Simple Event Correlator Tutorial

Risto Vaarandi

December 4, 2022

Abstract

Simple Event Correlator (SEC) is a lightweight and platform-independent UNIX tool that is designed for tackling a wide range of event correlation and other event processing tasks. This tutorial complements the official SEC documentation (SEC man page) and provides a gentle introduction to SEC. The tutorial first discusses how to run SEC and its most commonly used command line options. The tutorial then provides an introduction into SEC rules, event correlation operations, contexts, and synthetic events. Finally, the tutorial covers some advanced topics such as building hierarchical rulebases and using custom code in SEC rules.

Contents

1 Introduction

1.1 Introduction to SEC

Simple Event Correlator (SEC) is an event correlation tool which can be harnessed for event log monitoring, for network and security management, for fraud detection, and for any other task which involves event correlation. In research literature, *event correlation* is defined as a procedure where a stream of events is processed in order to detect (and act on) certain event groups that occur within predefined time windows [1].

Many traditional event log management systems store events in a database and execute database queries for implementing event correlation. However, such systems are heavyweight solutions and often involve a complex database infrastructure on dedicated hardware.

In contrast, SEC is a lightweight and platform-independent event correlator that is implemented as a UNIX tool and runs as a single process. For facilitating deterministic event processing, SEC is single-threaded. However, since it has a small memory footprint, it is straightforward to run several SEC processes with independent rulebases on the same system.

The user can start SEC as a daemon, employ it in shell pipelines, execute it interactively in a terminal, run many SEC processes simultaneously for different tasks, and use it in a wide variety of other ways. Note that unlike many traditional event correlation solutions, SEC implements *database-less event correlation* – it does not rely on any database technology and query-based event batch processing, but correlates incoming event streams in real-time with the help of memory-based data structures.

SEC reads lines from files, named pipes, or standard input, matches the lines with patterns (like regular expressions or Perl subroutines) for recognizing input events, and correlates events according to the rules in its configuration file(s). SEC can produce output by executing external programs (e.g., *snmptrap* or *mail*), by writing to files, by sending data to TCP and UDP based servers, by calling precompiled Perl subroutines, etc.

SEC is written in Perl and has been publicly available since March 23, 2001 under the terms of GNU GPL version 2.

1.2 The purpose of this tutorial

When SEC was released in 2001, it started as a relatively simple tool, but over time its complexity has grown, and so has the volume of its official documentation. To provide a compact introduction to SEC capabilities, several papers have been published over the last two decades [2, 3, 4, 5, 6, 7]. However, since SEC is constantly evolving, the past papers are not covering the newer features introduced after the papers were published.

The purpose of this tutorial is to address this issue, and be a live document that is updated after new major features have emerged. Note that this tutorial is *not* designed to be a replacement for the official documentation (SEC man page), but rather complement it by providing a gentle introduction to SEC.

1.3 Prerequisites

Reading this tutorial assumes familiarity with regular expressions, UNIX shells, and commonly used UNIX tools. The examples in this tutorial assume the use of SEC version 2.9.1 or higher.

2 Getting started

2.1 An example of a simple configuration file

SEC event correlation rules are stored in one or more configuration files (sometimes also called rule files). As an example of a simple configuration file that contains only one rule, consider the file *echo.sec* provided by Listing 1.

Listing 1: The content of echo.sec

```
# a simple rule example

type=Single
rem=the pattern provided below matches \
    any non-empty input line, assigning \
    the entire line to the $0 match variable
ptype=RegExp
pattern=.
desc=echo input line
action=write - %t: $0

# end of the rule definition
```

As can be seen from Listing 1, the rule definition consists of keyword-value pairs provided on separate lines. Each line that starts with the # character is treated as a comment line. Also, any comment line or empty line acts as a delimiter between rule definitions (that means you can't insert such lines in between keyword-value lines in the rule definition). In order to insert a comment into the rule definition, the *rem* keyword has to be used (see Listing 1).

As mentioned before, each keyword-value pair that is a part of the rule definition must reside on a separate line. If the value for the keyword is too long for one line, the value can be continued on the next line, ending the current line with \(backslash character). For example, see the value of the *rem* keyword in Listing 1.

The example configuration file from Listing 1 contains one rule of type *Single*. The *Single* rule is the simplest of all SEC rules which matches an input event with the pattern defined with the *ptype* and *pattern* keywords, and on successful match immediately executes the action list defined with the *action* keyword.

The *Single* rule from Listing 1 uses a regular expression pattern for matching input lines. The pattern type is defined with the *ptype* keyword – for example, *RegExp* denotes a regular expression for matching a 1-line input event, *RegExp2* denotes a regular expression pattern for matching a 2-line input event, etc.

The regular expression pattern is provided with the *pattern* keyword. Note that the pattern from Listing 1 (.) matches any non-empty input line. Also, note that capture groups of regular expression patterns set match variables,

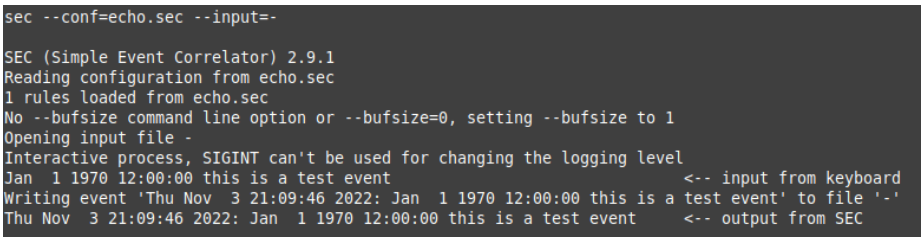


Figure 1: Running SEC in a terminal window

with the $0 variable being set to the entire matching event ($0 is set even if there are no capture groups in the expression).

The *action* keyword of the rule speficies an action list containing one *write* action. This action opens the file with a name provided by the first parameter (if not already open), and writes the string that follows the filename parameter to the given file, terminating the string with the newline character (ASCII 10). Since the special filename - (dash character) denotes standard output, the *write* action from Listing 1 sends the string *this is my message*<*NEWLINE*> to standard output.

Finally, the *desc* keyword defines the *operation description string* of the SEC event correlation rule. The value of this keyword determines how many event correlation operations a rule can start and what is the scope of each operation. However, since the *Single* rule takes an *immediate* action when a matching event has been observed, it doesn't start any event correlation operations. We will return to a more detailed discussion on the importance of the *desc* keyword in Section 3.1.

2.2 Running SEC interactively

Probably the simplest way of using SEC is to run it interactively in a terminal window, so that lines entered from the keyboard would be processed, using the rules from some configuration file. For example, for using the configuration file from Listing 1 in such a way, SEC can be started with the following command line (note that specifying - for the input file with `--input=-` denotes reading from standard input):

```
sec --conf=echo.sec --input=-
```

Suppose SEC is started in such a way and the user types in the following line at 21:09:46 on November 3, 2022:

```
Jan  1 1970 12:00:00 this is a test event
```

The *Single* rule from Listing 1 matches this event, and the *write - %t: $0* action from this rule writes the following line to standard output:

```
Thu Nov  3 21:09:46 2022: Jan  1 1970 12:00:00 this is a test event
```

In addition, SEC prints out messages about its work to standard error (see Figure 1).

The %t variable is an *action list variable* that represents the current time in human-readable format. In the case of the above example, the value of %t is *Thu Nov 3 21:09:46 2022* which appears in the beginning of the output string from the *write* action. After a separating colon and space, the value of the $0 match variable appears which is *Jan 1 1970 12:00:00 this is a test event.*

Although match variables and action list variables look similar, there are several important differences between these two variable classes which are summarized below:

- Match variables can be used in many rule fields, while action list variables are available *in action lists only.*
- After a rule has matched an event, the values of match variables are accessible in the current rule while the event is being processed (as discussed in Section 4.4, pattern match caching allows for making match variables available to other rules). However, after the event processing is complete, the values of match variables are no longer available. In contrast, action list variables have a global scope and most of them retain their values until explicitly set to another value (excluding builtin action list variables like %t that are automatically adjusted).
- Match variables are substituted with their values immediately after the pattern match, while action list variables are substituted at the moment of action list execution. Although in the case of the *Single* rule from Listing 1 the regular expression match is immediately followed by the execution of the *write* action, other rule types that will be discussed in Section 3 might introduce a significant time delay between the action list execution and the pattern match that sets the match variables. In Section 3.2, a relevant rule example is provided that illustrates the variable substitution process.

There is another observation we can make from the output in Figure 1. Although the input line had a timestamp from the past (Jan 1 1970 12:00:00), SEC did not consider it as the occurrence time of the event and did not set the %t variable to *Thu Jan 1 12:00:00 1970.*

The above observation reflects an important property of SEC input event handling model – *the occurrence time of the input event is the time when SEC observes the event according to local system clock.*

In other words, SEC does not implement its own internal clock which is driven by timestamps possibly found in input events, but rather queries the system clock with *time(2)* system call as input events arrive. Note that *time(2)* is also used for all other time measurements by SEC (*time(2)* returns the current time as seconds since Jan 1 00:00:00 1970 UTC).

2.3 Running SEC as a daemon

For running SEC as a daemon, one needs to provide the `--detach` command line option for SEC. However, using this option introduces the need for replacing relative path names with absolute names on command line and in rule definitions.

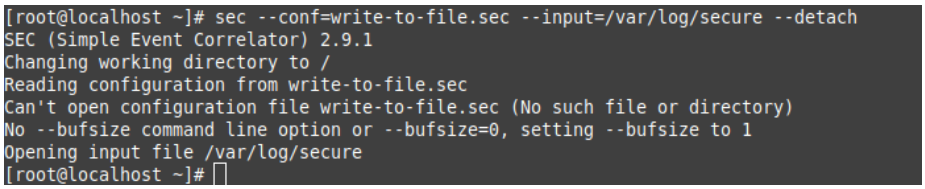


Figure 2: Failure to find a configuration file

Figure 2 depicts a scenario of running SEC as a daemon, so that the configuration file name is provided with a relative path. The process of becoming a UNIX daemon includes changing the working directory to the root directory / [8], and Figure 2 includes a relevant message. However, the configuration file *write-to-file.sec* is not found in that directory, and this triggers an error message.

In order to fix that issue, command line option like the following is needed:

`--conf=/etc/sec/write-to-file.sec`

Also, Listing 2 provides the content of *write-to-file.sec* where the file name for the *write* action has been provided with an absolute path (*/var/log/echo.log*).

Listing 2: The content of /etc/sec/write-to-file.sec

```
type=Single
ptype=RegExp
pattern=.
desc=write input line to file
action=write /var/log/echo.log %t: $0
```

When running SEC as a daemon, its debug and error messages that were previously appearing in a terminal (see Figures 1 and 2) are no longer available, but collecting these messages is important for troubleshooting purposes. For configuring the logging of such messages into a separate file, the `--log` command line option can be used, whereas the `--debug` option sets the logging verbosity level (the highest level is the default). Also, it is often worthwhile to store the process ID of the SEC daemon process into a dedicated file with the `--pid` option (for example, that eases the log rotation).

Here is an example command line for starting SEC as a daemon:

```
sec --conf=/etc/sec/write-to-file.sec --input=/var/log/secure
    --detach --log=/var/log/sec.log --pid=/run/sec.pid
```

With the above options, SEC logs its own messages to */var/log/sec.log*. Also, it uses the *write* action from Listing 2 to append messages to */var/log/echo.log*. If at some point these two SEC output files are rotated, SEC needs to be informed about the need to reopen these files, and that can be done by sending the USR2 signal to the SEC process:

```
kill -USR2 `cat /run/sec.pid`
```

Note that SEC is able to detect and handle input file rotations automatically. Therefore, when the input file */var/log/secure* is rotated, SEC will switch over to the new instance of */var/log/secure* without needing a signal.

Also, you can configure SEC to monitor any number of input files, for example:

```
sec --conf=/etc/sec/write-to-file.sec
    --input=/var/log/secure --input=/var/log/messages
    --input=/var/log/maillog --input=/var/log/cron
    --detach --log=/var/log/sec.log --pid=/run/sec.pid
```

When you update a configuration file, one option is to simply shut down SEC and start it again, but that implies the loss of all event correlation state stored in memory. For preventing that, you can send the ABRT signal to the SEC process which triggers the so called *soft restart*, where SEC reloads the modified configuration file and tries to preserve as much event correlation state as possible:

```
kill -ABRT `cat /run/sec.pid`
```

If you want to change SEC command line options without restarting it, you can store your command line options in a resource file, and provide the resource file path to SEC via the *SECRC* environment variable.

While the SEC log file provides a lot of information about SEC activities and the event correlation process, it is often wortwhile to query its event correlation state, performance data, and other information. For doing that, send the USR1 signal to the SEC process:

```
kill -USR1 `cat /run/sec.pid`
```

On the reception of that signal, a dump file will be created which holds detailed information about the event correlation state and other useful data. Default dump file path is */tmp/sec.dump* and it can be adjusted with the `--dump` command line option.

2.4 Rule application order

So far, we have discussed simple scenarios involving one configuration file with one rule. However, using many configuration files with a larger number of rules raises the following question – what is the rule application order when input events are processed?

To answer that question, consider a scenario where SEC has been started with the following command line:

```
sec --conf=ltr1.sec --conf=ltr2.sec --input=-
```

In that case, the rules from file *ltr1.sec* are applied first, followed by the application of rules from file *ltr2.sec*. If configuration files have been provided with a wildcard pattern (e.g., **.sec*), matching files are considered in the order determined by system locale.

The rules from one configuration file are applied in the same order as they appear in this file:

- if a rule does not match the input event, the next rule is tried,

- if a rule matches the input event, the value of rule's *continue* keyword determines what rule is tried next. By default, the following rules in the same configuration file are not tried.

For example, consider the configuration files *ltr1.sec* and *ltr2.sec* from Listings 3 and 4 that will be used in the remainder of this section for explaining the rule application order.

Listing 3: The content of ltr1.sec

```
type=Single
ptype=SubStr
pattern=AAA
rem=after this rule has matched, don't consider following \
    rules in this configuration file, since 'continue' \
    defaults to 'DontCont'
desc=Three A characters
action=write - three A characters were observed

type=Single
ptype=SubStr
pattern=BBB
rem=after this rule has matched, don't consider any other \
    rules (in other words, search for matching rules ends \
    for all configuration files)
continue=EndMatch
desc=Three B characters
action=write - three B characters were observed
```

The rules in Listings 3 and 4 feature patterns of type *SubStr*. The *SubStr* pattern is a substring that is searched in the input event, and the pattern matches if and only if the substring is found. Note that for the sake of matching speed, the *SubStr* pattern does *not* set any match variables.

