CCDCOE

NATO Cooperative Cyber Defence
Centre of Excellence Tallinn, Estonia

A Comparative Analysis of Open-Source Log Management Solutions for Security Monitoring and Network Forensics

Risto Vaarandi
Paweł Niziński

Tallinn 2013

Disclaimer

This publication is a product of the NATO Cooperative Cyber Defence Centre of Excellence (the Centre) and it represents the views and interpretations of the Centre. This publication does not represent the opinions or policies of NATO and is designed to provide an independent position.

Third-party sources are quoted as appropriate and the Centre is not responsible for the content of the external sources referenced in this publication. The Centre assumes no responsibility for any loss or harm arising from the use of information contained in this publication. Copies of this publication may be distributed for non-profit and non-commercial purpose only.

Contact

NATO Cooperative Cyber Defence Centre of Excellence
Filtri tee 12, Tallinn 10132, Estonia
publications@ccdcoe.org
www.ccdcoe.org

Contents

1. Introduction

Centralised event log management plays a crucial role in security monitoring and network forensics, since it allows for gathering events from thousands of nodes (servers, network devices, IPS sensors, etc.) to a few dedicated servers where central analysis is carried out. The analysis can be a real-time process, where security incidents are detected from incoming events through event correlation and other advanced monitoring techniques; it can also be an off-line forensics activity, where past events are investigated in order to study security incidents that have already occurred. Without event collection to central location(s), monitoring and forensics activities would have to be carried out at individual network nodes, which is time-consuming and prevents the timely resolution of security incidents. For example, if a security incident involves hundreds of network nodes, the event logs for all nodes would have to be analysed separately. Furthermore, the attacker may erase events from the local event log in order to remove any traces of his/her malicious activities. If all events are logged, both locally and to a central log collection server, the latter will always have a copy of the original event log, which makes forensics easier even if the original log is lost. For the above reasons, a number of commercial and open-source solutions have been created for event collection and centralised analysis. Some of these solutions are designed for log management only, while others are full-fledged SIEM (Security Information and Event Management) frameworks. Figure 1 provides an overview of the essential components of an event log management framework.

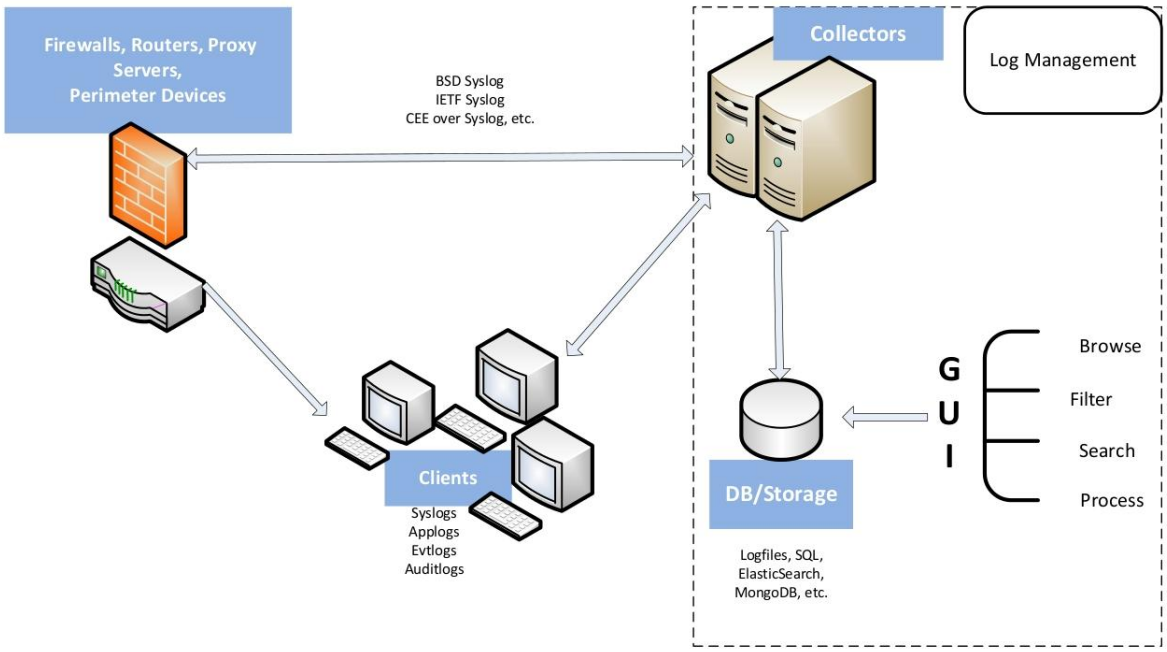

Figure 1. An architectural overview of an event log management framework.

As depicted in Figure 1, nodes of the IT system are using protocols like IETF syslog for sending events to the collector service(s) at the central log server(s). Collector services use various techniques for filtering and normalising events and store preprocessed events into some means of storage (e.g. a database or a flat file). Human analysts can access stored data through a GUI for carrying out searches, creating reports, and other analytical tasks.

While commercial log management solutions are regularly reviewed and compared by independent organisations (e.g. Gartner Magic Quadrant reports for SIEM solutions), such comparisons are often

hard to find for open-source tools, especially for recently created solutions. However, many institutions are using open-source tools for monitoring and forensics, since they allow for implementation of incident detection and analysis frameworks in a cost-efficient way. Furthermore, recently appeared open-source solutions have started a new architectural trend, where the log management system consists of independent and replaceable modules which interact through well-defined interfaces and protocols. In contrast, most commercial systems are essentially monolithic solutions where individual components (such as the GUI or event collector service) cannot be changed for a component from another vendor.

In this paper, we will focus on open-source solutions for log management and discuss recent developments in this field, covering novel technologies and solutions which have appeared during the past 2-3 years. First, we will provide an overview of commonly used log collection protocols and log storing techniques. We will then move on to an in-depth description of advanced event collection servers and graphical log management systems. The first contribution of this paper is an analytical comparison of presented tools; the second contribution is a detailed comparative performance evaluation of the tools. For this purpose, we conducted a set of experiments for assessing their resource consumption and event processing speed under a heavy load. To the best of our knowledge, such performance evaluations have not been conducted recently for state-of-the-art open-source log management solutions.

