Real-time log file analysis using the Simple Event Correlator (SEC)

John P. Rouillard
University of Massachusetts at Boston
For LISA 2004 Conference: Atlanta, GA

November 2004

Abstract

Originally published at Usenix LISA 2004 conference. November 2004 Atlanta, Georiga, USA.

Log analysis is an important way to keep track of computers and networks. The use of automated analysis always results in false reports, however these can be minimized by proper specification of recognition criteria. Current analysis approaches fail to provide sufficient support for the recognizing the temporal component of log analysis. Temporal recognition of event sequences fall into distinct patterns that can be used to reduce false alerts and improve the efficiency of response to problems. This paper discusses these patterns while describing the rationale behind and implementation of a ruleset created at the CS department of the University of Massachusetts at Boston for SEC - the Simple Event Correlation program.

1 Introduction

With today's restricted IT budgets, we are all trying to do more with less. One of the more time consuming, and therefore neglected, tasks is the monitoring of log files for problems. Failure to identify and resolve these problems quickly leads to downtime and loss of productivity. Log files can be verbose with errors hidden among the various status events indicating normal operation. For a human, finding errors among the routine events can be difficult, time consuming, boring, and very prone to error. This is exacerbated when aggregating events, using a mechanism such as syslog, due to the intermingling of events from different hosts that can submerge patterns in the event streams.

Many monitoring solutions rely on summarizing the log files for the previous days logs. This is very useful for accounting and statistics gathering. Sadly, if the goal is problem determination and resolution then reviewing these events the day after they are generated is less helpful. Systems administrators cannot proactively resolve or quickly respond to problems unless they are aware that there is a problem. It is not useful to find out in the next morning's summary that a primary NFS server was reporting problems five minutes before it went down. The sysadmin staff needs to discover these problems while there is still time to fix the problem and avert a catastrophic loss of service.

The operation of computers and computer networks evolves over time and requires a solution to log file analysis that address this temporal nature. This paper describes some of the current issues in log analysis and presents the rationale behind an analysis rule set developed at the Computer Science Department at the University of Massachusetts at Boston. This ruleset is implemented for the Simple Event Correlator (SEC), which is a Perl based tool designed to perform analysis of plain text logs.

1.1 Current Approaches

There are many programs that try to isolate error events by automatically condensing or eliminating routine log entries. In this paper, I do not consider interactive analysis tools like MieLog[Takada02]. I separate automatic analysis tools into offline or batch monitoring and on-line or real-time monitoring.

1.1.1 Offline Monitoring

Offline solutions include: **logwatch**[logwatch], **SLAPS-2**[SLAPS-2], or **Addamark LMS**[Sah02]. Batch solutions have to be invoked on a regular basis to analyze logs. They can be run once a day, or many times an hour. Offline tools are useful for isolating events for further analysis by real time reporting tools. In addition they provide statistics that allow the system administrator to identify the highest event sources for remedial action. However, offline tools do not provide the ability to provide automatic reactions to problems. Adam Sah in discussing the **Addamark LMS**[Sah02, p. 130] claims that real-time analysis is not required because a human being, with slow reaction times, is involved in solving the problem. I disagree with this claim. While it is true that initially a human is required to identify, isolate and solve the problem, once it has been identified, it is a candidate for being automatically addressed or solved. If a human would simply restart apache when a particular sequence of events occur, why not have the computer automatically restart apache instead? Automatic problem responses coupled with administrative practices can provide a longer window before the impact of the problem is felt. An "out of disk space" condition can be addressed by removing buffer files placed in the file system for this purpose. This buys the system administrator a longer response window in which to locate the cause of the disk full condition minimizing the impact to the computing environment.

Most offline tools do not provide explicit support for analyzing the log entries with respect to the time they were received. While they could be extended to try to parse timestamps from the log messages, this is difficult in general, especially with multiple log files and multiple machines, as ordering the events requires normalizing the time for all log messages to the same timezone. Performing analysis on log files that do not have timestamps eliminates the ability of these batch tools to perform analysis by time. Solutions such as the **Addamark LMS**[Sah02] parse and record the generation time, but the lack of real-time event-driven, as opposed to polled, triggers reduces its utility.

1.1.2 Online Monitoring

Online solutions include: **logsurfer**[logsurfer], **logsurfer+**[logsurfer+], **swatch**[swatch, Hansen1993], **2swatch**[2swatch], **SHARP**[Bing00], **ruleCore**[ruleCore], **LoGS**[LoGS] and SEC[SEC]. All of these programs run continuously watching one or more log files, or receiving input from some other program.

Swatch is one of the better known tools. **Swatch** provides support for ignoring duplicate events and for changing rules based on the time of arrival. However, **swatch's** configuration language does not provide the ability to relate arbitrary events in time. Also, it lacks the ability to activate/deactivate rules based on the existence of other events other than suppressing duplicate events using its throttle action.

Logsurfer dynamically changes its rules based on events or time. This provides much of the flexibility needed to relate events. However, I found its syntax difficult to use (similar to the earliest 1.x version of SEC) and I never could get complex correlations across multiple applications to work properly. The dynamic nature of the rules made debugging difficult. I was never able to come up with a clean, understandable, and reliable method of performing counting operations without resorting to external programs. Using SEC, I have been able to perform all of the operations I implemented in **logsurfer** with much less confusion.

LoGS is an analysis program written in Lisp that is still maturing. While other programs create their own configuration language, **LoGS's** rules are also written in Lisp. This provides more flexibility in designing rules than SEC, but may require too much programming experience on the part of the rule designers. I believe this reduces the likelihood of its widespread deployment. However, it is an exciting addition to the tools for log analysis research.

The Simple Event Correlator (SEC) by Risto Vaarandi uses static rules unlike **logsurfer**, but provides higher level correlation operations such as explicit pair matching and counting operations. These correlation operations respond to a triggering event and persist for some amount of time until they timeout, or the conditions of the correlation are met. SEC also provides a mechanism for aggregating events and modifying rule application based on the responses to prior events. Although it does not have the dynamic rule creation of **logsurfer**, I have been able to easily generate rules in SEC that provide the same functionality as my **logsurfer** rules.

1.1.3 Filter in vs. filter out

Should rules be defined to report just recognized errors, or should routine traffic be eliminated from the logs and the residue reported? There appear to be people who advocate using one strategy over the other.

I claim that both approaches need to be used and in more or less equal parts. I am aware of systems that are monitored for only known problems. This seems risky as it is more likely an unknown problem will sneak up and bite the unwary systems administrator. However, very specific error recognition is needed when using automatic responses to ensure that the best solution is chosen. Why restart apache if killing a stuck cgi program will solve the problem?

Filtering out normal event traffic and reporting the residue allows the system administrator to find signatures of new unexpected problems with the system. Defining "normal traffic" in such a way that we can be sure its routine is tricky especially if the filtering program does not have support for the temporal component of event analysis.

2 Event Modeling

Modeling normal or abnormal events requires the ability to fully specify every aspect of the event. This includes recognizing the content of the event as well as its relationship to other events in time. With this ability, we can recognize a composite or correlated event that is synthesized from one or more primitive events. Normal activity is usually defined by these composite events. For example a normal activity may be expressed as:

> 'sendmail -q' is run once an hour by root at 31 minutes after the hour. It must take less than 1 minute to complete.

```
>  CMD: /usr/lib/sendmail -q
>  root 25453 c Sun May 23 03:31:00 2004
<  root 25453 c Sun May 23 03:31:00 2004
```

Figure 1: Events indicating normal activity for a scheduled cron job.

This activity is shown by the cron log entries in Figure 1 and requires the following analysis operations:

- Find the sendmail CMD line verifying its arrival time is around 31 minutes after the hour. If the line does not come in, send a warning.
- The next line always indicates the user, process id and start time. Make sure that this line indicates that the command was run by root.
- The time between the CMD line arrival and the final line must be less than 1 minute. Because other events may occur between the start and end entries for the job, we recognize the last line by its use of the unique number from the second field of the second line.

This simple example shows how a tool can analyze the event log in time. Tools that do not not allow the specification of events in the temporal realm as well as in the textual/content space can suffer from the following problems:

- matching the right event at the wrong time. This could be caused by an inadvertent edit of the cron file, or a clock skew on the source or analyzing host.
- not noticing that the event took too long to run.
- not noticing that the event failed to complete at all.