For example, suppose the user provides the following input line:

```
AAABBBCCCDDDEEE
```

As mentioned above, SEC first applies rules from *ltr1.sec* and then rules from *ltr2.sec*. The first rule in *ltr1.sec* matches, and due to the default setting of *continue* keyword (i.e., *continue=DontCont*) the following rules are not considered in *ltr1.sec*. Also, the first rule in *ltr2.sec* matches, and *continue=GoTo lastRule* setting directs the further processing to the location defined by the *lastRule* label in *ltr2.sec* (i.e., the third rule in *ltr2.sec*). The third rule also matches, and all matching rules in *ltr1.sec* and *ltr2.sec* produce the following output:

```
three A characters were observed
three C characters were observed
three E characters were observed
```

Suppose the user provides the following input line:

```
BBBCCCDDDEEE
```

Listing 4: The content of ltr2.sec

```
type=Single
ptype=SubStr
pattern=CCC
rem=after this rule has matched, continue from last rule \
    in this configuration file
continue=GoTo lastRule
desc=Three C characters
action=write - three C characters were observed

type=Single
ptype=SubStr
pattern=DDD
rem=after this rule has matched, continue from next rule \
    in this configuration file
continue=TakeNext
desc=Three D characters
action=write - three D characters were observed

label=lastRule

type=Single
ptype=SubStr
pattern=EEE
desc=Three E characters
action=write - three E characters were observed
```

In this case, the first rule in *ltr1.sec* does not match, and therefore the following rule is tried. The second rule in *ltr1.sec* matches, and *continue=EndMatch* setting terminates all further search for matching rules (i.e., rules from *ltr2.sec* will not be tried). As a result, the following output is produced:

```
three B characters were observed
```

Finally, suppose the user provides the following input line:

```
DDDEEE
```

In this case, both the first and second rule in *ltr1.sec* are tried but do not match. The processing continues with rules from *ltr2.sec*, with the first rule not producing the match. However, the second rule in *ltr2.sec* matches, and due to *continue=TakeNext* setting the processing will continue with the next rule. The third rule in *ltr2.sec* also matches, and the following output is produced:

```
three D characters were observed
three E characters were observed
```

Suppose the user changes the order of `--conf` options on the example SEC command line used in this section, for example:

```
sec --conf=ltr2.sec --conf=ltr1.sec --input=-
```

Obviously, that will also change the application order of rules (rules from *ltr2.sec* would be applied first now). For establishing a rule application order that is independent from the order of command line options, hierarchically organized rulesets can be used that will be discussed in Section 6.1.

2.5 Commonly used SEC actions

We have already discussed the *write* action, and this section provides an overview of other commonly used actions that are employed by rule examples in this tutorial.

The *shellcmd* action forks a process for running a user-defined command line. If the command line contains shell metacharacters, it is parsed by shell. Listing 5 provides a rule example that matches SSH login failures, setting the $1 match variable to the IP address of the SSH client host.

Listing 5: An action list with write and shellcmd actions

```
# An example event matched by this rule:
# sshd[2181]: Failed password for bob from 10.1.1.1 port 55529 ssh2

type=Single
ptype=RegExp
pattern=sshd\[\d+\]: Failed .+ for \S+ from ([\d.]+) port \d+ ssh2
desc=login failure
action=write /var/log/ssh-failures.log SSH login failure from $1; \
       shellcmd echo 'SSH login failure from $1' | mail root@localhost
```

The *action* keyword of the rule defines an action list consisting of two actions – *write* and *shellcmd*. If the action list contains more than one action, the action definitions must be separated by semicolons.

The *write* action from Listing 5 writes the string *SSH login failure from* ⟨*ipaddress*⟩ to */var/log/ssh-failures.log*, whereas the *shellcmd* action forks a process for executing the following command line:

```
echo 'SSH login failure from <ipaddress>' | mail root@localhost
```

This command line sends the string *SSH login failure from* ⟨*ipaddress*⟩ to *root@localhost* via email. Also, since this command line contains the shell metacharacter |, the command line is parsed by shell.

Note that whenever SEC forks a process for executing a command line, the execution is *asynchronous* – SEC does not wait for the command to complete, but continues immediately after the new process has been forked.

The asynchronous nature of command line execution might create unexpected issues when multiple *shellcmd* actions are used in the same action list. For example, consider the following action list:

```
shellcmd cat /tmp/report | mail root; shellcmd rm -f /tmp/report
```

Due to asynchronous execution, the *rm* command might remove the file */tmp/report* before the *cat* command has a chance to open it. For addressing this issue, commands can be arranged into a shell list, making sure that the shell is not executing the second element of the list before the first element completes:

```
cat /tmp/report | mail root; rm -f /tmp/report
```

However, the use of the above shell list introduces another issue – in the list, commands are separated by a semicolon which is also used for separating actions in SEC rule definitions. For indicating that the semicolon of the shell

Listing 6: A shellcmd action with a shell list

```
type=Single
ptype=SubStr
pattern=SEND REPORTS
desc=send reports
action=shellcmd (cat /tmp/report | mail root; rm -f /tmp/report); \
       write - report sent!
```

list is part of the command line, the entire command line must be enclosed in parentheses, and Listing 6 provides a relevant example.

Apart from the *shellcmd* action, another commonly used action for executing command lines is *pipe*. Listing 7 illustrates a more efficient implementation of the rule from Listing 5 that employs the *pipe* action.

Listing 7: An action list with write and pipe actions

```
type=Single
ptype=RegExp
pattern=sshd\[\d+\]: Failed .+ for \S+ from ([\d.]+) port \d+ ssh2
desc=login failure
action=write /var/log/ssh-failures.log SSH login failure from $1; \
       pipe 'SSH login failure from $1' mail root@localhost
```

The *pipe* action forks a process for asynchronous execution of a command line, writing a user-defined string to the standard input of the executed command. By convention, the user-defined string must appear between apostrophes in order to disambiguate it from the following command line. As with the *shellcmd* action, the command line is parsed by shell if it contains shell metacharacters.

In some cases (most notably for security reasons), it is necessary to disable shell parsing for the executed command lines. For that purpose, SEC provides *cmdexec* and *pipeexec* actions that are similar to *shellcmd* and *pipe* actions respectively, but they execute command lines by calling *execvp(3)* without any shell parsing.

The *logonly* action writes a user-defined message to SEC log file, with the level of the message being 4 (that level corresponds to informational messages). For example, suppose SEC was started with the following commandline:

```
sec --conf=/etc/sec/write-to-file.sec --input=/var/log/secure
    --detach --log=/var/log/sec.log --pid=/run/sec.pid
```

In the case of the above command line, the following action writes the message *This is a test message* to SEC log file */var/log/sec.log*:

```
logonly This is a test message
```

Finally, the *none* action denotes no-op.

3 Event correlation

This section provides an introduction to event correlation with SEC, discussing some commonly used rule types for that purpose – the *SingleWithThreshold*,

EventGroup, and *PairWithWindow* rule.

In the remainder of this tutorial, we assume that the timestamps of example events represent the time SEC observes these events. For example, consider SSH login failure events in Listing 8 that originate from */var/log/secure*. It is assumed that when SEC is receiving input events from */var/log/secure*, it observes 5 events from Listing 8 on October 17 at 12:00:01, 12:01:09, 12:02:16, 12:03:43, and 12:04:56 according to local system clock.

Listing 8: Example sshd login failure events from /var/log/secure

```
Oct 17 12:00:01 host2 sshd[2181]: Failed password for bob from 10.1.1.1 port 55529 ssh2
Oct 17 12:01:09 host2 sshd[2183]: Failed password for jim from 10.6.1.9 port 55530 ssh2
Oct 17 12:02:16 host2 sshd[2181]: Failed password for bob from 10.1.1.1 port 55534 ssh2
Oct 17 12:03:43 host2 sshd[2187]: Failed password for jim from 10.1.1.1 port 55538 ssh2
Oct 17 12:04:56 host2 sshd[2189]: Failed password for bob from 10.1.1.1 port 55543 ssh2
```

3.1 SingleWithThreshold rule and introduction to event correlation operations

Suppose we have to address the following event correlation problem – when 3 SSH login failure events (like the events from Listing 8) are observed during 300 seconds (5 minutes), a warning email must be sent to the system administrator. This raises the following question: how exactly should the event counting be implemented?

For example, if only one event counter is maintained for *all* SSH login failures and SEC observes the events from Listing 8, the appearance of the third event at 12:02:16 should trigger the warning email.

On the other hand, if separate event counters are maintained for SSH client hosts, the third event for IP address 10.1.1.1 at 12:03:43 should trigger a warning email with a relevant message (e.g., *Too many SSH login failures from 10.1.1.1*).

Finally, if a separate event counter is maintained for each combination of user name and SSH client IP address, the events from Listing 8 contain three such combinations:

- ⟨bob, 10.1.1.1⟩ – observed 3 times at 12:00:01, 12:02:16, and 12:04:56
- ⟨jim, 10.6.1.9⟩ – observed once at 12:01:09
- ⟨jim, 10.1.1.1⟩ – observed once at 12:03:43

Therefore, the third event for the combination ⟨bob, 10.1.1.1⟩ at 12:04:56 should trigger a warning email with a relevant message (e.g., *Too many SSH login failures for user bob from 10.1.1.1*).

Listing 9 provides an example *SingleWithThreshold* rule that implements the event correlation scheme described above, maintaining a separate event counter for each user name and IP address combination.

The *thresh* and *window* keywords of the *SingleWithThreshold* rule define the counting threshold and the size of the event correlation window in seconds. The *desc* keyword defines the *operation description string* that determines what *event correlation operations* are started by the rule and what events are processed by these operations.

Listing 9: The content of /etc/sec/sshd.sec

```
type=SingleWithThreshold
ptype=RegExp
pattern=sshd\[\d+\]: Failed .+ for (\S+) from ([\d.]+) port \d+ ssh2
desc=user $1 ip $2
action=pipe 'Too many SSH login failures for user $1 from $2' \
           mail root@localhost
thresh=3
window=300
```

Whenever an event is matched by the rule that defines the correlation of several events over time, the event is processed as follows:

- after the event has matched the rule, the value of the *desc* keyword (*operation description string*) is found,
- event correlation operation ID is calculated that is a tuple ⟨*configuration file name, rule number in the configuration file, operation description string*⟩. Note that the first rule in the configuration file has the number 0, the second rule has the number 1, etc. If there is no operation with the given ID, the operation is started for this ID,
- the event is handed over for the event correlation operation with the calculated ID for further processing.

For example, suppose the rule from Listing 9 processes the events from Listing 8. The first event at 12:00:01 triggers the creation of the event counting operation with the ID ⟨/etc/sec/sshd.sec, 0, user bob ip 10.1.1.1⟩. Also, the first event is handed over to the operation for processing, and as a result, the operation sets its event counter to 1.

The events at 12:02:16 and 12:04:56 are also processed by this operation, incrementing the event counter by 1 in both cases. Since the counter becomes equal to 3 after the last event at 12:04:56 has been processed, the operation executes the *pipe* action, sending the warning message *Too many SSH login failures for user bob from 10.1.1.1* to *root@localhost*.

Also, the events at 12:01:09 and 12:03:43 trigger the creation of event counting operations with the ID ⟨/etc/sec/sshd.sec, 0, user jim ip 10.6.1.9⟩ and ⟨/etc/sec/sshd.sec, 0, user jim ip 10.1.1.1⟩. Since both of those operations will only process one event, their event counters will remain set to 1.

This example illustrates several key points about event correlation rules and event correlation operations:

- one rule can start many event correlation operations that run simultaneously, whereas each operation has only one parent rule,
- the event correlation operation inherits its type from the parent rule (e.g., *SingleWithThreshold* rule starts *SingleWithThreshold* operations),
- the value of the rule's *desc* keyword determines the number of event correlation operations started by the rule and how the rule divides matching events between operations,

```
List of event correlation operations:
============================================================
Key: /etc/sec/sshd.sec | 0 | user root ip 192.168.56.1
Operation started at: Tue Nov 22 19:31:49 2022
Correlation window begins at: Tue Nov 22 19:31:49 2022
Correlation window ends at: Tue Nov 22 19:36:49 2022
Configuration file: /etc/sec/sshd.sec
Rule number: 1
Rule internal ID: 0
Type: SingleWithThreshold
Pattern: regexp for 1 line(s): (?^:sshd\[\d+\]: Failed .+ for (\S+) from ([\d.]+) port \d+ ssh2)
Context:
Behavior after match: don't continue
Description: user root ip 192.168.56.1
Action: pipe Too many SSH login failures for user root from 192.168.56.1 mail root@localhost;
Action2:
Window: 300 seconds
Threshold: 3
2 events observed at (checking for threshold):
Tue Nov 22 19:31:49 2022
Tue Nov 22 19:31:54 2022

------------------------------------------------------------
Key: /etc/sec/sshd.sec | 0 | user student ip 192.168.56.1
Operation started at: Tue Nov 22 19:31:28 2022
Correlation window begins at: Tue Nov 22 19:31:28 2022
Correlation window ends at: Tue Nov 22 19:36:28 2022
Configuration file: /etc/sec/sshd.sec
Rule number: 1
Rule internal ID: 0
Type: SingleWithThreshold
Pattern: regexp for 1 line(s): (?^:sshd\[\d+\]: Failed .+ for (\S+) from ([\d.]+) port \d+ ssh2)
Context:
Behavior after match: don't continue
Description: user student ip 192.168.56.1
Action: pipe Too many SSH login failures for user student from 192.168.56.1 mail root@localhost;
Action2:
Window: 300 seconds
Threshold: 3
3 events observed at (seen before threshold was crossed):
Tue Nov 22 19:31:28 2022
Tue Nov 22 19:31:29 2022
Tue Nov 22 19:31:39 2022

------------------------------------------------------------
Total: 2 elements
```

Figure 3: Example event correlation operations from SEC dump file

- the presence of the configuration file name and rule number in the ID of the event correlation operation ensures that operations started by different rules can never have overlapping IDs,
- rules are static entities defined in configuration files, whereas event correlation operations are dynamic entities that exist in memory. As discussed in Section 2.3, a dump file with event correlation state is generated when the USR1 signal is sent to SEC process. Among other data, this dump file contains detailed information about the state of all currently running event correlation operations. For example, Figure 3 displays information about simultaneously running event correlation operations started by the rule from Listing 9.

3.2 Variable substitution for event correlation operations

Please note that as a general rule, match variables are substituted with their values when the event correlation operation is initialized, and these values are used throughout the lifetime of the operation (exceptions from that rule are described in SEC official documentation). For example, consider the rule from Listing 10.

Listing 10: The content of /etc/sec/sshd2.sec

```
type=SingleWithThreshold
ptype=RegExp
pattern=^(.{15}) \S+ sshd\[\d+\]: Failed .+ for (\S+) from ([\d.]+) port \d+ ssh2
desc=user $2 ip $3
action=pipe '%t: Too many SSH login failures for user $2 from $3 (first event $1)' \
            mail root@localhost
thresh=3
window=300
```

Note that the regular expression pattern of this rule sets the $1 match variable to the 15-character timestamp from the matching event. Although the events processed by the same operation share the same user name and SSH client IP address, such events can have different timestamps.

