Security Event Processing with Simple Event Correlator

By Risto Vaarandi and Michael R. Grimaila – ISSA member, Dayton, USA Chapter

This paper focuses on Simple Event Correlator – a lightweight event correlator written by one of the authors which is based on different design principles than commercial solutions. We will present an overview of SEC and discuss some real-life event correlation scenarios which highlight its capabilities.

In the early days of computer management, event logs were primarily used for diagnosing when an application or device stopped functioning properly. Event logs provided visibility of the internal state of the program or system to aid in debugging (Sah, 2002). In 1980, Anderson (1980) recognized the value of post-processing event logs to detect unauthorized access to files. Event correlation rose into prominence in early 1990s when it was employed for real-time processing of large volumes of network events to provide the human operator with a concise picture of current fault conditions present in the network (Grimaila et al., 2011). Event correlation provides the capability for real-time interpretation of an event stream where a new meaning is assigned to a group of events which occur in a predefined time window (Jakobson and Weissman, 1995).

For example, event correlation has been used for making sense of router `linkDown` and `linkUp` SNMP traps. In many networks most `linkDown` traps are immediately followed by `linkUp`, representing a very short network link outage. After such accidental outage has been seen, the link could operate flawlessly for many months. In large networks, thousands of `linkDown` traps might arrive from routers in a short time frame, with only a few of them manifesting hard errors which require human intervention. In order to detect relevant information from large volumes of such traps, many ISPs employ the following event correlation scheme:

1. If `linkDown` for a link is not followed by `linkUp` for this link within T1 seconds (e.g., T1=10), send an alarm to human operator about the broken link; otherwise generate an artificial event "Link Short Outage" inside the event correlator.
2. If N events "Link Short Outage" have been seen for a link within T2 seconds (e.g., N=3 and T2=3600), send an alarm to human operator about the degrading quality of this link.

This sample scheme illustrates three important aspects of event correlation. First, one of the main purposes of event correlation is to reduce large volumes of input events to a smaller set of more meaningful output events in real time. Second, some results from event correlation might not be reported as output, but rather treated as intermediate knowledge, which is used for further knowledge deriving inside the correlator (e.g., "Link Short Outage" events from the above example). Third, event correlation is not only about processing existing events, but it is equally important to detect that events do not appear when expected (e.g., no linkUp will occur after linkDown). It should also be noted that instead of producing an output alarm for a human, the event correlator might take another kind of action for output (e.g., reboot a network device).

Today, event correlation is not only used for network management, but it is employed in many other domains, including Security Information and Event Management (SIEM). SIEM encompasses all of the activities surrounding the collection, logging, and analysis of system and application events to identify potentially malicious activities and system errors (Swift, 2006). Most SIEM products intrinsically support event correlation, although the set of features and performance could vary significantly. Unfortunately, today's commercial event correlation solutions are often part of a larger event management framework (e.g., HP OpenView, Tivoli, ArcSight, RSA enVision). Therefore, their deployment requires a substan-

tial investment and the installation of the whole framework which is costly, complex, and a time-consuming procedure.

In this paper, we will focus on Simple Event Correlator (SEC) – a lightweight event correlator written by one of the authors, which is based on different design principles than commercial solutions. Unlike most other products, SEC is not a part of a heavyweight and expensive event management framework, but is rather an open-source UNIX tool which can be easily integrated into any setup. During the last decade, SEC has been used for a wide variety of purposes, including network fault management, processing of various security events (e.g., IDS and firewall messages), system and application monitoring, and fraud detection (Becklehimer et al., 2007; Grimaila et al., 2011; Myers et al., 2011; Rouillard, 2004; Vaarandi 2006; Vaarandi and Podins, 2010). In the remainder of this paper, we will present an overview of SEC and discuss some real-life event correlation scenarios which highlight its capabilities.

Overview of SEC

Simple Event Correlator[1] (SEC) is a compact tool for accomplishing various event correlation tasks. It runs as a single process, requires no graphical environment, and has moderate CPU and memory consumption. It is written in Perl and works on any UNIX platform with standard Perl distribution, without dependencies on any other software (e.g., non-standard Perl modules or other UNIX utilities). It has also been used on Windows systems, but requires the addition of ActivePERL (Active State, 2012) or CygWin Perl (CygWin, 2012). Although you can install SEC from source tarball in a straightforward way, it has been packaged for several major Linux and BSD distributions, and can thus be also installed from standard software repository on these platforms.

