Working with SEC- the Simple Event Correlator

Jim Brown

jpb@jimby.name

November 23, 2003

SEC is a powerful event correlation engine written entirely in Perl that is capable of handling a wide variety of event correlation tasks. This article presents a range of examples, from simple to complex, using SEC for event correlation tasks.

Table of Contents

---

1 Introduction

SEC was written by Risto Vaarandi, a professional software developer from Estonia, and is available from the SEC Home Page. Risto has provided a Manual Page, a FAQ and some Examples that are helpful in developing advanced solutions. There is also an active Mailing List.

SEC is a perl script which reads an input stream from a file or pipe and applies pattern matching operations to the input looking for patterns specified by rules, found in configuration files. SEC has several advanced features that make it ideal for a wide variety of event correlation tasks such as log-file analysis, state machine operations, logic analysis and more.

This article introduces basic SEC use and operations, using examples from simple to complex. At the end of this article, the reader should have a solid working knowledge of SEC and be able to create rules for event correlation operations.

> **Note: SEC** is copyrighted by Risto Vaarandi, (risto.vaarandi@eyp.ee) and is distributed under the terms of the GNU GPL. Work on SEC is supported by the Union Bank of Estonia.

---

2 Getting Started

To get started with SEC, download the latest version from the SEC download page at Sourceforge. Unpack the tarball, cd into the directory containing the **sec.pl** perl script.

2.1 A Simple Example

SEC uses a configuration file and takes input from a file or a named pipe. Perform the following steps to create the first example:

- Create a new text file `C2.1.01.conf` with your favorite editor, and copy in the following text:

```
# Example C2.1.01.conf
# Recognize a pattern and log it.
#
type=Single ❶
ptype=RegExp ❷
pattern=foo\s+(\S+) ❸
desc=$0 ❹
action=logonly❺
```

  This example illustrates the following:

  ❶ `Single` is the rule type. SEC includes several different types of rules that are useful in event correlation. This rule is of type *Single*.

  ❷ `RegExp` is the pattern type, one of *RegExp* (for ``Regular Expression'') matching or *SubStr*, for simpler string matching.

  ❸ `foo\s+(\S+)` is the actual pattern- in this case a perl regular expression pattern. This pattern matches the word `foo` followed by one or more spaces, followed by one or more non-space characters, such as `bar, grok, or 1:443z--?.` If you are new to perl regular expressions, you are well advised to read Nicholas Clarks introduction perlretut.

  ❹ `desc` is a variable definition for the pattern description. In this case a perl numbered variable, `$0`, is set to the entire matched pattern.

  ❺ The `action` statement describes the action taken when the pattern is recognized. In this case, the `logonly` action simply writes the pattern to the logfile if one is indicated on the command line, or to standard output if not.

- Save the file (`C2.1.01.conf`) and execute the following command:

```
% perl sec.pl -conf=C2.1.01.conf -input=-
```

  This example will take input from directly from the terminal. Type the following lines of input:

```
foo
foo bar
baz
```

```
bar foo
bar foo baz
```

Notice that SEC responds by replying (logging to standard output) every time a pattern is matched:

```
foo
foo bar
foo bar
baz
bar foo
bar foo baz
bar foo baz
```

In this example, you have created a SEC rule that matches a regular expression, and tested it with input from the terminal.

- To see how SEC operates on files, instead of standard input, copy the test input above into a temporary file, `T2.1.01.txt`. Next, create an empty file that you intend to monitor with SEC, `monitor.me`.

  Now execute SEC with the following command:

  ```
  % perl sec.pl -conf=C2.1.01.conf -input=monitor.me
  ```

  ```
  ~/sec-2.1.11% perl sec.pl -input=monitor.me -conf=C2.1.01.conf
  Simple Event Correlator version 2.1.11
  Reading configuration from C2.1.01.conf
  1 rules loaded from C2.1.01.conf
  ```

  SEC is now running in your terminal session, and reading input from `monitor.me`. In a separate window, or terminal session in the same directory, perform the following:

  ```
  % cat T2.1.01.txt >> monitor.me
  ```

  ```
  ~/sec-2.1.11% perl sec.pl -input=monitor.me -conf=C2.1.01.conf
  Simple Event Correlator version 2.1.11
  Reading configuration from C2.1.01.conf
  1 rules loaded from C2.1.01.conf
  foo bar
  bar foo baz
  ^C
  ```

  SEC parsed the input that arrived in `monitor.me` and performed it's actions (logging to standard output) when it recognized the patterns in the input stream.

  This is the basic operation of SEC. In this case, the ``events'' were the arrival of 2 matched pattern in the input stream. Although this is a simple example, SEC is capable of far more powerful matching and complex operation.

2.2 Other SEC Actions

SEC has well over a dozen different actions it can perform once it matches a pattern in the input stream. Some of the actions depend on *contexts* which will be described in a later section. The following sections describe the actions that can be used all by themselves.

2.2.1 write

`write <filename> [<event text>]`

This example uses the *write* action. The *write* action writes the specified text to the named `filename`.

Copy

```
# Example C2.2.1.01.conf
# Recognize a pattern and write to a file.
# Writes $0 Complete matched pattern, to file.
#

type=Single
ptype=RegExp
pattern=foo\s+(\S+)
desc=$0
action=write - Hello from SEC.  Matched text was $0
```

to `C2.2.1.01.conf`. Note that the action statement now reads:

```
action=write - Hello from SEC. Matched text was $0
```

A filename of ``-'' means write to standard output.

Save `C2.2.1.01.conf` and run the example with input taken from standard input:

```
% perl sec.pl -conf=C2.2.1.01.conf -input=-
```

Type the matched pattern:

```
foo baz
```

and you should see the new action statement thus:

```
~/SEC-examples%perl sec.pl -conf=C2.2.1.01.conf -input=-
Simple Event Correlator version 2.1.11
Reading configuration from C2.2.1.01.conf
1 rules loaded from C2.2.1.01.conf
foo baz
Writing event 'Hello from SEC.  Matched text was foo baz' to file -
Hello from SEC.  Matched text was foo baz
```

The line

```
Writing event 'Hello from SEC.  Matched text was foo baz' to file -
```

is an example of SEC informative debug output. To prevent SEC from writing it's informative output, change the debug level with the addition of the *-debug=n* command line parameter:

```
% perl sec.pl -conf=C2.2.1.01.conf -input=- -debug=4
```

A level of `-debug=4` outputs only notification messages and lower levels of debug output as in:

```
~/SEC-examples%perl sec.pl -conf=C2.2.1.01.conf -input=- -debug=4
Simple Event Correlator version 2.1.11
Reading configuration from C2.2.1.01.conf
foo baz
Hello from SEC.  Matched text was foo baz
^C
```

2.2.2 shellcmd

`shellcmd <shellcmd>`

The *shellcmd* action causes SEC to execute a shell command. The shell command can be any executable program permitted by normal user privileges.

Copy the following small shell script into `S2.2.2.01.sh`.

```
#!/bin/sh
# Shell command for Example C2.2.2.01.conf
#
echo "Start of S2.2.2.01.sh  shell script"
echo "Finding number of files in directory..."
ls | wc -l
echo "End of shell script"
```

Now copy the following SEC configuration into `C2.2.2.01.conf`.

```
# Example C2.2.2.01.conf
# Recognize a pattern and run a shell script.
#

type=Single
ptype=RegExp
pattern=foo\s+(\S+)
desc=$0
action=shellcmd S2.2.2.01.sh
```

Run with:

```
% perl sec.pl -conf=C2.2.2.01.conf -input=-
```

If the shell command is not executable at all, SEC will indicate with the warning:

```
Rule in C2.2.2.01.conf at line 5: Warning - '/home/jpb/SEC-examples/S2.2.2.01.sh'
is not executable
```

If other errors prevent operation of the command, SEC will indicate with suitable warning or error messages. Otherwise, when SEC matches the pattern in the input stream, it will execute the script as requested:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C2.2.2.01.conf -input=-
Simple Event Correlator version 2.1.11
Reading configuration from C2.2.2.01.conf
1 rules loaded from C2.2.2.01.conf
foo bar
Executing shell command '/home/jpb/SEC-examples/S2.2.2.01.sh'
Child 14809 created for command '/home/jpb/SEC-examples/S2.2.2.01.sh'
Start of S2.2.2.01.sh  shell script
Finding number of files in directory...
      33
End of shell script
^C
```

Control of the extra debug output is similar to that described above.

> **Note:** User programs or scripts that are run with the *shell* or *spawn* (see below) actions must be in the PATH available to SEC at run time and not just in the current directory where SEC was started. In particular, if SEC is started with the *-detach* option (described in Section 3) to make it a daemon process, the working directory for SEC is changed to the root directory.
>
> Use of fully qualified path names for scripts is recommended.

2.2.3 spawn

`spawn <shellcmd>`

The *spawn* action is identical to the *shellcmd* action, except that output from the command is fed back into SEC for pattern matching.

Copy the following into `C2.2.3.01.conf`:

```
# Example C2.2.3.01.conf
# Recognize a pattern and use the spawn action.
#

type=Single
ptype=RegExp
pattern=foo\s+(\S+)
desc=$0
action=spawn /home/jpb/SEC-examples/S2.2.3.01.sh

# This rule will match part of the output of S2.2.3.01.sh
type=Single
ptype=RegExp
pattern=ABCD_(.*)
desc=$0
action=write - Matched $0 from $1
```

Now copy the following to `S2.2.3.01.sh`:

```
#!/bin/sh
# Shell command for Example C2.2.3.01.conf
#
echo "Start of S2.2.3.01.sh  shell command"
echo "ABCD_"$$
echo "End of shell script"
```

The `C2.2.3.01.conf` rule file now contains two rules. The first rule matches the same pattern as the previous examples, and will *spawn* the shell script `S2.2.3.01.sh` when triggered.

The second rule matches the pattern `ABC_(.*)`, meaning match `ABCD_` followed by any string of characters until the end of the line.

The `S2.2.3.01.sh` script executed by the first rule echos a start banner then executes the line

`echo "ABCD_"$$`

echoing literal `ABCD_` followed by the process ID of the shell script.

We will need the informational debug output, so use the command:

```
% perl sec.pl -conf=C2.2.3.01.conf -input=-
```

Assuming no errors, the output looks like this:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C2.2.3.01.conf -input=-
Simple Event Correlator version 2.1.11
Reading configuration from C2.2.3.01.conf
2 rules loaded from C2.2.3.01.conf
foo bar
Spawning shell command '/home/jpb/SEC-examples/S2.2.3.01.sh'
Child 14893 created for command '/home/jpb/SEC-examples/S2.2.3.01.sh'
Creating event 'Start of S2.2.3.01.sh  shell command' (received from child 14893)
Creating event 'ABCD_14893' (received from child 14893)
Creating event 'End of shell script' (received from child 14893)
Writing event 'Matched ABCD_14893 from 14893' to file -
Matched ABCD_14893 from 14893
^C
```

The debug output shows how child process 14893 was created by SEC when the first pattern was matched. The next three lines indicate ``Creating event ...'' for each line of output by the `S2.2.3.01.sh` script.