Since match variables are substituted when the operation is created, the $1 variable will get its value from the *first* event that is processed by the operation. In contrast, action list variables are substituted with their values *when the action list is executed*. The builtin %t action list variable in Listing 10 will thus reflect the time when the *pipe* action is executed.

Therefore, when the rule from Listing 10 processes the events from Listing 8, the event correlation operation with the ID ⟨/etc/sec/sshd2.sec, 0, user bob ip 10.1.1.1⟩ sends the following email message to *root@localhost* at 12:04:56:

Mon Oct 17 12:04:56 2022: Too many SSH login failures for user bob from 10.1.1.1 (first event Oct 17 12:00:01)

3.3 SingleWithThreshold operations and sliding window based event correlation

In order to understand the working principle of *SingleWithThreshold* event correlation operations, let's consider the rule from Listing 9 and 8 example events from Figure 4. When these 8 events are processed by the rule, the rule starts an event correlation operation with the ID ⟨/etc/sec/sshd.sec, 0, user bob ip 10.1.1.1⟩, and all events are processed by this operation.

The first event at 12:39:01 starts the operation, setting the beginning of the 300 second (5 minute) event correlation window to 12:39:01. In other words, the event correlation windows covers the time frame 12:39:01–12:44:01 (see Figure 4). Also, after processing the first event, the event counter has the value of 1, and the second event at 12:43:46 increments the counter to 2.

At 12:44:02, the event correlation window expires, since the earliest event in the window (that reflects the beginning of the window) has occurred more than 300 seconds ago. Since the event counter has the value of 2, the threshold condition has not been met, and thus the rule action can not be executed.

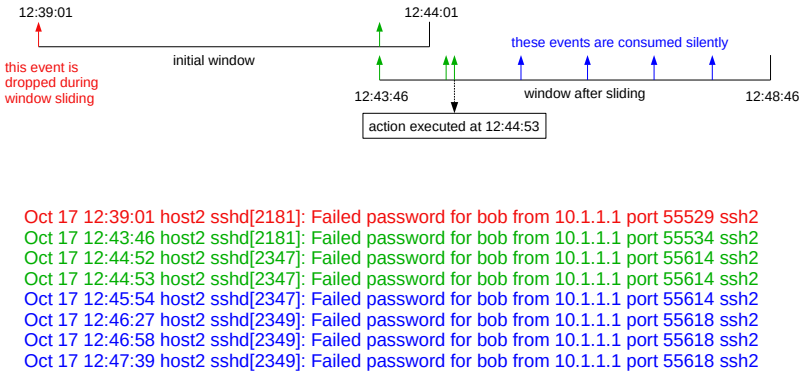


Figure 4: SingleWithThreshold event correlation operation

At that point, SEC could simply terminate the operation, but if that is done, SEC would miss event patterns that combine already observed events and potential future events. For example, see the pattern of 3 green events from Figure 4 that fit into the window of 300 seconds (the first green event has already been observed and the other two are going to happen in the near future after 12:44:02).

For facilitating the detection of such event patterns, the *SingleWithThreshold* operation employs a *sliding event correlation window* during event correlation process. With this approach, events that are older than the length of the window are dropped, and the beginning of the window is moved to the occurrence time of the earliest remaining event.

Note that sliding windows are used by *all* SEC event correlation operations that implement event counting – such operations are started by *SingleWithThreshold*, *SingleWith2Thresholds*, and *EventGroup* rules.

In the case of our example, the red event at 12:39:01 is dropped during window sliding at 12:44:02, since that event is older than 300 seconds (see Figure 4). Also, the beginning of the window is moved to 12:43:46 – that is the occurrence time of the earliest remaining event the operation has processed. Note that in the case of *no remaining events*, the operation *terminates*. After window sliding, the window will cover the time frame 12:43:46–12:48:46 (see Figure 4). Since one event was dropped during window sliding, the event counter of the operation is decremented to 1 (that reflects the fact that one event has remained in the window after sliding).

After window sliding at 12:44:02, the operation will see two events at 12:44:52 and 12:44:53 that increment the event counter to 3. Since the threshold condition has been met now (the operation has observed 3 events in 300 seconds, with relevant events having the green color in Figure 4), the operaton will execute the configured *pipe* action.

Note that after the action has been executed, the *SingleWithThreshold* operation will not terminate immediately, but it will continue to exist until the

end of the event correlation window. Also, all events observed during this time are consumed silently without executing the action again (see the blue events in Figure 4).

At 12:48:47, the event correlation window expires and the operation terminates. Note that if an event matches the rule from Listing 9 at 12:48:47, producing the same operation ID as the expired operation has, the event is processed by a new operation that is started immediately after the expired one has been destroyed.

If one wants the operation to terminate immediately after it has sent the warning email with the *pipe* action, the *reset* action can be utilized as illustrated by Listing 11.

Listing 11: The content of /etc/sec/sshd-reset.sec

```
type=SingleWithThreshold
ptype=RegExp
pattern=sshd\[\d+\]: Failed .+ for (\S+) from ([\d.]+) port \d+ ssh2
desc=user $1 ip $2
action=pipe 'Too many SSH login failures for user $1 from $2' \
            mail root@localhost; reset 0
thresh=3
window=300
```

The *reset* action terminates an event correlation operation started by some rule from the same configuration file (see the SEC official documentation for more details), and *reset 0* terminates the calling operation itself.

3.4 EventGroup rule

In previous sections, we discussed several *SingleWithThreshold* rule examples that were designed to track the count of events of specific type (e.g., SSH login failures). But what about scenarios that involve counting events of *different types* in one event correlation window? For example, consider SSH login failure events and HTTP access log events from Listing 12.

Listing 12: Example sshd login failure events from /var/log/secure and httpd GET events from /var/log/httpd/access_log

```
10.1.1.1 - - [08/Nov/2022:13:13:05 +0200] "GET /~bob HTTP/1.1" 404 196 "-" "curl/7.68.0"

Nov  8 13:13:20 host sshd[1551]: Failed password for bob from 10.1.1.1 port 50280 ssh2
Nov  8 13:13:25 host sshd[1551]: Failed password for bob from 10.1.1.1 port 50280 ssh2
Nov  8 13:13:29 host sshd[1551]: Failed password for bob from 10.1.1.1 port 50280 ssh2
Nov  8 13:13:36 host sshd[1553]: Failed password for bob from 10.1.1.1 port 41274 ssh2

10.1.1.1 - - [08/Nov/2022:13:13:39 +0200] "GET /~bob/test.html HTTP/1.1" 404 196 "-" "curl/7.68.0"

Nov  8 13:13:52 host sshd[1553]: Failed password for bob from 10.1.1.1 port 41274 ssh2
Nov  8 13:13:59 host sshd[1553]: Failed password for bob from 10.1.1.1 port 41274 ssh2
```

Also, suppose that we need to issue a warning email when the following events are observed within 60 seconds for the same ⟨user name, IP address of the remote host⟩ combination:

- 2 SSH login failure events for the user name from the remote host,

- 2 HTTP GET events to user's personal web page from the remote host, so that the HTTP status code is 404.

The *EventGroup* rule has been designed for tackling such tasks, and it allows to configure any number of patterns for matching different event types. With each pattern, a different threshold for the number of events can be specified. The number of patterns is provided with the rule type – for example, *EventGroup* denotes a rule with one event pattern, *EventGroup2* a rule with two event patterns, *EventGroup3* a rule with three event patterns, etc.

Listing 13 provides an example *EventGroup2* rule for addressing the event correlation task described above. In the rule definition, the first event pattern and the threshold are defined with *ptype*, *pattern*, and *thresh* keywords, while the second event pattern and relevant event threshold are defined with *ptype2*, *pattern2*, and *thresh2* keywords.

Listing 13: The content of /etc/sec/sshd-httpd.sec

```
type=EventGroup2
ptype=RegExp
pattern=sshd\[\d+\]: Failed password for (?<user>\S+) from (?<ip>[\d.]+) port \d+ ssh2
thresh=2
ptype2=RegExp
pattern2=^(?<ip>[\d.]+) - - \[.+?\] "GET /~(?<user>[^\s/]+)\S* HTTP/[\d.]+" 404
thresh2=2
desc=user $+{user} ip $+{ip}
action=pipe 'SSH and web probing for user $+{user} from $+{ip}' mail root@localhost
window=60
```

Note that for extracting the user name and IP address from SSH login failure and HTTP GET events, traditional $1 and $2 match variables can no longer be used, since their meaning depends on the event type (e.g., $1 represents the user name for SSH login failure events, and the IP address for HTTP GET events). For addressing this issue, named capture groups *(?<user>...)* and *(?<ip>...)* have been used in the *EventGroup2* rule from Listing 13 that set the match variables $+{user} and $+{ip}.

Since the value of the rule's *desc* keyword contains both match variables, the rule starts a separate event correlation operation for each combination of the user name and IP address of the remote host. According to the *window* keyword, the event correlation window of 60 seconds is used. Similarly to *SingleWithThreshold* operations, the event correlation window is sliding.

When the rule from Listing 13 processes the event from Listing 12, event correlation operation with the ID ⟨/etc/sec/sshd-httpd.sec, 0, user bob ip 10.1.1.1⟩ is created when the first event is observed at 13:13:05, and the event counter for HTTP GET events is set to 1.

After SSH login failure events at 13:13:20 and 13:13:25 have been observed, the event counter for SSH login failure events has the value of 2. Therefore, the threshold condition defined with the *thresh* keyword has been met, but since this is not the case for the second threshold condition given with the *thresh2* keyword, the warning email is not sent. For the same reason, the operation does not issue the email when two additional SSH login failure events appear at 13:13:29 and 13:13:36 (these events will increment the counter for SSH login failure events to 4).

When the second HTTP GET event at 13:13:39 arrives, the counter for HTTP GET events is incremented to 2, and now both threshold conditions have been met. Therefore, the operation will send an email warning to *root@localhost* with the following text: *SSH and web probing for user bob from 10.1.1.1.* After sending the email, the operation will continue to run until the end of the event correlation window at 13:14:05, consuming the events at 13:13:52 and 13:13:59 silently. At 13:14:06, the operation is terminated due to expiration of the event correlation window.

3.5 EventGroup operations with multiple actions

Previously presented *SingleWithThreshold* and *EventGroup* rule examples have involved the execution of the action *once* by the event correlation operation – the execution has taken place when all threshold conditions have been met for the first time, and after that, further events have been silently consumed by the operation without taking any action.

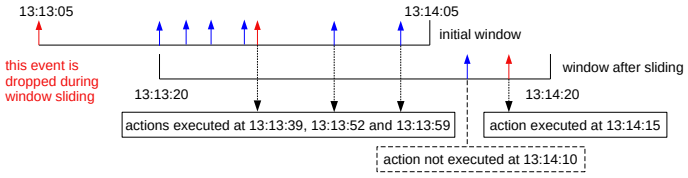
The *EventGroup* rule also allows for configuring a different behavior – each time the operation observes an event, it checks the threshold conditions, and in the case they are all satisfied, a configured action list is executed. Also, the operation continues its work as long as possible, sliding its event correlation window forward even if the action list has already been executed in the past. This behavior can be configured by setting the rule's *multact* keyword to *yes*, and Listing 14 provides a relevant rule example. Also, Figure 5 illustrates the work of the event correlation operation started by this rule.

Listing 14: The content of /etc/sec/sshd-httpd-multact.sec

```
type=EventGroup2
ptype=RegExp
pattern=sshd\[\d+\]: Failed password for (?<user>\S+) from (?<ip>[\d.]+) port \d+ ssh2
thresh=2
ptype2=RegExp
pattern2=^(?<ip>[\d.]+) - - \[.+?\] "GET /~(?<user>[^\s/]+)\S* HTTP/[\d.]+" 404
thresh2=2
desc=user ${user} ip ${ip}
action=pipe 'SSH and web probing for user ${user} from ${ip}' mail root@localhost
multact=yes
window=60
```

As can be seen from Figure 5, the operation executes the *pipe* action at 13:13:39 for the first time after the second HTTP GET event has been observed. By that time, 4 SSH login failure events have been seen already. The action is also executed at 13:13:52 and 13:13:59, since both threshold conditions are satisfied.

At 13:14:06, the event correlation window expires, and the HTTP GET event from 13:13:05 is dropped during window sliding. As a result, just one HTTP GET event will remain in the event correlation window, and the threshold condition given with the *thresh2* keyword is no longer met. For this reason, the action is not executed when the SSH login failure event arrives at 13:14:10. However, the arrival of the HTTP GET event at 13:14:15 will increment the event counter of HTTP GET events to 2, and as a result, both threshold conditions




```
10.1.1.1 - - [08/Nov/2022:13:13:05 +0200] "GET /~bob HTTP/1.1" 404 196 "-" "curl/7.68.0"
Nov  8 13:13:20 host sshd[1551]: Failed password for bob from 10.1.1.1 port 50280 ssh2
Nov  8 13:13:25 host sshd[1551]: Failed password for bob from 10.1.1.1 port 50280 ssh2
Nov  8 13:13:29 host sshd[1551]: Failed password for bob from 10.1.1.1 port 50280 ssh2
Nov  8 13:13:36 host sshd[1553]: Failed password for bob from 10.1.1.1 port 41274 ssh2
10.1.1.1 - - [08/Nov/2022:13:13:39 +0200] "GET /~bob/test.html HTTP/1.1" 404 196 "-" "curl/7.68.0"
Nov  8 13:13:52 host sshd[1553]: Failed password for bob from 10.1.1.1 port 41274 ssh2
Nov  8 13:13:59 host sshd[1553]: Failed password for bob from 10.1.1.1 port 41274 ssh2
Nov  8 13:14:10 host sshd[1560]: Failed password for bob from 10.1.1.1 port 51740 ssh2
10.1.1.1 - - [08/Nov/2022:13:14:15 +0200] "GET /~bob HTTP/1.1" 404 196 "-" "curl/7.68.0"
```

Figure 5: EventGroup2 event correlation operation with multiple actions

become satisfied again. Therefore, the event correlation operation executes the *pipe* action again at 13:14:15.

3.6 EventGroup rule for detecting ordered event sequences

By default, the *EventGroup* rule does not impose any order for matching events of different types. For example, the rules from Listings 13 and 14 allow the expected 2 SSH login failure events and 2 HTTP GET events to arrive in any order.

However, in some cases it is necessary to detect event sequences where events have a specific order. The *EventGroup* rule supports this functionality and for illustrating it, we will a use case that involves processing *iptables* firewall messages about blocked packets. Listing 15 depicts examples of such messages that describe denied access attempts from remote host 192.168.56.1 to ports 23/tcp (TELNET service) and 25/tcp (SMTP service) at the local system (192.168.56.116).

