Designing filters for ELK

Rocco Gagliardi
Defense Department, scip AG
roga@scip.ch
https://www.scip.ch

Marc Ruef (Editor)
Research Department, scip AG
maru@scip.ch
https://www.scip.ch

Keywords: Complexity, Dashboard, Firewall, Kibana, Policy, Report, Tool, Windows

1. Preface

This paper was written in 2014 as part of a research project at scip AG, Switzerland. It was initially published online at *https://www.scip.ch/en/?labs.20141023* and is available in English and German. Providing our clients with innovative research for the information technology of the future is an essential part of our company culture.

2. Introduction

Log messages are generated on many system components and applications on systems itself.

In many cases, the documentation does not exist or is very superficial. In order to make sense of this mass of information, applying calculations and making predictions is very complex.

This lab summarises experiences collected during the design and implementation of some *ELK* [1] at various customer sites.

3. Problem

Interpreting a log file means to extract the information we need and ignore the rest.

In data transferred via syslog, the field *data* has no strictly predefined format, and can also contain any data type in any order. There are about 1000 different log formats out there at the moment. Each application logs specific information in proprietary formats; sometimes the same application even uses different formats for different events.

Time is needed to create filters so that received data is interpreted in a structured and coherent way. In addition to that, with increasing amounts of log interpretation, the system becomes complex and prone to errors.

4. Practical Model Solution

To effectively and efficiently manage the filters, a good dose of flexibility and robustness is necessary. It is not merely a matter of programming. To obtain an effective system, accurate documentation is required of how data flows and how each filter modifies the original message.

4.1. Filter Organisation

It's a good idea to split the filter across different files:

Series	Filter	Comment
10_*	Input Definition	Here we find the definitions of all the input, such as syslog / lumberjack / file / other.
20_*	Input Classification	To boot, it is necessary to classify the messages. Matching regex, specific sources or programs and so on can be used to uniquely tag each message for further manipulation.
30_*	Message Interpretation	Based on the tags entered earlier, the message goes through the various filters and will be analysed by the ones deputed to handle the specific tag. Extracted fields are to be indexed and, if necessary, modified in different manners (e.g. normalization of the names of the extracted fields). Even if multiple mechanisms can be used to extract information from a message, the most useful tool is *grok* as mentioned in *Interpreting a Log file with Grok* [2].
90_*	Data Output	The analysis is complete, Now it is decided what to do with the data. For example, send the measurements to Carbon or save only a few messages or messages fields, etc.

For each analysis filter, develop a matrix of extracted fields and contents in order to normalise the field names. This will be a useful reference during the design of the dashboards and/or reports as well as for the correlation between different message types.

Separate filter details from filter configuration. The grok-filter (analysis) itself should just match a line against

patterns stored in the pattern database. This will enhance robustness because the config will be very simple. At the same time, the pattern database will help maintaining the field normalisation and the overview of the parsed messages and can be recursively optimised for all implemented filters.

To construct the pattern database, create a set of *atoms* grok expressions, then construct the line expression referred by the filter.

During the addition of more line parsers, some redundancy can be eliminated by increasing the complexity of the grok expression while maintaining good flexibility with the *atom* expressions.

During the expression development process, use *grokdebug* [3] to *interactively* test patterns. For the regex, use *Rubular* [4] to *interactively* test the regex.

With time, it will become useful to customise. *Buy* [5] your own environment and interact with grok using *IRB* [6].

Use a separate platform to syntactically test the *filter-set* and *pattern-db*. Note: If you use CentOS, the rpm package has some problems and will not start in *test mode*. Use the same ELK version on OSX/Windows/Debian to test the filters.

5. Example: Building the Fortigate Filter

The Logstash grok plugin automatically tags non matching messages with the *grokparsefailure* tag. Monitor this tag trend to find messages that are not captured by the filter.

Basically, the development pattern to use is:

1. Assure all messages from the Fortigate firewall are correctly tagged
2. Set the correct filter
3. Create a first message parser
4. repeat until failure rate drops to around zero messages
    1. Add parsers for additional messages
    2. Optimize patterns
    3. Monitor with Kibana the *grokparsefailure* trend

5.1. Patterns

Develop the pattern *atoms*, designing some expression dedicated to grep a specific part of the log message.

```
# Fortigate atoms parsers
FWFN_BASE %{SYSLOGTIMESTAMP} %
{IPORHOST:logsource}.*devname=%
{DATA:dev_name} device_id=%{DATA:dev_id}
log_id=%{NUMBER:log_id} type=%{DATA:type}
subtype=%{DATA:stype} pri=%{DATA:severity}
vd=%{WORD:vdom}
FWFN_SRCDST src=%{IP:src:ip} src_port=%
{NUMBER:src_port} src_int=%
{QUOTEDSTRING:src_intf} dst=%{IP:dst_ip}
dst_port=%{NUMBER:dst_port} dst_int=%
{QUOTEDSTRING:dst_intf}
FWFN_SN SN=%{NUMBER:sn} status=%{DATA:status}
policyid=%{NUMBER:pol_id}
FWFN_CNTR dst_country=%
{QUOTEDSTRING:dst_country} src_country=%
{QUOTEDSTRING:src_country}
FWFN_SVC service=%{DATA:service} proto=%
{NUMBER:proto} duration=%{NUMBER:duration}
sent=%{NUMBER:B_sent} rcvd=%{NUMBER:B_rcvd}
FWFN_MSG msg=%{QUOTEDSTRING:msg}
FWFN_TRAN dir_disp=%{DATA:direction}
tran_disp=%{DATA:trans_type} tran_sip=%
{IP:trs_ip} tran_sport=%{NUMBER:trs_port}
```

