Classification of Malicious Distributed SELinux Activities

Mathieu Blanc*, Patrice Clemente, Jonathan Rouzaud-Cornabas, Christian Toinard
ENSI de Bourges, LIFO EA 40 22, 88 Bld Lahitolle, 18020 Bourges Cedex, France
*CEA, DAM, DIF
F-91297 Arpajon, France
Email: {jonathan.rouzaud-cornabas,patrice.clemente,christian.toinard}@ensi-bourges.fr, mathieu.blanc@cea.fr

***Abstract*— This paper deals with the classification of malicious activities occurring on a network of SELinux hosts. SELinux system logs come from a high interaction distributed honeypot. An architecture is proposed to compute those events in order to assemble system sessions, such as malicious ones. Afterwards, recognition mechanisms are proposed to classify those activities. The paper presents the classification architecture using comprehensive examples. It is the first solution that supports SELinux sessions. In contrast with previous works, distributed sessions are better addressed using only SELinux logs. The results of experiments use real samples taken from our honeypot. A high performance architecture enables to compute a large amount of events captured during one year on our high interaction honeypot. Our approach enables the real-time reconstruction of system sessions. Moreover, sessions are compared to patterns in order to classify them according to specific attacks. The paper shows that the classification can be done in a linear time. An automatic recognition of new patterns is proposed.**

***Index Terms*— SELinux sessions, classification of attacks, distributed sessions.**

I. Introduction

When considering the detection of malicious activities in High Performance Computing (HPC) environments like computing clusters or grid architectures, one faces various challenges. First, they usually rely on high speed and low latency networks generating very high quantities of network traffic, making the traditional Network Intrusion Detection approach (so called NIDS) extremely inefficient. Then they can be deployed on wide networks, like in the case of grids, making it difficult to aggregate data from the various networks involved. Finally, HPC environments generally rely on ciphered network protocols like SSH, which are impossible to monitor from a pure network point of view. For all these reasons, it is mandatory to perform the detection at operating system level.

Integrated in the Linux kernel, the SELinux security facility provides a complete auditing framework, in particular full system call logging. Thus it becomes possible to analyze a malicious activity only by auditing system activity. Those analysis both provide fine characterization of malicious activities and allow the classification of these activities, which means a better comprehension of malicious system activities. Such results can lead to the improvement of the configuration of the protection and detection mechanisms as much as their efficiency.

The kernel of that paper is the reconstruction of sessions composed of system activities, assuming that their elementary system events (i.e. SELinux interactions or 'system calls') are all captured by the SELinux logs. In contrast with other approaches, reconstruction includes distributed sessions.

Afterwards, macro activities are computed using a combination of local and distributed sessions. The objective is to identify complex macro-activities that fit to classify a wide range of malicious activities such as connections to hosts, island hopping between machines, scans, massive activities, etc. Each macro activity has to be classified. The combination of those macro activities allows us to recognize generic patterns of attack based uniquely on SELinux logs.

SELinux provides a very secured kernel but produces large amounts of events in the log file. Moreover, SELinux events contain much more data than classical Unix traces. Experimental results show the efficiency for the reconstruction of distributed SELinux sessions. It is the first solution that enables to compute SELinux logs to reconstruct completely distributed activities. An advanced application is given for Distributed Deny of Service activities.

After presenting the motivations of this work through a state of the art, we will first introduce some formal concepts and basic concepts for the remaining of the paper. Then we will go into the fine description of our correlation architecture, from the filtering module to the Complex Scenario Detection module, without forgetting the migrating session reconstruction module and the massive activity classification modules, which are, with the reconstruction of SELinux sessions, the most significative aspects of these works. After that we will discuss on the learning of system session patterns needed to perform classification. The experiment section will lead us to our first significative results of learning and detecting massive activities (e.g. DDoS) within our real world logs. The last section will end this paper with conclusions and perspectives, in particular for new directions of research

This paper is based on "Correlation Of System Events: High Performance Classification Of SELinux Activities And Scenarios" by J. Rouzaud-Cornabas, P. Clemente, C. Toinard, which appeared in the Proceedings of the 2008 International Conference High Performance Computing & Simulation), Nicosia, Chypre, June 2008. © 2008 ECMS.

to overcome for example complexity limitations in automatically generating patterns.