2.1 Temporal relationships

The cron example mentions one type of temporal restriction that I call a schedule restriction. Schedule restrictions are defined by working on a defined (although potentially complex) schedule. Typical schedule restrictions include: every hour at 31 minutes past the hour, Tuesday morning at 10AM, every weekday morning between 1 and 3AM.

In addition to schedule restrictions, event recognition requires accounting for inter-event timing. The events may be from a single source such as the sequence of events generated by a system reboot. The statement that the first to last event in a boot sequence should complete in five minutes is an inter-event timing restriction. Also, events may arise from multiple sources. Multi-source inter-event timing restrictions might include multiple routers sending an SNMP authentication trap in five minutes, or excessive "connection denied" events spread across multiple hosts and multiple ports indicating a port scan of the network.

These temporal relationships can be explicit within a correlation rule: specifying a time window for counting the number of events, suppressing events for a specified time after an initial event. The timing relationship can also be implicit when one event triggers the search for subsequent events.

2.2 Event threading

Analysis of a single event often fails to provide a complete picture of the incident. In the example above, reporting only the final cron event is not as useful as reporting all three events when trying to diagnose a cause. Lack of proper grouping can lead to underestimating the severity of the events. Consider the following scenario:

1. A user logs in using ssh from a location that s/he has never logged in from before.

2. The ssh login was done using public key authentication.

3. The ssh session tries to open port 512 on the server. It is denied.

4. Somebody tries to run a program called "crackme" that tries to execute code on the stack.

5. The user logs out.

Looking at this sequence implies that somebody broke in and tried to execute an unsuccessful attempt to gain root privileges. However, in looking at individual events, it is easy to miss the connections. Taken in isolation, each event could be easily dismissed, or even filtered out of the reports. Reporting them as discrete events, as many analysis tools do, may even contribute to an increased chance of missing the pattern. Taken together they indicate a problem that needs to be investigated. A log analysis tool needs to provide some way to link these disparate messages from different programs into a single thread that paints a picture of the complete incident.

2.3 Missing Events

Log analysis programs must be able to detect missing log events[Finke2002]. These missing events are critical errors since they indicate a departure from normal operation that can result in a many problems. For example, cron reports the daily log rotation at 12:01AM. If this job is not done (say, because cron crashed), it is better to notice the failure immediately rather than three months later when the partition with the log files fills up.

The problem with detecting missing events is that log monitoring is – by its nature – an event-driven operation: if there is no event, there is no operation. The log analysis tool should provide some mechanism for detecting a missing event. One of the simpler ways to handle this problem is to generate an event or action on a regular basis to look for a missing event. An event-driven mechanism can be created using external tools such as cron to synthesize events, but I fail to see a mechanism that the log analysis tool can use to detect the failure of the external tool to generate these events.

2.4 Handling False Positives/False Negatives

A false negative occurs when an event that indicates a problem is not reported. A false positive results when a benign event is reported as a problem. False negatives impact the computing environment by failing to detect a problem. False positives must be investigated and impact the person(s) maintaining the computing environment. A false positive also has another danger. It can lead to the “boy who cried wolf” syndrome, causing a true positive to be ignored as a false positive.

Two scenarios for generating false negatives are mentioned above. Both are caused by incorrectly specifying the conditions under which the events are considered routine.

False positives are another problem caused by insufficiently specifying the conditions under which the event is a problem. In either case, it may not be possible to fully specify the problem conditions because:

- Not all of the conditions are known.
- Some conditions are not able to be monitored and cannot be added to the model.

It may be possible to find correlative conditions that occur to provide a higher degree of discrimination in the model. These correlative events can be use to change the application of the rules that cause the false positive to inhibit the false report.

To reduce these false positives and false negatives, the analysis program needs to have some way of generating and receiving these correlative events.

While it is impossible to eliminate all false reports, by proper specification of event parameters, false reports can be greatly reduced.

2.5 Single vs. Multiple Line Events

Programs can spread their error reports across multiple lines in a logfile. Recognizing a problem in these circumstances requires the ability to scan not just a single line, but a series of lines as a single instance. The series of lines can be treated as individual events, but key pieces of information needed to trigger a response or recognize an event sequence may occur on multiple lines. Consider the cron example of Figure 1: the first two lines provide the information needed to determine that it is an entry for sendmail started by root, and the process id is used in discovering the matching end event. Handling this multi-line event as multiple single line events complicates the rules for recognizing the events.

Multi-line error messages seem to be more prevalent in application and device logs that do not use the Unix standard syslog reporting method, but some syslog versions split long syslog messages into multiple parts when they store them in the logfile. Fortunately, when I have seen this happen, the log lines always occur adjacent to one other without any intervening events from other sources. This allows recognition provided that the split does not occur in the middle of a field of interest.

With syslog and other log aggregation tools, a single multi-line message can be distorted by the injection of messages from other sources. The logs from applications that produce multi-line messages should be directed to their own log file so that they are not distorted. Then a separate SEC process can analyze the log file and create single line events that are passed to a parent SEC for global correlation. This is similar to the method used by **Addamark**[Sah02].

Although keeping the log streams separate simplifies some log analysis tasks, it prevents the recognition of conditions that affect multiple event streams. Although SEC provides a mechanism for identifying the source of an event, performing event recognition across streams requires that the event streams be merged.

3 Sec correlation idioms and strategies

This section describes particular event scenarios that I have seen in my analysis of logs. It demonstrates idioms for SEC that model and recognize these scenarios.

3.1 SEC primer

A basic knowledge of SEC's configuration language is required to understand the rules presented below. There are nine basic rule types. I break them into two groups: basic and complex rules. Basic rules types perform actions and do not start an active correlation operation that persists in time. These basic types are described in the SEC man page as:

Suppress: suppress matching input event (used to keep the event from being matched by later rules).

Single: match input event and immediately execute an action that is specified by rule.

Calendar: execute an action at specific times using a cron like syntax.

Complex rules start a multi-part operation that exists for some time after the initial event. The simplest example is a `SingleWithSuppress` rule. It triggers on an event and remains active for some time to suppress further occurrences of the triggering event. A `Pair` rule recognizes a triggering event and initiates a search for a second (paired) event. It reduces two separate but linked events to a single event pair. The complex types are described in the SEC man page as:

SingleWithScript: match input event and depending on the exit value of an external script, execute an action.

SingleWithSuppress: match input event and execute an action immediately, but ignore following matching events for the next T seconds.

Pair: match input event, execute the first action immediately, and ignore following matching events until some other input event arrives (within an optional time window T). On arrival of the second event execute the second action.

PairWithWindow: match input event and wait for T seconds for another input event to arrive. If that event is not observed within a given time window, execute the first action. If the event arrives on time, execute the second action.

SingleWithThreshold: count matching input events during T seconds and if given threshold is exceeded, execute an action and ignore all matching events during rest of the time window.

SingleWith2Thresholds: count matching input events during T1 seconds and if a given threshold is exceeded, execute an action. Now start to count matching events again and if their number per T2 seconds drops below second threshold, execute another action.

SEC rules start with a `type` keyword and continue to the next `type` keyword. In the example rules below, `...` are used to take the place of keywords that are not needed for the example, they do not span rules. The order of the keywords is unimportant in a rule definition.

SEC uses Perl regular expressions to parse and recognize events. Data is extracted from events by using subexpressions in the Perl regular expression. The extracted data is assigned to numeric variables $1, $2,...,$N where N is the number of subexpressions in the Perl regular expression. The numeric variable $0 is the entire event. For example, applying the regular expression "`([A-z]*): test number ([0-9]*)`" to the event "`HostOne: test number 34`" will assign $1 the value "`HostOne`", $2 the value "`34`", and $0 will be assigned the entire event line.

Because complex rule types create ongoing correlation operations, a single rule can spawn many active correlation operations. Using the regular expression above, we could have one correlation that counted the number of events for `Host` and another separate correlation that counted events for `HostTwo`. Both counting correlations would be formed from the same rule, but by extracting data from the event the two correlations become separate entities.

