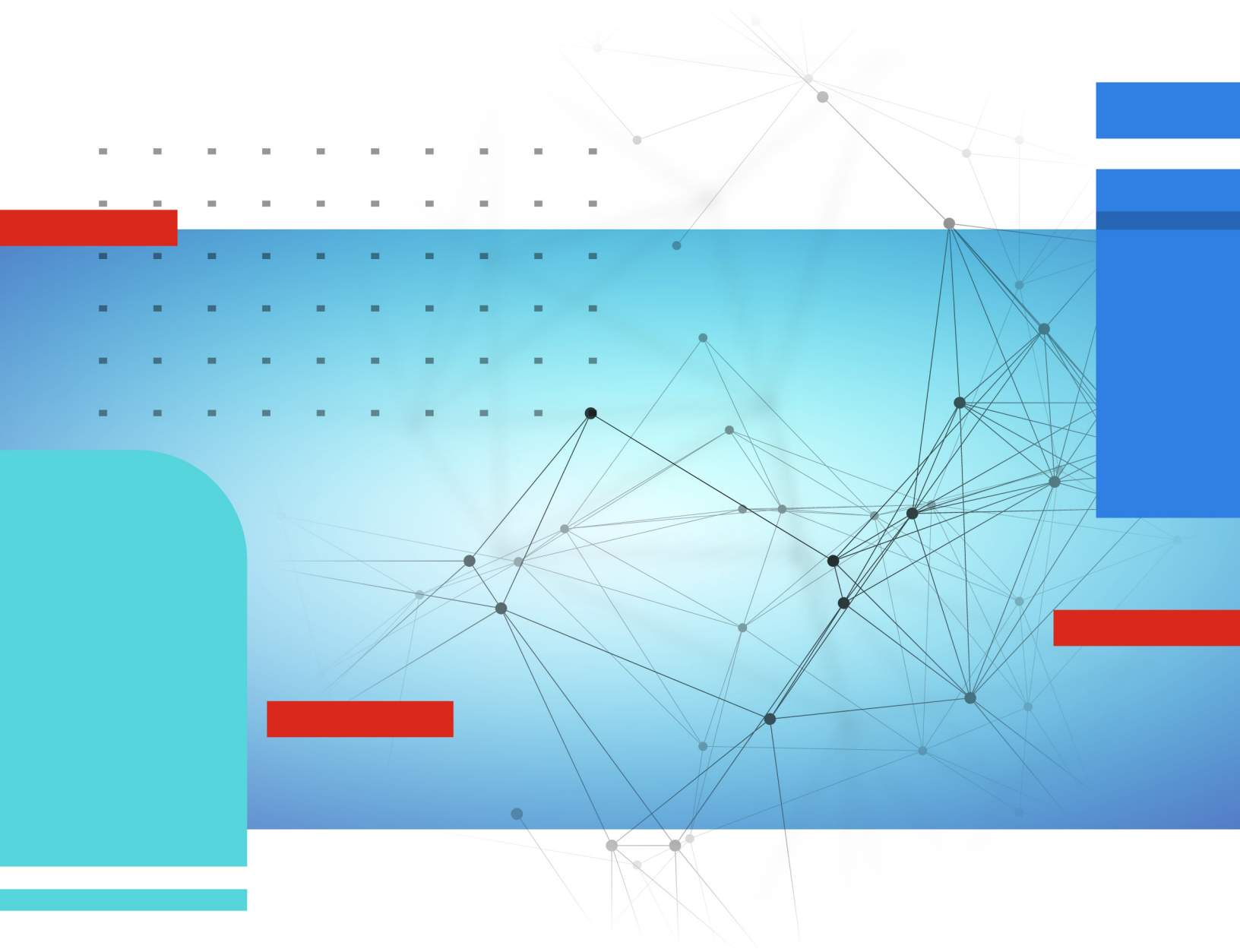
Custom Log Parsers

FortiAnalyzer 7.6



FORTINET DOCUMENT LIBRARY
https://docs.fortinet.com

FORTINET VIDEO LIBRARY
https://video.fortinet.com

FORTINET BLOG
https://blog.fortinet.com

CUSTOMER SERVICE & SUPPORT
https://support.fortinet.com

FORTINET TRAINING & CERTIFICATION PROGRAM
https://www.fortinet.com/training-certification