Then start to construct the line message parsers:

```
#
# Fortigate message line matchers (Version 1
- Straight forward)
FWFN_VAR_01 %{FWFN_BASE} %{FWFN_SRCDST} %
{FWFN_SN} %{FWFN_CNTR} %{FWFN_SVC}
FWFN_VAR_02 %{FWFN_BASE} %{FWFN_SRCDST} %
{FWFN_SN} %{FWFN_CNTR} %{FWFN_SVC} %
{FWFN_MSG}
FWFN_VAR_03 %{FWFN_BASE} %{FWFN_SRCDST} %
{FWFN_SN} %{FWFN_CNTR} %{FWFN_TRAN} %
{FWFN_SVC}
FWFN_VAR_04 %{FWFN_BASE} %{FWFN_SRCDST} %
{FWFN_SN} %{FWFN_CNTR} %{FWFN_TRAN} %
{FWFN_SVC} %{FWFN_MSG}
```

You can quickly notice redundancy in the code. Use the DRY principle and start recursively summarising the grok expressions:

```
#
# Fortigate message line matchers (Version 2
- Optimization step 1 )
FWFN_BASE_2 %{FWFN_BASE} %{FWFN_SRCDST} %
{FWFN_SN} %{FWFN_CNTR}
FWFN_VAR_01 %{FWFN_BASE_2} %{FWFN_SVC}
FWFN_VAR_02 %{FWFN_BASE_2} %{FWFN_SVC} %
{FWFN_MSG}
FWFN_VAR_03 %{FWFN_BASE_2} %{FWFN_TRAN} %
{FWFN_SVC}
FWFN_VAR_04 %{FWFN_BASE_2} %{FWFN_TRAN} %
{FWFN_SVC} %{FWFN_MSG}
```

Round 2:

```
#

    1. Fortigate message line matchers
       (Version 3 – Optimization step 1 )
       FWFN_BASE_2 %{FWFN_BASE} %
       {FWFN_SRCDST} %{FWFN_SN} %{FWFN_CNTR}

FWFN_VAR_01 %{FWFN_BASE_2} %{FWFN_SVC}
FWFN_VAR_02  %{FWFN_BASE_2}  %{FWFN_SVC}  %
{FWFN_MSG}
FWFN_VAR_03  %{FWFN_BASE_2}  %{FWFN_TRAN}  %
{FWFN_SVC}
FWFN_VAR_04  %{FWFN_BASE_2}  %{FWFN_TRAN}  %
{FWFN_SVC} %{FWFN_MSG}
```

Round 3:

```
#
# Fortigate Message line matchers (Version 4
- Optimization step 1 )
FWFN_VAR_01 %{FWFN_BASE_2}\s*(?:%
{FWFN_TRAN}|)\s*(?:%{FWFN_SVC}|)\s*(?:%
{FWFN_MSG}|)
```

Constantly check the *grokparsefailure* rate for any potentially occurring uptrend. If this happens, something is wrong in the summarisation and the filter does not work as expected. In that case, go over the summarising process again.

6. Final Filter Version

After the grok-pattern optimisation, the match line in grok filter can be reduced to a single line. This keeps the filter configuration file simple and clean, helping to quickly correct possible syntax errors.

```
filter {
    if "fortigate" in [tags] {
        mutate {
            add_tag       => [ "firewall" ]
            remove_tag    => [ "fortigate" ]
            replace       => [ "program",
"fortigate" ]
            replace       => [ "severity", ""
]
        }
        grok {
                 overwrite   => [ "message" ]
                 match       => [ "message" ,
"%{FWFN_VAR_01}" ]
        }
    }
}
```

After some recursive improvements and a few hours of tuning, the *grokparsefailure* rate drops to near zero. At this stage, the filter is working correctly for most incoming messages. A weekly review of the messages tagged with *grokparsefailure* will be enough to capture the remaining unparsed messages.

7. Why is This Important?

Mid-data interpretation may be a thankless job. The system necessary to get that sort of interpretation done may be very complex and, if not handled well, may become very unstable. Maybe not for the software itself but for the customisation.

A well-structured and documented configuration, increases stability and flexibility and facilitates the integration of plugins and the enhancement of the system.

8. External Links

[1] http://www.elasticsearch.org/overview/
[2] https://www.scip.chhttp://www.scip.ch/?labs.20130405
[3] https://grokdebug.herokuapp.com
[4] http://rubular.com
[5] https://www.ruby-lang.org/en/
[6] http://www.ruby-doc.org/stdlib-2.0/libdoc/irb/rdoc/IRB.html