Reading Log Files - Eric Sun

Introduction

I validated the first task of reading in the log files through the strategy handout on the 141B repository. The file Merged.log contains 99,968 lines of log messages, with five different log files of varying lengths. I'm splitting the lines based on 5 components in each log message: Date-Time, Host, App, Process ID, and Message. I will also add a new column that will specify which log file name the message is originally from for each message. An important feature of the data is that we assume that each log-file message is a single line, so we do not have to worry about losing data.

```
file <- readLines("MergedAuth.log") # there are 99,968 elements
length(file)
```

```
## [1] 99968
```

```
file = file[file != ""] # 3 white spaces
length(file)
```

```
## [1] 99965
```

I read in the MergedAuth.log file using readLines() and removed all the empty lines. This gave me a total of 99665 lines, with 5 of them being the log file name, and the rest being log messages.

Regular Expression

```
rx = gregexpr(
  "^(?P<Date_Time>[[:alpha:]]+ [0-9\\s]?[0-9] \\d{2}:\\d{2}:\\d{2}|# .*.log$)\\s?(?P<Host>[a-zA-Z0-9-]*)
  file,
  perl = TRUE
)

table(
  grepl(
    "^(?P<Date_Time>[[:alpha:]]+ [0-9\\s]?[0-9] \\d{2}:\\d{2}:\\d{2}|# .*.log$)\\s?(?P<Host>[a-zA-Z0-9-
    file,
    perl = TRUE
  )
)
```

```
##
##  TRUE
## 99965
```

The regex above matched every line in the file. It took a lot of trial and error to perfectly match every line since the structure of the messages was not consistent. Although the regex above matched all lines, I still had to ensure that the sub-patterns matched the correct content for all the lines. I verified that I grabbed every line correctly using the strategies shown below

Verification of Regex

We can verify that our expression matched every line by checking how many matches there are for each line.

```
table(sapply(rx, length)) # how many matches of sub-pattern per line (all should have length 1)
```

```
##
##     1
## 99965
```

Indeed, we see that 99965 of the lines matched rx exactly once. We can also verify that the value is the starting position for each match = 1.

```
table(sapply(rx, `[`, 1) == 1) # value for lines should be 1
```

```
##
##  TRUE
## 99965
```

Lastly, we can verify that the length for all of the regex matches is equal to the number of characters in each line.

```
rxlen = sapply(rx, function(x)
  attr(x, "match.length"))

table(rxlen)
```

```
## rxlen
##    10    11    23    27    45    46    47    48    49    51    52    53    55
##     1     1     1     2     7     1     1     1     1     8     8     4     5
##    56    57    58    59    60    61    62    63    64    65    66    67    68
##     4     2     4     1     2     5     1     7     7    52    14     8    25
##    69    70    71    72    73    74    75    76    77    78    79    80    81
##   148    35    74    56   112   218    31    35    70   155   312   820  2044
##    82    83    84    85    86    87    88    89    90    91    92    93    94
##  2352  2721  1965  1311   781   569   277   518  1219   447   528   278   191
##    95    96    97    98    99   100   101   102   103   104   105   106   107
##  1092  1857  4855  3814  1836  1144   753   550   945   969  3371 17913 12586
##   108   109   110   111   112   113   114   115   116   117   118   119   120
## 12621  1350   327  1728  3177    30   251    81   161   154   279   250    60
##   121   122   123   124   125   126   127   128   129   130   131   132   133
##    49   210    42    35    43     8    62    33    49    43   101    42    47
##   134   135   136   137   138   139   140   141   142   143   144   145   146
##    24    32    96    24    95   197   194   444    59   229    68    40    97
##   147   148   149   150   151   152   153   154   155   156   157   158   159
##   136   340    89    37    13    22    77    65   155   131   108    80    56
```

```
##   160   161   162   163   164   165   166   167   168   169   170   171   172
##   150   185   108    97   139   373   117   321    92   125   616   210   245
##   173   174   175   176   177   178   179   180   181   182   183   184   185
##    67    99    87   105  1764   103   508    15    30    63    54    41   224
##   186   187   188   189   190   191   194   196   197   198   199   200   201
##   231     6    72     1     3     2     2     1     4     9    17     4     3
##   202   203   204   205   206   207   208   209   210   211   212   213   214
##     5     1     2    12    19     5     8     7    12    19    36    11    31
##   215   217   218   219   220   221   223   225   229   230   232   233   234
##    19     1     9     1     2    18     1     1     8     1     1     1     2
##   235   237   238   239   241   242   244   245   246   247   254   257   266
##     1    12     1     1     1     3     2    12    10     2     1     3     8
##   267   273   274   275   277   294   295   302   303   307   315   317   321
##     1     3     2     1     3     6     5     1     1     4     7     1     1
##   327   334   368   375   376   382   385   386   387   396   397   448   457
##     1     1     4     3     7     1     1     3     1     1     2     7     1
##   488   489   495   496   524   533   534   536   619   625   637   718   730
##     8     3     1     1     5     2     7     1     1     1     1     1     1
##   779   813   861   873   926   940   967  1037  1103  1119  1195
##     1     1     2     1     1     1     1     1     1     1     3
```

```
table(rxlen == nchar(file)) # length of match = # of char in file
```

```
##
##  TRUE
## 99965
```

Part I.) CREATING THE LOG FILE DATAFRAME

After validating that the pattern for each line was read in correctly, I began to create the data frame.

```
# rx[[3]]
s = attr(rx[[3]], "capture.start")
substring(file[3], s, s + attr(rx[[3]], "capture.length") - 1)
```

```
## [1] "Nov 30 06:47:01"
## [2] "ip-172-31-27-153"
## [3] "CRON"
## [4] "22087"
## [5] " pam_unix(cron:session): session opened for user root by (uid=0)"
```

I extracted the five categories (date_time, app, etc.) by looking at what capture.start and capture.length my sub patterns extracted. For this, I checked only a couple elements of rx to ensure that the capture groups were being captured properly. Through using substring to get the matches and verifying it manually by counting the start and end values for every group, I verified that the capture.groups and extractions were accurate.

To do the same process above for the entire file, I used this piece of code below:

```
caps = mapply(function(str, match) {
  s = attr(match, "capture.start")
  substring(str, s, s + attr(match, "capture.length") - 1) # add -1 to not count last char
}, file, rx)
```

This returns a matrix with 5 rows and 99965 columns. I transposed the matrix and converted it into a data frame. After that, I cleaned the data frame by setting the row names to NULL and renaming the column names to their respective attribute.

```
log_files = as.data.frame(t(caps))
row.names(log_files) = NULL
colnames(log_files) = c("Date_Time", "Host", "App", "ID", "Message")

dim(log_files)
```

```
## [1] 99965     5
```

```
class(log_files)
```

```
## [1] "data.frame"
```

```
head(log_files)
```

```
##         Date_Time             Host  App    ID
## 1      # auth.log
## 2 Nov 30 06:39:00 ip-172-31-27-153 CRON 21882
## 3 Nov 30 06:47:01 ip-172-31-27-153 CRON 22087
## 4 Nov 30 06:47:03 ip-172-31-27-153 CRON 22087
## 5 Nov 30 07:07:14 ip-172-31-27-153 sshd 22116
```

```
## 6 Nov 30 07:07:35 ip-172-31-27-153 sshd 22118
##                                                                 Message
## 1
## 2             pam_unix(cron:session): session closed for user root
## 3  pam_unix(cron:session): session opened for user root by (uid=0)
## 4             pam_unix(cron:session): session closed for user root
## 5                    Connection closed by 122.225.103.87 [preauth]
## 6                    Connection closed by 122.225.103.87 [preauth]
```

```
tail(log_files)
```

```
##             Date_Time  Host  App    ID
## 99960 Dec 10 11:04:42 LabSZ sshd 25539
## 99961 Dec 10 11:04:42 LabSZ sshd 25539
## 99962 Dec 10 11:04:43 LabSZ sshd 25541
## 99963 Dec 10 11:04:43 LabSZ sshd 25541
## 99964 Dec 10 11:04:43 LabSZ sshd 25544
## 99965 Dec 10 11:04:45 LabSZ sshd 25539
##
## 99960                                                                  pam_unix(sshd:auth): chec
## 99961              pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser=
## 99962                                                    Failed password for root from 183.62.140
## 99963                                                  Received disconnect from 183.62.140.253: 1
## 99964  pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=183.6
## 99965                                           Failed password for invalid user user from 103.99.0
```

The dimension of log_files is 99965 x 5. The class is data frame. The head() and tail() were used to compare to the original MergedAuth.log file.

Now that I built the data frame based on the 5 components of the log file, I needed to add one more column that would specify which log file a message is under.

Adding the Additional Logfile Column

```
starts = grepl("^#", file) # logical
head(which(starts)) # index
```

```
## [1]     1 86841 93963 95964 97965
```

The first step to identify which message came from one of the five files was to grab the log file titles in the original file by using grepl(). I did not use grep(), which returns the actual index of the pattern. I will elaborate further down below.

```
g = cumsum(starts) # cumulative sum
```

The reason I used grepl() was because of the fact that I would have to use cumsum() later. In the code above, the variable "starts" returns a logical vector of TRUE and FALSE, which can also be represented as 1 and 0. In starts, it will return TRUE when it matches ^#, or a log file name, and FALSE otherwise. If we use cumsum(starts), it will produce an integer vector consisting of values from 1:5 that will steadily increase only when it finds another TRUE value (another log file name) because we have 5 different log file names.

From this, we can identify which message is part of a specific log file by matching it to the corresponding number, which represents the log file name.