FORTINET TRAINING INSTITUTE
https://training.fortinet.com

FORTIGUARD LABS
https://www.fortiguard.com

END USER LICENSE AGREEMENT
https://www.fortinet.com/doc/legal/EULA.pdf

FEEDBACK
Email: techdoc@fortinet.com

September 17, 2024
FortiAnalyzer 7.6 Custom Log Parsers
00-76-1058849-20240917

TABLE OF CONTENTS

Change Log

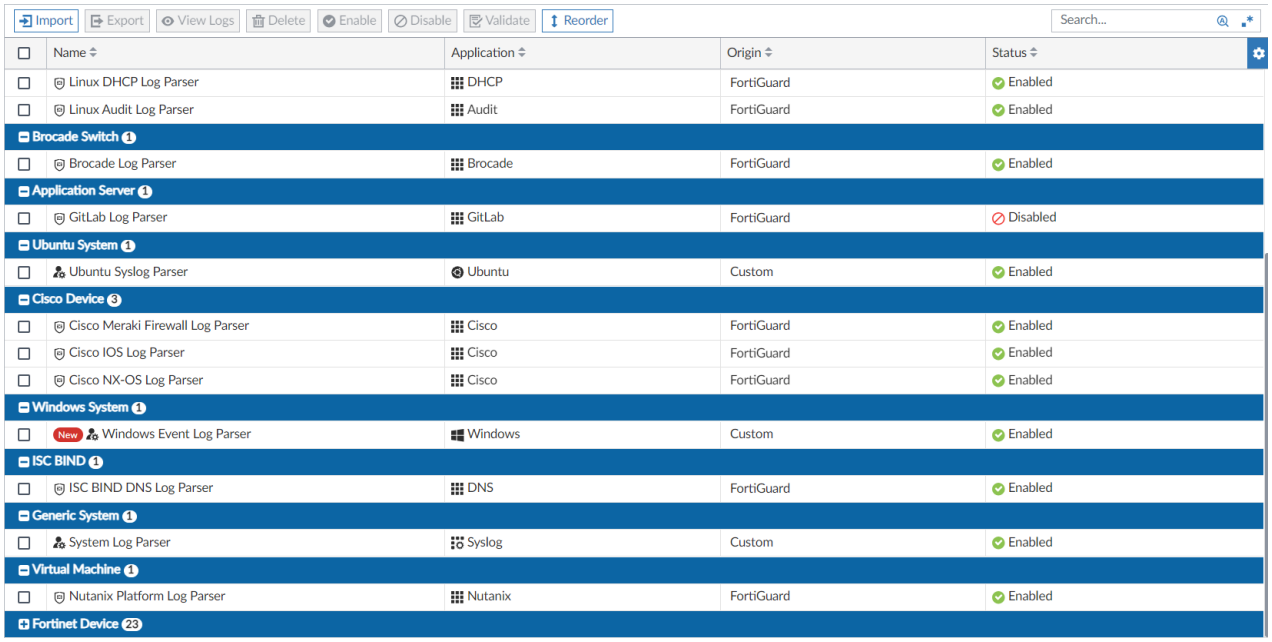
Date	Change Description
2024-09-17	Initial publication.

Introduction

As of FortiAnalyzer 7.4.2, there are 28 custom parsers, which includes 23 Fortinet devices and 5 for Apache, Nginx, Windows, Ubuntu, and Generic-System Applications.

Beginning in FortiAnalyzer 7.4.2, there is support to import custom log parsers. After custom logs are received by FortiAnalyzer, they will be auto-assigned or assigned manually by this custom parser and parsed into the SIEM database (siemdb).

To view log parsers in the FortiAnalyzer GUI, go to *Incidents & Events* > *Log Parser* > *Log Parsers*. The custom log parsers can be imported and Enabled from this pane as well.



Fortinet releases RHSP packages every month to add more third-party syslog parsers to FortiAnalyzer from FortiGuard. Log parsers added as part of the RHSP packages will display *FortiGuard* in the *Origin* column. For more information, see *Security Automation Service objects* in the FortiAnalyzer Administration Guide.

Building a custom log parser

This section explains the framework for log parsers using an existing example in FortiAnalyzer.

Example

In *Incidents & Events* > *Log Parser* > *Log Parsers*, each log parser can be exported and downloaded as a JSON file using the following naming convention: `<log parser name> Log Parser.json`.