SEC accepts its input from one or more log files, and can produce its output by executing custom command lines, writing to files, and by various other means. Event correlation configuration is specified as rules which are stored in text files. Rules are applied to input events in the order they are defined in the configuration file. If there are two or more configuration files, rule sequences from all files are applied to input (unless explicitly specified otherwise). Most rule definitions can have the following parts:

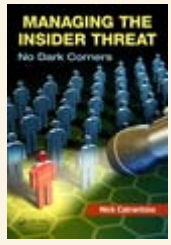
- Event matching pattern
- Boolean context expression
- Operation description string
- Event correlation information (e.g., size of event correlation window)
- Action(s) for producing output (e.g., output events or intermediate knowledge)

Regular expressions, search strings, custom Perl subroutines, and truth values can be employed for event matching patterns. In addition, results from pattern matching can be

1 Simple Event Correlator – http://simple-evcorr.sourceforge.net.

ISSA Members Receive a 50% Discount on Information Security Related Books

ISSA has arranged for a **50% discount on Auerbach Publications and CRC Press information security books**. Choose from hundreds of books on a variety of subject! Visit the Special Offers tab for details on this and other great offers (member login required). Titles include:



K13375 Managing the Insider Threat: No Dark Corners

An adversary who attacks an organization from within can prove fatal to the organization and is generally impervious to conventional defenses. The first comprehensive resource to use social science research to explain why traditional methods fail against these trust betrayers, this groundbreaking book identifies new management, security, and workplace strategies for categorizing and defeating insider threats. Each chapter offers questions to stimulate discussion and exercises or problems suitable for team projects. This practical text enables those charged with protecting an organization from internal threats to stop these predators before they jeopardize the workplace and sabotage business operations.

K13576 Electronically Stored Information: The Complete Guide to Management, Understanding, Acquisition, Storage, Search, and Retrieval

Using easy-to-understand language, the book explains exactly what electronic information is, the different ways it can be stored, why we need to manage it from a legal and organizational perspective, who is likely to control it, and how it can and should be acquired to meet legal and managerial goals. Its reader-friendly format means you can read it cover to cover or use it as a reference where you can go straight to the information you need. Complete with links and references to additional information, technical software solutions, helpful forms, and time-saving guides, it provides you with the tools to manage the increasingly complex world of electronic information that permeates every part of our world.

K10743 The 7 Qualities of Highly Secure Software

Providing a framework for designing, developing, and deploying hack-resilient software, this book uses engaging anecdotes and analogies—from Aesop's fables and athletics to architecture and video games—to illustrate the qualities needed for the development of highly secure software. Each chapter details one of the seven qualities that make software less susceptible to hacker threats. Filled with real-world examples, the book explains complex security concepts in language that's easy to understand to supply readers with the understanding needed to building secure software.

cached, in order to reuse them at later rules. All SEC patterns can be extended for multi-line matching, in order to monitor log files with messages spanning over several lines. When an input event matches a rule, SEC will check if there is already an event correlation operation running for this event (operation description string is used for identifying the operation). If the operation exists, it will receive the event for correlation; otherwise, SEC will start a new operation which will then get the event. A rule could start many operations that are running simultaneously, while each operation has only one parent rule that started it.

In the following sections we will have a look into some real-life event correlation scenarios.

Monitoring SSH login failures and blocking suspicious hosts

In this section, we will consider the monitoring of SSH login failures on a Linux platform, and blocking hosts from which a number of failed login attempts have been seen in a short time frame. Most commonly used Linux distributions are using the OpenSSH server which normally logs its events through the syslog daemon.

The following example is a successful login event for user *risto* with a public key authentication, with the connection coming from *10.16.18.20*:

```
Jun 15 16:17:57 myhost sshd[2891]: Accepted
publickey for risto from 10.16.18.20 port
42174 ssh2
```

The following example depicts a password login failure event for user risto, with the login attempt coming from 10.16.96.31:

```
Jun 15 16:18:25 myhost sshd[2899]: Failed
password for risto from 10.16.96.31 port
12801 ssh2
```

If a username is tried, which is not present in the local user base, the event looks slightly different from the regular login failure. The following example represents a password login attempt from host *10.16.18.21* for the non-existing user bob:

```
Jun 15 17:02:02 myhost sshd[3046]: Failed
password for invalid user bob from
10.16.18.21 port 43183 ssh2
```