Now, lets split the original file by variable g

```
gtext = split(file, g) # split by 1:5 factor
```

Splitting the entire file by cumsum(starts), we get a list that includes 5 different large character files. The different files show us how many messages are in each log file.

```
tbl = sapply(gtext, function(x)
  grepl("^#", x[1]))
table(tbl)
```

```
## tbl
## TRUE
##    5
```

```
Filename = g[tbl]
```

We store the information in Filename, and add it to our log_files data frame.

```
log_files = cbind(log_files, Filename)
```

After the column bind to log_files, I replaced the numbers with the actual log file name using gsub(). I also cleaned the df by removing the actual log file names within and setting the row names to NULL.

```
startpos = grep("^#", file)

# Adding additional column for log_file names
log_files$Filename = gsub("1", "auth.log", log_files$Filename)
log_files$Filename = gsub("2", "auth2.log", log_files$Filename)
log_files$Filename = gsub("3", "loghub/Linux/Linux_2k.log", log_files$Filename)
log_files$Filename = gsub("4", "loghub/Mac/Mac_2k.log", log_files$Filename)
log_files$Filename =  gsub("5", "loghub/OpenSSH/SSH_2k.log", log_files$Filename)

# Now remove log file names
log_files = log_files[-c(startpos), ]
row.names(log_files) = NULL # reorder the row values

# Trim white spaces for values
for (i in 1:ncol(log_files)) {
  log_files[, i] = trimws(log_files[, i])
}
```

This is how our log_files looks like now:

```
head(log_files)
```

```
##         Date_Time             Host  App    ID
## 1 Nov 30 06:39:00 ip-172-31-27-153 CRON 21882
```

```
## 2 Nov 30 06:47:01 ip-172-31-27-153 CRON 22087
## 3 Nov 30 06:47:03 ip-172-31-27-153 CRON 22087
## 4 Nov 30 07:07:14 ip-172-31-27-153 sshd 22116
## 5 Nov 30 07:07:35 ip-172-31-27-153 sshd 22118
## 6 Nov 30 07:08:13 ip-172-31-27-153 sshd 22120
##                                                          Message Filename
## 1            pam_unix(cron:session): session closed for user root auth.log
## 2 pam_unix(cron:session): session opened for user root by (uid=0) auth.log
## 3            pam_unix(cron:session): session closed for user root auth.log
## 4                  Connection closed by 122.225.103.87 [preauth] auth.log
## 5                  Connection closed by 122.225.103.87 [preauth] auth.log
## 6                  Connection closed by 122.225.103.87 [preauth] auth.log
```

```
tail(log_files)
```

```
##             Date_Time  Host  App    ID
## 99955 Dec 10 11:04:42 LabSZ sshd 25539
## 99956 Dec 10 11:04:42 LabSZ sshd 25539
## 99957 Dec 10 11:04:43 LabSZ sshd 25541
## 99958 Dec 10 11:04:43 LabSZ sshd 25541
## 99959 Dec 10 11:04:43 LabSZ sshd 25544
## 99960 Dec 10 11:04:45 LabSZ sshd 25539
##
## 99955                                                                pam_unix(sshd:auth): check
## 99956             pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser=
## 99957                                                    Failed password for root from 183.62.140.
## 99958                                                  Received disconnect from 183.62.140.253: 11
## 99959 pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=183.62
## 99960                                           Failed password for invalid user user from 103.99.0.
##                        Filename
## 99955 loghub/OpenSSH/SSH_2k.log
## 99956 loghub/OpenSSH/SSH_2k.log
## 99957 loghub/OpenSSH/SSH_2k.log
## 99958 loghub/OpenSSH/SSH_2k.log
## 99959 loghub/OpenSSH/SSH_2k.log
## 99960 loghub/OpenSSH/SSH_2k.log
```

The data frame looks good. Now all that is left to check is for NAs in the App/ID columns.

```
log_files$Date_Time = as.POSIXct(strptime(log_files$Date_Time, "%b %d %t %H:%M:%S"))
table(is.na(log_files$timestamp))
```

```
## < table of extent 0 >
```

```
# Checking for NAs
log_files$ID = replace(log_files$ID, log_files$ID == "", NA) # replace empty space with NA
length(which(is.na(log_files$ID), arr.ind = TRUE)) # count of NAs in PID -> 946
```

```
## [1] 946
```

```
log_files$App = replace(log_files$App, log_files$App == "", NA)
length(which(is.na(log_files$App), arr.ind = TRUE)) # count of NAs in App -> 0
```

```
## [1] 0
```

I converted all the date_times to POSIXct and checked for NA values afterwards. There were no NAs, so the format was correct. In addition, I replaced all the potential empty spaces in the App/ID columns with NAs, and checked the counts. There were 946 counts of NAs within IDs, and 0 for Apps.

This is the output of the final log_files:

```
head(log_files)
```

```
##             Date_Time              Host  App    ID
## 1 2023-11-30 06:39:00 ip-172-31-27-153 CRON 21882
## 2 2023-11-30 06:47:01 ip-172-31-27-153 CRON 22087
## 3 2023-11-30 06:47:03 ip-172-31-27-153 CRON 22087
## 4 2023-11-30 07:07:14 ip-172-31-27-153 sshd 22116
## 5 2023-11-30 07:07:35 ip-172-31-27-153 sshd 22118
## 6 2023-11-30 07:08:13 ip-172-31-27-153 sshd 22120
##                                                        Message Filename
## 1            pam_unix(cron:session): session closed for user root auth.log
## 2 pam_unix(cron:session): session opened for user root by (uid=0) auth.log
## 3            pam_unix(cron:session): session closed for user root auth.log
## 4                   Connection closed by 122.225.103.87 [preauth] auth.log
## 5                   Connection closed by 122.225.103.87 [preauth] auth.log
## 6                   Connection closed by 122.225.103.87 [preauth] auth.log
```

```
tail(log_files)
```

```
##                 Date_Time  Host  App    ID
## 99955 2023-12-10 11:04:42 LabSZ sshd 25539
## 99956 2023-12-10 11:04:42 LabSZ sshd 25539
## 99957 2023-12-10 11:04:43 LabSZ sshd 25541
## 99958 2023-12-10 11:04:43 LabSZ sshd 25541
## 99959 2023-12-10 11:04:43 LabSZ sshd 25544
## 99960 2023-12-10 11:04:45 LabSZ sshd 25539
##
## 99955                                                                    pam_unix(sshd:auth): check
## 99956             pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser=
## 99957                                                      Failed password for root from 183.62.140.
## 99958                                                    Received disconnect from 183.62.140.253: 11
## 99959 pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=183.62
## 99960                                              Failed password for invalid user user from 103.99.0.
##                        Filename
## 99955 loghub/OpenSSH/SSH_2k.log
## 99956 loghub/OpenSSH/SSH_2k.log
## 99957 loghub/OpenSSH/SSH_2k.log
## 99958 loghub/OpenSSH/SSH_2k.log
## 99959 loghub/OpenSSH/SSH_2k.log
## 99960 loghub/OpenSSH/SSH_2k.log
```

Some last important double checks:

```
stopifnot(nrow(log_files) == (length(file) - 5))
dim(log_files)
```

```
## [1] 99960     6
```

I had to double check that the number of rows in the data frame was equal to the length of the file. I subtracted five from the file because I removed the log file names from my data frame. The dimension of log_files looks good.

Part II.) VALIDATING AND EXPLORING LOG.FILES

Verifying PIDs

```
table(grepl("^[0-9]+$", log_files$ID))
```

```
##
## FALSE  TRUE
##   946 99014
```

```
table(log_files$ID == as.numeric(log_files$ID))
```

```
##
##  TRUE
## 99014
```

```
summary(as.numeric(log_files$ID))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##       0    7418   16720   16302   24849   39203     946
```

It is already known that 946 of the PIDs are not numbers (NAs). I re-verified this using table() and grepl([0-9]), and found that 946 of these values were not numerical. If we do not consider the NA values, then the remaining 99014 values are numeric. I confirmed this through summary(as.numeric).

Lines in Each File

```
summary(gtext)
```

```
##   Length Class  Mode
## 1 86840  -none- character
## 2  7122  -none- character
## 3  2001  -none- character
## 4  2001  -none- character
## 5  2001  -none- character
```

We know that from applying the cumulative sums, we get 5 groups. We count how many appearances there are for each group. For each of the groups (1:5), we want to subtract by 1 to remove the log file name in gtext and only include the messages.

auth.log = 86839

auth2.log = 7121

loghub/Linux/Linux_2k.log = 2000

loghub/Mac/Mac_2k.log = 2000

loghub/OpenSSH/SSH_2k.log = 2000

This should also sum up to the length of our originally modified file = 99960 (removed whitespaces and 5 log file names), which is true.

