

# **NXLOG Community Edition Reference Manual for v2.1.957**

Copyright © 2009-2012 nxsec.com

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Overview	1
1.2	Features	1
1.2.1	Multiplatform	1
1.2.2	Modular architecture	1
1.2.3	Client-server mode	2
1.2.4	Log message sources and destinations	2
1.2.5	Importance of security	2
1.2.6	Scalable multi-threaded architecture	2
1.2.7	High performance I/O	2
1.2.8	Message buffering	2
1.2.9	Prioritized processing	3
1.2.10	Avoiding lost messages	3
1.2.11	Apache-style configuration syntax	3
1.2.12	Built-in config language	3
1.2.13	Scheduled tasks	3
1.2.14	Log rotation	3
1.2.15	Different log message formats	4
1.2.16	Advanced message processing capabilities	4
1.2.17	Offline processing mode	4
1.2.18	Character set and i18n support	4
<b>2</b>	<b>Installation and quickstart</b>	<b>5</b>
2.1	Microsoft Windows	5
2.2	GNU/Linux	6
2.2.1	Installing from DEB packages (Debian, Ubuntu)	6
2.2.2	Installing from RPM packages (CentOS, RedHat)	6
2.2.3	Configuring nxlog on GNU/Linux	6

<b>3</b>	<b>Architecture and concepts</b>	<b>7</b>
3.1	History . . . . .	7
3.2	Concepts . . . . .	7
3.3	Architecture . . . . .	8
<b>4</b>	<b>Configuration</b>	<b>9</b>
4.1	File inclusion . . . . .	9
4.2	Constant and macro definitions . . . . .	9
4.3	Global directives . . . . .	10
4.4	Modules . . . . .	11
4.4.1	Common module directives . . . . .	12
4.4.1.1	Module . . . . .	12
4.4.1.2	Schedule . . . . .	12
4.4.1.3	Exec . . . . .	13
4.4.1.4	Processors . . . . .	14
4.4.1.5	InputType . . . . .	14
4.4.1.6	OutputType . . . . .	14
4.5	Routes . . . . .	15
4.5.1	Priority . . . . .	15
4.5.2	Path . . . . .	16
<b>5</b>	<b>Language</b>	<b>18</b>
5.1	Types . . . . .	18
5.2	Expressions . . . . .	19
5.2.1	Literals . . . . .	19
5.2.2	Fields . . . . .	20
5.2.3	Operations . . . . .	20
5.2.3.1	Unary operations . . . . .	20
5.2.3.2	Binary operations . . . . .	21
5.2.4	Functions . . . . .	24
5.3	Statements . . . . .	24
5.3.1	Assignment . . . . .	25
5.3.2	Block . . . . .	25
5.3.3	Procedures . . . . .	25
5.3.4	If-Else . . . . .	25
5.4	Variables . . . . .	26
5.5	Statistical counters . . . . .	27
5.6	List of available functions and procedures . . . . .	28
5.6.1	Functions and procedures exported by core . . . . .	28
5.6.1.1	Functions exported by core . . . . .	28
5.6.1.2	Procedures exported by core . . . . .	32
5.6.2	Functions and procedures exported by modules . . . . .	36

<b>6</b>	<b>Modules</b>	<b>37</b>
6.1	Extension modules	37
6.1.1	CSV (xm_csv)	37
6.1.1.1	Configuration	37
6.1.1.1.1	Specifying characters for quote, escape and delimiter	38
6.1.1.2	Functions and procedures exported by xm_csv	39
6.1.1.2.1	Functions exported by xm_csv	39
6.1.1.2.2	Procedures exported by xm_csv	39
6.1.1.3	Configuration examples	40
6.1.2	JSON (xm_json)	40
6.1.2.1	Configuration	40
6.1.2.2	Functions and procedures exported by xm_json	41
6.1.2.2.1	Functions exported by xm_json	41
6.1.2.2.2	Procedures exported by xm_json	41
6.1.2.3	Configuration examples	42
6.1.3	XML (xm_xml)	43
6.1.3.1	Configuration	43
6.1.3.2	Functions and procedures exported by xm_xml	43
6.1.3.2.1	Functions exported by xm_xml	43
6.1.3.2.2	Procedures exported by xm_xml	44
6.1.3.3	Configuration examples	44
6.1.4	GELF (xm_gelf)	45
6.1.4.1	Configuration	46
6.1.4.2	Configuration examples	46
6.1.5	Character set conversion (xm_charconv)	47
6.1.5.1	Configuration	47
6.1.5.2	Functions and procedures exported by xm_charconv	47
6.1.5.2.1	Functions exported by xm_charconv	47
6.1.5.2.2	Procedures exported by xm_charconv	48
6.1.5.3	Configuration examples	48
6.1.6	File operations (xm_fileop)	48
6.1.6.1	Configuration	49
6.1.6.2	Functions and procedures exported by xm_fileop	49
6.1.6.2.1	Functions exported by xm_fileop	49
6.1.6.2.2	Procedures exported by xm_fileop	50
6.1.6.3	Configuration examples	53
6.1.7	Multi-line message parser (xm_multiline)	53
6.1.7.1	Configuration	53
6.1.7.2	Configuration examples	56

6.1.8	Syslog (xm_syslog) . . . . .	60
6.1.8.1	Configuration . . . . .	60
6.1.8.2	Functions and procedures exported by xm_syslog . . . . .	61
6.1.8.2.1	Functions exported by xm_syslog . . . . .	61
6.1.8.2.2	Procedures exported by xm_syslog . . . . .	61
6.1.8.3	Fields generated by xm_syslog . . . . .	62
6.1.8.4	Configuration examples . . . . .	63
6.1.9	External program execution (xm_exec) . . . . .	67
6.1.9.1	Functions and procedures exported by xm_exec . . . . .	67
6.1.9.1.1	Procedures exported by xm_exec . . . . .	67
6.1.9.2	Configuration examples . . . . .	68
6.1.10	Perl (xm_perl) . . . . .	69
6.1.10.1	Configuration . . . . .	69
6.1.10.2	Functions and procedures exported by xm_perl . . . . .	70
6.1.10.2.1	Procedures exported by xm_perl . . . . .	70
6.1.10.3	Configuration examples . . . . .	70
6.2	Input modules . . . . .	72
6.2.1	Fields generated by core . . . . .	72
6.2.2	DBI (im_dbi) . . . . .	72
6.2.2.1	Configuration examples . . . . .	72
6.2.3	Program (im_exec) . . . . .	72
6.2.3.1	Configuration . . . . .	73
6.2.3.2	Configuration examples . . . . .	73
6.2.4	File (im_file) . . . . .	73
6.2.4.1	Configuration . . . . .	74
6.2.4.2	Functions and procedures exported by im_file . . . . .	75
6.2.4.2.1	Functions exported by im_file . . . . .	75
6.2.4.3	Configuration examples . . . . .	75
6.2.5	Internal (im_internal) . . . . .	75
6.2.5.1	Fields generated by im_internal . . . . .	76
6.2.5.2	Configuration examples . . . . .	76
6.2.6	Kernel (im_kernel) . . . . .	77
6.2.6.1	Configuration examples . . . . .	77
6.2.7	Mark (im_mark) . . . . .	77
6.2.7.1	Configuration . . . . .	78
6.2.7.2	Fields generated by im_mark . . . . .	78
6.2.7.3	Configuration examples . . . . .	79
6.2.8	MS EventLog for Windows XP/2000/2003 (im_mseventlog) . . . . .	79
6.2.8.1	Configuration . . . . .	79

6.2.8.2	Fields generated by im_mseventlog . . . . .	80
6.2.8.3	Configuration examples . . . . .	81
6.2.9	MS EventLog for Windows 2008/Vista and later (im_msvistalog) . . . . .	81
6.2.9.1	Configuration . . . . .	82
6.2.9.2	Fields generated by im_msvistalog . . . . .	83
6.2.9.3	Configuration examples . . . . .	84
6.2.10	Null (im_null) . . . . .	84
6.2.11	TLS/SSL (im_ssl) . . . . .	84
6.2.11.1	Configuration . . . . .	85
6.2.11.2	Fields generated by im_ssl . . . . .	85
6.2.11.3	Configuration examples . . . . .	86
6.2.12	TCP (im_tcp) . . . . .	86
6.2.12.1	Configuration . . . . .	86
6.2.12.2	Fields generated by im_tcp . . . . .	87
6.2.12.3	Configuration examples . . . . .	87
6.2.13	UDP (im_udp) . . . . .	87
6.2.13.1	Configuration . . . . .	88
6.2.13.2	Fields generated by im_udp . . . . .	88
6.2.13.3	Configuration examples . . . . .	88
6.2.14	Unix Domain Socket (im_uds) . . . . .	88
6.2.14.1	Configuration . . . . .	89
6.2.14.2	Configuration examples . . . . .	89
6.3	Processor modules . . . . .	89
6.3.1	Blocker (pm_blocker) . . . . .	89
6.3.1.1	Functions and procedures exported by pm_blocker . . . . .	89
6.3.1.1.1	Functions exported by pm_blocker . . . . .	89
6.3.1.1.2	Procedures exported by pm_blocker . . . . .	90
6.3.1.2	Configuration examples . . . . .	90
6.3.2	Buffer (pm_buffer) . . . . .	90
6.3.2.1	Configuration . . . . .	91
6.3.2.2	Functions and procedures exported by pm_buffer . . . . .	91
6.3.2.2.1	Functions exported by pm_buffer . . . . .	91
6.3.2.3	Configuration examples . . . . .	92
6.3.3	Filter (pm_filter) . . . . .	92
6.3.3.1	Configuration . . . . .	92
6.3.3.2	Configuration examples . . . . .	93
6.3.4	Message deduplicator (pm_norepeat) . . . . .	93
6.3.4.1	Configuration . . . . .	93
6.3.4.2	Fields generated by pm_norepeat . . . . .	93

6.3.4.3	Configuration examples . . . . .	94
6.3.5	Null (pm_null) . . . . .	94
6.3.6	Pattern matcher (pm_pattern) . . . . .	94
6.3.6.1	Configuration . . . . .	95
6.3.6.2	Pattern database file . . . . .	97
6.3.6.3	Fields generated by pm_pattern . . . . .	98
6.3.6.4	Configuration examples . . . . .	98
6.3.7	Message format converter (pm_transformer) . . . . .	98
6.3.7.1	Configuration . . . . .	98
6.3.7.2	Configuration examples . . . . .	100
6.4	Output modules . . . . .	100
6.4.1	Blocker (om_blocker) . . . . .	100
6.4.1.1	Configuration examples . . . . .	101
6.4.2	DBI (om_dbi) . . . . .	101
6.4.2.1	Configuration . . . . .	102
6.4.2.2	Configuration examples . . . . .	102
6.4.3	Program (om_exec) . . . . .	104
6.4.3.1	Configuration . . . . .	104
6.4.3.2	Configuration examples . . . . .	105
6.4.4	File (om_file) . . . . .	105
6.4.4.1	Configuration . . . . .	105
6.4.4.2	Functions and procedures exported by om_file . . . . .	105
6.4.4.2.1	Functions exported by om_file . . . . .	105
6.4.4.2.2	Procedures exported by om_file . . . . .	106
6.4.4.3	Configuration examples . . . . .	106
6.4.5	HTTP(s) (om_http) . . . . .	107
6.4.5.1	Configuration . . . . .	107
6.4.5.2	Functions and procedures exported by om_http . . . . .	108
6.4.5.2.1	Procedures exported by om_http . . . . .	108
6.4.5.3	Configuration examples . . . . .	108
6.4.6	Null (om_null) . . . . .	108
6.4.7	TLS/SSL (om_ssl) . . . . .	109
6.4.7.1	Configuration . . . . .	109
6.4.7.2	Configuration examples . . . . .	110
6.4.8	TCP (om_tcp) . . . . .	110
6.4.8.1	Configuration . . . . .	110
6.4.8.2	Configuration examples . . . . .	111
6.4.9	UDP (om_udp) . . . . .	111
6.4.9.1	Configuration . . . . .	111
6.4.9.2	Configuration examples . . . . .	111
6.4.10	UDS (om_uds) . . . . .	112
6.4.10.1	Configuration . . . . .	112
6.4.10.2	Configuration examples . . . . .	112



---

<b>7</b>	<b>Offline log processing</b>	<b>113</b>
7.1	nxlog-processor . . . . .	113
<b>8</b>	<b>Reading and receiving logs</b>	<b>114</b>
8.1	Operating Systems . . . . .	114
8.1.1	Microsoft Windows . . . . .	114
8.1.1.1	Windows EventLog . . . . .	114
8.1.1.2	Microsoft SQL Server . . . . .	114
8.1.1.3	Microsoft IIS . . . . .	115
8.1.1.3.1	W3C Extended Log File Format . . . . .	115
8.1.1.3.2	Microsoft IIS Format . . . . .	115
8.1.1.3.3	NCSA Common Log File Format . . . . .	115
8.1.1.3.4	ODBC Logging . . . . .	115
8.1.2	GNU/Linux . . . . .	115
8.1.3	Android . . . . .	115
8.2	Network . . . . .	116
8.2.1	UDP . . . . .	116
8.2.2	TCP . . . . .	116
8.2.3	TLS/SSL over TCP . . . . .	116
8.2.4	Syslog . . . . .	116
8.3	Database . . . . .	116
8.3.1	Using im_dbi . . . . .	116
8.3.2	Using im_odbc . . . . .	116
8.4	Files . . . . .	116
8.5	External programs and scripts . . . . .	116
8.6	Applications . . . . .	116
8.6.1	Apache HTTP Server . . . . .	117
8.6.1.1	Error log . . . . .	117
8.6.1.2	Access log - Common Log Format . . . . .	117
8.6.1.3	Access log - Combined Log Format . . . . .	117
8.6.2	Apache Tomcat and java application logs . . . . .	117
8.7	Devices . . . . .	118
8.7.1	Cisco . . . . .	118
<b>9</b>	<b>Processing logs</b>	<b>119</b>
9.1	Parsing various formats . . . . .	119
9.1.1	W3C Extended Log File Format . . . . .	119
9.1.2	NCSA Common Log File Format . . . . .	120
9.1.3	NCSA Combined Log Format . . . . .	121

---

9.1.4	WebTrends Enhanced Log Format (WELF)	121
9.1.5	Field delimited formats (CSV)	121
9.1.6	JSON	121
9.1.7	XML	121
9.2	Parsing date and time strings	121
9.3	Filtering messages	123
9.3.1	Using drop()	123
9.3.2	Filtering through pm_filter	124
9.4	Dealing with multi-line messages	124
9.4.1	Using module variables	124
9.4.2	Using xm_multiline	125
9.5	Alerting, calling external scripts and programs	125
9.5.1	Sending all messages to an external program	125
9.5.2	Invoking a script or program for each message	126
9.5.3	Alerting	126
9.6	Rewriting and modifying messages	126
9.7	Message format conversion	126
9.8	Character set conversion	127
9.9	Discarding messages	127
9.10	Rate limiting	127
9.11	Buffering	128
9.12	Pattern matching and message classification	128
9.12.1	Regular expressions in the Exec directive	128
9.12.2	Using pm_pattern	129
9.13	Event correlation	129
9.14	Log rotation and retention	130
9.15	Explicit drop	131
<b>10</b>	<b>Forwarding and storing logs</b>	<b>133</b>
10.1	Data format of the output	133
10.2	Forwarding over the network	134
10.3	Sending to sockets and files	134
10.4	Storing logs in a database	134
<b>11</b>	<b>Tips and tricks</b>	<b>135</b>
11.1	Detecting a dead agent or log source	135

---

<b>12 Troubleshooting</b>	<b>137</b>
12.1 nxlog's internal logs . . . . .	137
12.1.1 Check the contents of the LogFile . . . . .	137
12.1.2 Injecting own logs into a route . . . . .	137
12.1.3 LogLevel . . . . .	137
12.1.4 Running in foreground . . . . .	138
12.1.5 Using log_info() in the Exec directive . . . . .	138
12.2 Common problems . . . . .	138
12.2.1 Missing logdata . . . . .	138
12.2.2 nxlog failed to start, cannot read configuration file . . . . .	139
12.2.3 nxlog.log is in use by another application and cannot be accessed . . . . .	139
12.2.4 Connection refused when trying to connect to im_tcp or im_ssl . . . . .	139
12.3 Debugging and dumping messages . . . . .	139

---

# Chapter 1

## Introduction

### 1.1 Overview

Today's IT infrastructure can be very demanding in terms of event logs. Hundreds of different devices, applications, appliances produce vast amounts of event log messages. These must be handled in real time, forwarded or stored in a central location after filtering, message classification, correlation and other typical log processing tasks. In most organizations these tasks are solved by connecting a dozen different scripts and programs which all have their custom format and configuration. nxlog is a high-performance multi-platform log management solution aimed at solving these tasks and doing it all in one place.

nxlog can work in a heterogeneous environment collecting event logs from thousands of different sources in many formats. nxlog can accept event logs from tcp, udp, file, database and various other sources in different formats such as syslog, windows event log etc. It can perform log rewrite, correlation, alerting, pattern matching, execute scheduled jobs, or even log rotation. It was designed to be able to fully utilize today's multi-core CPU systems. Its multi-threaded architecture enables input, log processing and output tasks to be executed in parallel. Using a high-performance I/O layer it is capable of handling thousands of simultaneous client connections and process log volumes above the 100.000 EPS range. nxlog tries hard to minimize losing log messages, it does not drop any unless instructed to. It can process input sources in a prioritized order, meaning that a higher priority source will be always processed before others. This can further help avoiding UDP message loss for example. In case of network congestion or other log transmission problems, nxlog can buffer messages on the disk or in memory. Using loadable modules it supports different input sources and log formats, not only limited to syslog but windows event log, audit logs or even custom binary application logs. It is possible to further extend its functionality by using custom loadable modules similarly to the Apache Web server. In addition to the online log processing mode it can be used to process logs in batch mode in an offline fashion. A powerful configuration language with an Apache style configuration file syntax enables it to rewrite logs, send alerts, execute external scripts or do virtually anything based on any criteria specified using the nxlog configuration language.

### 1.2 Features

#### 1.2.1 Multiplatform

nxlog is built to utilize the Apache Portable Runtime Library (libapr), the same solid foundation as the Apache Webserver is built on which enables nxlog to run on many different operating systems including different Unix flavors (Linux, HP-UX, Solaris, \*BSD etc). It compiles and runs as a native Windows application without requiring the CygWin libraries on Microsoft Windows platforms.

#### 1.2.2 Modular architecture

nxlog has a lightweight modular architecture, pluggable modules are available to provide different features and functions similarly to the Apache HTTP server. Log format parsers, transmission protocol handlers, database handlers and nxlog language extensions are such modules. A module is only loaded if it is necessary, this helps reduce memory as well. The core of nxlog only contains

---

code to handle files and sockets in addition to the configuration parser and the lightweight built-in language engine. All transport protocol handlers, format parsers (such as syslog) etc reside in modules. Modules have a common API, developers can easily write new modules and extend the functionality of nxlog.

### 1.2.3 Client-server mode

nxlog can act as a client and/or a server. It can collect logs from local files and the operating system then forward it to a remote server. It can accept connections and receive logs over the network then write these to a database or files or forward it further. It all depends how it is configured.

### 1.2.4 Log message sources and destinations

In addition to reading from and writing to log files, nxlog supports different protocols on the network and transport layer such as TCP, UDP, TLS/SSL and Unix Domain Socket. It can both read and write from such sources and can convert between them, read input from an UDP socket and send out in TCP for example.

Many database servers are supported (PostgreSQL, MySQL, Oracle, MsSQL, SqlLite, Sybase, etc) through database input and output modules so that log messages or data extracted from log messages can be stored or read from a database.

### 1.2.5 Importance of security

On unix systems nxlog can be instructed to run as a normal user by dropping its root privileges. Modules requiring special privileges (e.g. kernel, tcp port bind below 1024) use Linux capabilities and do not require it to be running as root.

To secure data and event logs, nxlog provides TLS/SSL transport so that messages cannot be intercepted and/or altered during transmission.

### 1.2.6 Scalable multi-threaded architecture

Using an event based architecture, tasks within nxlog are processed in a parallel fashion. Non-blocking I/O is used wherever possible and a worker thread pool takes care of handling ready to be processed log messages. Reading input, writing output and log processing (parsing, pattern matching, etc) are all handled in parallel. For example when single threaded syslog daemons block trying to write output to a file or database, UDP input will be lost. The multi-threaded architecture of nxlog not only avoids this problem but enables to fully utilize today's multi-core and multi-processor systems for maximum throughput.

### 1.2.7 High performance I/O

Traditional POSIX systems provide the select(2) and/or poll(2) system calls to monitor file descriptors, unfortunately using these methods is not scalable. Modern operating systems have some I/O readiness notification API to enable handling a large number of open files and network connections simultaneously. nxlog is capable of using these high-performance I/O readiness notification APIs and can handle thousands of simultaneous network connections. Together with its massively multi-threaded architecture, this enables nxlog to process log messages from thousands of simultaneous network connections above the hundred thousand event per second (EPS) range.

### 1.2.8 Message buffering

When write blocks on the sending side, because of a network trouble for example, nxlog will throttle back on the input side using flow control. In some cases it is preferable that the logs are continued to be read on the input side, to avoid dropping UDP syslog messages for example. There is a module available which makes it possible to buffer log messages to disk and/or memory. When the problems are solved and the system is back in order and can send out messages faster than being received, then the buffer is automatically emptied. Together with the nxlog language it is also possible to do conditional buffering based on different parameters (time or system load for example).

---

## 1.2.9 Prioritized processing

Not all log sources are always equally important. Some systems send critical logs which should be processed at a higher priority than others. nxlog supports assigning priorities to log routes, this ensures that higher priority log messages are dealt with (read, processed and written/sent out) first, only then are the messages with lower priorities handled. For example this can help avoiding the situation where a TCP input can overload the system leading to dropped incoming UDP syslog messages.

## 1.2.10 Avoiding lost messages

Built-in flow control ensures that nxlog does not drop log messages and you will not see any logs such as the following:

```
Dec 18 18:42:42 server syslog-ng[1234]: STATS: dropped 42
```

Though nxlog can be explicitly instructed to drop log messages depending on certain conditions in order to avoid a possible resource exhaustion or filter out unwanted messages.

UDP syslog is a typical case where a message can be lost due to the nature of the UDP protocol. If the kernel buffer becomes full because it is not read, the operating system will drop any further received UDP messages. If a log processing system is busy processing logs, reading from TCP and UDP and writing to database or disk, the kernel UDP buffer can fill quickly. Utilizing the above mentioned **parallel processing**, **buffering** and **I/O prioritization** features it is possible to greatly reduce losing UDP syslog messages. Of course using TCP can help avoiding message loss, unfortunately there are many archaic devices which only support UDP syslog.

## 1.2.11 Apache-style configuration syntax

nxlog uses Apache style configuration file syntax. This format is in use by many other popular system daemons and tools as it is easy to read and/or generate by both humans and scripts.

## 1.2.12 Built-in config language

A built-in configuration language enables administrators to create complex rules, format or rewrite messages or execute some action. Using this language it is possible to do virtually anything without the need to forward messages to an external script. Loadable modules can register their own procedures and functions to further extend the capabilities of the nxlog language.

Perl is a highly popular language in solving log processing tasks. The built-in nxlog language is very similar in syntax to Perl. In addition to the normal operations it supports polymorphic functions and procedures, regular expressions with captured substrings. It should be fairly trivial to write and understand by people experienced in Perl programming unlike some macro based configuration languages found in other solutions.

## 1.2.13 Scheduled tasks

nxlog has a built-in scheduler similar to cron, but with more advanced capabilities to specify the timings. Using this feature, administrators can automate tasks such as log rotation or system health check from within nxlog without having to use external scheduler tools. Each module can schedule any number of actions to be executed through the built-in nxlog language.

## 1.2.14 Log rotation

Log files can be rotated by size or time without the need of external log rotation tools. Log rotation can also be scheduled in order to guarantee timely file rotation.

The file input reader module supports external log-rotation scripts, it can detect when an input file was moved/renamed and will reopen its input. Similarly, the file output writer module can also monitor when the file being written to is rotated and will reopen its original output. This way it is possible to keep using external log rotation tools without the need to migrate to the built-in log rotation.

### 1.2.15 Different log message formats

Nxlog supports both the older legacy syslog format (RFC 3164) and the newer IETF Syslog standard (RFC 5424) and it can also produce syslog in the Snare Agent format. nxlog is not only a syslog daemon but can handle many other protocols and log file formats such as Windows Event Log, Checkpoint logs through LEA, OS audit logs, log message data in comma separated (CSV) format or delimited, GELF, JSON, XML or custom application logs. It can parse and generate most of these formats as well. It is only a matter of selecting the appropriate log format parser. Log format parsers are also provided by loadable modules, nxlog will only use parsers which are configured and required for its log processing. For example if the log processing task does not deal with any syslog data, then there is no need to load the syslog module at all.

Using regular expressions and string operation functions of the built-in nxlog language, any data can be extracted from log messages and can be converted to any format required. It is possible to configure nxlog in such a way that it reads log messages in one format then converts it internally to a different one and sends the output to another destination enabling on-the-fly log conversion. For example it is possible to convert Windows Event Log to syslog on a Windows host and send it to a central syslog server.

By using log format parser functions, nxlog can handle multi-line log messages (such as the Apache Tomcat log) or even custom binary formats. A special nxlog message format can preserve the parsed fields of log messages and transfer these across the network or store in files which alleviates the need to parse the messages again at the reception without losing any information.

### 1.2.16 Advanced message processing capabilities

In addition to the features provided by the above mentioned built-in nxlog language, using additional modules nxlog is capable to solve all tasks related to log message processing such as message classification, event correlation, pattern matching, message filtering, rewrite, conditional alerting etc.

### 1.2.17 Offline processing mode

Sometimes messages need to be processed in an offline fashion, convert log files to another format, filter out messages or load files into a database for log analysis purposes. nxlog can also work in an offline mode when it processes log messages until there is no more input and then exits, so it is possible to do batch processing tasks with it as well.

It is an important factor that in offline mode the time of the event and the current time are not the same and are not even close. Many log processing tools assume the event time to be the current time, thus making offline processing impossible. Due to network problems and buffering it is possible that log messages are not received instantly but with some delay. Making decisions based on event reception time instead of the timestamp provided in the message is a big mistake and can lead to false alarms in event correlation engines for example. By using the event time available in messages, nxlog can work properly in both offline and online mode with log messages. This is especially important to be able to do proper time based event correlation in real-time and in offline mode as well.

### 1.2.18 Character set and i18n support

Log messages can be emitted in different languages and character sets. It is also a common problem that the messages use different character sets even for one language. For example Microsoft Windows systems use the UTF-16 character set, other systems can create messages using the UTF-8 encoding. UTF-8 has become a standard on Unix systems, yet some legacy applications and system settings create log messages using another codepage, for example latin-2 or ISO-8859-2 in Eastern Europe or EUC-JP in Japan.

Comparing two strings in different character sets can likely fail. Also some database engines only support storing text data in one character set only, trying to insert text in a different character set can result in an error and data loss. It is a good practice to normalize logs to a common character set such as UTF-8 in order to overcome these problems.

nxlog supports explicit character set conversion from one character set to another. In addition it can also detect the character set of a string and convert it to a specific character set. Using charset autodetection, nxlog is capable of normalizing log messages which can contain strings in mixed character sets even without knowing the exact encoding of the source log message.

## Chapter 2

# Installation and quickstart

This chapter will guide to quickly get nxlog set up and running.

### 2.1 Microsoft Windows

**Install the MSI package** Run the nxlog installer using the MSI package, accept the license agreement and click finish.

**Edit nxlog.conf** The nxlog configuration file `nxlog.conf` is put under `C:\Program Files\nxlog\conf` or `C:\Program Files (x86)\nxlog\conf` on 64bit architectures. Using a text editor such as `notepad.exe`, open `nxlog.conf`.

**Verify the ROOT path in nxlog.conf** The windows installer uses the `C:\Program Files\nxlog` directory for the installation. On 64bit machines this is `C:\Program Files (x86)\nxlog`. We refer to this as the ROOT path. Please verify the `nxlog.conf` configuration file and use the appropriate ROOT path:

```
define ROOT C:\Program Files\nxlog
or
define ROOT C:\Program Files (x86)\nxlog
```

**Configure nxlog** The most common use-case for nxlog on windows is to collect logs from the EventLog subsystem and forward it over the network. Here is a simple configuration which reads the EventLog and forwards it over UDP in the SNARE agent format.

```
<Extension syslog>
  Module      xm_syslog
</Extension>

<Input internal>
  Module      im_internal
</Input>

<Input eventlog>
  Module      im_msvistalog
</Input>

<Output out>
  Module      om_udp
  Host        192.168.1.1
  Port        514
  Exec        to_syslog_snare();
</Output>

<Route 1>
```



```
Path          eventlog, internal => out
</Route>
```

There are endless configurations for Windows systems depending on what to collect and how to send or store. Please read the relevant chapters from this manual:

[Reading and receiving logs](#)

[Processing logs](#)

[Forwarding and storing logs](#)

**Start nxlog** nxlog can be started using the following methods:

Start the Service Manager, find 'nxlog' in the list. Select it and start the service.

Double-click on nxlog.exe.

**Check the logs** The default configuration instructs nxlog to write its own logs to the file located at C:\Program Files\nxlog\data\nxlog.log or C:\Program Files (x86)\nxlog\data\nxlog.log. Open it with notepad.exe and check for errors. Note that some text editors (such as wordpad) need exclusive locking and will refuse to open the log file while nxlog is running.

## 2.2 GNU/Linux

### 2.2.1 Installing from DEB packages (Debian, Ubuntu)

**Install the dependencies first** To list the dependencies, use the following command:

```
dpkg-deb -f nxlog_1.4.581_amd64.deb Depends
```

Then make sure all listed dependencies are installed. Alternatively you can run **apt-get install -f** after trying to install the package with dpkg and getting an error due to the missing dependencies.

**Install the deb package** To install the deb package, issue the following command as root:

```
dpkg -i nxlog_1.4.581_amd64.deb
```

### 2.2.2 Installing from RPM packages (CentOS, RedHat)

Install the rpm package with the following command:

```
rpm -ivh nxlog-1.4.581-1.x86_64.rpm
```

### 2.2.3 Configuring nxlog on GNU/Linux

After the package is installed check and edit the configuration file located at /etc/nxlog.conf. It contains an example configuration which you will likely want to modify to suit your needs. Please read the relevant chapters from this manual on how to configure nxlog:

[Reading and receiving logs](#)

[Processing logs](#)

[Forwarding and storing logs](#)

## Chapter 3

# Architecture and concepts

### 3.1 History

For a few years we have been using a modified version of msyslog. It is also capable of using plugins for different inputs and outputs. Unfortunately, like many other syslog implementations, it was based on the BSD syslog with a single threaded architecture. Since it was a syslog daemon, everything had to be converted to syslog. We soon realized that something better is needed with the **features** required by a modern logging solution.