Listing 15: Linux iptables messages about blocked packets from /var/log/messages

```
Nov 10 16:04:44 localhost kernel: iptables: IN=enp0s8 OUT=
 MAC=08:00:27:1c:bd:21:0a:00:27:00:00:00:08:00 SRC=192.168.56.1
 DST=192.168.56.116 LEN=60 TOS=0x00 PREC=0x00 TTL=64 ID=47027 DF
 PROTO=TCP SPT=57302 DPT=23 WINDOW=64240 RES=0x00 SYN URGP=0

Nov 10 16:04:47 localhost kernel: iptables: IN=enp0s8 OUT=
 MAC=08:00:27:1c:bd:21:0a:00:27:00:00:00:08:00 SRC=192.168.56.1
 DST=192.168.56.116 LEN=60 TOS=0x00 PREC=0x00 TTL=64 ID=65333 DF
 PROTO=TCP SPT=49118 DPT=25 WINDOW=64240 RES=0x00 SYN URGP=0
```

Suppose we need to address the following event correlation task – a warning email needs to be issued if we observe an event sequence from Listing 15 (i.e., access attempt to port 23/tcp that is immediately followed by access attempt

to port 25/tcp) 2 times from the same remote host within 60 seconds. Listing 16 provides an example rule for addressing this problem.

Listing 16: The content of /etc/sec/telnet-smtp-sequences.sec

```
# This rule starts an event correlation operation for iptables
# blocked packet event for TELNET/SMTP service from an IP address.
# If from the same IP address an iptables event for port 23/tcp is
# immediately followed by event for port 25/tcp, and this sequence
# of two events is observed two times within 60 seconds, an e-mail
# alert is sent to local root user.

type=EventGroup2
ptype=RegExp
pattern=kernel: iptables:.* SRC=([\d.]+) .* PROTO=TCP .* DPT=23\b
thresh=2
ptype2=RegExp
pattern2=kernel: iptables:.* SRC=([\d.]+) .* PROTO=TCP .* DPT=25\b
thresh2=2
desc=Two TELNET->SMTP port probe sequences from host $1
egptype=RegExp
egpattern=1 2.*1 2
action=pipe '%s' mail root@localhost
window=60
```

Note that the rule from Listing 16 does not employ a minimalistic operation description string (set with the *desc* keyword) like the previous examples from this tutorial did, but a longer human-readable value has been configured. The operation description string can be accessed in action lists with the %s builtin action list variable, and the *pipe* action from Listing 16 uses the operation description string for the text of the email message.

In addition to numeric threshold conditions set by *thresh* and *thresh2* keywords, the rule from Listing 16 has additional keywords *egpattern* and *egptype* that define the *event group pattern* and its type respectively. In the case of this rule, the event group pattern is the following regular expression:

```
1 2.*1 2
```

The event group pattern is evaluated when all numeric threshold conditions have been met, and it is defining additional restrictions for the event sequence that the event correlation operation expects to observe. When event group pattern is evaluated, it is matched against the *event group string* that represents the sequence of events in the event correlation window. Note that if the event correlation window slides, dropped events will no longer be represented in the event group string.

By default, each event in the event group string is represented by a number that corresponds to the number of the matching pattern. For example, event matched by regular expression defined by *pattern2* keyword is represented by number 2. Also, the numbers appear in the event group string in the same order as matching events have been observed, and the numbers are separated by space characters.

For understanding how the event group pattern is used by event correlation operations, consider how example events from Figure 6 are processed by the

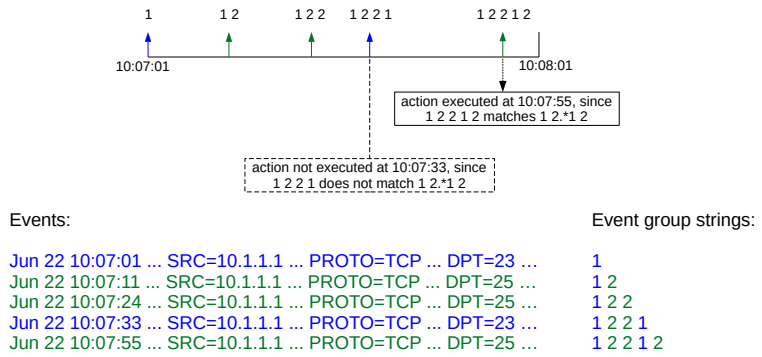


Figure 6: EventGroup2 event correlation operation with event group pattern

rule from Listing 16. On the appearance of the first event at 10:07:01, the rule starts an event correlation operation for remote host 10.1.1.1.

When the fourth event at 10:07:33 arrives, both threshold conditions have been satisfied, and therefore the event group string will be matched with the event group pattern. Since there is no match, the *pipe* action is not executed.

At 10:07:55, the event correlation operation observes the fifth event, and this time the event group string matches the event group pattern. Therefore, the *pipe* action is executed which sends the email message *Two TELNET->SMTP port probe sequences from host 10.1.1.1* to *root@localhost*.

As this example illustrates, event group patterns allow for matching specific event sequences where events have a particular order. Apart from regular expressions, more generic and powerful *PerlFunc* patterns can be used as event group patterns, and we will expore that topic in Section 6.4.

Also, the *EventGroup* rule has several other advanced features that did not fit into this tutorial due to space limitations, and a detailed description of these features can be found in SEC official documentation.

3.7 PairWithWindow rule

A rule example from Listing 9 in Section 3.1 generates a warning email if 3 SSH login failure events have been observed for the same combination of user name and SSH client IP address during 5 minutes. However, imagine the following situation – a user who has just returned from a longer vacation does not immediately recollect his/her password and will thus try a number of passwords in a short time frame, creating enough events for the rule from Listing 9 to trigger the warning email.

One way for addressing this problem is to memorize the time when the first login failure event for a user and a client host was observed, and then expect this user to successfully log in from the same client host within a reasonable amount of time (e.g., 5 minutes). If the expected successful login event does not

arrive within predefined time frame, a warning email is issued.

For example, consider example events from Listing 17. After the user alice has failed to log in from host 10.1.1.1 at 11:35:47, we are expecting this user to log in from host 10.1.1.1 by 11:40:47 the latest (i.e., during 5 minutes after the login failure). However, the expected event will not arrive, and there is only a second unsuccessful login attempt at 11:35:59. Therefore, a warning email should be issued for user alice and client host 10.1.1.1.

Listing 17: Example sshd login failure events from /var/log/secure

```
Nov 11 11:35:47 host sshd[1504]: Failed password for alice from 10.1.1.1 port 50406 ssh2
Nov 11 11:35:59 host sshd[1504]: Failed password for alice from 10.1.1.1 port 50406 ssh2
Nov 11 11:36:00 host sshd[1506]: Failed password for bob from 10.1.1.2 port 58942 ssh2
Nov 11 11:36:06 host sshd[1506]: Failed password for bob from 10.1.1.2 port 58942 ssh2
Nov 11 11:40:30 host sshd[1508]: Accepted password for bob from 10.1.1.2 port 36139 ssh2
```

In contrast, although user bob fails to log in from host 10.1.1.2 at 11:36:00, his third login attempt from the same client host 10.1.1.2 is successful at 11:40:30 (i.e., 4 minutes and 30 seconds after initial failure). In other words, we have seen the expected successful login event within the 5 minute time frame, and thus no warning email should be issued.

The *PairWithWindow* rule allows to address such event correlation tasks – it starts event correlation operations that expect the arrival of a specific event during some time frame. If an operation does not observe the expected event, it executes an action list and terminates. However, if the expected event arrives, the operation executes another action list and terminates.

Listing 18 displays an example *PairWithWindow* rule for processing events from Listing 17.

Listing 18: The content of /etc/sec/sshd-fail-success.sec

```
type=PairWithWindow
ptype=RegExp
pattern=sshd\[\d+\]: Failed .+ for (\S+) from ([\d.]+) port \d+ ssh2
desc=User $1 did not manage to log in from $2 within 5 minutes
action=pipe '%s' mail root@localhost
ptype2=RegExp
pattern2=sshd\[\d+\]: Accepted .+ for $1 from $2 port \d+ ssh2
desc2=User %1 logged in from %2 after initial failure
action2=pipe '%s' logger -p authpriv.debug -t sec
window=300
```

For understanding how the rule from Listing 18 works, the following provides a detailed description how each event from Listing 17 is processed:

- When the first event at 11:35:47 appears, the rule from Listing 18 matches this event (the regular expression given with the *pattern* keyword matches). Therefore, the rule starts the event correlation operation with the ID ⟨/etc/sec/sshd-fail-success.sec, 0, User alice did not manage to log in from 10.1.1.1 within 5 minutes⟩. According to the *window* keyword of the rule, this event correlation operation will run for the following 300 seconds.

  During this time frame, the operation will expect the event defined by the *pattern2* keyword of the rule. Note that *pattern2* keyword defines

not a regular expression, but a *regular expression template* which contains match variables $1 and $2. These match variables are substituted with the values when the event correlation operation is created – for example, $1 is substituted with alice.

Note that the substitution of $2 in the regular expression template involves the following issue – $2 holds the IP address 10.1.1.1, but each dot character (.) in the IP address is a regular expression atom that matches any single character. For avoiding any side effects, all characters that have a special meaning in regular expressions are masked during variable substitution.

After substituting $1 and $2 with their values in the regular expression template provided by *pattern2* keyword, the resulting regular expression is the following:

```
sshd\[\d+\]: Accepted .+ for alice from 10\.1\.1\.1 port \d+ ssh2
```

The operation with the ID ⟨/etc/sec/sshd-fail-success.sec, 0, User alice did not manage to log in from 10.1.1.1 within 5 minutes⟩ will run for 300 seconds, expecting an event that matches this regular expression.

- When the second event at 11:35:59 appears, it is handed over for processing to the previously created operation with the ID ⟨/etc/sec/sshd-fail-success.sec, 0, User alice did not manage to log in from 10.1.1.1 within 5 minutes⟩. That operation will silently consume the event.

  Since the operation will not observe successful login event for alice from 10.1.1.1 by 11:40:47, the event correlation window will expire at 11:40:48. Note that unlike *SingleWithThreshold* and *EventGroup* operations, *PairWithWindow* operations do not use sliding event correlation windows. Therefore, the beginning of the window is not moved forward to 11:35:59 (time of the second login failure), but the event correlation operation executes the *pipe* action defined with *action* keyword.

  The *pipe* action sends the following email message to *root@localhost* at 11:40:48: *User alice did not manage to log in from 10.1.1.1 within 5 minutes*. After sending the email with *pipe* action, the operation terminates.

- When the third event at 11:36:00 appears, the rule from Listing 18 starts the event correlation operation with the ID ⟨/etc/sec/sshd-fail-success.sec, 0, User bob did not manage to log in from 10.1.1.2 within 5 minutes⟩.

  This operation will run for 300 seconds, expecting an event that matches the following regular expression:

  ```
  sshd\[\d+\]: Accepted .+ for bob from 10\.1\.1\.2 port \d+ ssh2
  ```

- When the fourth event at 11:36:06 appears, it is silently consumed by the operation with the ID ⟨/etc/sec/sshd-fail-success.sec, 0, User bob did not manage to log in from 10.1.1.2 within 5 minutes⟩.

- When the fifth event at 11:40:30 appears, it is first matched against the regular expression of the *pattern* keyword like all previous events. Unlike the previous events, this event fails to match that expression. Therefore, the rule forwards this event to *all* event correlation operations it has started, so that these operations can match the event against their regular expressions.

  At 11:40:30, there are two such operations with the following IDs:

  1. ⟨/etc/sec/sshd-fail-success.sec, 0, User alice did not manage to log in from 10.1.1.1 within 5 minutes⟩ – this operation is expecting an event matching the regular expression

     ```
     sshd\[\d+\]: Accepted .+ for alice from 10\.1\.1\.1 port \d+ ssh2
     ```

  2. ⟨/etc/sec/sshd-fail-success.sec, 0, User bob did not manage to log in from 10.1.1.2 within 5 minutes⟩ – this operation is expecting an event matching the regular expression

     ```
     sshd\[\d+\]: Accepted .+ for bob from 10\.1\.1\.2 port \d+ ssh2
     ```

  It is easy to see that from these two operations, only the regular expression of the second operation produces a match. Therefore, the second operation executes the *pipe* action defined with the *action2* keyword of the rule.

  Note that the rule definition from Listing 18 has the *desc2* keyword that is connected to *action2* keyword. The purpose of *desc2* is to set the %s action list variable for the action list defined by *action2* keyword. The *desc2* keyword is especially useful if you want to employ the same human-readable message several times in a longer action list and want to refer to that message by %s, and the operation description string set by *desc* keyword is not appropriate for the message text.

- Before the *pipe* action defined by *action2* keyword can be executed at 11:40:30, the match variables need to be substituted with their values for *desc2* and *action2* keywords. However, this introduces the following issue – the regular expression

  ```
  sshd\[\d+\]: Accepted .+ for bob from 10\.1\.1\.2 port \d+ ssh2
  ```

  that has just produced a match does not have any capture groups, and the match variables $1 and $2 are thus unset. On the other hand, the regular expression that matched an event at 11:36:00 (and started the current event correlation operation) had $1 and $2 variables set to values that we need. In order to refer to previous values of these match variables when the operation was started, the %1 and %2 notation can be used.

  Therefore, when the event correlation operation executes the *pipe* action defined by the *action2* field at 11:40:30, the following *debug*-level syslog message will be produced with the *logger* tool: *User bob logged in from 10.1.1.2 after initial failure*.

  After executing the *pipe* action at 11:40:30, the operation ⟨/etc/sec/sshd-fail-success.sec, 0, User bob did not manage to log in from 10.1.1.2 within 5 minutes⟩ terminates immediately.

4 Contexts and pattern match caching

This section provides an introduction to contexts and a number of examples on how to use them for event processing. Also, the concept of pattern match caching is introduced in this section.

4.1 Introduction to contexts

Contexts are memory based objects that can be created and deleted from rules and event correlation operations. Also, as we will see in Section 4.5, *internal contexts* are created and deleted automatically by SEC. Note that contexts have a global scope and are both visible and accessible from all rules and event correlation operations. Contexts can be employed for several purposes, such as:

- representing the facts about past – for example, a context could represent the fact that a warning email has already been sent to security administrator about some offending host,
- implementing timers that execute specific action lists after given number of seconds – for example, after a firewall rule has been created from SEC, a context could be set up for removing that firewall rule after 600 seconds,
- collecting events and submitting them together for external processing – for example, a context could be used for collecting log events for a suspicious IP address during 300 seconds, and all collected events could be emailed to the security administrator for further review.

Figure 7 displays an example context data structure that resides in memory. A context data structure has at least one *name* which is used for referring to the context. If necessary, one can set up additional alias names with the *alias* action – for example, the context in Figure 7 has two names MYCONT and MYCONT2 that can be used interchangeably and refer to the same context data structure. Context names can also be removed with the *unalias* action, but when the last remaining context name disappears, the context data structure is dropped from memory (therefore, context names are similar to *hard links* of the UNIX file system). For explicitly destroying the context data structure with the removal of all context names, *delete* action can be used.