For example, see the readable JSON format for a log parser below.

```
{
    "header": {
        "name": "Linux DHCP Log Parser",
        "category": "Linux Device",
        "application": "DHCP",
        "author": "FortiAnalyzer, Fortinet Inc.",
        "change_time": "2024-03-26 00:00:00",
        "version": "7.4.2-rc2"
    },
    "data_source": {
        "matchers": [
            {
                "description": "Linux DHCP logs sent as syslog, matching by patterns.",
                "type": "syslog",
                "patterns": [
                    "^(?<header>.+)[[:space:]]+dhcpd:[[:space:]]+(?<dhcp_type>[[:word:]]+)
[[:space:]]+(?<msg_body>.+)$"
                ]
            }
        ]
    },
    "parser": {
        "rules": {
            "rule_inti_func": {
                "description": "Bypass the sub action func is not copied.",
                "actions": [
                    "header.contains(\"T\")",
                    "data_sourcetype = \"Linux DHCP\""
                ]
            },
            "rule_header": {
```

```
                "description": "parse date and time in header",
                "trigger": {
                    "expression": "${condition_1}",
                    "condition_1": "header.is_not_empty()"
                },
                "actions": [
                    "tmp_mon, tmp_day, tmp_hour, tmp_min, tmp_sec = header.match(\"
([[:alpha:]]+)[[:space:]]+([[:digit:]]+)[[:space:]]+([0-9][0-9]):([0-9][0-9]):([0-9][0-9])\")",
                    "tmp_mon = get_month_num(tmp_mon)",
                    "tmp_year = get_current_year()",
                    "data_timestamp = to_timestamp(tmp_year, tmp_mon, tmp_day, tmp_hour, tmp_
min, tmp_sec)",
                    {
                        "description": "parse date and time 2024-01-01T00:00:00.1234567+07:00",
                        "trigger": {
                            "expression": "${condition_1}",
                            "condition_1": "data_timestamp.is_empty()"
                        },
                        "if": {
                            "trigger": [
                                "tmp_date, tmp_time, tmp_tz = header.match(\"([0-9][0-9][0-9][0-
9]-[0-9][0-9]-[0-9][0-9])T([0-9][0-9]:[0-9][0-9]:[0-9][0-9])\\.[[:digit:]]+([+\\-A-Z0-9:]*)\")",
                                "data_timestamp = to_timestamp_by_date_time(tmp_date, tmp_time,
tmp_tz)"
                            ]
                        }
                    },
                    {
                        "description": "parse date and time 2024-01-01T00:00:00+07:00",
                        "trigger": {
                            "expression": "${condition_1}",
                            "condition_1": "data_timestamp.is_empty()"
                        },
                        "if": {
                            "trigger": [
                                "tmp_date, tmp_time, tmp_tz = header.match(\"([0-9][0-9][0-9][0-
9]-[0-9][0-9]-[0-9][0-9])T([0-9][0-9]:[0-9][0-9]:[0-9][0-9])([+\\-A-Z0-9:]*)\")",
                                "data_timestamp = to_timestamp_by_date_time(tmp_date, tmp_time,
tmp_tz)"
                            ]
                        }
                    },
                    "tmp_reptIp = header.match(\"([[:digit:]]+\\.[[:digit:]]+\\.[[:digit:]]+\\.
[[:digit:]]+)\")",
                    "tmp_reptName = header.match(\"[0-9][0-9]:[0-9][0-9]:[0-9][0-9][+:\\-\\.
[:alnum:]]*[[:space:]]+([[:word:]-]+)\")",
```

```
                    "reptDevIpAddr = tmp_reptIp",
                    "hostIpAddr = tmp_reptIp",
                    "reptDevName = tmp_reptName",
                    "hostName = tmp_reptName"
                ]
            },
            "rule_dhcp_type": {
                "description": "parse dhcp_type",
                "trigger": {
                    "expression": "${condition_1}",
                    "condition_1": "dhcp_type.is_not_empty()"
                },
                "actions": [
                    "eventType = concat(\"Linux\", \"_\", dhcp_type)",
                    "eventAction = \"0\""
                ]
            },
            "rule_2": {
                "description": "parse msg_body to set fields",
                "trigger": {
                    "expression": "${condition_1} and ${condition_2}",
                    "condition_1": "dhcp_type.is_not_empty()",
                    "condition_2": "msg_body.is_not_empty()"
                },
                "actions": [
                    {
                        "trigger": {
                            "expression": "${condition_1}",
                            "condition_1": "dhcp_type == \"DHCPDISCOVER\""
                        },
                        "if": {
                            "trigger": [
                                "hostMACAddr = msg_body.match(\"from[[:space:]]+([[:word:]
[:punct:]]+)[[:space:]]+via[[:space:]]+[[:word:][:punct:]]+\")"
                            ]
                        }
                    },
                    {
                        "trigger": {
                            "expression": "${condition_1}",
                            "condition_1": "dhcp_type == \"DHCPOFFER\""
                        },
                        "if": {
                            "trigger": [
                                "hostMACAddr,hostName = msg_body.match(\"on[[:space:]]+
```

```
[[:digit:]]+\\.[[:digit:]]+\\.[[:digit:]]+\\.[[:digit:]]+[[:space:]]+to[[:space:]]+([[:word:]
[:punct:]]+) \\(([[:word:][:punct:]]+)\\).*\")"
                            ]
                        }
                    },
                    {
                        "trigger": {
                            "expression": "${condition_1}",
                            "condition_1": "dhcp_type == \"DHCPREQUEST\""
                        },
                        "if": {
                            "trigger": [
                                "hostIpAddr,hostMACAddr,hostName = msg_body.match(\"for
[[:space:]]+([[:digit:]]+\\.[[:digit:]]+\\.[[:digit:]]+\\.[[:digit:]]+)[[:space:]]+\\
([[:digit:]]+\\.[[:digit:]]+\\.[[:digit:]]+\\.[[:digit:]]+\\)[[:space:]]+from[[:space:]]+
([[:word:][:punct:]]+)[[:space:]]+\\(([[:word:][:punct:]]+)\\).*\")"
                            ]
                        }
                    },
                    {
