Fargo

Theory of Operations

Configuration

Management

What is Fargo?

- Logging and processing component
- Useful as part of HotZone
  - Also useful as a standalone tool
- Basic Logging properties and advanced log processing
  - Backed by an *InSAneLy* complicated configuration language!! (Just kidding!)

Basic Logging Properties

(desirable in any logging system)

- Parsing
- Suppression
- Statistics
- Consolidation
- Coalescing
- Output
- Operational

Basics: Parsing

- The process of making “sense” out of log data rather than just treating it as strings
  - Ideally, detailed as possible: pick out “standard” fields within data (e.g.: date)
  - Import of external log data (e.g.: merging syslog data with http log data)
  - Enables matching and comparison of log records by field values and identifiers (the fundamental of advanced log analysis)

Basics: Suppression

- Too many log messages can easily overwhelm a human administrator
- Typically 2 processes:
  - Discarding - throw away messages of a certain type
  - Counting - count incidence of messages of a certain type over time (e.g.: 10,000 lines of code-red generated web logs get turned into 1 line reading “10,000 code red scans”)

Basics: Statistics

- Maintain statistics on messages, their suppression, and state of the system
  - Messages per second, inter-arrival times, etc.
- Trending
  - Are rates increasing/decreasing?
  - Averages
  - Variance (standard deviations/ average distance from mean)
  - Floor / Ceiling processing

Basics: Consolidation

- Move log data from multiple systems to a single location for combined processing
  - Requires reliable and secure delivery
  - Requires unique event-Ids
  - Requires record system-of-origin information
- This is a hard problem to solve without a database! (and databases bring their own problems)

Basics: Coalescing

- Joining multiple related messages into a single message
  - Requires good / accurate parse of incoming data
  - Requires knowledge-base that directs the coalescing process
    - Automating (machine learning) coalescing is an AI problem we're not trying to solve

Basics: Output

- Convert parsed and coalesced/processed messages *back* into human readable form
- Export into other formats can be done by programming output rules
  - Possibly into a format suitable for use in databases
  - ...or even HTML

Basics: Operations

- Load and unload configurations dynamically
- Do the right thing when:
  - New rules added
  - Existing rules are updated
  - New message types are seen
  - Syntax errors are found in configuration rules

High-Level Flow

- Parsing messages
- Assigning defaults
- Rewriting attributes
- Discarding

- Counting messages
- Checking triggers
- Coalescing logic

- Rewriting output
- Vectoring output to various channels by type
- Long-term storage

External Parse

```
whumpus$ stdinlogger -o /tmp/xx -N -C -c logger.httpdrules < http_log
no body match ""GET / HTTP/1.1" 304 -"
        (header "10.10.10.1 - - [17/Dec/1999:14:07:39 -0500]")
whumpus$
```

Log Data

Http log record

```
10.10.10.1 - - [17/Dec/1999:13:57:42 -0500] "GET / HTTP/1.1" 200 1565
```

```
header "$s - - [$d/$s/$d:$d:$d:$d $s] "
{
        <date> = "$2/$3/$4/$5:$6:$7"
        <srcprog> = "httpd"
        <srcpid> = "unknown"
        <srchost> = "$1"
}
```

Header parse rule matches

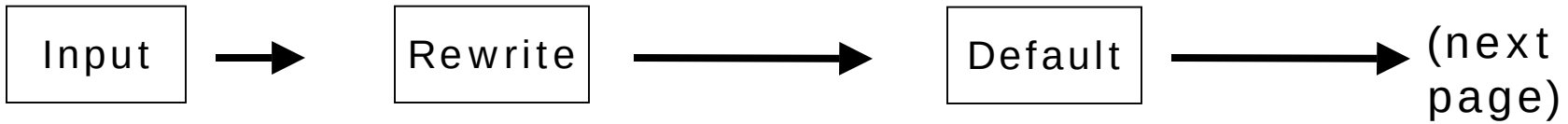
```
'"GET $s $s" $d $d' {
        <url> = "$1"
        <httpvers> = "$2"
        <size> = "$4"
}
```

Body parse rule matches

Output record in XML

```
<REC>
<DATE>17/Dec/1999/13:57:42</DATE>
<SRCPROG>httpd</SRCPROG>
<SRCPID>unknown</SRCPID>
<SRCHOST>10.10.10.1</SRCHOST>
<URL>/</URL>
<HTTPVERS>HTTP/1.1</HTTPVERS>
<SIZE>1565</SIZE>
</REC>
```

Input Flow



- Syntax checks
- Modifying attributes based on matching

```
rewrite  {
        attr <pri> lessthan ”5"
} as {
        attr <pri> = ”9”
}
```

- Apply default attributes where missing

```
default
{
        attr <DWELLTIME> = "99"
}
```

```
<rec>
<pri>8</pri>
</rec>
```

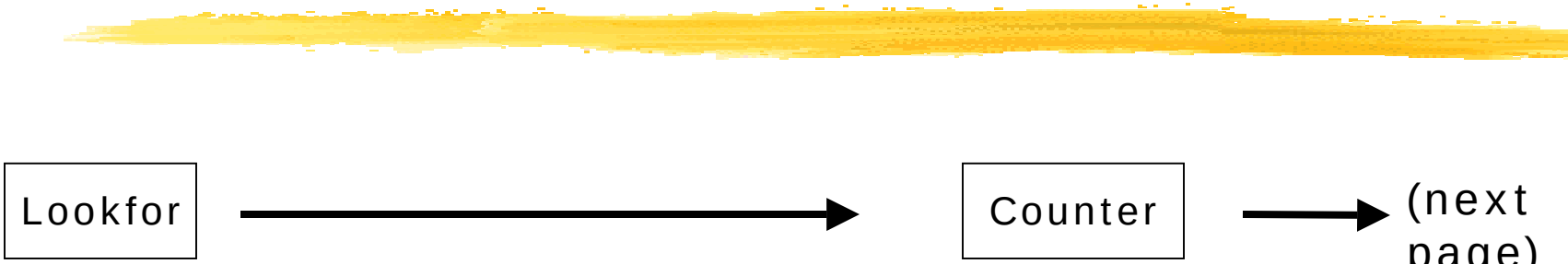
→

```
<rec>
<pri>9</pri>
</rec>
```

→

```
<rec>
<pri>9</pri>
<dwelltime>99</dwelltime>
</rec>
```

Input Flow *(cont)*



- Look for matching attributes and generate a new record if none seen within time window

Each lookfor is uniquely named

```
lookfor heartbeat
{
        attr <msg> contains “mark"
}
        120 seconds
{
        attr <msg> = ”missed beat”
        attr <sev> = “9”
}
```