We started looking for other solutions. There were a few possible alternatives to msyslog with some nice features (e.g. rsyslog, syslog-ng, etc), but none of them qualified. Most of them were still single threaded, syslog oriented without native support for MS Windows, in addition to awkward configuration syntax, ugly source-code and so on. So I decided that it would be easier for us on the long term to design and write nxlog from scratch instead of hacking something else. Thus nxlog was born in 2009 and has been a closed source product heavily used in several production deployments since. The source code of NXLOG Community Edition was released under the GPL/LGPL in November 2011.

### 3.2 Concepts

Most log processing solutions are built around the same concept. The input is read from a source, then the log messages are processed. Finally output is written or sent to a sink in other terminology.

When an event occurs in an application or a device, depending on its configuration a log message is emitted. This is usually referred to as an "event log" or "log message". These log messages can have different formats and can be transmitted over different protocols depending on the actual implementation.

There is one thing common in all event log messages. All contain important data such as user names, IP addresses, application names, etc. This way an event can be represented as a list of key-value pairs which we call a "field". The name of the field is the key and the field data is the value. In another terminology this meta-data is sometimes referred to as event property or message tag. The following example illustrates a syslog message:

```
<30>Nov 21 11:40:27 log4ensics sshd[26459]: Accepted publickey for log4ensics from 192.168.1.1 port 41193 ssh2
```

The fields extracted from this message are as follows:

AuthMethod	publickey
SourceIPAddress	192.168.1.1
AccountName	log4ensics
SyslogFacility	DAEMON
SyslogSeverity	INFO
Severity	INFO
EventTime	2009-11-21 11:40:27.0
Hostname	log4ensics

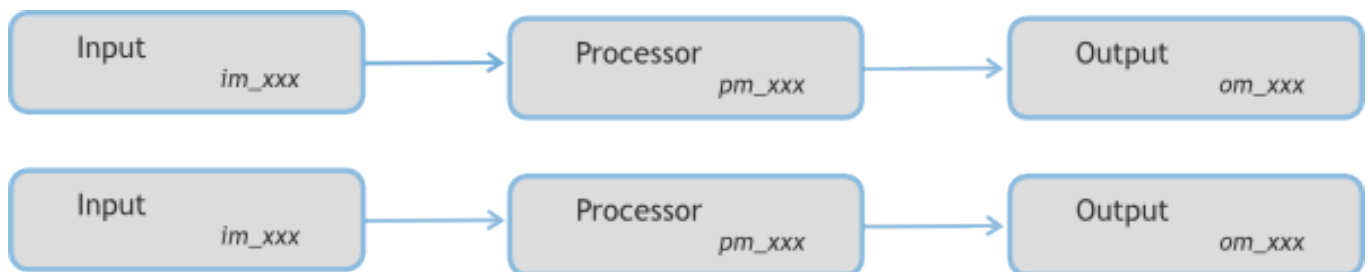
ProcessID	26459
SourceName	sshd
Message	Accepted publickey for log4ensics from 192.168.1.1 port 41193 ssh2

nxlog will try to use the **Common Event Expression standard** for the field names once the standard is stable.

nxlog has a special field, \$raw\_event. This field is handled by the transport (UDP, TCP, File, etc) modules to read input into and write output from it. This field is also used later to parse the log message into further fields by various functions, procedures and modules.

### 3.3 Architecture

By utilizing **loadable modules**, the plugin architecture of nxlog allows it to read data from any kind of input, parse and convert the format of the messages and then send it to any kind of output. Different input, processor and output modules can be used at the same time to cover all the requirements of the logging environment. The following figure illustrates the flow of log messages using this architecture.



*Architecture*

The core of nxlog is responsible for parsing the configuration file, monitoring files and sockets, and managing internal events. It has an event based architecture, all modules can dispatch events to the core. The nxlog core will take care of the event and will optionally pass it to a module for processing. nxlog is a multi-threaded application, the main thread is responsible for monitoring files and sockets. These are added to the core by the different **input** and **output** modules. There is a dedicated thread handling internal events. It sleeps until the next event is to be processed then wakes up and dispatches the event to a worker thread. nxlog implements a worker thread-pool model. Worker threads receive an event which must be processed immediately. This way the nxlog core can centrally control all events and the order of their execution making **prioritized processing** possible. Modules which handle sockets or files are written to use non-blocking I/O in order to ensure that the worker threads never block. The files and sockets monitored by the main thread also dispatch events which are then delegated to the workers. Each event belonging to the same module is executed in sequential order, not concurrently. This ensures that message order is kept and gives a great benefit of not having to deal with concurrency issues in modules. Yet the modules (worker threads) run concurrently, thus the global log processing flow is greatly parallelized.

When an input module receives data, it creates an internal representation of the log message which is basically a structure containing the raw event data and any optional fields. This log message is then pushed to the queue of the next module in the route and an internal event is generated to signal the availability of the data. The next module after the input module in a route can be either a processor module or an output module. Actually an input or output module can also process data through built in code or using the **nxlog language execution** framework. The only difference is that processor modules are run in another worker thread, thus parallelizing log processing even more. Considering that processor modules can also be chained, this can efficiently distribute work among multiple CPUs or CPU cores in the system.

## Chapter 4

# Configuration

nxlog uses **Apache-style** configuration files. The configuration file is loaded from its default location or it can be explicitly specified with the `-c` command line argument.

The config file is made up of blocks and directives. Blocks are similar to xml tags containing multiple directives. Directive names are case insensitive but arguments are not always. A directive and its argument must be specified on the same line. Values spanning multiple lines must have the newline escaped with the backslash `"\"`. A typical case for this is the **Exec** directive. Blank lines are ignored.

Lines starting with the hashmark `"#"` are comments and are ignored.

The configuration file can be logically divided into three parts: **global parameters**, **module definitions and their configuration** and **routes** which link the modules together according to the data flow required.

### 4.1 File inclusion

Using the `'include'` directive it is possible to specify a file which will be included in the current config file. Special care must be taken when specifying files with relative filenames. The **SpoolDir** directive will only take effect after the configuration was parsed, so files specified with the `'include'` directive are relative to the working directory where nxlog was started from.

---

**Example 4.1** File inclusion example

---

```
include modules/module1.conf
```

---

### 4.2 Constant and macro definitions

Defines are useful if there are many instances in the code where the same value must be used, directory or host names are typical cases. In such cases the value can be configured with a single definition. This can be used to not only define constants but any string like code snippets or parser rules.

An nxlog define works similarly as in C where the preprocessor substitutes the value in places where the macro is used, i.e. the nxlog configuration parser will first replace all occurrences of the defined name with its value, only after this substitution will the configuration check occur.

---

---

**Example 4.2** Example for using defines

---

```
define BASEDIR /var/log
define IGNORE_DEBUG if $raw_event =~ /debug/ drop();

<Input messages>
  Module      im_file
  File        '%BASEDIR%/messages'
</Input>

<Input proftpd>
  Module      im_file
  File        '%BASEDIR%/proftpd.log'
  Exec        %IGNORE_DEBUG%
</Input>
```

---

The following example shows an incorrect use of the define directive. After substitution the `drop()` procedure will be always executed, only the warning message is emitted conditionally.

---

**Example 4.3** Incorrect use of a define

---

```
define ACTION log_warning("dropping message"); drop();

<Input messages>
  Module      im_file
  File        '/var/log/messages'
  Exec        if $raw_event =~ /dropme/ %ACTION%
</Input>
```

---

To avoid this problem, the defined action should be one code block, i.e. it should be enclosed within curly braces:

```
define ACTION { log_warning("dropping message"); drop(); }
```

---

## 4.3 Global directives

**ModuleDir** By default the nxlog binaries have a compiled-in value for the directory to search for loadable modules. This can be overridden with this directive. The module directory contains subdirectories for each module type (extension, input, output, processor) and the module binaries are located in those.

**PidFile** Under Unix operating systems nxlog writes a pid file as other system daemons do. The default value can be overridden with this directive in case multiple daemon instances need to be running. This directive has no effect on MS Windows or with the **nxlog-processor**.

**LogFile** nxlog will write its internal log to this file. If this directive is not specified, self logging is disabled. Not that the **im\_internal** module can be also used to direct internal log messages to files or different output destinations, but this does not support loglevel below 'info'. This LogFile directive is especially useful for debugging.

**LogLevel** This directive has five possible values: CRITICAL, ERROR, WARNING, INFO, DEBUG. It will set the logging level used for LogFile and the standard output if nxlog is started in the foreground. By default the LogLevel is INFO.

**SuppressRepeatingLogs** Under some circumstances it is possible for nxlog to generate an extreme amount of internal logs consisting of the same message due to a misconfiguration or software bug. This can lead to an extreme usage of disk space by **LogFile** and nxlog can quickly fill up the disk. With this directive nxlog will write at most 2 lines per second if the same message is generated successively by emitting 'last message repeated x times' will suppress these messages. This directive takes a boolean value (TRUE or FALSE). If this directive is not specified in the config file, it defaults to TRUE, i.e. repeating message suppression is enabled.

---

**NoCache** Some modules save data to a cache file which is persisted across a shutdown/restart. Modules such as **im\_file** will save the file position in order to be able to continue reading from the same position where it left off after a restart. This caching mechanism can be explicitly turned off with this directive, this is mostly useful with the **nxlog-processor** in offline mode. This directive takes a boolean value (TRUE or FALSE). If this directive is not specified in the config file, it defaults to FALSE, i.e. caching is enabled.

**CacheDir** This directive specifies a directory where the cache file called `configcache.dat` should be written to. This directive has a compiled-in value which is used by default.

**User** nxlog will drop to user specified with this directive. This is useful if nxlog needs privileged access to some system resources such as kernel messages or port bind below 1024. On Linux systems it will use capabilities to be able to access these resources. In this case nxlog must be started as root. The user can be specified by name or by numeric id. This directive has no effect on MS Windows or with the **nxlog-processor**.

**Group** Similar to **User**, nxlog will set the group ID to be running under. The group can be specified by name or by numeric id. This directive has no effect on MS Windows or with the **nxlog-processor**.

**RootDir** nxlog will set its root directory to the value specified with this directive. If **SpoolDir** is also set, this will be relative to the value of RootDir, i.e. `chroot()` is called first. This directive has no effect on MS Windows or with the **nxlog-processor**.

**SpoolDir** nxlog will change its working directory to the value specified with this directive. This is useful with files created through relative filenames, e.g. with **om\_file** and in case of core dumps. This directive has no effect with the **nxlog-processor**.

**Threads** This optional directive specifies the number of worker threads to use. The number of the worker threads is calculated and set to an optimal value if this directive is not defined. You should not set this unless you know what you are doing.

**NoFreeOnExit** This directive has only a debugging purpose. When set to TRUE, nxlog will not free module resources on exit. Otherwise `valgrind` is unable to show proper stack trace locations in module function calls. The default value is FALSE if not specified.

**IgnoreErrors** If set to FALSE, nxlog will stop if it encounters a problem with the configuration file such as an invalid module directive or if there are other problems which would prevent all modules functioning correctly. If set to TRUE, nxlog will start after logging the problem. The default value is TRUE if the directive is not specified.

**Panic** A panic condition is a critical state which usually indicates a bug. Assertions are used in nxlog code for checking conditions where the code will not work unless the asserted condition is satisfied. Failing assertions result in a panic and these also suggest a bug in the code. A typical case is checking for NULL pointers before pointer dereference. Assertions have also a security value. This directive can take three different values: HARD, SOFT or OFF. HARD will cause an abort in case the assertion fails. This is how most C based programs work. SOFT will cause an exception to be thrown at the place of the panic/assertion. In case of NULL pointer checks this is identical to a `NullPointerException` in Java. It is possible that nxlog can recover from exceptions and can continue to process log messages, or at least the other modules can. In case of assertion failure the location and the condition is printed at CRITICAL loglevel in HARD mode and ERROR loglevel in SOFT mode. If Panic is set to OFF, the failing condition is only printed in the logs but the execution will continue on the normal code path. Most of the time this will result in a segmentation fault or other undefined behavior, though there can be a case where turning off a buggy assertion or panic lurking somewhere in the code will solve the problems caused by it in HARD/SOFT mode. The default value for Panic is SOFT.

## 4.4 Modules

nxlog will only load modules which are used and specified in the configuration file. The followin is a skeleton config block for an input module:

```
<Input instancename>
  Module      im_module
  ...
</Input>
```

The instance name must be unique, can contain only the characters `[a-zA-Z0-9_-]`. The instance name is referenced from the **route** definition as well as the **Processors** directive. Four types of **modules** exist in nxlog, these must be decalred with the Input, Processor, Output and Extension tags.

## 4.4.1 Common module directives

The following directives are common in all modules.

### 4.4.1.1 Module

This directive is mandatory as it specifies which loadable binary should be loaded. The module binary has a .so extension on Unix and a .dll on MS Windows platforms and resides under the **ModuleDir** location. All module binary names are prefixed with either im\_, pm\_, om\_, xm\_. These stand for input module, processor module, output module and extension module.

It is possible that multiple instances use the same loadable binary. In this case the binary is only loaded once but instantiated multiple times. Different module instances may have different configuration.

### 4.4.1.2 Schedule

The Schedule block can be used to execute periodic jobs such as log rotation or any other task. Scheduled jobs have the same priority as the module. The schedule block has the following directives:

**When** This directive takes a value similar to a crontab entry which consists of five space separated definitions for minute, hour, day, month and weekday. See the crontab(5) manual for the field definitions. It supports lists as comma separated values and/or ranges. Step values are also supported with the slash. Month and week days are not supported, these must be defined with numeric values. The following extensions are also supported:

@yearly	Run once a year, "0 0 1 1 *".
@annually	(same as @yearly)
@monthly	Run once a month, "0 0 1 * *".
@weekly	Run once a week, "0 0 * * 0".
@daily	Run once a day, "0 0 * * *".
@midnight	(same as @daily)
@hourly	Run once an hour, "0 * * * *".

**Every** In addition to the crontab format it is possible to schedule execution at periodic intervals. With the crontab format it is not possible to run jobs every five days for example, this directive enables it in a simple way. It takes an integer value with an optional unit. The unit can be one of the following: sec, min, hour, day, week. If the unit is not specified, the value is assumed to be in seconds.

**First** This directive sets the first execution time. If the value is in the past, the next execution time is calculated as if nxlog has been running since and jobs will not be run to make up the missed events in the past. The directive takes a **datetime** literal value.

**Exec** The Exec directive takes one or more nxlog **statement**. This is the code which is actually being scheduled. Multiple Exec directives can be specified within one Schedule block, so this behaves the same as the **Exec** directive of the modules. See that for more details. Note that it is not possible to use **fields** in statements here because execution is not triggered by log messages.

---

**Example 4.4** Two scheduled jobs in the context of the im\_tcp module

---

```
<Input in>
  Module im_tcp
  Port 2345

  <Schedule>
  Every 1 sec
  First 2010-12-17 00:19:06
  Exec log_info("scheduled execution at " + now());
  </Schedule>

  <Schedule>
  When 1 */2 2-4 * *
  Exec log_info("scheduled execution at " + now());
  </Schedule>
</Input>

<Output out>
  Module om_file
  File "tmp/output"
</Output>

<Route 1>
  Path in => out
</Route>
```

---

**4.4.1.3 Exec**

The Exec directive contains **statements** in the **nxlog language** which are executed when a module receives a log message. This directive is available in all **input**, **processor** and **output** modules. It is not available in **extension** modules because these don't handle log messages directly. More than one Exec may be specified. In this case these are executed in the order of appearance. Due to the limitations of the apache configuration file format, each directive must be one line unless it contains a trailing backslash "\ " character.

---

**Example 4.5** Exec statement spanning multiple lines

---

```
Exec if $Message =~ /something interesting/ \
      log_info("found something interesting"); \
      else \
      log_debug("found nothing interesting");
```

---

**Example 4.6** Equivalent use of statements in Exec

---

```
Exec log_info("first"); \
      log_info("second");
# The above is the same as the following:
Exec log_info("first");
Exec log_info("second");
```

---

**Note**

You cannot split the lines in the first example as the exec directive must contain a full statement. It is only possible to split the Exec arguments if it contains multiple statements as in the second example above.

---



#### 4.4.1.4 Processors

The 'Processors' directive has been obsoleted and is no longer available.

#### 4.4.1.5 InputType

This directive specifies the name of the registered input reader function to be used for parsing raw events from input data. Names are treated case insensitively.

This is a common directive only for stream oriented input modules: **im\_file**, **im\_exec**, **im\_ssl**, **im\_tcp**, **im\_udp**, **im\_uds**. Note that **im\_udp** may only work properly if log messages do not span multiple packets and log messages are within the UDP message size limit. Otherwise the loss of a packet may lead to parse errors.

These modules work by filling an input buffer with data read from the source. If the read operation was successful (i.e. there was data coming from the source), the module calls the specified callback function. If this is not explicitly specified, it will use the module default.

Modules may provide custom input reader functions. Once these are registered into the nxlog core, the modules listed above will be capable of using these. This makes it easier to implement custom protocols because these can be developed without the need of taking care about the transport layer.

The following input reader functions are provided by the nxlog core:

**LineBased** The input is assumed to contain log messages separated by newlines. Thus if an LF (\n) or CRLF (\r\n) is found, the function considers that it has reached the end of the log message.

**Dgram** Once the buffer is filled with data, it is considered to be one log message. This is the default for the **im\_udp** input module, since UDP syslog messages arrive in separate packets.

**Binary** The input is parsed in the nxlog binary format which is capable of preserving parsed fields of the log messages. The **LineBased** reader is capable of automatically detecting log messages in the Binary nxlog format, it is only recommended to configure InputType to Binary if no compatibility with other logging software is required.

---

#### Example 4.7 TCP input assuming nxlog format

---

```
<Input in>
  Module im_tcp
  Port 2345
  InputType Binary
</Input>

<Output out>
  Module om_file
  File "tmp/output"
</Output>

<Route 1>
  Path in => out
</Route>
```

---

#### 4.4.1.6 OutputType

This directive specifies the name of the registered output writer function to be used for formatting raw events when sending to different destinations. Names are treated case insensitively.

This is a common directive only for stream oriented output modules: **om\_file**, **om\_exec**, **om\_ssl**, **om\_tcp**, **om\_udp**, **om\_uds**.

---

These modules work by filling the output buffer with data to be written to the destination. The specified callback function is called before the write operation. If this is not explicitly specified, it will use the module default.

Modules may provide custom output formatter functions. Once these are registered into the nxlog core, the modules listed above will be capable of using these. This makes it easier to implement custom protocols because these can be developed without the need to take care about the transport layer.

The following output writer functions are provided by the nxlog core:

**LineBased** The output will contain log messages separated by newlines (CRLF).

**Dgram** Once the buffer is filled with data, it is considered to be one log message. This is the default for the **om\_udp** output module, since UDP syslog messages are sent in separate packets.

**Binary** The output is written in the nxlog binary format which is capable of preserving parsed fields of the log messages.

---

**Example 4.8** TCP output sending messages in nxlog format

---

```
<Input in>
  Module  im_file
  File    "tmp/input"
</Input>

<Output out>
  Module  om_tcp
  Port    2345
  Host     localhost
  OutputType Binary
</Output>

<Route 1>
  Path    in => out
</Route>
```

---

## 4.5 Routes

Routes define the flow and processing order of the log messages. The route must have a name and a Path. The name is specified similarly to the instance name in a module block.

---

**Example 4.9** Route block

---

```
<Route example>
  Path    in1, in2 => proc => out1, out2
</Route>
```

---

### 4.5.1 Priority

This directive is optional. It takes an integer value as a parameter, its value must be in the range of 1-100. It defaults to 10 if it is not explicitly specified. Log messages in routes with a lower priority value will be processed before others.

Actually this value is assigned to each module part of the route. The internal events of the modules are processed in priority order by the nxlog engine, thus modules of a route with a lower priority value (higher priority) will process log messages first.

This directive can be especially usefull to minimize syslog UDP message loss for example.

---

---

**Example 4.10** Prioritized processing

---

```
<Input tcpin>
  Module im_tcp
  Host localhost
  Port 514
</Input>

<Input udpin>
  Module im_udp
  Host localhost
  Port 514
</Input>

<Output tcpfile>
  Module om_file
  File "/var/log/tcp.log"
</Output>

<Output udpfile>
  Module om_file
  File "/var/log/udp.log"
</Output>

<Route udp>
  Priority 1
  Path udpin => udpfile
</Route>

<Route tcp>
  Priority 2
  Path tcpin => tcpfile
</Route>
```

---

### 4.5.2 Path

The Path directive is where the data flow is defined. First the instance name of input modules are specified. If more than one input reads log messages which feed data into the route, then these must be separated by a comma. Input modules are followed by an arrow "=>" sign. Either processor modules or output modules follow. Processor modules must be separated by arrows, not commas, because they receive log messages in order, unlike input and output modules which work in parallel. Output modules are separated by commas. The syntax for the PATH directive is illustrated by the following:

```
Path INPUT1[, INPUT2...] => [PROCESSOR1 [=> PROCESSOR2...] =>] OUTPUT1[, OUTPUT2...]
```

The Path must contain at least an input and an output module. The following example shows different routes.

---

---

**Example 4.11** Different routes

---

```
<Input in1>
  Module im_null
</Input>

<Input in2>
  Module im_null
</Input>

<Processor p1>
  Module pm_null
</Processor>

<Processor p2>
  Module pm_null
</Processor>

<Output out1>
  Module om_null
</Output>

<Output out2>
  Module om_null
</Output>

<Route 1>
  # no processor modules
  Path in1 => out1
</Route>

<Route 2>
  # one processor module
  Path in1 => p1 => out1
</Route>

<Route 3>
  # multiple modules
  Path in1, in2 => p1 => p2 => out1, out2
</Route>
```

---

## Chapter 5

# Language

The nxlog core contains support for using a built-in interpreted language. This language can be used to make complex decisions or build expressions in the nxlog configuration file. The code written in the nxlog language is similar to Perl which is a common tool for developers and administrators to solve log processing tasks. When nxlog starts and reads its configuration file, directives containing nxlog language code are parsed and compiled into a pseudo-code. If a syntax error is found, nxlog will print the error. The pseudo-code is then evaluated at run-time, similarly to other interpreted languages.

The nxlog language can be used in two ways. Some module directives (e.g. file names) require a value, for these **expressions** can be used if supported by the module. Other directives such as **Exec** take a **statement or statements** as argument.

In addition to the built-in **functions** and **procedures** provided by the nxlog core, modules can register additional functions and procedures. This enables developers to extend the language through loadable modules so that additional processing features can be executed such as message formatters and parsers or data lookup functions.

Due to the simplicity of the language there is no error handling (except for function return values) available to the administrator. If an error occurs during the execution of the nxlog pseudo-code, usually the error is printed in the nxlog logs. If an error occurs during log message processing it is also possible for the message to be dropped. In case sophisticated error handling or more complex processing is a requirement, the message processing can be implemented in an external script or program, in a dedicated nxlog module or in perl via the **xm\_perl** module.

### 5.1 Types

The nxlog language is a typed language, this allows stricter syntax checking when parsing the configuration while trying to enforce type-safety. Though **fields** and some functions can return values with a type which can only be determined at run-time. The language provides only simple types, complex types such as arrays and hashes (associative arrays) are not supported. See **xm\_perl** if you require such complex processing rules. The language also supports the **undefined** value similarly to Perl. The following types are provided by the nxlog language:

**Unknown** This is a special type for values where the type cannot be determined at compile time and for values which are uninitialized. The **undef literal** and **fields** without a value have also an unknown type. The unknown type can be also thought of as 'any' in case of function and procedure api declarations.

**Boolean** A boolean value which is either TRUE, FALSE or undefined. Note that an undefined boolean is not the same as a FALSE value.

**Integer** An integer which can hold a signed 64 bit value in addition to the undefined value. Floating point values are not supported.

**String** A string is an array of characters in any character set. The **binary** type should be used for values where the NUL byte can also occur. An undefined string is not the same as an empty string. Strings have a limited length to prevent resource exhaustion problems, this is a compile-time value currently set to 1M.

**Datetime** A datetime holds a microsecond value elapsed since the Epoch and is always stored in UTC/GMT.

**IPv4 Address** Stores a dotted quad IPv4 address in an internal format (integer).

**IPv6 Address** Stores an IPv6 address in an internal format.

**Regular expression** A regular expression can only be used with the `==` or `!~` operators.

**Binary** This type can hold an array of bytes.

**Variadic arguments** This is a special type only used in function and procedure api declarations to indicate variadic arguments.

## 5.2 Expressions

Expressions are a subset of the nxlog language. Some module directives take an expression as a parameter which is then dynamically evaluated at run-time to a value. Expressions can also be used in statements.

The following language elements are expressions: **literals**, **fields**, **binary and unary operations**, **functions**. In addition, brackets can be used around expressions as shown in the example below. Brackets can also help in writing more readable code.

---

### Example 5.1 Using brackets around expressions

---

```
if 1 + 1 == (1 + 1) log_info("2");
if (1 + 1) == (1 + 1) log_info("2");
if ((1 + 1) == (1 + 1)) log_info("2");
```

---

### 5.2.1 Literals

A literal is a representation of a fixed value. A literal is an **expression**.

**Undef** The undef literal has an **unknown** type. It can be also used in an **assignment** to unset a value of a **field**, for example:

---

### Example 5.2 Unsetting a value of a field

---

```
$ProcessID = undef;
```

---

**Boolean** A boolean literal is either TRUE or FALSE. It is case insensitive, so True, False, true, false are also valid.

**Integer** An integer starts with a minus "-" sign if it is negative. The "0X" or "0x" prepended modifier means a hexadecimal notation. The "K", "M" and "G" modifiers are also supported which can be appended to mean Kilo (1024), Mega (1024<sup>2</sup>) and Giga (1024<sup>3</sup>).

---

### Example 5.3 Setting an integer value

---

```
$Limit = 42M;
```

---

**String** String literals are quoted characters using either single or double quotes. String literals specified with double quotes can contain the following escape sequences.

- `\\` The backslash (\) character.
  - `\"` The double quote (") character.
  - `\n` Line feed (LF).
  - `\r` Carriage return (CR).
  - `\t` Horizontal tab.
  - `\b` Audible bell.
-

**\xXX** A single byte in the form of a two digit hexadecimal number. For example the line-feed character can also be expressed as `\x0A`.

---

**Example 5.4** Setting a string value

---

```
$Message = "Test message";
```

---

**Regular expression** Regular expressions must be quoted with slashes as in Perl. Captured substrings are accessible through a numeric reference such as `$1`. The full subject string is placed into `$0`.

---

**Example 5.5** A regular expression match operation

---

```
if $Message =~ /^Test (\S+)/ log_info("captured: " + $1);
```

---

**Datetime** The datetime literal is an unquoted representation of a time value expressing local time in the format of YYYY-MM-DD hh:mm:ss

---

**Example 5.6** Setting a datetime value

---

```
$EventTime = 2000-01-02 03:04:05;
```

---

**IPv4 Address** An IPv4 literal value is expressed in dotted quad notation such as `192.168.1.1`.

**IPv6 Address** An IPv6 literal value is expressed by 8 groups of 16-bit hexadecimal values separated by colons (:) such as `2001:0db8:85a3:0000:0000:8a2e:0370:7334`.

## 5.2.2 Fields

A log message can be broken up into fields by parsers or is already emitted as a list of fields as discussed [earlier](#). The field has a name and in the nxlog language it is represented with the dollar "\$" sign prepended to the name of the field, similarly to Perl's scalar variables. The name of the field is allowed to have the following characters:

```
[[:alpha:]] [[:alnum:]] \. _ *
```

A field which does not exist has an **unknown** type. A field is an **expression** which evaluates to a value. Fields are only available in an evaluation context which is triggered by a log message. For example using a value of a field in the **Exec directive of a schedule block** will result in a run-time error because this scheduled execution is not triggered by a log message. Fields are passed along the route and are available in each successive module in the chain. Eventually the output module is responsible for writing these. Stream oriented modules emit the data contained in `$raw_event` unless **OutputType** is set to something else (i.e. Binary).

## 5.2.3 Operations

Similarly to other programming languages and especially Perl, the nxlog language has unary and binary operations which are **expressions** and evaluate to a value.

### 5.2.3.1 Unary operations

Unary operations work with a single operand. Currently the following unary operations are available. It is possible to use brackets around the operand to which makes it look like a function call as in [this example](#).

---

**not** The 'not' operator expects a boolean value. It will evaluate to undef if the value is undefined. If it receives an unknown value which evaluates to a non-boolean, it will result in a run-time execution error.

---

**Example 5.7** Typical use of the 'not' operand

---

```
if not $success log_error("failure");
```

---

- The unary negation operator before an integer is very similar to a negative integer, except that two or more minus "-" signs are not valid for an integer literal.

---

**Example 5.8** Unary negation

---

```
if - -1 != 1 log_error("this should never be printed");
```

---

**defined** The defined operation will evaluate to TRUE if the operand is defined, otherwise it is FALSE.

---

**Example 5.9** Use of the unary 'defined' operation

---

```
if defined 1 log_info("1");  
if defined(2) log_info("2");  
if defined undef log_info("never printed");
```

---

### 5.2.3.2 Binary operations

Binary operations work with two operands and evaluate to a value. The type of the evaluated value depends on the type of the operands. Execution might result in a run-time error if the type of the operands are unknown at compile time and evaluate to types which are incompatible with the binary operation. The operations are described with the following syntax:

```
TYPE_OF_LEFT_OPERAND BINARY_OPERATION TYPE_OF_RIGHT_OPERAND = TYPE_OF_EVALUATED_VALUE
```

Below is a list of currently supported binary operations.

**=~** This is the regular expression match operation as in Perl. It takes a string and a regexp operand and evaluates to a boolean value which is TRUE if the regular expression matches the subject string. Captured substrings are accessible through a numeric reference such as \$1. The full subject string is placed into \$0.

**string =~ regexp = boolean**

**regexp =~ string = boolean**

---

**Example 5.10** Regular expression based string matching

---

```
if $Message =~ /^Test message/ log_info("matched");
```

---

**!~** This is the opposite of **=~**, the expression will evaluate to TRUE if the regular expression does not match on the subject string. It can be also written as **not LEFT\_OPERAND =~ RIGHT\_OPERAND**

**string !~ regexp = boolean**

**regexp !~ string = boolean**

---

**Example 5.11** Regular expression based string matching

---

```
if $Message !~ /^Test message/ log_info("didn't match");
```

---



**==** This operator compares two values for equality. Comparing a defined value with an undefined results in **undef**!

```
undef == undef = TRUE  
string == string = boolean  
integer == integer = boolean  
boolean == boolean = boolean  
datetime == datetime = boolean
```

---

**Example 5.12** Comparing integers

---

```
if $SeverityValue == 1 log_info("severity is one");
```

---

**!=** This operator compares two values for inequality. Comparing a defined value with an undefined results in **undef**!

```
undef != undef = FALSE  
string != string = boolean  
integer != integer = boolean  
boolean != boolean = boolean  
datetime != datetime = boolean
```

---

**Example 5.13** Comparing for inequality

---

```
if $SeverityValue != 1 log_info("severity is not one");
```

---

**<** This operation will evaluate to TRUE if the left operand is less than the operand on the right, FALSE otherwise. Comparing a defined value with an undefined results in **undef**!

```
integer < integer = boolean  
datetime < datetime = boolean
```

---

**Example 5.14** Less

---

```
if $SeverityValue < 1 log_info("severity is less than one");
```

---

**<=** This operation will evaluate to TRUE if the left operand is less than or equal to the operand on the right, FALSE otherwise. Comparing a defined value with an undefined results in **undef**!

```
integer <= integer = boolean  
datetime <= datetime = boolean
```

---

**Example 5.15** Less or equal

---

```
if $SeverityValue < 1 log_info("severity is less than or equal to one");
```

---

**>**

```
integer > integer = boolean  
datetime > datetime = boolean
```

---

---

### Example 5.16 Greater

---

```
if $SeverityValue > 1 log_info("severity is greater than one");
```

---

**>=**

**integer >= integer = boolean**

**datetime >= datetime = boolean**

---

### Example 5.17 Greater or equal

---

```
if $SeverityValue >= 1 log_info("severity is greater than or equal to one");
```

---

**and** This is the boolean 'and' operation which evaluates to TRUE if and only if both operands are TRUE. The operation will evaluate to **undef** if either operand is undefined.