                        "trigger": {
                            "expression": "${condition_1}",
                            "condition_1": "dhcp_type == \"DHCPACK\""
                        },
                        "if": {
                            "trigger": [
                                "hostIpAddr,hostMACAddr,hostName = msg_body.match(\"on
([[:digit:]]+\\.[[:digit:]]+\\.[[:digit:]]+\\.[[:digit:]]+) to ([[:word:][:punct:]]+) \\
(([[:word:][:punct:]]+)\\).*\")"
                            ]
                        }
                    },
                    {
                        "trigger": {
                            "expression": "${condition_1}",
                            "condition_1": "dhcp_type == \"DHCPINFORM\""
                        },
                        "if": {
                            "trigger": [
                                "hostIpAddr = msg_body.match(\"from ([[:digit:]]+\\.
[[:digit:]]+\\.[[:digit:]]+\\.[[:digit:]]+).*\")"
                            ]
                        }
                    },
                    {
```

```
                        "trigger": {
                            "expression": "${condition_1}",
                            "condition_1": "dhcp_type == \"DHCPDECLINE\""
                        },
                        "if": {
                            "trigger": [
                                "hostIpAddr,hostMACAddr,hostName = msg_body.match(\"of
([[:digit:]]+\\.[[:digit:]]+\\.[[:digit:]]+\\.[[:digit:]]+) from ([[:word:][:punct:]]+) \\
(([[:word:][:punct:]]+)\\).*\")"
                            ]
                        }
                    },
                    {
                        "trigger": {
                            "expression": "${condition_1}",
                            "condition_1": "dhcp_type == \"DHCPRELEASE\""
                        },
                        "if": {
                            "trigger": [
                                "hostIpAddr,hostMACAddr,hostName = msg_body.match(\"of
([[:digit:]]+\\.[[:digit:]]+\\.[[:digit:]]+\\.[[:digit:]]+) from ([[:word:][:punct:]]+) \\
(([[:word:][:punct:]]+)\\).*\")"
                            ]
                        }
                    }
                ]
            },
            "rule_3": {
                "trigger": {
                    "expression": "${condition_1}",
                    "condition_1": "hostName.is_not_empty()"
                },
                "actions": [
                    "hostName = hostName"
                ]
            }
        },
        "field_map": {
            "data_sourceid": [],
            "data_sourcename": [
                "data_sourcename"
            ],
            "data_sourcetype": [
                "data_sourcetype"
            ],
```

```
            "data_sourceversion": [],
            "data_timestamp": [
                "data_timestamp"
            ],
            "app_cat": [],
            "app_id": [],
            "app_name": [],
            "app_proc": [],
            "app_ref": [],
            "app_service": [],
            "app_state": [],
            "app_ver": [],
            "dns_query": [],
            "dns_querytype": [],
            "dns_response": [],
            "dst_domain": [],
            "dst_geo": [
                "destName"
            ],
            "dst_intf": [],
            "dst_ip": [],
            "dst_mac": [],
            "dst_natip": [],
            "dst_natport": [],
            "dst_port": [],
            "event_action": [
                "eventAction"
            ],
            "event_id": [],
            "event_message": [
                "msg_body"
            ],
            "event_outcome": [],
            "event_policy": [],
            "event_profile": [],
            "event_ref": [],
            "event_severity": [
                "eventSeverity"
            ],
            "event_subtype": [],
            "event_type": [
                "eventType"
            ],
            "event_cat": [],
            "file_accessetime": [],
```

```
            "file_createtime": [],
            "file_ext": [],
            "file_hash": [],
            "file_hashtype": [],
            "file_name": [],
            "file_path": [],
            "file_size": [],
            "host_classification": [],
            "host_hwvendor": [],
            "host_hwver": [],
            "host_ip": [
                "hostIpAddr"
            ],
            "host_location": [],
            "host_mac": [
                "hostMACAddr"
            ],
            "host_name": [
                "hostName"
            ],
            "host_osfamily": [],
            "host_osname": [],
            "host_osver": [],
            "host_owner": [],
            "host_type": [],
            "host_uid": [],
            "http_cookie": [],
            "http_method": [],
            "http_referer": [],
            "http_url": [],
            "http_useragent": [],
            "mail_from": [],
            "mail_size": [],
            "mail_subject": [],
            "mail_to": [],
            "net_direction": [],
            "net_name": [],
            "net_payloadid": [],
            "net_proto": [],
            "net_rcvdpkts": [],
            "net_recvbytes": [],
            "net_sentbytes": [],
            "net_sentpkts": [],
            "net_sessionduration": [],
            "net_sessionid": [],
```

```
            "net_ssid": [],
            "src_domain": [],
            "src_geo": [],
            "src_intf": [],
            "src_ip": [],
            "src_mac": [],
            "src_natip": [],
            "src_natport": [],
            "src_port": [],
            "threat_action": [],
            "threat_direction": [],
            "threat_id": [],
            "threat_name": [],
            "threat_pattern": [],
            "threat_ref": [],
            "threat_score": [],
            "threat_severity": [],
            "threat_type": [],
            "user_authtype": [],
            "user_classification": [],
            "user_domain": [],
            "user_email": [],
            "user_group": [],
            "user_id": [],
            "user_location": [],
            "user_name": [],
            "user_org": [],
            "user_phone": [],
            "user_role": [],
            "user_social": []
        }
    }
}
```