Range of Date Times for Whole and Individual Log Files

```
# DATETIME FOR ALL MESSAGES
date_time_all = c(min(log_files$Date_Time), max(log_files$Date_Time))
date_time_all
```

```
## [1] "2023-03-27 13:06:56 PDT" "2023-12-31 22:27:48 PST"
```

The entire log file ranges from March 27 13:06:56 to December 31 22:27:48. I verified this using the min() and max() functions. The functions match correctly because the date_time variables are of class POSIXct. Just in case, I manually verified the max and min by finding it in the MergedAuth.log file.

```
five_logfiles = split(log_files, log_files$Filename)

# DATETIME FOR INDIVIDUAL LOG FILES
for (i in 1:length(five_logfiles)) {
  print(c(
    min(five_logfiles[[i]]$Date_Time),
    max(five_logfiles[[i]]$Date_Time)
  ))
}
```

```
## [1] "2023-11-30 06:39:00 PST" "2023-12-31 22:27:48 PST"
## [1] "2023-03-27 13:06:56 PDT" "2023-04-20 14:14:29 PDT"
## [1] "2023-06-14 15:16:01 PDT" "2023-07-27 14:42:00 PDT"
## [1] "2023-07-01 09:00:55 PDT" "2023-07-08 08:10:46 PDT"
## [1] "2023-12-10 06:55:46 PST" "2023-12-10 11:04:45 PST"
```

```
# DAYS SPANNED FOR INDIVIDUAL LOG FILES
for (i in 1:length(five_logfiles)) {
  print(difftime(as.Date(max(
    five_logfiles[[i]]$Date_Time
  )), as.Date(min(
    five_logfiles[[i]]$Date_Time
  ))))
}
```

```
## Time difference of 32 days
## Time difference of 24 days
## Time difference of 43 days
## Time difference of 7 days
## Time difference of 0 secs
```

Auth.log = November 30 6:39:00 to December 31 22:27:48, 32 day span

Auth2.log = March 27 13:06:56 to April 20 14:14:29, 24 day span

Loghub/Linux/Linux_2k.log = June 14 15:16:01 to July 27 14:42:00, 43 day span

Loghub/Mac/Mac_2k.log = July 1 9:00:55 to July 8 08:10:46, 7 day span

Loghub/OpenSSH/SSH_2k.log = December 10 06:55:46 to December 10 11:04:45, 0 day span

Same concept of correct functionality applies to difftime() to find the span of days for each individual log file.

Applications/Versions

```
unique(log_files$App)
```

```
##   [1] "CRON"
##   [2] "sshd"
##   [3] "systemd-logind"
##   [4] "systemd"
##   [5] "sudo"
##   [6] "su"
##   [7] "chpasswd"
##   [8] "useradd"
##   [9] "groupadd"
##  [10] "sshd(pam_unix)"
##  [11] "su(pam_unix)"
##  [12] "logrotate"
##  [13] "ftpd"
##  [14] "cups"
##  [15] "syslogd 1.4.1"
##  [16] "snmpd"
##  [17] "klogind"
##  [18] "gpm"
##  [19] "login(pam_unix)"
##  [20] "-- root"
##  [21] "udev"
##  [22] "gdm(pam_unix)"
##  [23] "gdm-binary"
##  [24] "named"
##  [25] "xinetd"
##  [26] "syslog"
##  [27] "kernel"
##  [28] "irqbalance"
##  [29] "portmap"
##  [30] "rpc.statd"
##  [31] "nfslock"
##  [32] "rpcidmapd"
##  [33] "random"
##  [34] "rc"
##  [35] "sysctl"
##  [36] "hcid"
##  [37] "bluetooth"
##  [38] "network"
##  [39] "sdpd"
##  [40] "com.apple.CDScheduler"
##  [41] "QQ"
##  [42] "mDNSResponder"
##  [43] "symptomsd"
##  [44] "configd"
##  [45] "com.apple.cts"
##  [46] "corecaptured"
##  [47] "com.apple.WebKit.WebContent"
##  [48] "networkd"
##  [49] "netbiosd"
```

```
##  [50] "sandboxd"
##  [51] "Dock"
##  [52] "AddressBookSourceSync"
##  [53] "secd"
##  [54] "SpotlightNetHelper"
##  [55] "sharingd"
##  [56] "locationd"
##  [57] "UserEventAgent"
##  [58] "Dropbox"
##  [59] "cdpd"
##  [60] "com.apple.AddressBook.InternetAccountsBridge"
##  [61] "blued"
##  [62] "VDCAssistant"
##  [63] "WindowServer"
##  [64] "GoogleSoftwareUpdateAgent"
##  [65] "iconservicesagent"
##  [66] "quicklookd"
##  [67] "Safari"
##  [68] "cloudd"
##  [69] "Preview"
##  [70] "CalendarAgent"
##  [71] "syslogd"
##  [72] "ksfetch"
##  [73] "com.apple.xpc.launchd"
##  [74] "com.apple.ncplugin.WorldClock"
##  [75] "com.apple.ncplugin.weather"
##  [76] "com.apple.geod"
##  [77] "ntpd"
##  [78] "hidd"
##  [79] "pkd"
##  [80] "CrashReporterSupportHelper"
##  [81] "wirelessproxd"
##  [82] "identityservicesd"
##  [83] "BezelServices 255.10"
##  [84] "WeChat"
##  [85] "mdworker"
##  [86] "com.apple.AddressBook.ContactsAccountsService"
##  [87] "SCIM"
##  [88] "com.apple.SecurityServer"
##  [89] "Microsoft Word"
##  [90] "garcon"
##  [91] "QuickLookSatellite"
##  [92] "Google Chrome"
##  [93] "imagent"
##  [94] "Mail"
##  [95] "TCIM"
##  [96] "loginwindow"
##  [97] "ChromeExistion"
##  [98] "AirPlayUIAgent"
##  [99] "taskgated"
## [100] "com.apple.WebKit.Networking"
## [101] "mds"
## [102] "GPUToolsAgent"
## [103] "NeteaseMusic"
```

```
## [104] "CommCenter"
```

Through visual inspection, I see that majority of the apps do not contain numbers. There are apps that seem to have the numbers as additional structure, such as BezelServices 255.10 and Syslogd 1.4.1.

Host Value for Individual Files

```
# General
starts = grep("^#", file, value = TRUE)
unique(log_files$Host)
```

```
##  [1] "ip-172-31-27-153"          "ip-10-77-20-248"
##  [3] "combo"                     "calvisitor-10-105-160-95"
##  [5] "authorMacBook-Pro"         "calvisitor-10-105-163-202"
##  [7] "calvisitor-10-105-160-237" "calvisitor-10-105-160-184"
##  [9] "calvisitor-10-105-162-32"  "calvisitor-10-105-161-225"
## [11] "airbears2-10-142-110-255"  "calvisitor-10-105-162-105"
## [13] "calvisitor-10-105-163-10"  "calvisitor-10-105-160-179"
## [15] "calvisitor-10-105-160-226" "calvisitor-10-105-162-98"
## [17] "calvisitor-10-105-162-107" "airbears2-10-142-108-38"
## [19] "calvisitor-10-105-163-9"   "calvisitor-10-105-160-210"
## [21] "calvisitor-10-105-162-81"  "calvisitor-10-105-161-231"
## [23] "calvisitor-10-105-160-22"  "calvisitor-10-105-162-211"
## [25] "calvisitor-10-105-162-138" "calvisitor-10-105-163-28"
## [27] "calvisitor-10-105-160-37"  "calvisitor-10-105-163-168"
## [29] "calvisitor-10-105-163-253" "calvisitor-10-105-162-178"
## [31] "calvisitor-10-105-160-205" "calvisitor-10-105-161-77"
## [33] "calvisitor-10-105-160-85"  "calvisitor-10-105-160-47"
## [35] "calvisitor-10-105-163-147" "calvisitor-10-105-162-175"
## [37] "calvisitor-10-105-162-108" "calvisitor-10-105-162-228"
## [39] "calvisitor-10-105-161-176" "calvisitor-10-105-160-181"
## [41] "calvisitor-10-105-162-124" "LabSZ"
```

```
length(unique(log_files$Host))
```

```
## [1] 42
```

```
# Auth.log
unique(five_logfiles[[1]]$Host)
```

```
## [1] "ip-172-31-27-153"
```

```
# Auth2.log
unique(five_logfiles[[2]]$Host)
```

```
## [1] "ip-10-77-20-248"
```

```r
# Loghub/Linux
unique(five_logfiles[[3]]$Host)
```

```
## [1] "combo"
```

```r
# Loghub Mac
unique(five_logfiles[[4]]$Host)
```

```
##  [1] "calvisitor-10-105-160-95"  "authorMacBook-Pro"
##  [3] "calvisitor-10-105-163-202" "calvisitor-10-105-160-237"
##  [5] "calvisitor-10-105-160-184" "calvisitor-10-105-162-32"
##  [7] "calvisitor-10-105-161-225" "airbears2-10-142-110-255"
##  [9] "calvisitor-10-105-162-105" "calvisitor-10-105-163-10"
## [11] "calvisitor-10-105-160-179" "calvisitor-10-105-160-226"
## [13] "calvisitor-10-105-162-98"  "calvisitor-10-105-162-107"
## [15] "airbears2-10-142-108-38"   "calvisitor-10-105-163-9"
## [17] "calvisitor-10-105-160-210" "calvisitor-10-105-162-81"
## [19] "calvisitor-10-105-161-231" "calvisitor-10-105-160-22"
## [21] "calvisitor-10-105-162-211" "calvisitor-10-105-162-138"
## [23] "calvisitor-10-105-163-28"  "calvisitor-10-105-160-37"
## [25] "calvisitor-10-105-163-168" "calvisitor-10-105-163-253"
## [27] "calvisitor-10-105-162-178" "calvisitor-10-105-160-205"
## [29] "calvisitor-10-105-161-77"  "calvisitor-10-105-160-85"
## [31] "calvisitor-10-105-160-47"  "calvisitor-10-105-163-147"
## [33] "calvisitor-10-105-162-175" "calvisitor-10-105-162-108"
## [35] "calvisitor-10-105-162-228" "calvisitor-10-105-161-176"
## [37] "calvisitor-10-105-160-181" "calvisitor-10-105-162-124"
```

```r
# Loghub/OpenSSH
unique(five_logfiles[[5]]$Host)
```

```
## [1] "LabSZ"
```

The host value is constant for:

auth.log (ip-172-31-27-153)

auth2.log (ip-10-77-20-248)

loghub/Linux/Linux_2k.log (combo)

loghub/OpenSSH/SSH_2k.log (LabSZ)

The host is not constant only for (loghub/Mac/Mac_2k.log). There are 38 different hosts.