This data allows the creation of unique contexts, correlation descriptions and coupled patterns linked to the originating event. We will explore these items in more detail later. Remember that when applying a rule, the regular expression or pattern is always applied first regardless of the ordering of the keywords. As a result, references to $1, $2,...,$N anywhere else in the rule refer to the data extracted by the regular expression.

SEC provides a flow control and data storage mechanism called contexts. As a flow control mechanism, contexts allow rules to influence the application of other rules. Contexts have the following features:

- Contexts are dynamically created and often named using data extracted from an event to make names unique.
- Contexts have a defined lifetime that may be infinite. This lifetime can be increased or decreased as a result of rules or timeouts.
- Multiple contexts can exist at any one time.
- A context can execute actions when its lifetime expires.
- Contexts can be deleted without executing any end-of-lifetime actions.
- Rules (and the correlations they spawn) can use boolean expressions involving contexts to determine if they should apply. Existing contexts return a true value; non-existent contexts return a false value. If the boolean expression is true, the rule will execute, if false the rule will not execute (be suppressed).

In addition to a flow control mechanism, contexts also serve as storage areas for data. This data can be events, parts of events or arbitrary strings. All contexts have an associated data store. In this paper, the word "context" is used for both the flow control entity and its associated data store. When a context is deleted, its associated data store is also deleted. Contexts are most often used to gather related events, for example login and logout events for a user. These contexts can be reported to the system administrator if certain conditions are detected (e.g., the user tried to perform a su during the login session).

The above description might seem to imply that a single context has a single data store; this is not always the case. Multiple contexts can share the same data store using the alias mechanism. This allows events from different streams to be gathered together for reporting or further analysis. The ability to extract data from an event and linking the context by name to that event provides a mechanism for combining multiple event streams into a single context that can be reported. For example, if I extract the process ID 345 from syslog events, I can create a context called: `process_345` and add all of the syslog events with the same PID to that event. If I now link the context `process_346` to the `process_345` context, I can add all of the syslog events with the pid 346 to the same context (data store). So now the `process_345`/`process_346` context contains all of the syslog events from both processes.

In the paper, I use the term session. A session is simply a record of events of interest. In general these events will be stored in one or more contexts. If ssh errors are of interest, a session will record all the ssh events into a context (technically a context data store that may be known by multiple names/aliases) and report that context. If tracing the identities that a user assumes during a login is needed, a different series of data is recorded in a context (data store): the initial ssh connection information is recorded, the login event, the su event as the user tries to go from one user ID to another.

The rest of the elements of SEC rules will be presented as needed by the examples.

3.2 Responding to or filtering single events

The majority of items that we deal with in processing a log file are single items that we have to either discard or act upon. Discardable events are the typical noise where the problem is either not fixable, for example a failing reverse DNS lookups on remote domains from tcp wrappers, or are valueless information that we wish to discard.

Discardable events can be handled using the suppress rule. Figure 2 is an example of such a rule. Since all of my rule sets report anything that is not handled, we want to explicitly ignore all noise lines to prevent them from making it to the default "report everything" rule.

This is a good time to look at the basic anatomy of a SEC rule. All rules start with a `type` option as

```
type=suppress
desc=ignore non-specific paper problem report \
  since prior events have given us all we need.
ptype=regexp
pattern=.  printer: paper problem$
```

Figure 2: A suppress rule that is used to ignore a noise event sent during a printer error.

described in section 3.1. All rules have a `desc` option that documents the rule's purpose. For the complex correlation rules, the description is used differentiate between correlation operations derived from a single rule. We will see an example of this when we look at the horizontal port scan detection rules.

Most rules have a `pattern` option that is applied to the event depending on the `ptype` option. The pattern can be a regular expression, a substring, or a truth value (TRUE or FALSE). The `ptype` option specifies how the `pattern` option is to be interpreted: a regular expression (`regexp`), a substring (`substr`), or a truth value (`TValue`). It also determines if the pattern is successfully applied if it matches the event match (`regexp`/`substr`), or does not match (`nregexp`/`nsubstr`) the event. For `TValue` type patterns, TRUE matches any event (successfull application), while FALSE (not successfully applied) does not match any input event. If the pattern does not successfully apply, the rule is skipped and the next rule in the configuration file is applied.

A number can be added to the end of any of the `nregexp`, `regexp`, `substr`, or `nsubstr` values to make the pattern match across that many lines. So a `ptype` value of `regexp2` would apply the pattern across two lines of input.

By default when an event triggers a rule, the event is *not compared* against other rules in the same file. This can be changed on a per rule basis by using the `continue` option[1].

After single event suppression, the next basic rule type is the single rule. This is used to take action when a particular event is received. Actionable events can interact with other higher level correlation events: adding the event to a storage area (context), changing existing contexts to activate or deactivate other rules, activating a command to deal with the event, or just reporting the event. Figure 3 is an example of a single rule that will generate a warning if the printer is offline from an unknown cause.

In Figure 3 we see two more rule options: `context` and `action`. The `context` option is a boolean expression of contexts that further constrains the rule.

When processing the event "`lj2.cs.umb.edu: printer: Report Printer Offline if needed`", the single rule in Figure 3 checks to see if the pattern applies successfully. In this case the pattern matches the event, but if the `Report_Printer_lj2.cs.umb.edu_Offline` context does not exist, then the actions will not be executed. The context `Report_Printer_lj2.cs.umb.edu_Offline` is deleted by other rules in the ruleset (not shown) if a more exact diagnosis of the cause is detected. This suppresses the default (and incorrect) report of the problem.

The `action` option specifies the actions to take when the rule fires. In this case it writes the message `printer lj2.cs.umb.edu offline, unknown cause` to standard output (specified by the file name "`-`") . Then it deletes the context `Report_Printer_lj2.cs.umb.edu_Offline` since it is no longer needed.

There are many potential actions, including:

- creating, deleting, and performing other operations on contexts
- invoking external programs
- piping data or current contexts to external programs
- resetting active correlations
- evaluating Perl mini-programs
- setting and using variables
- creating new events
- running child processes and using the output from the child as a new event stream.

[1]Note: `continue` is not supported for the `suppress` rule type.

```
type=single
continue=dontcont
desc = Report Printer Offline if needed
ptype=regexp
pattern=^([\w._-]+): printer: Report Printer Offline if needed
context = Report_Printer_$1_Offline
action = write - "printer $1 offline, unknown cause" ; \
         delete Report_Printer_$1_Offline
```

Figure 3: A single command that writes a warning message and deletes a context that determines if it should execute.

We will discuss and use many of these actions later in this paper.

3.3 Scheduling events with with finer granularity

Part of modeling normal system activity includes accounting for scheduled activities that create events. For example, a scheduled weekly reboot is not worth reporting if the reboot occurs during the scheduled window, however it is worth reporting if it occurs at any other time.

For this we use the `calendar` rule. It allows the reader to schedule and execute actions on a cron like schedule. In place of the `ptype` and `pattern` options it has a `time` option that has five cron-like fields. It is wonderful for executing actions or starting intervals on a minute boundary. Sometimes we need to start intervals with resolution of a second rather than a minute.

Figure 4 shows a mechanism for generating a window that starts 15 seconds after the minute and lasts for 30 seconds. The key is to create two contexts and use both of them in the rules that should be active (or inactive) only during the given window. One context `wait_for_window` expires to begin the timed interval. The `window` context expires marking the end of the interval.

Creating an event on a non-minute boundary is trivial once the reader learns that the event command has a built in delay mechanism.

Triggering events generated by calendar rules or by expiring contexts can execute actions, define intervals, trigger rules or pass messages between rules. Triggering events are used extensively to detect missing events.

```
type=calendar
time=30 3 * * *
desc=create 30 second window
action=create window 45; \
       create wait_for_window 15

type=single
...
context=window && !wait_for_window
```

Figure 4: A mechanism for creating an timed interval that starts on a non-minute boundary.

3.4 Detecting missing events

The ability to generate arbitrary events and windows with arbitrary start and stop times is useful when detecting missing events. The rules in Figure 5 report a problem if a '`sendmail -q`' command is not run by root near 31 minutes after the hour. Because of natural variance in the schedule, I expect and accept a sendmail start event from 5 seconds before to 10 seconds after the 31st minute.