If the local syslog daemon logs the above events to `/var/log/secure`, SEC could be started with the following command line for correlating them:

```
/usr/bin/sec --conf=/etc/sec/ssh-login-
failure.rules --input=/var/log/secure --log=/
var/log/sec --detach
```

The `--conf` option tells SEC to read correlation rules from `/etc/sec/ssh-login-failure.rules` and to open `/var/log/secure` at the end-of-file, in order to process all lines which will be appended to the log. The `--log` option specifies the location of SEC's own log and `--detach` switches SEC into daemon mode. Suppose the file `/etc/sec/ssh-login-failure.rules` contains the following three rules (see figure 1).

The PairWithWindow rule matches any SSH login failure for a valid username with the regular expression `sshd\[\d+\]: Failed \S+ for (\S+) from ([\d.]+) port \d+ ssh2`, and sets $1 match variable to username and $2 to the IP address of remote host. For the sample login failure event above, $1 would be set to risto and $2 to 10.36.96.31. After that, the operation description string given with the 'desc' field is eval-

```
type=PairWithWindow
ptype=RegExp
pattern=sshd\[\d+\]: Failed \S+ for (\S+) from ([\d.]+) port \d+ ssh2
desc=User $1 has been unable to log in from $2 over SSH during 30s
action=event SSH_LOGIN_FAILURE_USER_$1_HOST_$2
ptype2=RegExp
pattern2=sshd\[\d+\]: Accepted \S+ for $1 from $2 port \d+ ssh2
desc2=SSH login successful for %1 from %2 after initial failure
action2=logonly
window=30

type=Single
ptype=RegExp
pattern=sshd\[\d+\]: Failed \S+ for invalid user (\S+) from ([\d.]+) port \d+ ssh2
desc=Login attempt for invalid user $1 from $2 over SSH
action=event SSH_LOGIN_FAILURE_USER_$1_HOST_$2

type=SingleWithThreshold
ptype=RegExp
pattern=SSH_LOGIN_FAILURE_USER_\S+_HOST_([\d.]+)
context=!NETFILTER_IP_$1_BLOCKED
desc=Repeated user probing from host $1
action=shellcmd /sbin/iptables -I INPUT -s $1 -j DROP; \
   create NETFILTER_IP_$1_BLOCKED 300 shellcmd /sbin/iptables -D INPUT -s $1 -j DROP
window=120
thresh=3
```

Rule 1 – Monitoring SSH login failures and blocking suspicious hosts

uated, which yields *User risto has been unable to log in from 10.36.96.31 over SSH during 30s*. This string is used for building the operation ID, which also contains the rule file name (`/etc/sec/ssh-login-failure.rules`) and the rule number (`0`, since the rule is the first rule in the file). The presence of the rule file name and rule number in the ID ensures that operations from different rules will never clash. Also, since the operation description string contains both the username and IP address, a separate PairWithWindow operation is started for each distinct username-IPaddress pair.

Each PairWithWindow operation will run for 30 seconds and wait for an event which would match a regular expression given with the `'pattern2'` field. All match variables within `'pattern2'` will be replaced with their values for the given operation, yielding `sshd\[\d+\]: Accepted \S+ for risto from 10\.36\.96\.31 port \d+ ssh2` for the example event. Note that special symbols in variable values (e.g., dots in IP addresses) are escaped with backslash before substitution. In other words, the operation expects the successful login event for user risto from 10.36.96.31 during the next 30 seconds. If other login failure events are seen for the same username and IP address within this time frame, they are silently consumed by the operation.

If the successful login event appears within 30 seconds and is matched by the above regular expression, the operation will evaluate the `'desc2'` field, which yields *SSH login successful for risto from 10.36.96.31 after initial failure* (%1 and %2 match variables hold the values from the first regular expression match). Then the logonly action given with `'action2'` field is executed which writes the string *SSH login successful for risto from 10.36.96.31 after initial failure* into SEC's own log `/var/log/sec` for debugging purposes.

If the successful login event does not appear within 30 seconds after the failure, the operation executes the event action, given with the `'action'` field. For example login failure scenario for risto from 10.36.96.31, the action produces a new event `SSH_LOGIN_FAILURE_USER_risto_HOST_10.36.96.31`, which is treated like a line read from `/var/log/secure`. This so-called synthetic event represents the fact that a login failure condition has not been quickly resolved, and we are not dealing with an accidental error (e.g., one-time typo in a password).