II. State Of The Art

Very few studies on system activities address the reconstruction of system session. A lack exists to manage system activities and distributed sessions. Finally, solutions lack to recognize SELinux activities and distributed SELinux activities.

[1] studied system activities for specific system services, in order to enforce some security properties. Their examples only concentrate on specific services like Apache. They do not really work on classifying system activities, rather on access control.

[3] uses code analysis to find the authorized system calls. That relies on statical analysis of programs and can only provide characterization of specific applications, but cannot lead to classification of a whole system.

[4] uses $strace$ to monitor system calls. They focus on statistical analysis but not on classifying activities, particularly malicious ones.

All these approaches partially monitor the existing system calls. None of them is able to monitor all the system calls in order to reconstruct the sessions occuring on the target operating system.

In [5], scenarios are seen in terms of sequences of legal operations and the entire system is monitored, but only in terms of intrusion detection: a policy graph is dumped into memory, and each sequence of interactions that violates a security property, raises an alarm. Moreover, sessions cannot be studied *a posteriori* in order to recognized distributed activities.

Finally, the limitations include the lack to analyze complex sessions such as Island Hopping, distributed sessions or massive system activities.

Correlation from multiple sources, are presented in [6], [7]. Those authors propose a generic framework to correlate any kind of information coming from multiple network sources in order to detect intrusions. A language [8] enables to describe an attack and/or to create a pattern of detection. But they only use data collected from networks and not at the operating system level. Moreover, their language does not support learning algorithms in order to recognize patterns and classify distributed attacks. However, our proposal reuses some of those ideas and provides extensions for the operating system events.

[9] combines several correlation algorithms to extract patterns from IDS data. But again, the data come only from network tools and sensors. This approach deals with very simple network sessions. That approach does not seem to be able to provide real analysis.

Finally, the limitations may include *inter alia* 1) the lack of a reconstruction and classification of SELinux activities, 2) the lack of support for distributed activities and 3) the lack of a unique framework using only operating system events.

III. Formal concepts

The novelty of our approach deals with the reconstruction of the system sessions. Sessions are sequences of system activites (i.e. SELinux interactions, i.e. 'system calls', e.g. file/socket read/write). Those system activities enable to compute different types of macro activities called macro-events. Our architecture is designed to work with information given by system loggers, host oriented sensors, network oriented sensors and also host IDS (HIDS) and network IDS (NIDS).

For the correlation process, each computer must log all the events that can be observed at the operating system level or the network level. Our implementation and the related experimentations use only system calls coming from the SELinux logs but could be easily extended to any kind of logging facility able to capture all the system calls.

A. Basics

Before presenting the various modules of our architecture, let us introduce some formal concepts. Other relevant concepts may be given all along the paper, when needed.

1) General definition of an event: Intuitively, an event is closed to the notion of an elementary operation, or SELinux interaction or also 'system call'. Formally, a system event e is a set of attributes $Attr$. Each attribute is associated with a label $l \in L$ (where L is the set of possible labels), $Attr = \{attr_l; l \in L\}$. Table I gives a subset of L including the labels for the major attributes. Some of those are inspired from the definition of SELinux Security Contexts [2].

Label	***Description***
e_id	unique event number
s_con	source security context
t_con	target security context
sec_perms	security permission (e.g., r, w)
sec_class	security class (e.g., file, dir)
$host$	where the event appeared
pid	process number
$ppid$	parent process number
$date$	date (in millisecond) of the event
$name$	name of the file involved
s_addr	source IP of the event
d_addr	target IP of the event
s_port	source $port$ of the event
d_port	target $port$ of the event
$id\_pattern$	event/alarm classification (type)
$level$	similarity between $id\_pattern$ and the event
se_id	unique system session number
$exchange\_type$	how the flow is directed
$named\_socket$	name of the socket
$direction$	specify the path between two sessions
$count$	number of occurrences of that event

TABLE I.
EVENT ATTRIBUTES

2) Meta-event : Let us introduce the notion of meta-event. A "meta-event" is a set of attributes and/or collections of attributes. A meta-event is also called a macro-activity.