**boolean and boolean = boolean**

---

### Example 5.18 And operation

---

```
if $SeverityValue == 1 and $FacilityValue == 2 log_info("1 and 2");
```

---

**or** This is the boolean 'and' operation which evaluates to TRUE if either operand is TRUE. The operation will evaluate to **undef** if both operands are undefined.

**boolean or boolean = boolean**

---

### Example 5.19 Or

---

```
if $SeverityValue == 1 or $SeverityValue == 2 log_info("1 or 2");
```

---

**+** This operation will result in an integer if both operands are integers. If either operand is a string, the result will be a string where non-string typed values are converted to a string. In this case it acts as a concatenation operator (which is the dot "." operator in Perl). Adding an undefined value to a non-string will result in undef.

**integer + integer = integer**

**string + undef = string**

**undef + string = string**

**undef + undef = undef**

**string + string = string** Concatenate two strings.

**datetime + integer = datetime** Add the number of seconds in the right value to the datetime stored in the left value.

**integer + datetime = datetime** Add the number of seconds in the left value to the datetime stored in the right value.

---

### Example 5.20 Concatenation

---

```
if 1 + "a" == "1a" log_info("this will be printed");
```

---

**-** Subtraction. The result will be undef if either operand is undefined.

**integer - integer = integer** Subtract two integers.

---

**datetime - datetime = integer** Subtract two datetime types. The result is the difference between two expressed in microseconds.

**datetime - integer = datetime** Subtract the number of seconds from the datetime stored in the left value.

---

#### Example 5.21 Subtraction

```
if 4 - 1 == 3 log_info("four minus one is three");
```

---

\* Multiply an integer with another. The result will be undef if either operand is undefined.

**integer \* integer = integer**

---

#### Example 5.22 Multiplication

```
if 4 * 2 == 8 log_info("four times two is eight");
```

---

/ Divide an integer with another. The result will be undef if either operand is undefined. Since the result is an integer, fractional parts are lost.

**integer / integer = integer**

---

#### Example 5.23 Division

```
if 9 / 4 == 2 log_info("9 divided by 4 is 2");
```

---

% This is the modulo operation. Divides an integer with another and returns the remainder. The result will be undef if either operand is undefined.

**integer % integer = integer**

---

#### Example 5.24 Modulo

```
if 3 % 2 == 1 log_info("three mod two is one");
```

---

### 5.2.4 Functions

A function is an **expression** which always returns a value. A function cannot be used without using its return value. In contrast to procedures, a function never modifies its arguments, the state of the nxlog engine or the state of a module. Functions can be polymorphic, the same function can take different arguments. Some functions also support variadic arguments denoted by the **varargs** argument type. See the [list of available functions](#).

---

#### Example 5.25 Function call

```
$current.time = now();  
if now() > 2000-01-01 00:00:00 log_info("we are in the 21st century");
```

---

## 5.3 Statements

Directives such as **Exec** take a statement as argument. After a statement is evaluated, usually the result will be a change in the state of the nxlog engine, the state of a module or the log message. A statement is terminated by a semicolon ";". Multiple statements can be specified and these will be evaluated and executed in order. The following elements can be used in statements. There is no loop operation (for, while) in the nxlog language.

---

---

### 5.3.1 Assignment

The assignment operation "=" loads the value from the expression evaluated on the right into a **field** on the left.

---

#### Example 5.26 Assignment

```
$event.rcvd = now();
```

---

### 5.3.2 Block

A block consists of one or more statements within curly braces "{}". This is typically used with **conditional statements** as in the example below.

---

#### Example 5.27 Conditional statement block

```
if now() > 2000-01-01 00:00:00
{
    log_info("we are in the");
    log_info("21st century");
}
```

---

### 5.3.3 Procedures

Though both functions can take arguments, procedures are the opposite of function calls. Procedures never return a value, thus these can be used as **statements**. A procedure can modify its argument if it is a field, or it can modify the state of the nxlog engine, the state of a module or the log message. Procedures can also be polymorphic, the same procedure can take different arguments. Some procedures also support variadic arguments denoted by the **varargs** argument type. See the [list of available procedures](#).

---

#### Example 5.28 Procedure call

```
log_info("this is a procedure call");
```

---

### 5.3.4 If-Else

A conditional statement starts with the "if" keyword followed by a boolean expression and a statement. The "else" with another statement is optional. Brackets around the expression are also optional.

---

---

**Example 5.29** Conditional statements

---

```
if now() > 2000-01-01 00:00:00 log_info("we are in the 21st century");

# same as above but with brackets
if ( now() > 2000-01-01 00:00:00 ) log_info("we are in the 21st century");

# conditional statement block
if now() > 2000-01-01 00:00:00
{
    log_info("we are in the 21st century");
}

# conditional statement block with an else branch
if now() > 2000-01-01 00:00:00
{
    log_info("we are in the 21st century");
}
else log_info("we are not yet in the 21st century");
```

---

Similiarly to Perl, the nxlog language doesn't have a switch statement. This can be accomplished by the appropriate use of conditional if-else statements as in the example below.

---

**Example 5.30** Emulating switch with if-else

---

```
if ( $value == 1 )
    log_info("1");
else if ( $value == 2 )
    log_info("2");
else if ( $value == 3 )
    log_info("3");
else
    log_info("default");
```

---

---

**Note**

The Perl shorthand "elsif" is not supported. There is no "unless" either.

---

## 5.4 Variables

**Fields** are not persistent because the scope of these is the log message itself, though fields can be used for storing temporary data during the processing of one log message or to pass values across modules along the route. Unfortunately if we need to store some value persistently, for example to set a state on a condition, then the fields cannot be used.

The nxlog engine supports module variables for this purpose. A module variable is referenced by a string value. A module variable can only be accessed from the same module due to concurrency reasons. A module variable with the same name is a different variable when referenced from another module. A module variable can be created with an expiry value or it can have an infinite lifetime. If a variable is created with a lifetime, it will be destroyed automatically when the lifetime expires. This can be also used as a means of a garbage collection method, or it can reset the value of the variable automatically. The module variables can store values of any type. Module variables are supported by all modules automatically. See [create\\_var\(\)](#), [delete\\_var\(\)](#), [set\\_var\(\)](#) and [get\\_var\(\)](#) for using module variables.

---

---

**Example 5.31** Simple event correlation using module variables

---

If the number of login failures exceeds 3 within 45 seconds, then we generate an internal log message.

```
if $Message =~ /login failure/
{
    if not defined get_var('login_failures')
    { # create the variable if it doesn't exist
        create_var('login_failures', 45);
        set_var('login_failures', 1);
    }
    else
    { # increase the variable and check if it is over the limit
        set_var('login_failures', get_var('login_failures') + 1);
        if get_var('login_failures') >= 3
            log_warning("3 or more login failures detected within 45 seconds");
    }
}
```

Note that this method is a bad example for this task, because the lifetime of the variable is not affected by `set_var()`. For example if there is one login failure at time 0s, then three login failures at 45s, 46s and 47sec, then this algorithm will not be able to detect this, because the variable will be automatically cleared at 45s, and the last three login failures are not noticed even though they happened within 3 seconds. Also note that this method can only work in real time because the timing is not based on values available in the log message, though this can be reprogrammed by storing the event time in another variable.

---

## 5.5 Statistical counters

Statistical counters are similar to **variables** but these only support integers. The difference is that statistical counters can use different algorithms to recalculate their value every time they are updated or read. A statistical counter can be created with the `create_stat()` procedure calls. The following types are available for statistical counters:

**COUNT** This will aggregate the values added, so the value of the counter will increase if only positive integers are added until the counter is destroyed, or indefinitely if the counter has no expiry.

**COUNTMIN** This will calculate the minimum value of the counter.

**COUNTMAX** This will calculate the maximum value of the counter.

**AVG** This algorithm calculates the average over the specified interval.

**AVGMIN** This algorithm calculates the average over the specified interval and the value of the counter is always the lowest which was ever calculated during the lifetime of the counter.

**AVGMAX** Similar to AVGMIN but returns the highest value calculated during the lifetime of the counter.

**RATE** This calculates the value over the specified interval, can be used to calculate events per second (EPS) values.

**RATEMIN** Will return the lowest rate calculated during the lifetime of the counter.

**RATEMAX** Will return the highest rate calculated during the lifetime of the counter.

**GRAD** This calculates the change of the rate of the counter over the specified interval, which is the gradient.

**GRADMIN** Lowest gradient calculated during the lifetime of the counter.

**GRADMAX** Highest gradient calculated during the lifetime of the counter.

A statistical counter will only return a value if the time specified in the interval argument has elapsed since it was created. Statistical counters can be also created with a lifetime. When they expire, they will be destroyed similarly to module variables.

After a statistical counter is created, it can be updated with the `add_stat()` procedure call. The value of the counter can be read with the `get_stat()` function call. The value of the statistical counter is recalculated during these calls, but it does never happen

---

automatically in a timed fashion, so this can lead to slight distortion of the calculated value if the add and read operations are infrequent.

Another feature of statistical counters is that it is possible to specify a time value both during **creation**, **update** and **read** making offline log processing possible.

---

**Example 5.32** Simple event correlation using statistical counters

---

If the number of login failures exceeds 3 within 45 seconds, then we generate an internal log message. This accomplishes the exact same task as our **previous algorithm** did with module variables, except that this is a lot simpler. In addition, this method is more precise, because it uses the timestamp from the log message instead of relying on the current time, so it is possible to use this for offline log analysis as well.

```
if $Message =~ /login failure/
{
    # create will no do anything if the counter already exists
    create_stat('login_failures', 'RATE', 45, $EventTime);
    add_stat('login_failures', 1, $EventTime);
    if get_stat('login_failures', $EventTime) >= 3
        log_warning("3 or more login failures detected within 45 seconds");
}
```

Note that this is still not perfect because the time window used in the rate calculation does not shift, so the problem **described in our previous example** also affects this version and it is possible that this algorithm does not work in some situations.

---

## 5.6 List of available functions and procedures

### 5.6.1 Functions and procedures exported by core

#### 5.6.1.1 Functions exported by core

**string lc(string arg);**

**description** Convert a string to lower case.

**arguments**

**arg** type: **string**

**return type** **string**

**string uc(string arg);**

**description** Convert a string to upper case.

**arguments**

**arg** type: **string**

**return type** **string**

**datetime now();**

**description** Return the current time.

**return type** **datetime**

**string type(unknown arg);**

**description** Returns the type of a variable. Can be "boolean", "integer", "string", "datetime", "ip4addr", "ip6addr", "regexp", "binary". For values with the unknown type, it returns undef.

**arguments**

**arg** type: **unknown**

---

return type **string**

**integer** `microsecond(datetime datetime);`

**description** Return the microsecond part from the time value.

**arguments**

**datetime** type: **datetime**

return type **integer**

**integer** `second(datetime datetime);`

**description** Return the second part from the time value.

**arguments**

**datetime** type: **datetime**

return type **integer**

**integer** `minute(datetime datetime);`

**description** Return the minute part from the time value.

**arguments**

**datetime** type: **datetime**

return type **integer**

**integer** `hour(datetime datetime);`

**description** Return the hour part from the time value.

**arguments**

**datetime** type: **datetime**

return type **integer**

**integer** `day(datetime datetime);`

**description** Return the day part from the time value.

**arguments**

**datetime** type: **datetime**

return type **integer**

**integer** `month(datetime datetime);`

**description** Return the month part from the datetime value.

**arguments**

**datetime** type: **datetime**

return type **integer**

**integer** `year(datetime datetime);`

**description** Return the year part from the datetime value.

**arguments**

**datetime** type: **datetime**

return type **integer**

**datetime** `fix_year(datetime datetime);`

**description** Set year value to current in a datetime which was parsed with a missing year such as BSD syslog or cisco timestamps.

**arguments**

---



**datetime** type: **datetime**

**return type** **datetime**

**integer dayofweek(datetime datetime);**

**description** The number of days since Sunday in the range of 0-6.

**arguments**

**datetime** type: **datetime**

**return type** **integer**

**integer dayofyear(datetime datetime);**

**description** Return the day number of the year in the range of 1-366.

**arguments**

**datetime** type: **datetime**

**return type** **integer**

**string string(unknown arg);**

**description** Convert the argument to string.

**arguments**

**arg** type: **unknown**

**return type** **string**

**integer integer(unknown arg);**

**description** Parse and convert the string argument to an integer. For datetime type it returns the number of microseconds since epoch

**arguments**

**arg** type: **unknown**

**return type** **integer**

**datetime datetime(integer arg);**

**description** Convert the integer argument expressing the number of microseconds since epoch to datetime.

**arguments**

**arg** type: **integer**

**return type** **datetime**

**datetime parsedate(string arg);**

**description** Parse a datetime argument.

**arguments**

**arg** type: **string**

**return type** **datetime**

**string strftime(datetime datetime, string fmt);**

**description** Convert a datetime to a string with the given format. See the manual of strftime(3) for the format specification.

**arguments**

**datetime** type: **datetime**

**fmt** type: **string**

**return type** **string**

**datetime strftime(string input, string fmt);**

---

**description** Convert a string to a datetime with the given format. See the manual of `strptime(3)` for the format specification.

**arguments**

**input** type: `string`

**fmt** type: `string`

**return type** `datetime`

`string hostname();`

**description** Return the hostname.

**return type** `string`

`unknown get_var(string varname);`

**description** Return the value of the variable or undef if it doesn't exist.

**arguments**

**varname** type: `string`

**return type** `unknown`

`integer get_stat(string statname);`

**description** Return the value of the statistical counter or undef if it doesn't exist.

**arguments**

**statname** type: `string`

**return type** `integer`

`integer get_stat(string statname, datetime time);`

**description** Return the value of the statistical counter or undef if it doesn't exist. The time argument specifies the current time.

**arguments**

**statname** type: `string`

**time** type: `datetime`

**return type** `integer`

`ip4addr ip4addr(integer arg);`

**description** Convert the integer argument to an ip4addr type.

**arguments**

**arg** type: `integer`

**return type** `ip4addr`

`ip4addr ip4addr(integer arg, boolean nto);`

**description** Convert the integer argument to an ip4addr type. If 'nto' is set to true, the integer is assumed to be in network byte order. Instead of '1.2.3.4' the result will be '4.3.2.1'.

**arguments**

**arg** type: `integer`

**nto** type: `boolean`

**return type** `ip4addr`

`string substr(string src, integer from);`

**description** Return the string starting at the byte offset specified in 'from'.

**arguments**

**src** type: `string`

---

**from** type: **integer**  
**return type** **string**

**string substr(string src, integer from, integer to);**

**description** Return a substring specified with the starting and ending positions as byte offsets from the beginning of the string.

**arguments**

**src** type: **string**  
**from** type: **integer**  
**to** type: **integer**  
**return type** **string**

**string replace(string subject, string src, string dst);**

**description** Replace all occurrences of 'src' with 'dst' in the 'subject' string.

**arguments**

**subject** type: **string**  
**src** type: **string**  
**dst** type: **string**  
**return type** **string**

**string replace(string subject, string src, string dst, integer count);**

**description** Replace 'count' number occurrences of 'src' with 'dst' in the 'subject' string.

**arguments**

**subject** type: **string**  
**src** type: **string**  
**dst** type: **string**  
**count** type: **integer**  
**return type** **string**

**integer size(string str);**

**description** Return the size of the string 'str' in bytes.

**arguments**

**str** type: **string**  
**return type** **integer**

**boolean dropped();**

**description** Return TRUE if the currently processed event has been already dropped.

**return type** **boolean**

#### 5.6.1.2 Procedures exported by core

**log\_debug(unknown arg, varargs args);**

**description** Print the argument(s) at DEBUG log level.

**arguments**

**arg** type: **unknown**  
**args** type: **varargs**

**debug(unknown arg, varargs args);**

---

**description** Print the argument(s) at DEBUG log level. Same as `log_debug()`.

**arguments**

**arg** type: `unknown`

**args** type: `varargs`

`log_info(unknown arg, varargs args);`

**description** Print the argument(s) at INFO log level.

**arguments**

**arg** type: `unknown`

**args** type: `varargs`

`log_warning(unknown arg, varargs args);`

**description** Print the argument(s) at WARNING log level.

**arguments**

**arg** type: `unknown`

**args** type: `varargs`

`log_error(unknown arg, varargs args);`

**description** Print the argument(s) at ERROR log level.

**arguments**

**arg** type: `unknown`

**args** type: `varargs`

`delete(unknown arg);`

**description** Delete the field from the event, i.e. `delete($field)`. Note that doing `'$field = undef'` is not the same, though after both operations the field will be undefined.

**arguments**

**arg** type: `unknown`

`create_var(string varname);`

**description** Create a module variable with the specified name. The variable will be created with an infinite lifetime.

**arguments**

**varname** type: `string`

`create_var(string varname, integer lifetime);`

**description** Create a module variable with the specified name with the lifetime given in seconds. If the lifetime expires, the variable is deleted automatically and `get_var(name)` will return `undef`.

**arguments**

**varname** type: `string`

**lifetime** type: `integer`

`create_var(string varname, datetime expiry);`

**description** Create a module variable with the specified name. Expiry specifies when the variable should be deleted automatically.

**arguments**

**varname** type: `string`

**expiry** type: `datetime`

`delete_var(string varname);`

---

**description** Delete the module variable with the specified name if it exists.

**arguments**

**varname** type: **string**

```
set_var(string varname, unknown value);
```

**description** Set a value of a module variable. If the variable does not exist, it will be created with an infinite lifetime.

**arguments**

**varname** type: **string**

**value** type: **unknown**

```
create_stat(string statname, string type);
```

**description** Create a module statistical counter with the specified name using the current time. The statistical counter will be created with an infinite lifetime. The type argument can be any of the following to select the required algorithm for calculating the value of the statistical counter: COUNT, COUNTMIN, COUNTMAX AVG, AVGMIN, AVGMAX, RATE, RATEMIN, RATEMAX, GRAD, GRADMIN, GRADMAX. See the statistical counters section for the description of these. This procedure with two parameters can only be used with COUNT, otherwise the interval parameter must be specified.

**arguments**

**statname** type: **string**

**type** type: **string**

```
create_stat(string statname, string type, integer interval);
```

**description** Create a module statistical counter with the specified name to be calculated over 'interval' seconds and using the current time. The statistical counter will be created with an infinite lifetime.

**arguments**

**statname** type: **string**

**type** type: **string**

**interval** type: **integer**

```
create_stat(string statname, string type, integer interval, datetime time);
```

**description** Create a module statistical counter with the specified name to be calculated over 'interval' seconds and the time value specified in the argument named 'time'. The statistical counter will be created with an infinite lifetime.

**arguments**

**statname** type: **string**

**type** type: **string**

**interval** type: **integer**

**time** type: **datetime**

```
create_stat(string statname, string type, integer interval, datetime time, integer lifetime);
```

**description** Create a module statistical counter with the specified name to be calculated over 'interval' seconds and the time value specified in the argument named 'time'. The statistical counter will expire after 'lifetime' seconds.

**arguments**

**statname** type: **string**

**type** type: **string**

**interval** type: **integer**

**time** type: **datetime**

**lifetime** type: **integer**

```
create_stat(string statname, string type, integer interval, datetime time, datetime expiry);
```

---

**description** Create a module statistical counter with the specified name to be calculated over 'interval' seconds and the time value specified in the argument named 'time'. The statistical counter will expire at 'expiry'.

**arguments**

**statname** type: **string**  
**type** type: **string**  
**interval** type: **integer**  
**time** type: **datetime**  
**expiry** type: **datetime**

**add\_stat(string statname, integer value);**

**description** Add 'value' to the statistical counter using the current time.

**arguments**

**statname** type: **string**  
**value** type: **integer**

**add\_stat(string statname, integer value, datetime time);**

**description** Add 'value' to the statistical counter using the time specified in the argument named 'time'.

**arguments**

**statname** type: **string**  
**value** type: **integer**  
**time** type: **datetime**

**sleep(integer interval);**

**description** Sleep the specified number of microseconds. This procedure is provided for testing purposes mostly. It can be used as a poor man's rate limiting tool, though its use is not recommended.

**arguments**

**interval** type: **integer**

**drop();**

**description** Drop the currently processed event's log.

**rename\_field(string old, string new);**

**description** Rename a field.

**arguments**

**old** type: **string**  
**new** type: **string**

**reroute(string routename);**

**description** Move the currently processed event data to the route specified in the argument. The event data will enter the route as if it was received by an input module there.

**arguments**

**routename** type: **string**

**add\_to\_route(string routename);**

**description** Copy the currently processed event data to the the route specified in the argument. This procedure makes a copy of the data and the original will be processed normally.

**arguments**

**routename** type: **string**

## 5.6.2 Functions and procedures exported by modules

**xm\_syslog** Functions and procedures exported by **xm\_syslog**

**om\_file** Functions and procedures exported by **om\_file**

## Chapter 6

# Modules

nxlog uses loadable modules similarly to the Apache HTTP server, these are also called as plugins in another terminology. There are four types of modules: **extension**, **input**, **processor** and **output** modules. This chapter deals with the features and configuration of each specific module. General concepts about **configuring modules** were discussed in the **Configuration** chapter earlier.

### 6.1 Extension modules

Extension modules do not process log messages directly, and for this reason their instances cannot be part of a **route**. These modules can enhance the features of nxlog in different ways such as exporting new **functions and procedures**, registering additional I/O reader and writer functions to be used with modules supporting the **OutputType** and **InputType** directives. Also there are many possibilities to hook an extension module into the nxlog engine, the following modules will illustrate this.

#### 6.1.1 CSV (xm\_csv)

This module provides functions and procedures to process data formatted as comma separated values (CSV) and allows to convert to CSV and parse CSV into **fields**.

The **pm\_transformer** module also provides a simple interface to parse and generate CSV lines, but with the API this **xm\_csv** module exports to the nxlog language, it is possible to solve a lot more complex tasks involving CSV formatted data.

---

#### Note

It is possible to use more than one **xm\_csv** module instance with different options in order to support different CSV formats at the same time. For this reason, functions and procedures exported by the module are public and must be referenced by the module instance name.

---

##### 6.1.1.1 Configuration

In addition to the **common module directives**, the following can be used to configure the **xm\_csv** module instance.

**QuoteChar** This optional directive takes a single character (see **below**) as argument to specify the quote character used to enclose fields. If **QuoteOptional** is TRUE, then only **string** type fields are quoted. If this directive is not specified, the default quote character is the double-quote character (").

**EscapeChar** This optional directive takes a single character (see **below**) as argument to specify the escape character used to escape special characters. The escape character is used to prefix the following characters: the escape character itself, the **quote character** and the **delimiter character**. If **EscapeControl** is TRUE, the \n, \r, \t, \b (newline, carriage-return, tab, backspace) control characters are also escaped. If this directive is not specified, the default escape character is the backslash character (\).

---



**Delimiter** This optional directive takes a single character (see [below](#)) as argument to specify the delimiter character used to separate fields. If this directive is not specified, the default escape character is the comma character (.). Note that there is no delimiter after the last field.

**QuoteOptional** This directive has been deprecated in favor of [QuoteMethod](#), please use that instead.

**QuoteMethod** This optional directive can take the following values:

**String** Only [string](#) type fields will be quoted. Has the same effect as [QuoteOptional](#) set to `TRUE`. This is the default behavior if the [QuoteMethod](#) directive is not specified.

**All** All fields will be quoted.

**None** Nothing will be quoted. This can be problematic if the field value (typically text that can contain any character) can contain the delimiter character. Make sure that this is escaped or replaced with something else.

Note that this directive only effects CSV generation when using [to\\_csv\(\)](#). The CSV parser can automatically detect the quotation.

**EscapeControl** If this optional boolean directive is set to `TRUE`, control characters are also escaped. See the [EscapeChar](#) directive for details. If this directive is not specified, control characters are escaped by default. Note that this is necessary in order to allow single line CSV field lists which contain line-breaks.

**Fields** This is a comma separated list of fields which will be filled from the input parsed. Field names with or without the dollar sign "\$" are also accepted. This directive is mandatory. The fields will be stored as [strings](#) by default unless their type is explicitly specified with the [FieldTypes](#) directive.

**FieldTypes** This optional directive specifies the list of types corresponding to the field names defined in [Fields](#). If specified, the number of types must match the number of field names specified with [Fields](#). If this directive is omitted, all fields will be stored as [strings](#). This directive has no effect on the fields-to-csv conversion.

#### 6.1.1.1.1 Specifying characters for quote, escape and delimiter

The [QuoteChar](#), [EscapeChar](#) and [Delimiter](#) can be specified in different ways, mainly due to the nature of the config file format. As of this writing, the module does not support multi character strings for these parameters.

**Unquoted single character** Printable characters can be specified as an unquoted character, except for the backslash '\'. Example:

```
Delimiter ;
```

The following non-printable characters can be specified with escape sequences:

**\a** audible alert (bell)

**\b** backspace

**\t** horizontal tab

**\n** newline

**\v** vertical tab

**\f** formfeed

**\r** carriage return

To use TAB delimiting:

```
Delimiter \t
```

**A character in single quotes** The config parser strips whitespace, so it is not possible to define space as the delimiter unless it is enclosed within quotes:

```
Delimiter ' '
```

Printable characters can also be enclosed:

```
Delimiter ' ; '
```

The backslash can be specified when enclosed within quotes:

```
Delimiter '\'
```

**A character in double quotes** Double quotes can be used similarly to single quotes:

```
Delimiter " "
```

The backslash can be specified when enclosed within double quotes:

```
Delimiter "\"
```

### 6.1.1.2 Functions and procedures exported by xm\_csv

#### 6.1.1.2.1 Functions exported by xm\_csv

**string to\_csv();**

**description** Convert the specified fields to a single CSV formatted string.

**return type** **string**

#### 6.1.1.2.2 Procedures exported by xm\_csv

**parse\_csv();**

**description** Parse the raw\_event field as csv input

**parse\_csv(string source);**

**description** Parse the given string as CSV format

**arguments**

**source** type: **string**

**to\_csv();**

**description** Format the specified fields as CSV and put it into the 'raw\_event' field.

---

### 6.1.1.3 Configuration examples

---

**Example 6.1** Complex CSV format conversion

---

This example illustrates the power of nxlog and the xm\_csv module. It shows that not only can the module parse and create CSV formatted input and output, but using multiple xm\_csv modules it is possible to reorder, add, remove or modify fields and output these in a different CSV format.

```
<Extension csv1>
  Module      xm_csv
  Fields      $id, $name, $number
  FieldTypes   integer, string, integer
  Delimiter    ,
</Extension>

<Extension csv2>
  Module      xm_csv
  Fields      $id, $number, $name, $date
  Delimiter    ;
</Extension>

<Input filein>
  Module      im_file
  File        "tmp/input"
  Exec        csv1->parse_csv(); \
              $date = now(); \
              if not defined $number $number = 0; \
              csv2->to_csv();
</Input>

<Output fileout>
  Module      om_file
  File        "tmp/output"
</Output>

<Route 1>
  Path        filein => fileout
</Route>
```

Samples for the input and output files processed by the above config are shown below.

```
1, "John K.", 42
2, "Joe F.", 43
```

```
1;42;"John K.";2011-01-15 23:45:20
2;43;"Joe F.";2011-01-15 23:45:20
```

---

## 6.1.2 JSON (xm\_json)

This module provides functions and procedures to process data formatted as **JSON** and allows to convert to JSON and parse JSON into **fields**.

### 6.1.2.1 Configuration

The module does not have any module specific configuration directives.

---

### 6.1.2.2 Functions and procedures exported by xm\_json

#### 6.1.2.2.1 Functions exported by xm\_json

**string to\_json();**

**description** Converts the fields to JSON and returns it as a string value. Fields having a leading dot (.) or underscore (\_) and the 'raw\_event' will be automatically excluded.

**return type** **string**

#### 6.1.2.2.2 Procedures exported by xm\_json

**parse\_json();**

**description** Parse the raw\_event field as json input

**parse\_json(string source);**

**description** Parse the given string as JSON format

**arguments**

**source** type: **string**

**to\_json();**

**description** Convert the fields to JSON and put this into the 'raw\_event' field. Fields having a leading dot (.) or underscore (\_) and the 'raw\_event' will be automatically excluded.

---

### 6.1.2.3 Configuration examples

---

#### Example 6.2 Syslog to JSON format conversion

---

The following configuration accepts syslog (both legacy and RFC5424) and converts it to JSON.

```
<Extension syslog>
  Module xm_syslog
</Extension>

<Extension json>
  Module xm_json
</Extension>

<Input in>
  Module im_tcp
  Port 1514
  Host 0.0.0.0
  Exec parse_syslog(); to_json();
</Input>

<Output out>
  Module om_file
  File "/var/log/json.txt"
</Output>

<Route r>
  Path in => out
</Route>
```

A sample is shown for the input and its corresponding output:

```
<30>Sep 30 15:45:43 host44.localdomain.hu acpid: 1 client rule loaded
```

```
{ "MessageSourceAddress": "127.0.0.1", "EventReceivedTime": "2011-03-08 14:22:41", " ←
  SyslogFacilityValue": 1, \
"SyslogFacility": "DAEMON", "SyslogSeverityValue": 5, "SyslogSeverity": "INFO", "SeverityValue ←
  ": 2, "Severity": "INFO", \
"Hostname": "host44.localdomain.hu", "EventTime": "2011-09-30 14:45:43", "SourceName": "acpid", " ←
  Message": "1 client rule loaded " }
```

---

---

**Example 6.3** Converting Windows EventLog to Syslog encapsulated JSON

---

The following configuration reads the Windows EventLog and converts it into the legacy syslog format where the message part contains the fields in JSON.

```
<Extension syslog>
  Module      xm_syslog
</Extension>

<Extension json>
  Module      xm_json
</Extension>

<Input in>
  Module      im_msvistalog
  Exec        $Message = to_json(); to_syslog_bsd();
</Input>

<Output out>
  Module      om_tcp
  Host        192.168.1.1
  Port        1514
</Output>

<Route r>
  Path        in => out
</Route>
```

A sample output is shown:

```
<14>Mar  8 14:40:11 WIN-OUNNPISDHIG Service_Control_Manager: {"EventTime":"2012-03-08 14:40:11", "EventTimeWritten":"2012-03-08 14:40:11", \
"Hostname":"WIN-OUNNPISDHIG", "EventType":"INFO", "SeverityValue":2, "Severity":"INFO", \
"SourceName":"Service Control Manager", \
"FileName":"System", "EventID":7036, "CategoryNumber":0, "RecordNumber":6788, "Message":"The nxlog service entered the running state. ", \
"EventReceivedTime":"2012-03-08 14:40:12"}
```

---

### 6.1.3 XML (xm\_xml)

This module provides functions and procedures to process data formatted as Extensible Markup Language (XML) and allows to convert to XML and parse XML into **fields**.