This event will match the regular expression of the SingleWithThreshold rule which implements counting for such events. Note that some field definitions of this rule span over several lines, with a backslash at the end of line continuing the field definition in the next line. Since the `'desc'` field of this rule contains the `$1` match variable which holds the IP address from matching events, a separate counting operation is started for each distinct IP address. If three events (the value of the `'thresh'` field) have been seen during 120 seconds (the value of the `'window'` field) for the same IP, the operation executes an action list given with the `'action'` field. After the execution, the operation consumes further matching events without further action until the event correlation window expires. The event correlation window of the SingleWithThreshold operation is sliding – if less than three matching events have been observed during 120 seconds, the beginning of the window is moved forward in order to drop events older than 120 seconds. If no events remain in the window after sliding, the operation terminates silently.

The `'action'` field specifies a list of shellcmd and create actions separated with a semicolon. If three non-accidental login failures have been seen from 10.36.96.31 within 2 minutes, shellcmd forks a separate process for executing commandline `/sbin/iptables -I INPUT -s 10.36.96.31 -j DROP`. This adds a rule to local Netfilter firewall for blocking all further network traffic from 10.36.96.31. In order to avoid blocking the host forever and prevent firewall rule duplicates, a context `NETFILTER_IP_10.36.96.31_BLOCKED` is created with the create action. This context represents the fact that the given IP address is currently blocked by the local firewall. Unlike synthetic events, the contexts do not match regular expressions, but their presence can rather be checked from Boolean context expressions. In the case of the above example, the SingleWithThreshold rule is implementing this with the `'context'` field, verifying that the context does not exist for the IP address that was obtained from the regular expression match (the ! operator denotes logical NOT). If the context is there, the rule is not considered matching, despite the regular expression match. Therefore, this rule is essentially disabling itself for IP addresses which have been already blocked.

Apart from the context name, the create action has two additional parameters – the context lifetime (300 seconds) and the action list which should be executed when the context expires. In the case of our example, the command line `/sbin/iptables -D INPUT -s 10.36.96.31 -j DROP` is executed when `NETFILTER_IP_10.36.96.31_BLOCKED` context expires, which removes the blocking rule for host 10.36.96.31. In other words, offending hosts are blocked for 5 minutes only.

Finally, the task of the Single rule is to react to login failures for non-existing users, and immediately create input events for the following SingleWithThreshold rule, since no successful login event can follow the failure for non-existing user. Note that unlike other two rules, Single does not start an event correlation operation, but rather executes an action immediately.

Cross-correlating offending events from Netfilter, SSH, and Apache logs

Some probing activities against the system are visible in several logs; for example, there could be entries both in the firewall and application logs. Also, apart from coming from different sources and having a different format, these events often occur in arbitrary order, which further complicates their correlation. In this section, we will discuss an example for detecting an unordered group of simultaneous offending events from the Netfilter firewall, OpenSSH daemon, and Apache web server on a Linux platform. Suppose we want to detect SSH login failure events described in the previous sec-

Figure 2 – Cross-correlating offending events from Netfilter, SSH, and Apache logs

```
type=EventGroup3
ptype=RegExp
pattern=sshd\[\d+\]: Failed \S+ for (?:invalid user )?\S+ from ([\d.]+) port \d+ ssh2
thresh=3
ptype2=RegExp
pattern2=^([\d.]+) \S+ \S+ \[[^]]+\] "[^"]+ HTTP\/[\d.]+" 4\d+ \d+
thresh2=1
ptype3=RegExp
pattern3=kernel: IN=\S+ OUT= MAC=\S+ SRC=([\d.]+)
thresh3=5
desc=Repeated probing from $1
action=pipe 'Repeated probing from host $1' /bin/mail root@localhost
window=120
```

tion, but also invalid web page requests from Apache access logs with error codes 4xx. The following example event from /var/log/httpd/access_log represents a request for a non-existing URL /aaa.cgi which produces error code 404:

```
10.1.6.22 - - [21/Jun/2012:16:02:04 +0300] "GET
/aaa.cgi HTTP/1.1" 404 285 "-" "Mozilla/5.0
(X11; U; Linux x86_64; en-US; rv:1.9.2.24)
Gecko/20111108 Fedora/3.6.24-1.fc14
Firefox/3.6.24"
```