Apart from one or more names, another important property of the context is its *lifetime*. For example, the example context in Figure 7 has a lifetime of 3600 seconds (1 hour), and the context exists starting from its creation at 14:12:47 until 15:12:47 (November 10, 2022). If the lifetime is not specified when the context is created, the context will not expire and thus exist forever.

The *action-on-expire* is a property of the context that specifies an action list that is executed when the context expires. For example, the context in Figure 7 has the following action list configured for that purpose:

```
report MYCONT mail root@localhost
```

That action list is executed at 15:12:48 when more than 3600 seconds have elapsed since the creation of the context. The *report* action from Figure 7 will write all events from the event store of MYCONT to the standard input of the following command line, keeping the order of events in the event store:

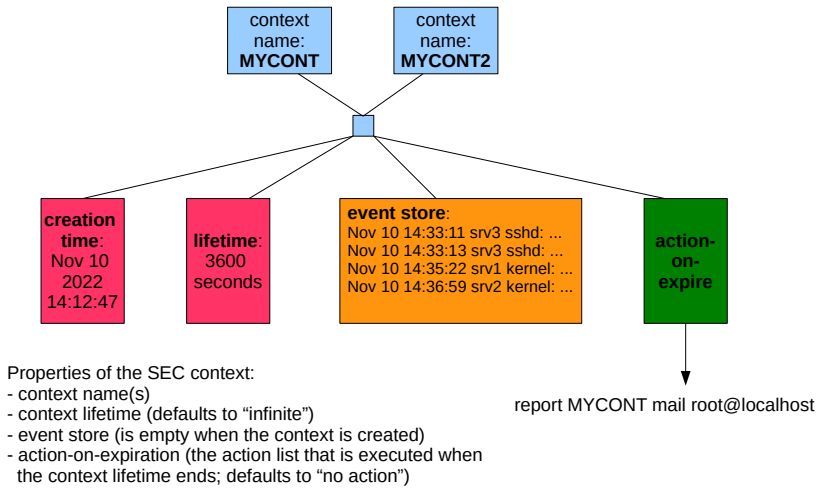


Figure 7: An example context

```
mail root@localhost
```

In other words, when the context MYCONT (also known as MYCONT2) expires, the content of its event store is emailed to *root@localhost*, and then the context is deleted.

The *event store* of the context is a memory based line buffer that is empty when the context is created. The *add* action can be used for appending lines to the end of the event store, whereas the *prepend* action prepends new lines to the beginning of the event store. Also, the *pop* and *shift* actions remove the last and first line from the event store respectively. For other event store related actions, please see SEC official documentation.

4.2 Using contexts for rule activation and deactivation

Consider the rule from Listing 19 for detecting the cases when the same user account is probed many times in a short time frame from the same remote host.

Listing 19: The content of /etc/sec/sshd-mass-scan.sec

```
type=SingleWithThreshold
ptype=RegExp
pattern=sshd\[\d+\]: Failed .+ for (\S+) from ([\d.]+) port \d+ ssh2
desc=user $1 ip $2
action=pipe 'Too many SSH login failures for user $1 from $2' \
            mail root@localhost
thresh=100
window=300
```

The rule from Listing 19 has one drawback – if the user account probing lasts at the same rate for several hours, a warning email with the same message will be sent after every 5 minutes to *root@localhost*.

To avoid repeated reporting of the same user name and IP address combination, this rule could be modified as in Listing 20.

Listing 20: The content of /etc/sec/sshd-mass-scan2.sec

```
type=SingleWithThreshold
ptype=RegExp
pattern=sshd\[\d+\]: Failed .+ for (\S+) from ([\d.]+) port \d+ ssh2
context=!USER_$1_HOST_$2_REPORTED
desc=user $1 ip $2
action=pipe 'Too many SSH login failures for user $1 from $2' \
            mail root@localhost; create USER_$1_HOST_$2_REPORTED 3600
thresh=100
window=300
```

The action list defined by the *action* keyword contains an additional *create* action that is executed after sending an email with *pipe*. The *create* action creates a new context with the name given with the first parameter, whereas the second parameter defines the lifetime of the context. Therefore, the *create* action in Listing 20 creates the context USER_⟨user name⟩_HOST_⟨IP address⟩_REPORTED with the lifetime of 3600 seconds (1 hour). For example, after sending the warning email *Too many SSH login failures for user alice from 10.1.1.1*, the context USER_alice_HOST_10.1.1.1_REPORTED is created in order to represent the fact that the email has already been sent for this user name and IP address combination.

Compared with the previous rule definition from Listing 19, the improved version from Listing 20 features an additional *context* keyword that defines the following Boolean expression:

```
!USER_$1_HOST_$2_REPORTED
```

In this expression, ! denotes logical NOT. The context name operand that follows ! evaluates true if and only if the context with the given name exists. Therefore, the above expression evaluates true if and only if the context with the given name does not exist.

Since the Boolean expression given with the *context* keyword uses context names as operands, it is called a *context expression*.

The above context expression is evaluated immediately after the regular expression given with the *pattern* keyword has matched an event, substituting $1 and $2 match variables with values from the regular expression match. The rule matches an event if and only if the context expression evaluates true.

Therefore, configuring such context expressions in rule definitions allows for restricting the matches produced by rules. For example, after the context USER_alice_HOST_10.1.1.1_REPORTED has been created, the rule from Listing 20 will no longer match SSH login failure events for user alice from SSH client host 10.1.1.1 during the following 3600 seconds.

Apart from the ! operator (logical NOT), you can use the following Boolean operators in context expressions – && (logical AND) and || (logical OR). In addition, you can use parentheses for grouping purposes. For example, the following context expression evaluates true if and only if contexts A and B do not exist and context C exists:

```
!(A || B) && C
```

Finally, if the rule has multiple event patterns (such as *EventGroup* and *PairWithWindow* rules), the pattern and the corresponding context expression must share the same number in the rule definition. For example, in order to define a context expression to be evaluated together with the pattern given with the *pattern2* keyword, this context expression must be provided with the *context2* keyword.

4.3 Using contexts for event collection and reporting

In order to illustrate how contexts can be used for event collection and reporting, consider the events from Listing 21 that represent SSH login failures for non-existing user accounts.

Listing 21: Example sshd login failure events for non-existing users from /var/log/secure

```
Nov 13 12:22:30 host sshd[1510]: Failed password for invalid user test from 10.1.1.1 port 54212 ssh2
Nov 13 12:22:33 host sshd[1510]: Failed password for invalid user test from 10.1.1.1 port 54212 ssh2
Nov 13 12:23:23 host sshd[1512]: Failed password for invalid user admin from 10.1.1.1 port 46350 ssh2
Nov 13 12:23:29 host sshd[1512]: Failed password for invalid user admin from 10.1.1.1 port 46350 ssh2
Nov 13 12:23:50 host sshd[1515]: Failed password for invalid user toor from 10.1.1.1 port 35574 ssh2
Nov 13 12:23:55 host sshd[1515]: Failed password for invalid user toor from 10.1.1.1 port 35574 ssh2
```

First, we will discuss a ruleset that involves the collection of such events in a fixed 60 second time window (see Listing 22).

Listing 22: The content of /etc/sec/sshd-invalid-report.sec

```
# If there has been an SSH probe of a non-existing user account from
# some host, create a reporting context for that host (provided that
# the context does not exist already). The context lifetime will be
# set to 60 seconds. When the context expires, all events recorded
# into the context event store will be mailed to root@localhost.

type=Single
ptype=RegExp
pattern=sshd\[\d+\]: Failed .+ for invalid user \S+ from ([\d.]+) port \d+ ssh2
context=!SSH_INVALID_ACCOUNT_PROBES_$1
continue=TakeNext
desc=create reporting context for $1
action=create SSH_INVALID_ACCOUNT_PROBES_$1 60 \
       ( report SSH_INVALID_ACCOUNT_PROBES_$1 mail root@localhost )

# Add the SSH probe event to the event store of the host's context.
# Note that if the context did not exist before the arrival of the
# current event, the context has been created by the previous rule.

type=Single
ptype=RegExp
pattern=sshd\[\d+\]: Failed .+ for invalid user \S+ from ([\d.]+) port \d+ ssh2
context=SSH_INVALID_ACCOUNT_PROBES_$1
desc=add SSH account probe event to reporting context for $1
action=add SSH_INVALID_ACCOUNT_PROBES_$1 $0
```

When the events from Listing 21 are processed by the ruleset from Listing 22, the first event at 12:22:30 matches the first rule. Note that since the context SSH_INVALID_ACCOUNT_PROBES_10.1.1.1 does not exist, the context expression given with the *context* keyword evaluates true. Therefore, the *create* action is executed which creates the SSH_INVALID_ACCOUNT_PROBES_10.1.1.1 context with the lifetime of 60 seconds. That context will be used for collecting all SSH login failures for non-existing users from SSH client host 10.1.1.1 during the following 60 seconds.

Also, the last parameter of the *create* action is the action list that will be executed when the context expires. That action list is enclosed in parentheses and consists of one *report* action, and the substitution of the $1 match variable with 10.1.1.1 in the action list yields the following result:

```
report SSH_INVALID_ACCOUNT_PROBES_10.1.1.1 mail root@localhost
```

In other words, when the SSH_INVALID_ACCOUNT_PROBES_10.1.1.1 context expires, its event store will be emailed to *root@localhost*.

Since the *continue* keyword of the first rule in Listing 22 has the value *TakeNext*, the event at 12:22:30 is passed to the following rule after the first rule has matched the event and created the context. The second rule in Listing 22 matches the event, since the SSH_INVALID_ACCOUNT_PROBES_10.1.1.1 context exists and the context expression given with the *context* keyword thus evaluates true[1].

Since the event matched the rule, the rule executes the following *add* action:

```
add SSH_INVALID_ACCOUNT_PROBES_10.1.1.1 <the value of the $0 variable>
```

Because the $0 match variable holds the entire matching line, the above *add* action appends the first event from Listing 21 to the event store of the context SSH_INVALID_ACCOUNT_PROBES_10.1.1.1. Since the event store was previously empty, that event becomes the first event in the store.

The following events at 12:22:33, 12:23:23, and 12:23:29 no longer match the first rule, since the SSH_INVALID_ACCOUNT_PROBES_10.1.1.1 context exists and the context expression given with the *context* keyword evaluates false. However, the second rule matches these events and they are appended to the end of the context event store in the order of arrival.

At 12:23:31, the context expires and 4 events in its event store are emailed to *root@localhost*. In the body of the email, the events are in the same order as in Listing 21 (i.e., in the order of arrival). After the email has been sent with the *report* action, the context is immediately deleted.

When the fifth event from Listing 21 appears at 12:23:50, the context is created again, with the event becoming the first event in the event store of the context. Also, the sixth event at 12:23:55 will become the second event in the event store. At 12:24:51, the context expires again, and 2 events from its event store will be emailed to *root@localhost*.

In Listing 23, a slightly modified version of the previous ruleset has been provided. The modified version introduces the *set* action into the second rule.

[1] Note that this rule does not actually need the *context* keyword, since the context always exists when an event is matched by the rule. This keyword has been included in the rule definition for illustrating the fact that contexts created by some rule can be used by in context expressions of other rules.

Listing 23: The content of /etc/sec/sshd-invalid-report2.sec

```
type=Single
ptype=RegExp
pattern=sshd\[\d+\]: Failed .+ for invalid user \S+ from ([\d.]+) port \d+ ssh2
context=!SSH_INVALID_ACCOUNT_PROBES_$1
continue=TakeNext
desc=create reporting context for $1
action=create SSH_INVALID_ACCOUNT_PROBES_$1 60 \
       ( report SSH_INVALID_ACCOUNT_PROBES_$1 mail root@localhost )

# Add the SSH probe event to the event store of the host's context.
# Also, extend the lifetime of the context by 60 seconds starting
# from the current moment.

type=Single
ptype=RegExp
pattern=sshd\[\d+\]: Failed .+ for invalid user \S+ from ([\d.]+) port \d+ ssh2
context=SSH_INVALID_ACCOUNT_PROBES_$1
desc=add SSH account probe event to reporting context for $1
action=add SSH_INVALID_ACCOUNT_PROBES_$1 $0; \
       set SSH_INVALID_ACCOUNT_PROBES_$1 60
```

The *set* action can be used for modifying context's lifetime and the action list that is executed on the expiration of the context. In the second rule, *set* changes the context lifetime by setting it to 60 seconds *starting from the current moment*. In other words, each time the rule appends a new event to context's event store, the lifetime of the context is extended for the following 60 seconds.

Therefore, the context will exist as long as any two consecutive events in its event store are separated by at most 60 seconds. The context expires when more than 60 seconds have elapsed since the last event was added to its event store. This kind of event collection and reporting method is useful for tracking error conditions that have no fixed duration, and the absence of relevant messages during some time frame indicates the end of the error condition.

For example, in the case of events from Listing 21, they are all added to the event store of the SSH_INVALID_ACCOUNT_PROBES_10.1.1.1 context after it has been created at 12:22:30. The context will expire at 12:24:56 (i.e., 61 seconds after the last event was added) and all 6 events are emailed to *root@localhost* before the context is deleted.

4.4 Pattern match caching

When one studies the rulesets from Listings 22 and 23 more closely, it is easy to see that the rules of both rulesets are sharing the same regular expression pattern:

```
sshd\[\d+\]: Failed .+ for invalid user \S+ from ([\d.]+) port \d+ ssh2
```

Due to the *continue=TakeNext* setting in the first rule, matching events are always passed to the second rule for further processing. Also, in the case the event does not match the first rule, the second rule is tried (again unsuccessfully). In other words, the rulesets from Listings 22 and 23 involve the matching of the same regular expression twice against every event.

For avoiding such redundant work, SEC supports *pattern match caching* – if a pattern matches, it is possible to store the result of the match (i.e., the values of all match variables) to the *pattern match cache*. Also, further rules can search the pattern match cache for entries that indicate a previous match by specific pattern. That allows to avoid repeated matching of the same pattern against the same input event.

For understanding this concept, consider the ruleset from Listing 24 that is a modification of the previous ruleset from Listing 23.

Listing 24: The content of /etc/sec/sshd-invalid-report3.sec

```
# If the regular expression of the rule matches, the result
# of the match (i.e., all match variables and their values)
# will be cached under the pattern match cache entry SSH_PROBE.

type=Single
ptype=RegExp
pattern=sshd\[\d+\]: Failed .+ for invalid user \S+ from ([\d.]+) port \d+ ssh2
varmap=SSH_PROBE
context=!SSH_INVALID_ACCOUNT_PROBES_$1
continue=TakeNext
desc=create reporting context for $1
action=create SSH_INVALID_ACCOUNT_PROBES_$1 60 \
       ( report SSH_INVALID_ACCOUNT_PROBES_$1 mail root@localhost )

# In order to match the input event, the presence of the SSH_PROBE
# entry in the pattern match cache is checked. If the SSH_PROBE
# entry is found, the pattern matches and all match variables and
# their values will be retrieved from the SSH_PROBE entry. If the
# entry is not found, the pattern does not match.

type=Single
ptype=Cached
pattern=SSH_PROBE
context=SSH_INVALID_ACCOUNT_PROBES_$1
desc=add SSH account probe event to reporting context for $1
action=add SSH_INVALID_ACCOUNT_PROBES_$1 $0; \
       set SSH_INVALID_ACCOUNT_PROBES_$1 60
```

The first rule of the ruleset contains an additional *varmap* keyword which creates a new *pattern match cache entry* with the name SSH_PROBE, provided that the regular expression pattern of the rule has matched an input event. The SSH_PROBE entry holds all match variables and their values produced by the regular expression match. If the regular expression does not match an input event, the entry is not created.