The ``Writing event ...'' line indicates that the second rule has matched its pattern. Finally, the pattern itself is written to standard output by the second rule action statement. This feedback mechanism can be very useful in many event correlation tasks.

spawn is quite often used to create multiple input streams for SEC to process. See Section 3.3 for an example of this usage.

2.2.4 assign And eval

`assign %<letter> [<text>]`

`eval %<letter> <code>`

Both *assign* and *eval* deal with ``% <letter>'' variables. These special variables are of the form ``%n'' where ``n'' is a single letter from the set ``[A-Za-z]''. These variables are *not* perl hashes. They are internal SEC variables that can be used in rules. SEC reserves the following special variables:

Variable	*Description*
%s	The event description string- that is, the entire matched pattern if the parameter is ``$0''. Otherwise it is the entire operand of the *desc* parameter, including backreference substitutions if any.
%t	Textual timestamp, as returned by date(1)
%u	Numeric timestamp, as returned by time(2)

Special variables have global scope across multiple SEC rule files. If an assignment is made by either the *assign* or *eval* actions, SEC maintains that assignment for the life of the program or until the next assignment to that variable.

Copy the following to `C2.2.4.01a.conf`:

```
# Example C2.2.4.01a.conf
# Recognize a pattern and use the assign and eval actions.
#
type=Single
ptype=RegExp
pattern=foo
desc=$0
```

```
action=assign %f Mary had a little lamb,
```

Copy the following to `C2.2.4.01b.conf`:

```
# Example C2.2.4.01b.conf
# This rule uses the eval action to assign its variable.
type=Single
ptype=RegExp
pattern=bar
desc=$0
action=eval %h ($t = "its fleece had neon glow")
```

Copy the following to `C2.2.4.01c.conf`:

```
# Example C2.2.4.01c.conf
# This rule uses the write action to write %f, %h, and %t
type=Single
ptype=RegExp
pattern=baz
desc=$0
action=write - %f %h at %t
```

Note that the patterns for this example have been simplified to match just the single words ``foo" ``bar" and ``baz".

For this example, SEC must be told to use all three configuration files:

```
% perl sec.pl -conf=C2.2.4.01a.conf \
      -conf=C2.2.4.01b.conf \
      -conf=C2.2.4.01c.conf \
      -input=-
```

and input the three words in order:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C2.2.4.01a.conf \
>           -conf=C2.2.4.01b.conf \
>           -conf=C2.2.4.01c.conf \
>           -input=-
Simple Event Correlator version 2.1.11
Reading configuration from C2.2.4.01a.conf
1 rules loaded from C2.2.4.01a.conf
Reading configuration from C2.2.4.01b.conf
1 rules loaded from C2.2.4.01b.conf
Reading configuration from C2.2.4.01c.conf
1 rules loaded from C2.2.4.01c.conf
foo
Assigning value 'Mary had a little lamb,' to variable '%f'
bar
Evaluating code '$t = "its fleece had a neon glow"' and setting variable '%h'
Assigning value 'its fleece had a neon glow' to variable '%h'
baz
Writing event 'Mary had a little lamb, its fleece had a neon glow at Thu Nov 20
04:34:32 2003' to file -
Mary had a little lamb, its fleece had a neon glow at Thu Nov 20 04:34:32 2003
^C
```

Each variable assignment is shown by the debug output. The ``%t" variable is set automatically by SEC.

2.2.5 event

`event [<time>] [<event text>]`

Finally, there is the *event* action. *event* allows the insertion of input to SEC from inside SEC itself. It is a simple feedback mechanism- one controlled by SEC's own rules. The *time* parameter is the number of seconds to wait before inserting the event text into SEC's input stream. For non-zero *time* values this is, in effect, asynchronous in that other input events can be processed between the time the *event* action is called and the time the event text is recognized. Both cases are considered below.

Note that this example also introduces the capability to use multiple actions (separated by a ``;'') within the same rule.

First, copy the following to `C2.2.5.01.conf`:

```
# Example C2.2.5.01.conf
# Use the event action to queue input in the future.
# Also an example of using multiple actions, and
# $variable usage.
#
type=Single
ptype=RegExp
pattern=foo
desc=$0
action=event 5 baz is now matched. ; \
       write - foo matched at %t.  baz event in 5 seconds...

#
type=Single
ptype=RegExp
pattern=bar
desc=$0
action=write - $0 is matched.

# This rule will match the 'baz' event in rule 1
type=Single
ptype=RegExp
pattern=baz
desc=$0
action=write - %s matched at %t
```

Run with:

```
% perl sec.pl -conf=C2.2.5.01.conf -input=-
```

Enter the first input pattern ``foo'' and then immediately enter the second input pattern ``bar''. Then wait and observe that SEC processes the input for ``baz'' after the requested delay of 5 seconds.

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C2.2.5.01.conf -input=-
Simple Event Correlator version 2.1.11
Reading configuration from C2.2.5.01.conf
3 rules loaded from C2.2.5.01.conf
foo
Writing event 'foo matched at Sat Nov 15 07:03:10 2003.  baz event in 5 seconds...'
to file -
foo matched at Sat Nov 15 07:03:10 2003.  baz event in 5 seconds...
bar
Writing event 'bar is matched.' to file -
bar is matched.
Creating event 'baz is now matched.'
Writing event 'baz is now matched. matched at Sat Nov 15 07:03:15 2003' to file -
baz is now matched. matched at Sat Nov 15 07:03:15 2003
^C
```

For a *time* value of zero however, SEC processes events immediately- even before the next line of input from the input stream. To demonstrate this, change action lines in the first rule to read:

```
action=event 0 baz is now matched.; \
       write - foo matched at %t.
```

and rerun the example.

The output now becomes:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C2.2.5.01.conf -input=-
Simple Event Correlator version 2.1.11
Reading configuration from C2.2.5.01.conf
3 rules loaded from C2.2.5.01.conf
foo
Creating event 'baz is now matched.'
Writing event 'foo matched at Sat Nov 15 07:13:35 2003.' to file -
foo matched at Sat Nov 15 07:13:35 2003.
Writing event 'baz is now matched. matched at Sat Nov 15 07:13:35 2003' to file -
baz is now matched. matched at Sat Nov 15 07:13:35 2003
bar
Writing event 'bar is matched.' to file -
bar is matched.
^C
```

This ordering of *event* is true even when input is read from a fast input stream.

---

3 SEC In Depth

SEC has many parameters that control it's operation. These are viewed by simply calling SEC with no parameters:

```
% perl sec.pl


Version: 2.1.11

Usage:
  sec.pl -input=<inputfile> -conf=<conffile pattern> ...

Optional flags:
  -input_timeout=<input timeout>
  -timeout_script=<timeout script>
  -reopen_timeout=<reopen timeout>
  -poll_timeout=<poll timeout>
  -blocksize=<io block size>
  -log=<logfile>
  -debug=<debuglevel>
  -pid=<pidfile>
  -dump=<dumpfile>
  -cleantime=<clean time>
  -bufsize=<input buffer size>
  -evstoresize=<event store size>
  -quoting, -noquoting
  -tail, -notail
  -fromstart, -nofromstart
  -detach, -nodetach
  -intevents, -nointevents
  -testonly, -notestonly

Obsolete flags:
  -separator=<separator>
```

All options are fully described in the SEC Manual Page. Options of the form `-name=value` are required to have a value. As noted above the ``-conf=<conffile pattern>" and ``-input=<inputfile>"

options are required when executing **perl sec.pl**. A brief review of some of the more common options follows:

Option	*Description*
-log=<logfile>	This option specifies the location of a logfile that SEC uses to track it's operation, such as pattern matches, actions, etc. The volume of information is controlled by the `-debug` option.
-debug=<debuglevel>	This option controls how verbose SEC is as it tracks it's operation. The values range between 1 (critical messages) and 6 (debug messages). Each level includes output from lower levels.
-pid=<pidfile>	This option provides for the location of a process ID file. SEC will write it's process ID to this file upon startup.
-dump=<dumpfile>	This option provides for the location of a dump file where SEC can dump its internal data structures, variables and other information upon receipt of the *USR1* signal. The default location is `/tmp/sec.dump`.
-detach	Specifying this option causes SEC to detach itself from the controlling terminal and run as a daemon process. The default is `-nodetach`.
-intevents	This option causes SEC to perform special processing at startup. This special processing is described in the SEC man page.
-testonly	The ``-testonly" option can be used to test for syntax errors in configuration files. It does not start SEC for operation.

The above options are the most common in ordinary usage. See the Manual Page for more information on these and other options.

3.1 SEC Rule Types

SEC contains several rule types (other than *Single* introduced in Section 2) for event correlation. The following table lists the other rule types and gives a short summary of each:

Rule Type	*Description*
SingleWithScript	The *SingleWithScript* rule integrates external scripts with SEC pattern matching, not as an action as used in Section 2.2.2. Its rule definition is similar to the Single rule, with the exception of an additional *script* parameter.
SingleWithSuppress	The *SingleWithSuppress* rule implements compression of multiple matched events into a single event. Its rule definition is also similar to the Single rule, with the exception of an additional *window* parameter.
Pair	The *Pair* combines two or more events into a single event within a given time window. The rule definition for *Pair* contains several new parameters.
PairWithWindow	The *PairWithWindow* also correlates two or more events inside a time window. The rule definition for *PairWithWindow* is similar to the *Pair*.
SingleWithThreshold	The *SingleWithThreshold* rule counts matching events within a time window up to a threshold. When the threshold is reached an action is executed and further matching events after the threshold are ignored for the remainder of the time window.
SingleWith2Thresholds	The *SingleWith2Thresholds* rule counts matching events within time window *t1* and executes an action when the the first threshold is reached. The count is reset and matched events are counted for *t2* seconds. Execute another action if

	the count falls below the second threshold within time window *t2*.
Suppress	The *Suppress* rule suppresses matching input events. This rule has no action, but simply filters out all matching events.
Calendar	The *Calendar* rule executes an action at specific times. The time parameters for this rule are similar to cron(8).

3.2 More SEC Rule Examples

This section presents examples of the rules listed in Section 3.1. For the most part, these rules do not depend on *contexts* which will be covered in a later section.

Some of the rules will require input from different sessions at the same time. Keep two or three sessions open to facilitate working with these rules.

3.2.1 SingleWithScript

The *SingleWithScript* rule combines matching a pattern and the execution of a separate program to determine if the rule is matched. Running a separate program to validate or confirm whether an event is valid is often necessary. For example, matching an IP address in a rule and checking whether the IP address is on a list of valid addresses can not be done by pattern matching in a rule alone. A separate program is required to determine if the matched IP address is on the list.

Copy the following to C3.2.1.01.conf:

```
# Example C3.2.1.01.conf
# Single with script.  Pass matched IP address
# to script for validation. If valid, execute
# action 1; if not valid execute action2.
#
# Note: change script path (and possibly perl path)
#       to match your system.

type=SingleWithScript
ptype=RegExp
pattern=(\d+)\.(\d+)\.(\d+)\.(\d+)
script=/usr/bin/perl /home/jpb/SEC-examples/S3.2.1.01.pl $0
desc=$0
action=write - IP address $0  matches.
action2=write - IP address $0  does NOT match.
```

Note that while this RegExp pattern used will match an IP address, it will also match expressions that are not real IP addresses, such as ``9999.8888.7777.6666''.

Also note that this rule takes two action statements. SEC checks the return value of the called program. If the program returns a zero value, the *action* is executed, if non-zero *action2* is executed.

Next, copy the following to script `S3.2.1.01.pl`

```
#!/usr/bin/perl
#
# Script S3.2.1.01.pl - check if IP argument
# matches a short list of IP addresses.
# Return zero on success, 1 on failure.