Also, suppose the local Netfilter firewall has been configured to log a syslog event for each blocked packet. For example, the following event from /var/log/messages depicts a blocked access from 10.26.96.19 to the local mail server (port 25/tcp):

```
Jun 21 16:42:57 myhost kernel: IN=eth0 OUT=
MAC=xx:xx:xx:xx:xx:xx:xx:xx:xx:xx:xx:xx:xx:xx
SRC=10.26.96.19 DST=10.1.6.22 LEN=60 TOS=0x00
PREC=0x00 TTL=63 ID=15040 DF PROTO=TCP
SPT=40217 DPT=25 WINDOW=5840 RES=0x00 SYN
URGP=0
```

In order to monitor these events, SEC could be started with the following commandline:

```
/usr/bin/sec --conf=/etc/sec/cross-corr.
rules --input=/var/log/messages --input=/var/
log/secure --input=/var/log/httpd/access_log
--detach
```

Suppose the /etc/sec/cross-corr.rules contains the following rule (see figure 2).

This rule implements cross-correlation for three different event types within the common window of 120 seconds. Events are matched with three different regular expressions, while each expression extracts the IP address of offending host from the matching event and assigns it to $1 match variable. Since the 'desc' field of the rule contains $1, for each offending host a separate event correlation operation is started. The regular expression given with the 'pattern' field matches SSH login failures and the 'thresh' field sets a threshold for these events to 3. The expression given with the 'pattern2' field matches Apache access log 4xx error messages with threshold 1. Finally, the 'pattern3' defines an expression for Netfilter firewall events with threshold 5. Each running operation expects matching events for one IP address, and maintains three counters for aforementioned three event types. After the arrival of each event, the relevant counter is incremented and three threshold conditions are evaluated. If all conditions are satisfied, a warning email with a text 'Repeated probing from host X' is sent to the local root-user with the pipe action, which executes /bin/mail utility and feeds the text to its standard input. Note that like with the SingleWithThreshold example from previous section, the operation silently consumes further matching events for host X after the output action has been triggered for X. Also, if threshold conditions have not been met after 120 seconds, the event correlation window will slide forward.

Integrating Perl code into SEC rules for custom Netfilter event correlation

Although existing event correlation products allow for tackling a wide variety of event processing tasks, sometimes standard features of the product come short and external shell scripts or programs have to be integrated into event correlation schemes. Unfortunately, invoking external tools involves the creation of new processes which can be costly if external tools are used very frequently. Furthermore, it is often not easy to share data between the event correlation engine and another process.

Fortunately, SEC not only supports the use of external tools, but also the employment of custom Perl code in several flexible ways. First, the user can include Perl code snippets (e.g., a condition '$1 < 1024') in rule definitions which are compiled before each execution. Furthermore, the user can enclose custom code into Perl functions which are compiled once at SEC startup. During SEC runtime, compiled functions are invoked through code pointers, and custom data can be passed to functions through input parameters. It is important to note that such compiled functions are running at the speed of SEC itself and they can share custom data structures of arbitrary complexity. The user can employ these functions as event matching patterns, as operands of context expressions, and also as actions for efficiently extending the power of SEC rules.

The following example rule set has been defined for correlating Netfilter firewall events. The purpose of this rule set is to send a warning email to the local root-user if within 60 seconds a host has probed either 10 distinct privileged ports,

20 distinct TCP ports, or 40 distinct UDP ports which are all firewalled (see figure 3).

Since three threshold conditions are joined with logical OR, and the threshold for privileged ports covers both TCP and UDP protocols, this problem can not be addressed with the EventGroup rule. In order to accomplish this task, Perl hash table called hosts is used for memorizing port probes from offending IP addresses. For each IP address, this table contains references to two additional hash tables for TCP and UDP port numbers. For evaluating a threshold condition, the number elements in relevant hash table(s) is found.

