Introduction

Bro is an open-source network security monitor which inspects network traffic looking for suspicious activity. The Bro framework provides an extensible scripting language that allows an analysis of application to protocol level traffic. All built-in and user added bro scripts output data to log files which can be further analyzed to detect malicious activities.

Logstash is an open-source log management tool which collects and normalizes log data, such as the logs output by Bro. The Logstash tool is combined with Elastic Search for storage and Kibana is used as a web interface to search and visualize the collected log data. Logstash filtering allows complex log data to be normalized and enhanced.

Both Logstash and Bro allow users to customize the inspection of consumed data. Commercial Threat Intelligence providers can be integrated automatically or with a bit of scripting to provide additional context around the vast amounts of data collected. This paper will describe how these technologies can be integrated to identify current and emerging threats in real-time, allowing infected systems to be identified quickly.

Background

Cyber-attacks are continually increasing in scope and complexity. Advanced persistent threats are becoming more difficult to detect. This is leading to what the 2015 Verizon Data Breach Report calls a detection deficit. While sixty percent of attackers are able to gain access within minutes, Mandiant has found that the average detection time of attacks is 205 days. The core of this detection deficit is the fact that the cost, complexity, and volume of data needing to be analyzed increases with the maturity of the security organization.

Most organizations are collecting log data from systems, applications, and network devices to generate operational statistics and/or alerts to abnormal behavior. Valuable security data is typically hidden from view in these massive log data files. Software engineers write the code that determines what gets logged within their applications. Unfortunately, a lot of valuable data is not written to logs, making it improbable for log management systems administrators to detect attacks quickly. The best method to detect attacks is to analyze the session, packet string, and full packet capture data within the environment. The sheer volume of packet capture data that traverses a typical enterprise network makes it infeasible to store for forensic purposes.

Mechanisms such as intrusion detection systems can analyze packet level data in real-time, but a major disadvantage with this approach is that they need to know which threats to look for at the time of analysis. To make packet level inspection detect a cyberattack in real-time as well as provide historical analysis, a network security monitoring application should be used. One such option is the Bro Network Security Monitor. Bro can inspect network traffic in real-time or look into a packet capture file that was previously recorded. As part of the analysis, Bro looks for known attacks in the same way a typical intrusion detection system would. The benefit of Bro is that all connections, sessions, and application level data are written to an extensive set of log files for later review.

To collect and review the log files output by Bro, consumers can choose from a wide array of commercial and open-source log management tools. This paper will take a deep look into the Logstash product, an open-source log management tool. Logstash has over forty inputs to collect data, 40 filtering options to perform on the collected data, and over fifty output destinations to store the log data. Logstash can input the Bro logs, apply filters to highlight and enhance critical data, and output the data into an Elastic Search data store. Once the data is in Elastic Search, the Kibana web interface allows users to visualize the data.

Solutions

Bro Network Security Monitor

Bro is an open-source network security monitor that has been in development since 1995. The power of Bro is in the extensible scripting engine that analyzes the packet data. There are a wide array of out-of-th- box, pre-written scripts that ship with Bro that analyze network traffic. These local scripts write to six different categories of logs; network protocols, files, detection, network observations, miscellaneous, and diagnostics. A full list and description of log files can be found in Appendix A.

By default, all Bro logs are written to `<BroInstallDir>/logs/current` and are rotated on a daily basis. If the Bro utility is launched manually, to analyze a packet capture for example, then log files are written to the current working directory. This is important to note when specifying which log files to collect when running Logstash later.

Elastic Search/Logstash/Kibana

Elastic Search, Logstash, and Kibana (ELK Stack) are all open-source products that allow the collection, normalization, storage, and visualization of log data. When combined with Bro, these tools can collect and install the ELK stack on the same system to collect local files, or ELK can be run on a separate server and the logs can be forwarded via syslog. Logstash has over forty input methods available to ingest logs, including local file, syslog, stdin, and the Logstash Forwarder (Lumberjack).