- Count instances of matching values

```
counter highpri count SEV as TSEV {
        attr <sev> morethan ”6"
} 600 seconds {
        attr <msg> = ”summary of high
            severity events"
}
```

Since the message <SEV> is more than 6 it is added to the counter

```
<rec>
<pri>9</pri>
<dwelltime>99</dwelltime>
</rec>
```

In this case there is no match so the lookfor remains un-updated

```
<rec>
<pri>9</pri>
<dwelltime>99</dwelltime>
</rec>
```

Input Flow *(cont)*

Discard → Aggregation Process

- Look for matching attributes & delete any records that completely match

```
discard
{
        attr <sev> lessthan “2"
}

discard
{
        attr <srcprog> = “sendmail”
        attr <sev> lessthan “5”
}
```

```
<rec>
<pri>9</pri>
<dwelltime>99</dwelltime>
</rec>
```

→

```
<rec>
<pri>9</pri>
<dwelltime>99</dwelltime>
</rec>
```

In this case there is **no** match so the record is not tossed

Aggregation

Aggregation

- Look for sets of records that have clusters of matching attributes and rewrite them into new records with new attributes

```
<rec>
<pri>9</pri>
<dwelltime>99</dwelltime>
<msg>foo</msg>
</rec>
```

```
<rec>
<pri>9</pri>
<dwelltime>99</dwelltime>
<msg>bar</msg>
</rec>
```

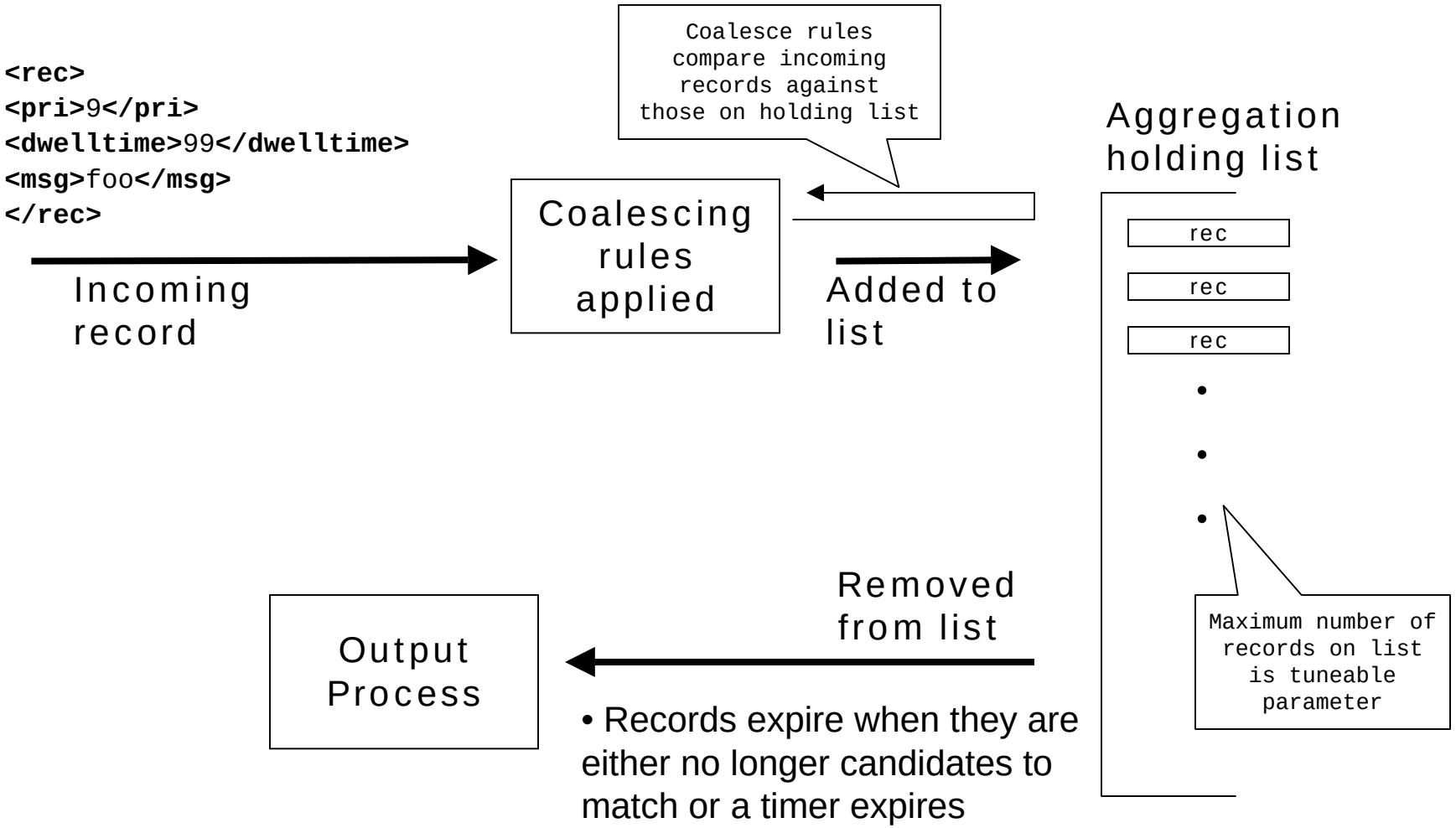
Coalesce rules look for several matching sets of records

```
coalesce
{
        attr <msg> contains ”foo"
        attr <pri> = ”99"
} preserve ( pri )
{
        attr <msg> contains ”bar"
} into {
        attr <msg> = ”foo bar attack!”
}
```