The remainder of this paper is organised as follows: section 2 provides an overview of event collection protocols, section 3 discusses commonly used log storing techniques, section 4 provides a discussion and performance evaluation for leading open-source syslog servers, section 5 presents an overview and performance assessment of graphical log management systems, and section 6 concludes the paper.

2. Overview of event collection protocols

2.1. BSD syslog protocol

Until the 1980s, event logging was mostly accomplished by writing events to a file residing in a local file system, or to some other means of local event storage (e.g. a memory-based ring buffer). In the 1980s, Eric Allman proposed the BSD syslog protocol for event log collection, which was initially used in the *sendmail* mailing system, but then gradually became widely accepted by most vendors (see [RFC3164]). According to the BSD syslog protocol, log messages are sent over the network as UDP datagrams, with each message encapsulated into a separate UDP packet. Each message has a certain facility and severity, where facility indicates the type of sender. The BSD protocol defines 24 facility and 8 severity values which range from 0 to 23 and 0 to 7 respectively. For example, the facility value 0 means an operating system kernel, 2 a mailing system, and 3 a system daemon, while the severity value 7 means a debug message, 4 a warning message, and 0 an emergency message. For reasons of convenience, textual acronyms are often used instead of numerals, e.g. *mail* denotes facility 2 and *warning* denotes severity 4. According to BSD syslog, the payload of the UDP packet carrying the message must have the format *<Priority>Timestamp Hostname MSG*, where *Priority* is defined as a string representation of a numeral (*8 * facility_value) + severity_value*. For example, the following message represents a warning: "ids[1299]: port scan from 192.168.1.102" for *daemon* facility which was issued on November 17 at 12:33:59 by the network node myhost2:

<28>Nov 17 12:33:59 myhost2 ids[1299]: port scan from 192.168.1.102

By convention, the alphanumerals that start the *MSG* field are regarded as the *Tag* subfield which represents the name of the sending program ("ids" in the above example), while the remainder of the *MSG* field is regarded as the *Content* subfield ("[1299]: port scan from 192.168.1.102" in the above example). The *Tag* subfield is often followed by a number in square brackets which represents the process ID of the sending program (1299 in the above example). Note that some syslog servers regard the process ID with square brackets and the colon as the part of the *Tag* field.

On one hand, since BSD syslog is UDP based, it is a lightweight and efficient protocol and consumes very little network bandwidth and system resources. On the other hand, the protocol has a number of drawbacks which are summarised below:
1) no support for reliable message transmission over TCP;
2) no support for encryption and authentication;
3) timestamps are not specific enough, lacking the year number, timezone information, and fractions of a second;
4) apart from the name of the sending program and its process ID, the *MSG* field has no structure.

In order to address the first issue, a TCP flavor of BSD syslog protocol was proposed during the previous decade, where a stream of messages in BSD syslog format is sent over a TCP connection, with a newline (ASCII 10) character acting as a separator between messages. This protocol flavour also allowed for using other utilities (e.g. *stunnel*) for setting up secure tunnels for logging, and thus for addressing the second issue.

2.2. IETF syslog protocol

In 2009, IETF syslog protocol was proposed that addresses the drawbacks of BSD syslog (see [RFC5424-5426]). The IETF syslog supports secure message transmission over TLS, but also unencrypted transmission over UDP. In addition, it uses a new message format with more detailed RFC3339 timestamps (see [RFC3339]) and structured data blocks. The following example depicts the previous sample message in a new format:

<28>1 2012-11-17T12:33:59.223+02:00 myhost2 ids 1299 - [timeQuality tzKnown="1" isSynced="1"][origin ip="10.1.1.2"] port scan from 192.168.1.102

The priority specification *<28>* is followed immediately by the protocol version number (currently set to 1). In addition, the sender is passing two structured data blocks *[timeQuality tzKnown="1" isSynced="1"]* and *[origin ip="10.1.1.2"]* to the sender, with the first one indicating that the sender's clock is synchronised to an external reliable time source, and the second one indicating the sender's IP address.

2.3. CEE syslog format

Another currently ongoing effort to introduce structure to log messages is the Common Event Expression (CEE) initiative. CEE has proposed JSON and XML formats for events, while also suggesting the use of BSD and IETF syslog protocols for transporting JSON-formatted events (see [CEE]). The following depicts an example event in JSON format which has been encapsulated into a BSD syslog message:

<28>Nov 17 12:33:59 myhost2 ids[1299]: @cee:{"pname":"ids","pid":1299,"msg":"port scan from 192.168.1.102","action":"portscan","dst":"192.168.1.102"}

Passing such a structure of keyword-value pairs from sender to receiver will ease the message parsing on the receiver side. Also, JSON format is compatible with BSD and IETF syslog protocols, since this format is used inside the *MSG* field, which has no predefined structure.

2.4. Other logging protocols

Apart from the aforementioned event logging protocols, other protocols could be employed for event collection. In the following, we will discuss briefly some alternatives to BSD and IETF syslog. The SNMP protocol is a well-established UDP-based network monitoring and management protocol which was introduced in the 1980s. Although the main focus of SNMP lies in fault and performance management of large networks, security-related information is sometimes transmitted with SNMP messages (trap or notification messages). In the open-source domain, there are tools for receiving SNMP notifications and storing them to files, e.g. see Net-SNMP and SNMPTT projects [Net-SNMP, SNMPTT]. Also there are some application-specific protocols for event transmission – for example, the rsyslog RELP protocol for reliable message transmission, GELF (Graylog Extended Log Format) protocol for structured logging, Sourcefire Estreamer protocol for transmitting IDS alarms with packet payloads, etc.

3. Overview of log storing techniques

3.1. Storing logs into files

When logs are collected to central server(s), they need to be written to a permanent storage, so that off-line analysis of past events can be conducted at a later time. The most common way of storing logs is to write incoming log messages into flat files that reside on the disk of the central server. With such an approach, separate log files are usually created by logging host, application, message type, or by some other criteria, and log files are often arranged into some directory structure, so that a relevant log file can be quickly located. While storing messages to flat files consumes little CPU time and allows for the log server to receive large volumes of events per second (see also the experiment results in section 4), searching relevant events from flat files can be time consuming and resource intensive. In today's computer networks, many gigabytes of log data can be generated each day, and even if these data are split between many files, individual files can still be quite large. As a result, it is beyond human capabilities to review those logs manually, and interesting events can usually be identified only with regular expression searches. For example, often system administrators are using command line tools like *egrep* or *pcregrep* for finding events with regular expressions. Unfortunately, using a regular expression language for each individual search is CPU intensive and takes time.