@match_list = ( '1.2.3.4',
                '2.3.4.5',
                '3.4.5.6'
```

```
             );
$ip_addr = $ARGV[0] or die "No IP address passed on command line";
foreach $ip (@match_list)
{
  exit (0) if $ip_addr eq $ip;
}
exit 1;
```

Script `S3.2.1.01.pl` accepts a single IP address on the command line passed from the matched rule. If the address matches one of the IPs on its small list of IP addresses, it returns zero, else it returns 1. If there is no IP address at all, the script dies and returns a non-zero value.

Run with:

```
% perl sec.pl -conf=C3.2.1.01.conf -input=-
```

Output looks like this:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C3.2.1.01.pl -input=-
Simple Event Correlator version 2.1.11
Reading configuration from C3.2.1.01.pl
Can't open configuration file C3.2.1.01.pl (No such file or directory)
^C
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C3.2.1.01.conf -input=-
Simple Event Correlator version 2.1.11
Reading configuration from C3.2.1.01.conf
1 rules loaded from C3.2.1.01.conf
1.2.3.4
Child 16396 created for command '/usr/bin/perl /home/jpb/SEC-examples/S3.2.1.01.pl
1.2.3.4'
Child 16396 terminated with exitcode 0
Writing event 'IP address 1.2.3.4  matches.' to file -
IP address 1.2.3.4  matches.
5.6.7.8
Child 16398 created for command '/usr/bin/perl /home/jpb/SEC-examples/S3.2.1.01.pl
5.6.7.8'
Child 16398 terminated with non-zero exitcode 1
Writing event 'IP address 5.6.7.8  does NOT match.' to file -
IP address 5.6.7.8  does NOT match.
^C
```

More robust IP address matching is possible with the **Net::IP_Addr** perl module.

3.2.2 SingleWithSuppress

With the *SingleWithSuppress* rule it is possible to become alerted to an event the first time it is seen, then ignore the same event within a time window.

Copy the following to C3.2.2.01.conf:

```
# Example C3.2.2.01.conf
# Example of SingleWithSuppress
#
type=SingleWithSuppress
ptype=RegExp
pattern=foo
desc=$0
action=write - $0 suppressed for 5 seconds at %t
window=5
```

Run with:

```
% perl sec.pl -conf=C3.2.2.01.conf -input=-
```

and continuously enter ``foo'' as rapidly as possible.

Output:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C3.2.2.01.conf -input=-
Simple Event Correlator version 2.1.11
Reading configuration from C3.2.2.01.conf
1 rules loaded from C3.2.2.01.conf
foo
Writing event 'foo suppressed for 5 seconds at Sat Nov 15 17:04:38 2003' to file -
foo suppressed for 5 seconds at Sat Nov 15 17:04:38 2003
foo
foo
foo
foo
foo
foo
Writing event 'foo suppressed for 5 seconds at Sat Nov 15 17:04:44 2003' to file -
foo suppressed for 5 seconds at Sat Nov 15 17:04:44 2003
foo
foo
foo
foo
foo
foo
Writing event 'foo suppressed for 5 seconds at Sat Nov 15 17:04:50 2003' to file -
foo suppressed for 5 seconds at Sat Nov 15 17:04:50 2003
foo
^C
```

3.2.3 Pair

The *Pair* rule handles two different events, matched by two different patterns in its rule definition. The rule uses a time window which is set upon the first occurrence of event A. If event B occurs within the time window, events A and B are considered correlated, and the entire rule is considered matched. Otherwise, the correlation operation for the pair terminates.

There are two action statements, each corresponding to its own pattern. Action one (*action*) is executed when event A is matched. Action two (*action2*) is executed if event B occurs within the time window.

Copy the following to `C3.2.3.01.conf`:

```
# Example C3.2.3.01.conf
# Example Pair rule.
# Match event A and B within window.

type=Pair
ptype=RegExp
pattern=foo
desc=$0
action=write - foo matched at %t. Start window of 5 seconds for bar ...
ptype2=RegExp
pattern2=bar
desc2=$0
action2=write - bar matched at %t.  bar is within window!
window=5
```

Run with:

```
% perl sec.pl -conf=C3.2.3.01.conf -input=-
```

When running this rule, first enter ``foo'' and ``bar'' close together (i.e. within 5 seconds). Then enter ``foo'' and wait to enter ``bar'' until the window is past (i.e. more than 5 seconds.) The first time the *Pair* rule will correlate them together, while the second time they are not correlated.

Output will look similar to:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C3.2.3.01.conf -input=-
Simple Event Correlator version 2.1.11
Reading configuration from C3.2.3.01.conf
1 rules loaded from C3.2.3.01.conf
foo
Writing event 'foo matched at Sat Nov 15 18:17:07 2003. Start window of 5 seconds
for bar ...' to file -
foo matched at Sat Nov 15 18:17:07 2003. Start window of 5 seconds for bar ...
bar
Writing event 'bar matched at Sat Nov 15 18:17:09 2003.  bar is within window!' to
file -
bar matched at Sat Nov 15 18:17:09 2003.  bar is within window!
foo
Writing event 'foo matched at Sat Nov 15 18:17:14 2003. Start window of 5 seconds
for bar ...' to file -
foo matched at Sat Nov 15 18:17:14 2003. Start window of 5 seconds for bar ...
bar
^C
```

3.2.4 PairWithWindow

At first glance, the *PairWithWindow* rule appears identical to the *Pair* rule. Both contain two patterns, two actions, and a time window.

The difference is that, in *PairWithWindow* the *action2* is executed if events A and B both occur within the time window. If A occurs, but B does not occur, then *action* is executed.

Copy the following to `C3.2.4.01.conf`:

```
# Example C3.2.4.01.conf
# Example PairWithWindow rule.
# Match both events A and B within window executes action2.
# If event B does not occur within window, execute action.

type=PairWithWindow
ptype=RegExp
pattern=foo
desc=$0
action=write - foo matched, bar NOT matched within window.
ptype2=RegExp
pattern2=bar
desc2=$0
action2=write - foo and bar both matched within 5 second window!
window=5
```

Run with:

```
% perl sec.pl -conf=C3.2.4.01.conf -input=-
```

When running this rule, first enter ``foo'' and ``bar'' close together (i.e. within 5 seconds). Then enter ``foo'' and wait to enter ``bar'' until the window is past (i.e. more than 5 seconds.) The first time the *PairWithWindow* rule will correlate them together, while the second time they are not correlated.

Output looks like:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C3.2.4.01.conf -input=-
Simple Event Correlator version 2.1.11
Reading configuration from C3.2.4.01.conf
1 rules loaded from C3.2.4.01.conf
foo
bar
Writing event 'foo and bar both matched within 5 second window!' to file -
foo and bar both matched within 5 second window!

foo
Writing event 'foo matched, bar NOT matched within window.' to file -
foo matched, bar NOT matched within window.
^C
```

3.2.5 SingleWithThreshold

The *SingleWithThreshold* rule is used to ``count'' the number of matched events within a time window. If the number exceeds the threshold, the *action* is executes

If the number of matched events, does not exceed the threshold within the time window, the time window ``slides''- that is, start time for the correlation window is moved to the second occurrence of the matched pattern. This process repeats, until the time window expires with no new matched events.

Copy the following to `C3.2.5.01.conf`:

```
# Example C3.2.5.01.conf
# Example SingleWithThreshold rule.
# Match event A thresh number of times in window
# and execute action.  Slide window if needed
# until window expires.

type=SingleWithThreshold
ptype=RegExp
pattern=foo
desc=$0
action=write - foo matched three times in 10 seconds!
window=10
thresh=3
```

Run with:

```
% perl sec.pl -conf=C3.2.5.01.conf -input=-
```

When running this rule, first enter ``foo'' three times close together (i.e. within 10 seconds). The *action* will execute.

Then enter ``foo'' slowly, waiting five to eight seconds between each entry. Since there are never three entries (*thresh=3*) entered within the sliding window, the rule is not matched and the *action* is not executed.

The first time the *SingleWithThreshold* rule will correlate them together, while the second time they are not correlated.

Output looks similar to:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C3.2.5.01.conf -input=-
Simple Event Correlator version 2.1.11
Reading configuration from C3.2.5.01.conf
1 rules loaded from C3.2.5.01.conf
foo
foo
foo
Writing event 'foo matched three times in 10 seconds!' to file -
foo matched three times in 10 seconds!

foo
foo
foo
foo
foo
foo
^C
```

3.2.6 SingleWith2Threshds

The *SingleWith2Thresholds* rule is very similar to the *SingleWithThreshold* rule, except that we can now definitely determine when events stop. This is done with a second threshold and a second timer window.

SingleWith2Thresholds counts the number of matched events and executes *action* when the number is above *thresh* events.

Once this low threshold (watermark) is reached, SEC starts *window2* and counts additional matched events. When the number of events falls below *thresh2* events within *window2*, SEC executes *action2*.

Note that both windows are sliding windows- that is, the begining time of the window moves to the time of the next match if the time window of the first match expires.

Copy the following to `C3.2.6.01.conf`:

```
# Example C3.2.6.01.conf
# Example SingleWith2Threshholds rule.
# Match thresh A events (go above low watermark) and execute action.
# Then switch to thresh2 and window2 to count more A events.
# If less than thresh2 A events occur in window2 (stay under high
# watermark), execute action2.


type=SingleWith2Thresholds
ptype=RegExp
pattern=foo
desc=$0
action=write - foo hit low watermark (3) at time %t
window=5
thresh=3
desc2=$0
action2=write - foo stayed under high watermark (5) at time %t
window2=10
thresh2=5
```

Run with:

```
% perl sec.pl -conf=C3.2.6.01.conf -input=-
```

In this example, it will be necessary to time your entries fairly closely. The first example shows three matched events followed by *action*, then less than *thresh2* (5 events) within *window2* (10 seconds).