Log parser framework

This topic describes the framework for custom log parsers using the example provided in Building a custom log parser on page 6.

There are three parts to the parser file:

- `header`
- `data_source`
- `parser`

Multiple `rules` can be included within the `parser`.

1. `header`: Basic information, including name, category, and application.
   See `#` below for further descriptions.

```
"header": {
    "name": "XXXX Log Parser",
    "category": "Generic System",    # Cisco_device/Fortinet_Device/… : will display in
Category by GUI
    "application": "Syslog",        # Syslog, DHCP, Cisco… : will display in Application by
GUI
    "author": "FortiAnalyzer, Fortinet Inc.",    # author info
    "change_time": "2024-06-06 00:00:00",    # add time or change time.
    "version": "7.6.0"        # version.
},
```

2. `data_source`: Checks if patterns match devices or not.

```
"data_source": {
    "matchers": [
        {
            "description": "Linux DHCP logs sent as syslog, matching by patterns.",    #
comment
            "type": "syslog"      # if third syslog, default is syslog
            "patterns": [
                "^(?<header>.+)[[:space:]]+dhcpd:[[:space:]]+(?<dhcp_type>[[:word:]]+)
[[:space:]]+(?<msg_body>.+)$"
            ]
        }
    ]
},
```

   The `data_source` is very important because FortiAnalyzer will use this pattern to decide if devices match the parser. If there is a match, FortiAnalyzer will auto assign the device by using this parser to parse raw logs into siemdb correctly.

   FortiAnalyzer use POSIX regular expressions to check for matches with the patterns. For more information about POSIX regular expressions, see Regular Expressions/POSIX Basic Regular Expressions in Wikibooks.

   Also, in addition to checking for a match, patterns will assign some variables by value from the content of raw logs. The variable is defined by `?<…>`. For example, from above patterns: `?<header>` , `?<dhcp_type>`, and `?<msg_body>` which will be used in the `parser` described below.

3. `parser`: Includes two sub parts: `rules` and `field_map`.

```
"parser": {
    "rules": {        # parse into some variables by different rules
        “rules_1”: { }
```

```
        “rules_2”: { }
        “rules_3”: { }
        ……
    }
    "field_map": {        # value to fields in siemdb table
        "data_sourceid": [        # data_sourceid is one field name in siemdb, valued by
variable
    “devid” or “devide_id” (        # the former has high priority: if devid is
NULL,  valued by device_id)
        "devid" , "device_id"        # variable “devid” or "device_id" should be valued in
former
    data_source/rules part
        ],
        "data_sourcename": [
            "data_source_name"
        ],
        "data_sourcetype": [
            "data_sourcetype"
        ],
        "data_sourceversion": [],        # data_ sourceversion is one field name, but valued
by null
        "data_timestamp": [
            "data_timestamp"
        ],
        ……
    }
}
```

For more information about fields in the siemdb, see SIEM database on page 20.

4. `rules` within the `parser`: Includes the `trigger` and the subsequent `actions`. In other words, if "`trigger`", then do "`actions`". Below are three example rules.

   Example 1

```
"rule_inti_func": {        # init mode
    "description": "Bypass the sub action func is not copied.",
    "actions": [        # no any trigger, must do “action”
        "header.contains(\"T\")",       # contains is predefine function
        "data_sourcetype = \"Linux DHCP\""
    ]
},
```

   Example 2

```
"rule_dhcp_type": {        # trigger mode
    "description": "parse dhcp_type",
    "trigger": {        # have one trigger: if meet "${condition_1}", do “action”; if not
meet, do nothing
        "expression": "${condition_1}",
        "condition_1": "dhcp_type.is_not_empty()"    # is_not_empty() is predefine function
in FortiAnalyzer
    },
    "actions": [
        "eventType = concat(\"Linux\", \"_\", dhcp_type)",
        "eventAction = \"0\""
    ]
},
```

Example 3

```
"rule_2": {        # complex  mode
    "description": "parse msg_body to set fields",
    "trigger": {
        "expression": "${condition_1} and ${condition_2}",
        "condition_1": "dhcp_type.is_not_empty()",
        "condition_2": "msg_body.is_not_empty()"
    },
    "actions": [        # in “actions”, use if/else logic to value variable
        {
        "trigger": {        # define new trigger
            "expression": "${condition_1}",
            "condition_1": "dhcp_type == \"DHCPDISCOVER\""
        },
           "if": {
            "trigger": [        # meet condition to trigger
            "hostMACAddr = msg_body.match(\"from[[:space:]]+([[:word:][:punct:]]+)
[[:space:]]+via[[:space:]]+[[:word:][:punct:]]+\")"        # use match function of POSIX
Regular Expressions to value one or more variables
            ]
        }
    },
    ……
    ]
 },
```

For more information about predefined functions available in FortiAnalyzer, see Predefined functions on page 17.