3.2. Storing logs into SQL databases

For the reasons stated above, inserting log messages into SQL databases has been another common approach for storing log data. However, inserting a message to a database is considerably more expensive than writing it to a flat file (this is also highlighted by experiment results in sections 4 and 5). Firstly, the message needs to be parsed and split into fields which correspond to columns of the database table that holds log data. If the log message text is unstructured, such parsing can quite often be done only with regular expressions. Secondly, the database insertion operation consumes a lot more CPU time than a write operation to a file, which is done with a single system call. Nevertheless, the additional expense of storing log data to a database allows for much more flexible and efficient searching. Moreover, new logging standards that support structured log messages (e.g. IETF syslog) reduce the cost of parsing, since the message already contains clearly defined fields. Therefore, many commercial SIEM solutions are using an SQL database as primary event storage.

However, the use of SQL databases introduces one issue – each database table contains a fixed number of columns, with each column representing a certain message field of a fixed type (e.g. integer or string). As a consequence, the fields of a log message have to comply with the structure of a database table that is holding the log data. In order to address this requirement, fields of all collected messages must be well known in advance, so that appropriate database schema can be defined. For instance, suppose the central log server is used for collecting Snort IDS alarms, with the following example depicting an incoming alarm in BSD syslog format:

<33>Nov 25 17:37:12 ids4 snort[776]: [1:1156:13] WEB-MISC apache directory disclosure attempt [Classification: Attempted Denial of Service] [Priority: 2]: {TCP} 10.2.1.99:41337 -> 192.168.1.1:80

For storing such alarms, a database table can be set up in a straightforward way with the following columns: Timestamp, Hostname, SignatureID, SignatureRevision, AlarmText, Classification, Priority, Transport, SourceIP, SourcePort, DestinationIP, and DestinationPort. For the above example alarm, the columns would be populated as follows: Timestamp="Nov 25 17:37:12"; Hostname="ids4";

SignatureID="1:1156"; SignatureRevision="13"; AlarmText="WEB-MISC apache directory disclosure attempt"; Classification="Attempted Denial of Service"; Priority="2"; Transport="TCP"; SourceIP="10.2.1.99"; SourcePort="41337"; DestinationIP="192.168.1.1"; DestinationPort="80".

3.3. Storing logs into document-oriented databases

Unfortunately, if logs are received from a wide variety of sources, the environment is constantly changing and log messages in previously unseen formats are likely to appear. The use of SQL databases involves a lot of overhead administrative work, since for each new message format a new parsing scheme has to be set up. Furthermore, if incoming messages have a wide variety of fields, it is cumbersome to store them into the same database table, since the table must have columns for all relevant message fields. Although some SIEM solutions define database tables with a large number of columns and leave a number of them for storing custom log data, this approach is nevertheless cumbersome and not scalable.

In order to address this problem, *document-oriented databases* have emerged as alternative log storage solutions during the last 1-2 years. Document-oriented databases are an important category of noSQL databases which do not employ tables with predefined columns for storing the data. Although the implementations of document-oriented databases vary, they can be viewed as a collection of *documents*, where each document is usually a record of fieldname-value pairs. The database engine allows for searching stored documents based on various criteria, e.g. retrieve documents with some fields having a certain value. Depending on the database, various formats like JSON and XML are supported for inserted documents. It is important to note that each inserted document can have a unique set of field names which do not need to be known in advance. For example, the following log messages in JSON format can be stored into the same database:

{"timestamp":"Nov 25 17:37:12","host":"ids4","program":"snort","processid":776,"sigid":"1:1156", "revision":13,"alarmtext":"WEB-MISC apache directory disclosure attempt","classification": "Attempted Denial of Service","priority":2,"transport":"TCP","sourceip":"10.2.1.99","sourceport": 41337,"destinationip": "192.168.1.1","destinationport":80}

{"timestamp":"Nov 25 17:38:49","host":"myhost","program":"sshd","processid":1022,"messagetext": "Failed password for risto from 10.1.1.1 port 32991 ssh2","username":"risto","sourceip":"10.1.1.1", "sourceport":32991}

During the last 1-2 years, Elasticsearch has emerged as one of the most widely used document-oriented database engines for storing log data [Elasticsearch]. Elasticsearch is written in Java and used as a backend by several recently created popular log management packages like Graylog2, Logstash and Kibana. From well-established syslog servers, it is also supported by rsyslog. Elasticsearch accepts new documents in JSON format over a simple HTTP interface, inserting the incoming document into a given database index and thus making the document searchable for future queries. If the index does not exist, it will be created automatically. For example, the following simple UNIX command line uses the *curl* utility for manual insertion of the above example JSON-formatted log message into the local Elasticsearch database (by default, the server accepts new documents at the TCP port 9200). The log message is inserted into the index *syslog*, with the message type set to *ssh* and message ID set to 1:

curl -XPUT 'http://localhost:9200/syslog/ssh/1' -d '{"timestamp":"Nov 25 17:38:49","host":"myhost", "program":"sshd","processid":1022,"messagetext": "Failed password for risto from 10.1.1.1 port 32991 ssh2","username":"risto","sourceip":"10.1.1.1", "sourceport":32991}'

Support for distribution is built into the core of Elasticsearch – several instances of Elasticsearch engines can be easily joined into a single cluster, while automated discovery of new cluster nodes is implemented through network multicast. Furthermore, each database index can be split into so-called *shards,* which can be located at different cluster members. Also, each index can have one or more *replicas* for implementing a fault tolerant cluster. By default, each index is divided into 5 shards and has 1 replica (i.e. 5 replica shards). With the default setup, each index can be distributed over the cluster of up to 10 nodes. Many administrative tasks can be carried out over the web interface in a simple way, as the message insertion example above. For instance, the following command line creates a new index *syslog2* with 10 shards and no replicas:

curl -XPUT 'http://localhost:9200/syslog2/' -d '{ "settings" : { "index" : { "number_of_shards":10, "number_of_replicas":0 } } }'

Also, the following command line changes the number of replicas from 0 to 1 for the *syslog2* index:

curl -XPUT 'http://localhost:9200/syslog2/_settings' -d '{ "number_of_replicas":1 }'

4. Syslog servers

In the previous sections, we reviewed log collection protocols and log storing techniques. In this section, we will cover leading open-source syslog servers which implement all syslog protocol flavors, and are able to store logs in ways described in the previous section. Three most widely used syslog servers in the open-source domain are rsyslog, syslog-ng and nxlog [Rsyslog, Syslog-ng, Nxlog]. The discussion and experiments presented in this section are based on most recent stable versions of each syslog server – rsyslog 7.2.3, syslog-ng 3.3.7 and nxlog ce-2.0.927.