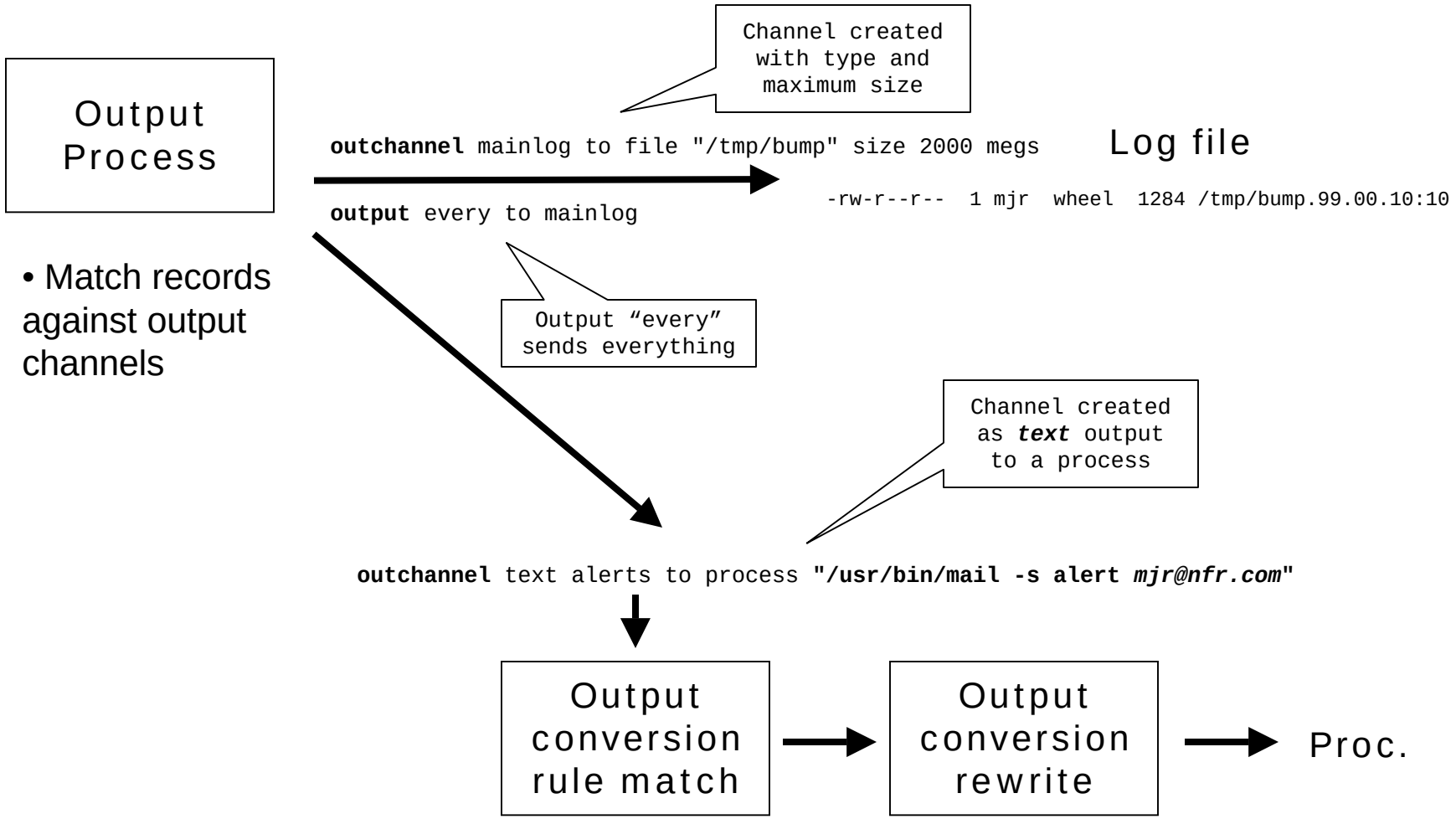
New record created

```
<rec>
<pri>99</pri>
<msg>foo bar attach!</msg>
</rec>
```

Aggregation



Output



- Match records against output channels

Output Conversion

Output Conversion Process

- Match records against output channels and output rewrite rules

```
outchannel text alerts to process "/usr/bin/mail -s alert mjr@nfr.com"
```

```
<rec>
<pri>9</pri>
<dwelltime>99</dwelltime>
<msg>code red scan</msg>
</rec>
```

```
output  {
                attr <pri> = ”9"
        }
        to alert as
        {
                "this is a system alert!\n"
                "\tpriority was ${PRI}"
                ”\t${MSG}”
                "message-id: ${ID}"
        }
```

Conversion

```
From: mjr
To: mjr@nfr.com
Subject: alert

this is a system alert!
        priority was 9
        code red scan
message-id:916011614.28192@whumpus
```

Programming Input

- Programmed rules for:
  - Rewriting attributes of selected records
  - Applying defaults where attributes do not exist
  - Counting records / checking triggers
  - Discarding records

Programming Correlation

- Programmed rules for:
  - Matching clusters of records
  - Turning clusters of records into new records and discarding the coalesced records
    - When coalescing occurs the “ancestor” record-Ids are included in the descendant record as “references”

```
<REC>
<REFERENCES>915983439.35746@whumpus,915983439.18838@whumpus,91598
   3439.27083@whumpus</REFERENCES>
...
```

Programming Output

- Programmed rules for:
  - Matching records when they are ready to be expired from the system
    - Discarding them
    - Saving them to long-term storage
    - Turning them into alerts and delivering them immediately
    - Bulking them up into reports to be delivered intermittently

Internal Storage

- Records are stored as an array of **attribute=value** pairs
  - No data typing; everything is treated as strings
  - **<** and **>** are used as external delimiters and are automatically replaced with HTML **<** **>**

Sample Record

- This was produced by logparser against an http daemon log on OpenBSD

```
<REC>
<SYSLOGMSG>10.10.10.1 - - [17/Dec/1999:13:57:42 -0500] "GET
   /openbsdpower.gif HTTP/1.1" 200 3334</SYSLOGMSG>
<DATE>17/Dec/1999/13:57:42</DATE>
<SOURCEPROG>httpd</SOURCEPROG>
<SOURCEPID>unknown</SOURCEPID>
<SOURCEHOST>10.10.10.1</SOURCEHOST>
<URL>/openbsdpower.gif</URL>
<HTTPVERS>HTTP/1.1</HTTPVERS>
<SIZE>3334</SIZE>
</REC>
```

Differences W/XML

- The record format used by Fargo is ***similar*** to XML
  - Designed to be consumable by XML parsers
  - **Does *not* support** nesting attributes or duplicated attributes
  - ***Requires*** termination tags
    - Invalid: **<ATTR>This attr has no end tag**
    - Valid: **<FOO>End tags are good</FOO>**

Rules and Rule Storage

- Rules are stored in /cf/fargo/rules
  - Fargo periodically (default: every 4 minutes) scans for new rules files
  - If a new file is found it is automatically loaded
  - If an existing file is replaced/updated all existing rules belonging to that file are first unloaded and then the file is reloaded
    - No need to restart/kill or signal Fargo

Input Engines: Loggers

- Multiple loggers capable of slightly different ways of getting input
- Each apply same kind of logic to parse messages into output
- Can output to file or fifo with locking based on command line
  - Inline (realtime) processing or offline (batch or post facto) analysis

Klogger

- Reads /dev/klog in a blocking read( ) loop
  - Extremely efficient, low CPU-usage
  - Usually prepends date/time stamp to beginning of log messages (on BSD)
  - Periodically outputs statistics
  - Runs as a daemon (detached from controlling terminal)
  - Writes to /cf/interact/wc.fifo

