Using SEC

DAVID LANG

David Lang is a Staff IT Engineer at Intuit, where he has spent more than a decade working in the Security Department for the Banking Division. He was introduced to Linux in 1993 and has been making his living with Linux since 1996. He is an Amateur Extra Class Radio Operator and served on the communications staff of the Civil Air Patrol California Wing, where his duties included managing the statewide digital wireless network. He was awarded the 2012 Chuck Yerkes award for his participation on various open source mailing lists.
david@lang.hm

As you build your enterprise logging infrastructure (as discussed in the prior articles in this series [1]), one of the most valuable things that you can do is to have something watch them and generate alerts when things go wrong. There are a lot of tools out there that can be used for this. One good, free tool is Simple Event Correlator (SEC) [2]. In this article, I will provide an introduction to SEC, how to use it, and the capabilities that it provides. In a future article, I will go into detail about how to tune your SEC installation to be able to handle high volumes of logs.

SEC can read log files directly on the local system, but in the context of an enterprise logging infrastructure, this is seldom the right thing to do.

Instead, because you have a consolidated feed of all your logs, you should run one or more instances of SEC on a central analysis farm server where it can see the logs from all your different systems. This allows you to create alerts for things that happen across multiple servers. For example, you don't want to alert on one failed login, but one failed login to each of 400 servers is something that you do want to be alerted about. Additionally, there is a lot of value in keeping your configuration all in one place so that the same rules will be applied across all systems.

Of course, with advantages come disadvantages. The fact that you see the logs from all your systems means that your alerts will fire no matter what environment your systems are in. You probably don't really want a wake-up call at 3 a.m. because a Dev or QA system had a problem, whereas you would want such a call if it was a production system. This is why the enterprise logging architecture defined a way to add metadata to the log messages. Among other benefits, this metadata provides your alerting farm more information than just what is in the log messages when deciding whether it should generate an alert.

The best way to feed log events into SEC is to make sure you are using SEC 2.7.4 or newer and then use the omprog output of rsyslog to have rsyslog start SEC (restarting it, if needed).

With rsyslog 7, this would be done with configuration lines like:

```
Module (load="omprog")
action(type="omprog" binary="/usr/sbin/sec --input=- --initevents
   --notail –conf=/path/to/conf" template="RSYSLOG_TraditionalFileFormat")
```

With older versions, the configuration would be something like:

```
$ModLoad omprog
$ActionOMProgBinary /usr/local/bin/sec.sh
*.* :omprog:
```

and you would have to define the script /usr/local/bin/sec.sh to be something like:

```
#!/bin/sh
/usr/sbin/sec --input=- --initevents --notail –conf=/path/to/conf
```

Understanding the SEC Config File

Sample Rule

SEC configuration consists of multiple rule definitions, along the lines of:

```
type=single
ptype=regexp
pattern= (\S+) sshd\[\d+\]: Accepted.*for (\S+) from (\S+) \
port  (\d+)\s
desc=ssh login to $1 from $3 for user $2
action=write - $2 logged in to $1 from $3 port $4
```

This rule would look for a line like this:

```
Sep 16 17:46:47 spirit sshd[12307]: Accepted password for rik from
                204.176.22.9 port 59926 ssh2
```

And when it is found, would generate an alert to stdout that said:

```
rik logged in from spirit IP 204.176.22.9 port 59926
```

Notes on syntax:

- The order of the keyword=value clauses within a rule does not matter.
- Keywords are case sensitive (unless otherwise specified in the man pages).
- Lines can be continued by ending them with a \.
- Rules are separated by blank lines.
- Comment lines start with #, because comment lines are treated by SEC as if they were blank; comments cannot appear in the middle of a rule but must be between rules.

Many of the values that you are providing are sensitive to case and whitespace. For example, the pattern provided in the code above is looking for a space ahead of the hostname.

Type

There are many different types of matches that SEC has built-in:

SINGLE

> If a match is found, take action immediately.

SUPPRESS

> Ignore anything that matches.

CALENDAR

> Cron type rule to take action at specific times.