4.1. Rsyslog, syslog-ng and nxlog

Rsyslog, syslog-ng and nxlog have been designed to overcome the weaknesses of traditional UNIX syslogd server implementations, which typically support only UDP-based BSD syslog protocol, and are able to match and process messages by facility and severity. For example, the following UNIX syslogd statement will match all messages for the daemon and user facility with the severity warning or higher, and store such messages to the file /var/log/warn.log:

daemon.warning;user.warning /var/log/warn.log

Rsyslog, syslog-ng and nxlog support not only such simple message matching, but advanced message recognition with regular expressions, conversion of messages from one format to another, authenticated and encrypted communications over IETF syslog protocol, etc. For syslog-ng and nxlog, two separate versions exist: apart from an open-source version, there is also a commercial edition with extended functionality. Rsyslog and syslog-ng run on UNIX platforms, while nxlog is also able to work on Windows.

The configuration of all servers is stored in one or more textual configuration files. Syslog-ng uses a highly flexible and readable configuration language which is not compatible with UNIX syslogd. The message sources, matching conditions and destinations are defined with named blocks, with each definition being reusable in other parts of the configuration. Syslog-ng is also very well documented, featuring a detailed administrator's manual consisting of hundreds of pages. This makes it easy to create fairly complex configurations, even for inexperienced users.

Rsyslog uses a quite different configuration language that supports UNIX syslogd constructs. This allows for easy migration of old syslogd setups to rsyslog platform. Apart from UNIX syslogd constructs, there are many additional features in the rsyslog configuration language. Unfortunately, over time, several different syntaxes have been included in the language which has introduced inconsistencies [Rsyslog-BSD]. Furthermore, rsyslog lacks a comprehensive administrator's manual, and at the rsyslog web site a limited set of configuration examples are provided that do not cover all configuration scenarios. Fortunately, a lot of information can be found from other web sites, and rsyslog has an active mailing list. Functionality-wise, rsyslog supports several highly useful features not present in the open-source versions, syslog-ng and nxlog. Firstly, it is possible to set up temporary message buffering to local disk for log messages which were not sent successfully over the network. The buffering is activated when the connection with a remote peer is disrupted, and when the peer becomes available again, all buffered messages are retransmitted. Secondly, the latest stable release of rsyslog has a support for Elasticsearch database (see [Rsyslog-ver7]).

Nxlog uses the Apache style configuration language. As with syslog-ng, message sources, destinations and other entities are defined with named blocks which allows them to be reused easily. Also, nxlog has a solid user manual. The advantages of nxlog over other syslog servers include native support for

Windows platform and Windows Eventlog. Also, nxlog is able to accept input events from various sources not directly supported by other servers, including SQL databases and text files in custom formats. Finally, nxlog is able to produce output messages in the GELF format, which makes it easy to integrate it with the Graylog2 log visualisation solution.

In order to illustrate the differences between the configuration languages of syslog-ng, rsyslog and nxlog, we have provided configuration statements in three languages for the same log processing scenario:

```
##### configuration for syslog-ng

@version:3.3

source netmsg { udp(port(514)); };
filter ntpmsg { program('^ntp') and level(warning..emerg); };
destination ntplog { file("/var/log/ntp-faults.log"); };

log { source(netmsg); filter(ntpmsg); destination(ntplog); };

##### configuration for rsyslog

$ModLoad imudp
$UDPServerRun 514

if re_match($programname, '^ntp') and $syslogseverity <= 4 then {
  action(type="omfile" file="/var/log/ntp-faults.log")
}

##### configuration for nxlog

<Input netmsg>
   Module  im_udp
   Host      0.0.0.0
   Port      514
   Exec      parse_syslog_bsd();
</Input>

<Output ntplog>
   Module   om_file
   File       "/var/log/ntp-faults.log"
   Exec       if $SourceName !~ /^ntp/ or $SyslogSeverityValue > 4 drop();
</Output>

<Route ntpfaults>
   Path      netmsg => ntplog
</Route>
```

First, the above configurations tell syslog servers to accept BSD syslog messages from UDP port 514 (in the case of syslog-ng and nxlog, the name *netmsg* is assigned to this message source). Then, the message filtering condition is defined for detecting messages with the *Tag* field matching the regular expression *^ntp* (in other words, the name of the sending program begins with the string “ntp”), and with the message severity falling between *warning* (code 4) and *emerg* (code 0). Note that for nxlog,

the inverse filter is actually used for dropping irrelevant messages. Finally, the file /var/log/ntp-faults.log is used as a destination for storing messages that have passed the filter (in the case of syslog-ng and nxlog, the name *ntplog* is assigned to this destination).

4.2. Experiments for evaluating the performance of rsyslog, syslog-ng and nxlog

In order to evaluate how well each server is suited for the role of a central syslog server, we conducted a number of experiments for assessing their performance. During the experiments we used three benchmarks for stress-testing the servers, and measured the CPU time consumption and overall execution time of each server during every test run. We call the benchmarks BSD-Throughput, IETF-Throughput and Filter-Throughput, and define them as follows:

1) BSD-Throughput – set up 1 client which sends 10,000,000 plain-text BSD syslog messages to the syslog server over TCP. The messages are written to one log file without any filtering.
2) IETF-Throughput – set up 1 client which sends 10,000,000 encrypted IETF syslog messages to the syslog server over TCP. The messages are written to one log file without any filtering.
3) Filter-Throughput – set up 5 clients, each sending 2,000,000 plain-text BSD syslog messages to the syslog server over TCP. The messages are identical to messages of the BSD-Throughput benchmark which allows for making performance comparisons between two benchmarks. The server is configured to process incoming log data with 5 filters and to write messages into 5 log files. All filters include regular expression match conditions for the message text (*Content* field) and/or program name (*Tag* field), and some filters also have additional match conditions for message facility and severity.