Udplogger

- Implements standard “syslogd” functionality
  - Listens on UDP port 514 in blocking read( )
  - Very CPU efficient but may still lose packets due to semantics of UDP layer
  - May add date/time stamp if not present
  - Periodically outputs statistics
  - Normally is ***not*** running on a HotZone
  - Writes to /cf/interact/wc.fifo

Devlogger

- Implements standard “syslog” /dev/log functionality
  - Very similar to udplogger but local machine only (unix domain socket)
  - Normally is ***not*** running on a HotZone

Filelogger

- File reader for arbitrary files
  - Monitors file in a read( ) / sleep ( ) loop
  - Checks for inode change or file size rollover and reopens file on demand
  - Useful for processing other logs (e.g.: httpd logs) into Fargo
  - Normally is ***not*** running on a HotZone

Stdinlogger

- One-shot processor for files or standard input
  - Particularly useful for rule-base testing with debugging turned on

```
hussar.nfr.com$ stdinlogger -d -N -C -c logger.httpdrules < httpd_log
"10.10.10.1 - - [17/Dec/1999:13:57:42 -0500] "GET / HTTP/1.1" 200 1565"
        selected header rule "$s - - [$d/$s/$d:$d:$d:$d $s] "
        body=""GET / HTTP/1.1" 200 1565"
        selected body rule ""GET $s $s" $d $d”
<REC>
<SYSLOGMSG>10.10.10.1 - - [17/Dec/1999:13:57:42 -0500] "GET / HTTP/1.1" 200 1565</SYSLOGMSG>
<DATE>17/Dec/1999/13:57:42</DATE>
<SOURCEPROG>httpd</SOURCEPROG>
<SOURCEPID>unknown</SOURCEPID>
<SOURCEHOST>10.10.10.1</SOURCEHOST>
<URL>/</URL>
<HTTPVERS>HTTP/1.1</HTTPVERS>
<SIZE>1565</SIZE>
</REC>
```

Logger Kludges:

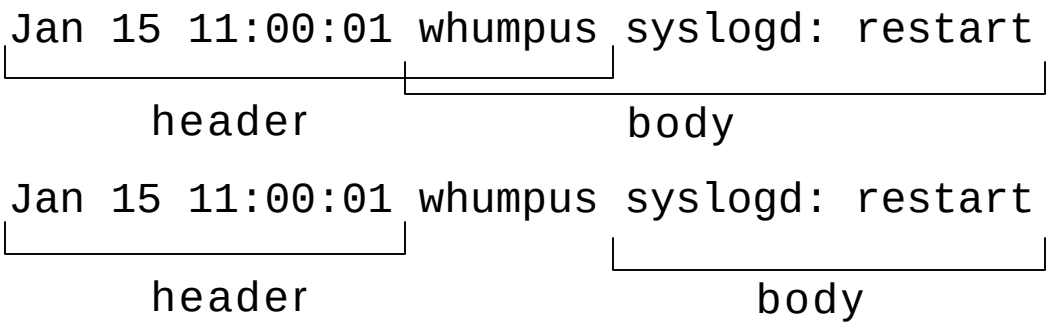
- Adding dates to badly formatted messages (may be turned off with -N flag)
  - Necessitated by some kernel implementations and Linux implementations
- Adding year to some messages
  - Necessitated by many syslog implementations
- Ability to disable locking on output file for performance if not needed

Log Parsing Rules

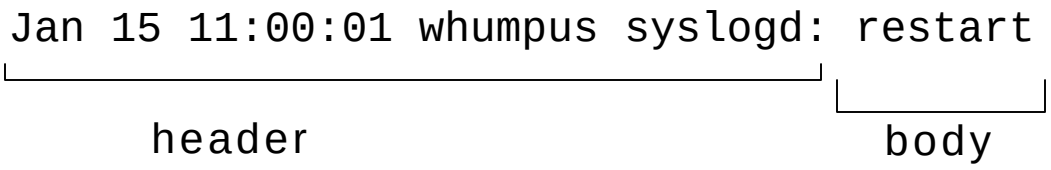
- Header parsing
  - Headers are treated separately from message bodies
  - Oddly many versions of UNIX have standard message bodies but nonstandard headers
- Message body parsing
  - Once a header has been matched look for best body match that combined with matched header

Log Parsing Rules *(Cont)*

- Bad:

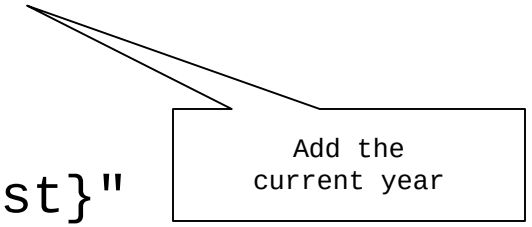


- Good:



Sample Header Rule

```
header "$s $d $d:$d:$d $s[$d]: " {
        <date> = "$1/$2/${year}/$3:$4:$5"
        <sourceprog> = "$6"
        <sourcepid> = "$7"
        <sourcehost> = "${host}"
}
```



- Matches most fields of a header for OpenBSD
  - Notice that since syslog doesn't add year to the date (what ***were*** they thinking?!!?) the header rewrite rule adds the current year automatically

Sample Header Rule

```
header "$s $d $d:$d:$d $s: " {
        <date> = "$1/$2/${year}/$3:$4:$5"
        <sourceprog> = "$6"
        <sourcepid> = "unknown"
        <sourcehost> = "${host}"
}
```

It's not there!

Puts the current host name in

- Matches most fields of a header for OpenBSD processes that didn't request LOG_PID in openlog( )
  - Since we don't know the pid we just stuff a value in as a place-holder

Syntax of Header Rules

header “match string" **{***output productions* **}**
output productions are a list in the form of:
<attribute> = “output string”

- There can be only one match string
  - String must completely match beginning of input line
- There can be multiple output productions
  - Will be used to form data record if there is a matching body rule found

Writing a Header Rule

- Take a look at your systems' log messages and identify the largest prefix set that doesn't change layout from record to record
  - Write some test match strings and run them with stdinlogger -d to turn on match debugging
  - Then write output rules that normalize to reasonable values