#### 6.1.3.1 Configuration

The module does not have any module specific configuration directives.

#### 6.1.3.2 Functions and procedures exported by xm\_xml

##### 6.1.3.2.1 Functions exported by xm\_xml

**string** `to_xml()`;

**description** Converts the fields to XML and returns it as a string value. Fields having a leading dot (.) or underscore (\_) and the 'raw\_event' will be automatically excluded.

**return type** **string**

---

### 6.1.3.2 Procedures exported by xm\_xml

**parse\_xml () ;**

**description** Parse the raw\_event field as xml input

**parse\_xml (string source) ;**

**description** Parse the given string as XML format

**arguments**

**source** type: **string**

**to\_xml () ;**

**description** Convert the fields to XML and put this into the 'raw\_event' field. Fields having a leading dot (.) or underscore (\_) and the 'raw\_event' will be automatically excluded.

### 6.1.3.3 Configuration examples

#### Example 6.4 Syslog to XML format conversion

The following configuration accepts Syslog (both legacy and RFC5424) and converts it to XML.

```
<Extension syslog>
  Module xm_syslog
</Extension>

<Extension xml>
  Module xm_xml
</Extension>

<Input in>
  Module im_tcp
  Port 1514
  Host 0.0.0.0
  Exec parse_syslog(); to_xml();
</Input>

<Output out>
  Module om_file
  File "/var/log/log.xml"
</Output>

<Route r>
  Path in => out
</Route>
```

A sample is shown for the input and its corresponding output:

```
<30>Sep 30 15:45:43 host44.localdomain.hu acpid: 1 client rule loaded
```

```
<Event><MessageSourceAddress>127.0.0.1</MessageSourceAddress><EventReceivedTime>2012-03-08  ←
  15:05:39</EventReceivedTime>\
<SyslogFacilityValue>3</SyslogFacilityValue><SyslogFacility>DAEMON</SyslogFacility><  ←
  SyslogSeverityValue>6</SyslogSeverityValue>\
<SyslogSeverity>INFO</SyslogSeverity><SeverityValue>2</SeverityValue><Severity>INFO</  ←
  Severity><Hostname>host44.localdomain.hu</Hostname>\
<EventTime>2012-09-30 15:45:43</EventTime><SourceName>acpid</SourceName><Message>1 client  ←
  rule loaded</Message></Event>
```

**Example 6.5** Converting Windows EventLog to Syslog encapsulated XML

The following configuration reads the Windows EventLog and converts it into the legacy syslog format where the message part contains the fields in XML.

```
<Extension syslog>
  Module      xm_syslog
</Extension>

<Extension xml>
  Module      xm_xml
</Extension>

<Input in>
  Module      im_msvistalog
  Exec        $Message = to_xml(); to_syslog_bsd();
</Input>

<Output out>
  Module      om_tcp
  Host        192.168.1.1
  Port        1514
</Output>

<Route r>
  Path        in => out
</Route>
```

A sample output is shown:

```
<14>Mar  8 15:12:12 WIN-OUNNPISDHIG Service_Control_Manager: <Event><EventTime>2012-03-08 15:12:12</EventTime>\
<EventTimeWritten>2012-03-08 15:12:12</EventTimeWritten><Hostname>WIN-OUNNPISDHIG</Hostname>\
<EventType>INFO</EventType>\
<SeverityValue>2</SeverityValue><Severity>INFO</Severity><SourceName>Service Control Manager</SourceName>\
<FileName>System</FileName><EventID>7036</EventID><CategoryNumber>0</CategoryNumber><RecordNumber>6791</RecordNumber>\
<Message>The nxlog service entered the running state. </Message><EventReceivedTime>2012-03-08 15:12:14</EventReceivedTime></Event>
```

### 6.1.4 GELF (xm\_gelf)

This module provides an output writer function which can be used to generate output in Graylog Extended Log Format (GELF) and feed that into **Graylog2** or GELF compliant tools.

The advantage of using this module over syslog (e.g. Snare Agent and others) is that the GELF format contains structured data in JSON and makes the fields available to analysis. This is especially convenient with sources such as the Windows EventLog which already generate logs in a structured format.

The **GELF** output generated by this module includes all fields, except the following:

- The 'raw\_event' field.
- Fields starting with a leading dot (.).
- Fields starting with a leading underscore (\_).

In order to make nxlog output GELF formatted data, the following needs to be done:

1. Make sure the xm\_gelf module is loaded:

```
<Extension gelf>
```



```
Module      xm_gelf
</Extension>
```

2. Set the **OutputType** to GELF in your output module (which is **om\_udp**):

```
OutputType  GELF
```

#### 6.1.4.1 Configuration

The module does not have any module specific configuration directives.

#### 6.1.4.2 Configuration examples

---

**Example 6.6** Sending Windows EventLog to Graylog2 in GELF

---

The following configuration reads the Windows EventLog and sends it to the Graylog2 server in GELF format.

```
<Extension gelf>
  Module      xm_gelf
</Extension>

<Input in>
  # Use 'im_mseventlog' for Windows XP and 2003
  Module      im_msvistalog
</Input>

<Output out>
  Module      om_udp
  Host        192.168.1.1
  Port        12201
  OutputType  GELF
</Output>

<Route r>
  Path        in => out
</Route>
```

---

---

**Example 6.7** Parsing a CSV file and sending it to Graylog2 in GELF

---

Using the following config file nxlog will read a CSV file containing 3 fields and forwards it in GELF so that the fields will be available on the server.

```
<Extension gelf>
  Module xm_gelf
</Extension>

<Extension csv>
  Module xm_csv
  Fields      $name, $number, $location
  FieldTypes  string, integer, string
  Delimiter   ,
</Extension>

<Input in>
  Module im_file
  File  "/var/log/app/csv.log"
</Input>

<Output out>
  Module om_udp
  Host  192.168.1.1
  Port  12201
  OutputType GELF
</Output>

<Route r>
  Path in => out
</Route>
```

---

### 6.1.5 Character set conversion (xm\_charconv)

This module provides functions and procedures to convert strings between different character sets (codepages). Reasons for the existence of this module are outlined in the [Character set and i18n support](#) section.

The [convert\\_fields\(\)](#) procedure and the [convert\(\)](#) function supports all encodings available to iconv. See **iconv -l** for a list of encoding names.

#### 6.1.5.1 Configuration

In addition to the [common module directives](#), the following can be used to configure the xm\_charconv module instance.

**AutodetectCharsets** This optional directive takes a comma separated list of character set names. When 'auto' is specified as the source encoding for [convert\(\)](#) or [convert\\_fields\(\)](#), these charsets will be tried for conversion.

#### 6.1.5.2 Functions and procedures exported by xm\_charconv

##### 6.1.5.2.1 Functions exported by xm\_charconv

```
string convert(string source, string srcencoding, string dstencoding);
```

**description** This function converts the source string to the encoding specified in 'dstencoding' from 'srcencoding'. 'srcencoding' can be 'auto' to request auto detection.

**arguments**

---

**source** type: **string**  
**srcencoding** type: **string**  
**dstencoding** type: **string**  
**return type** **string**

#### 6.1.5.2.2 Procedures exported by xm\_charconv

**convert\_fields(string srcencoding, string dstencoding);**

**description** Convert all string type fields of a log message from 'srcencoding' to 'dstencoding'. 'srcencoding' can be "auto" to request auto detection.

**arguments**

**srcencoding** type: **string**  
**dstencoding** type: **string**

#### 6.1.5.3 Configuration examples

This configuration shows an example of character set autodetection. The input file can contain differently encoded lines and using autodetection the module normalizes output to utf-8.

---

**Example 6.8** Character set autodetection of various input encodings

---

```
<Extension charconv>
  Module      xm_charconv
  AutodetectCharsets utf-8, euc-jp, utf-16, utf-32, iso8859-2
</Extension>

<Input filein>
  Module im_file
  File   "tmp/input"
  Exec   convert_fields("AUTO", "utf-8");
</Input>

<Output fileout>
  Module om_file
  File   "tmp/output"
</Output>

<Route 1>
  Path   filein => fileout
</Route>
```

---

#### 6.1.6 File operations (xm\_fileop)

This module provides functions and procedures to manipulate files. Coupled with a **Schedule** block, this allows to implement various log rotation and retention policies, e.g.:

- log file retention based on file size,
- log file retention based on file age,
- cyclic log file rotation and retention.

---

**Note**

Rotating, renaming or removing the file written by **om\_file** is also supported with the help of the **reopen** procedure.

---

### 6.1.6.1 Configuration

The module does not have any module specific configuration directives.

### 6.1.6.2 Functions and procedures exported by xm\_fileop

#### 6.1.6.2.1 Functions exported by xm\_fileop

**string file\_read(string file);**

**description** Return the contents of the file as a string value. On error undef is returned and an error is logged.

**arguments**

file type: **string**

**return type** **string**

**boolean file\_exists(string file);**

**description** Return TRUE if the file exists and is a regular file.

**arguments**

file type: **string**

**return type** **boolean**

**string file\_basename(string file);**

**description** Strip the directory name from the full file path. `basename('/var/log/app.log')` will return `'app.log'`.

**arguments**

file type: **string**

**return type** **string**

**string file\_dirname(string file);**

**description** Return the directory name of the full file path. `dirname('/var/log/app.log')` will return `'/var/log'`. Returns an empty string if 'file' does not contain any directory separators.

**arguments**

file type: **string**

**return type** **string**

**datetime file\_mtime(string file);**

**description** Return the last modification time of the file. On error undef is returned and an error is logged.

**arguments**

file type: **string**

**return type** **datetime**

**datetime file\_ctime(string file);**

**description** Return the creation or inode-changed time of the file. On error undef is returned and an error is logged.

**arguments**

file type: **string**

**return type** **datetime**

**string file\_type(string file);**

**description** Return the type of the file. The following string values can be returned: FILE, DIR, CHAR, BLOCK, PIPE, LINK, SOCKET, UNKNOWN. On error undef is returned and an error is logged.

**arguments**

**file** type: **string**

**return type** **string**

**integer file\_size(string file);**

**description** Return the size of the file. On error undef is returned and an error is logged.

**arguments**

**file** type: **string**

**return type** **integer**

**integer file\_inode(string file);**

**description** Return the inode number of the file. On error undef is returned and an error is logged.

**arguments**

**file** type: **string**

**return type** **integer**

**string dir\_temp\_get();**

**description** Return the name of a directory suitable as a temporary storage location.

**return type** **string**

**boolean dir\_exists(string path);**

**description** Return TRUE if the 'path' exists and is a directory. On error undef is returned and an error is logged.

**arguments**

**path** type: **string**

**return type** **boolean**

#### 6.1.6.2.2 Procedures exported by xm\_fileop

**file\_cycle(string file);**

**description** Do a cyclic rotation on 'file'. 'file' will be moved to "'file'.1". If "'file'.1" already exists it will be moved to "'file'.2" and so on. This procedure will reopen the LogFile if this is cycled. An error is logged if the operation fails.

**arguments**

**file** type: **string**

**file\_cycle(string file, integer max);**

**description** Do a cyclic rotation on 'file'. 'file' will be moved to "'file'.1". If "'file'.1" already exists it will be moved to "'file'.2" and so on. 'max' specifies the maximum number of files to keep. E.g. if 'max' is 5, "'file'.6" will be deleted. This procedure will reopen the LogFile if this is cycled. An error is logged if the operation fails.

**arguments**

**file** type: **string**

**max** type: **integer**

**file\_rename(string old, string new);**

**description** Rename the file 'old' to 'new'. If the file 'new' exists, it will be overwritten. Moving files or directories across devices may not be possible. This procedure will reopen the LogFile if this is renamed. An error is logged if the operation fails.

**arguments**

**old** type: **string**

new type: **string**

**file\_copy(string src, string dst);**

**description** Copy the file 'src' to 'dst'. If file 'dst' already exists, its contents will be overwritten. An error is logged if the operation fails.

**arguments**

src type: **string**

dst type: **string**

**file\_remove(string file);**

**description** Remove the file 'file'. It is possible to specify a wildcard in filenames (but not in the path). If you use backslash as the directory separator with wildcards, make sure to escape this (e.g. 'C:\\test\\\*.log'). This procedure will reopen the LogFile if this is removed. An error is logged if the operation fails.

**arguments**

file type: **string**

**file\_remove(string file, datetime older);**

**description** Remove the file 'file' if its creation time is older than the value specified in 'older'. It is possible to specify a wildcard in filenames (but not in the path). If you use backslash as the directory separator with wildcards, make sure to escape this (e.g. 'C:\\test\\\*.log'). This procedure will reopen the LogFile if this is removed. An error is logged if the operation fails.

**arguments**

file type: **string**

older type: **datetime**

**file\_link(string src, string dst);**

**description** Create a hardlink from 'src' to 'dst'. An error is logged if the operation fails.

**arguments**

src type: **string**

dst type: **string**

**file\_append(string src, string dst);**

**description** Append the contents of the file 'src' to 'dst'. 'dst' will be created if it does not exist. An error is logged if the operation fails.

**arguments**

src type: **string**

dst type: **string**

**file\_write(string file, string value);**

**description** Write value into 'file'. 'file' will be created if it does not exist. An error is logged if the operation fails.

**arguments**

file type: **string**

value type: **string**

**file\_truncate(string file);**

**description** Truncate the file to zero length. If 'file' does not exist, it will be created. An error is logged if the operation fails.

**arguments**

file type: **string**

---

**file\_truncate(string file, integer offset);**

**description** Truncate the file to the size specified in 'offset'. If 'file' does not exist, it will be created. An error is logged if the operation fails.

**arguments**

**file** type: **string**

**offset** type: **integer**

**file\_chown(string file, integer uid, integer gid);**

**description** Change file ownership. This function is only implemented on POSIX systems where chown() is available in the underlying OS. An error is logged if the operation fails.

**arguments**

**file** type: **string**

**uid** type: **integer**

**gid** type: **integer**

**file\_chmod(string file, integer mode);**

**description** Change file permission. This function is only implemented on POSIX systems where chmod() is available in the underlying OS. An error is logged if the operation fails.

**arguments**

**file** type: **string**

**mode** type: **integer**

**file\_touch(string file);**

**description** Update the last modification time of 'file' or create it if 'file' does not exist. An error is logged if the operation fails.

**arguments**

**file** type: **string**

**dir\_make(string path);**

**description** Create a directory recursively (i.e. as 'mkdir -p'). It succeeds if the directory already exists. An error is logged if the operation fails.

**arguments**

**path** type: **string**

**dir\_remove(string file);**

**description** Remove the directory from the filesystem.

**arguments**

**file** type: **string**

---

### 6.1.6.3 Configuration examples

---

**Example 6.9** Rotation of the internal LogFile

---

This example shows how to rotate the internal logfile based on time and size.

```
define LOGFILE "/var/log/nxlog/nxlog.log"

<Extension fileop>
  Module      xm_fileop

  # Check the size of our log file every hour and rotate if it is larger than 1Mb
  <Schedule>
    Every      1 hour
    Exec       if (file_size(%LOGFILE%) >= 1M) file_cycle(%LOGFILE%, 2);
  </Schedule>

  # Rotate our log file every week on sunday at midnight
  <Schedule>
    When       @weekly
    Exec       file_cycle(%LOGFILE%, 2);
  </Schedule>
</Extension>
```

---

### 6.1.7 Multi-line message parser (xm\_multiline)

Multi-line messages such as exception logs and stack traces are quite common in logs. Unfortunately when the log messages are stored in files or forwarded over the network without any encapsulation, the newline character present in messages spanning multiple lines confuse simple linebased parsers which treat every line as a separate event.

Usually each multi-line message has a header (e.g. timestamp + severity) or they can be fixed in line count. This information allows the message to be reconstructed, i.e. lines concatenated which belong to a single event. The xm\_multiline module works exactly this way.

The name of the xm\_multiline module instance can be used by input modules as the input reader specified with the **InputType** directive. For each input source a separate context is maintained by the module so that multi-line messages coming from several simultaneous sources can be still correctly processed. An input source is a file for **im\_file** (with wildcards it is one source for each file), a connection for **im\_ssl** and **im\_tcp**. Unfortunately **im\_udp** uses a single socket and is treated as a single source even if multiple UDP (e.g. syslog) senders are forwarding logs to it.

---

**Note**

By using **module variables** it is possible to accomplish the same what this module does. The advantages of using this module over module variables are the following:

- Processes messages more efficiently.
  - It yields a more readable configuration.
  - Module event counters are correctly updated (i.e. one increment for one multi-line message and not per line).
  - It works on message source level (each file for a wildcarded im\_file module instance and each tcp connection for an im\_tcp/im\_ssl instance) and not on module instance level.
- 

#### 6.1.7.1 Configuration

The following directives can be used to configure the xm\_multiline module instance:

---



**HeaderLine** This directive takes a **string** or a **regular expression** literal. This will be matched against each line. When the match is successful, the successive lines are appended until the next header line is read. This directive is mandatory unless **FixedLineCount** is used.

---

**Note**

Until there is a new header read, the previous message is stored in the buffers because the module does not know where the message ends. If there is no new message, the last may sit in the buffers indefinitely. It may be possible to flush the buffers using a timer or on EOF, unfortunately these solutions are not perfect either (though these may be implemented in a later version). If this behaviour is unacceptable, consider using some kind of an encapsulation method (JSON, XML, RFC5425, etc) if possible.

---

**FixedLineCount** This directive takes a positive integer number defining the number of lines to concatenate. This is mostly useful with log messages spanning a fixed number of lines. When this number is defined, the module knows where the event message ends, thus it does not suffer from the problem described above.

**Exec** This directive is almost identical to the behavior of the **Exec** directive used by the other modules with the following differences:

- Each line is passed in \$raw\_event as it is read. The line includes the line terminator.
- Other fields cannot be used. If you want to store captured strings from regular expression based matching in fields, you cannot do it here.

This is mostly useful for filtering out some lines with the **drop()** procedure or rewriting them.

---



### 6.1.7.2 Configuration examples

#### Example 6.10 Parsing DICOM logs

Each log message has a header (TIMESTAMP INTEGER SEVERITY) which is used as the message boundary. A regular expression is defined for this using the **HeaderLine** directive. Each log message is prepended with an additional line containing dashes and is output into a file.

```
<Extension dicom-multi>
  Module  xm_multiline
  HeaderLine  /^d\d\d\d-\d\d-\d\d\d\d:\d\d:\d\d\.\d+\s+\d+\s+\S+\s+/
</Extension>

<Input in>
  Module  im_file
  File    "modules/extension/multiline/xm_multiline4.in"
  SavePos FALSE
  ReadFromLast FALSE
  InputType dicom-multi
</Input>

<Output out>
  Module  om_file
  File    'tmp/output'
  Exec    $raw_event = "-----\n" + $raw_event;
</Output>

<Route 1>
  Path    in => out
</Route>
```

#### An input sample:

```
2011-12-1512:22:51.000000  4296  INFO  Association Request Parameteres:
Our Implementation Class UID:    2.16.124.113543.6021.2
Our Implementation Version Name: RZDCX_2_0_1_8
Their Implementation Class UID:
Their Implementation Version Name:
Application Context Name:    1.2.840.10008.3.1.1.1
Requested Extended Negotiation: none
Accepted Extended Negotiation: none
2011-12-1512:22:51.000000  4296  DEBUG  Constructing Associate RQ PDU
2011-12-1512:22:51.000000  4296  DEBUG  WriteToConnection, length: 310, bytes written: ←
310, loop no: 1
2011-12-1512:22:51.015000  4296  DEBUG  PDU Type: Associate Accept, PDU Length: 216 + 6 ←
bytes PDU header
02 00 00 00 00 d8 00 01 00 00 50 41 43 53 20 20
20 20 20 20 20 20 20 20 20 20 52 5a 44 43 58 20
20 20 20 20 20 20 20 20 20 20 00 00 00 00 00 00
2011-12-1512:22:51.031000  4296  DEBUG  DIMSE sendDcmDataset: sending 146 bytes
```

#### The following output is produced:

```
-----
2011-12-1512:22:51.000000  4296  INFO  Association Request Parameteres:
Our Implementation Class UID:    2.16.124.113543.6021.2
Our Implementation Version Name: RZDCX_2_0_1_8
Their Implementation Class UID:
Their Implementation Version Name:
Application Context Name:    1.2.840.10008.3.1.1.1
Requested Extended Negotiation: none
Accepted Extended Negotiation: none
-----
2011-12-1512:22:51.000000  4296  DEBUG  Constructing Associate RQ PDU
-----
2011-12-1512:22:51.000000  4296  DEBUG  WriteToConnection, length: 310, bytes written: ←
310, loop no: 1
-----
```

The following configuration will process messages having a fixed string header containing dashes. Each event is then prepended with a sharp (#) and is output to a file.

An input sample:

The following output is produced:

```
#-----
1
#-----
1
2
#-----
aaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
cccccccccccccccccccccccccccccccccccccc
ddddd
```

**Example 6.12** Multi-line messages with fixed line count

The following configuration will process messages having a fixed line count of 4. Lines containing only whitespace are ignored and removed. Each event is then prepended with a sharp (#) and is output to a file.

```
<Extension multiline>
  Module xm_multiline
  FixedLineCount 4
  Exec if $raw_event =~ /\s*$/ drop();
</Extension>

<Input in>
  Module im_file
  File "modules/extension/multiline/xm_multiline2.in"
  SavePos FALSE
  ReadFromLast FALSE
  InputType multiline
</Input>

<Output out>
  Module om_file
  File 'tmp/output'
  Exec $raw_event = "#" + $raw_event;
</Output>

<Route 1>
  Path in => out
</Route>
```

An input sample:

```
1
2
3
4
1asd

2asdassad
3ewrwerew
4xcbccvbc

1dsfsdfsd
2sfsdfsdrawrwe

3sfsdfssew
4werwerwrwe
```

The following output is produced:

```
#1
2
3
4
#1asd
2asdassad
3ewrwerew
4xcbccvbc
#1dsfsdfsd
2sfsdfsdrawrwe
3sfsdfssew
4werwerwrwe
```

**Example 6.13** Multi-line messages with a syslog header

Multi-line messages are frequently logged over syslog and they end up in log files. Unfortunately from the result it looks that each line is one event with its own syslog header. It can be a common requirement to merge these back into a single event message. The following configuration does just that. It strips the syslog header from the netstat output stored as a traditional syslog formatted file and each message is then printed again with a line of dashes used as a separator.

```
<Extension syslog>
  Module xm_syslog
</Extension>

<Extension netstat>
  Module xm_multiline
  FixedLineCount 4
  Exec parse_syslog_bsd(); $raw_event = $Message + "\n";
</Extension>

<Input in>
  Module im_file
  File "modules/extension/multiline/xm_multiline3.in"
  SavePos FALSE
  ReadFromLast FALSE
  InputType netstat
</Input>

<Output out>
  Module om_file
  File 'tmp/output'
  Exec $raw_event = ←
    "-----\n"
    n" + $raw_event;
</Output>

<Route 1>
  Path in => out
</Route>
```

**An input sample:**

Nov 21 11:40:27 hostname	app[26459]: Iface	MTU Met	RX-OK	RX-ERR	RX-DRP	RX-OVR	TX-OK	←
TX-ERR	TX-DRP	TX-OVR	Flg					
Nov 21 11:40:27 hostname	app[26459]: eth2	1500 0	16936814		0	0 0		←
30486067	0	8	0 BMRU					
Nov 21 11:40:27 hostname	app[26459]: lo	16436 0	277217234		0	0 0		←
277217234	0	0	0 LRU					
Nov 21 11:40:27 hostname	app[26459]: tun0	1500 0	316943		0	0 0		←
368642	0	0	0 MOPRU					
Nov 21 11:40:28 hostname	app[26459]: Iface	MTU Met	RX-OK	RX-ERR	RX-DRP	RX-OVR	TX-OK	←
TX-ERR	TX-DRP	TX-OVR	Flg					
Nov 21 11:40:28 hostname	app[26459]: eth2	1500 0	16945117		0	0 0		←
30493583	0	8	0 BMRU					
Nov 21 11:40:28 hostname	app[26459]: lo	16436 0	277217234		0	0 0		←
277217234	0	0	0 LRU					
Nov 21 11:40:28 hostname	app[26459]: tun0	1500 0	316943		0	0 0		←
368642	0	0	0 MOPRU					
Nov 21 11:40:29 hostname	app[26459]: Iface	MTU Met	RX-OK	RX-ERR	RX-DRP	RX-OVR	TX-OK	←
TX-ERR	TX-DRP	TX-OVR	Flg					
Nov 21 11:40:29 hostname	app[26459]: eth2	1500 0	16945270		0	0 0		←
30493735	0	8	0 BMRU					
Nov 21 11:40:29 hostname	app[26459]: lo	16436 0	277217234		0	0 0		←
277217234	0	0	0 LRU					
Nov 21 11:40:29 hostname	app[26459]: tun0	1500 0	316943		0	0 0		←
368642	0	0	0 MOPRU					

**The following output is produced:**

```
-----
Iface    MTU Met    RX-OK RX-ERR RX-DRP RX-OVR    TX-OK TX-ERR TX-DRP TX-OVR Flg
eth2     1500 0    16936814      0      0 0      30486067      0      8      0 BMRU
```

### 6.1.8 Syslog (xm\_syslog)

This module provides support for the archaic BSD Syslog protocol as defined in RFC 3164 and the current IETF standard defined by RFC 5424-5426. This is achieved by exporting functions and procedures usable from the nxlog language. The transport is handled by the respective input and output modules (i.e. **im\_udp**), this module only provides a parser and helper functions to create syslog messages and handle facility and severity values.

The older but still widespread BSD syslog standard defines both the format and the transport protocol in RFC 3164. The transport protocol is UDP, but to provide reliability and security, this line based format is also commonly transferred over TCP and SSL. There is a newer standard defined in RFC 5424 also known as the IETF syslog format which obsoletes the BSD syslog format. This format overcomes most of the limitations of the old BSD syslog and allows multi-line messages and proper timestamps. The transport method is defined in RFC 5426 for UDP and RFC 5425 for TLS/SSL.

Because the IETF Syslog format supports multi-line messages, RFC 5425 defines a special format to encapsulate these by prepending the payload size in ASCII to the IETF syslog message. Messages transferred in UDP packets are self-contained and do not need this additional framing. The following input reader and output writer functions are provided by the xm\_syslog module to support this TLS transport defined in RFC 5425. While RFC 5425 explicitly defines that the TLS network transport protocol is to be used, pure TCP may be used if security is not a requirement. Syslog messages can be also persisted to files with this framing format using these functions.

**InputType Syslog\_TLS** This input reader function parses the payload size and then reads the message according to this value. It is required to support Syslog TLS transport defined in RFC 5425.

**OutputType Syslog\_TLS** This output writer function prepends the payload size to the message. It is required to support Syslog TLS transport defined in RFC 5425.

---

#### Note

The **pm\_transformer** module can also parse and create BSD and IETF syslog messages but using the functions and procedures provided by this module makes it possible to solve more complex tasks which **pm\_transformer** is not capable of on its own.

---

Structured data in IETF syslog messages is parsed and put into nxlog fields. The SD-ID will be prepended to the field name with a dot unless it is 'NXLOG@XXXX'. Consider the following syslog message:

```
<30>1 2011-12-04T21:16:10.000000+02:00 host app procid msgid [exampleSDID@32473 eventSource ←  
="Application" eventID="1011"] Message part
```

After this IETF formatted syslog message is parsed with **parse\_syslog\_ietf()**, there will be two additional fields: \$exampleSDID.eventID and \$exampleSDID.eventSource. When SD-ID is NXLOG, the field name will be the same as the SD-PARAM name. The two additional fields extracted from the structured data part of the following IETF syslog message are \$eventID and \$eventSource:

```
<30>1 2011-12-04T21:16:10.000000+02:00 host app procid msgid [NXLOG@32473 eventSource=" ←  
Application" eventID="1011"] Message part
```

All fields parsed from the structured data part are **strings**.

#### 6.1.8.1 Configuration

In addition to the **common module directives**, the following can be used to configure the xm\_syslog module instance.

**SnareDelimiter** This optional directive takes a single character as argument to specify the delimiter character used to separate fields when using the **to\_syslog\_snare()** procedure. The character specification works the same way as with the **xm\_csv** module. If this directive is not specified, the default escape character is the tab character (\t). In latter versions of Snare4 this has changed to #, so you can use this configuration directive to specify an alternative delimiter. Note that there is no delimiter after the last field.

**SnareReplacement** This optional directive takes a single character as argument to specify the replacement character substituted in place of any occurrences of the **delimiter** character inside the \$Message field when invoking the **to\_syslog\_snare()** procedure. The character specification works the same way as with the **xm\_csv** module. If this directive is not specified, the default replacement character is space.

---

## 6.1.8.2 Functions and procedures exported by xm\_syslog

### 6.1.8.2.1 Functions exported by xm\_syslog

**integer syslog\_facility\_value(string arg);**

**description** Convert a syslog facility string to an integer

**arguments**

**arg** type: **string**

**return type** **integer**

**string syslog\_facility\_string(integer arg);**

**description** Convert a syslog facility value to a string

**arguments**

**arg** type: **integer**

**return type** **string**

**integer syslog\_severity\_value(string arg);**

**description** Convert a syslog severity string to an integer

**arguments**

**arg** type: **string**

**return type** **integer**

**string syslog\_severity\_string(integer arg);**

**description** Convert a syslog severity value to a string

**arguments**

**arg** type: **integer**

**return type** **string**

### 6.1.8.2.2 Procedures exported by xm\_syslog

**parse\_syslog();**

**description** Parse the raw\_event field as either BSD Syslog (RFC3164) or IETF Syslog (RFC5424) format

**parse\_syslog(string source);**

**description** Parse the given string as either BSD Syslog (RFC3164) or IETF Syslog (RFC5424) format

**arguments**

**source** type: **string**

**parse\_syslog\_bsd();**

**description** Parse the raw\_event field as BSD Syslog (RFC3164) format

**parse\_syslog\_bsd(string source);**

**description** Parse the given string as BSD Syslog (RFC3164) format

**arguments**

**source** type: **string**

**parse\_syslog\_ietf();**



**description** Parse the `raw_event` field as IETF Syslog (RFC5424) format

**parse\_syslog\_ietf(string source);**

**description** Parse the given string as IETF Syslog (RFC5424) format

**arguments**

**source** type: **string**

**to\_syslog\_bsd();**

**description** Create a BSD Syslog formatted log message in 'raw\_event' from the fields of the event

**to\_syslog\_ietf();**

**description** Create an IETF Syslog (RFC5424) formatted log message in 'raw\_event' from the fields of the event

**to\_syslog\_snare();**

**description** Create a SNARE Syslog formatted log message in 'raw\_event'. Uses the following fields: EventTime, Hostname, SeverityValue, FileName, EventID, SourceName, AccountName, AccountType, EventType, Category, Message

### 6.1.8.3 Fields generated by xm\_syslog

The following fields are set by `xm_syslog`:

**\$raw\_event** Type **string**

Will be set to a syslog formatted string after `to_syslog_bsd()` or `to_syslog_ietf()` is called.

