IP edges

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

# simplify the graph

**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))**

# we hates arrows

**E(g)$arrow.size <- 0**

# if we opt to show labels, we only care about ASN data, so

# blank out all the IP addresses

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

# 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')**

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

**main="ZeuS botnet nodes mapped across ASNs & ASN peers")**