The second example shows multiple events after the low watermark then a marked slowing of entries, that results in *action2*.

Output from this example:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C3.2.6.01.conf -input=-
Simple Event Correlator version 2.1.11
Reading configuration from C3.2.6.01.conf
1 rules loaded from C3.2.6.01.conf
foo
foo
foo
Writing event 'foo hit low watermark (3) at time Mon Nov 17 09:42:05 2003' to file
-
foo hit low watermark (3) at time Mon Nov 17 09:42:05 2003
foo
foo
Writing event 'foo stayed under high watermark (5) at time Mon Nov 17 09:42:16
2003' to file -
foo stayed under high watermark (5) at time Mon Nov 17 09:42:16 2003

foo
foo
foo
Writing event 'foo hit low watermark (3) at time Mon Nov 17 09:42:22 2003' to file
-
foo hit low watermark (3) at time Mon Nov 17 09:42:22 2003
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo slowing way down...
foo
foo
Writing event 'foo stayed under high watermark (5) at time Mon Nov 17 09:42:49
2003' to file -
foo stayed under high watermark (5) at time Mon Nov 17 09:42:49 2003
^C
```

3.2.7 Suppress

The *Suppress* rule is very intuitive- events matching the pattern are suppressed. Since this rule has no *action* statement, it does nothing.

Suppress is particularly useful with the *context* parameter. Contexts are discussed in Section 4, and an additional example of *Suppress* can be found there. This example does not depend on the use of contexts.

Copy the following to `C3.2.07.01.conf`:

```
# Example C3.2.7.01.conf
# Example of Suppress.
# First rule suppresses 'foo'.
# Second rule matches any pattern and
# executes write action.

type=Suppress
ptype=RegExp
pattern=foo
desc=$0

type=Single
ptype=RegExp
pattern=(.*)
desc=$0
action=write - entry was: $0
```

Run with:

```
% perl sec.pl -conf=C3.2.7.01.conf -input=-
```

In this example, the first rule suppresses ``foo'' while the second rule matches any pattern and writes it to standard output. Since ``foo'' is already suppressed by the first rule, it will never be written by the second rule.

Output looks similar to:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C3.2.7.01.conf -input=-
Simple Event Correlator version 2.1.11
Reading configuration from C3.2.7.01.conf
2 rules loaded from C3.2.7.01.conf
bar
Writing event 'entry was: bar' to file -
entry was: bar
baz
Writing event 'entry was: baz' to file -
entry was: baz
foo
foo
foo
bar
Writing event 'entry was: bar' to file -
entry was: bar
baz
Writing event 'entry was: baz' to file -
entry was: baz
^C
```

3.2.8 Calendar

The *Calendar* rule is another easy to understand rule. It executes *action* statements at specific times. The time specification is similar to that used by cron(8), and is detailed in crontab(5).

Note that the time specifications do not include the Vixie cron extensions.

Copy the following to `C3.2.8.01.conf`:

```
# Example C3.2.8.01.conf
# Example calendar rule.
# Write a message every minute.

type=Calendar
time=* * * * *
```

```
desc=$0
action=write - The time is now: %t
```

This example takes no user input. However, the ``-input'' parameter must still be present on the command line. Run with:

```
% perl sec.pl -conf=C3.2.8.01.conf -input=-
```

Output is similar to:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C3.2.8.01.conf -input=-
Simple Event Correlator version 2.1.11
Reading configuration from C3.2.8.01.conf
1 rules loaded from C3.2.8.01.conf
Writing event 'The time is now: Mon Nov 17 10:40:42 2003' to file -
The time is now: Mon Nov 17 10:40:42 2003
Writing event 'The time is now: Mon Nov 17 10:41:00 2003' to file -
The time is now: Mon Nov 17 10:41:00 2003
Writing event 'The time is now: Mon Nov 17 10:42:00 2003' to file -
The time is now: Mon Nov 17 10:42:00 2003
^C
```

Using the ``-debug=4'' parameter removes the informational debug statements and results in just:

```
The time is now: Mon Nov 17 10:46:35 2003
The time is now: Mon Nov 17 10:47:00 2003
The time is now: Mon Nov 17 10:48:00 2003
The time is now: Mon Nov 17 10:49:00 2003
The time is now: Mon Nov 17 10:50:00 2003
```

Note also that SEC invokes the *action* of all calendar rules at startup, but only at the top of each minute thereafter. Actions that must not occur too closely together must take this into account.

Running applications from SEC is similar. This example runs a script that checks MD5 checksums on a list of files every five minutes. The script takes a single parameter- ``MD5_CHECK'':

```
#
# Run the SystemCheck.sh script every five minutes.
#
type=Calendar
time=0,5,10,15,20,25,30,35,40,45,50,55 * * * *
desc=MD5_CHECK
action=shellcmd /home/jpb/SEC-examples/SystemCheck.sh %s
```

3.3 Multiple Input Streams

This section presents another use of the *spawn* action- obtaining input from multiple input streams. As shown in Section 2.2.3, *spawn* starts another program and redirects the program output as input to SEC. Both the original input stream and this new input stream are processed by SEC as a single input stream.

To set up SEC to read multiple files, the **tail** program is often used as in the following example.

Copy the following to `C3.3.01.conf`:

```
#
# Example C3.3.01.conf
# Multiple input files with spawn.
#
#
type=Single
ptype=RegExp
pattern=foo
continue=TakeNext
desc=$0
action=spawn /usr/bin/tail -f ./aaa.in ;\
       spawn /usr/bin/tail -f ./bbb.in ;\
       spawn /usr/bin/tail -f ./ccc.in ;


# Match lines beginning with aaa:
type=Single
ptype=RegExp
pattern=^aaa:(.*)
desc=$0
action=write aaa.out %s

# Match lines beginning with bbb:
type=Single
ptype=RegExp
pattern=^bbb:(.*)
desc=$0
action=write bbb.out %s

# Match lines beginning with ccc:
type=Single
ptype=RegExp
pattern=^ccc:(.*)
desc=$0
action=write ccc.out %s


# Match all other lines
type=Single
ptype=RegExp
pattern=(.*)
desc=$0
action=write other.out %s
```

In this example, the *spawn* action is part of a rule that matches input ``foo". This means that the spawn actions will not occur until ``foo" is recognized in the input stream.

After the *tail* commands will forward input from their respective files into SEC. SEC will treat all input streams the same, and parse input from all streams according to all rules.

Note that the input files `aaa.in`, `bbb.in`, and `ccc.in` must exist before running the example. Use the **touch** command to create these empty files as follows:

```
% touch aaa.in bbb.in ccc.in
```

Note also that the last rule is a catch-all rule: if the input does not get recognized by any other rule, it will be written to `other.out`

Run with:

```
% perl sec.pl -conf=C3.3.01.conf -input=-
```

The session starts as follows:

```
jpb@jpb-lt:~/SEC-examples/tmp$perl ../sec.pl -conf=C3.3.01.conf -input=-
Simple Event Correlator version 2.1.11
```

```
Reading configuration from C3.3.01.conf
5 rules loaded from C3.3.01.conf
foo
Spawning shell command '/usr/bin/tail -f ./aaa.in'
Child 15940 created for command '/usr/bin/tail -f ./aaa.in'
Spawning shell command '/usr/bin/tail -f ./bbb.in'
Child 15941 created for command '/usr/bin/tail -f ./bbb.in'
Spawning shell command '/usr/bin/tail -f ./ccc.in'
Child 15942 created for command '/usr/bin/tail -f ./ccc.in'
Writing event 'foo' to file other.out
aaa:input from terminal
Writing event 'aaa:input from terminal' to file aaa.out
bbb:input from terminal
Writing event 'bbb:input from terminal' to file bbb.out
ccc:input from terminal
Writing event 'ccc:input from terminal' to file ccc.out
ddd:input from terminal
Writing event 'ddd:input from terminal' to file other.out
```

So far all input has been from the terminal. In another window or session in the same directory, perform the following commands:

```
% echo "aaa:from other session copied into ccc.in" >> ccc.in

% echo "bbb:from other session copied into aaa.in" >> aaa.in

% echo "ddd:from other session copied into bbb.in" >> bbb.in
```

SEC processes these inputs as well:

```
Creating event 'aaa:from other session copied into ccc.in' (received from child
15956)
Writing event 'aaa:from other session copied into ccc.in' to file aaa.out
Creating event 'bbb:from other session copied into aaa.in' (received from child
15954)
Writing event 'bbb:from other session copied into aaa.in' to file bbb.out
Creating event 'ddd:from other session copied into bbb.in' (received from child
15955)
Writing event 'ddd:from other session copied into bbb.in' to file other.out
^C
```

Examine each output file to determine its contents:

```
jpb@jpb-lt:~/SEC-examples/tmp$cat aaa.out
aaa:input from terminal
aaa:from other session copied into ccc.in

jpb@jpb-lt:~/SEC-examples/tmp$cat bbb.out
bbb:input from terminal
bbb:from other session copied into aaa.in

jpb@jpb-lt:~/SEC-examples/tmp$cat ccc.out
ccc:input from terminal

jpb@jpb-lt:~/SEC-examples/tmp$cat other.out
foo
ddd:input from terminal
ddd:from other session copied into bbb.in
```

As shown above, SEC parsed the input, regardless of where it came from, and performed the actions indicated on each matched rule.

3.4 Pipe Output

This section looks at another feature of the *write* action; writing to a ``named pipe", also called a ``fifo". This feature provides a simple method of inter-process communication (IPC).

Most Unix systems already have the ability to create and use named pipes. With SEC, the only requirement is that the named pipe must exist before writing to it. Typically this is performed with the **mkfifo** or **mknod** command. Check your system documentation for how to create a named pipe. On BSD based systems, the command is:

```
% mkfifo SEC_fifo
```

This creates a named pipe in the current directory:

```
prw-r--r--  1 jpb  staff  -    0 Nov 15 12:38 SEC_fifo
```

To be useful, there must be a way to get data out of the named pipe. Copy the following perl script to file `S3.4.01.pl`:

```
#!/usr/bin/perl
#
# Example S3.4.01.pl - Script to read data out of a named pipe.
#
$| = 1;
$filename = "./SEC_fifo";
open(FIFO, "+< $filename") or die "fifo  $!";
print "Start of S3.4.01.pl.  Reading $filename...\n";

while (<FIFO>)
{
  $inputline = $_;
  print "Got: $inputline";
}
```

This script simply copies whatever it receives from the named pipe to standard output.

Run with:

```
% perl S3.4.01.pl
```

The program announces its startup banner and waits for input:

```
jpb@jpb-lt:~/SEC-examples/tmp$perl S3.4.01.pl
Start of S3.4.01.pl.  Reading ./SEC_fifo...
```

In a separate window or session, create `C3.4.01.conf` with the following rule:

```
# Example C3.4.01.conf
# Recognize any pattern and write to a FIFO.
#
type=Single
ptype=RegExp
pattern=(.*)
desc=$0
action=write SEC_fifo %s
```

Run with:

```
% perl sec.pl -conf=C3.4.01.conf -input=-
```

and type some text:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C3.4.01.conf -input=-
Simple Event Correlator version 2.1.11
Reading configuration from C3.4.01.conf
1 rules loaded from C3.4.01.conf
This is
Writing event 'This is' to file SEC_fifo
a test of
Writing event 'a test of' to file SEC_fifo
SEC named pipes.
Writing event 'SEC named pipes.' to file SEC_fifo
^C
```

The output from the reader script looks like:

```
jpb@jpb-lt:~/SEC-examples$perl S3.4.01.pl
Start of S3.4.01.pl.  Reading ./SEC_fifo...
Got: This is
Got: a test of
Got: SEC named pipes.
^C
```

SEC will complain if it can't write to the pipe. In this example, the reading script was manually terminated after the first input line. All remaining lines received an error:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C3.4.01.conf -input=-
Simple Event Correlator version 2.1.11
Reading configuration from C3.4.01.conf
1 rules loaded from C3.4.01.conf
This is
Writing event 'This is' to file SEC_fifo
another test
Writing event 'another test' to file SEC_fifo
Can't open pipe SEC_fifo for writing event 'another test'!
of SEC named pipes.
Writing event 'of SEC named pipes.' to file SEC_fifo
Can't open pipe SEC_fifo for writing event 'of SEC named pipes.'!
^C
```

While trivial, this example illustrates the basic mechanism for passing event correlation analysis to another program. Part Two of this article contains an example of joining SEC with a database system and uses this technique to pass input into the database.

---

4 Using Contexts

SEC has the ability to define and use *contexts* with rules. A context is ``the interrelated conditions in which something exists or occurs". In SEC, a context ``exists or occurs" when it is created by a rule action. In addition to the actions covered in Section 2.2, SEC provides additional actions that create, destroy, add to, report on, or otherwise manipulate a context.

Contexts can act as event stores. Events can be added to contexts as they occur. A collection of events in a context can be input to a script to be saved in a file.

Initially, contexts might be hard to mentally grasp. They are a logical instance of data. They act more like perl variables- springing into existence when created. In fact, they are implemented as perl hashes. Their chief characteristic is their logical existence as they relate to rules.

Most rules have a ``context'' parameter that is used to interact with contexts. SEC actually uses this parameter in two different ways. The operand is either:

- A logical expression containing a context name
- A perl mini-program if the operand begins with a ``=''

Both types are shown below.

```
Below is a  rule that uses a context parameter:
#
type=Single
ptype=RegExp
pattern=foo
context=FOO_CONTEXT  ❶
desc=$0
action=logonly
```

```
Below is a rule that uses a context mini-program:

#
type=Single
ptype=RegExp
pattern=foo
context= =({my $var = 1; return $var})  ❷
desc=$0
action=logonly
```

❶

FOO_CONTEXT is a context name. The name is tested to see if the context exists as a part of evaluating whether the rule matches the input.

This rule says:

> Match the pattern ``foo'' and if the context named ``FOO_CONTEXT'' exists, execute the action statement.

❷

This *context* statement contains a perl mini-program. The operand begins with a ``=''. The perl mini-program is evaluated (with the perl **eval** statement) and the return value is checked. If true (non zero), the *context* parameter is considered true.

A natural question is: ``How do contexts get created in the first place?'' The answer is that they are created by another rule.

4.1 Simple Context Example

The first example in this section simply checks for the existence of a context. If the context exists, the rule is executed. In order to create the context, a second rule is needed, with the *create* action to create the context for the first rule.

Copy the following to `C4.1.01.conf`:

```
# Example C4.1.01.conf
# Context example.
# Write action if context named FOO_CONTEXT exists.
# Create context with second rule, matching 'bar'
#

type=Single
ptype=RegExp
pattern=foo
context=FOO_CONTEXT
desc=$0
action=write - context exists, writing foo here

type=Single
ptype=RegExp
pattern=bar
desc=$0
action=create FOO_CONTEXT
```

In this example, ``foo'' is recognized by the pattern in the first rule, but the rule does not trigger until the context `FOO_CONTEXT` is created. The context is created by the second rule which matches the ``bar'' pattern, and has an action to create context `FOO_CONTEXT`.

Run with:

```
% perl sec.pl -conf=C4.1.01.conf -input=-
```

Output looks like:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C4.1.01.conf -input=-
Simple Event Correlator version 2.1.11
Reading configuration from C4.1.01.conf
2 rules loaded from C4.1.01.conf
foo
foo
foo
bar
Creating context 'FOO_CONTEXT'
foo
Writing event 'context exists, writing foo here' to file -
context exists, writing foo here
foo
Writing event 'context exists, writing foo here' to file -
context exists, writing foo here
foo
Writing event 'context exists, writing foo here' to file -
context exists, writing foo here
^C
```

Once the second rule was matched, and the *create* action was executed, the context `FOO_CONTEXT` sprang into existence. After that, the first rule was triggered when pattern ``foo'' was matched in the input stream.

4.1.1 Context Lifetimes

In the example above, the *create* action created context `FOO_CONTEXT` with, by default, infinite lifetime. The *create* action actually has the following syntax:

```
create [<name> [<time> [<action list>] ] ]
```

The *name* is the name of the context. *time* is the lifetime of the context in seconds. The *action list* parameter is a list of any SEC action statements- each action statement separated by semi-colons, ``;''.

To see the effect of the *time* (context lifetime) parameter, change the line in `C4.1.01.conf` from

```
action=create FOO_CONTEXT
```

to

```
action=create FOO_CONTEXT 5
```

This creates context `FOO_CONTEXT` with a lifetime of 5 seconds. At the end of its lifetime, the context expires, at which point it is no longer valid. SEC notes the context expiration in its informative debug output.

Re-running the above example:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C4.1.01.conf -input=-
Simple Event Correlator version 2.1.11
Reading configuration from C4.1.01.conf
2 rules loaded from C4.1.01.conf
foo
foo
foo
bar
Creating context 'FOO_CONTEXT'
foo
Writing event 'context exists, writing foo here' to file -
context exists, writing foo here
foo
Writing event 'context exists, writing foo here' to file -
context exists, writing foo here
Deleting stale context 'FOO_CONTEXT'
foo
foo
foo
^C
```

Here context `FOO_CONTEXT` did not exist until the second rule was matched (by matching ``bar'') at which time it was created with a lifetime of five seconds.

During its five second lifetime the first rule pattern ``foo'' was successfully matched twice and the action statement for the first rule executed twice.

At the end of its lifetime, context `FOO_CONTEXT` expired, which is noted by the SEC informative debug output ``Deleting stale context 'FOO_CONTEXT'''.

After the context expired, the first rule is no longer successfully triggered, even though ``foo'' is matched, because context `FOO_CONTEXT` no longer exists.

Contexts are one of the most powerful features of SEC. It is possible to develop complex and intricate event correlation scenarios by combining pattern matching, context creation and manipulation, and actions.

4.1.2 *create* Action Lists

As shown above, contexts themselves can have an *action list* that is part of the *create* statement. The *action list* is a list of any SEC permissible actions.

Action lists on the *create* statement are executed when the context expires. This allows for contexts to perform actions just before they disappear. The next example shows this usage.

Copy the following to `C4.1.02.conf`:

```
# Example C4.1.02.conf
# Context example.
# Write action if context named FOO_CONTEXT exists.
# Create context with second rule, matching 'bar'
#

type=Single
ptype=RegExp
pattern=foo
context=FOO_CONTEXT
desc=$0
action=write - context exists, writing foo here

type=Single
ptype=RegExp
pattern=bar
desc=$0
action=create FOO_CONTEXT 10 (write - Context FOO_CONTEXT is ending; write -
Goodbye...;)
```

This configuration is almost identical to `C4.1.01.conf` with the exception of an *action list* on the *create* statement. When context `FOO_CONTEXT` expires after 10 seconds, it performs two *write* actions.

Run with:

```
% perl sec.pl -conf=C4.1.02.conf -input=-
```

Output looks like:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C4.1.02.conf -input=-
Simple Event Correlator version 2.1.11
Reading configuration from C4.1.02.conf
2 rules loaded from C4.1.02.conf
foo
foo
foo
bar
Creating context 'FOO_CONTEXT'
foo
Writing event 'context exists, writing foo here' to file -
context exists, writing foo here
foo
Writing event 'context exists, writing foo here' to file -
context exists, writing foo here
foo
Writing event 'context exists, writing foo here' to file -
context exists, writing foo here
Deleting stale context 'FOO_CONTEXT'
```

```
Writing event 'Context FOO_CONTEXT is ending' to file -
Context FOO_CONTEXT is ending
Writing event 'Goodbye...' to file -
Goodbye...
^C
```

Just before context `FOO_CONTEXT` ended, its *action list* was executed, resulting in the two *write* statement sending their text to standard output. Without the informative debug output, the results look like:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C4.1.02.conf -input=- -debug=4
Simple Event Correlator version 2.1.11
Reading configuration from C4.1.02.conf
foo
foo
foo
bar
foo
context exists, writing foo here
foo
context exists, writing foo here
Context FOO_CONTEXT is ending
Goodbye...
^C
```

4.2 Other Actions Pertaining To Contexts

So far, contexts have been used as simple boolean operators- they exist or they do not exist. Contexts have an additional, very useful feature- they can store events. This makes them ideal for event reporting.

Context parameters, such as the lifetime and the action list can also be changed. The context name however, can not be changed.

This section presents additional actions that are useful in manipulating contexts.

4.2.1 Set

`set <name> <time> [<action list>]`

set is used to change the settings for a context. The context identified by *name* must already exist. Use *set* to change the *time* and/or the *action list* for the context.

Typically, a simple context will be created with *create* which will be later *set* with the desired lifetime and action list. Below is an example:

Copy the following to `C4.2.01.conf`:

```
# Example C4.2.01.conf
# Context example.
# Create context with second rule, matching 'bar'
# Use set action to extend life of context with first rule.
#

type=Single
ptype=RegExp
pattern=foo
```

```
context=FOO_CONTEXT
desc=$0
action=write - setting context FOO_CONTEXT lifetime to 10 seconds at %t; \
       set FOO_CONTEXT 10 (write - Context FOO_CONTEXT died at %t)

type=Single
ptype=RegExp
pattern=bar
desc=$0
action=create FOO_CONTEXT; write - creating context FOO_CONTEXT;
```

Run with the `-debug=4` parameter to skip the informative output:

```
% perl sec.pl -conf=C4.2.01.conf -input=- -debug=4
```

Output looks like:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C4.2.01.conf -input=- -debug=4
Simple Event Correlator version 2.1.11
Reading configuration from C4.2.01.conf
foo
foo
bar
creating context FOO_CONTEXT
foo
setting context FOO_CONTEXT lifetime to 10 seconds at Mon Nov 17 22:37:17 2003
foo
setting context FOO_CONTEXT lifetime to 10 seconds at Mon Nov 17 22:37:21 2003
foo
setting context FOO_CONTEXT lifetime to 10 seconds at Mon Nov 17 22:37:24 2003
foo
setting context FOO_CONTEXT lifetime to 10 seconds at Mon Nov 17 22:37:29 2003
foo
setting context FOO_CONTEXT lifetime to 10 seconds at Mon Nov 17 22:37:35 2003
Context FOO_CONTEXT died at Mon Nov 17 22:37:46 2003
^C
```

This example creates a context (with the second rule) then keeps extending the lifetime of the context (with the first rule) to 10 seconds every time the first rule is triggered. Only when there is no more ``foo'' input does the context finally expire.