Most common App on different hosts

The approach for this was to create a frequency table for the counts and find the max counts of an app for a particular Host column.

```r
apps = table(log_files$App, log_files$Host) # Create a table showing frequency counts of Apps for each
apply(apps, 2, FUN = max) # Actual value of Max counts of an app (not the actual name)
```

```
##   airbears2-10-142-108-38  airbears2-10-142-110-255         authorMacBook-Pro
##                         4                        35                       192
## calvisitor-10-105-160-179 calvisitor-10-105-160-181 calvisitor-10-105-160-184
##                         4                         5                        18
## calvisitor-10-105-160-205 calvisitor-10-105-160-210  calvisitor-10-105-160-22
##                        16                         6                         5
## calvisitor-10-105-160-226 calvisitor-10-105-160-237  calvisitor-10-105-160-37
##                         6                        27                         9
##  calvisitor-10-105-160-47  calvisitor-10-105-160-85  calvisitor-10-105-160-95
##                         5                        26                        67
## calvisitor-10-105-161-176 calvisitor-10-105-161-225 calvisitor-10-105-161-231
##                         3                         5                         2
##  calvisitor-10-105-161-77 calvisitor-10-105-162-105 calvisitor-10-105-162-107
##                         2                       105                         8
## calvisitor-10-105-162-108 calvisitor-10-105-162-124 calvisitor-10-105-162-138
##                         2                        14                         2
## calvisitor-10-105-162-175 calvisitor-10-105-162-178 calvisitor-10-105-162-211
##                         3                       100                         2
## calvisitor-10-105-162-228  calvisitor-10-105-162-32  calvisitor-10-105-162-81
##                         4                        18                         2
##  calvisitor-10-105-162-98  calvisitor-10-105-163-10 calvisitor-10-105-163-147
##                         7                        21                         3
## calvisitor-10-105-163-168 calvisitor-10-105-163-202 calvisitor-10-105-163-253
##                         4                        43                         6
##  calvisitor-10-105-163-28   calvisitor-10-105-163-9                     combo
##                         2                         2                       916
##           ip-10-77-20-248          ip-172-31-27-153                     LabSZ
##                      4095                     85246                      2000
```

```
apps = as.data.frame.matrix(as.matrix(apps)) # Turn into df
common_app = rownames(apps)[apply(apps, 2, which.max)] # Apply to find the index at which the max occur
```

The numbers above corresponding to each Host represents the max number of counts (most frequent) for a particular App.

To see which Apps were the most common:

```
table(common_app)
```

```
## common_app
##              ChromeExistion com.apple.WebKit.WebContent
##                           1                           1
##                corecaptured                        ftpd
##                           1                           1
##                      kernel                      Safari
##                          33                           1
##                    sandboxd                       sshd
##                           1                           3
```

There are many frequencies of the app “Kernel” on different hosts (33). The numbers above sum up to 42, which correspond to the number of hosts and verifies that this is correct.

```r
tail(rbind(apps, "Most_Common" = common_app), 1)
```

```
##             airbears2-10-142-108-38 airbears2-10-142-110-255 authorMacBook-Pro
## Most_Common                  kernel                  kernel           kernel
##             calvisitor-10-105-160-179 calvisitor-10-105-160-181
## Most_Common                    Safari                    kernel
##             calvisitor-10-105-160-184 calvisitor-10-105-160-205
## Most_Common                    kernel                    kernel
##             calvisitor-10-105-160-210 calvisitor-10-105-160-22
## Most_Common                    kernel                   kernel
##             calvisitor-10-105-160-226 calvisitor-10-105-160-237
## Most_Common                    kernel                    kernel
##             calvisitor-10-105-160-37 calvisitor-10-105-160-47
## Most_Common                   kernel                   kernel
##             calvisitor-10-105-160-85 calvisitor-10-105-160-95
## Most_Common                   kernel                   kernel
##             calvisitor-10-105-161-176   calvisitor-10-105-161-225
## Most_Common                    kernel com.apple.WebKit.WebContent
##             calvisitor-10-105-161-231 calvisitor-10-105-161-77
## Most_Common                    kernel                    kernel
##             calvisitor-10-105-162-105 calvisitor-10-105-162-107
## Most_Common                    kernel                    kernel
##             calvisitor-10-105-162-108 calvisitor-10-105-162-124
## Most_Common                    kernel                    kernel
##             calvisitor-10-105-162-138 calvisitor-10-105-162-175
## Most_Common                  sandboxd              corecaptured
##             calvisitor-10-105-162-178 calvisitor-10-105-162-211
## Most_Common                    kernel                    kernel
##             calvisitor-10-105-162-228 calvisitor-10-105-162-32
## Most_Common                    kernel                   kernel
##             calvisitor-10-105-162-81 calvisitor-10-105-162-98
## Most_Common                   kernel                   kernel
##             calvisitor-10-105-163-10 calvisitor-10-105-163-147
## Most_Common                   kernel                    kernel
##             calvisitor-10-105-163-168 calvisitor-10-105-163-202
## Most_Common                    kernel                    kernel
##             calvisitor-10-105-163-253 calvisitor-10-105-163-28
## Most_Common            ChromeExistion                   kernel
##             calvisitor-10-105-163-9 combo ip-10-77-20-248 ip-172-31-27-153
## Most_Common                  kernel  ftpd            sshd             sshd
##             LabSZ
## Most_Common  sshd
```

LOGINS - VALID AND INVALID

Valid Logins - User/IP

My approach to finding valid/successful logins was to search for related keywords within the entire file. I used the following (ignoring case) keywords to represent this idea: Accepted, New Session, Connection From, Systemd-login, and Session Opened. There were 3796 lines.

```
valid = grep(
  "(accepted|new session|connection from|systemd-login|session opened)",
  file,
  value = TRUE,
  ignore.case = TRUE
) # 3796

table(
  grepl(
    "(accepted|new session|connection from|systemd-login|session opened)",
    file,
    ignore.case = TRUE
  )
) # 3796
```

```
##
## FALSE  TRUE
## 96169  3796
```

After extracting the successful messages, I essentially created a subset of my original data frame with only the messages with the successful related keywords (still including the 5 components of the message). Then, I extracted only the "message" component pf the new data frame, and trimmed the whitespaces.

```
# Create new df based on successful messages and extract user/ip from messages column using regex
valid_rx = gregexpr(
  "^(?P<Date_Time>[[:alpha:]]+ [0-9\\s]?[0-9] \\d{2}:\\d{2}:\\d{2}|# .*.log$)\\s?(?P<Host>[a-zA-Z0-9-]*
  valid,
  perl = TRUE
)

# Valid_extract = new df
valid_extract = as.data.frame(t(mapply(function(str, match) {
  s = attr(match, "capture.start")
  substring(str, s, s + attr(match, "capture.length") - 1)
}, valid, valid_rx)))

# Structuring
row.names(valid_extract) = NULL
valid_usersip = data.frame(valid_extract$V5)

# Extract message and trim any white spaces
valid_usersip$valid_extract.V5 = trimws(valid_usersip$valid_extract.V5)
```

The next step is to find the users and IPs of the successful login messages.

Valid Users

```
table(grepl("root", valid, ignore.case = TRUE))
```

```
##
## FALSE  TRUE
##  2172  1624
```

```
# Searching for Users (Process of search and narrow by elimination)
table(
  grepl(
    "(root|ubuntu|elastic_user_[0-9]|test|cyrus|news|fztu)",
    valid,
    ignore.case = TRUE
  )
)
```

```
##
## FALSE  TRUE
##  1155  2641
```

My strategy to find all the users was to first quickly inspect the lines to see if any user existed. By just looking at the file, I saw many instances of the user "root". For clarification of how I knew the user was root:

"Nov 30 06:47:01 ip-172-31-27-153 CRON[22087]: pam_unix(cron:session): session opened for user root by (uid=0)"

The word "root" follows after user. I used this same process to extract the users from most of the other lines. The reason I say most of the other lines is because few lines did not have the "user" before the "name" (i.e root).

I also wanted to know how many other lines shared the same user root, and used grepl() nested in table() to get the counts. Then, I just repeated this process of search and elimination for users in the file. In total, I found that 2641 out of the 3796 successful messages had a specified user.