The event stream from Figure 1 is used as input to the rules. Figure 6 displays the changes that occur while processing the events in time. Contexts are represented by rectangles, the length of the rectangle is the context's lifetime. Upside down triangles represent the arrival of events. Regular triangles

```
# rule 1: detect the sendmail event
type = single
desc = sendmail has run, don't report it as failed
ptype = regexp2
pattern = ^\>  CMD: /usr/lib/sendmail -q.*\n\>  root ([0-9]+) c .*
context = sendmail_31_minute && ! sendmail_31_minute_inhibit
action = delete sendmail_31_minute

# rule 2: define the time window and prep to report a missing event
type = calendar
desc = Start searching for sendmail invocation at 31 past hour
time=30 * * * *
action = create sendmail_31_minute 70 write - \
              Sendmail failed to run detected at %t; \
         create sendmail_31_minute_inhibit 55
```

Figure 5: Rules to detect a missed execution of a sendmail process at the appointed time.

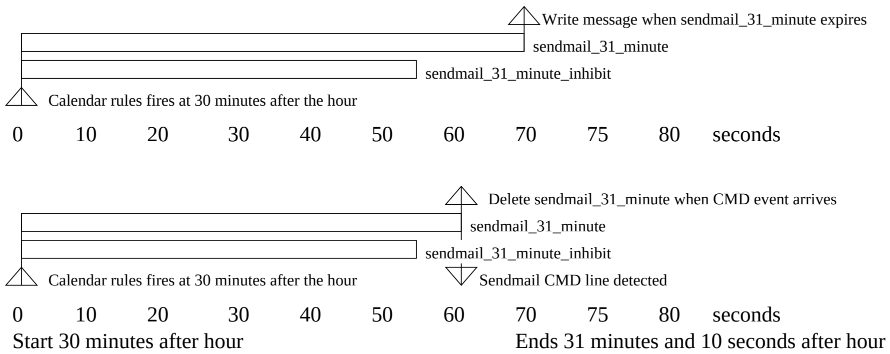

Figure 6: Two timelines showing the events and contexts involved in detecting a missing, or present, sendmail invocation from cron.

represent actions within SEC. The top graph shows the sequence when the sendmail event fails to arrive, while the bottom graph shows the sequence when the sendmail program is run.

The correlation starts when rule 2 (the `calendar` rule) creating the context `sendmail_31_minute` that will execute an action (write a message to standard output) when it times out after 70 seconds (near 31 minutes and 10 seconds) ending the interval. The calendar rule creates a second context, sendmail_31_minute_inhibit, that will timeout in 55 seconds (near 30 minutes and 55 seconds) starting the 15 second interval for the arrival of the sendmail event. Looking at the top graph in Figure 6, we see the creation of the two contexts on the second and third lines. No event arrives within the 15 second window, so the `sendmail_31_minute` expires and executes the "write" action. The bottom graph shows what happens if the sendmail event is detected. Rule 1 is triggered by the sendmail event occurring in the 15 seconds window and deletes the `sendmail_31_minute` context. The deletion also prevents the "write" action associated with the context from being executed.

Note that the boolean context of rule 1 prevents its execution if the sendmail event were to occur less than 5 seconds before the 31st minute since `! sendmail_31_minute_inhibit` is false because `sendmail_31_minute_inhibit` exists and is therefore true. If the sendmail event occurs after 31 minutes and 10 seconds, the context is again false since `sendmail_31_minute` does not exist, and is false.

The example rules use the write action to report a problem. In a real ruleset, the systems administrator could use the SEC `shellcmd` action to invoke logger(1) to generate a syslog event to be forwarded to a central syslog server. This event would be found by SEC running on the syslog master. The rule matching the event could notify the administrator via email, pager, wall(1) or send a trap to an NMS like HPOV or Nagios. Besides reporting, the event could be further processed with a threshold rule that would try to restart cron as soon as two or more "missed sendmail events" events are reported, and report a problem only if a third consecutive "missed sendmail event" arrived.

3.5 Repeat Elimination/Compression

I have dealt with real-time log file reporters that generated 300 emails when a partition filled up overnight. There must be a method to condense or de-duplicate repeated events to provide a better picture of a problem, and reduce the number of messages spamming the administrators.

The `SingleWithSuppress` rule fills this deduplication need. To handle file system full errors, the rule in Figure 7 is used.

This rule reports that the filesystem is full when it receives its first event. It then suppresses the event message for the next hour. Note that the `desc` keyword includes the filesystem and hostname ($2 and $1 respectively). This makes the correlation operation that is generated from the rule unique so that a disk full condition on the same host for the filesystem /mount/fs2 will generate an error event if it occurs 5 minutes after the /mount/sd0f event. If the filesystem was not included in the `desc` option, then only one alert for a full filesystem would be generated regardless of how many filesystems actually filled up during the hour.

3.6 Report on analysis of event contents

Unlike most other programs, SEC allows the reader to extract and analyze data contained within an event. One simple example is the rule that analyzes NTP time adjustments. I consider any clock with less than 1/4 a second difference from the NTP controlled time sources to be within a normal range. Figure 8 shows the rules that are applied to analyze the xntpd time adjustment events. We extract the value of the time change from the step messages. This value is assigned to the variable $1. The context expression executes a Perl mini-program to see if the absolute value of the change is larger than the threshold of 0.25 seconds. If it is, the context is satisfied and the rule's actions fire.

The context expression uses a mechanism to run arbitrary Perl code. It then uses the result of the expression to determine if the rule should fire. It can be used to match networks after applying a netmask,

```
# Example:
#   Apr 13 15:08:52 host4.example.org ufs: [ID 845546 \
#       kern.notice] NOTICE: alloc: /mount/sd0f: file system full
type=SingleWithSuppress
desc=Full filesystem $2 on $1
ptype=regexp
pattern=([\w._-]+) ufs: \[.* NOTICE: alloc: ([\w/._-]+): file system full
action= write - filesystem $2 on host $1 full
window=3600
```

Figure 7: A rule to report a file system full error and suppress further errors for 60 minutes.

perform calculations with fields of the event or other tasks to properly analyze the events.

3.7 Detect identical events occurring across multiple hosts

A single incident can affect multiple hosts. Detecting a series of identical events on multiple hosts provides a measure of the scope of the problem. The problem can be an NFS server failure affecting only one host that does not need to be paged out in the middle of the night, or it may affect 100 hosts, which requires recovery procedures to occur immediately. Other problems such as time synchronization, or detection of port scans also fall into this realm.

One typical example of this rule is to detect horizontal port scans. The rules in Figure 9 identify a horizontal port scan as three or more connection denied events from different server hosts within 5 minutes from a particular external host or network. So 20 connections to different ports on the same host would not result in the detection of a horizontal scan. In the example, I assume that the hosts are equipped with TCP wrappers that report denied connections. The set of rules in Figure 9 implements the detection of a horizontal port scan by counting unique client host/server host combinations.

A timeline of these three rules is shown in Figure 10.

The key to understanding these rules is to realize that the description field is used to match events with correlation operations. When rule 1, the threshold correlation rule, sees the first rejected connection from 192.168.1.1 to 10.1.2.3, it generates a `Count denied events from 192.168.1.1` correlation. The next time a deny for 192.168.1.1 arrives, it will be tested by rule 1, the description field generated from this new event will match an ongoing correlation threshold operation and it will be considered part of the `Count denied events from 192.168.1.1` threshold correlation. If a rejection event for the source 193.1.1.1 arrives, the generated description field will not match an active threshold correlation, so a new correlation operation will be started with the description `Count denied events from source2`. Figure 10 shows a correlation operation from start to finish. First the event E1 reports a denial from host 192.168.1.1 to connect/scan 10.1.2.3. The correlation operation `Count denied events from 192.168.1.1` is started by rule 1, rule 2 is skipped because the pattern does not match, and rule 3 creates the 5-minute-long context `seen_connection_from_192.168.1.1_to_10.1.2.3` that is used to filter arriving event to make sure that only unique events are counted. The rest of rule 3's actions will be discussed later.