**\$Message** Type **string**

The message part of the syslog line, filled after `parse_syslog_bsd()` or `parse_syslog_ietf()` is called.

**\$SyslogSeverityValue** Type **integer**

The severity part of the syslog line, filled after `parse_syslog_bsd()` or `parse_syslog_ietf()` is called. The default severity is 5 (= "notice").

**\$SyslogSeverity** Type **string**

The severity part of the syslog line, filled after `parse_syslog_bsd()` or `parse_syslog_ietf()` is called. The default severity is "notice".

**\$SeverityValue** Type **integer**

Normalized severity number of the event.

**\$Severity** Type **string**

Normalized severity name of the event.

**\$SyslogFacilityValue** Type **integer**

The facility part of the syslog line, filled after `parse_syslog_bsd()` or `parse_syslog_ietf()` is called. The default facility is 1 (= "user").

**\$SyslogFacility** Type **string**

The facility part of the syslog line, filled after `parse_syslog_bsd()` or `parse_syslog_ietf()` is called. The default facility is "user".

**\$EventTime** Type **datetime**

Will be set to the timestamp found in the syslog message after `parse_syslog_bsd()` or `parse_syslog_ietf()` is called. If the year value is missing, it is set to the current year.

**\$Hostname** Type **string**

The hostname part of the syslog line, filled after `parse_syslog_bsd()` or `parse_syslog_ietf()` is called.

**\$SourceName** Type **string**

The application/program part of the syslog line, filled after `parse_syslog_bsd()` or `parse_syslog_ietf()` is called.

**\$MessageID** Type **string**

The MSGID part of the syslog message, filled after `parse_syslog_ietf()` is called.

**\$ProcessID** Type **string**

The process id in the syslog line, filled after `parse_syslog_bsd()` or `parse_syslog_ietf()` is called.

#### 6.1.8.4 Configuration examples

---

**Example 6.14** Collecting BSD style syslog messages over UDP

---

To collect BSD style syslog messages over UDP, use the `parse_syslog_bsd()` procedure coupled with the `im_udp` module as in the following example.

```
<Extension syslog>
  Module xm_syslog
</Extension>

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

<Output out>
  Module om_file
  File "/var/log/logmsg.txt"
</Output>

<Route 1>
  Path in => out
</Route>
```

---

---

**Example 6.15** Collecting IETF style syslog messages over UDP

To collect IETF style syslog messages over UDP as defined by RFC 5424 and RFC 5426, use the `parse_syslog_ietf()` procedure coupled with the `im_udp` module as in the following example. Note that the default port is 514 (as defined by RFC 5426), this is the same as for BSD syslog.

```
<Extension syslog>
  Module xm_syslog
</Extension>

<Input in>
  Module im_udp
  Host 0.0.0.0
  Port 514
  Exec parse_syslog_ietf();
</Input>

<Output out>
  Module om_file
  File "/var/log/logmsg.txt"
</Output>

<Route 1>
  Path in => out
</Route>
```

---

**Example 6.16** Collecting both IETF and BSD style syslog messages over the same UDP port

To collect IETF and BSD style syslog messages over UDP, use the `parse_syslog()` procedure coupled with the `im_udp` module as in the following example. This procedure is capable of detecting and parsing both syslog formats. Since 514 is the default UDP port number for both BSD and IETF syslog, this can be useful to collect both formats simultaneously. If you want to accept both formats on different ports then it makes sense to use the appropriate parsers as in the previous two examples.

```
<Extension syslog>
  Module xm_syslog
</Extension>

<Input in>
  Module im_udp
  Host 0.0.0.0
  Port 514
  Exec parse_syslog();
</Input>

<Output out>
  Module om_file
  File "/var/log/logmsg.txt"
</Output>

<Route 1>
  Path in => out
</Route>
```

---

---

**Example 6.17** Collecting IETF style syslog messages over TLS/SSL

To collect IETF style syslog messages over TLS/SSL as defined by RFC 5424 and RFC 5425, use the `parse_syslog_ietf()` procedure coupled with the `im_ssl` module as in the following example. Note that the default port is 6514 in this case (as defined by RFC 5425). The payload format parser is handled by the `Syslog_TLS` input reader.

```
<Extension syslog>
  Module xm_syslog
</Extension>

<Input in>
  Module im_ssl
  Host localhost
  Port 6514
  CAFile %CERTDIR%/ca.pem
  CertFile %CERTDIR%/client-cert.pem
  CertKeyFile %CERTDIR%/client-key.pem
  KeyPass secret
  InputType Syslog_TLS
  Exec parse_syslog_ietf();
</Input>

<Output out>
  Module om_file
  File "/var/log/logmsg.txt"
</Output>

<Route 1>
  Path in => out
</Route>
```

---

---

**Example 6.18** Forwarding IETF syslog over TCP

---

The following configuration uses the `to_syslog_ietf()` procedure to convert input to IETF syslog and forward it over TCP:

```
<Extension syslog>
  Module xm_syslog
</Extension>

<Input in>
  Module im_file
  File "/var/log/input.txt"
  Exec $TestField = "test value"; $Message = $raw_event;
</Input>

<Output out>
  Module om_tcp
  Host 127.0.0.1
  Port 1514
  Exec to_syslog_ietf();
  OutputType Syslog-TLS
</Output>

<Route 1>
  Path in => out
</Route>
```

Because of the Syslog-TLS framing, the raw data sent over TCP will look like the following:

```
130 <13>1 2012-01-01T16:15:52.873750Z - - - [NXLOG@14506 EventReceivedTime="2012-01-01 17:15:52" TestField="test value"] test message
```

This example shows that all fields - except those which are filled by the syslog parser - are added to the structured data part.

---

---

**Example 6.19** Conditional rewrite of the syslog facility - version 1

---

```
<Extension syslog>
  Module xm_syslog
</Extension>

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

<Output fileout>
  Module om_file
  File "/var/log/logmsg.txt"
  Exec if $Message =~ /error/ $SeverityValue = syslog_severity_value("error");
  Exec to_syslog_bsd();
</Output>

<Route 1>
  Path in => fileout
</Route>
```

---

---

**Example 6.20** Conditional rewrite of the syslog facility - version 2

The following example does almost the same thing as the previous example, except that the syslog parsing and rewrite is moved to a processor module and rewrite only occurs if the facility was modified. This can make it work faster on multi-core systems because the processor module runs in a separate thread. This method can also minimize UDP packet loss because the input module does not need to parse syslog messages and can process UDP packets faster.

```
<Extension syslog>
  Module xm_syslog
</Extension>

<Input in>
  Module im_udp
  Host 0.0.0.0
  Port 514
</Input>

<Processor rewrite>
  Module pm_null
  Exec parse_syslog_bsd();\
  if $Message =~ /error/ \
  { \
    $SeverityValue = syslog_severity_value("error");\
    to_syslog_bsd(); \
  }
</Processor>

<Output fileout>
  Module om_file
  File "/var/log/logmsg.txt"
</Output>

<Route 1>
  Path in => rewrite => fileout
</Route>
```

---

### 6.1.9 External program execution (xm\_exec)

This module provides two procedures which make it possible to execute external scripts or programs. The reason for providing these two procedures through this additional extension module is to keep the nxlog core small. A security advantage is that an administrator won't be able to execute arbitrarily scripts if this module is not loaded.

---

**Note**

The **om\_exec** and **im\_exec** modules also provide support for running external programs, though the purpose of these is to pipe data to and read data from programs. The procedures provided by the **xm\_exec** module do not pipe log message data, these are intended for multiple invocations. Though data can be still passed to the executed script/program as command line arguments.

---

#### 6.1.9.1 Functions and procedures exported by xm\_exec

##### 6.1.9.1.1 Procedures exported by xm\_exec

**exec(string command, varargs args);**

**description** Execute the command passing it the supplied arguments and wait for it to terminate. The command is executed in the caller module's context. Note that the module calling this procedure will block until the process terminates.

---

Use the `exec_async()` procedure to avoid this problem. All output written to `STDOUT` and `STDERR` by the spawned process is discarded.

**arguments**

**command** type: `string`

**args** type: `varargs`

**exec\_async(string command, varargs args);**

**description** This procedure executes the command passing it the supplied arguments and does not wait for it to terminate.

**arguments**

**command** type: `string`

**args** type: `varargs`

### 6.1.9.2 Configuration examples

---

**Example 6.21** nxlog acting as a cron daemon

---

```
<Extension exec>
  Module xm_exec
  <Schedule>
    Every 1 sec
    Exec exec_async("/bin/true");
  </Schedule>
</Extension>
```

---

**Example 6.22** Sending email alerts

---

```
<Extension exec>
  Module xm_exec
</Extension>

<Input in>
  Module im_tcp
  Host 0.0.0.0
  Port 1514
  Exec if $raw_event =~ /alertcondition/ { ↵
                                ↵
                                exec_async("/bin/sh", "-c", 'echo "' + $Hostname + '\n\nRawEvent:\n' + ↵
                                $raw_event + \
                                '|usr/bin/mail -a "Content-Type: text/plain; charset=UTF-8" -s ↵
                                "ALERT" ' \
                                + 'user@domain.com' ); ↵
                                ↵
                                }
</Input>

<Output out>
  Module om_file
  File "/var/log/messages"
</Output>

<Route r>
  Path in => out
</Route>
```

---

For another example see this configuration for [file rotation](#).

### 6.1.10 Perl (xm\_perl)

This module makes it possible to execute perl code and process event data using the [perl language](#) via a built-in perl interpreter. The perl interpreter is only loaded if the module is declared in the configuration. While the [nxlog language](#) is already a powerful framework, it is not intended to be a full featured programming language. For example it does not provide lists, arrays, hashes and other features available in many high-level languages. Perl is widely used for log processing and it comes with a broad set of modules bundled or available from [CPAN](#).

You can sometimes write faster code in C, but you can always write code faster in Perl. Code written in perl is also a lot safer because it is unlikely to crash. Exceptions in the perl code (croak/die) are handled properly and this will only result in an unfinished attempt at executing the log processing code but will not take down the whole nxlog process.

The module will parse the file specified in the [PerlCode](#) directive when nxlog and the module is started. This file should contain one or more methods which can be called from the [Exec](#) directive of any module which wants to do any log processing in perl. See the [example](#) below which illustrates the use of this module.

To access the fields and the event data from the perl code, you need to include the `Log::Nxlog` module. This exports the following methods:

**set\_field\_integer(event, key, value)** Sets the integer value in the field named 'key'.

**set\_field\_string(event, key, value)** Sets the string value in the field named 'key'.

**set\_field\_boolean(event, key, value)** Sets the boolean value in the field named 'key'.

**get\_field(event, key)** Retrieve the value associated with the field named 'key'. The method returns a scalar value if the key exist and the value is defined, otherwise it returns undef.

**delete\_field(event, key)** Delete the value associated with the field named 'key'.

**field\_type(event, key)** Return a string representing the type of the value associated with the field named 'key'.

**field\_names(event)** Return a list of the field names contained in the event data. Can be used to iterate over all the fields.

**log\_debug(msg)** Send the message in the argument to the internal logger on DEBUG loglevel. Does the same as the procedure named `log_debug()` in nxlog.

**log\_info(msg)** Send the message in the argument to the internal logger on INFO loglevel. Does the same as the procedure named `log_info()` in nxlog.

**log\_warning(msg)** Send the message in the argument to the internal logger on WARNING loglevel. Does the same as the procedure named `log_warning()` in nxlog.

**log\_error(msg)** Send the message in the argument to the internal logger on ERROR loglevel. Does the same as the procedure named `log_error()` in nxlog.

You should be able to read the POD documentation contained in `Nxlog.pm` with **perldoc Log::Nxlog**.

#### 6.1.10.1 Configuration

The following directives can be used to configure the xm\_perl module instance:

**PerlCode** This mandatory directive expects a file which contains valid perl code. This file is read and parsed by the perl interpreter. Methods defined in this file can be called with the [call\(\)](#) procedure.



### 6.1.10.2 Functions and procedures exported by xm\_perl

#### 6.1.10.2.1 Procedures exported by xm\_perl

**call(string subroutine);**

**description** Calls the perl subroutine provided in the first argument.

**arguments**

**subroutine** type: **string**

**perl\_call(string subroutine);**

**description** Calls the perl subroutine provided in the first argument.

**arguments**

**subroutine** type: **string**

#### 6.1.10.3 Configuration examples

In this example logs are parsed as syslog then the data is passed to a perl method which does a GeoIP lookup on the source address of the incoming message.

**Example 6.23** Using the built-in perl interpreter

```
<Extension syslog>
  Module  xm_syslog
</Extension>

<Extension perl>
  Module  xm_perl
  PerlCode modules/extension/perl/processlogs.pl
</Extension>

<Input in>
  Module  im_file
  File    'test.log'
  ReadFromLast FALSE
  SavePos FALSE
</Input>

<Output out>
  Module  om_file
  File    'tmp/output'
  # First we parse the input natively from nxlog
  Exec    parse_syslog_bsd();
  # Now call the 'process' subroutine defined in 'processlogs.pl'
  Exec    perl_call("process");
  # You can also invoke this public procedure 'call' in case
  # of multiple xm_perl instances like this:
  # Exec    perl->call("process");
</Output>

<Route 1>
  Path    in => out
</Route>
```

The contents of the `processlogs.pl` perl script is as follows:

```
use strict;
use warnings;

use Carp;
# FindBin is for adding a path to @INC, this not needed normally
use FindBin;
use lib "$FindBin::Bin/../../../../src/modules/extension/perl";

# Without Log::Nxlog you cannot access (read or modify) the event data
# so don't forget this:
use Log::Nxlog;

use Geo::IP;

my $geoip;

BEGIN
{
  # This will be called once when nxlog starts so you can use this to
  # initialize stuff here
  $geoip = Geo::IP->new(GEOIP_MEMORY_CACHE);
}

# this is the method which is invoked from 'Exec' for each event log
sub process
{
  # The event data is passed here when this method is invoked by the module
  my ( $event ) = @_;

  # We look up the county of the sender of the message
  my $msgsrcaddr = Log::Nxlog::get_field($event, 'MessageSourceAddress');
  if ( defined($msgsrcaddr) )
```

## 6.2 Input modules

Input modules are responsible for collecting event log data from various sources. The nxlog core will add a few fields in each input module, see the following section for the list of these.

### 6.2.1 Fields generated by core

The following fields are set by core:

**\$raw\_event** Type **string**

Filled with data received from stream modules (im\_file, im\_tcp, etc).

**\$EventReceivedTime** Type **datetime**

Set to the time when the event is received. The value is not modified if the field already exists.

**\$SourceModuleName** Type **string**

The name of the module instance is stored in this field for input modules. The value is not modified if the field already exists.

**\$SourceModuleType** Type **string**

The type the module instance (such as 'im\_file') is stored in this field for input modules. The value is not modified if the field already exists.

### 6.2.2 DBI (im\_dbi)

FIXME

#### 6.2.2.1 Configuration examples

---

**Example 6.24** Reading from a MySQL database

---

```
<Input dbiin>
  Module im_dbi
  SavePos TRUE
  Driver mysql
  Option host 127.0.0.1
  Option username mysql
  Option password mysql
  Option dbname logdb
</Input>

<Output out>
  Module om_file
  File "tmp/output"
</Output>

<Route 1>
  Path dbiin => out
</Route>
```

---

### 6.2.3 Program (im\_exec)

This module will execute a program or script on startup and will read its standard output.

---

---

**Note**

The executed program or script should not exit because `im_exec` will not restart it. Support for restart will be added in a future version. Restarting can lead to a significant overhead if it occurs frequently, this is the main reason why this feature was left out in order to avoid abusing the module.

---

### 6.2.3.1 Configuration

In addition to the **common module directives**, the following can be used to configure the `im_exec` module instance.

**Command** This directive is mandatory. It specifies the name of the script/program to be executed.

**Arg** This is an optional parameter, multiple can be specified for each argument needed to pass to the **Command**. Note that specifying multiple arguments with one **Arg** directive separated with spaces will not work because the **Command** will receive it as one argument, so you will need to split them up.

**InputType** See the description about **InputType** in the global module config section.

### 6.2.3.2 Configuration examples

---

**Example 6.25** Emulating `im_file`

---

```
<Input input>
  Module im_exec
  Command /usr/bin/tail
  Arg -f
  Arg /var/log/messages
</Input>

<Output fileout>
  Module om_file
  File "tmp/output"
</Output>

<Route 1>
  Path input => fileout
</Route>
```

---

This exact same configuration is not recommended for real use because `im_file` was designed to read log messages from files. This example only demonstrates the use of the `im_exec` module.

## 6.2.4 File (`im_file`)

This module can be used to read log messages from files. The file position can be persistently saved across restarts in order to avoid reading from the beginning again when `nxlog` is restarted. It also supports external rotation tools. When the module cannot read any more data from the file, it checks whether the opened file descriptor belongs to the same filename it opened originally. If the inodes differ, the module assumes the file was moved and reopens its input.

`im_file` uses a 1 second interval to monitor files for new messages. This method was implemented because polling a regular file is not supported on all platforms. If there is no more data to read, the module will sleep for 1 second.

By using wildcards, the module can read multiple files simultaneously and will open new files as they appear. It will also enter newly created directories if recursion is enabled.

---

### 6.2.4.1 Configuration

In addition to the **common module directives**, the following can be used to configure the `im_file` module instance.

**File** This mandatory directive specifies the name of the input file to open. It must be a **string** type **expression**. For relative filenames you should be aware that `nxlog` changes its working directory to `'/'` unless the global `SpoolDir` is set to something else. On Windows systems the directory separator is backslash. For compatibility reasons the forward slash `'/'` character can be also used as the directory separator, but this only works for filenames which don't contain wildcards. If the filename is specified using wildcards, you should use backslash for the directory separator.

Wildcards are supported in filenames only, directory names in the path cannot be wildcarded. Wildcards are not regular expressions, these are patterns commonly used by unix shells to expand filenames which is also known as globbing.

? Matches a single character only.

\* Matches zero or more characters.

\\* Matches the asterisk `'*'` character.

\? Matches the question mark `'?'` character.

[...] Used to specify a single character. If the first character of the class description is `^` or `!`, the sense of the description is reversed. The rest of the class description is a list of single characters or pairs of characters separated by `-`. Any of those characters can have a backslash in front of them, which is ignored; this lets you use the characters `]` and `-` in the character class, as well as `^` and `!` at the beginning.

---

#### Note

The backslash character `'\'` is used to escape the wildcard characters, unfortunately this is the same as the directory separator on Windows. Take this into account when specifying wildcarded filenames on this platform. Lets suppose that we have log files under the directory `C:\test` which need to be monitored. Specifying the wildcard `'C:\test\*.log'` will not match because `'\'` becomes a literal asterisk, thus it is treated as a non-wildcarded filename. For this reason the directory separator needs to be escaped, so the `'C:\test\\*.log'` will match our files. `'C:\\test\\*.log'` will also work. When specifying the filename using double quotes, this would become `"C:\\test\\*.log"` because the backslash is also used as an escape character inside double quoted **string literals**. Filenames on Windows systems are treated case-insensitively. Unix/Linux is case-sensitive.

---

**SavePos** This directive takes a boolean value of `TRUE` or `FALSE` and specifies whether the file position should be saved when `nxlog` exits. The file position will be read from the cache file file upon startup. The file position is saved by default if this directive is not specified in the configuration. Even if `SavePos` is enabled, it can be explicitly turned off with the **NoCache** directive.

**ReadFromLast** This optional directive takes a boolean value. If it is set to `TRUE`, it instructs the module to only read logs which arrived after `nxlog` was started in case the saved position could not be read (for example on first start). When `SavePos` is `TRUE` and a previously saved position value could be read, the module will resume reading from this saved position. If this is `FALSE`, the module will read all logs from the file. This can result in quite a lot of messages which is usually not the expected behaviour. If this directive is not specified, it defaults to `TRUE`.

**Recursive** This directive takes a boolean value of `TRUE` or `FALSE` and specifies whether input files should be searched recursively under subdirectories. The default value is `TRUE`. This option takes effect only if wildcards are used in the filename. For example if `'var/log/*.log'` is specified in the **File** directive, then `'var/log/apache2/access.log'` will also match. Because wildcards in directory names of the path are not supported, this directive makes it possible to read multiple files from different subdirectories with a single `im_file` module instance only.

**PollInterval** This directive specifies in seconds how frequently the module will check for new files and new log entries. If this directive is not specified it defaults to 1 second. Fractional seconds may be specified, i.e. to check twice every second you should set the following: `PollInterval 0.5`

**InputType** See the description about **InputType** in the global module config section.

---

## 6.2.4.2 Functions and procedures exported by im\_file

### 6.2.4.2.1 Functions exported by im\_file

**string** file\_name();

**description** Return the name of the file currently open which the log was read from.

**return type** string

### 6.2.4.3 Configuration examples

---

**Example 6.26** Forwarding logs from a file to a remote host

---

```
<Input in>
  Module im_file
  File  "/var/log/messages"
  SavePos TRUE
</Input>

<Output out>
  Module om_tcp
  Host  192.168.1.1
  Port  514
</Output>

<Route 1>
  Path  in => out
</Route>
```

---

## 6.2.5 Internal (im\_internal)

This module makes it possible to insert internal log messages of nxlog into a route. nxlog generates messages about error conditions, debugging messages, etc. In addition internal messages can be also generated from the nxlog language using the `log_info()`, `log_warning()` and `log_error()` procedure calls.

---

**Note**

Only messages with loglevel INFO and above are supported. Debug messages are ignored due to technical reasons. For debugging purposes the direct logging facility should be used, see the global `LogFile` and `LogLevel` directives.

One must be careful about the use of the im\_internal module because it is easy to cause a message loop. For example consider the situation when the internal messages are sent to a database. If the database is experiencing errors which result in internal error messages then these are again routed to the database and this will trigger further error messages and is easy to see that this will result in a loop. In order to avoid a resource exhaustion, the im\_internal module will drop its messages when the queue of next module in the route is full. It is recommended to always put the im\_internal module instance in a separate route.

---

The im\_internal does not have any module specific configuration directives in addition to the `common module directives`.

---

**Note**

If you require internal messages in syslog format, you need to explicitly convert them with `pm_transformer` or using the `to_syslog_bsd()` procedure of the `xm_syslog` module, because the `$raw_event` field is not generated in syslog format.

---

### 6.2.5.1 Fields generated by im\_internal

The following fields are set by im\_internal:

**\$raw\_event** Type **string**

Will be set to the string passed to the log\_info() and other log() procedures.

**\$Message** Type **string**

Set to the same value as \$raw\_event.

**\$SeverityValue** Type **integer**

Depending on the log level of the internal message, the syslog severity is set to the value corresponding to "debug", "info", "warning", "error" or "critical".

**\$Severity** Type **string**

The severity name of the event.

**\$EventTime** Type **datetime**

Will be set to the current time.

**\$SourceName** Type **string**

Will be set to 'nxlog'.

**\$ProcessID** Type **integer**

The field is filled with the process id of the nxlog process.

**\$Hostname** Type **string**

The hostname where the log is produced

**\$ErrorCode** Type **integer**

If an error is logged resulting from an OS error, this field contains the error number provided by the Apache portable run-time library.

### 6.2.5.2 Configuration examples

---

#### Example 6.27 Forwarding internal messages over syslog udp

---

```
<Extension syslog>
  Module      xm_syslog
</Extension>

<Input internal>
  Module im_internal
</Input>

<Output out>
  Module om_udp
  Host 192.168.1.1
  Port 514
  Exec to_syslog_bsd();
</Output>

<Route internal>
  Path internal => out
</Route>
```

---

### 6.2.6 Kernel (im\_kernel)

This module can collect kernel log messages from the kernel log buffer. Currently this module works on linux only. On Linux the klogctl() system call is used for this purpose. In order to be able to read kernel logs, special privileges are required. For this reason nxlog needs to be started as root. Using the **User** and **Group** global directives nxlog can then drop its root privileges but it will keep the CAP\_SYS\_ADMIN capability to be able to read the kernel log buffer.

---

**Note**

Unfortunately it is not possible to read from the /proc/kmsg pseudo file for an unprivileged process even if the CAP\_SYS\_ADMIN capability is kept. For this reason the /proc/kmsg interface is not supported by the im\_kernel module. The im\_file module should work fine with the /proc/kmsg pseudo file if one wishes to collect kernel logs this way, though this will require nxlog to be running as root.

---

The kernel messages are emitted in the following form.

```
<[0-7]>Some message from the kernel.
```

---

**Note**

Kernel messages are valid BSD syslog messages but do not contain timestamp and hostname fields. These can be parsed with pm\_transformer or using the parse\_syslog\_bsd() procedure of the xm\_syslog module, this will set the timestamp and hostname fields.

---

#### 6.2.6.1 Configuration examples

---

**Example 6.28** Storing raw kernel logs into a file

---

```
# drop privileges after being started as root
User nxlog
Group nxlog

<Input kern>
  Module      im_kernel
</Input>

<Output fileout>
  Module      om_file
  File        "tmp/output"
</Output>

<Route 1>
  Path        kern => fileout
</Route>
```

---

### 6.2.7 Mark (im\_mark)

Mark messages are used to indicate periodic activity in order to be assured that the logger is running in case there are no log messages coming in from other sources.

By default, without specifying any of the module specific directives, a log message is emitted every 30 minutes containing "--MARK --".

---



---

**Note**

If you require mark messages in syslog format, you need to explicitly convert them with `pm_transformer` or using the `to_syslog_bsd()` procedure of the `xm_syslog` module, because the `$raw_event` field is not generated in syslog format.

---

---

**Note**

The functionality of the `im_mark` module can be also achieved using the `Schedule` block with a `log_info("--MARK--")` Exec statement which would insert the messages via the `im_internal` module into a route. Using a single module for this task can simplify and possibly make the configuration easier to understand. Just wanted to point out that "there is more than one way to do it" :)

---

### 6.2.7.1 Configuration

In addition to the `common module directives`, the following can be used to configure the `im_mark` module instance.

**Mark** This optional directive can set the string for the mark message. If not specified, the default is "-- MARK --".

**MarkInterval** This optional directive sets the interval for mark messages in minutes. If not specified, the default value of 30 minutes is used.

### 6.2.7.2 Fields generated by im\_mark

The following fields are set by `im_mark`:

**\$raw\_event** Type `string`

Will be set to "-- MARK --" or the value defined with the Mark configuration directive.

**\$Message** Type `string`

Set to the same value as `$raw_event`.

**\$SeverityValue** Type `integer`

Its value will be set to 6 which is the "info" severity level.

**\$Severity** Type `string`

The severity name of the event.

**\$EventTime** Type `datetime`

Will be set to the current time.

**\$SourceName** Type `string`

Will be set to 'nxlog'.

**\$ProcessID** Type `integer`

The field is filled with the process id of the nxlog process.

---

### 6.2.7.3 Configuration examples

---

**Example 6.29** Using the im\_mark module

---

```
<Input mark>
  Module im_mark
  MarkInterval 1
  Mark ==| MARK |==
</Input>

<Output fileout>
  Module om_file
  File "tmp/output"
</Output>

<Route 1>
  Path mark => fileout
</Route>
```

---

### 6.2.8 MS EventLog for Windows XP/2000/2003 (im\_mseventlog)

This module can be used to collect EventLog messages on Microsoft Windows platforms. The module looks up the available EventLog sources stored under the registry key "SYSTEM\\CurrentControlSet\\Services\\Eventlog" and will poll logs from each of these or only the sources defined with the **Sources** directive.

---

**Note**

Windows Vista, Windows 2008 and later use a new EventLog API which is not backward compatible. Messages in some events produced by sources in this new format cannot be resolved with the old API which is used by this module. If such an event is encountered, a Message similar to the following will be set:

```
The description for EventID XXXX from source SOURCE cannot be read by im_mseventlog ←
because this does not support the newer WIN2008/Vista EventLog API.
```

Though the majority of event messages can be read with this module even on Windows 2008/Vista and later, it is recommended to use the **im\_msvistalog** module instead.

---

**Note**

Strings are stored in dll and executable files and these need to be looked up by the module when reading eventlog messages. If a program (dll/exe) is already uninstalled and cannot be opened to look up the strings in the message, the following message will appear instead:

```
The description for EventID XXXX from source SOURCE cannot be found.
```

---

#### 6.2.8.1 Configuration

In addition to the **common module directives**, the following can be used to configure the im\_mseventlog module instance.

**SavePos** This directive takes a boolean value of TRUE or FALSE and specifies whether the file position should be saved when nxlog exits. The file position will be read from the cache file upon startup. The file position is saved by default if this directive is not specified in the configuration. Even if SavePos is enabled, it can be explicitly turned off with the **NoCache** directive.

---

**ReadFromLast** This optional directive takes a boolean value. If it is set to TRUE, it instructs the module to only read logs which arrived after nxlog was started in case the saved position could not be read (for example on first start). When SavePos is TRUE and a previously saved position value could be read, the module will resume reading from this saved position. If this is FALSE, the module will read all logs from the EventLog. This can result in quite a lot of messages which is usually not the expected behaviour. If this directive is not specified, it defaults to TRUE.

**Sources** This optional directive takes a comma separated list of eventlog file names, such as 'Security, Application', to read only specific eventlog sources. If this directive is not specified, then all available eventlog sources are read (as listed in the registry). This directive should not be confused with the **SourceName** contained within the eventlog and it is not a list of such names. The value of this is stored in the **FileName** field.

**UTF8** This optional directive takes a boolean value. If it is set to TRUE, all strings will be converted to UTF-8 encoding. Internally this calls the **convert\_fields** procedure. The **xm\_charconv** module must be loaded for the character set conversion to work. If this UTF8 directive is not defined, it defaults to TRUE, but conversion will only occur if the **xm\_charconv** module is loaded, otherwise strings will be in the local codepage.

### 6.2.8.2 Fields generated by im\_mseventlog

The following fields are set by im\_mseventlog:

**\$raw\_event** Type **string**  
Contains the timestamp, hostname, severity and message from the event

**\$Message** Type **string**  
Contains the message from the event

**\$EventTime** Type **datetime**  
Will be set to the TimeGenerated field of the EventRecord.

**\$EventTimeWritten** Type **datetime**  
Will be set to the TimeWritten field of the EventRecord.

**\$Hostname** Type **string**  
The host or computer name field of the EventRecord.

**\$SourceName** Type **string**  
The event source which produced the event, this is the subsystem or application name.

**\$EventID** Type **integer**  
The event id of the EventRecord.

**\$CategoryNumber** Type **integer**  
The category number, stored as Category in the EventRecord.

**\$Category** Type **string**  
The category name resolved from CategoryNumber.

**\$FileName** Type **string**  
The logfile source (e.g. Security, Application) of the event.

**\$AccountName** Type **string**  
The username associated with the event.

**\$AccountType** Type **string**  
The type of the account. Possible values are: User, Group, Domain, Alias, Well Known Group, Deleted Account, Invalid, Unknown, Computer.