For simplicity, everything can be installed on the same server and the local log files can be collected there. For troubleshooting normalization and other Logstash configurations, it is useful to have two working versions of the Logstash configuration file. The first file will accept stdin input and use stdout as an output to display the results. The second configuration file will accept a file input and output to Elastic Search for long term storage. Both configuration files will share the same filter code to normalize and enhance the log data as necessary.

```
input {
  stdin { }
}
filter {…}
output {
  stdout {
    codec => rudydebug
  }
}
```

Figure 1: Logstash Temporary Configuration

```
input {
  file {
    path =>
"/opt/bro/logs/current/*.log"
  }
}
filter {…}
output {
  elasticsearch {
    host => localhost
    cluster => "elasticsearch "
  }
}
```

Figure 2: Logstash Permanent Configuration

Figure 1 and Figure 2 show sample configurations for the temporary and permanent configuration files for Logstash. Next, the filters will need to be added to normalize the data in order to extract the metadata that matters. The filter plugin to do this is the grok plugin. The way this plugin is used is to match a message to a set of Onigimura regular expression strings. Logstash ships with a set of expressions that can be used, however a set of custom expressions will need to be created for the Bro logs.

```
filter {
 grok {
   match => {
     "message" => "<RegEx Goes Here>"
   }
 }
}
```

Figure 3: Basic Grok Matching

To put metadata into a specific field, the correct syntax will be `(?<column>regex)`. Each message match line can contain as many patterns as necessary to get a match. Alternatively, to leverage the built-in patterns, the syntax would look like `%{PATTERNNAME:columnname}`. While this is a quick and easy way to normalize a handful of messages, this approach involves a lot of management overhead and does not scale, especially when adding in custom regular expressions to match the Bro messages.

To solve the management scalability problem, all of the regular expressions will be stored in a custom patterns directory and referenced in the grok plugin code. A method that I have found to be workable is to create a rule file for each device that is being normalized by the Logstash instance. For example, the custom patterns directory will contain linux.rule, bro.rule, apache.rule, etc. Doing this simplifies the configuration file, moves the complex regular expression to another location, and will allow us to optimize Logstash normalization.

```
filter {
 grok {
   match => {
     patterns_dir => "/path/to/patterns"
     "message" => "%{291001}"
     "message" => "%{291002}"
     "message" => "%{291003}"
     "message" => "%{291004}"
   }
 }
}
```

Figure 4: Logstash Custom Patterns

Figure 4 shows that the configuration will look like when utilizing the custom patterns feature. The regular expression used in Figure 3 would be moved to the rule files in the /path/to/patterns directory referenced in Figure 4.

```
291001 <RegEx Goes Here>
291002 <RegEx Goes Here>
291003 <RegEx Goes Here>
291004 <RegEx Goes Here>
```

Figure 5: Sample Logstash Rule File

This simple change allows Logstash to normalize the Bro logs and pull out the valuable metadata such as IP addresses, file names, ports, etc. In order to improve performance Logstash normalization and enhance the data with action, status, object, and device type data users can split the grok patterns into their own code blocks for each normalization rule. To do this, IF/ELSE statements will be used against the message in order to match it against the known Bro log patterns. A quick way to get the match code for the IF statements is to use the regular expression code from the rule files and remove the column data from the code. For example, if the code match was `(?<column>regex)(?<column2>regex2)`, the code match would be `(regex)(regex2)`. It is possible to have the IF statements look at the name of the log file being used to collect data. However, if the input is from something such as syslog, this will not work properly. To allow the configuration to be plugged into as many environments as possible, it’s easier to choose to go with the regular expression matching of the message.

```
filter {
  if [message] =~ /^((regex)(regex2))/ {
    grok {
      match => {
        patterns_dir => "/path/to/patterns"
        "message" => "%{291001}"
      }
    }
  }
  else if [message] =~ /^((regex3)(regex4))/ {
    grok {
      match => {
        patterns_dir => "/path/to/patterns"
        "message" => "%{291002}"
      }
    }
  }
}
```

Figure 6: Logstash IF Statements

Figure 6 shows an example of how to utilize the IF statements in Logstash. It is important to remember to use the ELSE IF for additional message matching so the message is not matched multiple times by accident. While this may look like unnecessary processing, it is required to use the add_field function in the grok plugin. The add_field allows each log message to have additional key value pairs added to the message stored in the Elastic Search database. For each message, users should assign a device type, object, action, status, and rule ID. The device type, object, action, and status are part of the Common Event Expression tags to help identify similar events across multiple devices. These will help when comparing Bro log data with firewall logs, system logs, or other IDS logs which may not use the same naming conventions. The rule ID is what will help performance tune Logstash going forward. Each normalized message will now be tagged with the rule ID which was used for normalization. Since Logstash is using a top down approach with the IF statements, we can report on the most commonly used rule ID's found in the environment and place those near the top of the filter plugin for normalization. This will bypass the processor intensive regular expression matching for messages which are rarely seen.