4.2.2 Add and Report

`add <name> [<event text>]`

`report <name> [<shellcmd>]`

The *add* action adds an event to a context. The context parameters do not change, only the contents of the context. Think of the context as a box- *add* adds events as they occur to the box. Other actions in this section (*delete*, *fill*,*report*, *copy*,and *empty*) operate on the contents of that box.

While its nice to have the capability to add events to a context, its necessary to have a way to get them out. The *report* action can be used to list the contents of a context through a system command (*shellcmd*). SEC runs the named *shellcmd* (subject to normal user permissions) and passes the content of the context to standard input of the command.

The following example creates a context (`MY_CONTEXT`), adds events to it, and reports the contents with **/bin/cat**.

The first attempt is shown in `C4.2.02.conf`:

```
# Example C4.2.02.conf
# Rule 1: create context MY_CONTEXT
# Rule 2: Add to context MY_CONTEXT
# Rule 3: report context MY_CONTEXT
# Third rule reports context
#

# Pattern 'CreateMe' creates the action
type=Single
ptype=RegExp
pattern=CreateMe
continue=TakeNext
desc=$0
action=create MY_CONTEXT

# Anything else gets added to context 'MY_CONTEXT'
type=Single
ptype=RegExp
pattern=(.*)
desc=$0
action=add MY_CONTEXT $0

# Pattern 'ReportMe' executes the report action
type=Single
ptype=RegExp
pattern=ReportMe
context=MY_CONTEXT
desc=$0
action=report MY_CONTEXT /bin/cat
```

When run, this attempt looks like:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C4.2.02.conf -input=-
Simple Event Correlator version 2.1.11
Reading configuration from C4.2.02.conf
3 rules loaded from C4.2.02.conf
CreateMe
Creating context 'MY_CONTEXT'
Adding event 'CreateMe' to context 'MY_CONTEXT'
line one
Adding event 'line one' to context 'MY_CONTEXT'
line two
Adding event 'line two' to context 'MY_CONTEXT'
ReportMe
Adding event 'ReportMe' to context 'MY_CONTEXT'
ReportMe
Adding event 'ReportMe' to context 'MY_CONTEXT'
???
Adding event '???' to context 'MY_CONTEXT'
^C
```

What is happening in this case is that the pattern match for rule 2 ``(.*)'' is swallowing the input ``ReportMe'' before it gets to the third rule.

A possible fix is to add `continue=TakeNext` to rule 2:

```
# Anything else gets added to context 'MY_CONTEXT'
type=Single
ptype=RegExp
pattern=(.*)
continue=TakeNext
desc=$0
action=add MY_CONTEXT $0
```

This produces:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C4.2.02.conf -input=-
```

```
Simple Event Correlator version 2.1.11
Reading configuration from C4.2.02.conf
3 rules loaded from C4.2.02.conf
CreateMe
Creating context 'MY_CONTEXT'
Adding event 'CreateMe' to context 'MY_CONTEXT'
line one
Adding event 'line one' to context 'MY_CONTEXT'
line two
Adding event 'line two' to context 'MY_CONTEXT'
ReportMe
Adding event 'ReportMe' to context 'MY_CONTEXT'
Reporting the event store of context 'MY_CONTEXT' through shell command '/bin/cat'
Child 2180 created for command '/bin/cat'
CreateMe
line one
line two
ReportMe
^C
```

This works, but the ``CreateMe'' and ``ReportMe'' input ends up in the context and itself is reported.

A bit more elegant solution is to reverse the order of rules 2 and 3. And now, no `continue=TakeNext` is needed:

```
# Example C4.2.02.conf
# Rule 1: create context MY_CONTEXT
# Rule 2: report context MY_CONTEXT
# Rule 3: Add to context MY_CONTEXT
# Third rule reports context
#

# Pattern 'CreateMe' creates the action
type=Single
ptype=RegExp
pattern=CreateMe
desc=$0
action=create MY_CONTEXT

# Pattern 'ReportMe' executes the report action
type=Single
ptype=RegExp
pattern=ReportMe
context=MY_CONTEXT
desc=$0
action=report MY_CONTEXT /bin/cat

# Anything else gets added to context 'MY_CONTEXT'
type=Single
ptype=RegExp
pattern=(.*)
desc=$0
action=add MY_CONTEXT $0
```

Without the informative debug statements the output is:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C4.2.02.conf -input=- -debug=4
Simple Event Correlator version 2.1.11
Reading configuration from C4.2.02.conf
CreateMe
line one
line two
ReportMe
line one
line two
^C
```

This example highlights the fact that rule order can be important in SEC.

The last example in this section highlights a very common use of SEC- match patterns, keep various patterns in different contexts, and report those patterns; either ``on demand" as in this example, or though a context action (Section 4.1.2). The SEC Manual Page provides an excellent example of capturing FTP transactions and reporting on those actions. See the section on the *Single* rule.

4.2.3 Fill

`fill <name> [<event text>]`

The *fill* action was introduced in SEC version 2.1.11.. This action behaves like *add* with one important difference- the context is emptied before the *event text* is added.

Copy the following to `C4.2.03.conf`:

```
# Example C4.2.03.conf
# Rule 1: create context MY_CONTEXT
# Rule 2: report context MY_CONTEXT
# Rule 3: 'fill' action clears context
# Rule 4: add to context MY_CONTEXT
#

# Pattern 'CreateMe' creates the action
type=Single
ptype=RegExp
pattern=CreateMe
desc=$0
action=create MY_CONTEXT

# Pattern 'ReportMe' executes the report action
type=Single
ptype=RegExp
pattern=ReportMe
context=MY_CONTEXT
desc=$0
action=report MY_CONTEXT /bin/cat

# Pattern 'FillMe' executes fill action- clears context and
#  adds text specified
type=Single
ptype=RegExp
pattern=FillMe
desc=$0
action=fill MY_CONTEXT  fill executed

# Anything else gets added to context 'MY_CONTEXT'
type=Single
ptype=RegExp
pattern=(.*)
desc=$0
action=add MY_CONTEXT $0
```

Run without informative debug:

```
% perl sec.pl -conf=C4.2.03.conf -input=- -debug=4
```

Output looks similar to:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C4.2.03.conf -input=- -debug=4
Simple Event Correlator version 2.1.11
Reading configuration from C4.2.03.conf
CreateMe
line one
line two
ReportMe
line one
line two
```

```
line three
line four
ReportMe
line one
line two
line three
line four
FillMe
line five
line six
ReportMe
fill executed
line five
line six
^C
```

As shown above, the *fill* action clears out the context. If the *event text* is not given, *%s* (the ``desc'' parameter) is assumed for it's value. The next matching event will be added to the context immediately after any text included with the *fill* action.

4.2.4 Delete

`delete <name>`

The *delete* action simply deletes the context. Happily, if the context doesn't exist no operation is performed. Just as in perl, there is no harm in undefining a variable that doesn't exist.

To see the effect of this action, copy the following to `C4.2.04.conf`:

```
# Example C4.2.04.conf
# Rule 1: create context MY_CONTEXT
# Rule 2: report context MY_CONTEXT
# Rule 3: 'delete' action deletes context
# Rule 4: add to context MY_CONTEXT
#

# Pattern 'CreateMe' creates the action
type=Single
ptype=RegExp
pattern=CreateMe
desc=$0
action=create MY_CONTEXT

# Pattern 'ReportMe' executes the report action
type=Single
ptype=RegExp
pattern=ReportMe
context=MY_CONTEXT
desc=$0
action=report MY_CONTEXT /bin/cat


# Pattern 'DeleteMe' executes delete action- deletes context
type=Single
ptype=RegExp
pattern=DeleteMe
desc=$0
action=delete MY_CONTEXT

# Anything else gets added to context 'MY_CONTEXT'
type=Single
ptype=RegExp
pattern=(.*)
desc=$0
action=add MY_CONTEXT $0
```

Run without informative debug:

```
% perl sec.pl -conf=C4.2.04.conf -input=- -debug=4
```

The output should look similar to:

```
pb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C4.2.04.conf -input=- -debug=4
Simple Event Correlator version 2.1.11
Reading configuration from C4.2.04.conf
CreateMe
line one
line two
line three
ReportMe
line one
line two
line three
DeleteMe
ReportMe
DeleteMe
ReportMe
^C
```

4.2.5 Copy

`copy <name> %<letter>`

The *copy* action takes the contents of a context and assigns it to a SEC letter variable. In previous versions of SEC, there was no way to access the contents of a context from inside SEC. The only way was to run *report*, or *shellcmd* actions which feed context content to external programs.

This can be useful for determining current state in a complex set of events. Suppose that there were several motors used in an experiment, each one sending state information such as ``ON" ``OFF", ``SLOWSPEED", ``FASTSPEED", etc. to a central console. As the experiment proceeds and the motors change state, it is now possible to get the current state (and all previous states) into a single variable.

4.3 Perl Mini-Programs

Introduced in Section 4, perl mini-programs are a feature designed to provide additional flexibility in rule matching. Instead of a checking for a fixed context name or being concerned with context content, perl mini-programs are evaluated by the perl **eval** command and the results returned to SEC. A non-zero return results in the context being TRUE, and any action statement is executed.

The mini-program feature allows an external 'program' to determine whether an matched string should be processed. Consider the following example:

```
# Example C4.3.01.conf
# Mini-program examples
# Use mini-program to determine if matching string should
# be processed.
# This mini-program returns true if a randomly generated
# number is between 5 and 9.
#

type=Single
ptype=RegExp
pattern=bar
desc=$0
context= =({(int rand(10) >= 5)})
```

```
action=write - mini-program was true this time
```

Copy the above to `C4.3.01.conf`.