Writing a Header Rule *(cont)*

```
10.10.10.1 - - [17/Dec/1999:13:57:42 -0500] "GET / HTTP/1.1" 200 1565
10.10.10.1 - - [17/Dec/1999:13:57:42 -0500] "GET /bsdpower.gif HTTP/1.1" 200 3334
10.10.10.1 - - [17/Dec/1999:13:57:42 -0500] "GET /bsd_small.gif HTTP/1.1" 200 4090
10.10.10.1 - - [17/Dec/1999:13:57:42 -0500] "GET /blowfish.jpg HTTP/1.1" 200 41318
10.10.10.1 - - [17/Dec/1999:13:57:42 -0500] "GET /lock                 l.1" 200 5417
10.10.10.1 - - [17/Dec/1999:13:57:42 -0500] "GET /logo                 /1.1" 200 46246
10.10.10.1 - - [17/Dec/1999:13:57:43 -0500] "GET /logo24.jpg HTTP/1.1" 200 35570
10.10.10.1 - - [17/Dec/1999:13:57:43 -0500] "GET /smalltitle.gif HTTP/1.1" 200 2220
10.10.10.1 - - [17/Dec/1999:14:07:39 -0500] "GET / HTTP/1.1" 304 -
```

Header

Body

- Pretty obvious, huh?
  - 10.10.10.1 : looks like a client IP address
  - - - ? Who knows?
  - [dd/mm/yy:hh:mm:ss -GMToffset] looks like date
- Try: "$s - - [$d/$s/$d:$d:$d:$d $s] "

Input Matching ($-tokens)

- Tokens are matched in the input string as $-value which map to $-number in the output strings
  - First match is $1, second $2, etc
  - Maximum system supports is 500 matches totalling 2048bytes of data (plenty)

Input Matching ($-tokens)

- Types:
  - $s - string
  - $d - integer
  - $f - floating point number
  - $* - rest of line (use with ***caution***!)
  - whitespace (multiple spaces *match*) as do tabs! (I.e.: “foo bar” matches “$s $s”)
  - $$ - a dollar sign
  - Everything else is a literal string match

Input Matching ($-tokens)

- Types can be preceeded by a numeric length in characters
  - $5s - exactly 5 characters
- This can be used to chop tightly packed values apart:

  "$3s$4s" will match "*catfood*" and "*dogfood*"

  setting $1 to "*cat*" and "*dog*" respectively

  and $2 to "*food*"

Input Matching ($-tokens)

```
hussar.nfr.com$ stdinlogger -N -d -c example
catfood
"catfood"
        selected header rule "$3s” from example/header
        body="food"
        selected body rule "food” from example/bod
<REC>
<SYSLOGMSG>catfood</SYSLOGMSG>
<ANIMAL>cat</ANIMAL>
<FOOD>food!</FOOD>
</REC>
```

Typed to standard input

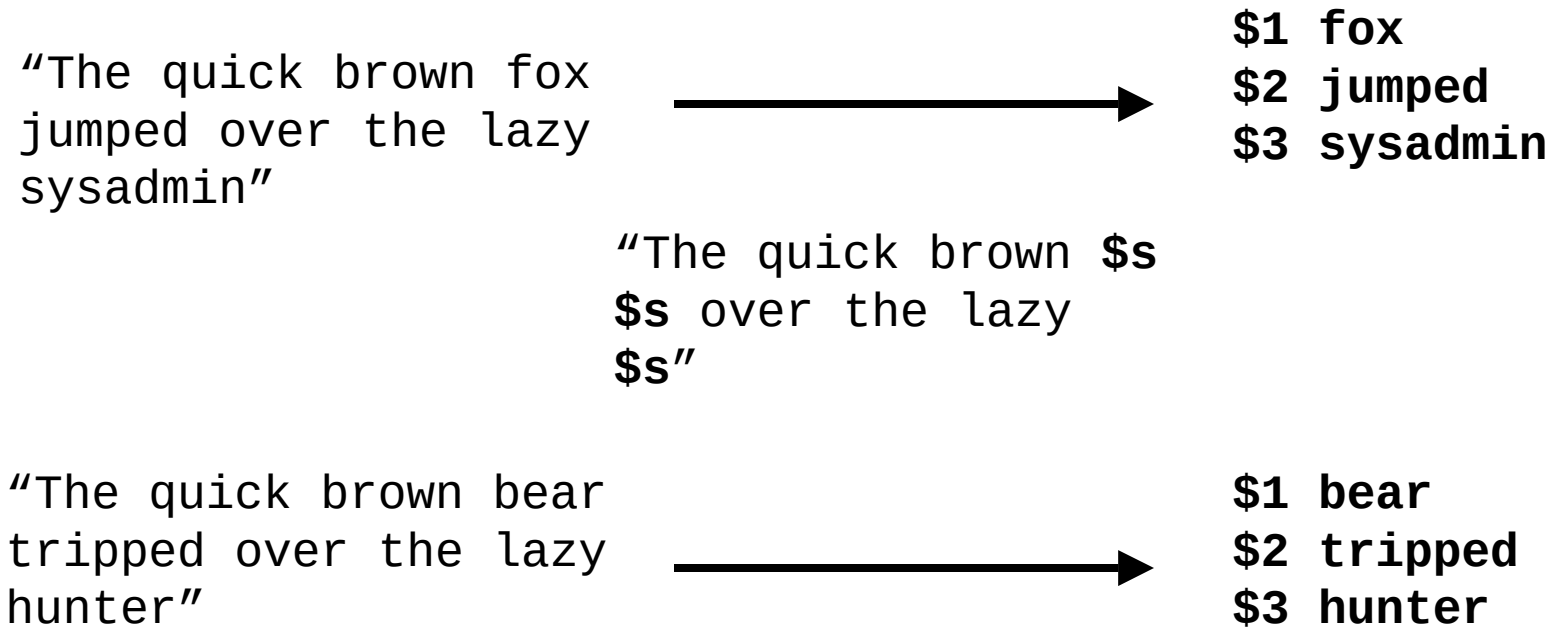
This is the directory for our example rule base

This is the debug output (-d) note how it gives the rule and the file it’s from

- Note how it puts the original record in <syslogmsg> by default

Output Rewriting ($-out)

- Output rewriting is based on the position of the match value in the input:



Special $-values

- ${host}
  - In case you need a host name but the record doesn’t have one
  - Outputs the current host’s name gotten by gethostname( )
- ${year}
- ${month}
  - Outputs date elements of current date

Sample Body Rule

```
"BAD SU $s to $s on /dev/$s" {
        <user> = "$1 to $3"
        <tty> = "$3"
}
```

- Matches a typical BSD “BAD SU” message
  - Normalize the “user” attribute into something reasonable
  - Normalize the tty attribute to something reasonable we might later combine with output from bad login messages from telnetd (for example)

Syntax of Body Rules

```
“match string" {output productions }
output productions are a list in the form of:
<attribute> = “output string”
```

- There can be only one match string
  - String must completely match *starting where the header match rule left off*
  - Remainder of body must match completely to end of line in order to be accepted (be *careful* not to overuse $*)