In order to process Netfilter events, three Single rules are used. Since the regular expressions for those rules would be identical, the match from the first expression is cached for further reuse by the following rules, in order to save CPU time. After the expression `kernel: IN=\S+ OUT= MAC=\S+ SRC=([\d.]+) .* PROTO=(TCP|UDP) SPT=\d+ DPT=(\d+)` has matched, the `'varmap'` field of the first rule creates a match

```
type=Single
ptype=RegExp
pattern=kernel: IN=\S+ OUT= MAC=\S+ SRC=([\d.]+) .* PROTO=(TCP|UDP) SPT=\d+ DPT=(\d+)
varmap=netfilter; ip=1; proto=2; port=3
context=!NETFILTER_COUNTING_$+{ip} && !NETFILTER_COUNTING_OFF_$+{ip}
desc=Create data structures for host $+{ip}
action=create NETFILTER_COUNTING_$+{ip} 60 lcall %o $+{ip} -> ( sub { delete $hosts{$_[0]}; } ); \
       lcall %o $+{ip} $+{proto} $+{port} -> \
         ( sub { my($ip) = $_[0]; my($proto) = $_[1]; my($port) = $_[2]; \
                 $hosts{$ip} = { "TCP" => {}, "UDP" => {} }; \
                 $hosts{$ip}->{$proto}->{$port} = time(); } )
type=Single
ptype=Cached
pattern=netfilter
context=!NETFILTER_COUNTING_OFF_$+{ip}
continue=TakeNext
desc=Count netfilter event for host $+{ip}
action=set NETFILTER_COUNTING_$+{ip} 60; \
       lcall %o $+{ip} $+{proto} $+{port} -> \
         ( sub { my($ip) = $_[0]; my($proto) = $_[1]; my($port) = $_[2]; \
                 $hosts{$ip}->{$proto}->{$port} = time(); } )
type=Single
ptype=Cached
pattern=netfilter
context=!NETFILTER_COUNTING_OFF_$+{ip} && \
        $+{ip} -> ( sub { my($ip) = $_[0]; my($port); \
                    my($priv) = 0; my($time) = time(); \
                    foreach $port (keys %{$hosts{$ip}->{"TCP"}}) { \
                      if ($time - $hosts{$ip}->{"TCP"}->{$port} > 60) \
                        { delete $hosts{$ip}->{"TCP"}->{$port}; } \
                      elsif ($port < 1024) { ++$priv; } \
                    } \
                    foreach $port (keys %{$hosts{$ip}->{"UDP"}}) { \
                      if ($time - $hosts{$ip}->{"UDP"}->{$port} > 60) \
                        { delete $hosts{$ip}->{"UDP"}->{$port}; } \
                      elsif ($port < 1024) { ++$priv; } \
                    } \
                    return ($priv == 10 || \
                            scalar(keys %{$hosts{$ip}->{"TCP"}}) == 20 || \
                            scalar(keys %{$hosts{$ip}->{"UDP"}}) == 40); } )
desc=Host $+{ip} has accessed 10 privileged, 20 TCP or 40 UDP ports
action=lcall %ports $+{ip} -> ( sub { my($ip) = $_[0]; \
                     my($tcp) = join(" ", keys %{$hosts{$ip}->{"TCP"}}); \
                     my($udp) = join(" ", keys %{$hosts{$ip}->{"UDP"}}); \
                     return "TCP: $tcp\nUDP: $udp"; } ); \
       pipe '%ports' /bin/mail -s 'Portscan from $+{ip}' root@localhost; \
       create NETFILTER_COUNTING_OFF_$+{ip} 300; \
       obsolete NETFILTER_COUNTING_$+{ip}
```

Figure 3 – Netfiler firewall rule set

cache entry netfilter and stores match variables $1, $2 and $3 under this name. For the purposes of readability, 'varmap' field creates additional match variables $+{ip}, $+{proto} and $+{port} which are synonyms for $1, $2 and $3, respectively. The second and third rule are not re-evaluating the same expression again, but are rather performing a cache lookup for the name netfilter.

Apart from Perl hash tables, the rules are using contexts for event counting purposes. The NETFILTER_COUNTING_X context (created by the first rule) indicates that event counting is currently ongoing for IP address X. The context lifetime is 60 seconds, and when it expires, the hash tables for X are deleted. The deletion is done with the lcall action, which passes $+{ip} match variable to the previously compiled Perl function sub { delete $hosts{$_[0]}; }. This function drops all hash tables for the IP address held by the input parameter, and the function return value is assigned to the %o action list variable. The NETFILTER_COUNTING_OFF_X context (created by the third rule) indicates that a warning email has already been sent for IP address X, and further warnings should not be issued for X.