It is important to note that in the case of the regular expression match, the SSH_PROBE entry is created *before* the context expression given with the *context* keyword is evaluated. In other words, the creation of pattern match cache entries depends *solely* on pattern matching.

The second rule of the ruleset from Listing 24 has a pattern of type *Cached*. In the case of this pattern type, the pattern is the name of a pattern match cache entry. The pattern matches an input event if and only if the pattern match cache entry with the given name exists in the pattern match cache. Also, if the pattern matches, all match variables and their values will be retrieved from the pattern match cache entry.

As can be seen from Listing 24, the second rule searches the pattern match cache for the entry SSH_PROBE. Finding that entry indicates that the regular expression of the first rule has matched the current input event, and also produces a match by the *Cached* pattern of the second rule. In the case of the match, the *Cached* pattern creates exactly the same match variables as the first rule did, setting these variables to the same values as in the first rule. Therefore, the use of the *Cached* pattern in the ruleset from Listing 24 helps to avoid redundant work and needless repeated matching of the same regular expression against the same input.

Finally, note some facts about the pattern match cache:

- The pattern match cache entries that have been created for some input event remain in the cache only as long as *this input event is being processed by rules*. When the processing of the input event ends, the pattern match cache is cleared.
- For any input event, more than one pattern match cache entry can be created – when the event is being processed by rules, the patterns of several rules can match the event, with each rule creating a different cache entry.

4.5 Internal contexts

When SEC is monitoring several input files, the following issue can arise – a rule that has been designed for processing events from one input file accidentally matches an event from another input file. For example, generic regular expression patterns like `error: (.+)` can be one reason for such unexpected matches.

For addressing this issue, SEC supports internal contexts. *Internal contexts* are not explicitly created from rules and event correlation operations, but are rather set up automatically for indicating that the currently processed event originates from a specific input file. Similarly to a pattern match cache entry, an internal context exists while the current input event is being processed by rules. When the processing of the event is complete, the context is deleted.

The creation of internal contexts can be activated with the `--intcontexts` command line option, for example:

```
sec --intcontexts --conf=echo2.sec
    --input=/var/log/messages --input=/var/log/secure
```

With this option, internal context _FILE_EVENT_/var/log/messages is created before processing an input event from */var/log/messages*. This internal context is removed once the processing of the input event is complete. Also, internal context _FILE_EVENT_/var/log/secure is set up while an input event from */var/log/secure* is being processed.

For example, suppose SEC was started with the above command line and *echo2.sec* configuration file containing the rule from Listing 25.

The rule from Listing 25 will echo all input lines from */var/log/messages* to standard output, ignoring input lines from */var/log/secure*. As this example illustrates, the use of internal contexts allows for restricting the matches by generic regular expression patterns to events from specific input files only.

Listing 25: The content of /etc/sec/echo2.sec

```
type=Single
ptype=RegExp
pattern=.
context=_FILE_EVENT_/var/log/messages
desc=echo input line from /var/log/messages
action=write - $0
```

The user can also configure custom names for internal contexts. For example, the following command line will set up internal context SECURE for input events from */var/log/secure*:

```
sec --intcontexts --conf=echo2.sec
    --input=/var/log/messages --input=/var/log/secure=SECURE
```

Note that configuring custom internal context name(s) on command line will automatically enable the `--intcontexts` option, and therefore its presence is not strictly required in the above command line.

Finally, for patterns that support match variables (e.g., regular expression patterns), SEC automatically sets up match variables $+{_inputsrc} and $+{_intcontext} that hold the input file name and its internal context name respectively. If the creation of internal contexts has not been activated, the $+{_intcontext} variable remains unset.

For example, the rule from Listing 26 will echo input lines from */var/log/messages* and */var/log/secure* to standard output, preceding each line with the names of input file and internal context.

Listing 26: The content of /etc/sec/echo3.sec

```
type=Single
ptype=RegExp
pattern=.
context=_FILE_EVENT_/var/log/messages || _FILE_EVENT_/var/log/secure
desc=echo input lines from /var/log/messages and /var/log/secure
action=write - $+{_inputsrc} $+{_intcontext} $0
```

As we have learned before, the context expression given with the *context* keyword is evaluated *after* the regular expression pattern match (that allows to include match variables in context names). However, in the case of the rule from Listing 26, internal context names do not contain any match variables. Also, in such cases it is often worthwhile to evaluate the context expression *before* a more expensive regular expression match. For example, if most input lines are not originating from */var/log/messages* and */var/log/secure*, the rule from Listing 26 spends a lot of CPU time on needless regular expression matching.

For evaluating the context expression *before* the pattern match, it has to be enclosed in square brackets. Listing 27 provides a relevant example that is a modification of the rule from Listing 26.

Listing 27: The content of /etc/sec/echo4.sec

```
type=Single
ptype=RegExp
pattern=.
context=[ _FILE_EVENT_/var/log/messages || _FILE_EVENT_/var/log/secure ]
desc=echo input lines from /var/log/messages and /var/log/secure
action=write - $+{_inputsrc} $+{_intcontext} $0
```

5 Synthetic events

In the previous section, we discussed contexts that allow for joining several rules into one event processing scheme. This section discusses synthetic events that offer another opportunity for creating more complex event correlation schemes from individual rules.

5.1 Introduction to synthetic events

Synthetic events are special kind of input events for SEC rules that are not read from input files, but rather created with SEC actions. Some synthetic events are generated by SEC itself on startup and reception of specific signals. Synthetic events receive the same treatment as other input events and are processed by rules in the same way.

The simplest SEC action for generating a synthetic event is the *event* action. Consider the following example action:

```
event 5 This is an example event
```

This action schedules the synthetic event *This is an example event* to be generated after 5 seconds. The following action generates the synthetic event *This is an example event* immediately (i.e., with a time delay of 0 seconds):

```
event 0 This is an example event
```

Also, if the text of the synthetic event does not begin with a numeral, 0 can be omitted, and the above example can be thus rewritten as follows:

```
event This is an example event
```

If SEC has been started with the `--intcontexts` option, synthetic events that have been created with SEC actions have the internal context _INTERNAL_EVENT. For generating synthetic events with custom internal context names, the *cevent* action can be used. For example, the following action triggers the synthetic event *This is a test message* immediately with internal context MESSAGES:

```
cevent MESSAGES 0 This is a test message
```

Listing 28 provides an example ruleset of two rules, where event correlation operations started by the first rule use the *event* action to provide input events for the second rule.

Listing 28: The content of /etc/sec/sshd-fail-wo-success-count.sec

```
# if an SSH login failure is observed for some user from host
# <ip> which is not followed by a successful login for the same
# user from this host during 2 minutes, generate a synthetic event
# 'SSH_PROBE_FROM_HOST_<ip>'

type=PairWithWindow
ptype=RegExp
pattern=sshd\[\d+\]: Failed .+ for (\S+) from ([\d.]+) port \d+ ssh2
desc=user $1 ip $2
action=event SSH_PROBE_FROM_HOST_$2
ptype2=RegExp
pattern2=sshd\[\d+\]: Accepted .+ for $1 from $2 port \d+ ssh2
desc2=SSH login successful for %1 from %2
action2=logonly %s
window=120

# match synthetic events generated by event correlation operations
# of the previous rule, and send a warning email to root@localhost
# if the same host has made 3 SSH probes in 300 seconds

type=SingleWithThreshold
ptype=RegExp
pattern=SSH_PROBE_FROM_HOST_([\d.]+)
desc=ip $1
action=pipe 'Repeated account probing from $1' mail root@localhost
thresh=3
window=300
```

If an SSH login failure event is observed for some user name and SSH client host, the first rule in Listing 28 starts a *PairWithWindow* event correlation operation for that user name and client host combination. The event correlation operation will wait for a successful login event for this user from the same client host. If the expected event does not arrive within 120 seconds (2 minutes), the event correlation operation generates the following synthetic event:

```
SSH_PROBE_FROM_HOST_<IP address of the SSH client host>
```

This synthetic event serves as an input for the second rule in Listing 28, since the regular expression pattern of the rule matches the synthetic event. The rule starts a *SingleWithThreshold* event correlation operation for each IP address observed in matching synthetic events. If an operation that is running for some IP address observes 3 synthetic events during 300 seconds (5 minutes) for this IP address, a warning email is sent to *root@localhost*.

Rules and event correlation operations that are accepting synthetic events for input can also produce synthetic events for output. This output can be processed by further rules and event correlation operations which facilitates the creation of event processing pipelines of any length.

5.2 Generating synthetic events from Calendar rule

Suppose you have a script that makes a critical backup of your system and starts every evening at 6:00 PM. Also, the script normally runs about 2-3 minutes and produces the following *syslog* message after successfully made backup:

```
Nov 17 18:02:03 myhost backup.sh[23119]: Backup completed
```

Suppose you need to generate an alert if the backup script fails to start without producing any message, or runs too long without completing by 6:05 PM. It is a subtle task since an alert needs to be triggered not by the appearance of some backup script message pattern in some time window, but rather by the absence of any messages between 6:00 and 6:05 PM.

SEC *Calendar* rule has been designed for addressing such tasks and it executes action lists at specific times. The *Calendar* rule uses a *crontab*-like time specification with some subtle differences that are discussed below. Also, Listing 29 provides an example ruleset that illustrates the use of the *Calendar* rule.

Listing 29: The content of /etc/sec/backup.sec

```
# generate a synthetic event 'Checking if backup is done'
# every evening at 5:59 PM

type=Calendar
time=59 17 * * *
desc=trigger backup check
action=event Checking if backup is done

# after observing the synthetic event, wait for the backup
# completion message during the following 6 minutes, and
# send a warning email if it does not arrive

type=PairWithWindow
ptype=SubStr
pattern=Checking if backup is done
desc=checking for backup
action=pipe 'Backup not completed on time' mail root@localhost
ptype2=RegExp
pattern2=backup\.sh\[\d+\]: Backup completed
desc2=Backup completed
action2=logonly %s
window=360
```

The first rule in Listing 29 is the *Calendar* rule with the following time specification:

```
59 17 * * *
```

The first two fields denote 5:59 PM, and since asterisks are provided for the day (third field), month (fourth field), and weekday (fifth field), the entire time specification denotes "5:59 PM every day". Therefore, the *Calendar* rule generates a synthetic event *Checking if backup is done* every evening at 17:59.

This synthetic event is matched by the following *PairWithWindow* rule that starts an event correlation operation at 17:59. This operation will expect the *Backup completed* message from the backup script *backup.sh*, and if this message does not arrive during the following 360 seconds (6 minutes, i.e., by 18:05), the operation sends the warning message *Backup not completed on time* to *root@localhost*.

Note that the *context* keyword with a context expression can be included in the *Calendar* rule definition that allows for executing the rule action list only if the context expression evaluates true. Also, apart from generating synthetic

events, the *action* keyword of the *Calendar* rule can be used for specifying any action list, and SEC can thus act as a simple replacement for the UNIX *cron* daemon. However, the time specification of the *Calendar* rule has some subtle differences from standard *crontab* specification.

First, although conditions set by time specification fields are mostly joined by logical AND, *crontab* joins conditions for the day (third field) and weekday (fifth field) by logical OR. For example, consider the following time specification:

```
0 6 25-31 10 7
```

With *crontab*, that time specification denotes the following time moments:

- 6:00 AM every day starting from October 25 until October 31 (the last 7 days in October),
- 6:00 AM every Sunday in October.

In contrast, the *Calendar* rule joins *all* conditions with logical AND, and the above time specification denotes the following – 6:00 AM on a Sunday in October, provided that this Sunday is among the last 7 days in October. This time description can be rephrased as follows – 6:00 AM on the *last Sunday* of October.

The previous example leads to the following question – how to denote the last day of the month? Whereas standard *crontab* allows values from 1 to 31 for day (third field), the *Calendar* rule also supports the value of 0 which matches the last day of the month. For example, consider the following time specification:

```
55 23 0 * *
```

This time specification denotes 11:55 PM on the last day of every month. For example, in February 2023 that would mean February 28 11:55 PM, whereas in February 2024 that would mean February 29 11:55 PM.

Unlike *crontab*, the time specification of the *Calendar* rule also has an optional sixth field for matching years. Allowed values range from 0 to 99 that denote the last two digits of the year. For example, consider the following time specification:

```
0 1 1 1 * */2
```

This time specification denotes 1:00 AM on January 1 in every even year (e.g., 2020 and 2022, but not 2021). For matching 1:00 AM on January 1 for odd years, the following time specification can be used:

```
0 1 1 1 * 1-99/2
```

5.3 Receiving synthetic events from periodically executed commands

In previous sections, we discussed how to generate synthetic events with the *event* and *cevent* actions. While these actions are powerful, it is often useful to receive data from commands started by SEC, and process received data with SEC rules.

The *spawn* action addresses this task – it starts a user-defined command line and reads its standard output, so that each line in the standard output becomes a synthetic event. For example, consider the following action:

```
spawn echo This is a test
```

This action forks a process for executing the *echo* command asynchronously. Since that command writes a line *This is a test* to standard output, this line becomes a synthetic event. If the command runs for longer amount of time and does not produce immediate output, SEC will check after short time periods whether new data have become available for reading.

In order to disambiguate synthetic events of external commands from other input, the *cspawn* action can be used for setting specific internal context for synthetic events. For example, consider the following action:

```
cspawn TEST echo This is a test
```

This action generates the synthetic event *This is a test* with internal context TEST.

As another example, Listing 30 depicts the output from the *df* command that provides information about the usage of file systems (see the third column in the output).

Listing 30: Example output from df command

```
df -t xfs --output="source,target,pcent"

Filesystem      Mounted on Use%
/dev/md125      /             9%
/dev/md126      /boot        29%
/dev/md124      /data        96%
```

Also, Listing 31 provides an example ruleset for detecting file systems that have been used by 95% or more (this ruleset assumes that SEC has been started with the `--intcontexts` command line option).

Listing 31: The content of /etc/sec/df.sec

```
type=Calendar
time=* * * * *
desc=run df command in every minute
action=cspawn DF df -t xfs --output="source,target,pcent"

type=Single
ptype=RegExp
pattern=^(\S+)\s+(\S+)\s+(9[5-9]|100)%
context=DF && !FS_ALERT_$1
desc=file system $1 is getting full
action=pipe 'File system $1 mounted on $2 is used by $3%' \
            mail root@localhost; create FS_ALERT_$1 3600
```

The first rule (*Calendar*) in Listing 31 executes the *df* command once in every minute with the *cspawn* action, creating a synthetic event from every line that has been read from the standard output of *df*. For disambiguating these synthetic events from regular input events received from input fules, internal context DF is used.