Run this example with:

```
% perl sec.pl -conf=C4.3.01.conf -input=-
```

Output will look similar to:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C4.3.01.conf -input=-
Simple Event Correlator version 2.1.11
Reading configuration from C4.3.01.conf
1 rules loaded from C4.3.01.conf
bar
Writing event 'mini-program was true this time' to file -
mini-program was true this time
bar
bar
Writing event 'mini-program was true this time' to file -
mini-program was true this time
bar
bar
bar
Writing event 'mini-program was true this time' to file -
mini-program was true this time
bar
Writing event 'mini-program was true this time' to file -
mini-program was true this time
bar
bar
bar
bar
Writing event 'mini-program was true this time' to file -
mini-program was true this time
bar
Writing event 'mini-program was true this time' to file -
mini-program was true this time
bar
bar
^C
```

4.4 Calendar Rules and Contexts

The *calendar* rule (discussed previously in Section 3.2.8) is similar to the Unix **cron** utility. Calendar rules are often used with contexts. A frequent question is how to set up SEC to understand the notion of ``Business Hours" and ``Off Hours". One solution might be:

```
type=Calendar
time=0 6 * * *
desc=BUSINESS_HOURS
action=create %s 54000 (create OFF_HOURS 32400)
```

This solution creates context BUSINESS_HOURS at 6:00am every morning that lasts for fifteen hours (54000 seconds), they creates context OFF_HOURS which lasts for nine hours (32400 seconds). Unfortunately, this solution doesn't take weekends into account, and more importantly, it only works if SEC happens to be running at precisely 6:00am.

A more robust solution actually takes three rules- one to set BUSINESS_HOURS every minute between 6:00am - 8:59pm, M-F; and set OFF_HOURS 9:00pm - 5:59am M-F and all day Saturday and Sunday:

```
# Business Hours- applies 6:00am - 8:59pm, M-F
# Off hours -  9:00pm - 5:59am M-F and all day Sat and Sun.
# Set every minute by the calendar rules below
# See  crontab(5)
#
# Three calendars are needed since calendar is run every minute
# and we don't actually know when SEC will be started (or restarted)
#
#
# Calendar Rule 1- set BUSINESS_HOURS
type=Calendar
time=* 6-20 * * 1,2,3,4,5
desc=BUSINESS_HOURS
context=!BUSINESS_HOURS
action=create %s;\
       write - Switched to Business Hours;\
       delete OFF_HOURS;

# Calendar Rule 2- set OFF_HOURS during M-F
type=Calendar
time=* 0-5,21-23 * * *
context=!OFF_HOURS
desc=OFF_HOURS
action=create %s;\
       write - Switched to Off Hours;\
       delete BUSINESS_HOURS;

# Calendar Rule 3- set OFF_HOURS during Sat,Sun
type=Calendar
time=* * * * 6,7
context=!OFF_HOURS
desc=OFF_HOURS
action=create %s;\
       write - Switched to Off Hours;\
       delete BUSINESS_HOURS;
```

On startup, this rule set will write to standard out:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=C4.4.01.conf -input=- -debug=4
Simple Event Correlator version 2.1.11
Reading configuration from C4.4.01.conf
Switched to Business Hours
^C
```

At the appropriate time SEC will shuffle it's contexts and write ``Switched to Off Hours" or ``Switched to Business Hours" to standard output.

---

5 Applications to Logfile Monitoring

SEC excels at logfile monitoring. With an appropriate set of rules, SEC can be configured to monitor logfiles for a single host, a small group of systems, or a whole enterprise. And since rules are simply text files, any time a new rule is needed it can be quickly developed and added to SEC on the fly.

This section begins with some simple logging entries, examines some typical problems and solutions, then explores issues of robustness, and scalability. The configurations in this section have been developed for BSD systems, but the basic principles can be applied to any Unix system. In Part Two of this article, there is also a section on monitoring Windows security event logs.

> **Important:** Before beginning to monitor logfiles, first make sure you are allowed to do so! If you are using your own system, or are already the system administrator for your system, fine. Otherwise, obtain permission from your system administrator before mucking around in logfiles.

5.1 Logfile Monitoring Examples

To begin, login as a normal user and deliberately fail a privilege escalation to root:

```
% su -
Password: (deliberately type a wrong password)
```

Next, examine your logfile (`/var/log/messages` on most BSD systems) for the logged notification:

```
Nov 18 09:37:38 jpb-lt su: BAD SU jpb to root on /dev/ttyp3
```

A SEC rule to pick up this logfile entry is:

```
#   Bad su
# -----------
#
type=Single
ptype=RegExp
desc=$0
pattern=\S+\s+\d+\s+\S+\s+(\S+)\s+su: BAD SU (\S+) to (\S+) on (\S+)
action=write - $2 failed SU to $3 on $1 at %t
```

Save the above rule as `L01.conf` and run with:

```
% perl sec.pl -conf=L01.conf -input=/var/log/messages
```

Leave the above command running in one screen, and in another screen or terminal session, try to deliberately fail an **su** command as noted above.

Output should be similar to:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=L01.conf -input=/var/log/messages
Simple Event Correlator version 2.1.11
Reading configuration from L01.conf
1 rules loaded from L01.conf
Writing event 'jpb failed SU to root on jpb-lt at Tue Nov 18 10:21:36 2003' to file
-
jpb failed SU to root on jpb-lt at Tue Nov 18 10:21:36 2003
```

SEC has successfully detected a bad **su** to root.

Depending on your level of paranoia, you may decide that you can use a *SingleWithThreshold* rule to allow for some fat on the fingers.

5.1.1 Monitoring SEC Notifications

There are two problems with the above examples. The first is that by using the *write* action, all we have done is replaced one logging mechanism with another. Using *write* to generate a new logfile, or even to write to standard output, is equivalent to syslog. We haven't gained anything.

A partial solution is to move to an 'interrupt' method of notification. In the above example, the *write* action can be replaced with an action to email the system administrator using the *pipe* action:

```
action=pipe '$2 failed SU to $3 on $1 at %t' /usr/bin/mailx -s "LoginFailure"
root@localhost
```

Output now looks like:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=L01.conf -input=/var/log/messages
Simple Event Correlator version 2.1.11
Reading configuration from L01.conf
1 rules loaded from L01.conf
Feeding event 'jpb failed SU to root on jpb-lt at Sat Nov 22 13:24:16 2003' to
shell command '/usr/bin/mailx -s "LoginFailure" root@localhost'
Child 1053 created for command '/usr/bin/mailx -s "LoginFailure" root@localhost'
```

The email message contains the completed text with all backreferences filled in:

```
Date: Sat, 22 Nov 2003 13:24:16 -0500 (EST)
From: "Jim B." <jpb@jpb-lt>
To: root@jpb-lt
Subject: LoginFailure

jpb failed SU to root on jpb-lt at Sat Nov 22 13:24:16 2003
```

The second problem is that this doesn't scale well. There is no aggregation of event data. It's quite common for email and paging notification systems to be turned off in large organizations because there are simply too many notifications- too many pages and emails. Unless there is some sort of aggregation and/or filtering mechanism, pages and emails are likely to be ignored.

A solution to this problem is presented in Part Two of this article.

5.2 Using SEC with Syslog on BSD Systems

In order to use SEC to process **syslog** messages, we first need to decide which messages we want to monitor. Below is a small collection of syslog messages organized into various categories.

This collection contains entries that are generally *security related*. There are many other syslog entries in the BSD kernel, system, and user programs. A more comprehensive collection will be used in Part Two of this article,

---

5.2.1 MONITOR Rules

The MONITOR group of syslog messages concerns syslogd itself, and messages that infer that promiscuous mode has been enabled. The syslog messages for the MONITOR group are:

```
Logs involving syslogd disabled or unusual promiscuous mode (MONITOR)
----------------------------------------------------------------------
Nov 15 20:02:48 foohost syslogd: exiting on signal 15
Nov 22 02:00:02 foohost syslogd: restart
Nov 11 15:58:55 foohost /kernel: de0: promiscuous mode enabled
Nov 11 15:58:57 foohost /kernel: de0: promiscuous mode disabled
```

SEC rules for the MONITOR group are as follows:

```
# MONITOR.conf - SEC rules to pick up disruptive monitoring
# events.
#
#Logs involving syslogd disabled or unusual promiscuous mode (MONITOR)
#----------------------------------------------------------------------
#Nov 15 20:02:48 foohost syslogd: exiting on signal 15
#Nov 22 02:00:02 foohost syslogd: restart
#Nov 11 15:58:55 foohost /kernel: de0: promiscuous mode enabled
#Nov 11 15:58:57 foohost /kernel: de0: promiscuous mode disabled
#

#
# Syslog Exit
# -----------
#
type=Single
ptype=RegExp
pattern=\S+\s+\d+\s+\S+\s+(\S+)\s+syslogd: exiting on signal (\d+)
desc=$0
action=write - MONITOR: $1 syslog exit on signal $2 at %t

#
# Syslog Restart
# ---------------
#
type=Single
ptype=RegExp
pattern=\S+\s+\d+\s+\S+\s+(\S+)\s+syslogd: restart
desc=$0
action=write - MONITOR: $1 syslog restart at %t

#
# Syslog Exit
# -----------
#
type=Single
ptype=RegExp
pattern=\S+\s+\d+\s+\S+\s+(\S+)\s+/kernel: (\S+) promiscuous mode (\S+)
desc=$0
action=write - MONITOR: $1 $2 promiscuous mode $3 at %t
```

Copy the above syslog entries to `MONITOR.txt`.

5.2.2 PHYSMOD Rules

The PHYSMOD group concerns messages that infer that physical modifications have been performed on the host such as a new PCMCIA card, or that a cable has been disconnected. The syslog messages for the PHYSMOD group are:

```
Logs involving physical modifications (PHYSMOD)
-----------------------------------------------
Nov 14 21:11:19 foohost /kernel: pccard: card inserted, slot 0
Nov 14 22:28:09 foohost /kernel: pccard: card removed, slot 0
Nov 12 19:46:31 foohost /kernel: de0: link down: cable problem?
```

```
Nov 12 19:46:42 foohost /kernel: de0: autosense failed: cable problem?
Oct 18 06:26:37 foohost pccardd[49]: ep0: 3Com Corporation (/3C589/) inserted.
Oct 18 06:26:42 foohost pccardd[49]: pccardd started
```

Copy the above syslog entries to `PHYSMOD.txt`.