I wanted to make sure that I extracted ALL of the possible users from the messages. I knew that there possibly existed 3796-2641 = 1155 messages without a user, and I verified that through below:

```
# Non users
na_users = valid[!grepl("(root|ubuntu|elastic_user_[0-9]|test|cyrus|news|fztu)", valid, perl = TRUE)]
head(na_users)
```

```
## [1] "Mar 27 13:06:56 ip-10-77-20-248 systemd-logind[1118]: Watching system buttons on /dev/input/ever
## [2] "Mar 27 13:06:56 ip-10-77-20-248 systemd-logind[1118]: Watching system buttons on /dev/input/ever
## [3] "Mar 27 13:06:56 ip-10-77-20-248 systemd-logind[1118]: New seat seat0."
## [4] "Mar 27 13:44:19 ip-10-77-20-248 systemd-logind[1118]: Removed session 1."
## [5] "Mar 27 14:02:16 ip-10-77-20-248 systemd-logind[1118]: Removed session 3."
## [6] "Mar 27 17:08:23 ip-10-77-20-248 systemd-logind[1118]: Removed session 6."
```

```
length(na_users)
```

```
## [1] 1155
```

```r
# Show that remaining success messages do not have users specified
head(grep("connection from", na_users, value = TRUE, ignore.case = TRUE))
```

```
## [1] "Mar 28 19:13:31 ip-10-77-20-248 sshd[29543]: Connection from 85.245.107.41 port 61663 on 10.77.
## [2] "Mar 28 19:13:50 ip-10-77-20-248 sshd[29628]: Connection from 85.245.107.41 port 61667 on 10.77.
## [3] "Mar 28 20:14:00 ip-10-77-20-248 sshd[29936]: Connection from 186.219.213.14 port 59547 on 10.77
## [4] "Mar 28 20:21:08 ip-10-77-20-248 sshd[29951]: Connection from 85.245.107.41 port 63494 on 10.77.
## [5] "Mar 28 20:21:20 ip-10-77-20-248 sshd[29953]: Connection from 85.245.107.41 port 63497 on 10.77.
## [6] "Mar 28 20:21:52 ip-10-77-20-248 sshd[30039]: Connection from 85.245.107.41 port 63502 on 10.77.
```

```r
table(grepl("connection from|removed session|new seat|watching", na_users, ignore.case = TRUE))
```

```
##
## TRUE
## 1155
```

```r
na_users[!grepl("connection from|removed session|new seat|watching", na_users, ignore.case = TRUE)]
```

```
## character(0)
```

I extracted the lines that did not specify a user that I saw in the successful messages and put the information into var na_users. As expected, there were 1155 messages.

Then, I constructed another table() + grepl() expression to extract the messages that I know do not have users by visual inspection. The table counted 1155 of these expressions as TRUE, which verifies my statement.

Using the pattern I constructed for users, I put it in gregexpr() and obtained the capture groups. After extracting the groups, I manipulated the data frame and cleaned it to contain all the information needed.

```r
# Get expression for finding users
user_rx = gregexpr("(root|ubuntu|elastic_user_[0-9]|test|cyrus|news|fztu)", valid, perl = TRUE)

users_success = mapply(function(str,match){
  s = attr(match, "capture.start")
  substring(str,s,s+attr(match,"capture.length")-1)
}, valid, user_rx)

# Data Manipulation
users_success = as.data.frame(users_success)
users_success = t(users_success)
users_success = users_success[,-2]
users_success = as.data.frame(users_success)
row.names(users_success) = NULL

# Equals 1155, same value above when computing for NA users (verified)
table(users_success[users_success == ""])
```

```
##
##
## 1155
```

```
# Replace no users with NA
users_success$users_success = replace(users_success$users_success, users_success$users_success == "", NA

col = cbind(users_success, valid_usersip$valid_extract.V5)
head(col)
```

```
##   users_success                                    valid_usersip$valid_extract.V5
## 1          root pam_unix(cron:session): session opened for user root by (uid=0)
## 2          root pam_unix(cron:session): session opened for user root by (uid=0)
## 3          root pam_unix(cron:session): session opened for user root by (uid=0)
## 4          root pam_unix(cron:session): session opened for user root by (uid=0)
## 5          root pam_unix(cron:session): session opened for user root by (uid=0)
## 6          root pam_unix(cron:session): session opened for user root by (uid=0)
```

Valid IPs

I used a similar process to find the IPs for each successful login. I knew that the syntax for a regular IP contained only numbers and dots. From this, I constructed a regular expression to extract this specific pattern.

```
# Searching for IP
table(grepl("(\\d+\\.\\d+\\.\\d+\\.\\d+)", valid, ignore.case = TRUE)) # 1153 counts
```

```
##
## FALSE  TRUE
##  2643  1153
```

```
tail(valid[!grepl("(\\d+\\.\\d+\\.\\d+\\.\\d+)", valid, perl = TRUE)])
```

```
## [1] "Jul 27 04:16:07 combo su(pam_unix)[30999]: session opened for user cyrus by (uid=0)"
## [2] "Jul 27 04:21:39 combo su(pam_unix)[31373]: session opened for user news by (uid=0)"
## [3] "Jul  2 17:56:40 calvisitor-10-105-163-202 com.apple.ncplugin.WorldClock[32583]: host connection
## [4] "Jul  4 13:56:31 calvisitor-10-105-162-105 com.apple.AddressBook.ContactsAccountsService[289]: [
## [5] "Jul  8 06:02:25 calvisitor-10-105-162-124 com.apple.AddressBook.ContactsAccountsService[289]: [
## [6] "Dec 10 09:32:20 LabSZ sshd[24680]: pam_unix(sshd:session): session opened for user fztu by (uid=
```

Next, we want to create a data frame based on the sub pattern of IP successes.

```
library(stringr)
# Extracts IPs
ip_success = as.data.frame(str_extract(valid, regex("(\\d+\\.\\d+\\.\\d+\\.\\d+)")))
colnames(ip_success) = "IP"
dim(ip_success)
```

```
## [1] 3796    1
```

```
# Counts of Unique IPs
nrow(unique(ip_success))
```

```
## [1] 46
```

```r
# Combine DF for successful logins
success = cbind(valid_usersip, users_success, ip_success)
success = success[c("users_success","IP","valid_extract.V5")]
colnames(success) = c("User","IP","Message")

# Showing Results
tail(success)
```

```
##      User             IP
## 3791 <NA>    218.38.58.3
## 3792 <NA>           <NA>
## 3793 <NA>           <NA>
## 3794 <NA>           <NA>
## 3795 fztu 119.137.62.142
## 3796 fztu           <NA>
##
## 3791                                                                    connection from 218.38.58.3 () at
## 3792                                  host connection <NSXPCConnection: 0x7fddbbb015c0> connection fi
## 3793   [Accounts] Current connection, <NSXPCConnection: 0x7fda74805bf0> connection from pid 487, does
## 3794 [Accounts] Current connection, <NSXPCConnection: 0x7fda72614240> connection from pid 30318, does
## 3795                                                                     Accepted password for fztu from 119.13
## 3796                                                                   pam_unix(sshd:session): session opened
```

Now, we have a table that shows the successful users and IPs for the successful log file messages subsetted from the MergedAuth.log file.

Invalid Logins - User/IP

I did virtually the same process for invalid as I did with valid. The only difference is the usage of different keywords.

```
invalid = grep(
  "(failed|fatal|warning|invalid|error|failure|fail)",
  file,
  value = TRUE,
  ignore.case = TRUE
) #39128

table(
  grepl(
    "(failed|fatal|warning|invalid|error|failure|fail)",
    invalid,
    ignore.case = TRUE
  )
)
```

```
##
##  TRUE
## 39128
```

```
# Regex for Invalid User
invalid_rx = gregexpr("^(?P<Date_Time>[[:alpha:]]+ [0-9\\s]?[0-9] \\d{2}:\\d{2}:\\d{2}|# .*.log$)\\s?(?

# Create invalid data frame
invalid_extract = as.data.frame(t(mapply(function(str,match){
  s = attr(match, "capture.start")
  substring(str,s,s+attr(match,"capture.length")-1)
}, invalid, invalid_rx)))

# Structuring
row.names(invalid_extract) = NULL
invalid_userip = data.frame(invalid_extract$V5)

invalid_userip$invalid_extract.V5= trimws(invalid_userip$invalid_extract.V5)
head(invalid_userip$invalid_extract.V5)
```

```
## [1] "Invalid user admin from 187.12.249.74"
## [2] "input_userauth_request: invalid user admin [preauth]"
## [3] "Invalid user admin from 122.225.109.208"
## [4] "input_userauth_request: invalid user admin [preauth]"
## [5] "fatal: Read from socket failed: Connection reset by peer [preauth]"
## [6] "fatal: Read from socket failed: Connection reset by peer [preauth]"
```

I found 39128 counts failures/invalid logins for the entire file based on the keywords I selected. I also extracted only the "message" component of the entire log file message for parsing later on.

Invalid Users

Out of the 39128 counts of invalid log file messages, I found that 30084 of those lines have users, while the rest (9044) do not.

```
# Non users
na_users_invalid = invalid[!grepl(
  "(root|ubuntu|elastic_user_[0-9]|test|cyrus|news|fztu|user [a-z]+|user [0-9]+)",
  invalid,
  perl = TRUE,
  ignore.case = TRUE
)]
head(na_users_invalid)
```

```
## [1] "Nov 30 11:54:50 ip-172-31-27-153 sshd[22341]: fatal: Read from socket failed: Connection reset b
## [2] "Nov 30 14:04:18 ip-172-31-27-153 sshd[22463]: fatal: Read from socket failed: Connection reset b
## [3] "Nov 30 14:04:59 ip-172-31-27-153 sshd[22500]: fatal: Read from socket failed: Connection reset b
## [4] "Nov 30 14:05:12 ip-172-31-27-153 sshd[22508]: fatal: Read from socket failed: Connection reset b
## [5] "Nov 30 14:05:24 ip-172-31-27-153 sshd[22517]: fatal: Read from socket failed: Connection reset b
## [6] "Nov 30 14:05:46 ip-172-31-27-153 sshd[22528]: fatal: Read from socket failed: Connection reset b
```

```
length(na_users_invalid)
```

```
## [1] 9044
```

For below, I created the data frame for invalid users.

```
# Get expression for finding users
user_invalid_rx = gregexpr(
  "(root|ubuntu|elastic_user_[0-9]|test|cyrus|news|fztu|user [a-z]+|user [0-9]+)",
  invalid,
  perl = TRUE,
  ignore.case = TRUE
)

users_invalid = mapply(function(str, match) {
  s = attr(match, "capture.start")
  substring(str, s, s + attr(match, "capture.length") - 1)
}, invalid, user_invalid_rx)

# Data Manipulation
users_invalid = as.data.frame(users_invalid)
users_invalid = t(users_invalid)
users_invalid = users_invalid[, -2]
users_invalid = as.data.frame(users_invalid)
row.names(users_invalid) = NULL

# Equals 9044
table(users_invalid[users_invalid == ""])
```

```
##
##
## 9044
```

```
# Replace no users with NA
users_invalid$users_invalid = replace(users_invalid$users_invalid,
                                      users_invalid$users_invalid == "",
                                      NA)
```

I did some basic data manipulations to clean the df.