Writing a Body Rule

```
10.10.10.1 - - [17/Dec/1999:13:57:42 -0500] "GET / HTTP/1.1" 200 1565
10.10.10.1 - - [17/Dec/1999:13:57:42 -0500] "GET /bsdpower.gif HTTP/1.1" 200 3334
10.10.10.1 - - [17/Dec/1999:13:57:42 -0500] "GET /bsd_small.gif HTTP/1.1" 200 4090
10.10.10.1 - - [17/Dec/1999:13:57:42 -0500] "GET /blowfish.jpg HTTP/1.1" 200 41318
10.10.10.1 - -                              :13:57:42 -0500] "GET /lock.gif HTTP/1.1" 200 5417
10.10.10.1 - -                              :13:57:42 -0500] "GET /logo23.jpg HTTP/1.1" 200 46246
10.10.10.1 - - [17/Dec/1999:13:57:43 -0500] "GET /logo24.jpg HTTP/1.1" 200 35570
10.10.10.1 - - [17/Dec/1999:13:57:43 -0500] "GET /smalltitle.gif HTTP/1.1" 200 2220
10.10.10.1 - - [17/Dec/1999:14:07:39 -0500] "GET / HTTP/1.1" 304 -
```

Ignore

- Now parse the right hand side
  - “GET”: a string literal
  - Next token is URL
  - Then protocol ID, return code, and file size…
- Try: ‘“GET $s HTTP/$f “ $d $d’

Writing a Body Rule *(cont)*

Try: ‘“GET $s HTTP/$f “ $d $d’

A single and a double

This quote is treated as a literal for matching purposes

- Notice the quotes?
  - String matchers can be quoted either with matching single quotes (‘) or double (“) as long as they are ***balanced***
  - Enclosing the “ in single quotes lets it match the “ as literals in the match string

Example: sendmail log

```
"$s: to=$s, ctladdr=$s ($d/$d), delay=$d:$d:$d, xdelay=$d:$d:$d, mailer=$s, stat=$s"
{
        <status> = "$13"
        <recipient> = "$2"
        <sender> = "$1"
}
```

You don't have to use all (or even any) of the matched tokens in your output rule

- OK, that's ugly! I admit it!
  - But then just about *everything* to do with *sendmail* is... ;)

Transmitting into Fargo

- The various logger agents “know” the location of the Fargo fifo and will send records to it by default
- Ex: Normal invocation of devlogger:

```
devlogger
```

  - (It understands the rest)
- You could use:

```
devlogger -o /cf/interact/wc.fifo
```

Transmitting into Fargo *(cont)*

- By default locking is enabled for loggers that talk through the fifo
  - You can turn it off if you only have a single input source (but why? Locking is *fast*)
- If you want output to a file just use -o file

```
udplogger -o /tmp/udpcaptured.out
```

- Don't detach from controlling tty

```
udplogger -d -e /tmp/log.errors
```

Matching Attributes

- Attribute matches are widely used within Fargo to match sets of attributes
  - For a record to match *all* the specified attributes must match without error

Matching Attributes

- Attribute matches accept syntax like:
  - <attr> regexp “expression string”
  - <attr> = “string”
  - <attr> exactly “string”

These are the same thing

  - <attr> contains “string”
  - <attr> lessthan “string” *(must be a #)*
  - <attr> morethan “string” *(must be a #)*

Matching Attributes

- Attribute match modifiers:
  - casesense
    - Make matches case sensitive; default is *not* to be case sensitive
    - <attr> casesense contain “sTudLyCaps”
  - not
    - Invert sense of match
    - <attr> not contain “foo”
    - <attr> not = “6”

Matching Example

```
rewrite  {
        attr <logmsg> contain ”this is a useless alert”
        attr <prio> lessthan “5”
} as {
        attr <logmsg> = ”junk”
        attr <prio> = “0”
}
```

- This is a simple example of matching a record on two rules (logmsg and prio) in the context of rewriting the record into something else
  - We might be doing this to map some meaningless alert into something that gets counted and tossed

Sample Rewrite Rule

```
rewrite  {
        attr <prio> morethan “6”
} as {
        attr <alert> = ”true”
}
```

- This is an example where we might attach a new attribute (“alert”) to a record based on the presence of a certain matching value
  - In this case we might have an output channel for anything that has `attr <alert> = “true”` going to a near-realtime notification system

Sample Rewrite Rule

```
rewrite {
        attr <prio> morethan “6”
} as {
        attr <alert> = ”true”
        attr <prio> = NULL
}
```

- Outputs modify *only* the attributes specified
  - Other attributes remain unaltered
  - Attributes can be deleted in a rewriting rule by setting them to NULL

Syntax of a Rewrite Rule

rewrite { match rules **} as {***output productions* **}**
match rules are a list of matching criteria
output productions are a list in the form of:
<attribute> = “output string”

- There can be a large number of match rules or output productions
  - Match rules are widely used in Fargo, as are output productions; they use the same syntax everyplace

Why do a Rewrite?

- Rewrites might be useful for converting one type of record into a different one
  - Or unsetting / resetting a value
  - Or tagging a record for some other purpose
    - Setting an attribute called “junk” so you can later match on “junk” and discard it
    - Setting an attribute called “urgent” so you can later match it in an output rule and treat it as special

Sample Default Rule

```
default {
   attr <prio> = “0”
}
```

- **Use default rules sparingly!**
  - Defaults will assign all attributes that are *un*assigned in the record when the default rule is applied

Syntax of a Default Rule

default {*output productions* **}**

output productions are a list in the form of:

<attribute> = “output string”

and are applied to any attribute in the output production list that is unassigned at the time when the default rule is applied

- There can be multiple separate default rules or a single large one
  - Do not make assumptions about the order in which they will be applied - do ***not*** duplicate or overlap output productions

Message-ID Defaults

- If Fargo gets a record that has no message-ID assigned:

```
<ID>916023922.16897@hussar.nfr.com</ID>
```

- ... it will ***make one up*** and assign it
  - Current algorithm is system time (seconds since the millenium) '.' and a pseudorandom number between 0 and 50,000 '@' hostname
  - The algorithm may change someday...

Why do a Default?

- Defaults might be used to provide information where there is none that makes sense
  - If there's no priority, assign one
  - If there's no host name, assign one

There's a secret attribute in Fargo *(shh!)* called <dwelltime> that can be used to override how *long* in seconds a record stays in the coalescing work area; reset this only for selected records with extreme care!! If you have a big machine with a lot of RAM you might want to bump it up slightly past WC_DEFAULT_DWELLTIME (see fargo.h)

Sample Discard Rule

```
discard {
        attr <prio> lessthan “3”
}
```

- Deletes matched records from the system
  - Note that this deletion takes place after counting and triggers are checked but before coalescing
  - This is really useful, since you can count the number of things you think are uninteresting and then throw them away

Syntax of a Discard Rule

discard {*match rules* **}**
deletes the record if it completely matches the match rules
deletion is performed after lookfor, counting, and triggering but before coalescing

- This is the easiest way to weed out noise records from your system

Why do a Discard?