Note that multi-threading is built into the core of rsyslog and nxlog, while for syslog-ng this mode has to be enabled manually. During the testing we discovered that for BSD-Throughput and IETF-Throughput syslog-ng, performance decreased in multi-threading mode. According to the syslog-ng manual, this mode yields performance benefits in the presence of many clients, filters and message destinations, while for 'single-client, single-destination' scenario, computing overhead could be involved. Therefore, we ran syslog-ng in a default single-threaded mode for BSD-Throughput and IETF-Throughput tests. Also, we discovered that the tested nxlog version was not able to handle IETF syslog messages as required by RFC5425 – in a stream of incoming messages, only the first syslog frame was properly recognised. Also, the tested version was not able to recognise 'Z' as the timezone specification in the RFC3339 timestamp, but incorrectly considered it as the name of the sending host. For this reason, we had to modify the IETF-Throughput benchmark for nxlog, so that instead of proper RFC5425 frames, a stream of newline-separated IETF messages was sent to nxlog over TLS connection (note that this unofficial data transmission mode is supported by all tested servers as an extension to RFC-defined modes). The tests were carried out on a Fedora Linux node with 8GB of memory and an Intel Core i5 650 processor. We repeated each test 10 times, and the results are presented in Table 1.

The results reveal several interesting aspects of server performances. Firstly, the performance of rsyslog is superior to other servers, both in terms of raw message throughput from single client and efficiency of message filtering for multiple clients. Also, rsyslog is able to share its workload between several CPU cores with multi-threading, and thus the execution times are less than overall consumed CPU times. Multi-threading is used very efficiently by nxlog, resulting in execution times being 1.5-3 times lower than used CPU time. Unfortunately, the performance of nxlog filters is very poor and even the introduction of a few regular expression filters decreases the performance significantly: compared with the BSD-Throughput test, the average CPU time consumption increased about 26

times. In contrast, CPU time consumption for syslog-ng increased only 2.27 times, while the average execution time actually decreased by almost a half due to the manually enabled multi-threading mode.

	rsyslog	syslog-ng	nxlog
BSD-Throughput maximum, minimum and average CPU time consumption (seconds)	17.994 14.601 16.024	97.094 88.942 94.090	86.809 83.929 85.100
BSD-Throughput maximum, minimum and average execution time (seconds)	12.778 10.736 11.853	98.961 90.715 96.079	54.261 52.253 53.281
IETF-Throughput maximum, minimum and average CPU time consumption (seconds)	47.190 41.448 43.883	115.684 106.813 111.823	166.455 161.536 164.605
IETF-Throughput maximum, minimum and average execution time (seconds)	33.268 30.184 31.337	128.055 108.911 115.084	71.605 69.434 70.404
Filter-Throughput maximum, minimum and average CPU time consumption (seconds)	50.265 45.626 47.661	216.093 211.143 213.683	2237.954 2191.502 2216.758
Filter-Throughput maximum, minimum and average execution time (seconds)	44.389 39.941 41.624	60.496 58.886 59.792	715.320 701.210 707.933

Table 1. Comparative performance of rsyslog, syslog-ng and nxlog.

5. Log visualisation and preprocessing applications

The syslog servers discussed in the previous section are able to receive large volumes of log data over the network, and store these data into files and databases. While the use of databases for storing log messages facilitates fast searching with flexible query language, it would be tedious and time-consuming for the user to write separate queries for each search, report or other analytical task. Furthermore, the output from database queries is textual and the user would have to use a separate tool, or even programming language, for visualising this output. In order to solve this problem, several open-source log visualisation applications have been developed during the last 1-2 years, which are all using Elasticsearch as their main database engine. In this section, we will review and conduct performance evaluation experiments for Logstash (version 1.1.5), Graylog2 (version 0.9.6) and Kibana (version 0.2.0).

5.1. Logstash

Logstash is a Java-based utility where a graphical user interface and embedded Elasticsearch engine are encapsulated into a standalone jar-file [Logstash]. This eases the installation of Logstash since the user does not have to download and install all product components separately. One advantage of Logstash is its support for many different input types, through more than 20 input plugins. Currently, there are input plugins for accepting unencrypted syslog messages over TCP and UDP, but also for many other messaging protocols like AMPQ, RELP, GELF, IRC, XMPP, twitter, etc. Unfortunately, there is no support for receiving encrypted IETF syslog messages and for that purpose a relay syslog server has to be used which decrypts messages and forwards them to Logstash. Another advantage of Logstash is a number of different message filter types which allow for flexible recognition, filtering, parsing and conversion of messages. For instance, it is possible to convert multi-line messages into single line messages, filter out messages with regular expressions, add new fields to messages from external queries and accomplish many other advanced message manipulation tasks.

One of the most commonly used Logstash filter types is *grok*. While most log management tools use regular expression language for message matching and parsing, *grok* filters employ many predefined patterns that represent regular expressions for common matching tasks [GrokPatterns]. For instance, the pattern PROG is defined as the regular expression *(?:[\w._/%-]+)* and is designed to match the name of the logging program, the pattern POSINT is defined as the regular expression *\b(?:[1-9][0-9]*)\b* and matches a positive integer, etc. For parsing a certain field out of the message, the *%{pattern:field}* syntax is used. Also, already defined patterns can be used as building blocks for defining new patterns. Using predefined *grok* patterns, a person who is not familiar with the regular expression language can accomplish event parsing tasks in an easier way. For example, the following grok pattern matches a syslog message from Snort IDS

snort(?:\[%{POSINT:process_id}\])?: \[%{DATA:signature}:\d+\] %{DATA:alarmtext}
\{%{DATA:proto}\} %{IP:srcip}(?::%{POSINT:srcport})? -> %{IP:dstip}(?::%{POSINT:dstport})?

and sets message fields *process_id*, *signature*, *alarmtext*, *proto*, *srcip*, *srcport*, *dstip*, and *dstport*.
Apart from writing to Elasticsearch database, Logstash support many other outputs, with more than 40 output plugins being currently available. Among outputs, other monitoring and visualisation systems are supported, including Nagios, Zabbix, Loggly, Graphite and Graylog2.