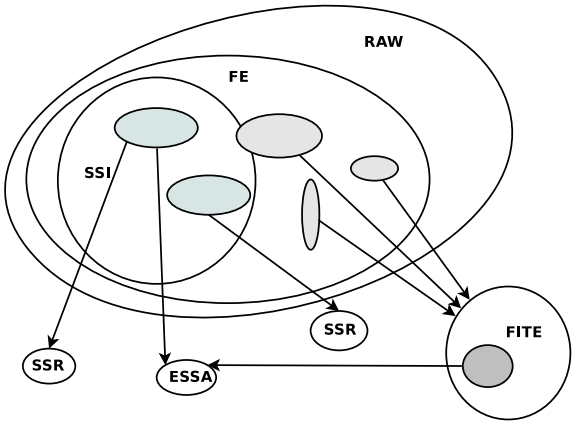

Figure 1. Kinds of computed information used by the correlation process

For example, the meta-event $me = (\{s\_con\}, t\_con, \{s\_addr\}, d\_addr, \{s\_port\}, d\_port)$ represents a meta-event with multiple source contexts, source IP and source ports (e.g. a DDoS attack).

IV. Software Architecture

In this section, we present the overall correlation architecture. Each computing module of our architecture works on information of different kinds. In Figure 1, one can see all the kinds of information the computing modules of our architecture deal with, starting from raw events to enhanced system sessions for example. This figure will be more detailed hereafter.

A. Event Filtering (EF)

This very first computing module is a pre-processing module normalizing and formatting the logs, in order to prepare the correlation phase. It filters the raw events into well formatted ones.

Let us present what are raw events.

1) Raw SELinux event: A *raw event* is an event stored by SELinux into system logs. Such an event includes the attributes labeled with the followings: e_id, s_con, t_con, sec_perms, sec_class, $host$, pid, $ppid$, id, $date$, $name$, s_addr, d_addr, s_port, d_port.

An example of 8 RAW events is shown in Table II, where only 6 attributes appear among the 15 listed above. Those events are sorted by e_id according to their arrival order on the target monitored system.

Starting from those raw events, the Event Filtering module computes filtered events. In this paper, we work only with SELinux logs but our model would work as well with other system logs (e.g. GRSecurity, SystemTap, DTrace). For each logging tool, we need a mapping table in order to normalize each system event. That mapping has to follow the labels given in Table I.

2) Filtered event: Formally, a *filtered event* has exactly the same attributes as a RAW event.

For example, the EF module will exclude each raw event that has not been correctly logged by SELinux,

e_id	s_con	t_con	$ppid$	pid	$host$	$\cdots$
1	init	ftpd	1	330	www	$\cdots$
2	init	sshd	1	2045	www	$\cdots$
3	init		1	2036	www	$\cdots$
4	sshd	passwd	2045	3066	www	$\cdots$
5	sshd	passwd	-1	3068	www	$\cdots$
6	sshd	passwd	2045	3075	www	$\cdots$
7	passwd	bash	3075	4587	www	$\cdots$
8	bash	ls	4587	5874	www	$\cdots$

TABLE II.
RAW SELINUX EVENTS

i.e. that has missing or malformed attributes (e.g. no/incomplete date). An example of the output for that module is shown in Table III, where the events #3 and #5 have been filtered (removed) from Table II.

e_id	s_con	t_con	$ppid$	pid	$host$	$\cdots$
1	init	ftpd	1	330	www	$\cdots$
2	init	sshd	1	2045	www	$\cdots$
4	sshd	passwd	2045	3066	www	$\cdots$
6	sshd	passwd	2045	3075	www	$\cdots$
7	passwd	bash	3075	4587	www	$\cdots$
8	bash	ls	4587	5874	www	$\cdots$

TABLE III.
FILTERED EVENTS

As shown in Figure 1, only the Filtered Events (FE) are used by the other correlation modules to compute the other kinds of information. So, only the normalized events are used by the correlation process. The FE set is a subset of the Raw Events (RAW).