The count for rule 1, the threshold correlation operation, is incremented only if the `seen_connection_from_192.168.1.1_to_10.1.2.3` context does not exist. When the E1/2 (event 1 number 2) arrives, this context still exists and all the rules ignore the event. When E2/1 arrives, it triggers rule 1 and rule 3 creating the appropriate context and incrementing the threshold operation's

```
type=single
desc = report large xntpd corrections for host $1
continue = dontcont
ptype=regexp
context= =(abs($2) > 0.25)
pattern=([A-z0-9._-]+) xntpd\[[0-9]+\]:.*time reset \(step\) ([-]?[0-9.]+) s
action= write - "large xntpd correction($2) on $1"
```

Figure 8: Rule to analyze time corrections in NTP time adjustment events. The absolute value of the time adjustment must be greater than 0.25 seconds to generate a warning.

count.

When five minutes have passed since E1/1's arrival and the threshold rule has not been triggered by the arrival of three events, the start of the threshold rule is moved to the second event that it counted, and the count is decremented by 1. This occurs because the threshold rule uses a sliding window by default. When events 3/1 and 4/1 arrive, they are counted by the shifted threshold correlation operation started by rule 1. With the arrival of E2/1, E3/1, and E4/1, three events have occurred within five minutes and a horizontal port scan is detected. As a result, the action reporting the context `conn_deny_from_192.168.1.1` is executed and the events counted during the correlation operation (maintained by the add action of rule 3) are reported to the file report_log.

Rule 2 and the final actions of rule 3 allow detection of horizontal port scans even if they come from different hosts such as: 192.168.3.1, 192.168.1.1, and 192.168.7.1. If each of these hosts scans a different host on the 10 network, it will be detected as a horizontal scan from the 192.168.0.0 network. This is done by creating three events replacing the real source address with a corresponding network address. One event is created for each class A, B and C network that the original host could belong to: 192.168.1.0, 192.168.0.0, and 192.0.0.0. The response to these synthesized events are not shown in Figure 10, but they start a parallel series of correlation operations and contexts using the network address of the client in place of 192.168.1.1.

Vertical scans can use the same framework with the following changes:

- the filtering context needs to include port numbers so that only unique client host/server host/port triples are counted by the threshold rule.
- the description of rule 1 to include the server host IP so that it only counts connections to a specific server host.

This will count the number of unique server ports that are accessed on the server from the client host.

In general, using rules 1 and 3, you can count unique occurrences of a value or group of values. The context used to link the rules must include the unique values in its name. The description used in rule 1 will not include the unique values and will create a bucket in which the events will be counted. In the horizontal port scan case, case, my bucket was any connection from the same client host. The unique value was the server IP address connected to by the the client host. In detecting a vertical port scan, the value is the number of unique ports connected to while the bucket is the client/server host pair.

These two changes allow the counting ruleset to count the number of unique occurrences of the parameter that is present in the filtering rule, but missing from the rule 1 description (the bucket). E.g., if the context specifies serverhost, clienthost, serverport and rule 1 specifies clienthost and serverhost in its description, then the rules above implement counting of unique ports for a given clienthost and serverhost. The rules as presented above specified clienthost and serverhost, rule 1 specified the clienthost, so the ruleset counted unique serverhost's for a given clienthost.

Other counting methods can also be implemented

```
# Example input:
# May 10 13:52:13 cyber TCPD-Event cyber:127.6.7.1:3424:sshd deny \
#     badguy.example.com:192.268.15.45 user unknown
#  Variable = description (value from example above)
#  $3  = server ip address (127.6.7.1)
#  $5  = daemon or service connected to on server (sshd)
#  $8  = ip address of client (attacking) machine (192.268.15.45)
#  $9  = 1st quad of client host ip address (192)
#  $10  = 2nd quad of client host ip address (6)
#  $11 = 3rd quad of client host ip address (7)
#  $12 = 4th quad of client host ip address (1)
# Rule 1: Perform the counting of unique destinations by client host/net
type = SingleWithThreshold
desc = Count denied events from $8
continue = takenext
ptype = regexp
pattern = ^(.*) TCPD-Event ([A-z0-9_.]*):([0-9.]*):([0-9]*):([^ ]*) (deny) \
            ([^:]*):(([0-9]*)\.([0-9]*)\.([0-9]*)\.([0-9]*)) user (.*)
action = report conn_deny_from_$8 /bin/cat >> report_log
context = ! seen_connection_from_$8_to_$3
thresh = 3
window = 300

## Rule 2: Insert a rule to capture synthesized network tcpd events.
type=single
...
pattern = ^(.*) TCPD-Event ([A-z0-9_.]*):([0-9.]*):([0-9]*):([^ ]*) (deny) \
            ([^:]*):(([0-9]*)\.([0-9]*)\.([0-9]*)\.([0-9]*)) user (.*) net$
action=none

# Rule 3: Generate network counting rules and maintain contexts
type = single
desc = maintain counting contexts for deny service $5 from $8 event
continue = takenext
ptype = regexp
pattern = ^(.*) TCPD-Event ([A-z0-9_.]*):([0-9.]*):([0-9]*):([^ ]*) (deny) \
            ([^:]*):(([0-9]*)\.([0-9]*)\.([0-9]*)\.([0-9]*)) user (.*)
context = ! seen_connection_from_$8_to_$3
action = create seen_connection_from_$8_to_$3 300; \
         add conn_deny_from_$8 $0 ; \
 event 0 $1 TCPD-Event $2:$3:$4:$5 $6 $7:$9.$10.$11.0 user $13 net; \
 event 0 $1 TCPD-Event $2:$3:$4:$5 $6 $7:$9.$10.0.0 user $13 net; \
 event 0 $1 TCPD-Event $2:$3:$4:$5 $6 $7:$9.0.0.0 user $13 net; \
         add conn_deny_from_$9.$10.$11.0 $0 ; \
         add conn_deny_from_$9.$10.0.0 $0 ; \
         add conn_deny_from_$9.0.0.0 $0
```

Figure 9: Rules to detect horizontal port scans defined by connections to 3 different server hosts from the same client host within 5 minutes. Note: patterns are split for readability. This is not valid for sec input.

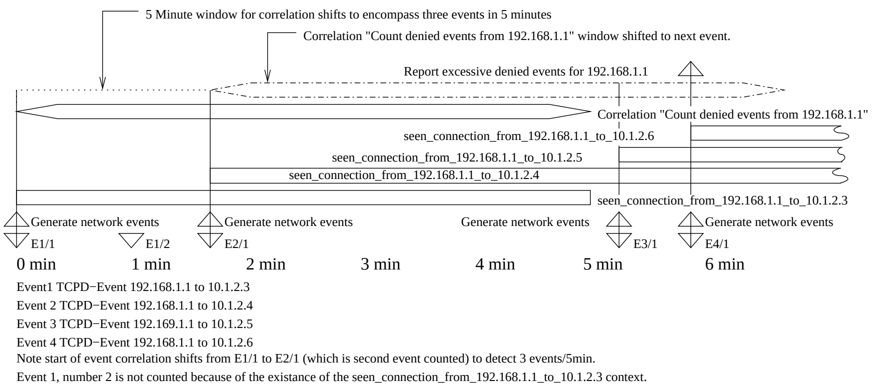

Figure 10: Timeline showing the application of rules to detect horizontal port scans.

using mixtures of the vertical and horizontal counting methods.

While I implemented a "pure" SEC solution, the ability to use Perl functions and data structured from SEC rules provides other solutions[Vaarandi7_2003] to this problem.

3.8 Creating threads of events from multiple sources

Many thread recognition operations involve using one of the pair type rules. Pair rules allow identification of a future (child) event by searching for identifying information taken from the present (parent) event. This provides the ability to stitch a thread through various events by providing a series of pair rules.

There are three times when you need to trigger an action with pair rules:

1. Take action upon receipt of the parent event
2. Take action upon receipt of the child event
3. Take action after some time when the child event has not been received (expiration of the pair rule).

The `Pair` rule provides actions for triggers 1 and 2. The `PairWithWindow` rule provides actions for triggers 2 and 3. None of the currently existing pair rules provides a mechanism for taking actions on all three triggers. Figure 11 shows a way to make up for this limitation by using a context that expires when the pair rule is due to be deleted. Since triggers 2 and trigger 3 are mutually exclusive, part of trigger 2's action is to delete the context that implements the action for to trigger three.