SINGLEWITHSUPPRESS

> If a match is found, take action immediately and suppress additional alerts for a time.

PAIR

> Watch for pairs of log entries and take one action when the first entry arrives, and a second if the second entry arrives in time.

PAIRWITHWINDOW

> Watch for pairs of log entries, take one action if the second event arrives in time, and take a different action if it does not. Unlike Pair, no action is taken when the first entry arrives; an action is only taken when the second entry arrives or the timeout hits.

SINGLEWITHTHRESHOLD

> Take action if there are X matches in Y time.

SINGLEWITH2THRESHOLDS

> If there are more than X matches in Y time, take one set of actions, and then wait until there are fewer than X2 actions in Y2 time and take another set of actions—i.e., send a notification when a problem happens (too many messages) and a second notification when it clears up (the problem messages disappear).

EVENTGROUP

> This rule is a generalization of the SingleWithThresh old rule; instead of counting and thresholding one event type, this rule is able to track unlimited number of different events types in a common window (e.g., generate an alarm if ten firewall events and five IDS events have been seen for the same IP address during one minute).

SINGLEWITHSCRIPT

> If a match is found, run a script and take one of two actions depending on whether the script returns success or not.

JUMP

> If a match is found, process one or more other config files against this event.

Ptype

For each rule, you must tell SEC which of the many possible pattern types this rule is using. The available pattern types are:

1. RegExp: Perl regular expression

This pattern type can set variables based on match terms; items enclosed with () in the regexp become $# variables for the rest of the rule. So the sample rule example sets four variables:

```
pattern= (\S+) sshd\[\d+\]: Accepted.*for (\S+) from (\S+) port
(\d+)\s
Sep 16 17:46:47 spirit sshd[12307]: Accepted password for rik from
      204.176.22.9 port 59926 ssh2
```

$1 hostname (spirit),

$2 username (rik),

$3 source IP (204.176.22.9),

$4 port number (59962).

Plus the default $0, which refers to the entire line.

2. SubStr: Substring

Substrings are simple text matches, have no special characters like a regular expression, and don't return any values ($1, $2...). They are much faster to process than a regexp.

3. PerlFunc: Perl function

This executes a Perl function and match if the function returns true and is an extremely powerful capability that I will talk about more in a later article.

This pattern type can also set variables. The Perl snippet can return a list, and the elements of that list become the $# variables.

4. Cached: uses the results of a prior rule match.
5. Tvalue: either matches everything (TRUE) or matches nothing (FALSE).

Cached and Tvalue are normally combined with context conditions, which are described below.

Each of these pattern types will have a negated version (e.g., NregExp, NsubStr, etc.).

Pattern

Most rules require one (or more) pattern lines, and the syntax of the pattern is defined by the ptype defined for the rules.

Desc

Desc fields are critical to understand when configuring SEC. They seem simple (a description of the match), but they play a critical role when doing anything more than a single match.

Proper use of the desc field allows one rule to run many event correlations in parallel and track the state of the correlations independently. Desc defines a "scope" for the correlation state.

When SEC is evaluating any type that has to look at more than one log entry, SEC considers the desc field to be part of the rule. This means that if the desc field evaluates to a different value for the log event, the scope is different and progress towards generating an alert (or suppressing events after an alert has been generated) will be tracked independently of log events that result in the desc field evaluating to a different value.

So if we were to take the sample rule from above and change it to a SingleWithSuppress rule (we don't want alerts every time someone logs in), the rule would become:

```
type=singlewithsuppress
ptype=regexp
pattern= (\S+) sshd\[\d+\]: Accepted.*for (\S+) from (\S+) \
port (\d+)\s
desc=ssh login to $1 from $3 for user $2
action=write - $2 logged in to $1 from $3 port $4
window=60
```

With this rule, we would only get one alert per minute for the same user logging in to one server from another server.