In order to use the GUI of Logstash, Logstash must be configured to insert events into its embedded Elasticsearch database. With the GUI it is possible to carry out basic searches from log messages in the embedded database. Unfortunately, compared with other log visualisation tools, the GUI of

Logstash has quite limited functionality. However, since Logstash has powerful event filtering and conversion capabilities, it is used mostly as an event preprocessor for different systems, including other log visualisation systems.

5.2. Graylog2

Graylog2 (see [Graylog2]) is a log management system which consists of a Java-based server and a web interface written in Ruby-on-Rails. Graylog2 server can receive BSD syslog messages over UDP and TCP, but it also features its own GELF protocol (see [GELF]) that facilitates structured logging. In addition, Graylog2 can accept syslog and GELF messages over the AMPQ messaging protocol. Unfortunately, Graylog2 is not able to parse structured IETF syslog messages and recognise already defined fieldname-value pairs. For BSD syslog messages, Graylog2 can recognise the standard *Priority*, *Timestamp*, *Hostname* and *MSG* fields, but cannot by default parse the unstructured *MSG* field.

The parsing problem for unstructured messages can be overcome in several ways. First, Graylog2 server supports message parsing and rewriting through Drools Expert rules and regular expressions [Graylog2-Drools]. Second, since Logstash supports advanced conversions between many message formats with flexible parsing, many sites are using Logstash for receiving syslog messages, parsing out relevant message fields with *grok* filters, and finally sending the parsed messages in structured GELF format to Graylog2. Finally, since the nxlog syslog server supports the GELF protocol, it can be used as a front end for receiving both encrypted and plain-text syslog messages and converting them into structured GELF messages.

For storing the parsed messages, Graylog2 uses Elasticsearch as its main back end (some information about messages is also stored in the MongoDB document-oriented database [MongoDB], which is used for graph creation purposes). Unfortunately, all log messages are stored into a single index called *graylog2,* which might cause performance issues as many log messages are inserted into it over time. This issue can be partly addressed by configuring a shorter message retention time through the Graylog2 web interface (by default, messages are kept in the database for 60 days). Nevertheless, it would be much more efficient to create a separate Elasticsearch index on a weekly or daily basis (the latter technique is used by Logstash). Fortunately, the developers of Graylog2 are aware of this problem and it is supposed to be fixed in the next version.

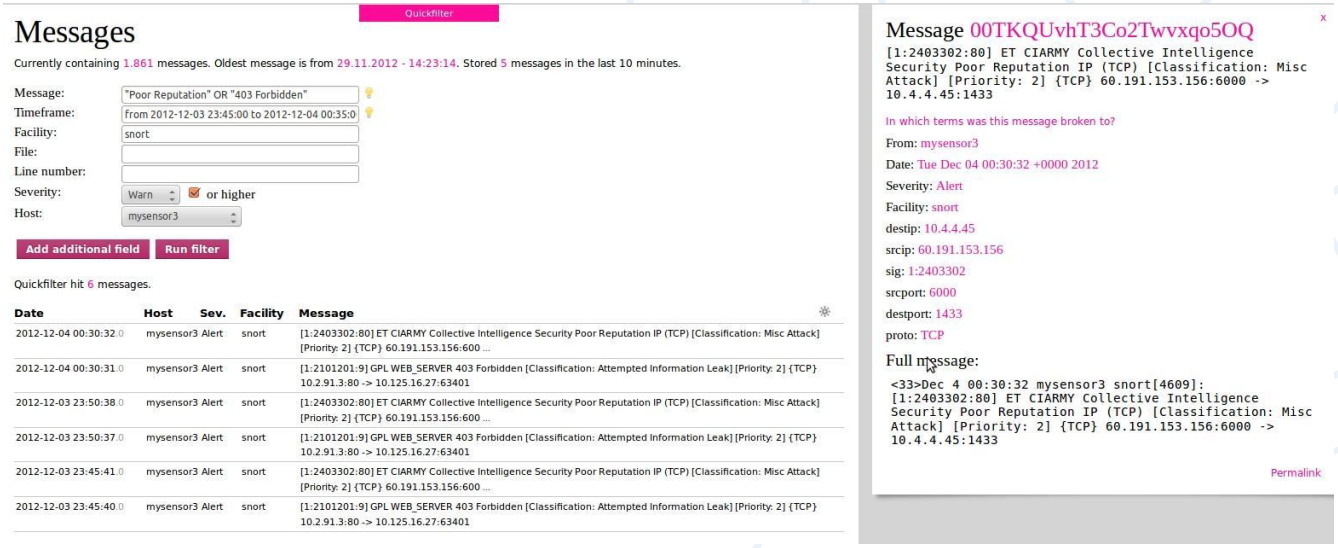

Figure 2. Graphical user interface of Graylog2.

In order to visualise collected log data, Graylog2 provides a well-written and comprehensive web interface. For accessing the interface, different password-protected user accounts can be set up, with each user having either full or limited rights (the remainder of the discussion will concern the user interface with full admin rights). The interface is divided into several parts. The message view (see Figure 2) allows for getting an overview of stored log messages, presenting the messages in a browser with the most recent messages coming first. In the browser, the timestamp, host, severity, facility and message text fields are displayed. By clicking on an individual message, a detailed view of the message is provided which contains all fieldnames with values (see the right-hand part of Figure 2). Clicking on each individual value will carry out a search for messages with the same fieldname-value pair, and discovered messages will be displayed in the main message browser. Message search can also be initiated through a separate 'Quickfilter' button in the message view which allows for specifying more than one search condition. For searching a message text field, the Apache Lucene query syntax can be used (see [Lucene]). It supports searches for individual strings, approximate string matching based on Levenshtein distance, proximity searches (e.g. find two strings which have up to 10 words in between), and combining individual search conditions with Boolean operators. Figure 2 depicts an example search for Snort IDS alarms.