I have used this method for triggering an automatic repair action upon receipt of the first event. The arrival of the second event indicated that the repair worked. If the second event failed to arrive, an alert would be sent when the context timed out. Also, I have triggered additional data gathering scripts from the first event. The second event in this case reported the event and additional data when the the end of the additional data was seen. If the additional data did not arrive on time, I wanted the event to be reported.

This mechanism can replace combinations of `PairWithWindow` and `Single` rules. It simplifies the rules by eliminating duplicate information, such as patterns, that must be kept up to date in both rules.

3.8.1 Correlating across processes

One of more difficult correlation tasks involves creating a session made up of events from multiple processes.

Figure 12 shows a ruleset that sets up a link between parent and child ssh processes. Its application is show in Figure 13.

When a connection to ssh occurs, the parent process, running as root, reports the authentication

```
type=pair
...
action = write - rule triggered ;\
         create take_action_on_pair_expiration 60 (write - rule expired)
...
pattern2=
action2 = write - pattern 2 seen ;\
          delete take_action_on_pair_expiration
...
window=60
```

Figure 11: A method to take an action on all three trigger points in a pair rule.

events and generates information about a user's login. After the authentication process, a child sshd is spawned that is responsible for other operations including port forwarding and logout (disconnect) events. The ruleset in Figure 12 captures all of the events generated by the parent or child ssh process. This includes errors generated by the parent and child ssh processes.

A session starts with the initial network connection to the parent sshd and ends with a connection closed event from the child sshd. I accumulate all events from both processes into a single context. I also have rules (not shown in the example) to report the entire context when unexpected events occur.

The tricky part is accumulating the events from both processes into a single context. The connection between the event streams is provided by a tie event that encompasses unique identifying elements from both event streams and thus ties together the two streams into a single stream.

Each ssh process has its own unique event stream stored in the context `session_log_<hostname>_<pid>`. There is a `Single` rule, omitted for brevity, that accumulates ssh events into this context. When the tie event is seen, it provides the link between the parent sshd session_log context and the child session_log context. The data from the two contexts is merged and the two context names (with the parent and child pid's) are assigned to the same underlying context. Hence the child's `session_log_<hostname>_<child pid>` context and the parent's `session_log_<hostname>_<parent pid>` contexts refer to the same data. After the contexts are linked, actions using either the child context name or the parent context name operate on the same underlying context. Reporting or adding to the context using one of the linked names acts the same regardless of which name is used.

In Figure 13 the first event E1 triggers rule 1 from Figure 12, the `PairWithWindow` rule, to recognize the start of the session. The second half of rule 1 looks for a tie event for the following 60 seconds. There may be many tie events, but there should be only one tie event that contains the the pid of the parent sshd. Since we have that stored in $2, we use it in pattern2. The start of session event is passed onto additional rules (not shown) by setting the continue option on rule 1 to `takenext`. These additional rules record the events in the session_log context identified by system and pid, as in the `session_log_example.org_10240` context of Figure 13.

If the tie event is not found within 60 seconds, the `session_log_example.org_10240` context is reported. However, if the tie event is found as in Figure 13, then a number of other operations occur. The tie event is generated by a script that is run by the child sshd. Therefore it is possible for the child sshd to generate events before the tie event is created. Because of the default rule that adds events to the `session_log_example.org_10245`, additional work must be done when the tie event arrives to preserve the data in the child's session_log. The second part of rule 1 in Figure 12 copies child's session_log context into the variable %b. The child's session_log

```
# rule 1 - recognize the start if an ssh session,
#          and link parent and child event contexts.
type=PairWithWindow
continue=takenext
desc=Recognize ssh session start for $1[$2]
ptype=regexp
pattern=([A-Za-z0-9._-]+) sshd\[([0-9]+)\]: \[[^]]+\] Connection from ([0-9.]+) port [0-9]+
action=report session_log_$1_$2 /bin/cat
desc2=Link parent and child contexts
ptype2=regexp
pattern2=([A-Za-z0-9._-]+) [A-z0-9]+\[[0-9]+\]: \[[^]]+\] SSHD child process +([0-9]+) spawned by $2
action2=copy session_log_$1_$2 %b; \
        delete session_log_$1_$2; \
        alias session_log_$1_%2 session_log_$1_$2; \
        add session_log_$1_$2 $0; \
        event 0 %b; \
alias session_log_owner_$1_%2 session_log_owner_$1_$2; \
window=60

# rule 2 - recognize login event and save username for later use
type=single
desc=Start login timer
ptype=regexp
pattern=([A-Za-z0-9._-]+) sshd\[([0-9]+)\]: \[[^]]+\] Accepted (publickey|password) for ([A-z0-9_-]+) from [0-9.]+ port [0-9]+ (.*)
action=add session_log_$1_$2 $0; add session_log_owner_$1_$2 $4

# rule 3 - handle logout
type=single
desc=Recognize ssh session end
ptype=regexp
pattern=([A-Za-z0-9._-]+) sshd\[([0-9]+)\]: \[[^]]+\] Closing connection to ([0-9.]+)
action= delete session_log_$1_$2; delete session_log_owner_$1_$2
```

Figure 12: Accumulating output from ssh into a single context.

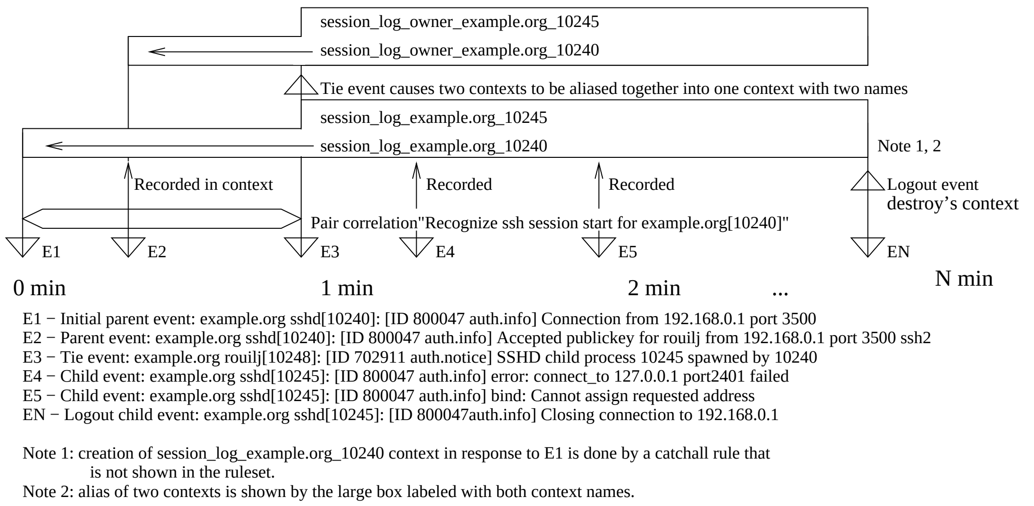

Figure 13: The application of the ssh ruleset showing the key events in establishing the link between parent and child processes.

is then deleted and aliased to the parent session_log. The %2 variable is the value of $2 from the first pattern, the parent process's PID. After `pattern2` is applied, the parent PID is referenced as %2 because $2 is now the second subexpression of `pattern2`. Next the data copied from the child log is injected into the event stream to allow re-analysis and reporting using the combined parent and child context.

The last action for the tie event is to alias the login username stored in the context `session_log_owner_<hostname>_<parent pid>` to a similar context under the child pid. Then any rule that analyzes a child event can obtain the login name by referencing the alias context. Rule 2 in Figure 12 handles the login event and creates the context `session_log_owner_<hostname>_<parent pid>` where it stores the login name for use by the other rules in the ruleset. Rule 2 also stores the login event in the session_log context.

The last rule is very simple. It detects the "close connection" (logout) event and deletes the contexts created during the session. The delivery of event N (EN) in Figure 13 causes deletion of contexts. Deleting an aliased contexts deletes the context data store as well as all the names pointing to the context data store. Rule 3 uses the child PID to delete `session_log_example.org_10245` and `session_log_owner_example.org_10245`, which cleans up all 4 context names (2 from the parent PID and 2 from the child) and both context data stores.