Invalid IPs

```
library(stringr)

table(grepl("(\\d+\\.\\d+\\.\\d+\\.\\d+)", invalid, ignore.case = TRUE)) # 20922 counts
```

```
##
## FALSE  TRUE
## 18206 20922
```

```
tail(invalid[!grepl("(\\d+\\.\\d+\\.\\d+\\.\\d+)", invalid, perl = TRUE)])
```

```
## [1] "Dec 10 11:04:12 LabSZ sshd[25492]: input_userauth_request: invalid user ubnt [preauth]"
## [2] "Dec 10 11:04:25 LabSZ sshd[25513]: input_userauth_request: invalid user admin [preauth]"
## [3] "Dec 10 11:04:30 LabSZ sshd[25521]: input_userauth_request: invalid user cisco [preauth]"
## [4] "Dec 10 11:04:34 LabSZ sshd[25527]: input_userauth_request: invalid user test [preauth]"
## [5] "Dec 10 11:04:38 LabSZ sshd[25534]: input_userauth_request: invalid user guest [preauth]"
## [6] "Dec 10 11:04:42 LabSZ sshd[25539]: input_userauth_request: invalid user user [preauth]"
```

```
ip_invalid = as.data.frame(str_extract(invalid, regex("(\\d+\\.\\d+\\.\\d+\\.\\d+)")))
colnames(ip_invalid) = "IP"
dim(ip_invalid)
```

```
## [1] 39128     1
```

```
# IP Count
nrow(unique(ip_invalid))
```

```
## [1] 1692
```

```
invalid_df = cbind(invalid_userip, users_invalid, ip_invalid)
invalid_df = invalid_df[c("users_invalid", "IP", "invalid_extract.V5")]
colnames(invalid_df) = c("User", "IP" , "Invalid")

# Change IPs to factor
invalid_df$IP = as.factor(invalid_df$IP)
table(is.na(invalid_df$IP)) # Did not lose data to coercion
```

```
##
## FALSE  TRUE
## 20922 18206
```

```
# Showing Results
head(invalid_df)
```

```
##         User              IP
## 1 user admin   187.12.249.74
## 2 user admin            <NA>
## 3 user admin 122.225.109.208
## 4 user admin            <NA>
## 5       <NA>            <NA>
## 6       <NA>            <NA>
##                                                                 Invalid
## 1                                Invalid user admin from 187.12.249.74
## 2                input_userauth_request: invalid user admin [preauth]
## 3                                Invalid user admin from 122.225.109.208
## 4                input_userauth_request: invalid user admin [preauth]
## 5 fatal: Read from socket failed: Connection reset by peer [preauth]
## 6 fatal: Read from socket failed: Connection reset by peer [preauth]
```

The strategy for finding invalid IPs was the same with valid IPs. I found 20922 counts of IPs for invalid messages, and 1691 are unique IPs.

After I created the invalid data frame for users and IPs, I conducted some exploratory data analysis.

Multiple Invalid Users from Same IP

To find if multiple invalid users came from the same IP, I first grouped the data frame by IP.

```
library(tidyverse)
multiple_invalid_ip = invalid_df %>% arrange(IP)
head(multiple_invalid_ip)
```

```
##   User             IP
## 1 root  1.189.205.173
## 2 root  1.189.205.173
## 3 root  1.189.205.173
## 4 root  1.189.205.173
## 5 root  1.189.205.173
## 6 <NA> 1.2.840.113635
##
## 1
## 2
## 3
## 4
## 5
## 6 2017-07-04 09:42:57.924 GoogleSoftwareUpdateAgent[34603/0x700000323000] [lvl=2] +[KSCodeSigningVer
```

Next, I wanted to only see the unique IP/User pairs to determine if there were multiple invalid users for every IP.

```
head(unique(multiple_invalid_ip[, c("User", "IP")]), 20) # Total 1692 unique IPs
```

```
##             User             IP
## 1           root  1.189.205.173
## 6           <NA> 1.2.840.113635
## 7   user ftpuser   1.234.21.115
## 8     user admin   1.234.21.115
## 9         user D   1.234.21.115
## 19   user vyatta   1.246.219.50
## 20 user PlcmSpIp   1.246.219.50
## 21    user admin   1.30.211.144
## 22          <NA>   1.30.211.144
## 31    user admin 101.226.249.53
## 34    user agata 101.226.249.53
## 35    user arbab 101.226.249.53
## 36   user oracle 101.231.72.111
## 37    user admin     101.4.63.2
## 38    user guest     101.4.63.2
## 39  user support     101.4.63.2
## 40    user admin 101.64.236.146
## 41    user admin  101.78.202.26
## 42          <NA>   101.99.3.131
## 43    user admin   101.99.3.131
```

```
# Store new df
df1 = unique(multiple_invalid_ip[, c("User", "IP", "Invalid")]) # 9626
```

Now that I have extracted all the unique pairs, I needed to examine which IPs actually had multiple invalid user logins. However before I do this, I noticed that there existed some NA values in the user/IP columns. We want to remove these NAs as they can be misleading.

```
# Remove NAs in User or IP
df1 = df1 %>% filter_at(vars(User,IP), all_vars(!is.na(.))) # 8373 counts
```

We see that there were 1300 counts of NAs. Next, I constructed a tibble to illustrate the user counts per IP. To extract the appropriate IPs, I removed the IPs that did not have multiple counts of (num_users) invalid users.

```
# Get the user count for every unique IP
ip_counts = df1 %>% group_by(IP) %>% summarize(num_users = n_distinct(User))
head(ip_counts, 25)
```

```
## # A tibble: 25 x 2
##    IP              num_users
##    <fct>               <int>
##  1 1.189.205.173           1
##  2 1.234.21.115            3
##  3 1.246.219.50            2
##  4 1.30.211.144            1
##  5 101.226.249.53          3
##  6 101.231.72.111          1
##  7 101.4.63.2              3
##  8 101.64.236.146          1
##  9 101.78.202.26           1
## 10 101.99.3.131            2
## # ... with 15 more rows
```

This shows us how many different users there were for every IP. The next step is to remove the IPs with num_users = 1.

```
# Remove IPs that do not have multiple users
multi_user_ips = ip_counts %>% filter(num_users > 1) %>% select(IP)

# IPs with multiple invalid users
head(multi_user_ips)
```

```
## # A tibble: 6 x 1
##   IP
##   <fct>
## 1 1.234.21.115
## 2 1.246.219.50
## 3 101.226.249.53
## 4 101.4.63.2
## 5 101.99.3.131
## 6 103.10.151.156
```

Now, we just extract the IP values

```
# Finished data including multiple invalid users
multi_invalid_ip = df1 %>% filter(IP %in% multi_user_ips$IP)
dim(multi_invalid_ip)
```

```
## [1] 7168    3
```

```
head(multi_invalid_ip, 20)
```

```
##             User             IP                                  Invalid
## 1   user ftpuser   1.234.21.115    Invalid user ftpuser from 1.234.21.115
## 2     user admin   1.234.21.115      Invalid user admin from 1.234.21.115
## 3         user D   1.234.21.115     Invalid user D-Link from 1.234.21.115
## 4    user vyatta   1.246.219.50     Invalid user vyatta from 1.246.219.50
## 5  user PlcmSpIp   1.246.219.50   Invalid user PlcmSpIp from 1.246.219.50
## 6     user admin 101.226.249.53    Invalid user admin from 101.226.249.53
## 7     user agata 101.226.249.53    Invalid user agata from 101.226.249.53
## 8     user arbab 101.226.249.53    Invalid user arbab from 101.226.249.53
## 9     user admin     101.4.63.2        Invalid user admin from 101.4.63.2
## 10    user guest     101.4.63.2        Invalid user guest from 101.4.63.2
## 11  user support     101.4.63.2      Invalid user support from 101.4.63.2
## 12    user admin   101.99.3.131      Invalid user admin from 101.99.3.131
## 13    user arbab   101.99.3.131      Invalid user arbab from 101.99.3.131
## 14        user D 103.10.151.156   Invalid user D-Link from 103.10.151.156
## 15  user ftpuser 103.10.151.156  Invalid user ftpuser from 103.10.151.156
## 16    user admin 103.10.151.156    Invalid user admin from 103.10.151.156
## 17  user ftpuser 103.12.158.218  Invalid user ftpuser from 103.12.158.218
## 18 user PlcmSpIp 103.12.158.218 Invalid user PlcmSpIp from 103.12.158.218
## 19   user vyatta 103.12.158.218   Invalid user vyatta from 103.12.158.218
## 20  user default   103.12.84.51    Invalid user default from 103.12.84.51
```

```
tail(multi_invalid_ip, 20)
```

```
##              User            IP                                  Invalid
## 7149 user ftpuser  98.109.76.36  Invalid user ftpuser from 98.109.76.36
## 7150   user admin  98.109.76.36    Invalid user admin from 98.109.76.36
## 7151       user D  98.109.76.36   Invalid user D-Link from 98.109.76.36
## 7152   user xbian  98.109.76.36    Invalid user xbian from 98.109.76.36
## 7153    user xbmc 98.130.211.38    Invalid user xbmc from 98.130.211.38
## 7154   user debug 98.130.211.38   Invalid user debug from 98.130.211.38
## 7155 user ftpuser 98.130.222.45 Invalid user ftpuser from 98.130.222.45
## 7156   user admin 98.130.222.45   Invalid user admin from 98.130.222.45
## 7157       user D 98.130.222.45  Invalid user D-Link from 98.130.222.45
## 7158    user xbmc 98.130.222.45    Invalid user xbmc from 98.130.222.45
## 7159   user arbab 98.130.222.45   Invalid user arbab from 98.130.222.45
## 7160   user xbian 98.130.222.45   Invalid user xbian from 98.130.222.45
## 7161   user agata 98.130.222.45   Invalid user agata from 98.130.222.45
## 7162    user bill 98.130.222.45    Invalid user bill from 98.130.222.45
## 7163   user debug 98.130.222.45   Invalid user debug from 98.130.222.45
## 7164 user default 98.130.222.45 Invalid user default from 98.130.222.45
## 7165 user dreamer 98.130.222.45 Invalid user dreamer from 98.130.222.45
## 7166 user ftpuser  98.191.25.65  Invalid user ftpuser from 98.191.25.65
## 7167   user admin  98.191.25.65    Invalid user admin from 98.191.25.65
## 7168       user D  98.191.25.65   Invalid user D-Link from 98.191.25.65
```