Apart from viewing all messages, the user can configure streams that are collections of messages satisfying some message filtering condition. Streams are updated with matching incoming messages in real-time. Messages under each stream can be viewed separately, and also it is possible to configure thresholding and alarming conditions for each stream (e.g. send an alarm to a security administrator if more than 10 messages have appeared under the stream during 1 minute). In order to define a filtering condition for a stream, message field values can be compared with fixed values, but in the case of some fields, also with regular expressions. When writing regular expressions for matching message text, by convention the expression must match the entire message (e.g. for matching messages which contain the string 'test' in the message text, the regular expression has to be written not as *test*, but rather as *.*test.** or *^.*test.*$*). In addition to streams, Graylog2 also contains a so called analytics shell which supports a flexible query language for finding individual messages and for creating various reports. Unfortunately, currently all reports are text-based, although in the future releases support for graphical reporting might be added.

During our experiments, we attempted to establish the performance of Graylog2 in terms of event throughput. We expected the performance of Graylog2 to be significantly slower than for syslog servers tested in the previous section. First, both Graylog2 server and Elasticsearch database engine are written in Java, while rsyslog, syslog-ng and nxlog are coded in C. Second, Graylog2 server has to insert log messages into Elasticsearch index, which requires considerably more CPU time than writing messages into flat files. During the tests, we ran Graylog2 on a Fedora Linux node with 8GB of memory and an Intel Core i5 650 processor. We set up one client for Graylog2, which was issuing large amounts of BSD syslog messages over TCP. The combined performance of Graylog2 server and Elasticsearch backend was measured in terms of message transmission throughput observed at the client side. During several tests, we were able to reach a throughput of 3,500 messages per second. This illustrates that one Graylog2 server instance is not scalable to very large environments with many thousands of logging hosts and heavy message loads. Fortunately, the developers are planning to add support into Graylog2 for multiple server instances, which should substantially increase its overall scalability.

5.3. Kibana

Kibana is another application for visualising collected log data. Unlike Graylog2, Kibana consists only of a Ruby-based web interface which uses Elasticsearch as a back end, and there is no server to receive log messages over the network and store them into a database. For this reason, Kibana cannot run as a standalone system, but has to be used with an application that receives, parses, and stores log messages into Elasticsearch. Many sites are employing Logstash for this task, and Kibana's default configuration is Logstash-compliant. Also, Kibana expects that log messages in Elasticsearch database have some fields that are created by Logstash (for example, *@timestamp* and *@message*). However, if another application is configured to insert log messages into Elasticsearch database with these fieldname-value pairs, Kibana is able to work on stored log data.

In order to search log messages, Kibana supports full Apache Lucene query syntax for all message fields (Figure 3 depicts an example search for Snort IDS alarm data). One advantage of Kibana over Graylog2 is the support for the creation of various graphical reports. Reports can be created based on a selected message field and time frame, either for all messages or for some message matching criteria. Kibana supports the creation of pie charts which reflect the distribution of field values, trend analysis reports and count reports for field values. By selecting some value from the report form, the user can go to relevant log messages. Reports can also be created directly from searches – for example, the query *@fields.srcip=10.1.1.1* selects all messages where the field @fields.srcip (source IP address) has the value 10.1.1.1, while the query

@fields.srcip=10.1.1.1 | terms @fields.dstip

creates a pie graph about the distribution of @fields.dstip (destination IP address) values for source IP 10.1.1.1.

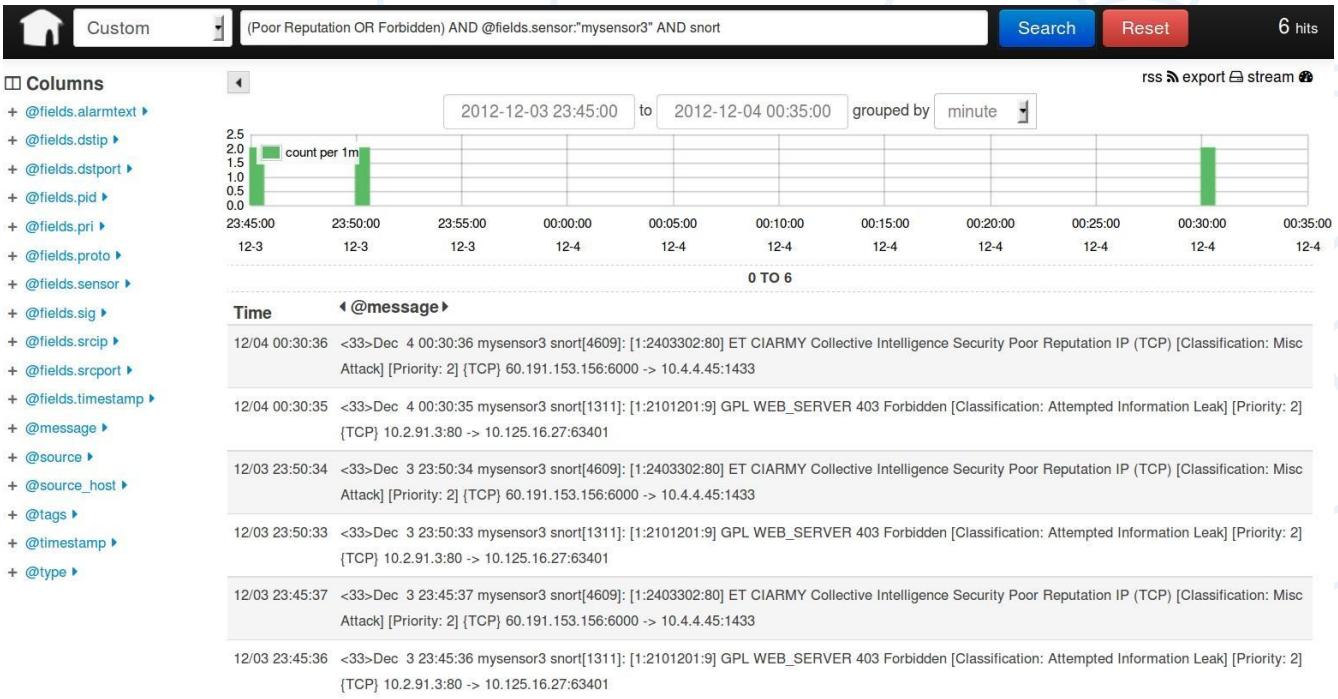

Figure 3. Graphical user interface of Kibana.