- If you are getting a specific record that is irritating, make it go away
  - Can be very specific (if it's a sendmail retry message from a particular host discard it)
  - Can be very general (if it's a sendmail message discard it)
- Very useful for tuning/throttling load of useless information hitting the coalescing engine

LookFors

- What they do
  - Look for occurrence of a periodic event
  - Rollover based on a specified interval
  - Output a new record if they didn't see the periodic event within the specified interval
- What's it for?
  - Heartbeat monitoring
- Lookfor status is preserved on hard disk across reboot/restart

Sample Lookfor Rule

```
lookfor whumpusmark  {
        attr <prog> = “syslogd”
        attr <msg> contains “restart”
} 25 hours {
       attr <pri> = “8”
       attr <msg> = “whumpus syslogd did not get restarted”
}
```

- Watches for appearance of specified attributes in a record within a specified interval
  - Outputs result record if record not seen
  - Otherwise it “remembers” and stays quiet

Syntax of a Lookfor Rule

lookfor {*match rules* **} *time-interval* {** output productions **}**

monitors data stream for appearance of specified record

outputs productions if that record is not seen in time-interval

time intervals may be a combination of:

- N seconds
- N hours
- N minutes
- N days

e.g.: 1 day 2 hours = 93600 seconds

Sample Lookfor Output

```
<REC>
<PRI>8</PRI>
<LOOKFOR-RULENAME>whumpusmark</LOOKFOR-RULENAME>
<INTERVAL>60</INTERVAL>
<LASTSEEN>Sun Jan 10 22:06:56 1999</LASTSEEN>
<DWELLTIME>90000</DWELLTIME>
<MSG>whumpus syslog did not get restarted</MSG>
<ID>916024016.16451@whumpus.ranum.com</ID>
</REC>
```

- Note how the lookfor rule added some internal attributes (rule name, last seen time, etc.) to the output record

Why do a Lookfor Rule?

- In some cases the non-apparance of an event is an event!
  - Lookfors let you trigger an event when something you expected to see doesn't appear
    - Process restarts
    - Log rotations
    - System heartbeats

Counters

- What they do
  - Count events or summarize events within a specified time interval
  - Statistics are kept on the event and event rate as well as some basic trending inference
  - Output a new record at specified time interval when the counter rolls over
- Counter status is preserved on hard disk across reboot/restart

Sample Counter Rule

```
counter sendmailsumm sum MSGSIZE as TOTALBYTES {
        attr <srcprog> contains ”sendmail”
        attr <status> = “sent”
} 5 minutes {
        attr <msg> = "yes"
}
```

- Sums an attribute named “MSGSIZE” into a new attribute called “TOTALBYTES” and outputs the result every 5 minutes
  - Additionally outputs possibly interesting statistics that it gathers about the event

Syntax of a Counter Rule

counter ***name*** (sum|count) ***ATTRIBUTE*** as ***NEWATTRIBUTE***
{ *match rules* **}**
time-interval
{ output productions **}**

monitors data stream for appearance of matching record

outputs summary (if “sum” requested) or count (if “count” requested at specified time-interval

output productions include the attribute named as ***newattribute*** which contains the sum/count of the specified attribute

Sample Counter Output

```
<REC>
<COUNTERRULE>sendmailsumm</COUNTERRULE>
<INTERVAL>300</INTERVAL>
<LASTSEEN>Sun Jan 10 22:05:56 1999</LASTSEEN>
<TOTALBYTES>98372</TOTALBYTES>
<SAMPLES>45</SAMPLES>
<AVERAGE>2186</AVERAGE>
<ID>916023956.46758@whumpus.ranum.com</ID>
</REC>
```

- If enough samples are collected it may also output other values:
  - Trend, deviation from mean, etc.

More Counter Output

```
<REC>
<COUNTERRULE>TESTCOUNT</COUNTERRULE>
<INTERVAL>300</INTERVAL>
<LASTSEEN>Mon Jan 11 08:39:48 1999</LASTSEEN>
<TOTALRANDOM>74933</TOTALRANDOM>
<SAMPLES>144</SAMPLES>
<AVERAGE>520.37</AVERAGE>
<HISTORICAVERAGE>23938.62</HISTORICAVERAGE>
<INCREASING>6</INCREASING>
<VARIANCE>244.16</VARIANCE>
<AVERAGEUPDINTERVAL>2.08</AVERAGEUPDINTERVAL>
<ID>916061988.23526@whumpus.ranum.com</ID>
</REC>
```

Historical average of past running samples

Generally our samples are increasing rather than decreasing or fluctuating!

This is the average “distance” each value is from the average value

On average we got a new hit in the counter every 2.08 seconds

Why do a Counter Rule?

- Counters are generally useful for providing basic statistics
  - Messages thrown away
  - Messages of certain types
  - Web hits (or -404 messages)
  - Amounts of data transferred
- Counts can also be applied to the output of coalescing rules!

Performance of Counter Rules

- Counters are not very expensive to maintain in terms of memory or disk space
  - Each uses about 1kb of memory to hold all its past statistical values
  - So have fun with them!

Triggers on Counters

- What they do
  - Monitor the status of a counter over time
  - Output a new record if the counter goes above a ceiling, below a floor, or varies too far from its running average
  - Floor values are checked when a counter rolls
  - Ceiling values and deviance are checked when a counter is updated

Sample Trigger Rule

```
trigger on MAILTOTAL ceiling 100000 {
        attr <prio> = “9”
        attr <msg2> = ”Too much mail passing through system!!"
}
```

- This trigger will fire when the MAILTOTAL counter goes over 100000 in one of its roll cycles
  - Useful for watching for extreme values or placing a “boundary” for a value

Syntax of a Trigger Rule

trigger on *countername* trigger-spec
{
  output productions
}

trigger specs may be in the form of:

```
floor numericvalue      - triggers if less than
ceiling numericvalue    - triggers if more than
% numericvalue          - triggers if variance
                          exceeds
```

output productions create a new record which also has added information from the trigger that fired