Predefined functions

FortiAnalyzer includes some predefined functions that can be used in custom log parsers. For example, `is_not_empty(),to_timestamp(),is_valid_ip()`, and so on.

For a complete list of predefined functions, see the table below.

Function	Description	Example
`format2(format_str, format_parameter_1, format_parameter_2)`	This function is *only* used to format two strings.	`format2("%s-%s", "a", "b")` returns `"a-b"`
`format(format_str, ...)`	This function is used to multiple parameters.	`format("<%s>%s</%s>", "a", "b", "c")` returns `"<a>b</c>"` `format("This is %s", "one")` returns `"This is one"`
`concat(...)`	This function is used to multiple parameters.	`concat(<empty>, <empty>)` returns `""` `concat("12", "345", "6789")` returns `"123456789"`

`coalesce(...)`	This function returns first string in {p1, p2, ...} which is not empty (NULL).	`coalesce("p1","p2", <empty>)` retruns `"p1"`
		`coalesce(<empty>,"p2","p3")` returns `"p2"`
		`coalesce(<empty>, <empty>, "p3","p4")` returns `"p3"`

`is_in(...)`	This function returns true if given string *is* in {p1, p2, ...}, else returns false.	If foo is <empty>, `foo.is_in("1", "2", "3")` returns `False` If foo is "3", `foo.is_in("1", "2", "3")` returns `True`
`is_not_in(...)`	This function returns true if given string *is not* in {p1, p2, ...}, else returns false.	If foo is <empty>, `foo.is_not_in("1", "2", "3")` returns `True` If foo is "3", `foo.is_not_in("1", "2", "3")` returns `False`

`segment(src, split)`	This function returns two strings from 'src' by using 'split' as segment.	`ret1, ret2 = segment("this is a test.", " [-]")`
		`ret1 = "this is a test."` and `ret2 = ""`
		
		`ret1, ret2 = segment("this is HEAD. - - - this is MSG.", " [-] [-] [-] ")`
		`ret1 == "this is HEAD."` and `ret2 == "this is MSG."`

`starts_with (str)`	This function returns True if given string starts_with from 'str', else returns False.	If foo is "text", `foo.starts_with(<empty>)` returns `False` If foo is <empty>, `foo.starts_with("text")` returns `False`

Function	Description	Example
		If foo is "text-1", `foo.starts_with("text")` returns `True` If foo is "text-1", `foo.starts_with("ext")` returns `False`
`contains (l,str)`	This function returns True if given string contains 'str', else returns False.	If foo is "text", `foo.(<empty>)` returns `False` If foo is <empty>, `foo.("text")` returns `False` If foo is "0-text-1", `foo.("text")` returns `True`
`not_contains (l, r)`	This function returns True if given string *not* contains 'str', else returns False.	If foo is "text-1", `foo.not_contains("exit")` returns `True` If foo is "text-1", `foo.not_contains("text-1")` returns `False`
`is_empty ()`	This function returns True if given string is <empty>, else returns False.	If foo is <empty>, `foo.is_empty()` returns `True` If foo is "", `foo.is_empty()` returns `False` If foo is "123.456", `foo.is_empty()` returns `False`
`is_not_empty(l)`	This function returns True if given string is *not* <empty>, else returns False.	If foo is <empty>, `foo.is_not_empty()` returns `False` If foo is "", `foo.is_not_empty()` returns `True` If foo is "123.456", `foo.is_not_empty()` returns `True`
`replace(src, tag, rep)`	This function replace 'tag' by 'rep' in given string 'src'.	`replace("LUA magic char ^%()[]-?+.*$", "^%()[]-?+.*$", "is passed.")` returns `"LUA magic char is passed."` `replace("LUA is cute.", "cute.", "great!")` returns `"LUA is great!"`

`trim(src)`	This function trim more space before or after given string 'src'.	`trim(" ")` returns `""`
		`trim(" trim result ")` returns `"trim result"`
		`trim("123")` returns `"123"`

`is_ipv4()`	This function returns True if given string is a valid ipv4 address, else returns False.	If foo is "172.18.3.47", `foo.is_ipv4()` returns `True`
`is_ipv6()`	This function returns True if given string is a valid ipv6 address, else returns False.	If foo is "172.18.3.47", `foo.is_ipv6()` returns `False`
`is_valid_ip()`	This function returns True if given string is a valid ipv4 or ipv6 address, else returns False.	If foo is "172.18.3.47", `foo.is_valid_ip()` returns `True`