I only used the first 20 rows as demonstration. But we see that there are multiple invalid users for one IP address. There are 7168 rows in this new data frame.

Valid Logins from IP with Multiple Users

To find valid logins that correspond to the IP addresses obtained for multiple invalid users above, I used inner_join() on the data frames containing the successful messages and failed messages. By joining the two by their unique IP and USER, we can see if there are any matches that indicates that there were both valid and invalid logins from the IPs.

```
joined_df = inner_join(success, multi_invalid_ip, by = c("User", "IP"))
head(joined_df[!is.na(joined_df$IP),] %>% group_by(User), 10)
```

```
## # A tibble: 10 x 4
## # Groups:   User [3]
##    User           IP            Message                 Invalid
##    <chr>          <chr>         <chr>                   <chr>
##  1 elastic_user_2 85.245.107.41 Accepted password for ~ Failed publickey for el~
##  2 elastic_user_2 85.245.107.41 Accepted password for ~ pam_unix(sshd:auth): au~
##  3 elastic_user_2 85.245.107.41 Accepted password for ~ Failed password for ela~
##  4 elastic_user_2 85.245.107.41 Accepted password for ~ Failed password for ela~
##  5 elastic_user_8 85.245.107.41 Accepted password for ~ Failed publickey for el~
##  6 elastic_user_8 85.245.107.41 Accepted password for ~ Failed publickey for el~
##  7 elastic_user_8 85.245.107.41 Accepted password for ~ pam_unix(sshd:auth): au~
##  8 elastic_user_8 85.245.107.41 Accepted password for ~ Failed password for ela~
##  9 elastic_user_8 85.245.107.41 Accepted password for ~ Failed password for ela~
## 10 elastic_user_5 85.245.107.41 Accepted password for ~ Failed publickey for el~
```

```
unique(joined_df$IP)
```

```
## [1] "85.245.107.41" "24.151.103.17"
```

We observe that there were valid logins for two IPs that had multiple invalid user logins: 85.245.107.41, and 24.151.103.17.

Multiple IPs using same Invalid Login

The process of finding multiple IPs using the same invalid login should be similar to finding multiple invalid user logins from the same IP address. The variables are just switched around.

```
# Arrange by User
multiple_ip_invalid = invalid_df %>% arrange(User)
head(multiple_ip_invalid, 10)
```

```
##              User            IP
## 1  elastic_user_0          <NA>
## 2  elastic_user_0          <NA>
## 3  elastic_user_0          <NA>
## 4  elastic_user_0 85.245.107.41
## 5  elastic_user_0 85.245.107.41
## 6  elastic_user_0 24.151.103.17
## 7  elastic_user_0 24.151.103.17
## 8  elastic_user_0 24.151.103.17
## 9  elastic_user_0 24.151.103.17
## 10 elastic_user_0 24.151.103.17
##
## 1                                                                                failed adding user 'elas
## 2                                                                                failed adding user 'elas
## 3                                                                                failed adding user 'elas
## 4  pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=85.245.10
## 5                                                          Failed password for elastic_user_0 from 85.24
## 6  pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=24.151.103
## 7                                                          Failed password for elastic_user_0 from 24.15
## 8  pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=24.151.103
## 9                                                          Failed password for elastic_user_0 from 24.15
## 10 pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=24.151.103
```

From the first 20 rows, we see that the 85.245.107.41 and 24.151.103.17 IPs share the same invalid user login. There were many instances of those two IPs having invalid logins for elastic_users[0-9], which is worth nothing.

```
# Find unique User/IP pairs
head(unique(multiple_ip_invalid[, c("User", "IP")]), 20)
```

```
##              User            IP
## 1   elastic_user_0          <NA>
## 4   elastic_user_0 85.245.107.41
## 6   elastic_user_0 24.151.103.17
```

```
## 300 elastic_user_1          <NA>
## 303 elastic_user_1 85.245.107.41
## 305 elastic_user_2          <NA>
## 308 elastic_user_2 85.245.107.41
## 313 elastic_user_3          <NA>
## 316 elastic_user_3 24.151.103.17
## 318 elastic_user_3 85.245.107.41
## 320 elastic_user_4          <NA>
## 323 elastic_user_4 24.151.103.17
## 325 elastic_user_5          <NA>
## 328 elastic_user_5 85.245.107.41
## 329 elastic_user_5 24.151.103.17
## 333 elastic_user_6          <NA>
## 336 elastic_user_6 85.245.107.41
## 338 elastic_user_6 24.151.103.17
## 340 elastic_user_7          <NA>
## 343 elastic_user_7     127.0.0.1
```

```
df2 = unique(multiple_ip_invalid[, c("User", "IP")])
```

And again, extracting the appropriate users with multiple IPs.

```
df2 = df2 %>% filter_at(vars(User,IP), all_vars(!is.na(.)))

# Get the user count for every unique User (462 counts)
user_counts = df2 %>% group_by(User) %>% summarize(num_ips = n_distinct(IP))

# Remove users that do not have multiple IPs
multi_ips_user = user_counts %>% filter(num_ips > 1) %>% select(User)

# Finished data including multiple invalid IPs
multi_ip_invalid = df2 %>% filter(User %in% multi_ips_user$User)
head(multi_ip_invalid, 20)
```

```
##              User             IP
## 1  elastic_user_0  85.245.107.41
## 2  elastic_user_0  24.151.103.17
## 3  elastic_user_3  24.151.103.17
## 4  elastic_user_3  85.245.107.41
## 5  elastic_user_5  85.245.107.41
## 6  elastic_user_5  24.151.103.17
## 7  elastic_user_6  85.245.107.41
## 8  elastic_user_6  24.151.103.17
## 9  elastic_user_7      127.0.0.1
## 10 elastic_user_7  24.151.103.17
## 11 elastic_user_8  85.245.107.41
## 12 elastic_user_8  24.151.103.17
## 13           root 122.176.37.221
## 14           root  90.144.183.19
## 15           root 186.128.152.44
## 16           root   190.178.62.6
## 17           root   68.33.123.70
## 18           root  190.49.42.132
```

```
## 19           root  31.162.29.148
## 20           root   179.39.18.38
```

Related IPs from same Domain

Based on the previous question, we want to find if the related multiple IPs that use the same invalid login are from the same domain. The process I used for this was to first extract the domains, and add it to the data frame:

```
ip_domains = (str_extract(multi_ip_invalid$IP, regex("(\\d+\\.\\d+\\.\\d+)")))
multi_ip_invalid$Domain = ip_domains
head(multi_ip_invalid)
```

```
##             User            IP     Domain
## 1 elastic_user_0 85.245.107.41 85.245.107
## 2 elastic_user_0 24.151.103.17 24.151.103
## 3 elastic_user_3 24.151.103.17 24.151.103
## 4 elastic_user_3 85.245.107.41 85.245.107
## 5 elastic_user_5 85.245.107.41 85.245.107
## 6 elastic_user_5 24.151.103.17 24.151.103
```

Now we have a data frame that contains the user, the multiple IPs associated with each user, and the corresponding domain for the IPs. To see if the IPs are from the same network given the specified user, we want to extract the count of IPs (for each domain) to be greater than 1. This is because we want to know if there exists multiple IPs in the same domain for a given user.

```
domain_counts = multi_ip_invalid %>% group_by(Domain) %>% summarise(num_ips = n_distinct(IP))
# Include domains with multiple unique IPs
multi_ip_domains = domain_counts %>% filter(num_ips > 1) %>% select(Domain)

result = df2 %>% filter(User == User, str_extract(IP, "^(\\d+\\.\\d+\\.\\d+)") %in% multi_ip_domains$Dom
head(result, 20)
```

```
##       User              IP
## 1     root    49.4.143.105
## 2     root      49.4.143.5
## 3     root    49.4.143.181
## 4     root 119.193.140.203
## 5     root 150.183.249.110
## 6  user 54     54.183.6.51
## 7  user 54     54.183.6.61
## 8  user 54     54.183.6.89
## 9  user 54     54.183.6.44
## 10 user 54    54.183.6.110
## 11 user 54     54.183.6.84
## 12 user 54    54.183.6.138
## 13 user 54    54.183.6.217
## 14 user 54    54.183.6.216
## 15 user 54    54.177.15.73
## 16 user 54   54.177.15.157
## 17 user 54   54.177.15.191
## 18 user 54   54.177.15.205
```

```
## 19 user 54   54.177.15.229
## 20 user 54     54.177.16.7
```

From the data above, we see that for each given invalid user, there are corresponding and different IPs that come from the same domain.