But if we wanted to change this alert so we only got one alert per minute about the user logging in, no matter what server the user logged in to or where the user came from, we could change the desc field to:

```
desc=ssh login for user $2
```

If we wanted to suppress messages only if the user is logging in from the same source, we could change it to:

```
desc=ssh login from $3 for user $2
```

Note that SEC doesn't actually care what this text is, so it would be just as valid as far as SEC is concerned to have the desc field be:

```
desc=$3 $2
```

But it is much nicer to the humans who have to read the file if you make the field more descriptive. SEC combines the desc field with the rule number, so if you have multiple rules that produce the same desc string, SEC will still keep them straight.

Action

An action is what SEC should do when it finds some condition. A single rule can invoke many different actions, semicolon separated. SEC supports many different actions in several different categories. The more important ones to understand include output actions to let you write to a file, a TCP, UDP, or UNIX socket, or execute a script and pass data to stdin on that script.

These commands all have the form:

```
action=<action> <destination> <string>
```

Especially notable are the udgram and spawn actions.

The udgram action lets you send a message to a UNIX socket like /dev/log, which is a great way to have SEC generate feedback into the logging system that can be acted on by other analysis engines. In an enterprise environment, this is also the best way

to generate new events for SEC to process because it will work across multiple instances of SEC, and the event will be visible to all your different analysis farms:

```
action=udgram /dev/log <30>sec-alert: alert text
```

Note that "<30>" is the over-the-wire representation of priority: facility (3 daemon) <<3 + severity (6 info) [3].

The spawn action runs an external program and reads any output from that program as additional log events to analyze.

Context actions let you `create`, `delete`, or redefine (`set`) a context. There are also actions to manipulate and output the list of strings associated with a context (including `add`, `prepend`, `report`, `pop`, `shift`, `copy`).

And, finally, there are actions to set variables. Because it is possible to set a variable to be the output of Perl code, and that Perl code is allowed to have side effects, these actions turn out to be the most powerful.

Additional Important Rule Options

Continue

By default, SEC stops processing a log entry the first time a rule matches that entry. Continue tells SEC whether it should continue processing rules if this rule matches. The default is DontCont, which stops processing rules as soon as one matches the event being processed. By adding a line to the rule that says:

```
continue=takenext
```

SEC will continue processing the rules for the current log entry.

If you wanted to use the different desc examples together—for example, alerting if one user is logging in too many times, or if one machine has too many logins to it—you would need to make sure that the earlier rules all include `continue=takenext` or SEC will never get to the later rules.

Contexts

Contexts (and the desc field described earlier) are the heart of SEC and are what makes it more than simply a fancy regexp engine. Whereas the desc field lets one rule run many event correlation operations simultaneously, and thus act as if it uses many rules, contexts allow you to stitch multiple rules together.

Contexts have four properties:

- Existence—manipulated by the create, delete, obsolete actions
- Defined lifetime—defined at creation or reset by the set action
- Storage—manipulated by the add, . . . actions
- Expiration action—again set during creation or by using the set action and can be used for a number of different things:
- Controlling the actions of other rules by testing to see if a context exists. This allows you to dynamically switch rules on and off by checking for combinations of one or more contexts.
- Storing events and other strings via the add, . . . actions. The stored information can then be reported using the report action.
- Scheduling actions to occur in the future by setting an expire action and a lifetime in seconds.

Contexts are created and manipulated by the action section, but are tested by adding a context= clause to your rule

For example, if you want to alert if you see logs foo, bar, and baz all happen within one minute from the same machine, you could create the rule file:

```
type=single
ptype=regexp
pattern=^.{16}(\S+) .*foo
continue=takenext
action=create foo_$1 60

type=single
ptype=regexp
pattern=^.{16}(\S+) .*bar
continue=takenext
action=create bar_$1 60

type=single
ptype=regexp
pattern=^.{16}(\S+) .*baz
continue=takenext
action=create baz_$1 60

type=single
ptype=regexp
pattern=^.{16}(\S+)
context=foo_$1 && bar_$1 && baz_$1
continue=takenext
action=write - warning foo bar baz on $1; \
  delete foo_$1; delete bar_$1; delete baz_$1
```

Note that this set of tests works even if the logs arrive in a different order than you expected.