This mechanism can be used for correlating any series of events and passing information between the rules that comprise an analysis mechanism. The trick is to find suitable tie events to allow the thread to be followed. The tie event must contains unique elements found in the events streams that are to be tied together. In the ssh correlation I create a tie event using the pid's of the parent and child events. Every child event includes the PID of the child sshd so that I can easily construct the context name that points to the combined context data store. For the ssh correlation, I create the tie event by running shell commands using the sshrc mechanism and use the logger(1) command to inject the tie event into the data stream. This creates the possibility that the tie event arrives after events from the child process. It would make the correlation easier if I modified the sshd code to provide this tie event since this would generate the events in the correct order for correlation.

Having the events arriving in the wrong order for cross correlation is a problem that is not easily remedied. I suppress reporting of the child events while waiting for the tie event (not shown). Then once the tie event is received, the child events are resubmitted for correlation. This is troublesome and error prone and is an area that warrants further investigation.

4 Strategies to improve performance

One major issue with real-time analysis and notification is the load imposed on the system by the analysis tool and the rate of event processing. The rules can be restructured to reduce the computational load. In other cases the rule analysis load can be distributed across multiple systems or across multiple processes to reduce the load on the system or improve event throughput for particular event streams.

The example rule set from UMB utilizes a number of performance enhancing techniques. Originally these techniques were implemented in a locally modified version of SEC. As of SEC version 2.2.4, the last of the performance improvements has been implemented in the core code.

4.1 Rule construction

For SEC, construction of the rules file(s) plays a large role in improving performance. In SEC, the majority of computation time is occupied with recognizing events using Perl regular expressions. Optimizing these regular expressions to reduce the amount of time needed to apply them improves performance.

However, understanding that SEC applies each rule sequentially allows the reader to put the most often matched rules first in the sequence. Putting the most frequently used rules first reduces the search time needed to find an applicable rule. Sending a USR1 signal to SEC causes it to dump its internal state showing all active contexts, current buffers, and other information including the number of times each rule has been matched. This information is very useful in efficiently restructuring a ruleset

Using rule segmentation to reduce the number of rules that must be scanned before a match is found proves the biggest gains for the least amount of work.

4.1.1 Rule Segmentation

In August 2003, I developed a method of using SEC's multiple configuration file mechanism to prune the number of rules that SEC would have to test before finding a matching rule.

This mechanism provides a limited branching facility within SEC's ruleset. A single criteria filtering rule is shown in Figure 14:

This rule depends on the `Nregexp` pattern type. This causes the rule to match if the pattern *does not* match. The pattern is crafted to filter out events that can not possibly be acted upon by the other rules in the file. In this example I show another guard that is used to prevent this ruleset from considering the event if it has been handled. It consists of a rule that matches all events[2] and fires if the `handle` context is set. If it does not eliminate the event from consideration, I set the `handled` context to prevent other rulesets from processing the event and pass the event to the ruleset. If the final rule triggers, then the event was not handled by any rule in the ruleset. The final rule deletes the `handled` context so that the following rulesets will have a chance to analyze the event.

Note that this last rule is repeated in the final rule file to be applied. There it resets the `handled` context so that the next event will be properly processed by the rulesets.

In addition to a single regexp, multiple patterns can be applied and if any of them select the event, the event will be passed through the rest of the rules in the file. A rule chain to accept an event based on multiple patterns is shown in Figure 15. The multiple filter criteria can be set up to accept/reject the event using complex boolean expressions so that the event must match some patterns, but not other patterns.

```
type=suppress
continue=dontcont
ptype=NRegExp
pattern=^[ABCD]
desc=guard for abcd rules

type=single
continue=dontcont
ptype=TValue
pattern=true
desc=guard for events handled by other \
     ruleset files
action=logonly
context = [handled]

type=single
continue=takenext
ptype=TValue
pattern=true
desc=report handled
action=create handled

<rules here>

type=single
ptype=TValue
pattern=true
desc=Guess we didn't handle this event after all
action=delete handled
```

Figure 14: A sample rule set to allow events to be filtered and prevented from matching other rules in the file.

[2] The `TValue` ptype is only available in SEC 2.2.5 and newer. Before that use regexp with a pattern of `.?`.

```
type= single
desc= Accept event if match2 is seen.
continue= takenext
ptype= regexp
pattern= match2
action= create accept_rule

type= single
desc= Accept event if match3 is seen.
continue= takenext
ptype= regexp
pattern= match3
action= create accept_rule

type= single
desc= Skipping ruleset because neither \
      match2 or match3 were seen.
ptype= TValue
pattern= true
context=  ! accept_rule
action= logonly

type= single
desc= Cleaning up accept_rule context \
      since it has served its purpose.
continue=takenext
ptype= TValue
pattern= true
context= accept_rule
action= delete accept_rule; logonly

<other rules here>
```

Figure 15: A ruleset to filter the input event against multiple criteria. The words “match2” or “match3” must be seen in the input event to be processed by the other rules.

The segmentation method can be arbitrary, however it is be most beneficial to group rules by some common thread such as the generator, using a file/ruleset for analyzing sshd events and another one for xntp events. Another segmentation may be by host type. So hosts with similar hardware are analyzed by the same rules. Hostname is another good segmentation property for rules that are applicable to only one host.

The segmentation can be made more efficient by grouping the input using SEC's ability to monitor multiple files. When SEC monitors multiple files, each file can have a context associated with it. While processing a line from the file, the context is set. For example, reading a line from /var/adm/messages may set the `adm_messages` context, while reading a line from /var/log/syslog would set the `log_syslog` context and clear the `adm_messages` context. This allows segmentation of rules by source file. Offloading the work of grouping to an external application such as syslog-ng provides the ability to group the events not only by facility and level as in classic syslog, but also by other parameters including host name, program, or by a matching regular expression. Since syslog-ng operates on the components of a syslog message rather than the entire message, it is expected to be more efficient in segmenting the events than SEC.

Restructuring the rules for a single SEC process using a simple 5 file segmentation based on the first letter of the event using an 1800 rule ruleset increased throughput by a factor of 3. On a fully optimized ruleset of 50 example rules, running on a SunBlade 150 (128MB of memory, 650Mhz), I have seen rates exceeding 300 lines/sec with less than 40% processor utilization. In tests run under the Cygwin environment on Microsoft windows 2000, 40 rules produced a throughput of 115 log entries per second. This single file path of 40 rules is roughly equivalent to a segmented ruleset of 17 files with 20 rules each for a total of 340 rules, with events equally distributed across the rulesets.

Note that these throughput numbers depend on the event distribution, the length of the events etc. Your mileage may vary.

4.2 Parallelization of rule processing

In addition to optimizing the rules, multiple SEC processes can be run, feeding their composite events to a parent SEC. SEC can watch multiple input streams. It merges all these streams into a single stream for analysis. This merging can interfere with recognition of multi-line events as well as acting to increase the size of an event queue, slowing down the effective throughput rate of a single event stream. Running a child SEC process on an event stream allows faster response to that stream.

SEC's `spawn` action creates a process and creates an event from every line emitted by the child process. The events from these child processes are placed on the front of the event queue for faster processing.

These features allow the creation of a hierarchy of SEC processes to process multiple rules files. This reduces the burden on the parent SEC process by distributing the total number of rules across different processes. In addition, it simplifies the creation of rules when multi-line events must be considered, by preventing the events from being distorted by the injection of other events in the middle of the multi-line event.

SEC is not threaded, so use of concurrent processes is the way to make SEC utilize multiprocessor systems. However, even on uniprocessor systems, it seems to provide better throughput by reducing the mean number of rules that SEC has to try before finding a match.

4.3 Distribution across nodes

SEC has no built-in mechanism for distributing or receiving events with other hosts. However, one can be crafted using the ideas from the last two sections. Although this has not been tested, it is expected to provide a significant performance improvement.

The basic idea is to have the parent SEC process use ssh to spawn child SEC processes on different nodes. These nodes have rules files that handle a portion of the event stream. The logging mechanisms are set up to split the event streams to the nodes so that each node has to work on only a portion of the event stream. Even if the logs are not split across nodes, the reduced number of rules on each node is expected to allow greater throughput.