The regular expression pattern of the second rule (*Single*) in Listing 31 matches these synthetic events, provided that the file system usage is at least

95%. Also, the context expression given the *context* keyword verifies that a line that matches the regular expression is indeed originating from the *cspawn* action in the first rule. In addition, the context expression also verifies that the context FS_ALERT_⟨file system⟩ does not exist for the given file system.

For file systems with high usage, the second rule sends a warning email to *root@localhost* and creates the FS_ALERT_⟨file system⟩ context with the lifetime of 3600 seconds (1 hour). The presence of that context suppresses repeated emails about the same file system for the following 1 hour.

For example, when lines from Listing 30 are processed by the second rule, the last line for the file system */dev/md124* matches, and the following email warning message is sent to *root@localhost*: *File system /dev/md124 mounted on /data is used by 96%*. After sending that email, the context FS_ALERT_/dev/md124 is created that suppresses further emails about the */dev/md124* file system for 1 hour.

Note that the *context* keyword of the second rule from Listing 31 defines a context expression that has been configured to be evaluated after the regular expression match, since one of the context names (FS_ALERT_$1) contains a match variable that is set by the regular expression. For having the context expression evaluated *before* the regular expression is tried, one can employ the *SingleWithSuppress* rule as illustrated in Listing 32. Note that the context expression of the *SingleWithSuppress* rule is no longer containing any match variables and can thus be enclosed in square brackets, forcing its evaluation before the regular expression match is attempted (if the context expression evaluates false, the regular expression is not tried).

Listing 32: The content of /etc/sec/df2.sec

```
type=Calendar
time=* * * * *
desc=run df command in every minute
action=cspawn DF df -t xfs --output="source,target,pcent"

type=SingleWithSuppress
ptype=RegExp
pattern=^(\S+)\s+(\S+)\s+(9[5-9]|100)%
context=[ DF ]
desc=file system $1 is getting full
action=pipe 'File system $1 mounted on $2 is used by $3%' \
            mail root@localhost
window=3600
```

The *SingleWithSuppress* rule from Listing 32 starts a separate event correlation operation for each file system, and the operation processes events as follows:

- when the operation observes the first event (i.e., the event that triggered the creation of the operation), the *pipe* action defined with rule's *action* keyword is executed,
- all following events (i.e., the second, the third, etc.) are consumed silently by the operation during 3600 seconds (the value of the *window* keyword),
- when the window of 3600 seconds expires (i.e., more than 3600 seconds have elapsed since the arrival of the first event), the operation terminates.

Therefore, the rulesets from Listings 31 and 32 produce the same output.

5.4 Receiving synthetic events from indefinitely running commands

In some cases, it is necessary to collect synthetic events from commands that run indefinitely. Ideally, the command should start and terminate together with SEC. That raises the following question – how to detect from rules that SEC has just started or is going to terminate?

For addressing this issue, the `--intevents` command line option activates the generation of special synthetic events when SEC starts or receives specific signals. For example, on startup the synthetic event SEC_STARTUP is generated that will be the very first event for SEC rules to process. On the reception of the HUP signal, SEC goes through so called *full restart* – it drops all event correlation state, terminates all child processes, reloads all configuration files, and reopens all input files. When full restart is complete, the synthetic event SEC_RESTART is generated. Also, when SEC is terminated with the TERM signal, the synthetic event SEC_SHUTDOWN is generated.

For other synthetic events that are activated by the `--intevents` option, see the official documentation. For all such synthetic events, internal context SEC_INTERNAL_EVENT is used.

Listing 33 displays an example ruleset for executing the *nc* command when SEC starts up or has gone through full restart, so that input events can be received from port 10514/tcp. The ruleset assumes that SEC has been started with `--intevents` and `--intcontexts` options.

Listing 33: The content of /etc/sec/nc.sec

```
type=Single
ptype=RegExp
pattern=^(?:SEC_STARTUP|SEC_RESTART)$
context=[ SEC_INTERNAL_EVENT ]
desc=listen on 10514/tcp for incoming events
action=cspawn NETCAT nc -l -k 10514

type=Single
ptype=RegExp
pattern=.
context=[ NETCAT ]
desc=echo events from 10514/tcp
action=write - $0
```

The first *Single* rule in Listing 33 matches synthetic events SEC_STARTUP and SEC_RESTART, and for disambiguating them from other similarly looking events, the presence of internal context SEC_INTERNAL_EVENT is verified before the regular expression match is attempted. When either of these synthetic events is observed, the following command line is started with the *cspawn* action:

```
nc -l -k 10514
```

That command line receives input lines from port 10514/tcp and prints them to standard output, and the *cspawn* action turns them into synthetic events with internal context NETCAT.

Note that if the first rule in Listing 33 would only match the SEC_STARTUP event, the *nc* command would not be restarted after SEC has gone through full restart that involves terminating the previous instance of *nc*.

The second *Single* rule in Listing 33 matches synthetic events received from the *nc* command and echoes them to standard output. To avoid matching other input lines against the regular expression pattern of the rule, the rule verifies the presence of the NETCAT internal context before the regular expression match is attempted.

When SEC terminates, all its child processes (including the *nc* process) will be terminated with the TERM signal by default. Therefore, there is no need to have another rule in Listing 33 for terminating the *nc* command when the SEC_SHUTDOWN synthetic event appears.

6 Advanced topics

This section covers some advanced topics such as creating hierarchical rulebases and using custom Perl code in rule definitions.

6.1 Hierarchical rulebases

As discussed in Section 2.4, when several configuration files have been specified on SEC command line, by default all configuration files are applied for processing an input event. However, using a larger number of configuration files with many rules might introduce a significant computational overhead.

For example, consider a scenario with 10 configuration files, each containing 30 rules for processing messages from a specific application (i.e., there are 300 rules in total). When an input message arrives that has relevant rules only in one of the configuration files, 270 rules from 9 other configuration files are needlessly matched against the input event.

For addressing this issue, rulesets can be organized in a hierarchical fashion with the help of *Jump* and *Options* rules. This section describes an example hierarchical rulebase that involves 4 configuration files:

- *main-rules.sec* – level 1 ruleset of *Jump* rules that are applied against all input events from */var/log/messages* and */var/log/secure*, so that the *Jump* rules direct input events to relevant level 2 configuration files for further processing,
- *iptables-rules.sec* – level 2 ruleset for processing *iptables* events. The ruleset receives all input events from a level 1 *Jump* rule in *main-rules.sec*, using the *Options* rule to subscribe to these input events,
- *sshd-rules1.sec* – level 2 ruleset for processing SSH authentication events. The ruleset receives all input events from a level 1 *Jump* rule in *main-rules.sec*, using the *Options* rule to subscribe to these input events,
- *sshd-rules2.sec* – level 2 ruleset for processing SSH authentication events and related synthetic events. The ruleset receives all input events from level 1 *Jump* rules in *main-rules.sec*, using the *Options* rule to subscribe to these input events.

Also, suppose SEC has been started with the following command line:

```
sec --conf=/etc/sec/main-rules.sec --conf=/etc/sec/iptables-rules.sec
    --conf=/etc/sec/sshd-rules1.sec --conf=/etc/sec/sshd-rules2.sec
    --input=/var/log/messages --input=/var/log/secure --intcontexts
```

Listing 34 provides the level 1 ruleset of 3 *Jump* rules. Note that *Jump* rules do not start any event correlation operations and their sole purpose is to direct matching events to specific rulesets for further processing.

Listing 34: The content of /etc/sec/main-rules.sec

```
type=Jump
ptype=SubStr
pattern=kernel: iptables:
context=[ _FILE_EVENT_/var/log/messages ]
cfset=iptables

type=Jump
ptype=RegExp
pattern=sshd\[\d+\]:
context=[ _FILE_EVENT_/var/log/secure ]
cfset=sshd1 sshd2

type=Jump
ptype=TValue
pattern=True
context=[ SSH ]
cfset=sshd2
```

The first *Jump* rule in Listing 34 directs *iptables* events that have been received from */var/log/messages* to the ruleset in *iptables-rules.sec*. It is assumed that *iptables* events have the format depicted in Figure 15 (see Section 3.6). For recognizing *iptables* events, the first rule verifies the presence of the internal context of the */var/log/messages* input file (_FILE_EVENT_/var/log/messages), and then uses the *SubStr* pattern for matching a message header of *iptables* events.

The first *Jump* rule also features the *cfset* keyword that provides the name of the configuration file set (*iptables*) where the processing of matching events should continue. *Configuration file set* is a collection of one or more configuration files that have a common name and can be targeted from *Jump* rules. A configuration file can be assigned to a configuration file set with the *joincfset* keyword of the *Options* rule.

For example, the configuration file in Listing 35 has the *Options* rule that joins that configuration file to the configuration file set *iptables*. Since the first *Jump* rule in Listing 34 directs matching events to *iptables* configuration file set for further processing, the ruleset from Listing 35 will receive these events as input.

The set names for the *joincfset* keyword of the *Options* rule can be freely chosen. If a new set name is introduced that has not been used in other configuration files, a new configuration file set with the given name is created that has the current configuration file as its only member. Note that the *joincfset* keyword of the *Options* rule can be employed for joining the current configuration file to more than one set if several set names are provided.

Listing 35: The content of /etc/sec/iptables-rules.sec

```
type=Options
procallin=no
joincfset=iptables

type=SingleWithThreshold
ptype=RegExp
pattern=kernel: iptables:.* SRC=([\d.]+)
context=!PACKET_DROPS_REPORTED_$1
desc=packet drops IP $1
action=pipe 'Too many dropped packets from $1' mail root@localhost; \
       create PACKET_DROPS_REPORTED_$1 3600
window=10
thresh=100
```

Also, note that the *procallin* keyword has the value *no* in the *Options* rule in Listing 35. That setting means that the ruleset in the configuration file is accepting input events from *Jump* rules only. Setting *procallin* to *yes* would mean that the ruleset in Listing 35 is accepting the following input events:

- input events received from */var/log/messages* and */var/log/secure* (i.e., the input events of the level 1 ruleset in Listing 34),
- input events received from *Jump* rules that submit them to *iptables* configuration file set.

As can be seen from Listing 34, the *cfset* keyword of the second *Jump* rule specifies two configuration file sets *sshd1* and *sshd2*. In such cases, a matching input event is submitted to configuration file sets in the same order as they have been provided for the *cfset* keyword. For example, after the second *Jump* rule in Listing 34 has matched an SSH authentication event, it is first submitted to configuration file set *sshd1* (see Listing 36) and then to configuration file set *sshd2* (see Listing 37).

If the *Jump* rule submits events to a configuration file set with more than one configuration file, they are applied in traditional order as discussed in Section 2.4. Namely, the application order is determined by the order of `--conf` command line options, and if one `--conf` option matches several files, their application order is determined by system locale.

The third *Jump* rule in Listing 34 is matching synthetic events with internal context SSH, submitting them to configuration file set *sshd2* for further processing (see Listing 37). This *Jump* rule is featuring a *TValue* pattern set to *True* that matches all input events without setting any match variables. If the *TValue* pattern is set to *False*, it does not match any events. Note that the synthetic events matched by the third *Jump* rule in Listing 34 rule originate from the ruleset in Listing 37, and the *Jump* rule directs these synthetic events back to the same ruleset for further processing.

Finally, similarly to other SEC rules, the *Jump* rule supports the *continue* keyword that allows to continue the search for other matching rules in the same configuration file after the configuration file set specified in the *Jump* rule has processed an input event (see Section 2.4 for more details). For example, if the first *Jump* rule in Listing 34 would have the *continue=TakeNext* setting, all *iptables* events would be passed to further rules after they have been processed by

Listing 36: The content of /etc/sec/sshd-rules1.sec

```
type=Options
procallin=no
joincfset=sshd1

type=SingleWithThreshold
ptype=RegExp
pattern=sshd\[\d+\]: Failed .+ for invalid user \S+ from ([\d.]+) port \d+ ssh2
desc=ip $1
action=pipe 'Repeated probing of non-existing accounts from $1' \
           mail root@localhost
thresh=10
window=300
```

Listing 37: The content of /etc/sec/sshd-rules2.sec

```
type=Options
procallin=no
joincfset=sshd2

type=PairWithWindow
ptype=RegExp
pattern=sshd\[\d+\]: Failed .+ for (\S+) from ([\d.]+) port \d+ ssh2
desc=user $1 ip $2
action=cevent SSH 0 SSH_PROBE_FROM_HOST_$2
ptype2=RegExp
pattern2=sshd\[\d+\]: Accepted .+ for $1 from $2 port \d+ ssh2
desc2=SSH login successful for %1 from %2
action2=logonly %s
window=120

type=SingleWithThreshold
ptype=RegExp
pattern=SSH_PROBE_FROM_HOST_([\d.]+)
desc=ip $1
action=pipe 'Repeated account probing from $1' mail root@localhost
thresh=3
window=300
```

the ruleset in Listing 35 (the second and third *Jump* rule would not be matching *iptables* events, though). Although the ruleset in Listing 34 does not use that approach, *continue* keywords can be employed for configuring the processing of the same input event by several *Jump* rules in the same configuration file.

Organizing rulesets in a hierarchical fashion as described in this section can greatly reduce the CPU time consumption, since it prevents irrelevant rules from being matched against input events. For example, as reported in [7], the CPU load decreased 4-5 times after the introduction of a hierarchical arrangement for a larger rulebase of 375 rules.

6.2 Using custom code in context expressions

In Section 5.3, we discussed rule examples for processing synthetic events from the *df* command. The rule examples from Listings 31 and 32 triggered email alerts if the disk usage reached 95..100%. For checking that the disk usage is at least 95%, a regular expression is used in the rule examples, but such checks are often much more convenient to implement with arithmetic match conditions.

For example, consider the following highly unusual SSH login failure that originates from a *privileged* port of a remote host (i.e., the port number is less than 1024):

```
Nov 20 13:19:38 myhost sshd[13477]: Failed password for charles
 from 192.168.56.1 port 80 ssh2
```

Although it is possible to write a regular expression for detecting such login failures from privileged ports, it is cumbersome and an arithmetic match condition $port < 1024$ would be more convenient. Listing 38 provides an example *Single* rule that illustrates the use of arithmetic match conditions in rule definitions.

Listing 38: The content of /etc/sec/sshd-priv-port.sec

```
type=Single
ptype=RegExp
pattern=sshd\[\d+\]: Failed .+ for (\S+) from ([\d.]+) port (\d+) ssh2
context=$3 -> ( sub { $_[0] < 1024 } )
desc=failed login attempt from privileged port
action=write - SSH login failure for $1 from privileged port $3 at $2
```

The example rule in Listing 38 employs a context expression that consists of one operand:

```
$3 -> ( sub { $_[0] < 1024 } )
```

That operand involves an execution of custom Perl code, and in the following the structure of this operand is discussed more closely.