Sample Trigger Output

```
<REC>
<MSG2>Too much mail passing through system!</MSG2>
<TRIGGER-TARGET>MAILCOUNT</TRIGGER-TARGET>
<TRIGGERED>Mon Jan 11 12:36:37 1999</TRIGGERED>
<REASON>value was above ceiling</REASON>
<TOTALRANDOM>35565</TOTALRANDOM>
<SAMPLES>67</SAMPLES>
<ID>916076197.39149@whumpus.ranum.com</ID>
</REC>
```

- This record shows the merge of data from the counter and the trigger's firing

Sample Coalesce Rule

```
coalesce {
        attr <msg> contains ”port scan"
}  {
        attr <msg> contains ”nmap scan"
} into {
        attr <msg> = ”yet another scan"
}
```

- Coalesces events that match on the specified attributes into another event containing selected attributes from those events

Syntax of a Coalesce Rule

coalesce

{ *match rules* **}** **[preserve (attr1, attrN…)]**

{ *match rules* **}** **[more match rules]**

into

{ output productions **}**

automatically clusters records that match on the specified attributes and outputs a new record containing output productions

may preserve specified attributes from the coalesced records

coalesce match rules allow extended matching in the form of <attr> = <attr> (e.g.: <ipsrc> = <ipdst>)

Sample Coalesce Output

```
<REC>
<REFERENCES>916063565.42812@whumpus.ranum.com,916063565.22296@whu
   mpus.ranum.com,916063565.10472@whumpus.ranum.com</REFERENCES>
<MOREJUNK>yes</MOREJUNK>
<YETMOREJUNK>check</YETMOREJUNK>
<DWELLTIME>99</DWELLTIME>
<ID>916063565.43258@whumpus.ranum.com</ID>
</REC>
```

- REFERENCES attribute was created to point to the coalesced entries
  - New ID attribute was assigned automatically in default processing

Why do a Coalesce Rule?

- Convenient way to roll multiple messages into a single message
  - Preserved references let you go back and examine the details if you need them
  - Could be used to summarize summaries or summarize events and create a counted value from them (e.g: turn many instances of port scans into a single event then count times that event happens)

Chaining Types of Events

- Coalescing to Discard events related to an incident
  - Coalesce low-level events into a single high-level event
  - Then set an output rule that *doesn't* output the low level events but *does* output the high-level event

Ok!

- Now we're done processing events!

  ...*What* do we do with the results?!

Outputs

- Outputs can either save records in XML format or perform text conversion based on matching rules
- Generic notion of “output channels” that receive records
  - Then apply a match; if it matches send it out specified channels
  - Channels can be Email, processes, or files

Establishing Out-Channels

```
outchannel mainlog to file "/tmp/bump" size 200 megs
output every to mainlog
```

- Establishes an output channel called “mainlog” which goes to a file named /tmp/bump
  - The size of the files is limited to 200megs total
  - The file output manager will start to delete old files when the total sizes reaches close to 200megs

Out-Channel File Rolling

```
whumpus$ ls -l /tmp/bump*
-rw-r--r--  1 mjr  wheel   248359 Jan 11 09:00 /tmp/bump.99.00.11:08
-rw-r--r--  1 mjr  wheel   621952 Jan 11 10:00 /tmp/bump.99.00.11:09
-rw-r--r--  1 mjr  wheel  1086725 Jan 11 11:00 /tmp/bump.99.00.11:10
-rw-r--r--  1 mjr  wheel   352256 Jan 11 11:19 /tmp/bump.99.00.11:11
whumpus$
```

- File names are constructed with the date and time appended to them
  - yy.mm.11:hh
  - If files grow fast they will have extra -0, -1, etc.
- No process needed to perform file rolling

Establishing Out-Channels

```
outchannel mainlog to file "/tmp/bump" size 200 megs
output every to mainlog
```

- The “output every” rule is a separate rule that controls output
  - Automatically match all records
  - Sends them out that channel

Establishing Out-Channels

```
outchannel mainlog to file "/tmp/bump" size 200 megs
output {
   <prio> morethan “5”
} to mainlog

outchannel maillog to file "/tmp/smtplog" size 200 megs
output {
   <proc> = “sendmail”
} to maillog
```

- The matching rule can be applied to control what records go to which output channel
  - Can be used to sort or categorize data easily

Emailing Outputs

```
outchannel text alertlog to process "/usr/bin/mail -s alert mjr"
```

- This establishes an output channel attached to a process
  - Output is fed to selected process on standard input
  - Standard input is closed for delivery on intervals
    - Default is every 2 minutes
    - Can be configured when channel is created

File Output Channel Syntax

outchannel [**text**] ***name*** **to file** "filename" [**number megs**]

text option specifies whether to apply output conversion rules or just output the record as XML

the filename is automatically constructed and rolled based on date/time and chunk size

[**size spec**] allows a specification of the maximum total amount of storage that should be allocated to this file. It should be sufficient to hold at least 2 hours worth of data. E.g:

2000 megs

100 megs

Mail Output Channel Syntax

outchannel [**text**] ***name*** **mail to** {
  ”recipient1”, “recipient2…”
  }[**every** ***timespec***]

text option specifies whether to apply output conversion rules or just output the record as XML

the recipient list is specified as a set of quoted strings

[**every** ***timespec***] allows a specification frequency with which Email should be sent. E.g:

30 seconds

2 minutes

Program Output Channel Syntax

outchannel [**text**] ***name*** **to process** "commandline" [**every** ***time***]

text option specifies whether to apply output conversion rules or just output the record as XML

the command-line is invoked with popen( ) be careful!

[every *time*] allows a time specification for how often the output channel should be flushed, e.g.:
every 40 minutes
every 2 hours

Rewriting Outputs

```
output  {
        attr <prio> = ”9"
        } to alertlog as
        {
           “this is a high priority alert!”
           "message-id: ${ID}"
           "\ttarget was ${IPDST}"
           “\tsource was ${IPSRC}"
           "\t${MSG}"
}
```

- Converts an XML-ized record into a human-readable output

Syntax of Output Rules

output
{ *match rules* **} to *channelname* as**
{
“string 1…”
“string 2…” …
}

outputs matching records to the specified channel using the listed string formats

strings are passed through substitution where all ${value} are replaced with the matching attribute from the record that is being output

Output from Output Rules

```
To: mjr@nfr.com
From: HotZone@hotzone
Subject: alert

   Priority 9 event
   Event-ID: 916023956.46758@whumpus.ranum.com
   Code red scan
   Source=10.10.1.100
   Dest=10.10.1.20
```

- Multiple messages will get packed into a single Email within time-limit

Summary

- Now you know everything you need to know about writing Fargo rules
  - It’s complex on the surface but really is fairly straightforward
  - Fargo’s just a big string processor that normalizes input under your control and produces output under your control
  - The only limit is your creativity