SEC rules for the PHYSMOD group are:

```
# PHYSMOD.conf - Events concerning physical modifications
#                to the system.
#
#
#Logs involving physical modifications (PHYSMOD)
#-------------------------------------------------
#Nov 14 21:11:19 foohost /kernel: pccard: card inserted, slot 0
#Nov 14 22:28:09 foohost /kernel: pccard: card removed, slot 0
#Nov 12 19:46:31 foohost /kernel: de0: link down: cable problem?
#Nov 12 19:46:42 foohost /kernel: de0: autosense failed: cable problem?
#Oct 18 06:26:37 foohost pccardd[49]: ep0: 3Com Corporation (/3C589/) inserted.
#Oct 18 06:26:42 foohost pccardd[49]: pccardd started
#

#
# PCMCIA Card Insertion, Removal
# ------------------------------
#
type=Single
ptype=RegExp
pattern=\S+\s+\d+\s+\S+\s+(\S+)\s+/kernel: pccard: card (\S+), slot (\d+)
desc=$0
action=write - PHYSMOD: $1 pccard: card $2 in slot $3 at %t

#
# PCMCIA Card Daemon
# ------------------------------
#
type=Single
ptype=RegExp
pattern=\S+\s+\d+\s+\S+\s+(\S+)\s+pccardd\[\d+\]: (.*)
desc=$0
action=write - PHYSMOD: $1 pccardd: $2 at %t

#
# Cabling Problem
# ---------------
#
type=Single
ptype=RegExp
pattern=\S+\s+\d+\s+\S+\s+(\S+)\s+/kernel: (\S+)\s+(.*?:) cable problem
desc=$0
action=write - PHYSMOD: $1 cable problem on $2, text: $3 at %t
```

5.2.3 USERACT Rules

The USERACT group concerns messages that are relevant to user activity, such as login failures or change of userid. The syslog messages for the USERACT group are:

```
Logs involving logins, change of UID and privilege escalations (USERACT)
------------------------------------------------------------------------
Nov 14 12:14:58 foohost sshd[3388]: fatal: Timeout before authentication for
192.168.1.1
Nov 14 19:58:34 foohost sshd[6597]: Bad protocol version identification
'^B^S^D^Q^L' from 192.168.1.100
Nov 15 04:02:24 foohost login: 1 LOGIN FAILURE ON ttyp2
Nov 15 04:02:24 foohost login: 1 LOGIN FAILURE ON ttyp2, mysql
Oct 18 02:52:04 foohost login: ROOT LOGIN (root) ON ttyv1
Oct 18 03:20:46 foohost login: 2 LOGIN FAILURES ON ttyv0
Nov 10 19:40:03 foohost su: jpb to root on /dev/ttyp0
Nov 18 09:37:38 foohost su: BAD SU jpb to root on /dev/ttyp3
Nov 22 12:26:44 foohost su: BAD SU badboy to root on /dev/ttyp0
```

```
Oct 18 06:11:11 foohost login: login on ttyv0 as root
Oct 18 06:16:53 foohost sshd[131]: Accepted keyboard-interactive/pam for jpb from
192.168.1.1 port 1077 ssh2
Nov 14 12:55:29 foohost sshd[3425]: Accepted keyboard-interactive/pam for jpb from
fe80::2c0:4fff:fe18:13fd%ep0 port 27492 ssh2
```

Copy the above syslog entries to `USERACT.txt`.

SEC rules for the USERACT group are:

```
# USERACT.conf - Events concerning user activities.
#
#Logs involving logins, change of UID and privilege escalations (USERACT)
#-------------------------------------------------------------------------
#Nov 14 12:14:58 foohost sshd[3388]: fatal: Timeout before authentication for
192.168.1.1
#Nov 14 19:58:34 foohost sshd[6597]: Bad protocol version identification
'^B^S^D^Q^L' from 192.168.1.100
#Oct 18 06:16:53 foohost sshd[131]: Accepted keyboard-interactive/pam for jpb from
192.168.1.1 port 1077 ssh2
#Nov 14 12:55:29 foohost sshd[3425]: Accepted keyboard-interactive/pam for jpb from
fe80::2c0:4fff:fe18:13fd%ep0 port 27492 ssh2
#Nov 15 04:02:24 foohost login: 1 LOGIN FAILURE ON ttyp2
#Nov 15 04:02:24 foohost login: 1 LOGIN FAILURE ON ttyp2, mysql
#Oct 18 03:20:46 foohost login: 2 LOGIN FAILURES ON ttyv0
#Oct 18 02:52:04 foohost login: ROOT LOGIN (root) ON ttyv1
#Oct 18 06:11:11 foohost login: login on ttyv0 as root
#Nov 10 19:40:03 foohost su: jpb to root on /dev/ttyp0
#Nov 18 09:37:38 foohost su: BAD SU jpb to root on /dev/ttyp3
#Nov 22 12:26:44 foohost su: BAD SU badboy to root on /dev/ttyp0
#

#
# sshd Problems
# --------------
#
type=Single
ptype=RegExp
pattern=\S+\s+\d+\s+\S+\s+(\S+)\s+sshd\[\d+\]: (fatal|Bad)(.*)
desc=$0
action=write - USERACT: $1 sshd $2 problem, text: $3 at %t

#
# sshd Accepted
# --------------
#
type=Single
ptype=RegExp
pattern=\S+\s+\d+\s+\S+\s+(\S+)\s+sshd\[\d+\]: Accepted (.*)
desc=$0
action=write - USERACT: $1 sshd accepted login, text: $2 at %t

#
# login FAILURES
# ---------------
#
type=Single
ptype=RegExp
pattern=\S+\s+\d+\s+\S+\s+(\S+)\s+login: (.*?FAILURE.)(.*?ON) (.*)
desc=$0
action=write - USERACT: $1 login $2 on $4 at %t

#
# su  bad
# -----------------
#
type=Single
ptype=RegExp
pattern=\S+\s+\d+\s+\S+\s+(\S+)\s+su: (BAD SU) (\S+) to (\S+) on (\S+)
desc=$0
action=write - USERACT: $1 su: $2 $3 to $4 on $5 at %t

#Nov 10 19:40:03 foohost su: jpb to root on /dev/ttyp0
#Nov 18 09:37:38 foohost su: BAD SU jpb to root on /dev/ttyp3
#Nov 22 12:26:44 foohost su: BAD SU badboy to root on /dev/ttyp0
#
#
# su  good to root
# -----------------
```

```
#
type=Single
ptype=RegExp
pattern=\S+\s+\d+\s+\S+\s+(\S+)\s+su: (\S+) to root on (\S+)
desc=$0
action=write - USERACT: $1 su: $2 to ROOT on $4  at %t
```

5.3 Testing The Rule Set

Testing the SEC rule set for the above three groups of rules is accomplished by running SEC with all three configuration files and dumping the collection of syslog entries into SEC's input.

Use the following command line to test the rule sets:

```
% perl sec.pl -conf=MONITOR.conf \
        -conf=PHYSMOD.conf \
        -conf=USERACT.conf \
        -input=SEC.test \
        -debug=4
```

This command sets up `SEC.test` as SEC's input file. Create an empty `SEC.test` and run the above command. Then, in a separate window or terminal session, perform the following command:

```
% cat MONITOR.txt PHYSMOD.txt USERACT.txt >> SEC.test
```

Check SEC's output against the rules for any missing matches on input. The results will look similar to:

```
jpb@jpb-lt:~/SEC-examples$perl sec.pl -conf=MONITOR.conf \
>                 -conf=PHYSMOD.conf \
>                 -conf=USERACT.conf \
>                 -input=SEC.test \
>                 -debug=4
Simple Event Correlator version 2.1.11
Reading configuration from MONITOR.conf
Reading configuration from PHYSMOD.conf
Reading configuration from USERACT.conf
MONITOR: foohost syslog exit on signal 15 at Sat Nov 22 17:44:07 2003
MONITOR: foohost syslog restart at Sat Nov 22 17:44:07 2003
MONITOR: foohost de0: promiscuous mode enabled at Sat Nov 22 17:44:07 2003
MONITOR: foohost de0: promiscuous mode disabled at Sat Nov 22 17:44:07 2003
PHYSMOD: foohost pccard: card inserted in slot 0 at Sat Nov 22 17:44:07 2003
PHYSMOD: foohost pccard: card removed in slot 0 at Sat Nov 22 17:44:07 2003
PHYSMOD: foohost cable problem on de0:, text: link down: at Sat Nov 22 17:44:07
2003
PHYSMOD: foohost cable problem on de0:, text: autosense failed: at Sat Nov 22
17:44:07 2003
PHYSMOD: foohost pccardd: ep0: 3Com Corporation (/3C589/) inserted. at Sat Nov 22
17:44:07 2003
PHYSMOD: foohost pccardd: pccardd started at Sat Nov 22 17:44:07 2003
USERACT: foohost sshd fatal problem, text: : Timeout before authentication for
192.168.1.1 at Sat Nov 22 17:44:07 2003
USERACT: foohost sshd Bad problem, text:  protocol version identification
'^B^S^D^Q^L' from 192.168.1.100 at Sat Nov 22 17:44:07 2003
USERACT: foohost login 1 LOGIN FAILURE  on ttyp2 at Sat Nov 22 17:44:07 2003
USERACT: foohost login 1 LOGIN FAILURE  on ttyp2, mysql at Sat Nov 22 17:44:07 2003
USERACT: foohost login 2 LOGIN FAILURES on ttyv0 at Sat Nov 22 17:44:07 2003
USERACT: foohost su: jpb to ROOT on $4  at Sat Nov 22 17:44:07 2003
USERACT: foohost su: BAD SU jpb to root on /dev/ttyp3 at Sat Nov 22 17:44:07 2003
USERACT: foohost su: BAD SU badboy to root on /dev/ttyp0 at Sat Nov 22 17:44:07
2003
USERACT: foohost sshd accepted login, text: keyboard-interactive/pam for jpb from
192.168.1.1 port 1077 ssh2 at Sat Nov 22 17:44:07 2003
USERACT: foohost sshd accepted login, text: keyboard-interactive/pam for jpb from
fe80::2c0:4fff:fe18:13fd%ep0 port 27492 ssh2 at Sat Nov 22 17:44:07 2003
```

```
^C
```

Once you are satisfied, run the following command to monitor syslog message on your BSD logfile, `/var/log/messages`:

```
% perl sec.pl -conf=MONITOR.conf \
        -conf=PHYSMOD.conf \
        -conf=USERACT.conf \
        -input=/var/log/messages \
        -debug=4
```

Until we have a more comprehensive approach to correlating events you may wish to keep the action statements limited to simple *write* actions shown in the above rules. If you do decide to use email or paging for your actions (as shown in Section 5.1.1), test on a small subset of rules first.

Use the *-detach* to put the SEC monitoring application in the background:

```
% perl sec.pl -conf=MONITOR.conf \
        -conf=PHYSMOD.conf \
        -conf=USERACT.conf \
        -input=/var/log/messages \
        -debug=4  \
        -detach
```

---

6 Part One Conclusion

In this article, we've looked at **SEC**, the Simple Event Correlator, from Risto Vaarandi. Using **perl** regular expression patterns along with SEC rule types and actions, we have examined how SEC works, and how to create and use rule sets for logfile monitoring.

In **Part Two**, we will expand on these rules, and provide more meaningful event correlation. We will also build a relational data base around SEC and use it to categorize and report on log entries.