B. System Session Identification (SSI)

SSI adds meta-knowledges to specific subsets of FE.

Starting from the $init$ process, it builds a tree of PID and PPID for each session.

In real time, each new event $e \in$ FE is 1) added to an existing subtree that represents the (P)PID link or 2) a new subtree is created in which the considered event is added.

The main purpose of this module is to affect a unique attribute to each branch of the tree: the identification number of the session. Currently, only the events that are associated to a user session or system services (e.g. Apache, MySQL) are taken into account.

Formally, a SSI event e^{ssi} is a filtered event with an extra attribute se_id.

The initialization of the se_id attribute (made during the construction of *pid-ppid* tree) uses an $entry\_point\_set$ that describes the interesting transitions between processes. An interesting transition depends of the scope that you want to analyze. In our case, the interesting transitions are the ones that initialize a new session. For example, the transition from the 'sshd' context to the 'user' context enables to compute a new se_id attribute.

Figure 2 shows the *pid-ppid* tree obtained from the filtered event of Table III. It shows that two branches below the '$sshd$' event have different se_id.

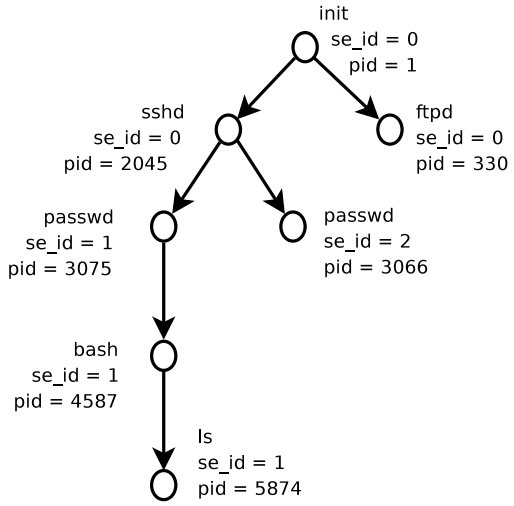

Figure 2. System Session Identification

The Table IV also presents those filtered events (of Table III) extended by the SSI module with a se_id attribute :

e_id	s_con	t_con	$ppid$	pid	$host$	se_id	$\cdots$
1	init	ftpd	1	330	www	0	$\cdots$
2	init	sshd	1	2045	www	0	$\cdots$
4	sshd	passwd	2045	3066	www	1	$\cdots$
6	sshd	passwd	2045	3075	www	2	$\cdots$
7	passwd	bash	3075	4587	www	2	$\cdots$
8	bash	ls	4587	5874	www	2	$\cdots$

TABLE IV.
SSI EVENTS

C. Factorizing InTeraction Event (FITE)

This module aggregates all the events corresponding to the same SELinux interaction (i.e., 'IT') inside a session. This module tackles the problem that an activity (i.e. the execution of a single process) can produce a very large number of system calls. For example, the reading of a file can generate thousands of events for a single activity (i.e. $'\mathrm{cat} < \mathrm{filename} >'$). So, FITE generates a single meta-event instead of thousands of events. This module is largely inspired from [6], [7], [9].

An FITE meta-event me^{fite} includes several attributes: $itfe\_id$, s_con, t_con, sec_class, sec_perms, $host$, $begin\_date$, end_date and $count$. Also, several sets of attributes are computed for all the factorized events: e_id, pid, $ppid$, $name$, s_port, d_port, s_addr, d_addr and optionally se_id.

As an example, let us consider a meta-event factorizing all the read system calls of $/etc/rc.conf$ during a given session $se\_id = 1142$ on a given $host = selinux\_computer$. This meta-event will be composed of: $s\_con = user\_u : user\_r : user\_t$, $t\_con = system\_u : object\_r : etc\_t$, $host = selinux\_computer$, $idession = 1142$, $sec\_class = file$, $sec\_perms = read$, $name = rc.conf$.

D. System Session Reconstruction (SSR)

This module rebuilds the system sessions. As shown in Figure 1, the SSR module takes a subset of SSI events and builds a meta-event representing the session. The SSR meta-event me^{ssr} contains the following attributes: ssr_id, se_id, $host$, $begin\_date$, end_date and $count$.