While Bro can leverage the LibGeoIP library for geolocating IP addresses, I recommend moving this functionality to Logstash. This allows Logstash to geolocate IP addresses from devices other than Bro as well. The geoip plugin will be placed after all the IF statements in the filter plugin. The requirements for geoip are a source column, a destination column, location of the GeoIP database, and fields to be added from the GeoIP database. Logstash ships with a built-in GeoLiteCity database, but it may be useful to provide a separate one that can be updated on demand if needed. For the built-in GeoLiteCity database, the following are available: city_name, continent_code, country_code2, country_code3, country_name, dma_code, ip, latitude, longitude, postal_code, region_name and timezone. For Kibana to be able to plot coordinates on the map, only the longitude and latitude are required, the others are optional for additional contextual information.

```
filter {
 geoip {
   source => "evt_dstip"
   target => "geoip"
   database => "/path/to/GeoLiteCity.dat"
   add_field => [ "[geoip][coordinates]",
"%{[geoip][longitude]}" ]
   add_field => [ "[geoip][coordinates]",
"%{[geoip][latitude]}"  ]
   add_field => [ "[geoip][coordinates]",
"%{[geoip][city\_name]}"  ]
   add_field => [ "[geoip][coordinates]",
"%{[geoip][continent\_code]}"  ]
   add_field => [ "[geoip][coordinates]",
"%{[geoip][country\_code2]}"  ]
   add_field => [ "[geoip][coordinates]",
"%{[geoip][country\_code3]}"  ]
   add_field => [ "[geoip][coordinates]",
"%{[geoip][country\_name]}"  ]
   add_field => [ "[geoip][coordinates]",
"%{[geoip][dma\_code]}"  ]
   add_field => [ "[geoip][coordinates]",
"%{[geoip][postal\_code]}"  ]
   add_field => [ "[geoip][coordinates]",
"%{[geoip][region\_name]}"  ]
 }
}
```

Figure 7: Logstash GeoIP Configuration

The Elastic Search template that stores the Logstash data has a built-in mapping for the geoip column which was used as the target in Figure 7. This takes the geoip field and sets it as a geo_point object type. When geolocating multiple IP fields, multiple targets will need to be used. As such, the template will need to be modified to provide mappings for the new geoip fields. To get the current Elastic Search template for logstash, enter the "`curl -XGET localhost:9200/_template/logstash`" command. Appendix D shows a sample of the default template that is shipped with Elastic Search. The geoip mapping will need to be modified or copied to add additional geoip fields. Appendix D also shows a sample template that can be used with two geoip mappings. To update the template, enter the "`curl -XPUT localhost:9200/_template/logstash -d '<template>'`" command , where <template > is the text of the template. Once this is done, multiple geoip plugins can be used with Logstash, changing the target from "geoip" to "geoip_dst" and "geoip_src".

The final optional piece of configuration for Logstash will be modifying the timestamp of the collected logs. When collecting Bro logs in real-time, this will not be an issue. However, if the user wants to analyze packet capture files, Bro will use the timestamp from the packet capture file as the timestamp in the logs. When Logstash collects the log file, it will use the collect time as the timestamp for the log messages. If any forensics analysis is going to be done, the timestamp should be preserved for posterity. This date plugin will need to go after the IF statements in order to use the appropriate time column from the log message.

```
filter {
  date {
    match => ["start_time", "UNIX"]
  }
}
```

Figure 8: Logstash Time Preservation

Figure 8 demonstrates taking the start_time column and matching it to the UNIX epoch time function and uses that for the timestamp of the log.

Threat Intelligence Integrations

Threat intelligence feeds are shared indicators of compromise, generally shared across industry verticals such as finance, healthcare, industrial, retail, etc. Combining threat intelligence with log data adds a form of predictive modeling to defensive tools by collecting known attackers, attack vectors, and attack tools and sharing this information with peers. When company A detects an attack on their network, that information can be shared with company B immediately. Company B can then update their signatures to detect the attack before it occurs. Both Bro and Logstash allow for integrations with threat intelligence providers.