This can be used in a cluster to allow each host to process its own event streams and report composite events to the parent SEC process for cross-machine correlation operations.

5 Limitations

Like any tool, SEC is not without its limitations. The serial nature of applying SEC's rules limits its throughput. Some form of tree-structured mechanism for specifying the rules would allow faster application. One idea that struck me as interesting is the use of ripple-down rulesets for event correlation[Clark2000] that could simply the creation and maintenance or rulesets as well as speed up execution of complex correlation operations.

As can be seen above, a number of idioms consist of mating a single rule to a more complex correlation rule to receive the desired result. This makes it easy to get lost in the interactions of more complex rulesets. I think more research into commonly used idioms, and the generation of new correlation operations to support these idioms will improve the readability and maintainability of the correlation rules.

The power provided by the use of Perl regular expressions is tempered by the inability to treat the event as a series of fields rather than a single entity. For example, I would prefer to parse the event line into a series of named fields, and use the presence, absence and content of those fields to make the decisions on what rules were executed. I think it would be more efficient and less error prone to come up with a standard form for the event messages and allow SEC to tie pattern matches to particular elements of the event rather then match the entire event. However, implementation of the mechanism may have to wait for the "One True Standard for Event Reporting", and I do not believe I will live long enough to see that become a reality.

The choice of Perl as an implementation language is a major plus because it is a more widely known language than C among the audience for the SEC tool this increases the pool of contributers to the application. Also, Perl allows much more rapid development than C. However, using an interpreted language (even one turned into highly optimized bytecode) does cause a slowdown in execution speed compared to native executable.

SEC does not magically parse timestamps. Its timing is based on the arrival time of the event. This can be a problem in a large network if the travel time cannot be neglected in the event correlation operations.

6 Future Directions

Refinement of the available rule primitives and actions (e.g. the expire action) is an area for investigation. A number of idioms presented above are more difficult to use than I would like. In some cases these idioms could be made easier by adding new correlation types to the language. In other cases a mechanism for storing and retrieving redundant information (such as regular expressions and timing periods) will simplify the idioms. This may be external using a preprocessor such as filepp or m4, or may be an internal mechanism.

Even though SEC development is ongoing, not every idea needs to be implemented in the core. Using available Perl modules and custom libraries it is possible to create functions and routines to enhance the available functionality without making changes to the SEC core. Developing libraries of add-on routines – as well as standard ways of loading and accessing these routines is an ongoing project. This form of extension permits experimentation without bloating SEC's core.

I would like to see some work done in formalizing the concept of rule segmentation and improving the ability to branch within the rule sets to decrease the time spent searching for applicable rules.

7 Availability

SEC is available from http://kodu.neti.ee/ risto/sec/

In addition to the resources at the primary SEC site above, a very good tutorial has been written by Jim Brown[Brown2003] and is available at: http://sixshooter.v6.thrupoint.net/SEC-examples/article.html

An annotated collection of rules files is available from http://www.cs.umb.edu/ rouilj/sec/sec_rules-1.0.tgz. This expands on the rules covered in this talk and provides the tools for the performance testing as well as a sample sshrc file for the ssh correlation example.

8 Conclusion

SEC is a very flexible tool that allows many complex correlations to be specified. Many of these complex correlations can be used to model[Prewett] normal and abnormal sequences of events. Precise modeling of events reduces both the false positive and false negative rates easing the burden on system administrators.

The increased accuracy of the model provided by SEC results in faster recognition of problems leading to reduced downtime, less stress and higher more consistent service levels.

This paper has just scratched the surface of SEC's capabilities. Refinements in rule idioms and linkage of SEC to databases are just a few of the future directions for this tool. Just as prior log analysis applications such as **logsurfer** influenced the design and capabilities of SEC, I believe SEC will serve to foster research and push the envelope of current log analysis and event correlation.

9 Author Biography

John Rouillard is a system administrator whose first Unix experience was on a PDP-11/44 running BSD Unix in 1978. He graduated with a B.S. in Physics from the University of Massachusetts at Boston in 1990. He specializes in automation of sysadmin tasks and as a result is always looking for his next challenging position.

In addition to his system administration job he is also an emergency medical technician. Over the past few years, when not working on an ambulance, he has worked as a planetarium operator, and built test beds for domestic hot water solar heating systems. He has been a member of IEEE since 1987 and can be reached at rouilj@ieee.org.

10 References

2swatch Homepage: ftp://ftp.sdsc.edu/pub/security/PICS/2swatch/README

Brown2003 Brown, Jim, "Working with SEC - the Simple Event Correlator", Web publication, November 23, 2003. URL: http://sixshooter.v6.thrupoint.net/SEC-examples/article.html

Brown5_2004 Brown, Jim, "SEC Logfuscator Project Announcement", simple-evcorr-users mailing list, 3 May 2004. URL: http://sourceforge.net/mailarchive/forum.php?thread_id=2712448&forum_id=2877

Clark2000 Clark, Veronica, "To Maintain an Alarm Correlator", Bachelor's thesis, The University of New South Wales, 2000. URL: http://www.hermes.net.au/pvb/thesis/

Finke2002 Finke, John, "Process Monitor: Detecting Events That Didn't Happen", USENIX Systems Administration (LISA 16) Conference Proceedings, pp. 145-154, USENIX Association, 2002.

Hansen1993 Hansen, Stephen E. and E. Todd Atkins, "Automated System Monitoring and Notification with Swatch", USENIX Systems Administration (LISA VII) Conference Proceedings, pp. 145-156, USENIX Association, November 1993. URL http://www.usenix.org/publications/library/proceedings/lisa93/hansen.html

logsurfer Homepage: http://www.cert.dfn.de/eng/logsurf/

LoGS Prewett, James E., "Listening to Your Cluster with LoGS", The 5th LCI International Conference on Linux Clusters: The HPC Revolution 2004, Linux Cluster Institute, May 2004. URL: http://www.linuxclustersinstitute.org/Linux-HPC-Revolution/Archive/PDF04/05-Prewett_J.pdf

logwatch Bauer, Kirk. Homepage: http://www.logwatch.org/

logsurfer+ Homepage: http://www.crypt.gen.nz/logsurfer/

NNM "Managing Your Network with HP OpenView Network Node Manager", Hewlett-Packard Company, January 2003. Part number J5323-90000.

Prewett Prewett, James E. via private email, March 2004.

ruleCore Homepage: http://www.rulecore.com

Sah02 Sah, Adam, "A New Architecture for Managing Enterprise Log Data", USENIX Systems Administration (LISA XVI) Conference Proceedings, pp. 121-132, USENIX Association, November 2002.

SEC Vaarandi, Risto. Homepage: http://kodu.neti.ee/ risto/sec/

SECman Simple Event Correlator (SEC) manpage URL: http://kodu.neti.ee/ risto/sec/sec.pl.html

SLAPS-2 Homepage: SLAPS-2 http://www.openchannelfoundation.org/projects/SLAPS-2

SHARP Bing, Matt and Carl Erickson, "Extending UNIX System Logging with SHARP", USENIX Systems Administration (LISA XIV) Conference Proceedings, pp. 101-108, USENIX Association, December 2000. URL: http://www.usenix.org/publications/library/proceedings/lisa2000/full_papers/bing/bing_html/index.htm

Snare InterSect Alliance. Homepage: http://www.intersectalliance.com/projects/SnareWindows/index.html

swatch Atkins, Todd. Homepage: http://swatch.sourceforge.net/

Takada02 Takada, Tetsuji and Hideki Koike, "MieLog A Highly Interactive Visual Log Browser Using Information Visualization and Statistical Analysis", USENIX Systems Administration (LISA XVI) Conference Proceedings, pp. 133-144, USENIX Association, November 2002. URL: http://www.usenix.org/events/lisa02/tech/takada.html

Vaarandi7_2003 Vaarandi, Risto, "Re: is this possible with SEC", simple-evcorr-users mailing list, 4 Jul 2003. URL: tt http://sourceforge.net/mailarchive/forum.php?thread_id=2712448&forum_id=2877