For example, SSR merges all the events associated with the same se_id (i.e. se_id = '1051') and the same host (e.g. host = 'www-server').

In table V, SSR reconstructs two sessions based on the SSI events shown on table IV, the session #1 (se_id) having only 1 event and the session #3 having 3 events. The session #0 does not appear as it is not a real session: it contains only the $init$ process and other pre-sessions related interactions.

ssr_id	se_id	$host$	$count$	$\cdots$
1	1	www	1	$\cdots$
2	2	www	3	$\cdots$

TABLE V.
SSR EVENTS

E. Enhanced System Session Activities (ESSA)

The purpose of the ESSA module is to find all the events related to a session that are external events (i.e. all the events coming from a communication channel).

In contrast with [6] [7], this module supports Inter Process Communication (like pipes, unix sockets) and not only network communications. A ESSA meta-event me^{essa} includes the following elements: $essa\_id$, se_id, $host$, $begin\_date$, end_date, $count$ and a set of attributes: $\{e\_id\}$.

The relationship between communicating processes and/or users and/or sessions can be revealed using ESSA. For example, ESSA produces a meta-event for a transmission between two processes through a Unix socket, or between a user session and many related IPC events, etc.

ESSA provides a powerful means to overcome the SELinux viewing window: ESSA 'sees' what is above the system sessions, in terms of communication between those sessions. It is worth to notice that, using ESSA, it is thus possible to follow users (and thus attackers) over all the monitored machines, allowing to detect distributed or even migrating sessions. These final correlations between the sessions are achieved by the MDSR module (cf. MDSR, IV-F).

Table VI presents two connections to a ssh server 1) from the session #3 runing on the host www (see event #11 and #13) and 2) a ssh connexion (see event #12) with the creation of a $bash$ (see event #14). For example, ESSA finds the 'link' between the session #3 on www and the event #12 on ftp. Then, it generates a meta-event me^{essa} like the one in the table VII. That meta-event defines a relationship between those two sessions, thus permitting MDSR to establish connections between them, (cf. IV-F). That meta-event will be an entry point for the next module (IV-F) in order to compute migrating/distributed sessions.

e_id	s_con	t_con	$host$	se_id	d_addr	$\cdots$
11	bash	ssh	www	3	ftp	$\cdots$
12	ssh	passwd	ftp	4	ftp	$\cdots$
13	bash	ssh	www	3	ftp	$\cdots$
14	passwd	bash	ftp	4	ftp	$\cdots$

TABLE VI.
SSI EVENTS

$essa\_id$	se_id	$host$	e_id	d_addr	$\cdots$
1	3	www	12	ftp	$\cdots$

TABLE VII.
ESSA EVENTS

F. Migrating/Distributed Session Reconstruction (MDSR)

As said previously, this module aims at relating the system sessions sharing some communication events.

In contrast with [6], MDSR authorizes multiple relationships. A MDSR meta-event me^{mdsr} contains the following unique attributes: $mdsr\_id$, $begin\_date$, end_date, $exchange\_type$, $named\_socket$, and the following multiple attributes: se_id_k, $host_i$, s_addr_i, d_addr_i, s_con_i and t_con_i, with $i \in [1..n]$ and $k \geq n$, where n equals the number of computers involved in the distributed session and k the number of local system sessions.

That module takes the (SSR) sessions or the enhanced (ESSA) sessions as inputs. For example, sessions #3 and #4 of Table VI will be linked as a migrating session, according to the $essa\_id$ #1 meta-event of Table VII. Thus, MDSR generates the meta-event of Table VIII that represents this migrating session.

$mdsr\_id$	ssr_id_1	ssr_id_2	$\cdots$
1	3	4	$\cdots$

TABLE VIII.
MDSR EVENTS

More generally, this module allows to detect every migrating session (e.g. information flows) or distributed sessions (e.g. DDoS).

G. Macro activity classification

1) System Session Classification (SSC): This module allows the correlation process to classify (recognize) SSR and ESSA. It uses the sessions reconstructed by SSR or ESSA and compares them to system patterns.