The first rule initializes data structures for the IP address X if the context expression:

```
!NETFILTER_COUNTING_X && !NETFILTER_
COUNTING_OFF_X
```

evaluates true (! and && denote logical NOT and AND, respectively). In other words, the expression is true if event counting is not ongoing and previous warning has not been sent for X. If this is the case, the rule sets up the NETFILTER_COUNTING_X context. Data structures are then initialized with the lcall action which passes match variables $+{ip}, $+{proto} and $+{port} to a Perl function as input parameters. After creating hash tables for X, an entry for the probed local port number is stored to the relevant table. For example, if remote host 10.1.1.1 has tried to access local port 25/tcp, the entry 10.1.1.1 would be created in the hosts hash table which points to hash tables for TCP and UDP traffic from 10.1.1.1. In the TCP table for 10.1.1.1; the entry 25 would be created, which holds the port access time in seconds since Epoch (as returned by the Perl time() function).

The second rule handles further events for already tracked IP addresses and updates their hash tables with the lcall action. If an already accessed port is probed, its access time is updated, otherwise a new entry for the port number is created. In addition, the lifetime of the NETFILTER_COUNTING_X context is extended for 60 seconds with the set action which ensures that Perl hash tables for IP address X will exist during the next minute. Also note that the second rule has the 'continue' field set to TakeNext which passes every matching event to the following rule for further processing (the default is not to pass matching events to following rules).

The third rule employs Perl functions both in the 'context' and 'action' fields. The 'context' field verifies that the context NETFILTER_COUNTING_OFF_X does not exist and that the Perl function (second operand of logical AND) returns true in the boolean context. The function first inspects the TCP and UDP hash tables for the IP address X, and removes port number entries with access times older than 60 seconds. It will then find the number of remaining entries for both hash tables, and also set the $priv variable to the number of privileged ports in both tables. The function will return TRUE if either 10 privileged ports were found, TCP table contained 20 entries, or UDP table contained 40 entries. If the context expression evaluates true, the port numbers from both hash tables are extracted with the lcall action, and assigned as a two-line string to the %ports action list variable (the first line lists ports for TCP and the second for UDP). The content of the %ports variable is then mailed to the local root-user with the pipe action, and the NETFILTER_COUNTING_OFF_X context is set up for suppressing further mails for address X during 5 minutes. Also, the lifetime of the NETFILTER_COUNTING_X context is expired with the obsolete action which deletes all hash tables for address X from memory.

Conclusion

In this article, we briefly reviewed an open-source and lightweight event correlation tool, SEC, and provided some rule set examples which illustrate the capabilities of this tool. SEC has become a widely used event correlating utility during the last decade, and is employed by many institutions in academia, government, financial sector, telecom sector, and other industries. SEC is licensed under the terms of GNU GPL and can be freely downloaded.[2] In addition, the website contains links to useful information and tutorials that will help you exploit the power of SEC for event correlation.

Disclaimer

The views expressed in this paper are those of the authors and do not reflect the official policy or position of CCDCOE, NATO, the United States Air Force, the Department of Defense, or the US Government.

References

Active State (2012). Active PERL, Available as of 23 June 2012 at http://www.activestate.com/activeperl.

Anderson, J. P. (1980). Computer Security Threat Monitoring and Surveillance, James P. Anderson Co., Fort Washington, PA.

Becklehimer, J., Willis, C., Lothian, J., Maxwell, D., and Vasil, D. (2007). Real Time Health Monitoring of the Cray XT3/XT4 Using the Simple Event Correlator (SEC). Proceedings of the 49th Cray User Group Conference.

CygWin (2012). Cygwin Perl, Available as of 23 June 2012 at http://www.cygwin.com.

Grimaila, M.R., Myers, J., Mills, R.F., and Peterson, G. (2011), Design and Analysis of a Dynamically Configured Log-based Distributed Security Event Detection Methodology, The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology, DOI: 10.1177/1548512911399303, pp. 1-23, March 2011.

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

Jakobson, G., and Weissman, M. (1995). Real-time telecommunication network management: Extending event correlation with temporal constraints, Proceedings of the 4th International Symposium on Integrated Network Management, pp. 290-301, 1995

Myers, J., Grimaila, M.R., and Mills, R.F. (2011). Log-Based Distributed Security Event Detection Using Simple Event Correlator. Proceedings of the 44th Hawaii International Conference on System Sciences, pp. 1-7.

Rouillard, J.P. (2004). Real-time Logfile Analysis Using the Simple Event Correlator (SEC). Proceedings of the USENIX 18th System Administration Conference, pp. 133-149.

