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. 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 plus the plethora of “badness” still on IPv4 networks, 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 represent 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 and dice 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

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.

Since we know an 8-bit byte can range in value from 0 through 255, that means 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). You can explore public ASN information at the CIDR Report (http://www.cidr-report.org/as2.0/) and 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/) that will be covered in “Chapter 5” where we will go into more detail on both using the data and the expected level of accuracy. 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-geo-located 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 daa from the Coords

# column and create a new data frame

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. The—albeit, handsome—map would also be partially obscured with map markers. With the AlienVault data set mapped with Google (Figure 4.1), it looks 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.)

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 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.

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

Augmenting IP Address Data

In an analyst’s dream world, every data set you 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 when it comes to imperfect data sets and highly distributed referential data or more 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, enabling us to perform 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) of 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
* performing any necessary munging of the existing data to make it easier to incorporate 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.

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

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

})

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

desig <- desig[order(-desig)]

desig

APNIC **RIPE NCC**

93775 **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

iana.df <- data.frame(table(iana$Designation),stringsAsFactors=FALSE)

colnames(iana.df) <- c("reg","ct")

av.reg <- iana.df[df$reg %in% names(desig),]

av.reg[with(av.reg, order(-ct)),]

reg ct

8 APNIC 45

**3 Administered by ARIN 44**

10 ARIN 36

**39 RIPE NCC 35**

**33 LACNIC 9**

2 Administered by APNIC 6

5 Administered by RIPE NCC 4

6 AFRINIC 4

1 Administered by AFRINIC 2

34 Level 3 Communications, Inc. 2

12 AT&T Bell Laboratories 1

18 Digital Equipment Corporation 1

28 Hewlett-Packard Company 1

38 PSINet, Inc. 1

There is some variation, but overall—as expected—the larger blocks contribute the majority of malicious hosts. We’ve highlighted “RIPE NCC”, “Administered by ARIN” and “LACNIC” in the output since “RIPE NCC” has a significantly larger number of malicious hosts than it’s assignment 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 very different allocation counts. Delving into why might be an interesting exercise (for the intrepid reader).

Mapping Outside the Continents