Executing Perl code as part of the context test is also possible. When combined with cached pattern types, this allows for specific and fast rule evaluation.

Contexts allow you to combine multiple rules in one alerting decision. You can alert only if several different conditions are true by having one rule for each condition you are interested in (each one setting a context), and then another rule to detect that all of the other criteria have been met.

The most common use of Contexts is to set a flag (with a timeout) so that other rules can know that a particular condition has taken place.

Another use for Contexts is to alert when something stops happening. For example, if you have your systems running "vmstat 60 |logger -t vmstat", they will log a vmstat output line every minute. You can then use a rule similar to:

```
type=single
ptype=regexp
pattern= (\S+) vmstat:
desc=vmstat_$1
action=create vmstat_heartbeat_$1 180 ( shellcmd notify.sh $1 )
```

to generate a notification whenever a system ($1) stops generating a log message. It does this by creating a context that will expire in three minutes, and if the context expires, it sends a notification. If another vmstat message arrives from that system, SEC resets the context to expire three minutes from when that message arrived.

The ability to associate a list of strings with a context allows you to create a context when you see the first event that makes you suspicious, add all log events as strings to the context, so that when the context expires (or some other condition happens), you can make all of the logs that occurred during this period be part of the alert that you send out.

Internal Events

When started with –initevents (as in the example of how to start SEC from rsyslog), SEC generates internal events as it is running; this allows you to create actions that only take place once when SEC is started, restarted, shutdown, etc. For example, if you want a log entry every time that SEC is started or restarted, you could use a rule like:

```
type=single
ptype=regexp
pattern=(SEC_STARTUP|SEC_RESTART)
context=SEC_INTERNAL_EVENT
desc=Init counters with 0
action=udgram /dev/log <30> sec-status: SEC initialized
```

Using Perl in SEC

The ability to use snippets of Perl in your SEC rules is one of the things that makes SEC so incredibly powerful. SEC runs your Perl snippets in a different namespace than SEC itself, so your Perl snippets are not going to conflict or interfere with the SEC internals, although it is possible to get access to the SEC internal variables if you really need to.

As an example of the capabilities that this provides, you could extend the sample rule above to produce daily ssh login reports by changing the action in the sample rule above to:

```
action=write - $2 logged in to $1 from $3 port $4; \
      eval %o ($ssh_summary{user}{$2}{count}++; \
      $ssh_summary{total_sessions}++; )
```

SEC doesn't actually have a command only to execute Perl code, but it has actions that allow you to run any Perl code and put the output of that code in a variable. In this case we put the output of the Perl code into the variable %o, but we never use it. The exec action compiles the code each time; there is a similar action lcall that compiles the code once at startup. This is faster, and it can avoid the need to escape Perl variables.

Add a rule to initialize the variables at startup (and restart).

```
type=single
ptype=regexp
pattern=(SEC_STARTUP|SEC_RESTART)
context=SEC_INTERNAL_EVENT
desc=Init counters with 0
action=lcall %o ->( sub { %ssh_summary=(); } )
# note that if exec was used instead of lcall,
# the prior line would need to escape the % and would be:
# action=exec %o ( %%ssh_summary=(); )
```

Then add a rule to output the stats daily and reset them.

```
type=calendar
time=0 0 * * *
desc=output daily stats
action=lcall %o -> ( sub { $ssh_summary{total_sessions}; } ); \
  udgram /dev/log <30>sec-summary: There were %o ssh sessions
today; \
  lcall %n -> ( sub { my($ret); \
        foreach (keys %{$ssh_summary{user}}) { \
           $ret .= "$_ = $ssh_summary{user}{$_}{count} "; } \
        $ssh_summary{total_sessions} = 0; return $ret; } ); \
  if %n ( udgram /dev/log <30>sec-summary: Number of SSH \
sessions for each user: %n )
```

Another good use of Perl is to load a table at startup, and then test it during the rules.