The part of the operand that follows the arrow sign (->) and is enclosed in parentheses is a definition of an anonymous Perl function (subroutine):

```
sub { $_[0] < 1024 }
```

In Perl, the function definition always starts with the *sub* keyword. Normally, the function name would be given after *sub*, but since the above function is anonymous, the body of the function in curly braces ({ ... }) will immediately follow:

```
$_[0] < 1024
```

In the function body, the $_[0] variable denotes the first input parameter of the function. The function body contains just one statement that checks if the first input parameter is less than 1024. Since it is the only statement in the function, the result of the comparison becomes the return value of the function.

The return value of the function determines if the entire operand evaluates true or false. Obviously, the operand is true if and only if the value of the first input parameter is less than 1024.

Before the arrow sign (->), the list of input parameters for the function is given. Input parameters are separated by whitespace characters and in the case of the current example, the only input parameter is the $3 match variable. Since this match variable is set to the port number from the SSH login failure event, the context expression in Listing 38 checks if the port is a privileged one.

The evaluation of this context expression has an important aspect – the anonymous Perl function is *compiled* when SEC loads and parses the configuration file in Listing 38. When the context expression needs to be evaluated during further event processing, a fast previously compiled function body is called, passing the value of the $3 match variable to the compiled code.

Apart from arithmetic conditions, any custom Perl code can be used in context expressions, but this involves several aspects you should be aware of:

- Unlike SEC actions (e.g., *pipe* or *shellcmd*) that fork processes for asynchronous execution of commands, running custom Perl code involves *synchronous execution*. Therefore, if the Perl code you have written spends a lot of time on a complex task, SEC does not continue with other event processing activities *until your code completes*.

- If your code experiences a run time error (for example, division by zero), SEC traps such errors without crashing. However, you will get a warning message into the SEC log for letting you know that your code is not perfect.

- You have to remember that your custom code is part of the SEC process. For avoiding naming clashes between SEC data structures and your custom variables, your code is executed in a separate name space *main::SEC*. However, that still gives you an access to SEC data structures when you explicitly access the *main* name space, but doing that is recommended for *advanced users only*. Furthermore, since your code is part of the SEC process, calling *exit()* from your code terminates the entire SEC process.

The context expression in Listing 38 has only one operand, but you can combine it with other regular context name operands for creating larger context expressions. For example, consider the following context expression that involves the && operator (logical AND):

```
MYCONTEXT && $3 -> ( sub { $_[0] < 1024 } )
```

However, you have to be aware that the && (logical AND) and || (logical OR) operators are *short-circuiting*, and do not evaluate the second operand if the result has been established after evaluating the first operand. For example, if the context MYCONTEXT does not exist when the above context expression is evaluated, the custom Perl code in that context expression is *not* executed.

6.3 Using custom code in event matching patterns

Although regular expression patterns allow for matching many event formats, sometimes a regular expression is not sufficient for that purpose. To address such scenarios, SEC supports *PerlFunc* event matching patterns that are anonymous Perl functions. In the function body, the user can implement any event processing code that allows for much more powerful event matching than supported by a regular expression pattern.

Like anonymous functions discussed in the previous section, *PerlFunc* patterns are compiled when configuration files are loaded and parsed, and the resulting patterns are therefore very fast. Also, *PerlFunc* patterns feature all the aspects of executing anomymous functions discussed in the previous section (such as synchronous nature of the execution).

To illustrate the use of a *PerlFunc* pattern, Listing 39 provides a pattern that matches an SSH login failure originating from a privileged port. Note that since the anonymous Perl function covers more than one line, all lines apart from the last one are ending with backslash, so that they would be treated as a single line value for the *pattern* keyword.

Listing 39: The content of /etc/sec/sshd-priv-port2.sec

```
type=Single
ptype=PerlFunc
pattern=sub { my(%var); \
  if ($_[0] !~ /sshd\[\d+\]: Failed .+ for (\S+) from ([\d.]+) port (\d+) ssh2/) { \
    return 0; \
  } \
  if ($3 > 1023) { return 0; } \
  $var{"user"} = $1; $var{"ip"} = $2; $var{"port"} = $3; \
  return \%var; \
}
desc=failed login attempt from privileged port
action=write - SSH login failure for $+{user} from privileged port $+{port} at $+{ip}
```

The *PerlFunc* pattern in Listing 39 is designed for matching a single line input event (e.g., for matching an input event consisting of two lines, *PerlFunc2* pattern would be needed). By convention, the input event is passed to *PerlFunc* pattern as the first parameter of the function (i.e., the $_[0] variable in the function body).

In order to indicate that the pattern is *matching*, the function has to return either several values, or a single value that evaluates true in Perl Boolean context (a value that is not 0, empty string, or *undef*).

In order to indicate that the pattern is *not matching*, the function has to return either no values, or a single value that is false in Perl Boolean context (0, empty string, or *undef*). For that purpose, the function in Listing 39 is using the value of 0.

The function first checks if the input event matches the regular expression for SSH login failure and returns 0 if there is no match. If the regular expression matches, the $1, $2, and $3 match variables are set. The function then checks the port number and returns 0 if the number is greater than 1023 (i.e., the port is not a privileged one).

In the case of the successful regular expression match and privileged port, the function creates the keys *user*, *ip*, and *port* in the Perl hash (associative array) %var. The values of those keys are set to the user name (held by the $1 match variable), IP address of the remote host (held by the $2 match variable), and port number (held by the $3 match variable).

The function then returns a *reference* to the hash (denoted by \%var). If SEC receives a single value that is a reference to a hash, it creates match variables based on the keyword-value pairs in the hash: $+{user} that is set to user name, $+{ip} that is set to IP address, and $+{port} that is set to port number.

In addition, several match variables are created automatically: $0 that is set to the entire matching line, $+{_inputsrc} that is set to input file name, and $+{_intcontext} that is set to internal context of the input file (if the creation of internal contexts has not been activated on SEC command line, the $+{_intcontext} variable remains unset).

Finally, using custom code in SEC rules might be cumbersome if the code involves more than just a few lines. In such cases, it is recommended to put the custom code into a Perl module that is loaded when SEC starts up, so that custom code can be called through the module interface. Listing 40 illustrates this principle by loading the SecJson.pm module and calling the *json2matchvar()* function from this module in the *PerlFunc* pattern (i.e., the *PerlFunc* pattern acts as a wrapper for more complex code).

6.4 Using custom code in event group patterns

Consider the SSH login failure events from Listing 41, and suppose that the following event correlation task needs to be addressed – a warning email must be sent to *root@localhost* if 3 login failures are observed for the same user account within 60 seconds, so that these login failures are originating from 3 *different* SSH client hosts.

Without the requirement for 3 unique SSH client hosts, the third event occurring at 23:04:24 would have to trigger the email, since that event is the third login failure for the user bob within 60 seconds. However, the 3 login failures at 23:04:00, 23:04:13, and 23:04:24 involve only 2 SSH client hosts – 10.1.1.7 and 10.1.1.9. Therefore, when considering the requirement for SSH client hosts, the email warning should be issued at 23:04:47 when the fourth login failure from previously unseen SSH client host 10.1.1.2 is observed.

An example *EventGroup* rule in Listing 42 addresses the above event correlation task with the help of a *PerlFunc* event group pattern.

According to the rule's *desc* keyword, the rule starts a separate event correlation operation for each user name observed in SSH login failure events. Also, the rule features the *egtoken* field that configures a custom value for representing a matching event in the event group string. As discussed in Section 3.6, by default each event is represented by the number that reflects the number of the matching pattern in the rule definition. Because the rule from Listing 42 has

Listing 40: The content of /etc/sec/json.sec

```
# When SEC starts, load the SecJson.pm module (module is available at
# https://github.com/simple-evcorr/rulesets/tree/master/parsing-json)

type=Single
ptype=SubStr
pattern=SEC_STARTUP
context=SEC_INTERNAL_EVENT
desc=load SecJson.pm module
action=eval %o ( require '/etc/sec/perl/SecJson.pm' ); \
       if %o ( logonly Module SecJson.pm loaded ) \
        else ( logonly Failed to load SecJson.pm; eval %o exit(1) )

# When an event is observed which contains the @cee: substring,
# it is assumed that the string which follows @cee: is in JSON format
# and the rule attempts to parse it, storing the results into pattern
# match cache under the CEE entry. If the JSON string contains
# the field "test" (e.g., @cee: {"test":"abc","test2":"def",...),
# the value of this field is printed to standard output.

type=Single
ptype=PerlFunc
pattern=sub { if ($_[0] =~ /\@cee: (.+)/) { \
              return SecJson::json2matchvar($1); } return 0; }
varmap=CEE
desc=test JSON parsing
action=write - The value of test variable: $+{test}
```

Listing 41: Example sshd login failure events from /var/log/secure

```
Nov 24 23:04:00 test sshd[1137]: Failed password for bob from 10.1.1.7 port 32182 ssh2
Nov 24 23:04:13 test sshd[1145]: Failed password for bob from 10.1.1.9 port 42176 ssh2
Nov 24 23:04:24 test sshd[1212]: Failed password for bob from 10.1.1.7 port 34191 ssh2
Nov 24 23:04:47 test sshd[1226]: Failed password for bob from 10.1.1.2 port 18999 ssh2
```

Listing 42: The content of /etc/sec/sshd-3-unique-hosts.sec

```
type=EventGroup
ptype=RegExp
pattern=sshd\[\d+\]: Failed .+ for (\S+) from ([\d.]+) port \d+ ssh2
desc=user $1
egtoken=$2
egptype=PerlFunc
egpattern=sub { my(%hosts) = map { $_ => 1 } @{$_[1]}; \
                return scalar(keys %hosts) >= 3; }
action=pipe 'SSH login failures from 3 different hosts for user $1' \
           mail root@localhost
window=60
thresh=3
```

only one regular expression pattern (given with the *pattern* keyword), the event group string would have the following format without the *egtoken* keyword:

```
1 1 ... 1
```

However, since the *egtoken* keyword has the value $2, the event correlation operations started by the rule from Listing 42 build event group strings from *IP addresses of SSH client hosts* extracted from matching events.

As discussed in Section 3.2, event correlation operations generally substitute match variables *only once* when the operation is initialized. The value set by the *egtoken* field is one notable exception from that rule – if the value contains match variables, they are substituted for *every event* that the operation is processing.

When the rule in Listing 42 processes example events from Listing 41, all events are processed by one event correlation operation with the ID ⟨/etc/sec/sshd-3-unique-hosts.sec, 0, user bob⟩. As the operation is processing these 4 events, the event group string is updated as follows:

```
10.1.1.7
10.1.1.7 10.1.1.9
10.1.1.7 10.1.1.9 10.1.1.7
10.1.1.7 10.1.1.9 10.1.1.7 10.1.1.2
```

When the third event at 23:04:24 appears, the numeric threshold condition set by the *thresh* keyword becomes satisfied, and therefore the event group string is matched by the *PerlFunc* pattern provided with the *egpattern* keyword. The anonymous Perl function given with the *egpattern* keyword receives the following event group string as its first parameter:

```
10.1.1.7 10.1.1.9 10.1.1.7
```

In addition, the second parameter ($_[1]) provides a *reference* to the event group string the list format[2]:

```
(10.1.1.7, 10.1.1.9, 10.1.1.7)
```

In the function body, @{$_[1]} represents the list referenced by $_[1], and the list is processed by Perl builtin *map* subroutine:

```
map { $_ => 1 } @{$_[1]}
```

The above *map* call creates a Perl hash from the list, assigning the result to the hash %hosts. Unique list members become keys in the hash, with all keys having the value of 1. Therefore, from the list submitted to the function at 23:04:24, the following Perl hash is created and assigned to %hosts:

```
10.1.1.7 => 1
10.1.1.9 => 1
```

After creating %hosts, the following statement extracts all keys from %hosts and finds their number:

[2]Passing a reference to the function is significantly cheaper than passing the entire list which could potentially be very large.

```
scalar(keys %hosts)
```

The function returns true if %hosts has at least 3 keys, and false otherwise. In other words, the anomymous function provided with the *egpattern* keyword returns true if and only if the event group string (and its list representation) contain at least 3 unique SSH client host IP addresses[3].

Obviously, when the third event from Listing 41 appears at 23:04:24, the *PerlFunc* event group pattern does not produce a match. On the other hand, the arrival of the fourth event at 23:04:47 produces a match, and the event correlation operation sends the following email to *root@localhost*: *SSH login failures from 3 different hosts for user bob*.

Finally, note that the event correlation scenario described in this section can also be addressed by combining the *EventGroup* rule with contexts, and relevant rule examples have been provided in [7] and SEC official documentation.

7 Conclusion

In this tutorial, we have looked into the essentials of SEC and discussed the most commonly used rule types, event correlation operations, contexts, and synthetic events. We have also had a look into some advanced topics like building hierarchical rulebases and using custom code in rules.

While this tutorial might seem like a large document, we have only covered a relatively small part of SEC functionality. There are many more things that did not fit into this tutorial due to space limitations. However, the essentials you have learned should be sufficient for tackling more advanced event correlation tasks on your own.

As we mentioned in the very beginning of this tutorial – it does not aim to be a replacement for the official documentation (SEC man page). You are therefore encouraged to consult the official documentation as you develop your own event correlation rulesets.

References

[1] Gabriel Jakobson and Mark Weissman, "Real-time telecommunication network management: Extending event correlation with temporal constraints," Proceedings of the 1995 IEEE International Symposium on Integrated Network Management, pp. 290-301, 1995

[2] Risto Vaarandi, "SEC – a Lightweight Event Correlation Tool," Proceedings of the 2002 IEEE Workshop on IP Operations and Management, pp. 111-115, 2002

[3] John P. Rouillard, "Real-time Logfile Analysis Using the Simple Event Correlator (SEC)," Proceedings of the 2004 USENIX Large Installation System Administration Conference, pp. 133-149, 2004.

[3] Since the event correlation operation executes its action list only once, we could replace the >= operator with == in the function. In that case, the event group pattern evaluates true if and only if the event group string contains *exactly* 3 unique IP addresses.

[4] Risto Vaarandi, "Simple Event Correlator for real-time security log monitoring," Hakin9 Magazine 1/2006 (6), pp. 28-39, 2006.

[5] Risto Vaarandi and Michael R. Grimaila, "Security Event Processing with Simple Event Correlator," Information Systems Security Association (ISSA) Journal 10(8), pp. 30-37, 2012

[6] David Lang, "Using SEC," USENIX ;login: Magazine 38(6), pp. 38–43, 2013

[7] Risto Vaarandi, Bernhards Blumbergs and Emin Çalişkan, "Simple Event Correlator – Best Practices for Creating Scalable Configurations," Proceedings of the 2015 IEEE CogSIMA Conference, pp. 96-100, 2015

[8] https://en.wikipedia.org/wiki/Daemon_(computing)