What IP had too many failures

It is important to monitor successful and unsuccessful login attempts to see potential invasions on the machine. I extracted all the lines that had the keyword "authentication failure(s)", and did some analysis.

```
# Grabs the messages in the file w/ authentication failures
auth_failures = grep("(authentication failures|maximum authentication attemps)", file, value = TRUE) # A
tail(auth_failures)
```

```
## [1] "Dec 10 08:39:59 LabSZ sshd[24408]: PAM 5 more authentication failures; logname= uid=0 euid=0 tty
## [2] "Dec 10 09:08:59 LabSZ sshd[24419]: PAM 2 more authentication failures; logname= uid=0 euid=0 tty
## [3] "Dec 10 09:10:32 LabSZ sshd[24421]: PAM 4 more authentication failures; logname= uid=0 euid=0 tty
## [4] "Dec 10 09:11:41 LabSZ sshd[24437]: PAM 4 more authentication failures; logname= uid=0 euid=0 tty
## [5] "Dec 10 10:14:13 LabSZ sshd[24833]: Disconnecting: Too many authentication failures for admin [pr
## [6] "Dec 10 10:14:13 LabSZ sshd[24833]: PAM 5 more authentication failures; logname= uid=0 euid=0 tty
```

First, I found 2916 out of 99960 lines that had authentication failures.

```
# Df that extracts IPs
ip_failures = as.data.frame(str_extract(auth_failures, regex("(\\d+\\.\\d+\\.\\d+\\.\\d+)")))
colnames(ip_failures) = "IP Failures"

# Unique IPs that failed
unique_ip_fails = unique(ip_failures)
unique_ip_fails
```

```
##          IP Failures
## 1               <NA>
## 2634   181.25.206.27
## 2638  218.60.136.106
## 2640    106.57.58.19
## 2642  181.23.168.176
## 2643  151.241.67.217
## 2645  181.25.201.155
## 2649   111.40.168.90
## 2651 122.189.198.238
## 2653   60.187.118.40
## 2655   122.191.89.89
## 2657  42.184.142.151
## 2659      82.64.2.59
## 2661  114.32.100.101
## 2663   122.244.28.82
## 2665 123.153.146.183
## 2667   181.26.186.35
## 2677   183.152.79.79
```

```
## 2679    1.30.211.144
## 2681    5.167.75.191
## 2683  93.120.176.237
## 2685    78.106.21.86
## 2687 112.251.168.248
## 2689 119.193.140.176
## 2691    58.19.144.50
## 2693 122.112.235.133
## 2695 122.189.193.214
## 2697    49.4.143.105
## 2737    61.166.73.66
## 2739  122.191.88.115
## 2741   186.130.83.53
## 2743    123.96.5.168
## 2745   111.40.30.206
## 2747  188.18.252.218
## 2749 125.107.136.165
## 2751  190.107.182.33
## 2753 112.101.164.200
## 2755  223.244.185.76
## 2757    182.243.87.6
## 2759  91.243.236.123
## 2761      49.4.143.5
## 2763    222.89.76.13
## 2765   191.81.42.216
## 2767  183.93.215.158
## 2769  122.190.143.18
## 2771  61.183.117.250
## 2773  123.120.200.51
## 2775    73.231.4.205
## 2777   110.78.174.75
## 2779    49.4.143.181
## 2781 186.128.141.232
## 2783   46.89.129.145
## 2785    59.174.52.12
## 2787  103.230.120.26
## 2789 122.189.197.241
## 2791 123.119.111.172
## 2793  123.164.142.82
## 2795 175.162.187.121
## 2797   60.165.208.28
## 2799   218.91.34.237
## 2801     49.84.87.84
## 2803   222.187.86.51
## 2804  77.231.252.103
## 2808   95.190.198.34
## 2810   94.154.25.149
## 2812  183.146.159.20
## 2814   126.59.251.31
## 2815  222.186.56.220
## 2819  183.93.253.159
## 2821  219.82.145.223
## 2823  122.163.61.218
## 2833   14.54.210.101
```

```
## 2837  105.101.221.33
## 2839    37.110.24.51
## 2840 116.255.253.137
## 2842  181.25.189.115
## 2844 178.219.248.139
## 2846  177.38.145.209
## 2848   191.83.152.32
## 2850  170.79.155.119
## 2852    46.30.160.83
## 2854 201.178.245.106
## 2856 181.211.173.182
## 2858  122.144.136.83
## 2860   1.189.205.173
## 2862   181.23.26.185
## 2864  201.27.216.125
## 2866 179.208.151.103
## 2868 119.193.140.203
## 2870   182.243.85.75
## 2872   61.174.116.31
## 2874  111.40.166.130
## 2876 186.129.147.223
## 2878   37.78.105.176
## 2880   58.100.135.31
## 2881  221.194.44.190
## 2885   178.161.33.80
## 2887   179.38.76.250
## 2889   186.47.222.98
## 2891  201.178.81.113
## 2903    68.182.39.76
## 2905    122.5.240.60
## 2907      5.36.59.76
## 2908    5.188.10.180
## 2911     106.5.5.195
## 2912  185.190.58.151
## 2916    119.4.203.64
```

```
# Subtract 1 to exclude NA value (106 ip that failed)
nrow(unique_ip_fails) - 1
```

```
## [1] 106
```

```
# IP with greatest failures
which.max(table(ip_failures))
```

```
## 49.4.143.105
##           83
```

In total, there were 106 IPs in the entire file that had too many authentication failures, and IP-49.4.143.105 had the most failures with 83.

SUDO Problems

User/Machine

```
sudo = grep("sudo:\\s", file, value = TRUE, ignore.case = TRUE) # 557 TRUE
```

There are 557 lines in the MergedAuth.log file that run on the Sudo app. I verified this value by also checking on the file in a text editor (Cmd F for finding, it tells you how many matches there are).

Creating a data frame w/ columns for Sudo to Extract Info

```
sudo_rx = gregexpr(
  "^(?P<Date_Time>[[:alpha:]]+ [0-9\\s]?[0-9] \\d{2}:\\d{2}:\\d{2}|# .*.log$)\\s?(?P<Host>[a-zA-Z0-9-]*
  sudo,
  perl = TRUE
)

sudo_df = as.data.frame(t(mapply(function(str, match) {
  s = attr(match, "capture.start")
  substring(str, s, s + attr(match, "capture.length") - 1) # add -1 to not count last char
}, sudo, sudo_rx)))
```

I used the same process of extracting capture groups from the expression and subsetting the data frame. After this, I needed to extract the unique values in the machine and user columns.

```
unique(sudo_df$V2) # unique machine(s) for Sudo
```

```
## [1] "ip-10-77-20-248"
```

```
sudo_user = as.data.frame(str_extract(sudo, regex("user\\s?=?[[:alpha:]]+", ignore_case = TRUE)))
unique(sudo_user)
```

```
##   str_extract(sudo, regex("user\\\\s?=?[[:alpha:]]+", ignore_case = TRUE))
## 1                                                                USER=root
## 2                                                                user root
```

For the apps run by Sudo, the machine and user, respectively, are:

ip-10-77-20-248 and Root

Executables/Programs

I found the programs ran by sudo by looking for the "COMMAND = exec" sub patterns.

```
sudo_programs = as.data.frame(str_extract(sudo, regex("COMMAND=[a-z/]+", ignore_case = TRUE)))
colnames(sudo_programs) = "Program"

# Finding the programs
unique(sudo_programs)
```

```
##                                     Program
## 1                       COMMAND=/usr/bin/curl
## 2                                        <NA>
## 4                        COMMAND=/usr/bin/apt
## 10                       COMMAND=/usr/bin/tee
## 19                    COMMAND=/usr/sbin/update
## 34                       COMMAND=/usr/bin/vim
## 55                     COMMAND=/bin/hostname
## 58                COMMAND=/usr/bin/hostnamectl
## 64                   COMMAND=/usr/sbin/service
## 67                       COMMAND=/usr/bin/dpkg
## 103                             COMMAND=/bin/su
## 130 COMMAND=/usr/share/filebeat/bin/filebeat
## 136                             COMMAND=/bin/cp
## 148                  COMMAND=/usr/bin/filebeat
## 151                             COMMAND=/bin/rm
## 199                          COMMAND=/bin/chmod
## 253                          COMMAND=/bin/mkdir
## 322                 COMMAND=/usr/sbin/groupadd
## 331                   COMMAND=/sbin/resolvconf
## 334                   COMMAND=/usr/bin/hexdump
## 412                          COMMAND=/bin/chown
## 418                        COMMAND=/usr/bin/vi
## 430                     COMMAND=/sbin/auditctl
## 436                     COMMAND=/sbin/ausearch
## 442                      COMMAND=/usr/bin/tail
## 514                             COMMAND=/bin/ls
```

I found 25 unique executables ran via sudo (excluding the NA).