Function	Description	Example

`to_timestamp(year, month, day, hour, min, sec, tz_abb)`	This function returns timestamp from all parameters. If no tz_abb, default tz is "GMT".	`to_timestamp ("2019","12","12","15","40","20","GMT+01")` returns `"1576161620"`
		`to_timestamp ("2019","12","12","15","40","20","-0100")` returns `"1576168820"`
		`to_timestamp ("2019","12","12","15","40","20","UTC+01")` returns `"1576161620"`
		`to_timestamp ("2019","12","12","15","40","20",)` returns `"1576165220"`

`to_timestamp_by_date_time(date, time, tz_abb)`	This function returns timestamp from date, time, and tz_abb. If no tz_abb, default tz is "GMT".	`to_timestamp_by_date_time("2019-12-12","15:40:20")` returns `"1576165220"` `to_timestamp_by_date_time ("12/Dec/2019","15:40:20")` returns `"1576165220"`
`to_timestamp_by_date_time_gmt (date, time)`	This function returns timestamp from date, time, and tz_gmt (default).	`to_timestamp_by_date_time_gmt(<empty>, <empty>)` returns `<empty>` `to_timestamp_by_date_time_gmt("2019-12-12","15:40:20")` returns `"1576165220"`
`epoch_timestamp(ts)`	This function returns timestamp from given timestamp string.	`epoch_timestamp("1572894043")` returns `"1572894043"` `epoch_timestamp("1583281149018500794")` returns `"1583281149"`
`is_json_str()`	This function returns True if given string is JSON string, else returns False.	If foo is "{\"key\": value}", `foo.is_json_str()` returns `True`

`match_key_value(src, eq, seg)`	This function matches "key(eq)value(seg)" log format in given string 'src'.	If foo is "key1=value1,key2=value2":
		`match_key_value(foo, "=", ",")`
		`${_key1}` is `"value1"`
		`${_key2}` is `"value2"`

`match_logfmt(s)`	This function matches "key=value" log format, 'logfmt', including escape double string case.	If foo is "key1=123 key2=\"this is a test\" key3=true":
		`match_logfmt(foo)`
		`${_key1}` is `"123"`
		`${_key2}` is `"this is a test"`
		`${_key3}` is `"true"`

SIEM database

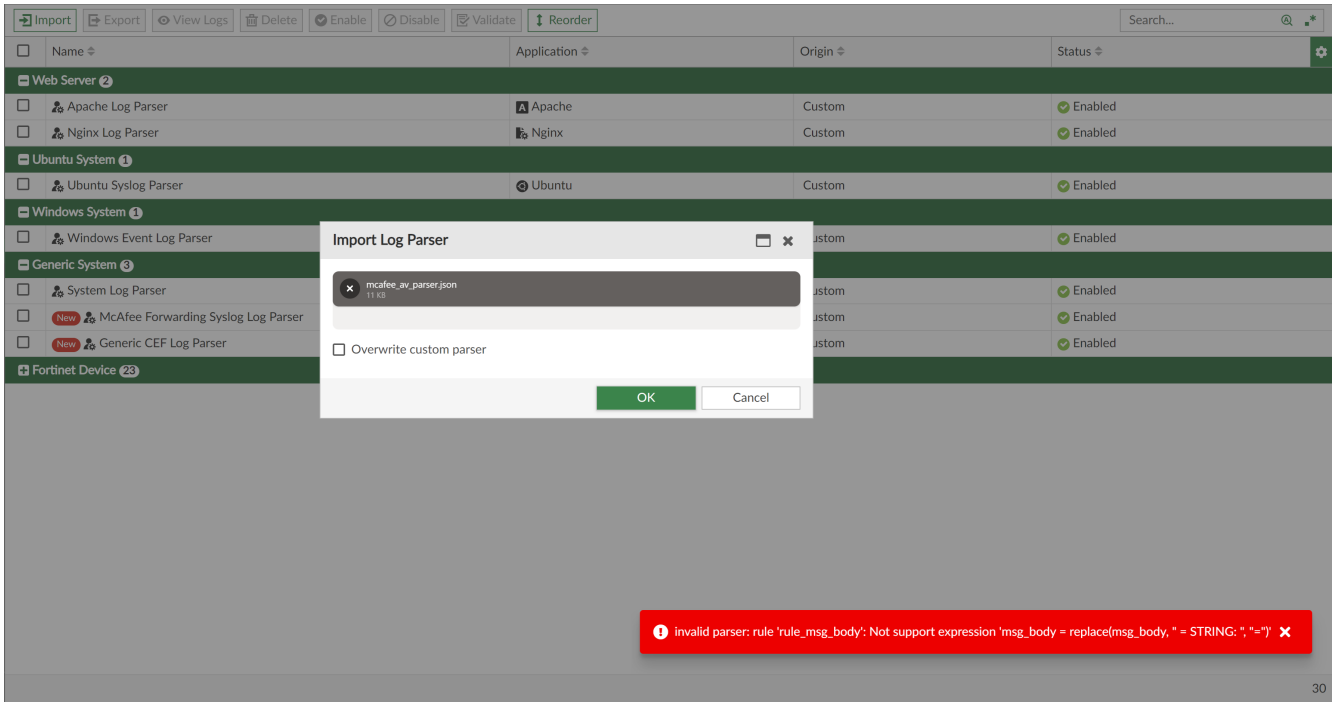
The aim of log parsers is to parse raw logs into the siemdb. FortiAnalyzer will parse the content of raw log into fields and later analyze these fields. Thus, it is important to know what fields are in the siemdb.