**\$Domain** Type **string**  
The domain name of the user.

**\$SeverityValue** Type **integer**  
Normalized severity number of the event.

**\$Severity**   Type   string

Normalized severity name of the event.

**\$EventType**   Type   string

The type of the event which is a string describing the severity. It takes the following values: "ERROR", "AUDIT\_FAILURE", "AUDIT\_SUCCESS", "INFO", "WARNING", "UNKNOWN"

**\$RecordNumber**   Type   integer

The number of the event record.

### 6.2.8.3 Configuration examples

---

**Example 6.30** Forwarding EventLogs from a windows machine to a remote host

---

```
<Input in>
  Module      im_mseventlog
</Input>

<Output out>
  Module      om_tcp
  Host        192.168.1.1
  Port        514
</Output>

<Route 1>
  Path        in => out
</Route>
```

---

### 6.2.9 MS EventLog for Windows 2008/Vista and later (im\_msvistalog)

This module can be used to collect EventLog messages on Microsoft Windows platforms which support the newer EventLog API (also known as the Crimson Eventlog subsystem), namely Windows 2008/Vista and later. You can refer to the official Microsoft documentation about [Event Logs](#). The module supports reading both Windows Logs and Applications and Services Logs. It looks up the available EventLog sources stored under the registry key "SYSTEM\CurrentControlSet\Services\Eventlog" and "SOFTWARE\Microsoft\Windows\CurrentVersion\WINEVT\Channels" and monitors logs in each of these unless the **Query** and the **Channel** directives are explicitly defined.

---

**Note**

This module will not work on Windows 2003 and earlier because Windows Vista, Windows 2008 and later use a new EventLog API which is not available in earlier Windows versions. If you need to collect EventLog messages on these platforms, you should use the **im\_mseventlog** module instead.

---

---

**Note**

The Windows EventLog subsystem does not support subscriptions to Debug and Analytic channels, thus it is not possible to collect these type of events with this module.

---

In addition to the standard set of **fields** which are listed under the System section, event providers can define their own additional schema which enables logging additional data under the EventData section. The Security log makes use of this new feature and such additional fields can be seen as in the following XML snippet:

```
<EventData>
  <Data Name="SubjectUserSid">S-1-5-18</Data>
  <Data Name="SubjectUserName">WIN-OUNNPISDHIG$</Data>
```

---

```
<Data Name="SubjectDomainName">WORKGROUP</Data>
<Data Name="SubjectLogonId">0x3e7</Data>
<Data Name="TargetUserSid">S-1-5-18</Data>
<Data Name="TargetUserName">SYSTEM</Data>
<Data Name="TargetDomainName">NT AUTHORITY</Data>
<Data Name="TargetLogonId">0x3e7</Data>
<Data Name="LogonType">5</Data>
<Data Name="LogonProcessName">Advapi</Data>
<Data Name="AuthenticationPackageName">Negotiate</Data>
<Data Name="WorkstationName" />
<Data Name="LogonGuid">{00000000-0000-0000-0000-000000000000}</Data>
<Data Name="TransmittedServices">-</Data>
<Data Name="LmPackageName">-</Data>
<Data Name="KeyLength">0</Data>
<Data Name="ProcessId">0x1dc</Data>
<Data Name="ProcessName">C:\Windows\System32\services.exe</Data>
<Data Name="IpAddress">-</Data>
<Data Name="IpPort">-</Data>
</EventData>
```

nxlog can extract this data when fields are logged using this schema. The values will be available in the fields of the internal nxlog log structure. This is especially useful because there is no need to write pattern matching rules to extract this data from the message. These fields can be used in filtering rules, writing them into SQL tables or to trigger actions. Consider the following example which filters using the **Exec** directive:

```
<Input in>
  Module      im_msvistalog
  Exec        if ($TargetUserName == 'SYSTEM') OR ($EventType == 'VERBOSE') drop();
</Input>
```

### 6.2.9.1 Configuration

In addition to the **common module directives**, the following can be used to configure the im\_msvistalog module instance.

**SavePos** This directive takes a boolean value of TRUE or FALSE and specifies whether the file position should be saved when nxlog exits. The file position will be read from the cache file upon startup. The file position is saved by default if this directive is not specified in the configuration. Even if SavePos is enabled, it can be explicitly turned off with the **NoCache** directive.

**ReadFromLast** This optional directive takes a boolean value. If it is set to TRUE, it instructs the module to only read logs which arrived after nxlog was started in case the saved position could not be read (for example on first start). When SavePos is TRUE and a previously saved position value could be read, the module will resume reading from this saved position. If this is FALSE, the module will read all logs from the EventLog. This can result in quite a lot of messages which is usually not the expected behaviour. If this directive is not specified, it defaults to TRUE.

**Query** This directive specifies the query if one wishes to pull only specific eventlog sources. See the MSDN docs about **Event Selection**. Note that this directive needs a single-line parameter, so multi-line query XML should be specified with line continuation marks (\) as in the following example:

```
Query <QueryList>                                     \
  <Query Id='1'>                                       \
    <Select Path='Security'>*[Security/Level=4]</Select> \
  </Query>                                             \
</QueryList>
```

When the Query contains an XPath style expression, the **Channel** must also be specified. Otherwise if an XML Query is specified, the **Channel** should not be used.

**Channel** The name of the Channel to query. If not specified, the module will read from all sources defined in the registry. See the MSDN docs about **Event Selection**.

### 6.2.9.2 Fields generated by im\_msvistalog

The following fields are set by im\_msvistalog:

- \$raw\_event** Type **string**  
Contains the EventTime, Hostname, Severity EventID and Message from the event.
- \$Message** Type **string**  
Contains the message from the event.
- \$EventTime** Type **datetime**  
Will be set to the EvtSystemTimeCreated field.
- \$Hostname** Type **string**  
Contains the EvtSystemComputer field.
- \$SourceName** Type **string**  
The event source which produced the event, this is the EvtSystemProviderName field.
- \$EventID** Type **integer**  
The event id as in EvtSystemEventID.
- \$Task** Type **integer**  
The task number as in EvtSystemTask.
- \$Category** Type **string**  
The category name resolved from Task.
- \$Keywords** Type **integer**  
The value of the Keywords field from EvtSystemKeywords.
- \$Channel** Type **string**  
The Channel (e.g. Security, Application) of the event source.
- \$AccountName** Type **string**  
The username associated with the event.
- \$AccountType** Type **string**  
The type of the account. Possible values are: User, Group, Domain, Alias, Well Known Group, Deleted Account, Invalid, Unknown, Computer.
- \$Domain** Type **string**  
The domain name of the user.
- \$UserID** Type **string**  
The SID which resolves to AccountName, stored in EvtSystemUserID.
- \$SeverityValue** Type **integer**  
Normalized severity number of the event.
- \$Severity** Type **string**  
Normalized severity name of the event (CRITICAL|ERROR|WARNING|INFO|DEBUG).
- \$EventType** Type **string**  
The type of the event which is a string describing the severity. This is translated to its string representation from EvtSystemLevel. It takes the following values: "CRITICAL", "ERROR", "AUDIT\_FAILURE", "AUDIT\_SUCCESS", "INFO", "WARNING", "VERBOSE"
- \$ProviderGuid** Type **string**  
The GUI of the event's provider as stored in EvtSystemProviderGuid. This corresponds to the name of the provider stored in the SourceName field.
-

**\$Version** Type **integer**  
The Version number of the event as in EvtSystemVersion.

**\$OpcodeValue** Type **integer**  
The Opcode number of the event as in EvtSystemOpcode.

**\$Opcode** Type **string**  
The opcode string resolved from OpcodeValue.

**\$ActivityID** Type **string**  
The ActivityID as stored in EvtSystemActivityID.

**\$RelatedActivityID** Type **string**  
The RelatedActivityID as stored in EvtSystemRelatedActivityID.

**\$ProcessID** Type **integer**  
The process identifier of the event producer as in EvtSystemProcessID.

**\$ThreadID** Type **integer**  
The thread identifier of the event producer as in EvtSystemThreadID.

**\$RecordNumber** Type **integer**  
The number of the event record.

### 6.2.9.3 Configuration examples

---

#### Example 6.31 Forwarding EventLogs from a windows machine to a remote host

---

```
<Input in>
  Module      im_msvistalog
  ReadFromLast TRUE
</Input>

<Output out>
  Module      om_tcp
  Host        192.168.1.1
  Port        514
</Output>

<Route 1>
  Path        in => out
</Route>
```

---

### 6.2.10 Null (im\_null)

This module does not generate any input, so basically it does nothing. Yet it can be useful for creating a dummy route, testing purposes, and it can have **Scheduled** nxlog code execution as well, so it is not completely useless. This module does not have any module specific configuration directives. See [this example](#) for usage.

### 6.2.11 TLS/SSL (im\_ssl)

The im\_ssl module provides an SSL/TLS transport using the OpenSSL library beneath the surface. It behaves similarly to the **im\_tcp** module, except that an SSL handshake is performed at connection time and the data is sent over a secure channel. Because log messages transferred over plain TCP can be eavasdropped or even altered with a man-in-the-middle attack, using the im\_ssl module provides a secure log message transport.

---

### 6.2.11.1 Configuration

In addition to the **common module directives**, the following can be used to configure the `im_ssl` module instance.

**Host** This specifies the IP address or a dns hostname which the module should listen on to accept connections.

**Port** This specifies the port number which the module will listen on for incoming connections.

**CertFile** This specifies the path of the certificate file to be used in the SSL handshake.

**CertKeyFile** This specifies the path of the certificate key file to be used in the SSL handshake.

**KeyPass** Optional password of the certificate key file defined in **CertKeyFile**. For passwordless private keys the directive is not needed.

**CAFile** This specifies the path of the certificate of the CA which will be used to check the certificate of the remote socket against.

**CADir** This specifies the path of CA certificates which will be used to check the certificate of the remote socket against. The cert file names in this directory must be in the OpenSSL hashed format.

**CRLFile** This specifies the path of the certificate revocation list (CRL) which will be used to check the certificate of the remote socket against.

**CRLDir** This specifies the path of certificate revocation lists (CRLs) which will be used to check the certificate of the remote socket against. The file names in this directory must be in the OpenSSL hashed format.

**RequireCert** This takes a boolean value of TRUE or FALSE and specifies whether the remote must present a certificate. If set to TRUE and there is no certificate presented during the handshake of the accepted connection, the connection will be refused. The default value is TRUE if this directive is not specified, meaning that all connections must use a certificate by default.

**AllowUntrusted** This takes a boolean value of TRUE or FALSE and specifies whether the remote connection should be allowed without certificate verification. If set to TRUE the remote will be able to connect with unknown and self-signed certificates. The default value is FALSE if this directive is not specified, meaning that all connections must present a trusted certificate by default.

**InputType** See the description about **InputType** in the global module config section.

### 6.2.11.2 Fields generated by `im_ssl`

The following fields are set by `im_ssl`:

**\$raw\_event** Type **string**  
Will be set to the string received.

**\$MessageSourceAddress** Type **string**  
Set to the IP address of the remote host.

---



### 6.2.11.3 Configuration examples

---

**Example 6.32** Reading binary data forwarded from another nxlog agent

---

```
<Input ssl>
  Module im_ssl
  Host localhost
  Port 23456
  CAFile %CERTDIR%/ca.pem
  CertFile %CERTDIR%/client-cert.pem
  CertKeyFile %CERTDIR%/client-key.pem
  KeyPass secret
  InputType Binary
</Input>

<Output fileout>
  Module om_file
  File "tmp/output"
</Output>

<Route 1>
  Path ssl => fileout
</Route>
```

### 6.2.12 TCP (im\_tcp)

This module accepts TCP connections on the address and port specified in the configuration. It can handle multiple simultaneous connections. The TCP transfer protocol provides more reliable log transmission than UDP. If security is a concern, consider using the **im\_ssl** module instead.

---

**Note**

There is no access control built in the module. If you need to deny some hosts connecting to the module's TCP port, you should use appropriate firewall rules for this purpose.

---

#### 6.2.12.1 Configuration

In addition to the **common module directives**, the following can be used to configure the im\_tcp module instance.

**Host** This specifies the IP address or a dns hostname which the module should listen on to accept connections.

---

**Note**

Because of security reasons the default listen address is *localhost* if this directive is not specified (the *localhost* loopback address is not accessible from the outside). You will most probably want to send logs from remote hosts, so make sure that the address specified here is accessible. The any address *0.0.0.0* is commonly used here.

---

**Port** This specifies the port number which the module will listen on for incoming connections. The default port is 514 if this directive is not specified.

**InputType** See the description about **InputType** in the global module config section.

---

### 6.2.12.2 Fields generated by im\_tcp

The following fields are set by im\_tcp:

**\$raw\_event** Type **string**  
Will be set to the string received.

**\$MessageSourceAddress** Type **string**  
Set to the IP address of the remote host.

### 6.2.12.3 Configuration examples

---

**Example 6.33** Using the im\_tcp module

---

```
<Input in>
  Module im_tcp
  Host 0.0.0.0
  Port 1514
</Input>

<Output out>
  Module om_file
  File "tmp/output"
</Output>

<Route r>
  Path in => out
</Route>
```

---

### 6.2.13 UDP (im\_udp)

This module accepts UDP datagrams on the address and port specified in the configuration. UDP is the transport protocol of the old BSD syslog standard as described in RFC 3164, so this module can be particularly useful to receive such messages from older devices which do not support other transports.

---

**Note**

There is no access control built in the module. If you need to deny some hosts sending logs to the module's UDP port, you should use appropriate firewall rules for this purpose.

---

---

**Note**

UDP packets can be dropped by the operating system because the protocol does not guarantee reliable message delivery. It is recommended to use the **tcp** or **ssl** transport modules instead if message loss is a concern.

Though nxlog **was designed** to minimize message loss even in the case of UDP, adjusting the kernel buffers could also help in avoiding UDP message loss on a loaded system. The **Priority** directive in the route block can also help in this situation.

---

For parsing syslog messages, take a look at the **pm\_transformer** module or the **parse\_syslog\_bsd()** procedure of the **xm\_syslog**.

---

### 6.2.13.1 Configuration

In addition to the **common module directives**, the following can be used to configure the `im_udp` module instance.

**Host** This specifies the IP address or a dns hostname which the module should listen on to accept connections. The default address is "localhost" if this is not specified.

**Port** This specifies the port number which the module will listen on for incoming connections. The default port is 514 if this directive is not specified.

**InputType** See the description about **InputType** in the global module config section.

### 6.2.13.2 Fields generated by im\_udp

The following fields are set by `im_udp`:

**\$raw\_event** Type **string**  
Will be set to the string received.

**\$MessageSourceAddress** Type **string**  
Set to the IP address of the remote host.

### 6.2.13.3 Configuration examples

---

**Example 6.34** Using the `im_udp` module

---

```
<Input in>
  Module im_udp
  Host 192.168.1.1
  Port 514
</Input>

<Output out>
  Module om_file
  File "tmp/output"
</Output>

<Route r>
  Path in => out
</Route>
```

---

## 6.2.14 Unix Domain Socket (im\_uds)

This module allows log messages to be received over a unix domain socket. Traditionally unix systems have a socket, typically `/dev/log` used by the system logger to accept messages from. Applications wishing to send messages to the system log use the `syslog(3)` system call.

---

**Note**

This module supports `SOCK_DGRAM` type sockets only. `SOCK_STREAM` type sockets will be supported in the future.

---

For parsing syslog messages, take a look at the **pm\_transformer** module or the **parse\_syslog\_bsd()** procedure of the **xm\_syslog**.

---

### 6.2.14.1 Configuration

In addition to the **common module directives**, the following can be used to configure the `im_uds` module instance.

**UDS** This specifies the path of the unix domain socket. The default is `/dev/log` if this is not specified.

**InputType** See the description about **InputType** in the global module config section. This defaults to `dgram` if not specified because unix domain sockets are `SOCK_DGRAM` type on Linux and the module does not yet support `SOCK_STREAM` sockets.

### 6.2.14.2 Configuration examples

---

**Example 6.35** Using the `im_uds` module

---

```
<Input unix>
  Module im_uds
  uds    /dev/log
</Input>

<Output out>
  Module om_file
  File   "/var/log/messages"
</Output>

<Route 1>
  Path unix => out
</Route>
```

---

## 6.3 Processor modules

### 6.3.1 Blocker (`pm_blocker`)

This module can block log messages and can be used to simulate when a route is blocked. When the module blocks the data flow, log messages are first accumulated in the buffers, and then the flow-control mechanism pauses the input modules. Using the `block()` procedure it is possible to programatically stop or resume the data flow. It can be useful for real-world scenarios as well as testing. See the examples below. When the module starts, the blocking mode is disabled by default, i.e. it operate just like `pm_null` would.

#### 6.3.1.1 Functions and procedures exported by `pm_blocker`

##### 6.3.1.1.1 Functions exported by `pm_blocker`

**boolean** `is_blocking()`;

**description** Return TRUE if the module is currently blocking the data flow, FALSE otherwise.

**return type** **boolean**

---

### 6.3.1.1.2 Procedures exported by pm\_blocker

**block(boolean mode) ;**

**description** When mode is TRUE, the module will block. You should call block(FALSE) from a schedule block or another module, otherwise it might not get invoked if the queue is already full.

**arguments**

**mode** type: **boolean**

### 6.3.1.2 Configuration examples

---

#### Example 6.36 Using the pm\_blocker module

In this example we collect messages received over UDP and forward it to another host via TCP. The log data is forwarded during non-working hours between 19 pm and 8 am. During the other half of the day data is buffered on the disk to be sent out only after 19 pm.

```
<Input in>
  Module im_udp
  Host 0.0.0.0
  Port 1514
</Input>

<Processor buffer>
  Module pm_buffer
  # 100Mb disk buffer
  MaxSize 102400
  Type disk
</Processor>

<Processor blocker>
  Module pm_blocker
  <Schedule>
    When 0 8 * * *
    Exec blocker->block(TRUE);
  </Schedule>
  <Schedule>
    When 0 19 * * *
    Exec blocker->block(FALSE);
  </Schedule>
</Processor>

<Output out>
  Module om_tcp
  Host 192.168.1.1
  Port 1514
</Output>

<Route 1>
  Path in => buffer => blocker => out
</Route>
```

---

### 6.3.2 Buffer (pm\_buffer)

Messages received over UDP may be dropped by the operating system unless packets are read from the message buffer fast enough. Some logging subsystems using a small circular buffer can also overwrite logs old logs in the buffer if it is not read, thus

---

there is a chance of missing important log data. Such situations can lead to **dropped or lost messages** and other problems where buffering can help.

The pm\_buffer module supports disk and memory based log message buffering. If both are required, multiple pm\_buffer instances can be used with different settings. Because a memory buffer can be faster, though its size is limited, combining memory and disk based buffering can be a good idea in case buffering is frequently used.

The disk based buffering mode stores the log message data in chunks. When all the data is successfully forwarded from a chunk, it is then deleted in order to save disk space.

---

#### Note

Using pm\_buffer is only recommended when there is a chance of message loss. The built-in flow-control in nxlog ensures that messages will not be read by the input module until the output side can send/store/forward. When reading from files (with **im\_file**) or the Windows Eventlog (with **im\_mseventlog** or **im\_msvistalog**) it is rarely necessary to use the pm\_buffer module unless there is a chance of log rotation (and thus a possibility of missing some data) while the output module (e.g. TCP or SSL) is being blocked.

---

### 6.3.2.1 Configuration

In addition to the **common module directives**, the following can be used to configure the pm\_buffer module instance.

**MaxSize** Specifies the size of the buffer in kilobytes. This parameter is mandatory.

**WarnLimit** Specifies an optional limit smaller than **MaxSize** which will trigger a warning message when reached. The log message will not be emitted again until the buffer size drops to half of WarnLimit and reaches it again in order to protect against a warning message flood.

**Type** Type can be either 'Mem' or 'Disk' to select memory or disk based buffering respectively.

**Directory** Name of the directory used to store the disk buffer file chunks. This is only valid with **Type** set to 'Disk' mode.

### 6.3.2.2 Functions and procedures exported by pm\_buffer

#### 6.3.2.2.1 Functions exported by pm\_buffer

```
integer buffer_size();
```

**description** Return the size of the memory buffer in bytes.

**return type** **integer**

```
integer buffer_count();
```

**description** Return the number of log messages held in the memory buffer.

**return type** **integer**

---

### 6.3.2.3 Configuration examples

---

**Example 6.37** Using a memory buffer to protect against udp message loss

---

```
<Input udp>
  Module im_udp
  Host 0.0.0.0
  Port 514
</Input>

<Processor buffer>
  Module pm_buffer
  # 1Mb buffer
  MaxSize 1024
  Type Mem
  # warn at 512k
  WarnLimit 512
</Processor>

<Output tcp>
  Module om_tcp
  Host 192.168.1.1
  Port 1514
</Output>

<Route 1>
  Path udp => buffer => tcp
</Route>
```

---

### 6.3.3 Filter (pm\_filter)

This is a simple module which forwards log messages if the specified condition is TRUE.

---

**Note**

This module has been obsoleted by the nxlog language because filtering is now possible in any module using the **drop()** procedure conditionally in the **Exec** directive.

---

**Example 6.38** Dropping messages conditionally

---

```
if $raw_event =~ /^Debug/ drop();
```

---

#### 6.3.3.1 Configuration

In addition to the **common module directives**, the following can be used to configure the pm\_filter module instance.

**Condition** This mandatory directive takes an expression as argument which must evaluate to a **boolean** value. If the expression does not evaluate to TRUE, the log message is discarded.

---

### 6.3.3.2 Configuration examples

---

**Example 6.39** Filtering messages

---

```
<Input unix>
  Module im_uds
  uds    /dev/log
</Input>

<Processor filter>
  Module pm_filter
  Condition $raw_event =~ /failed/ or $raw_event =~ /error/
</Processor>

<Output out>
  Module om_file
  File    "/var/log/error"
</Output>

<Route 1>
  Path unix => filter => out
</Route>
```

---

### 6.3.4 Message deduplicator (pm\_norepeat)

This module can be used to filter out repeating messages. Similarly to syslog daemons, this module checks the previous message against the current. If they match, the current message is dropped. The module waits one second for duplicated messages to arrive. If duplicates are detected, the first message is forwarded, the rest is dropped and a message containing "last message repeated X times" is sent instead.

#### 6.3.4.1 Configuration

In addition to the **common module directives**, the following can be used to configure the pm\_norepeat module instance.

**CheckFields** This optional directive takes a comma separated list of field names which are used to compare log messages. Only the fields listed here are compared, the others are ignored. For example the 'EventTime' field will be different in repeating messages, so this field should not be used in the comparison. If this directive is not specified, the default field to be checked is 'Message'.

#### 6.3.4.2 Fields generated by pm\_norepeat

The following fields are set by pm\_norepeat:

**\$raw\_event** Type **string**  
Will be set to "last message repeated X times".

**\$Message** Type **string**  
Set to the same value as \$raw\_event.

**\$SeverityValue** Type **integer**  
Its value will be set to 6 which is the "info" severity level.

**\$Severity** Type **string**  
The severity name of the event.

---



**\$EventTime** Type **datetime**

Will be set to the time of the last event or the current time if EventTime was not present in the last event.

**\$SourceName** Type **string**

Will be set to 'nxlog'.

**\$ProcessID** Type **integer**

The field is filled with the process id of the nxlog process.

### 6.3.4.3 Configuration examples

---

**Example 6.40** Filtering out duplicated messages

---

```
<Input in>
  Module im_uds
  UDS    /dev/log
</Input>

<Processor norepeat>
  Module pm_norepeat
  CheckFields Hostname, SourceName, Message
</Processor>

<Output out>
  Module om_file
  File    "/var/log/messages"
</Output>

<Route 1>
  Path in => norepeat => out
</Route>
```

---

### 6.3.5 Null (pm\_null)

This module does not do any special processing, so basically it does nothing. Though the **Exec** and the **Schedule** directive is available in this module just like in any other, thus it can be quite useful. This module does not have any module specific configuration directives. See [this example](#) for usage.

### 6.3.6 Pattern matcher (pm\_pattern)

This module makes it possible to execute pattern matching efficiently using a pattern database file in XML format. Using this module is more efficient than having nxlog regular expression rules listed in **Exec** directives, because the pm\_pattern module was designed in such a way that patterns need not to be matched linearly. In addition, the module does an automatic on-the-fly pattern reordering internally for further speed improvements and it has a feature which can be used to tag messages with additional fields useful for message classification. See the [Pattern matching and message classification](#) section for additional examples.

Regular expressions are the most widely used in pattern matching. Unfortunately using a large number of regular expression based patterns does not scale well, because these need to be evaluated linearly. There are other techniques such as the radix tree which solve the linearity problem, the drawback is that usually these require a special syntax for specifying patterns which users must learn. If the log message is already parsed and is not treated as single line of message, then it is possible to process only a subset of the patterns which partially solves the linearity problem. With the other performance improvement tricks employed within the pm\_pattern module, its speed can compare to the other techniques such as a radix tree based pattern matcher. Yet the pm\_pattern module can keep using regular expressions which all programmers and system administrators are familiar with and this also provides an easy migration of regexp patterns from other tools and already existing patterns.

---

Traditionally pattern matching on log messages has employed a technique where the log message was one string and the pattern (regular expression or radix tree based pattern) was executed against it. To match patterns against logs which contain structured data (such as the Windows EventLog), this structured data (the fields of the log) must be converted to a single string. This is a simple but inefficient method used by many tools.

The nxlog patterns defined in the XML pattern database file can contain more than one field, this allows multi-dimensional pattern matching. Thus with nxlog's pm\_pattern module there is no need to convert all fields into a single string as it can work with multiple fields.

When the pm\_pattern module finds a matching pattern, the *PatternID* and *PatternName* fields are set on the log message. These can be used later in conditional processing and correlation rules for example.

---

**Note**

The pm\_pattern module does not process all patterns. It exits after the first matching pattern is found. This means that at most one pattern can match a log message. You should avoid writing a pattern to be used with pm\_pattern which can match a subset of logs that match another pattern. For example if you have two regular expression patterns `^\d+` and `^\d\d`, the second may be never matched because of the first. The internal order of patterns and pattern groups is changed dynamically by pm\_pattern. Those patterns are placed and tried first which have the highest match count. Reasons for this operation mode are:

- Performance optimization,
- Setting the value of *\$PatternID* would be problematic with multiple values because the language does not support arrays.

If you want a strictly linearly executing pattern matcher, you should use the **Exec** directive and write your rules there.

---

#### 6.3.6.1 Configuration

In addition to the **common module directives**, the following can be used to configure the pm\_pattern module instance.

**PatternFile** This mandatory directive specifies the name of the **pattern database file**.



### 6.3.6.2 Pattern database file

---

**Example 6.41** A simple pattern database

---

This pattern database contains two patterns to match ssh authentication messages. Both extract *AuthMethod*, *AccountName* and *SourceIP4Address* from the log message when the pattern matches the log. Additionally *TaxonomyStatus* and *TaxonomyAction* are set. The second pattern utilizes the **Exec** block which is evaluated when the pattern matches.

---

**Note**

For this pattern to work, the logs must be parsed with `parse_syslog()` prior to feeding it to the `pm_pattern` module because it uses the *SourceName* and *Message* fields.

---

```
<?xml version='1.0' encoding='UTF-8'?>
<patterndb>
  <created>2010-01-01 01:02:03</created>
  <version>42</version>

  <group>
    <name>ssh</name>
    <id>42</id>

    <pattern>
      <id>1</id>
      <name>ssh auth success</name>

      <matchfield>
        <name>SourceName</name>
        <type>exact</type>
        <value>sshd</value>
      </matchfield>

      <matchfield>
        <name>Message</name>
        <type>regex</type>
        <!-- Accepted publickey for nxlogfan from 192.168.1.1 port 4242 ssh2 -->
        <value>^Accepted (\S+) for (\S+) from (\S+) port \d+ ssh2</value>
        <capturedfield>
          <name>AuthMethod</name>
          <type>string</type>
        </capturedfield>
        <capturedfield>
          <name>AccountName</name>
          <type>string</type>
        </capturedfield>
        <capturedfield>
          <name>SourceIP4Address</name>
          <type>string</type>
        </capturedfield>
      </matchfield>

      <set>
        <field>
          <name>TaxonomyStatus</name>
          <value>success</value>
          <type>string</type>
        </field>
        <field>
          <name>TaxonomyAction</name>
          <value>authenticate</value>
          <type>string</type>
        </field>
      </set>
    </pattern>

    <pattern>
      <id>2</id>
      <name>ssh auth failure</name>
```

### 6.3.6.3 Fields generated by pm\_pattern

The following fields are set by pm\_pattern:

- \$PatternID** Type **integer**  
Set to the id number of the pattern which matched the message.
- \$PatternName** Type **string**  
Set to the name of the pattern which matched the message.

### 6.3.6.4 Configuration examples

---

**Example 6.42** Using the pm\_pattern module

---

```
<Extension syslog>
  Module      xm_syslog
</Extension>

<Input in>
  Module im_uds
  UDS    /dev/log
  Exec   parse_syslog_bsd();
</Input>

<Processor pattern>
  Module pm_pattern
  PatternFile /var/lib/nxlog/patterndb.xml
</Processor>

<Output out>
  Module om_file
  File   "/var/log/messages"
</Output>

<Route 1>
  Path   in => pattern => out
</Route>
```

---

## 6.3.7 Message format converter (pm\_transformer)

The pm\_transformer module provides parsers for syslog (both legacy and the newer IETF standard), CSV, JSON and XML formatted data and can also convert between. This module is now obsoleted by the functions and procedures provided by the following modules:

**xm\_syslog**  
**xm\_csv**  
**xm\_json**  
**xm\_xml**

Though using this pm\_transformer module can be slightly faster than calling these procedures from an **Exec** directive.

### 6.3.7.1 Configuration

In addition to the **common module directives**, the following can be used to configure the pm\_transformer module instance.

**InputFormat** This directive specifies the input format of the \$raw\_event field so that it is further parsed into fields. If this directive is not specified, no parsing will be done.

---

**syslog\_rfc3164** Input is parsed in bsd syslog format as defined by RFC 3164. This does the same as the [parse\\_syslog\\_bsd\(\)](#) procedure.

**syslog\_bsd** Same as [syslog\\_rfc3164](#).

**syslog\_rfc5424** Input is parsed in IETF syslog format as defined by RFC 5424. This does the same as the [parse\\_syslog\\_ietf\(\)](#) procedure.

**syslog\_ietf** Same as [syslog\\_rfc5424](#).

**CSV** Input is parsed as a comma separated list of values. See [xm\\_csv](#) for similar functionality. The input fields must be defined by [CSVInputFields](#)

**XML** Input is parsed as XML. This does the same as the [parse\\_xml\(\)](#) procedure.

**JSON** Input is parsed as JSON. This does the same as the [parse\\_json\(\)](#) procedure.

**CSVInputFields** This is a comma separated list of fields which will be filled from the input parsed. The field names must have the dollar sign "\$" prepended.

**CSVInputFieldTypes** This optional directive specifies the list of types corresponding to the field names defined in [CSVInputFields](#). If specified, the number of types must match the number of field names specified with [CSVInputFields](#). If this directive is omitted, all fields will be stored as [strings](#). This directive has no effect on the fields-to-csv conversion.