Previous works [10], [11] (with their so-called 'Prerequisites and consequences') were related to classification of network sessions. But they do not fit with an automatic learning of sessions. Indeed, prerequisites and consequences must be written by end users, which is quite a difficult task. Our approach is substantially different. SSC creates an automaton that represents a session in order to compare it with system patterns also represented as automata. Those patterns have to be constructed by another process (cf. section V).

When a comparison succeed, i.e. if the whole session, or even just a subset of it, matches to an existing pattern, the SSC module generates a SSC meta-event me^{ssc} with the following attributes: se_id, $host$, $id\_pattern$ and $level$. Each SSC meta-event me_i^{ssc} associates a pattern to a real session of the monitored operating system. Each session can be associated to multiple patterns.

For example, as seen in figure 3, SSC allows to compare a session represented as an automaton (on the left side) with a system pattern learned before (on the right side). They are both real sessions and patterns. More precisely, the session on the left side represents a SSH connection followed by: (1) the execution of a shell, (2) the transition from the $sshd$ context to the $user$ context, (3) the opening of a virtual console, (4) some interactions, i.e. read and write, from this console.

The pattern (on the right side) represents: (1) the detection of a connection through a SSH (transition from $sshd$ to the user context), (2) then the opening of a virtual console.

To allow a more flexible and powerful recognition, this module compares the two automata using a similarity level (i.e. the number of attributes that have to be equal for each pair of nodes). The lowest similarity level uses only two security context attributes. Different levels of classification can be used for each pattern.

Complexity. In terms of complexity, the SSC module compares each unclassified (i.e. unrecognized) session with each System Pattern. Let n be the number of the sessions to classify and p the number of patterns. The complexity is $\mathcal{C}_{ssc} = \Theta(np)$ in number of comparisons. Each comparison is actually quite complicated. Indeed, it addresses the problem of counting the isomorphisms of the sub-graphs, which in general is at least in $\Omega(e!)$ where e is the number of events (nodes) of the graph (i.e. the System Pattern). Hopefully, recent works [12] have provided interesting theoretical results. They show that for a graph G that is simply a tree (i.e. a system session tree here), with a fixed number a of attributes per node (which is also the case here, where $2 < a < a\_max$) where $a\_max = 4$[1]), the counting of the sub-graphs, that are isomorphic to another a fixed graph P, (i.e. a Session Pattern here) can be done linearly to e. With those results, the SSC complexity is reduced to $\mathcal{C}_{ssc} = \Theta(np \times ke_G)$, where k is a multiplicative constant of the number of events e_G of G.

This is an important result as it can lead to a real-time recognition of the session patterns. However, until now we could not experiment this as we still lack of some representative patterns (cf. section V and VII-B.5).

2) Massive IT Reconstruction (MITR): In contrast to [6] [7], this module classifies massive system activities. It merges and counts all the events (i.e. interactions or IT) that are the same (all the attributes are equal) during

[1]At this stage of our work, a_max is limited to a maximum of 4 for performance's sake, especially for the construction of the patterns. But, ideally, a_max could increase up to the maximum number of SSR events' attributes.