For a list of fields, see the FortiAnalyzer Fabric Normalization Reference in the Fortinet Document Library. Normalized fields in the siemdb can change, so you must review the document for your version of FortiAnalyzer.

For example, see the fields available for the latest patch release of FortiAnalyzer 7.6 Fabric Normalization Reference.

Importing a custom log parser

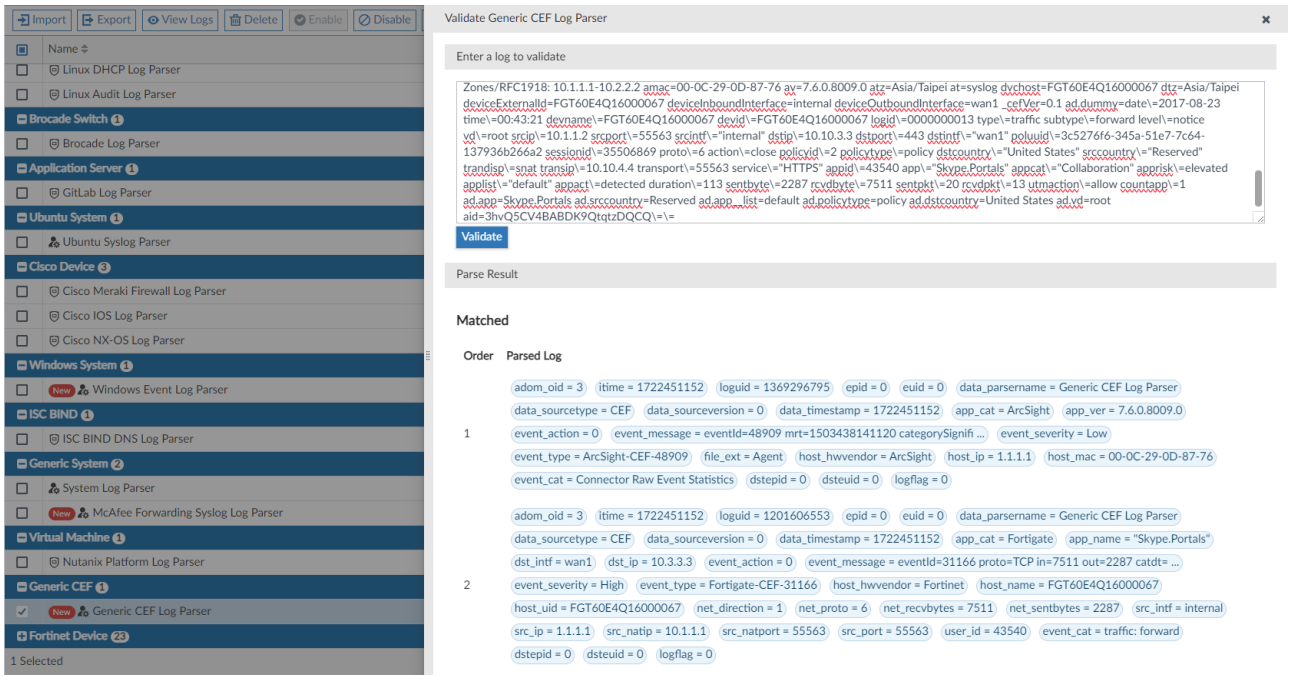
After a custom log parser is created, it can be imported into FortiAnalyzer from *Incidents & Events* > *Log Parser* > *Log Parsers*. The custom log parser must be a JSON file type. During the import process, FortiAnalyzer verifies if the file has errors. If there are errors in the file, it will fail to import and FortiAnalyzer displays an error message.



If the import is successful, you can validate the parser to confirm the field-map is correct.

To validate a custom log parser:

1. In *Incidents & Events* > *Log Parser* > *Log Parsers*, select the checkbox for the custom log parser.
2. Click *Validate*.
   The *Validate Log Parser* pane opens.
3. Enter a log to validate and click *Validate*.
   A *Parse Result* displays in the *Validate Log Parser* pane.



To enable the custom log parser:

1. In *Incidents & Events* > *Log Parser* > *Log Parsers*, select the checkbox for the custom log parser.
2. Click *Enable*.

www.fortinet.com