One such provider for Bro is Critical Stack Intel. The Critical Stack agent is installed on the Bro system and is configured to pull feeds from the server. Critical Stack maintains a list of more than 98 threat feeds, including malicious IP addresses, known phishing email addresses, malicious file hashes, and domains known to host malware. These feeds contain over 800,000 indicators of compromise. A free account needs to be created on the Critical Stack website (https://intel.criticalstack.com/) to obtain an API key for the agent to use to pull data. On the website, lists of feeds are displayed, requiring only a click to add them to the agent's list of feeds to pull. On the agent system, the feeds are pulled and converted into Bro scripts. To integrate these scripts into Bro, just reference the target directory at the end of the Bro command. For example: `./bro -r file.pcap local /path/to/criticalstack/feeds`. When malicious activity is detected by the Critical Stack scripts, logs will be written to the intel.log file in the Bro log directory. From within Kibana, reports and dashboards can be created using data found in this intel log file for further analysis.

Logstash does not have a direct integration with threat intelligence providers like Critical Stack. However, the filter plugin has a translation plugin that allows users to perform similar lookups. Just as Critical Stack has 98 threat feeds, there are many other publicly available alternatives which we can utilize for Logstash translations. A simply python script can pull the data and transform the information into a usable format for Logstash.

The transform plugin takes a normalized field as a source, a destination field to populate, and a dictionary path to perform the lookup. The dictionary file is a YAML formatted file that contains two columns. The first column is the value that is compared to the source field from the translation. If there is a match, the second column in the YAML file is placed into the destination column from the translation. If there is no match, the column will not be created or populated for that log file.

```
filter {
  translate {
    field => "evt_dstip"
    destination => "tor_exit_ip"
    dictionary_path => "/path/to/yaml"
  }
}
```

Figure 9: Logstation Threat Intel Translation

The example in Figure 9 shows a lookup of the evt_dstip column. When a match is found it will populate the tor_exit_ip column with the corresponding data. For reporting purposes, I have been using "IP_Address": "YES" as the format for the YAML file. This allows reports and dashboards containing the translated fields with a value of YES to be displayed.

Conclusion

Even without trying to add packet capture level data for analysis organizations are bombarded with data from system logs. By leveraging network security monitoring tools such as Bro, the packet data can be analyzed and stored in real-time, or saved in packet captures for future analysis. The ELK stack provides a wide array of functionality that can ingest the Bro log data into normalized fields for more efficient analysis. All of the data collected by both Bro and Logstash can then be enhanced with Threat Intelligence provider feeds to predict and detect attacks more quickly, lowering the detection deficit and allowing organizations to detect cyberattacks before valuable data is exfiltrated. Using these tools, organizations can observe, orient, decide, and act quickly to the advanced threats facing them today.

Appendix A: Bro Log Files

Network Protocols

Log File	Description
conn.log	TCP/UDP/ICMP connections
dhcp.log	DHCP leases
dnp3.log	DNP3 requests and replies
dns.log	DNS activity
ftp.log	FTP activity
http.log	HTTP requests and replies
irc.log	IRC commands and responses
kerberos.log	Kerberos
modbus.log	Modbus commands and responses
modbus_register_change.log	Tracks changes to Modbus holding registers
mysql.log	MySQL
radius.log	RADIUS authentication attempts
rdp.log	RDP
sip.log	SIP
smtp.log	SMTP transactions
snmp.log	SNMP messages
socks.log	SOCKS proxy requests
ssh.log	SSH connections
ssl.log	SSL/TLS handshake info
syslog.log	Syslog messages
tunnel.log	Tunneling protocol events

Files

Log File	Description
files.log	File analysis results
pe.log	Portable Executable (PE)
x509.log	X.509 certificate info

Detection

Log File	Description
intel.log	Intelligence data matches
notice.log	Bro notices
notice_alarm.log	The alarm stream
signatures.log	Signature matches
traceroute.log	Traceroute detection

Network Observations

Log File	Description
app_stats.log	Web app usage statistics
known_certs.log	SSL certificates
known_devices.log	MAC addresses of devices on the network
known_hosts.log	Hosts that have completed TCP handshakes
known_modbus.log	Modbus masters and slaves
known_services.log	Services running on hosts
software.log	Software being used on the network

Miscellaneous

Log File	Description
barnyard2.log	Alerts received from Barnyard2
dpd.log	Dynamic protocol detection failures
unified2.log	Interprets Snort's unified output
weird.log	Unexpected network-level activity