Sah, A. (2002). "A New Architecture for Managing Enterprise Log Data," LISA, pp. 121–132.

Swift, D. (2006), "A Practical Application of SIM/SEM/SIEM Automating Threat Identification," Technical report, SANS Institute, December 23, 2006.

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

Vaarandi, R., and Podins, K. (2010). Network IDS Alert Classification with Frequent Itemset Mining and Data Clustering, Proceedings of the 6th Conference on Network and Service Management, pp. 451-456.

About the Authors

Dr. Risto Vaarandi received his PhD in Computer Engineering from the Tallinn University of Technology, Estonia, in June 2005. Since 2006, he has been working as a scientist in NATO Cooperative Cyber Defence Centre of Excellence. He is the creator of Simple Event Correlator, an open-source event correlation tool used widely in the security community. Risto's research interests include event correlation, system and network monitoring technologies, data mining, and cyber security. He may be reached at risto.vaarandi@gmail.com.

Dr. Michael R. Grimaila, CISM, CISSP, is a Professor of Systems Engineering and a member of the Center for Cyberspace Research at the Air Force Institute of Technology (AFIT), Wright-Patterson AFB, Ohio USA. He is a member of the ACM, a Senior Member of the IEEE, and a Fellow of the ISSA. Dr. Grimaila's research interests include quantum cryptography, mission assurance, network management and security, and systems engineering. He may be reached at michaelgrimaila@yahoo.com.

Implementing Least Privilege for Interconnected, Agile SOAs/Clouds

Continued from page 23.

matically. It is "Open" because it is based on open standards where possible (e.g., Eclipse EMF, and web app server security APIs, XACML, syslog) and is designed as a customizable, future-proof toolkit, easily expandable to both legacy devices and new kinds of devices from different vendors.

Figure 1 illustrates the process at a high abstraction level: A model-driven development process is depicted in the right half of the figure. Application interactions are modeled using a service orchestration (or similar) tool, which basically allows application modules to be "plugged together." The actual application is the integrated orchestration of those modules (1). The orchestration tool automatically deploys and integrates the modules as modeled. This process provides valuable, reliable information about the application with its interaction to the model-driven security process, which works as follows.

The first step of automation involves meta-modeling the features of the security policy using a Domain-Specific Language (DSL) (2). Then the security policy is modeled (3) using features specified in the meta-modeled DSL. If necessary the policy generation workflow can be customized (4). The model-driven security component policy enforcement points (PEP) are installed into the runtime platform (5). The security workflow can then automatically generate fine-grained, contextual, technical security policy rules (6), taking into account various system and context information, including the application integration model (1) used to build the application (7). The technical security rules are then automatically pushed into the policy enforcement points for enforcement. Whenever applications change (esp. the integration), the technical security rules can be automatically re-

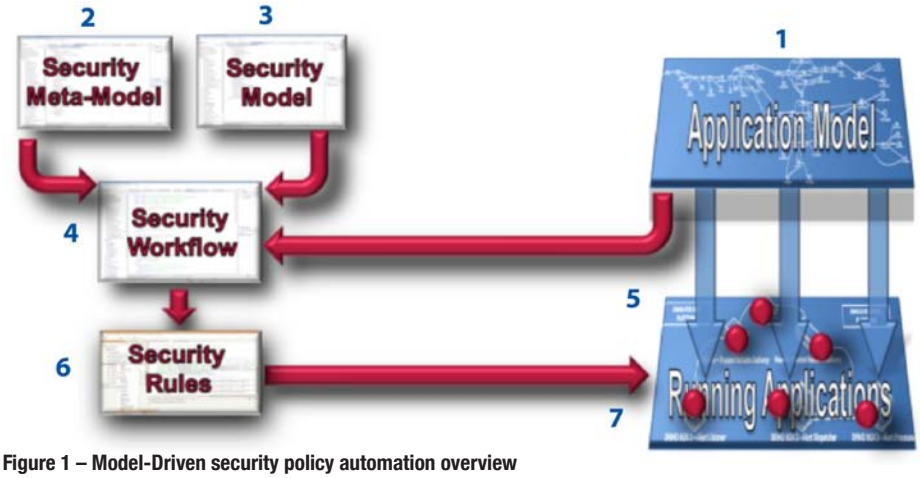


Figure 1 – Model-Driven security policy automation overview