Since rsyslog has had Elasticsearch support since 2012, it can be used instead of Logstash for receiving and preparing log data for Kibana. The following rsyslog configuration statements accomplish a simple gateway between BSD syslog messages and Elasticsearch:

```
$ModLoad /usr/local/lib/rsyslog/omelasticsearch.so

$template Syslog2Json, "{\"@timestamp\":\"%timereported:::date-rfc3339%\",
\"@message\":\"%msg:::json%\",\"@source\":\"unknown\",\"@type\":\"syslog\",\"@tags\":[],
\"@fields\":{\"receptiontime\":\"%timegenerated:::date-rfc3339%\",
\"host\":\"%HOSTNAME:::json%\",\"tag\":\"%syslogtag:::json%\"}}"

$template SyslogIndex, "rsyslog-%timereported:1:10:date-rfc3339%"

daemon.* action(type="omelasticsearch" template="Syslog2Json" dynSearchIndex="on"
searchIndex="SyslogIndex" server="localhost" bulkmode="on")
```

The first statement loads the Elasticsearch output module for rsyslog. The second statement defines a template called *Syslog2Json,* which converts BSD syslog message to JSON format. The values between percent signs, like *timereported,* denote fields extracted from the syslog message, while the *:::date-rfc3339* suffix converts the given timestamp into RFC3339 format. Also, the *:::json* suffix after the field name means that the field is made JSON-compatible (for example, double quotes need to be escaped with a backslash, since they are used as delimiters in JSON records). The third statement defines a separate Elasticsearch index for each day, which has the format *rsyslog-YYYY-MM-DD* (e.g. *rsyslog-2012-12-02*). The fourth statement matches all messages with the *daemon* facility and inserts them to *syslog-YYYY-MM-DD* index of the local Elasticsearch database. The *bulkmode="on"* statement enables bulk inserts – instead of inserting each log message separately, larger numbers of messages are inserted in one batch with a single insert operation which increases the message throughput significantly.

In order to assess the performance of Logstash and rsyslog, we installed Kibana with Elasticsearch on a Fedora Linux node with 8GB of memory and an Intel Core i5 650 processor, and set up both rsyslog and Logstash at this node. Both solutions were configured to insert messages into Elasticsearch in bulk mode (for rsyslog, the message batch size was 16, while for Logstash a batch size of 100 was used). For performance evaluation, we sent 100,000 BSD syslog messages over TCP to the receiver, and measured the processing time of these messages. At the end of each test, a query was made to Elasticsearch for verifying that all messages were successfully inserted into the database. We repeated this test 100 times for rsyslog and Logstash, deleting all inserted messages between consecutive test runs. The results of our experiments are presented in Table 2. For rsyslog, 100,000 messages were processed in an average of 17.066 seconds, yielding the average processing speed of 5859.6 messages per second. In the case of Logstash, the average processing speed was 1732.952 messages per second. In other words, rsyslog is able to insert messages into Elasticsearch more than 3 times faster than Logstash.

	Minimum processing time (seconds)	Maximum processing time (seconds)	Average processing time (seconds)	Average event processing speed (events per second)
rsyslog	15.998	21.031	17.066	5859.604
Logstash	56.084	73.695	57.705	1732.952

Table 2. Comparative performance of rsyslog and Logstash for Elasticsearch bulk insert operations.

During the testing, we also discovered one annoying feature of Logstash – for bulk inserts, messages are accumulated into a buffer which is flushed only when full. Therefore, log messages could stay in the buffer for long periods of time if the buffer still contains some free space. In contrast, rsyslog implements a more efficient bulk insertion algorithm which does not leave unprocessed messages pending in memory.

6. Summary

In this paper, we have reviewed a number of widely used and efficient open-source solutions for collecting log data from IT systems. We have also described some novel technologies and setups for tackling the logging in large networks. One of the major contributions of this paper is the performance evaluation of described solutions through a series of benchmarks which mimic heavy workload in real-life environments. Through the experiments conducted, we have been able to identify specific advantages and weaknesses of each tool (e.g. efficient multi-threading or event filtering). Although our tests indicate that some tools have superior performance under specific circumstances, each tool offers a unique set of features to the end user. Therefore, the selection of a log management tool depends heavily on the specific nature of the environment. In order to automate the experiments for evaluating log management tools, we have written a simple toolkit consisting of a few Perl and UNIX shell scripts. For future work, we plan to elaborate our testing tools and release them to the public domain.

References:

[RFC3164] http://www.ietf.org/rfc/rfc3164.txt
[RFC3339] http://www.ietf.org/rfc/rfc3339.txt
[RFC5424] http://www.ietf.org/rfc/rfc5424.txt
[RFC5425] http://www.ietf.org/rfc/rfc5425.txt
[RFC5426] http://www.ietf.org/rfc/rfc5426.txt
[CEE] http://cee.mitre.org/language/1.0-beta1/
[Net-SNMP] http://net-snmp.sourceforge.net
[SNMPTT] http://snmptt.sourceforge.net
[Elasticsearch] http://www.elasticsearch.org
[Rsyslog] http://www.rsyslog.com
[Syslog-ng] http://www.balabit.com/network-security/syslog-ng/
[Nxlog] http://http://nxlog-ce.sourceforge.net/
[Rsyslog-BSD] http://blog.gerhards.net/2012/09/bsd-style-blocks-will-go-away-in.html
[Rsyslog-ver7] http://www.rsyslog.com/main-advantages-of-rsyslog-v7-vs-v5/
[Logstash] http://logstash.net
[GrokPatterns] https://github.com/logstash/logstash/blob/master/patterns/grok-patterns
[Graylog2] http://graylog2.org
[GELF] http://www.graylog2.org/about/gelf/
[Graylog2-Drools] https://github.com/Graylog2/graylog2-server/wiki/Message-processing-rewriting
[MongoDB] http://www.mongodb.org/
[Lucene]
https://lucene.apache.org/core/old_versioned_docs/versions/3_5_0/queryparsersyntax.html
[Kibana] http://kibana.org