Bro Diagnostics

Log File	Description
capture_loss.log	Packet loss rate
cluster.log	Bro cluster messages
communication.log	Communication events between Bro or Broccoli instances
loaded_scripts.log	Shows all scripts loaded by Bro
packet_filter.log	List packet filters that were applied
prof.log	Profiling statistics (to create this log, load policy/misc/profiling.bro)
reporter.log	Internal error/warning/info messages
stats.log	Memory/event/packet/lag statistics
stderr.log	Captures standard error when Bro is started from BroControl
stdout.log	Captures standard output when Bro is started from BroControl

Appendix B: Useful Links

Notes on creating new templates for Elastic Search

http://spuder.github.io/elasticsearch,/logstash/elasticsearch-default-shards/

Access to GeoLite City Database

http://dev.maxmind.com/geoip/legacy/geolite/

PCAP Files for Analysis

https://github.com/LiamRandall/BroMalware-Exercise/blob/master/README.md
http://www.netresec.com/?page=PcapFiles
https://www.mediafire.com/?a49l965nlayad

Product Documentation

https://www.bro.org/documentation/index.html
https://www.bro.org/sphinx/script-reference/log-files.html
https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html
https://www.elastic.co/guide/en/logstash/current/index.html
https://www.elastic.co/guide/en/kibana/current/index.html

Appendix C: Useful Commands

Elastic Search:

List All Templates

```
curl -XGET localhost:9200/_template/logstash
```

Insert/Update Template

```
curl -XPUT localhost:9200/_template/logstash -d '_____'
```

List All Indexes

```
curl 'localhost:9200/_cat/indices?v'
```

Delete Indexes

```
curl -XDELETE 'localhost:9200/logstash*?pretty'
```

Logstash:

Verify Config File Before Normalizing Logs

```
./logstash -f logstash.conf –configtest
```

Bro:

Read Packet Capture File

```
./bro –r file.pacp
```

Appendix D: Elastic Search Templates

Default Template

```
{
 "logstash" : {
  "order" : 0,
  "template" : "logstash-*",
  "settings" : {
   "index.refresh_interval" : "5s"
  },
  "mappings" : {
   "_default_" : {
    "dynamic_templates" : [ {
     "message_field" : {
      "mapping" : {
       "index" : "analyzed",
       "omit_norms" : true,
       "type" : "string"
      },
      "match" : "message",
      "match_mapping_type" : "string"
     }
    }, {
     "string_fields" : {
      "mapping" : {
       "index" : "analyzed",
       "omit_norms" : true,
       "type" : "string",
       "fields" : {
        "raw" : {
         "index" : "not_analyzed",
         "ignore_above" : 256,
         "type" : "string"
        }
       }
      },
      "match" : "*",
      "match_mapping_type" : "string"
     }
    } ],
    "properties" : {
     "geoip" : {
      "dynamic" : true,
      "properties" : {
       "location" : {
        "type" : "geo_point"
       }
      },
      "type" : "object"
     },
     "@version" : {
      "index" : "not_analyzed",
      "type" : "string"
     }
    },
    "_all" : {
     "enabled" : true,
     "omit_norms" : true
    }
   }
  },
  "aliases" : { }
 }
}
```

Updated Template

```
{
 "logstash" : {
  "order" : 0,
  "template" : "logstash-*",
  "settings" : {
   "index.refresh_interval" : "5s"
  },
  "mappings" : {
   "_default_" : {
    "dynamic_templates" : [ {
     "message_field" : {
      "mapping" : {
       "index" : "analyzed",
       "omit_norms" : true,
       "type" : "string"
      },
      "match" : "message",
      "match_mapping_type" : "string"
     }
    }, {
     "string_fields" : {
      "mapping" : {
       "index" : "analyzed",
       "omit_norms" : true,
       "type" : "string",
       "fields" : {
        "raw" : {
              "index" : "not_analyzed",
              "ignore_above" : 256,
              "type" : "string"
            }
          }
        },
        "match" : "*",
        "match_mapping_type" : "string"
      }
    } ],
    "properties" : {
      "geoip_dst" : {
        "dynamic" : true,
        "properties" : {
          "location" : {
            "type" : "geo_point"
          }
        },
        "type" : "object"
      },
      "geoip_src" : {
        "dynamic" : true,
        "properties" : {
          "location" : {
            "type" : "geo_point"
          }
        },
        "type" : "object"
      },
      "@version" : {
        "index" : "not_analyzed",
        "type" : "string"
      }
    },
    "_all" : {
      "enabled" : true,
      "omit_norms" : true
    }
  }
 },
 "aliases" : { }
 }
}
```