For example, if you create a file that contains a list of your production server names, and then create a startup rule like:

```
type=single
ptype=regexp
pattern=(SEC_STARTUP|SEC_RESTART)
context=SEC_INTERNAL_EVENT
desc=Load Production Server Table
action=eval %o (%%prodservers=();open(infile, \
"</etc/prodservers.txt"); \
```

```
   while <infile> {chomp; $prodservers{$_}=1; }; close(infile))
```

you can then add a context test to our original example rule to only alert if the log is from a production server.

```
context= =(exists $prodservers{$1})
```

Similar to the exec command, this compiles the code on each run (and requires escaping % characters). There is the equivalent to lcall that would look like:

```
context= $1 -> ( sub { exists $prodservers{$_[0]} } )
```

You can have SEC reload the table on demand by adding a rule like:

```
type=single
ptype=regexp
pattern=SEC reload production server table
desc=Reload Production Server Table
action=eval %o (%%prodservers=();open(infile, \
"</etc/prodservers.txt"); \
   while <infile> {chomp; $prodservers{$_}=1; }; close(infile))
```

Note that you probably want to have additional restrictions so that the reload can only be generated by logs from a specific set of servers.

Caching Match Results

When you have a number of tests that you want to do with a single log event, doing the same regexp repeatedly is inefficient.

Using our example, let's say you want to do multiple alerts on ssh logins. Instead of each of the rules needing to rerun the regexp against the log line, you could add the following line to the original example rule:

```
varmap=ssh; line=0; server=1; user=2;source =3; port=4
```

Then you could create a second rule such as:

```
type=singlewiththreshold
ptype=cached
pattern=ssh
desc=lots of logins for user $+{user}
action=write - $+{user} logged in to more than 5 servers in one
minute
window=60
thresh=5
continue=takenext
```

With the use of Perl in the context, you could configure this to only fire if the user has logged in to more than ten servers all day (to avoid getting alerts when users arrive in the morning and log in to a bunch of places to start their day) by adding a line:

```
context=ssh :> (sub { \
   return $ssh_summary{user}{$_[0]->{user}}{count} > 10} )
```

Debugging

Debugging alerting systems is always an interesting exercise. You need to be able to generate events to trigger the rules, but when they don't fire as expected, you need to be able to figure out what the internal state of your alerting engine is. SEC provides this option by way of dump files. If you start SEC with the option `--dump=/path/to/dumpfile`, you can send it the signal USR1, and if the dump file does not already exist, SEC will dump its internal state. This includes performance stats, how many matches there have been for each rule, the last several lines that it has processed, and information about every context that it is tracking.

Another approach to debugging is to run SEC from the command line with it reading from stdin or a file. SEC generates a lot of output as it processes each message, telling you what it does each step of the way; however, the types of problems that are the hardest to troubleshoot tend to involve timing, which means that you can't just use a test file. The timing in SEC is based on when SEC sees the log entry, not any timestamp that may appear in the log entry. You are better off generating the input to SEC with a script so that you have a repeatable test that generates the correct messages with the right timing, either echo+sleep or logger+sleep if you want to test any filtering in rsyslog as well.

Conclusion

This is a brief overview of the capabilities of SEC, and there are a lot of nuances and other capabilities that I did not go into. With the different test types, contexts, desc fields, alerting scripts, and embedded Perl snippets, there is little that SEC cannot do.

SEC does take some training and expertise to master and configure for your environment, but any serious alerting engine that you use is going to require customization to your needs. The biggest problem with SEC is that there is not a good pool of examples available for people to work from, but the users mailing list [4] is extremely responsive to requests.

References

[1] David Lang, Enterprise Logging: https://www.usenix.org/publications/login/august-2013-volume-38-number-4/enterprise-logging and https://www.usenix.org/publications/login/october-2013-volume-38-number-5/log-filtering-rsyslog.

[2] http://simple-evcorr.sourceforge.net/.

[3] http://en.wikipedia.org/wiki/Syslog.

[4] https://lists.sourceforge.net/lists/listinfo/simple-evcorr-users/.