**OutputFormat** This directive specifies the output transformation. If this directive is not specified, fields are not converted and \$raw\_event is left unmodified.

**syslog\_rfc3164** Output in \$raw\_event is formatted in bsd syslog format as defined by RFC 3164. This does the same as the [to\\_syslog\\_bsd\(\)](#) procedure.

**syslog\_bsd** Same as [syslog\\_rfc3164](#).

**syslog\_rfc5424** Output in \$raw\_event is formatted in IETF syslog format as defined by RFC 5424. This does the same as the [to\\_syslog\\_ietf\(\)](#) procedure.

**syslog\_ietf** Same as [syslog\\_rfc5424](#).

**syslog\_snare** Output in \$raw\_event is formatted in SNARE syslog format. This does the same as the [to\\_syslog\\_snare\(\)](#) procedure. This is to be used in conjunction with the [im\\_mseventlog](#) or [im\\_msvistalog](#) module to produce an output compatible with [Snare Agent for Windows](#).

**CSV** Output in \$raw\_event is formatted as a comma separated list of values. See [xm\\_csv](#) for similar functionality.

**XML** Output in \$raw\_event is formatted in XML. This does the same as the [to\\_xml\(\)](#) procedure.

**JSON** Output in \$raw\_event is formatted as JSON. This does the same as the [to\\_json\(\)](#) procedure.

**CSVOutputFields** This is a comma separated list of message fields which are placed in the CSV lines. The field names must have the dollar sign "\$" prepended.

### 6.3.7.2 Configuration examples

---

**Example 6.43** Using the pm\_transformer module

---

```
<Extension syslog>
  Module      xm_syslog
</Extension>

<Input filein>
  Module im_file
  File "tmp/input"
</Input>

<Processor transformer>
  Module pm_transformer
  InputFormat syslog_rfc3164
#   OutputFormat syslog_rfc3164
  OutputFormat csv
  CSVOutputFields $facility, $severity, $timestamp, $hostname, $application, $pid, ↵
    $message
</Processor>

<Output fileout>
  Module om_file
  File "tmp/output"
</Output>

<Route 1>
  Path filein => transformer => fileout
</Route>
```

---

## 6.4 Output modules

### 6.4.1 Blocker (om\_blocker)

This module serves testing purposes mostly. It will block log messages in order to simulate a blocked route. This can easily create a similar situation when a network transport output module such as **om\_tcp** blocks because of a network problem. See the **sleep()** procedure which can delay log message output and can also help testing similar situations.

### 6.4.1.1 Configuration examples

---

**Example 6.44** Testing buffering with the om\_blocker module

---

```
<Input in>
  Module im_udp
  Host 0.0.0.0
  Port 1514
</Input>

<Processor buffer>
  Module pm_buffer
  # 100Mb disk buffer
  MaxSize 102400
  Type disk
</Processor>

<Processor blocker>
  Module pm_blocker
  <Schedule>
    When 0 8 * * *
    Exec blocker->block(TRUE);
  </Schedule>
  <Schedule>
    When 0 19 * * *
    Exec blocker->block(FALSE);
  </Schedule>
</Processor>

<Output out>
  Module om_tcp
  Host 192.168.1.1
  Port 1514
</Output>

<Route 1>
  Path in => buffer => blocker => out
</Route>
```

---

### 6.4.2 DBI (om\_dbi)

The om\_dbi module utilizes the **libdbi** database abstraction library to enable storing logs in various database engines supported by the library such as MySQL, PostgreSQL, MSSQL, Sybase, Oracle, SQLite, Firebird. This module makes it possible to insert logs directly into an SQL database. You can specify the INSERT statement which will be executed for each log, this enables inserts into any table schema.

---

**Note**

libdbi needs **drivers** to be able to access the database engines. These are in the libdbd-\* packages on Debian and Ubuntu Linux. On Centos 5.6 there is a libdbi-drivers rpm package but this does seem to contain any driver binaries under /usr/lib64/dbd. The real driver for MySQL lives in libdbi-dbd-mysql. Same for PostgreSQL. Make sure you have these installed, otherwise you will get a libdbi driver initialization error.

---

**Note**

On Windows libdbi can be compiled with cygwin only, so the binary msi package does not contain this module.

---



#### 6.4.2.1 Configuration

In addition to the **common module directives**, the following can be used to configure the om\_dbt module instance.

**Driver** This mandatory directive specifies the name of the libdbi driver which will be used to connect to the database. You will need to provide a DRIVER name here for which a loadable driver module exists under the name libdbdDRIVER.so (usually under /usr/lib/dbd/). The mysql driver is in the libdbdmysql.so file.

**SQL** This directive should specify the INSERT statement which is executed for each log message. The fields names (names with a \$ sign) will be replaced with the value they contain. String types will be quoted.

**Option** This directive can be used to specify additional driver options such as the connection parameters. The manual of the libdbi driver should contain the available options which can be used here.

#### 6.4.2.2 Configuration examples

These two examples below are for the plain syslog fields. Depending on your requirements, you may want to store additional or other fields which were generated by parsers, regexp rules, the **pm\_pattern** pattern matcher module or input modules. Notably the **im\_msvisialog** and **im\_mseventlog** modules generate different fields which you may want to store in an SQL database similarly to these examples.

---

**Example 6.45** Storing syslog in a PostgreSQL database

---

Below is a table schema which can be used to store syslog data.

```
CREATE TABLE log (  
  id serial,  
    timestamp timestamp not null,  
  hostname varchar(32) default NULL,  
  facility varchar(10) default NULL,  
  severity varchar(10) default NULL,  
  application varchar(10) default NULL,  
  message text,  
  PRIMARY KEY (id)  
);
```

And the config file:

```
<Extension syslog>  
  Module xm_syslog  
</Extension>  
  
<Input in>  
  Module im_tcp  
  Port 1234  
  Host 0.0.0.0  
  Exec parse_syslog_bsd();  
</Input>  
  
<Output dbi>  
  Module om_db  
  SQL      INSERT INTO log (facility, severity, hostname, timestamp, application, ↵  
    message) \  
    VALUES ($SyslogFacility, $SyslogSeverity, $Hostname, '$EventTime', ↵  
    $SourceName, $Message)  
  Driver postgresql  
  Option host 127.0.0.1  
  Option username dbuser  
  Option password secret  
  Option dbname logdb  
</Output>  
  
<Route 1>  
  Path in => dbi  
</Route>
```

---

---

**Example 6.46** Storing logs in a MySQL database

---

```
<Extension syslog>
  Module      xm_syslog
</Extension>

<Input in>
  Module      im_uds
  UDS         /dev/log
  Exec        parse_syslog_bsd();
</Input>

<Output dbi>
  Module      om_db
  SQL         INSERT INTO log (facility, severity, hostname, timestamp, application, ←
              message) \
              VALUES ($SyslogFacility, $SyslogSeverity, $Hostname, '$EventTime', ←
                      $SourceName, $Message)
  Driver      mysql
  Option      host 127.0.0.1
  Option      username mysql
  Option      password mysql
  Option      dbname logdb
</Output>

<Route 1>
  Path        in => dbi
</Route>
```

---

### 6.4.3 Program (om\_exec)

This module will execute a program or script on startup and will write (pipe) the log data to the program's standard input. Unless **OutputType** is set to something else, only the contents of the `$raw_event` field are sent over the pipe. The execution of the program or script will terminate when the module is stopped, which usually happens when nxlogs exits and the pipe is closed.

---

**Note**

The program or script is started when nxlog start and must not exit until the module is stopped. To invoke a script for each log message, use **xm\_exec** instead.

---

#### 6.4.3.1 Configuration

In addition to the **common module directives**, the following can be used to configure the om\_exec module instance.

**Command** This directive is mandatory. It specifies the name of the script/program to be executed.

**Arg** This is an optional parameter, multiple can be specified for each argument needed to pass to the **Command**. Note that specifying multiple arguments with one Arg directive separated with spaces will not work because the Command will receive it as one argument, so you will need to split them up.

**OutputType** See the description about **OutputType** in the global module config section.

---

### 6.4.3.2 Configuration examples

---

**Example 6.47** Piping logs to an external program

---

```
<Input in>
  Module im_uds
  uds    /dev/log
</Input>

<Output out>
  Module om_exec
  Command /usr/bin/someprog
  Arg     -
</Output>

<Route 1>
  Path in => out
</Route>
```

---

This exact same configuration is not recommended for real use because **im\_file** was designed to read log messages from files. This example only demonstrates the use of the **om\_exec** module.

## 6.4.4 File (om\_file)

This module can be used to write log messages to a file.

### 6.4.4.1 Configuration

In addition to the **common module directives**, the following can be used to configure the **om\_file** module instance.

**File** This mandatory directive specifies the name of the output file to open. It must be a **string** type **expression**. Note that the filename must be quoted to be a valid string literal unlike in other directives which take a filename argument. For relative filenames you should be aware that nxlog changes its working directory to '/' unless the global **SpoolDir** is set to something else.

**CreateDir** This optional directive takes a boolean value of TRUE or FALSE. If not specified, the default value is FALSE. If it is set to TRUE, the directory will be created if it doesn't exist before opening the file for writing.

**Truncate** This optional directive takes a boolean value of TRUE or FALSE. If set to TRUE, the file will be truncated before each write, meaning that only the most recent log message is saved. By default this is FALSE.

**Sync** This optional directive takes a boolean value of TRUE or FALSE. If set to TRUE, the file is synced after each log message, ensuring that it is really written to disk from the buffers. This can hurt performance, thus by default it is turned off.

**OutputType** See the description about **OutputType** in the global module config section.

### 6.4.4.2 Functions and procedures exported by om\_file

#### 6.4.4.2.1 Functions exported by om\_file

**string file\_name();**

**description** Return the name of the file currently open which was specified using the **File** directive. Note that this will be the old name if the file name changes dynamically. If you want the new name, use the expression you specified for the **File** directive instead of using this function.

---

**return type** `string`

**integer** `file_size()` ;

**description** Return the size of the currently open output file in bytes. It will return undef if the file is not open. This can happen if 'File' is not a string literal expression and there was no log message.

**return type** `integer`

#### 6.4.4.2.2 Procedures exported by `om_file`

**rotate\_to(string filename)** ;

**description** Rotate the current file to the filename specified. The module will then open the original file specified with the 'File' directive. Note that the `rename(2)` system call is used internally which does not support moving files across different devices on some platforms. If this is a problem, first rotate the file on the same device and then using the `exec_async()` procedure of the `xm_exec` module you can copy it to another device or file system or use the `file_copy()` procedure call provided by the `xm_fileop` module.

**arguments**

**filename** type: `string`

**reopen()** ;

**description** Reopen the File. This function should be called if the file has been removed or renamed e.g. with the `file_cycle()`, `file_remove()`, `file_rename()` functions of the `xm_file` module. This does not need to be called after `rotate_to()` because that reopens the file automatically.

#### 6.4.4.3 Configuration examples

---

##### Example 6.48 Storing raw syslog messages into a file

---

```
<Input in>
  Module im_uds
  UDS    /dev/log
</Input>

<Output out>
  Module om_file
  File   "/var/log/messages"
</Output>

<Route 1>
  Path  in => out
</Route>
```

---

**Example 6.49** File rotation based on size

```
<Extension exec>
  Module xm_exec
</Extension>

<Extension syslog>
  Module xm_syslog
</Extension>

<Input in>
  Module im_tcp
  Port 1514
  Host 0.0.0.0
  Exec parse_syslog_bsd();
</Input>

<Output out>
  Module om_file
  File "tmp/output_" + $Hostname + "_" + month(now())
  Exec if out->file_size() > 15M \
      { \
        $newfile = "tmp/output_" + $Hostname + "_" + strftime(now(), "%Y%m%d%H%M %S"); \
        out->rotate_to($newfile); \
        exec_async("/bin/bzip2", $newfile); \
      }
</Output>

<Route 1>
  Path in => out
</Route>
```

## 6.4.5 HTTP(s) (om\_http)

This module will connect to the **url** specified in the configuration in either plain HTTP or HTTPS mode. Each event data is transferred in a single POST request. The module then waits for a response containing a successful status code (200, 201 or 202). It will reconnect and retry the delivery if the remote has closed the connection or a timeout is exceeded while waiting for the response. This HTTP-level acknowledgement ensures that no messages are lost during transfer.

### 6.4.5.1 Configuration

In addition to the **common module directives**, the following can be used to configure the om\_http module instance.

**Url** This mandatory directive specifies the URL where the module should POST the event data. The module checks the url whether to operate in plain HTTP or HTTPS mode. It connects to the hostname specified in the url. If the port number is not explicitly indicated it defaults to 80 and 443 for HTTP and HTTPS respectively.

**HTTPSCertFile** This specifies the path of the certificate file to be used in the HTTPS handshake.

**HTTPSCertKeyFile** This specifies the path of the certificate key file to be used in the HTTPS handshake.

**HTTPSKeyPass** Optional password of the certificate key file defined in **HTTPSCertKeyFile**. For passwordless private keys the directive is not needed.

**HTTPSCAFile** This specifies the path of the certificate of the CA which will be used to check the certificate of the remote HTTPS server.

**HTTPSCADir** This specifies the path of CA certificates which will be used to check the certificate of the remote HTTPS server. The cert file names in this directory must be in the OpenSSL hashed format.

**HTTPSCRLFile** This specifies the path of the certificate revocation list (CRL) which will be used to check the certificate of the remote HTTPS server.

**HTTPSCRLDir** This specifies the path of certificate revocation lists (CRLs) which will be used to check the certificate of the remote HTTPS server. The file names in this directory must be in the OpenSSL hashed format.

**HTTPSAllowUntrusted** This takes a boolean value of TRUE or FALSE and specifies whether the connection should be allowed without certificate verification. If set to TRUE the connection will be allowed even if the remote HTTPS server presents unknown and self-signed certificates. The default value is FALSE if this directive is not specified, meaning that the remote end must present a trusted certificate by default.

### 6.4.5.2 Functions and procedures exported by om\_http

#### 6.4.5.2.1 Procedures exported by om\_http

**set\_http\_request\_path(string path);**

**description** Sets the path in the HTTP request to the string specified. This is useful if the URL is dynamic and parameters such as event id need to be included in the URL. Note that the string must be url encoded if it contains reserved characters.

**arguments**

**path** type: **string**

### 6.4.5.3 Configuration examples

---

#### Example 6.50 Sending logs over HTTPS

---

```
<Input in>
  Module      im_file
  File        'input.log'
  ReadFromLast FALSE
</Input>

<Output out>
  Module      om_http
  URL          https://server:8080/
  HTTPSCertFile %CERTDIR%/client-cert.pem
  HTTPSCertKeyFile %CERTDIR%/client-key.pem
  HTTPSCAFile   %CERTDIR%/ca.pem
  HTTPSAllowUntrusted FALSE
</Output>

<Route httpout>
  Path in => out
</Route>
```

---

### 6.4.6 Null (om\_null)

Log messages sent to the om\_null module instance are discarded, this module does not write its output anywhere. It can be useful for creating a dummy route, testing purposes. It can have **Scheduled** nxlog code execution as well like any other module, so it is not completely useless. This module does not have any module specific configuration directives. See [this example](#) for usage.

---

### 6.4.7 TLS/SSL (om\_ssl)

The om\_ssl module provides an SSL/TLS transport using the OpenSSL library beneath the surface. It behaves similarly to the **om\_tcp** module, except that an SSL handshake is performed at connection time and the data is received over a secure channel. Because log messages transferred over plain TCP can be eavasdropped or even altered with a man-in-the-middle attack, using the om\_ssl module provides a secure log message transport.

#### 6.4.7.1 Configuration

In addition to the **common module directives**, the following can be used to configure the om\_ssl module instance.

**Host** This specifies the IP address or a dns hostname where the module should connect to.

**Port** This specifies the port number where the module should connect to.

**Reconnect** Specifies an optional interval in seconds to wait after the remote end has disconnected. The default value is 10 seconds.

**CertFile** This specifies the path of the certificate file to be used in the SSL handshake.

**CertKeyFile** This specifies the path of the certificate key file to be used in the SSL handshake.

**KeyPass** Optional password of the certificate key file defined in **CertKeyFile**. For passwordless private keys the directive is not needed.

**CAFile** This specifies the path of the certificate of the CA which will be used to check the certificate of the remote socket against.

**CADir** This specifies the path of CA certificates which will be used to check the certificate of the remote socket against. The cert file names in this directory must be in the OpenSSL hashed format.

**CRLFile** This specifies the path of the certificate revocation list (CRL) which will be used to check the certificate of the remote socket against.

**CRLDir** This specifies the path of certificate revocation lists (CRLs) which will be used to check the certificate of the remote socket against. The file names in this directory must be in the OpenSSL hashed format.

**AllowUntrusted** This takes a boolean value of TRUE or FALSE and specifies whether the connection should be allowed without certificate verification. If set to TRUE the connection will be allowed even if the remote server presents unknown and self-signed certificates. The default value is FALSE if this directive is not specified, meaning that the remote end must present a trusted certificate by default.

**OutputType** See the description about **OutputType** in the global module config section.

---



### 6.4.7.2 Configuration examples

---

**Example 6.51** Writing nxlog binary data to another nxlog agent

---

```
<Input in>
  Module im_uds
  UDS    tmp/socket
</Input>

<Output sslout>
  Module om_ssl
  Host   localhost
  Port   23456
  CAFile %CERTDIR%/ca.pem
  CertFile %CERTDIR%/client-cert.pem
  CertKeyFile %CERTDIR%/client-key.pem
  KeyPass secret
  AllowUntrusted TRUE
  OutputType Binary
</Output>

<Route 1>
  Path in => sslout
</Route>
```

---

### 6.4.8 TCP (om\_tcp)

This module initiates a TCP connection to a remote host and transfers log messages. The TCP transfer protocol provides more reliable log transmission than UDP. If security is a concern, consider using the **om\_ssl** module instead.

#### 6.4.8.1 Configuration

In addition to the **common module directives**, the following can be used to configure the om\_tcp module instance.

**Host** This mandatory directive specifies the IP address or a dns hostname where the module should connect to.

**Port** This specifies the port number where the module should connect to. The default port is 514 if this directive is not specified.

**Reconnect** Specifies an optional interval in seconds to wait after the remote end has disconnected. The default value is 10 seconds.

**OutputType** See the description about **OutputType** in the global module config section.

---

### 6.4.8.2 Configuration examples

---

**Example 6.52** Transferring raw logs over TCP

---

```
<Input in>
  Module im_uds
  UDS    /dev/log
</Input>

<Output out>
  Module om_tcp
  Host   192.168.1.1
  Port   1514
</Output>

<Route 1>
  Path   in => out
</Route>
```

---

### 6.4.9 UDP (om\_udp)

This module sends log messages as UDP datagrams to the address and port specified in the configuration. UDP is the transport protocol of the old BSD syslog standard as described in RFC 3164, so this module can be particularly useful to send such messages to devices or syslog daemons which do not support other transports.

#### 6.4.9.1 Configuration

In addition to the **common module directives**, the following can be used to configure the om\_udp module instance.

**Host** This mandatory directive specifies the IP address or a dns hostname which the module will send UDP datagrams to.

**Port** This specifies the port number which the module will send UDP packets to. The default port is 514 if this directive is not specified.

**OutputType** See the description about **OutputType** in the global module config section.

#### 6.4.9.2 Configuration examples

---

**Example 6.53** Sending raw syslog over udp

---

```
<Input in>
  Module im_uds
  UDS    /dev/log
</Input>

<Output out>
  Module om_udp
  Host   192.168.1.1
  Port   1514
</Output>

<Route 1>
  Path   in => out
</Route>
```

---

### 6.4.10 UDS (om\_uds)

This module allows log messages to be sent to a unix domain socket. Traditionally unix systems have a socket, typically /dev/log used by the system logger to accept messages from. Applications wishing to send messages to the system log use the syslog(3) system call. nxlog can use this module to send log messages to the socket (=system logger) directly if another syslog daemon is in use.

---

**Note**

This module supports SOCK\_DGRAM type sockets only. SOCK\_STREAM type sockets will be supported in the future.

---

#### 6.4.10.1 Configuration

In addition to the **common module directives**, the following can be used to configure the om\_uds module instance.

**UDS** This specifies the path of the unix domain socket. The default is /dev/log if this is not specified.

**OutputType** See the description about **OutputType** in the global module config section.

#### 6.4.10.2 Configuration examples

---

**Example 6.54** Using the om\_uds module

---

```
<Extension syslog>
  Module xm_syslog
</Extension>

<Input in>
  Module im_file
  File "/var/log/custom_app.log"
</Input>

<Output out>
  Module om_uds
  # defaulting syslog fields and creating syslog output
  Exec parse_syslog_bsd(); to_syslog_bsd();
  uds /dev/log
</Output>

<Route 1>
  Path in => out
</Route>
```

---

## Chapter 7

# Offline log processing

### 7.1 nxlog-processor

The nxlog-processor tool is similar to the nxlog daemon, it uses the same configuration file. The difference is that it runs in foreground and will exit if there are no more log messages available from the input sources. The input sources are typically file and database sources. This tool is useful for offline log processing tasks such as loading a bunch of files into a database, converting between different formats (e.g. CSV and syslog), testing patterns, doing offline event correlation or HMAC message integrity checking.

FIXME manpage and usage

## Chapter 8

# Reading and receiving logs

This chapter deals with log sources such as [operating systems](#), [network](#), [database](#), [files](#), [applications](#) and [special devices](#). Some of these log sources need a dedicated input module or a special parser to be able to read and interpret the log messages. A parser can be a dedicated module implementing the parser routines and exporting these to the nxlog core as procedures and functions such as the [xm\\_syslog](#) module. Alternatively parsers can be directly implemented in the nxlog language by constructs such as [regular expressions](#) using capturing or the other built-in functions and procedures available for string manipulation. Writing parsers in the nxlog language is especially useful if there is no dedicated parser module for the specific log source.

### 8.1 Operating Systems

This section provides information and examples about collecting system messages of various operating systems. OS specific applications are also discussed here. The following sources are not operating system specific but work on most supported platforms:

- [Network](#)
- [Database](#)
- [Files](#)
- [External programs and scripts](#)
- [Multi-platform applications](#)

#### 8.1.1 Microsoft Windows

##### 8.1.1.1 Windows EventLog

To collect log messages from the EventLog subsystem on Windows 2000 and 2003, use the [im\\_mseventlog](#) module. This will also work on later versions, but the [im\\_msvistalog](#) module is recommended for use on Windows 2008, Vista and later. See the [im\\_mseventlog](#) and [im\\_msvistalog](#) configuration examples.

##### 8.1.1.2 Microsoft SQL Server

Microsoft SQL Server stores its logs in UTF-16 encoding using a line-based format. It is recommended to normalize the encoding to UTF-8. The following config snippet will do that.

```
<Extension _charconv>
  Module      xm_charconv
</Extension>

<Input in>
  Module      im_file
```

```
File "C:\\MSSQL\\ERRORLOG"
Exec convert_fields('UCS-2LE','UTF-8'); if $raw_event == '' drop();
</Input>
```

As of this writing, the **LineBased** parser, the default **InputType** for **im\_file** is not able to properly read the double-byte UTF-16 encoded files and will read an additional empty line (because of the double-byte CRLF). The above `drop()` call is intended to fix this. `convert_fields('UTF-16','UTF-8');` might also work instead of UCS-2LE.

#### 8.1.1.3 Microsoft IIS

Microsoft Internet Information Server supports **different log file formats**. Log files created by IIS are line-based and can be read with **im\_file**.

```
<Input IIS>
Module im_file
File 'C:\\inetpub\\logs\\LogFiles\\u_ex*'
</Input>
```

The above needs to be extended with appropriate parser rules if you want to parse the individual fields. See the following options depending on the format which is configured for your instance.

##### 8.1.1.3.1 W3C Extended Log File Format

See the **W3C Extended Log File Format** section in the **Processing messages** chapter, the **W3C Extended Log File Format (IIS 6.0)** docs and the **W3C Extended Log File Examples (IIS 6.0)**.

##### 8.1.1.3.2 Microsoft IIS Format

The IIS format is line-based, fields are comma separated. It can be parsed with the help of the **xm\_csv** module or with regular expressions.

##### 8.1.1.3.3 NCSA Common Log File Format

See the **NCSA Common Log Format** section in the **Processing messages** chapter.

##### 8.1.1.3.4 ODBC Logging

To read IIS logs from the ODBC datastore, use the **im\_odbc** module.

### 8.1.2 GNU/Linux

**Kernel logs** The **im\_kernel** module is dedicated to read the kernel log buffer.

**Local syslog** Local syslog is sent to the unix domain socket at `/dev/log`. The **im\_uds** module should be used together with **xm\_syslog** or **pm\_transformer**.

### 8.1.3 Android

**Kernel logs** The Linux kernel log can be read with the **im\_kernel** module.

**Android device logs** Android has a special in-kernel logging system. The **im\_android** module can read this.

## 8.2 Network

This section provides information and examples about receiving log messages from the network over various protocols.

### 8.2.1 UDP

See [im\\_udp](#).

### 8.2.2 TCP

See [im\\_tcp](#).

### 8.2.3 TLS/SSL over TCP

See [im\\_ssl](#).

### 8.2.4 Syslog

If you need to receive syslog over the network, the [xm\\_syslog](#) or [pm\\_transformer](#) module should be coupled with one of the network modules above. Syslog parsing is not even required if you only want to forward or store syslog as is.

## 8.3 Database

With special modules it is possible to read logs directly from database servers.

### 8.3.1 Using im\_dbi

The [im\\_dbi](#) module can be used on POSIX systems where libdbi is available. See the [im\\_dbi](#) module documentation.

### 8.3.2 Using im\_odbc

The [im\\_odbc](#) module can be used with ODBC compatible databases on Windows, Linux and Unix.

## 8.4 Files

The [im\\_file](#) module can be used to read logs from files. See [Processing messages](#) chapter about parsing various formats.

## 8.5 External programs and scripts

The [im\\_exec](#) module can be used to read logs from external programs and scripts over a pipe. See [Processing messages](#) chapter about parsing various formats.

## 8.6 Applications

This section provides information and examples about collecting log messages from various operating system independent (i.e. multi-platform) applications. Operating system applications are discussed in the [previous](#) section.

---

## 8.6.1 Apache HTTP Server

The Apache HTTP Server provides very comprehensive and flexible **logging capabilities**

### 8.6.1.1 Error log

FIXME

### 8.6.1.2 Access log - Common Log Format

See the **NCSA Common Log Format** section in the **Processing messages** chapter.

### 8.6.1.3 Access log - Combined Log Format

See the **NCSA Combined Log Format** section in the **Processing messages** chapter.

## 8.6.2 Apache Tomcat and java application logs

Apache tomcat and java applications are **very flexible** and can be configured for different transports and formats.

Here is a log sample consisting of 3 events. The log message of the second event spans multiple lines.

```
2001-01-25 17:31:42,136 INFO [com.nxsec.somepackage.Class] - single line
2001-01-25 17:41:16,268 ERROR [com.nxsec.somepackage.Class] - Error retrieving names: ; ←
    nested exception is:
        java.net.ConnectException: Connection refused
AxisFault
  faultCode: {http://schemas.xmlsoap.org/soap/envelope/}Server.userException
  faultSubcode:
  faultString: java.net.ConnectException: Connection refused
  faultActor:
  faultNode:
  faultDetail:
    {http://xml.apache.org/axis/}stackTrace:java.net.ConnectException: Connection ←
    refused
2001-01-25 17:57:38,469 INFO [com.nxsec.somepackage.Class] - third log message
```



---

**Example 8.1** Parsing tomcat logs into fields

---

This can be very useful to filter messages by the emitting java class for example.

```
<Input log4j>
  Module im_file
  File "/var/log/tomcat6/catalina.out"
  Exec if $raw_event =~ /^(?(\d{4})\-(\d{2})\-(\d{2}) \d{2}:\d{2}:\d{2}),\d{3} (\S+) \[(\S+)\]) ←
    \- (.*)/ \
    { \
      $log4j.time = parsedate($1); \
      $log4j.loglevel = $2; \
      $log4j.class = $3; \
      $log4j.msg = $4; \
    }
</Input>

<Output out>
  Module om_null
</Output>

<Route tomcat>
  Path log4j => out
</Route>
```

---

To parse and process multi-line messages such as the above, see the [Dealing with multi-line messages](#) section.

## 8.7 Devices

This section deals with special devices such as routers, switches and appliances.

### 8.7.1 Cisco

See [xm\\_syslog](#). FIXME

## Chapter 9

# Processing logs

This chapter deals with various tasks that might be required after a log message is received by nxlog.

### 9.1 Parsing various formats

There are a couple standard log file formats in use by various applications such as web servers, firewalls, ftp servers etc. This section tries to give a hand by providing config snippets to parse these formats.

Data is parsed and processed in multiple steps. For stream based data (e.g. files, TCP, SSL) the input module must know the log message boundary in order to be able to read each message frame. The framing depends on the format. The most common type is line-based, where each log message is separated by a linebreak. Log messages may be separated by the header only such as multi-line messages (e.g. stack and exception traces in java). So the first step during message reception is to read the frames, i.e. the log messages. This task is done by the input reader functions which can be specified with the **InputType** directive. There are a couple built-in input reader functions, others may be registered by modules.

There may be additional parsing involved or required after a message is read. For example when a BSD syslog message is read, the message frame is read by the **LineBased** input reader. Then this message may be further parsed (i.e. to extract the hostname, date, severity) by modules (such as **xm\_syslog**) or using nxlog language constructs in the **Exec** directive. This will result in nxlog message **fields** filled with value. There may be additional processing taken place to further tokenize or parse specific field contents (e.g. \$Message) using regular expressions or the **pm\_pattern** module.

#### 9.1.1 W3C Extended Log File Format

See the **specification draft** of the format, it's not all that long. The important header line is the one which starts with **#Fields**. Using this information you can set up a parser rule to tokenize the fields using either **xm\_csv** (as shown in the example below), **pm\_transformer** or using regular expressions directly (similarly to how it's done in the **Parsing apache logs in Combined Log Format** example).

---

---

**Example 9.1** Parsing the W3C Extended Log File Format using `xm_csv`

---

Here is a sample log in this format which we need to parse:

```
#Version: 1.0
#Date: 2011-07-01 00:00:00
#Fields: date time cs-method cs-uri
2011-07-01 00:34:23 GET /foo/bar1.html
2011-07-01 12:21:16 GET /foo/bar2.html
2011-07-01 12:45:52 GET /foo/bar3.html
2011-07-01 12:57:34 GET /foo/bar4.html
```

The following configuration reads this file, tokenizes it with the `csv` parser. Header lines starting with a leading sharp (#) are ignored. The `EventTime` field is constructed from the `date` and `time` fields and is converted to a **datetime** type. Finally the fields are output as JSON into another file.

```
<Extension w3c>
  Module      xm_csv
  Fields      $date, $time, $HTTPMethod, $HTTPURL
  FieldTypes  string, string, string, string
  Delimiter   ' '
</Extension>

<Extension json>
  Module      xm_json
</Extension>

<Input in>
  Module      im_file
  File        "tmp/iis.log"
  ReadFromLast FALSE
  Exec        if $raw_event =~ /^#/ drop(); \
              else \
              { \
                w3c->parse_csv(); \
                $EventTime = parsedate($date + " " + $time); \
              }
</Input>

<Output out>
  Module      om_file
  Exec        $raw_event = to_json();
  File        "tmp/output.json"
</Output>

<Route 1>
  Path        in => out
</Route>
```

---

## 9.1.2 NCSA Common Log File Format

FIXME

---

### 9.1.3 NCSA Combined Log Format

---

**Example 9.2** Parsing apache logs in Combined Log Format

---

The following configuration shows an example for filtering access logs and only storing those related to the user 'john':

```
<Input access_log>
  Module im_file
  File "/var/log/apache2/access.log"
  Exec if $raw_event =~ /^(S+) (S+) (S+) \[([^\]]+)\] \"(S+) (.+) HTTP.\d\.\d ↵
    \" (\d+) (\d+) \"([^\"]+)\" \"([^\"]+)\"/
    { \
      $Hostname = $1; \
      if $3 != '-' $AccountName = $3; \
      $EventTime = parsedate($4); \
      $HTTPMethod = $5; \
      $HTTPURL = $6; \
      $HTTPResponseStatus = $7; \
      $FileSize = $8; \
      $HTTPReferer = $9; \
      $HTTPUserAgent = $10; \
    }
</Input>

<Output out>
  Module om_file
  File '/var/log/john_access.log'
  Exec if not (defined($AccountName) and ($AccountName == 'john')) drop();
</Output>

<Route apache>
  Path access_log => out
</Route>
```

---

### 9.1.4 WebTrends Enhanced Log Format (WELF)

FIXME

### 9.1.5 Field delimited formats (CSV)

Comma, space, semicolon separated field list is a frequently used format. See the [xm\\_csv](#) and/or [pm\\_transformer](#) modules.

### 9.1.6 JSON

See the [xm\\_json](#) module about parsing structured data in JSON.

### 9.1.7 XML

See the [xm\\_xml](#) module about parsing structured data in XML.

## 9.2 Parsing date and time strings

The [parsedate\(\)](#) function can be used to efficiently parse strings representing a date. See the [Parsing apache logs in Combined Log Format](#) example for sample usage.

---

The following formats are supported by `parsedate`:

**RFC 3164 date** Legacy syslog messages contain the date in this format which lacks the year. Example:

```
Sun 6 Nov 08:49:37
```

Unfortunately there are some deviations in some implementations, so the following are also recognized:

```
Sun 06 Nov 08:49:37
Sun 6 Nov 08:49:37
```

**RFC 1123** RFC 1123 compliant dates are also supported, including a couple others which are similar such as those defined in RFC 822, RFC 850 and RFC 1036. Here is the concrete list:

```
Sun, 06 Nov 1994 08:49:37 GMT ; RFC 822, updated by RFC 1123
Sunday, 06-Nov-94 08:49:37 GMT ; RFC 850, obsoleted by RFC 1036
Sun Nov 6 08:49:37 1994 ; ANSI C's asctime() format
Sun, 6 Nov 1994 08:49:37 GMT ; RFC 822, updated by RFC 1123
Sun, 06 Nov 94 08:49:37 GMT ; RFC 822
Sun, 6 Nov 94 08:49:37 GMT ; RFC 822
Sun, 6 Nov 94 08:49:37 GMT ; RFC 822
Sun, 06 Nov 94 08:49 GMT ; Unknown
Sun, 6 Nov 94 08:49 GMT ; Unknown
Sun, 06 Nov 94 8:49:37 GMT ; Unknown [Elm 70.85]
Sun, 6 Nov 94 8:49:37 GMT ; Unknown [Elm 70.85]
Mon, 7 Jan 2002 07:21:22 GMT ; Unknown [Postfix]
Sun, 06-Nov-1994 08:49:37 GMT ; RFC 850 with four digit years
```

The above formats are recognized even without the leading day of week and without a timezone.

**Apache/NCSA date** This format can be found in Apache access logs ([NCSA Combined Log Format](#)) and possibly other sources. Example:

```
24/Aug/2009:16:08:57 +0200
```

**ISO and RFC 3339 date** `nxlog` can parse the ISO format with or without subsecond resolution, and with or without timezone information. It accepts either a comma (,) or a dot (.) in case there is sub-second resolution. Examples:

```
1977-09-06 01:02:03
1977-09-06 01:02:03.004
1977-09-06T01:02:03.004Z
1977-09-06T01:02:03.004+02:00
2011-5-29 0:3:21
2011-5-29 0:3:21+02:00
2011-5-29 0:3:21.004
2011-5-29 0:3:21.004+02:00
```

**CISCO syslog date** This is RFC 3164 format with millisecond precision. Example:

```
Nov 3 14:50:30.403
Nov 3 14:50:30.403
Nov 03 14:50:30.403
```

**Windows timestamp format** Example:

```
20100426151354.537875-000
20100426151354.537875000
```

Dates without timezone information are treated as local time. The year will be set to 1970 for dates missing the year such as in the [RFC 3164 date format](#). Use the [fix\\_year\(\)](#) function to correct the year in such cases.

The [datetime\(\)](#) function can be used to convert an integer expressing the number of microseconds elapsed since the epoch (Unix timestamp format).

---

**Example 9.3** Parsing a microsecond resolution timestamp expressed with a fractional part

---

The integer format of nxlog does not support fractional parts, so we use a regular expression to get around this.

```
$raw_event = ".... 1338872750.954349 .....";
if $raw_event =~ /(\d+)\.(\d+)/
{
    $EventTime = datetime(integer($1) * 1000000 + integer($2));
}
```

---

**Note**

The `%s` formatter for [strptime\(\)](#) is a GLIBC extension on GNU/Linux and is not available on other platforms.

---

If the above doesn't cut it, there is also [strptime\(\)](#) to parse more exotic formats.

---

**Example 9.4** Parsing date and time from Exchange logs

---

The following example is from an Exchange log. The date and time are delimited by a tab (i.e. they are two distinct fields). Also it uses a non standard single digit format instead of fixed width with double digits:

```
2011-5-29 0:3:2 GMT ...
```

To parse this, we can use a regexp and [strptime\(\)](#):

```
if $raw_event =~ /^( \d+ - \d+ - \d+ \t \d+ : \d+ : \d+ ) GMT / {
    $EventTime = strtotime($1, '%Y-%m-%d %t %H:%M:%S');
}
```

---

## 9.3 Filtering messages

Message filtering is a process where only a subset of the messages is let through. Filtering is possible using regular expressions or other operators using any of the fields.

### 9.3.1 Using drop()

Use the [drop\(\)](#) procedure to conditionally discard messages in an [Exec](#) directive.

```
<Input file>
  Module im_file
  File "/var/log/myapp/*.log"
  Exec if not ($raw_event =~ /failed/ or $raw_event =~ /error/) drop();
</Input>

<Output out>
  Module om_file
  File "/var/log/myapp/errors.txt"
</Output>

<Route 1>
```

```
    Path file => => out
</Route>
```

### 9.3.2 Filtering through pm\_filter

The other option is to use the **pm\_filter** module as in the following example:

```
<Input unix>
  Module im_uds
  uds    /dev/log
</Input>

<Processor filter>
  Module pm_filter
  Condition $raw_event =~ /failed/ or $raw_event =~ /error/
</Processor>

<Output out>
  Module om_file
  File    "/var/log/error"
</Output>

<Route 1>
  Path unix => filter => out
</Route>
```

## 9.4 Dealing with multi-line messages

### 9.4.1 Using module variables

This example uses regular expressions and module variables to concatenate lines belonging to a single event.

**Example 9.5** Parsing multiline messages using module variables

```
<Input log4j>
  Module im_file
  File "/var/log/tomcat6/catalina.out"
  Exec if $raw_event =~ /^\\d{4}\\-\\d{2}\\-\\d{2} \\d{2}:\\d{2}:\\d{2},\\d{3} \\S+ \\[\\S+\\] \\- \\S*/ <
    { \\
      if defined(get_var('saved')) \\
      { \\
        $tmp = $raw_event; \\
        $raw_event = get_var('saved'); \\
        set_var('saved', $tmp); \\
        $tmp = undef; \\
        log_info($raw_event); \\
      } \\
    } \\
    else \\
    { \\
      set_var('saved', $raw_event); \\
      drop(); \\
    } \\
  } \\
  else \\
  { \\
    set_var('saved', get_var('saved') + "\\n" + $raw_event); \\
    drop(); \\
  }
</Input>

<Output out>
  Module om_null
</Output>

<Route tomcat>
  Path log4j => out
</Route>
```

Unfortunately this solution has a minor flaw. The log message of an event is only forwarded if a new log is read, otherwise it is kept in the 'saved' variable indefinitely.

## 9.4.2 Using xm\_multiline

There is a dedicated extension module **xm\_multiline** which makes it easier to deal with multi-line messages without the need to use module variables and write complex rules.

## 9.5 Alerting, calling external scripts and programs

There are a couple ways to invoke external scripts and pass data to them.

### 9.5.1 Sending all messages to an external program

Using the **om\_exec** module, all messages can be piped to an external program or script which should be running until the module (or nxlog) is stopped.



## 9.5.2 Invoking a script or program for each message

The `xm_exec` module provides two procedure calls, `exec()` and `exec_async()`, to spawn an external script or program. See this [file rotation](#) example where `bzip` is executed to compress a logfile.

## 9.5.3 Alerting

Alerting is a process when a notification message is triggered if a certain condition is satisfied. Alerting can be implemented using one of the previous two modules. When using `om_exec`, the alerting script will receive all messages. The following example shows how to send an email using `xm_exec` when a regexp condition is met:

```
<Extension exec>
  Module xm_exec
</Extension>

<Input in>
  Module im_tcp
  Host 0.0.0.0
  Port 1514
  Exec if $raw_event =~ /alertcondition/ {
    exec_async("/bin/sh", "-c", 'echo "' + $Hostname + '\n\nRawEvent:\n' +
      $raw_event + \
      '|usr/bin/mail -a "Content-Type: text/plain; charset=UTF-8" -s \
      "ALERT" ' \
      + 'user@domain.com' );
  }
</Input>

<Output out>
  Module om_file
  File "/var/log/messages"
</Output>

<Route r>
  Path in => out
</Route>
```

## 9.6 Rewriting and modifying messages

There are many ways to modify log messages. A simple method which does not always work is to modify the `$raw_event` field (in case of syslog) without parsing the message. This can be done with regular expressions using capturing, for example:

```
if $raw_event =~ /^(aaaa)(replaceME)(.+)/ $raw_event = $1 + 'replaceMENT' + $3;
```

The more complex method is to parse the message into fields, modify some fields and finally reconstruct the message from the fields. The [conditional rewrite of the syslog facility](#) example shows such a syslog message modification method.

## 9.7 Message format conversion

To convert between CSV formats, see [this example](#).

The following example shows an nxlog configuration which receives IETF syslog over UDP and forwards in the old BSD syslog format over TCP:

```
<Extension syslog>
  Module xm_syslog
</Extension>

<Input in>
  Module im_udp
  Port 514
  Host 0.0.0.0
  Exec parse_syslog_ietf(); to_syslog_bsd();
</Input>

<Output out>
  Module om_tcp
  Host 1.2.3.4
  Port 1514
</Output>

<Route 1>
  Path in => out
</Route>
```

Take a look at the [pm\\_transformer](#) module which can do format conversion.

The requirements and possibilities for format conversion are endless. It is possible to do this using the nxlog language, dedicated modules, functions and procedures. For special cases a processor or extension module can be crafted to achieve this.

## 9.8 Character set conversion

It is recommended to normalize logs to UTF-8. Even if you don't, there may be cases where you need to convert a string (a field) or the whole message to another character set. See the [xm\\_charconv](#) module which adds support for character set conversion.

## 9.9 Discarding messages

See the [drop\(\)](#) procedure which can be invoked conditionally in the [Exec](#) directive. The [Filtering messages](#) section shows an example for using [drop\(\)](#). There is also [om\\_null](#) which could work in some situations.

## 9.10 Rate limiting

The poor man's tool for rate limiting is the [sleep\(\)](#) procedure.

---

**Example 9.6** Using sleep for rate limiting

In the following example `sleep` is invoked with 500 microseconds. This means that the input module will be able to read at most 2000 messages per second.

```
<Input in>
  Module  im_tcp
  Host    0.0.0.0
  Port    1514
  Exec    sleep(500);
</Input>

<Output out>
  Module  om_null
</Output>

<Route r>
  Path    in => out
</Route>
```

---

While this is not very precise because the module can do additional processing which can add some to the total execution time, it gets fairly close.

---

**Note**

Be careful if you are planning to add rate limiting to a route which reads logs over UDP.

---

## 9.11 Buffering

Each input module has its own read buffer which is used to fill with data during a read on a socket or file. Each processor and output has a limited queue where the log messages are put by the preceeding module in the route. The default limit for these internal queues is 100. Nevertheless, for buffering bigger amount of data, the `pm_buffer` module can do disk and memory based buffering.

## 9.12 Pattern matching and message classification

Pattern matching is commonly used for message classification. When certain strings are detected in a log message, the message gets tagged with classifiers. Thus it is possible to query or take action on these type of messages via the classifier only.

### 9.12.1 Regular expressions in the Exec directive

The first option is to use the `=~` operator in an `Exec` directive. The following code snippet shows an example for message classification.

---

---

**Example 9.7** Regular expression based message classification

When the contents of the Message field match against the regular expression, the AccountName and AccountID fields are filled with the appropriate values from the referenced captured substrings. Additionally the value LoginEvent is stored in the Action field.

```
if $Message =~ /^pam_unix\(sshd:session\) : session opened for user (\S+) by \(uid=(\d+)\)/ { ←
    {
        $AccountName = $1;
        $AccountID = integer($2);
        $Action = 'LoginEvent';
    }
}
```

---

### 9.12.2 Using pm\_pattern

When there are a lot of patterns, writing these in the configuration file will make it bloated, ugly and is not as efficient as using the **pm\_pattern** module. The above pattern matching rule can be defined in pm\_pattern's XML format the following way which will accomplish the same.

```
<pattern>
  <id>42</id>
  <name>ssh_pam_session_opened</name>
  <description>ssh pam session opened</description>
  <matchfield>
    <name>Message</name>
    <type>REGEXP</type>
    <value>^pam_unix\(sshd:session\) : session opened for user (\S+) by \(uid=(\d+)\)</ ←
      value>
    <capturedfield>
      <name>AccountName</name>
      <type>STRING</type>
    </capturedfield>
    <capturedfield>
      <name>AccountID</name>
      <type>INTEGER</type>
    </capturedfield>
  </matchfield>
  <set>
    <field>
      <name>Action</name>
      <type>STRING</type>
      <value>LoginEvent</value>
    </field>
  </set>
</pattern>
```

## 9.13 Event correlation

It is possible to write correlation rules in the nxlog language using the builtin features such as the **variables** and **statistical counters**. While these are quite powerful, some cases cannot be detected with these, especially thoses conditions which require a sliding window.

A dedicated nxlog module, pm\_evcorr is available for advanced correlation requirements. It has similar features as **SEC** and greatly enhances the correlation capabilites of nxlog.

---

## 9.14 Log rotation and retention

nxlog makes it possible to implement custom log rotation and retention policies for files written by nxlog and files written by other sources. The `om_file` and `xm_fileop` modules export various procedures which can be used for this purpose:

`rotate_to`  
`reopen`  
`file_cycle`  
`file_rename` Should these native language construct be insufficient, it is always possible to **call an external script or program**.  
`file_remove`  
`file_copy`  
`file_truncate`

---

### Example 9.8 Rotation of the internal LogFile

---

This example shows how to rotate the internal logfile based on time and size.

```
define LOGFILE "/var/log/nxlog/nxlog.log"

<Extension fileop>
  Module      xm_fileop

  # Check the size of our log file every hour and rotate if it is larger than 1Mb
  <Schedule>
    Every      1 hour
    Exec        if (file_size(%LOGFILE%) >= 1M) file_cycle(%LOGFILE%, 2);
  </Schedule>

  # Rotate our log file every week on sunday at midnight
  <Schedule>
    When        @weekly
    Exec        file_cycle(%LOGFILE%, 2);
  </Schedule>
</Extension>
```

---

---

**Example 9.9** File rotation based on size

---

```
<Extension exec>
  Module xm_exec
</Extension>

<Extension syslog>
  Module xm_syslog
</Extension>

<Input in>
  Module im_tcp
  Port 1514
  Host 0.0.0.0
  Exec parse_syslog_bsd();
</Input>

<Output out>
  Module om_file
  File "tmp/output_" + $Hostname + "_" + month(now())
  Exec if out->file_size() > 15M \
      { \
        $newfile = "tmp/output_" + $Hostname + "_" + strftime(now(), "%Y%m%d%H%M %S"); \
        out->rotate_to($newfile); \
        exec_async("/bin/bzip2", $newfile); \
      }
</Output>

<Route 1>
  Path in => out
</Route>
```

---

## 9.15 Explicit drop

nxlog does not drop messages voluntarily. The built-in flow control mechanism ensures that the input modules will pause until the output modules can write. This can be problematic in some situations when it is preferable to drop messages than to block. The following example illustrates the use of the **drop()** procedure used in conjunction with **pm\_buffer**.

**Example 9.10** Explicitly dropping messages when the network module is blocked

In the following configuration we use two routes which send the input read from the UDP socket to two outputs, a file and a TCP destination. Without this setup when the TCP connection can transmit slower than the rate of incoming UDP packets or the TCP connection is down, the whole chain (both routes) would be blocked which would result in dropped UDP packets. In this situation it is preferable to only drop log messages in the *tcp* route. In this route a **pm\_buffer** module is used and the size of the buffer is checked. If the buffer size goes over a certain limit we assume that the TCP output is blocked (or sending too slow) and instruct to drop log messages using the **drop()** procedure. This way the UDP input will not get paused and all messages will be written to the output file regardless of the state of the TCP connection.

```
<Processor buffer>
  Module pm_buffer
  WarnLimit 800
  MaxSize 1000
  Type Mem
  Exec if buffer_size() >= 80k drop();
</Processor>

<Input udpin>
  Module im_udp
  Host 0.0.0.0
  Port 1514
</Input>

<Output tcpout>
  Module om_tcp
  Host 192.168.1.1
  Port 1515
</Output>

<Output fileout>
  Module om_file
  File 'out.txt'
</Output>

<Route tcp>
  Path udpin => buffer => tcpout
</Route>
<Route file>
  Path udpin => fileout
</Route>
```

## Chapter 10

# Forwarding and storing logs

This chapter deals with the output side, i.e. how to forward, send and store messages to various destinations.

### 10.1 Data format of the output

In addition to the transport protocol, the data format is an important factor. If the remote receiver cannot parse the message, it will likely discard or it may be just improperly processed.

**Syslog** There are two formats, the older BSD Syslog and the newer IETF syslog format as defined by RFC 3164 and RFC 5424. The transport protocol in syslog can be UDP, TCP or SSL. See the [xm\\_syslog](#) module about formatting and sending syslog messages to remote hosts over the network.

**Syslog SNARE** The SNARE agent format is a special format on top of BSD Syslog which is used and understood by several tools and log analyzer frontends. This format is most useful when forwarding Windows EventLog data, i.e. in conjunction with [im\\_mseventlog](#) and/or [im\\_msvistalog](#). The [to\\_syslog\\_snare](#) procedure call can construct SNARE syslog formatted messages. The following example shows a configuration for reading the windows eventlog and forwarding it over UDP in the SNARE Agent format.

---

**Example 10.1** Forwarding EventLogs from a windows machine to a remote host in the SNARE Agent format

---

```
<Extension syslog>
  Module      xm_syslog
</Extension>

<Input in>
  Module      im_msvistalog
</Input>

<Output out>
  Module      om_udp
  Host        192.168.1.1
  Port        514
  Exec        to_syslog_snare();
</Output>

<Route 1>
  Path        in => out
</Route>
```

---

**NXLOG Binary format** The [binary](#) format is only understood by nxlog. All the fields are preserved when the data is sent in this format so there is no need to parse it again. You need to add this to the output module:

---



OutputType	Binary
------------	--------

And the receiver nxlog module should contain this:

InputType	Binary
-----------	--------

**CSV** To send logs in CSV, use the `xm_csv` or the `pm_transformer` module.

**Graylog Extended Log Format (GELF)** The `xm_gelf` module can be used to generate GELF output.

**JSON** See the `xm_json` module docs about generating JSON output.

**BSON** The BSON output format is currently unsupported. Support is planned and should be available in an upcoming nxlog release.

**XML** See the `xm_xml` module docs about generating XML output.

## 10.2 Forwarding over the network

The following network protocols can be used. There is a trade-off between speed, reliability, compatibility and security.

**UDP** To send logs in UDP packets, use the `om_udp` module.

**TCP** To send logs over TCP, use the `om_tcp` module.

**SSL/TLS** To send logs over a trusted secure SSL connection, use the `om_ssl` module.

## 10.3 Sending to sockets and files

**Files** To store logs in local files, use the `om_file` module.

**Piping to an external script or program** To send logs to an external program or script, use the `om_exec` module.

**Unix Domain Socket** To send logs to a unix domain socket, use the `om_uds` module.

## 10.4 Storing logs in a database

The `om_dbi` module can be used to store logs in databases which are supported by the libdbi database abstraction library. This is only supported on POSIX platforms and is not available on Windows.

## Chapter 11

# Tips and tricks

This chapter addresses some common problems or log management requirements.

### 11.1 Detecting a dead agent or log source

It is a common requirement to be able to detect conditions when there are no log messages coming from a source. This usually indicates a problem with the log source, such as a broken network connection, server down or an application/system service is stuck. Usually this problem should be detected by monitoring tools (nagios, openview etc), but the absence of logs can be also a good reason to investigate such a situation.

---

#### Note

The `im_mark` module exists for a similar purpose. It can emit messages periodically in order to show that the system logger is not suffering problems.

---

The solution to this problem is the combined use of `statistical counters` and `Scheduled` checks. The input module can update a statistical counter configured to calculate events per hour for example. In the same input module a `Schedule` block is defined which checks the value of the statistical counter periodically. When the event rate is zero or drops below a certain limit, an appropriate action can be `executed` such as sending out an alert email or generating an internal warning message. Note that probably there are other ways to solve this issue and this method might not be the optimal for all situations.

---

**Example 11.1** Alerting on absence of log messages

The following configuration example creates a statistical counter in the context of the `im_tcp` module to calculate the number of events received per hour. The **Schedule** block within the context of the same module checks the value of the "msgrate" statistical counter and generates an internal error message when there were no logs received in the past hour.

```
<Input in>
  Module im_tcp
  Port 2345
  Exec create_stat("msgrate", "RATE", 3600); add_stat("msgrate", 1);

  <Schedule>
  Every 3600 sec
  Exec create_stat("msgrate", "RATE", 10); add_stat("msgrate", 0);
  Exec if defined get_stat("msgrate") and get_stat("msgrate") <= 1 \
    { \
      log_error("No messages received from the source!"); \
    }
  </Schedule>
</Input>

<Output out>
  Module om_file
  File "tmp/output"
</Output>

<Route 1>
  Path in => out
</Route>
```

---

## Chapter 12

# Troubleshooting

According to Murphy, anything that can go wrong will go wrong. This chapter is to help diagnosing problems, be that configuration errors or possible bugs.

### 12.1 nxlog's internal logs

While nxlog is a tool to handle logs from external sources, it can and will emit logs about its own operations. These are essential to troubleshoot problems.

#### 12.1.1 Check the contents of the LogFile

nxlog will log important events including errors and warnings into its logfile. So the first place to look for errors is the **LogFile**. If this directive is not specified in the configuration, you should add it.

---

**Note**

Some windows applications (e.g. wordpad) cannot open the logfile while nxlog is running because of exclusive file locking. Use a text file viewer which does not lock the file (e.g. notepad).

---

#### 12.1.2 Injecting own logs into a route

Internal logs can be read as a log source with the **im\_internal** module. This makes it possible to forward the internal logs over the network for example.

---

**Note**

This method will not work if the route which **im\_internal** is part of is not functional. Logging with **LogFile** is more fault-tolerant and this is the recommended way for troubleshooting.

---

#### 12.1.3 LogLevel

Internal logs are emitted only on LogLevel of INFO and above. It is possible to get detailed information about what nxlog is doing by setting **LogLevel** to DEBUG. This can produce an extreme amount of logs, it is recommended to enable this only for troubleshooting.

---

## 12.1.4 Running in foreground

When nxlog is running in foreground, it will emit logs to STDOUT and STDERR so the logs will be visible in the running terminal. This is the same log which is written to the **LogFile**. It can be started to run in foreground with **nxlog -f**.

## 12.1.5 Using log\_info() in the Exec directive

Internal logs can be emitted from the configuration via the **log\_info()** procedure call in the **Exec** directive. This can be extremely useful to print and debug message contents. Consider the following example:

```
<Input in>
  Module im_udp
  Port 514
  Exec      if $raw_event =~ /keyword/ log_info("FOUND KEYWORD IN MSG: [" + $raw_event ←
    + "]);
</Input>
```

Anything which is printed with the **log\_info()** and family of procedure calls will appear in **LogFile**, on the STDOUT/STDERR of nxlog in foreground mode and will be emitted by **im\_internal**.

## 12.2 Common problems

This section will list a couple problems which you are likely to run into.

### 12.2.1 Missing logdata

As discussed in the **architecture** chapter, logs are received by input modules, forwarded to the processor modules and finally handled by the output modules. When these modules handle a log message, the **Exec** directive is evaluated. There are a few situations when such statements can be evaluated but the required log is not available in the current context. When the so called logdata is not available in the current context, any dependent operation will fail and the evaluation of the Exec code will terminate. Most notably these operations are **field assignments** and function or procedure calls which access fields such as **convert\_fields()**. Consider the following example.

---

#### Example 12.1 Assignment after drop()

---

In this example the message is conditionally dropped. When the **\$raw\_event** field matches the keyword, the **drop()** operation is invoked which discards the log completely. If a subsequent statement follows which accesses the log, it will fail.

```
<Input in>
  Module im_udp
  Port 514
  Exec      if $raw_event =~ /keyword/ drop(); $EventTime = now();
</Input>
```

In this case the following internal error log will be emitted:

```
missing logdata, assignment possibly after drop()
```

The following config snippet fixes the above error by correctly using conditional statements.

```
<Input in>
  Module im_udp
  Port 514
  Exec      if $raw_event =~ /keyword/ \
            drop();                      \
            else                          \
            $EventTime = now();
</Input>
```

---

The "logdata will be missing" in the following cases:

Accessing a field or calling a procedure/function which needs the logdata after the **drop()** procedure.

Accessing a field or calling a procedure/function which needs the logdata from the **Exec directive of a Schedule block**. Since this sch

### 12.2.2 nxlog failed to start, cannot read configuration file

You may receive this error message in nxlog.log when nxlog fails to start:

```
nxlog failed to start: Invalid keyword: ýþ# at C:\Program Files (x86)\nxlog\conf\nxlog.conf ←  
:1
```

Some text editors may save the configuration file in UTF-16 or in UTF-8 with a BOM header. The configuration file must be encoded in ASCII or plain UTF-8, otherwise you will get this error. On windows using notepad.exe should work properly.

### 12.2.3 nxlog.log is in use by another application and cannot be accessed

You may receive this error message on Windows when trying to open the logfile (usually `nxlog.log`) with a text editor which uses exclusive locking. You can only open the log after nxlog is stopped. By using a text viewer or text editor which does not use exclusive locking (such as notepad.exe), you can open the logfile without the need to stop nxlog.

### 12.2.4 Connection refused when trying to connect to im\_tcp or im\_ssl

Make sure that you have no firewall blocking the connection. The interface address or the hostname which resolves to the interface address must be accessible from the outside. See the **Host** directive of `im_tcp`.

## 12.3 Debugging and dumping messages

When creating complex processing rules and configurations, you will most likely run into a problem and need to debug the stream of event log messages to see

- what has been received/read by the input(s),
- whether some required field exists and what its value is,
- if the parser is working correctly and populating the fields as it should,
- all the fields and their values contained in the event log after parsing.

The following configuration snippets show some examples how this can be done.

---

**Example 12.2** Writing the values of fields to an external file

The `file_write()` procedure provided by the `xm_fileop` module can be used to dump information into an external file.

```
<Extension fileop>
  Module      xm_fileop
</Extension>

<Extension syslog>
  Module      xm_syslog
</Extension>

<Input in>
  Module      im_tcp
  Host        0.0.0.0
  Port        1514
  Exec        parse_syslog_bsd();
  # Debug SyslogSeverity and Hostname fields
  Exec        file_write("/tmp/debug.txt", "Severity: " + $SyslogSeverity + ", Hostname: " +
    " + $Hostname);
</Input>

<Output out>
  Module      om_null
</Output>

<Route r>
  Path        in => out
</Route>
```

---

**Example 12.3** Writing the values of fields to the internal log

Using the `log_info` procedure the values can be sent to the internal log. This will be visible in the file defined with the `LogFile` global directive, in the input from the `im_internal` module and on standard output when running `nxlog` in foreground with the `-f` command line switch.

```
<Extension syslog>
  Module      xm_syslog
</Extension>

<Input in>
  Module      im_tcp
  Host        0.0.0.0
  Port        1514
  Exec        parse_syslog_bsd();
  # Debug SyslogSeverity and Hostname fields
  Exec        log_info("Severity: " + $SyslogSeverity + ", Hostname: " + $Hostname);
</Input>

<Output out>
  Module      om_null
</Output>

<Route r>
  Path        in => out
</Route>
```

---

**Example 12.4** Dumping all the fields

Using the `to_json` procedure provided by the `xm_json` module, all the fields can be dumped. If you prefer, you can use the `to_xml` procedure provided by the `xm_xml` module.

```
<Extension syslog>
  Module      xm_syslog
</Extension>

<Extension json>
  Module      xm_json
</Extension>

<Input in>
  Module      im_tcp
  Host        0.0.0.0
  Port        1514
  Exec        parse_syslog_bsd();
  # Dump $raw_event
  Exec        log_info("raw event is: " + $raw_event);
  # Dump fields in JSON
  Exec        log_info("Other fields are: " + to_json());
</Input>

<Output out>
  Module      om_null
</Output>

<Route r>
  Path        in => out
</Route>
```

This will produce the following output in the logs:

```
2012-05-18 13:11:35 INFO raw event is: <27>2010-10-12 12:49:06 host app[12345]: test  ←
message
2012-05-18 13:11:35 INFO Other fields are: {"MessageSourceAddress":"127.0.0.1"," ←
  EventReceivedTime":"2012-05-18 13:11:35",
    "SourceModuleName":"in","SourceModuleType":"im_tcp","SyslogFacilityValue":3," ←
    SyslogFacility":"DAEMON",
    "SyslogSeverityValue":3,"SyslogSeverity":"ERR","SeverityValue":4,"Severity":" ←
    ERROR","Hostname":"host",
    "EventTime":"2010-10-12 12:49:06","SourceName":"app","ProcessID":"12345"," ←
    Message":"test message"}
```

In some cases nxlog is already receiving some invalid data it cannot grok. To verify that indeed this is the case, use a network traffic analyzer such as wireshark or tcpdump.