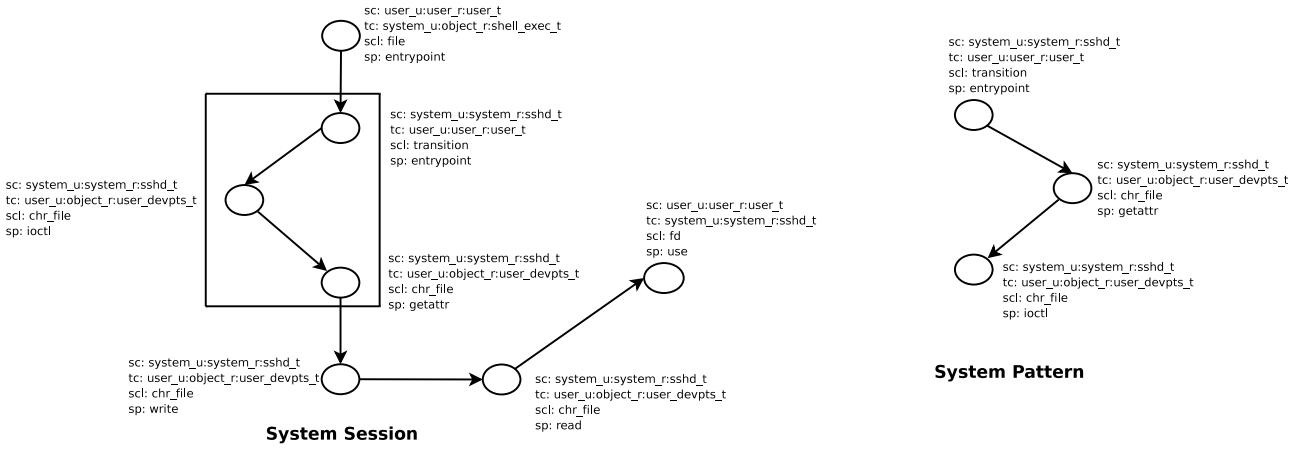

Figure 3. System Pattern Recognition

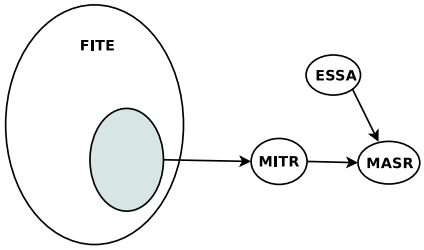

Figure 4. Massive Activity Detection

e_id	s_con	t_con	$host$	se_id	$\cdots$
12	sshd	passwd	www	1	$\cdots$
15	bash	mysql_socket	www	3045	$\cdots$
16	bash	mysql_socket	www	3045	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
414	bash	mysql_socket	www	3045	$\cdots$
415	bash	mysql_socket	www	3045	$\cdots$

TABLE IX.
SSI EVENTS

$mitr\_id$	s_con	t_con	$host$	$count$	$\cdots$
1	bash	mysql_socket	www	400	$\cdots$

TABLE X.
MITR EVENTS

a given time window in order to create a MITR meta-event. If they are too many above a given threshold, that module creates a meta-event MITR. The definition of these thresholds remains outside the scope of this work. Such a meta-event contains relevant information about the massive activity like its source(s) and destination(s). It is not limited to a one to one massive activity, it can also classify many to many and any other variations of a massive activity.

A MITR meta-event me^{mitr} contains a simple attribute $mitr\_id$ and many set of attributes: s_con_{i1}, t_con_{i2}, sec_class_{i3}, sec_perms_{i4}, $begin\_date$, end_date, $exchange\_type_{i5}$, $named\_socket_{i6}$, s_addr_{i7}, d_addr_{i8}, s_port_{i9}, d_port_{i10}, $count$, where each ik exclusively equals 1 or n (with n the number of machines involved in the massive interaction).

This module takes all the events as shown in table IX (that is very small to be able to fit in the article). Then, it generates a meta-event MITR that represents the massive activity as shown on table X from $bash$ to $mysql\_socket$ on the host www, thus grouping the events #15 to #415 of the table IX.

*3) Massive Activity Session Reconstruction (*MASR*):* This module allows to link a massive activity with the sessions that created it or have been created by it, for example a ssh bruteforce succeeding in the opening of a system session. When a link is found, it creates a meta-event MASR (see Figure 4).

A MASR meta-event includes the following attributes: $masr\_id$, $mitr\_id$, se_id, $host$ and $direction$.

For example, looking at Table IX and Table X, that module is able to generate a meta-event $masr\_id$ #1 as shown on the table XI. It shows that the massive activity #1 can be linked with the session #3045 occured on the host www.

$masr\_id$	$mitr\_id$	se_id	$host$	$\cdots$
1	1	3045	www	$\cdots$

TABLE XI.
MASR EVENTS

*H. Complex Scenario Detection (*CSD*)*

In other correlation approaches, a special language is required to express the links between " meta-events " [6]–[8], [10], [11].

Our approach is very different. CSD is not really a module like the other ones, but a combination of modules that represents the way of linking the previously modules all together (with none, one or multiple implementations of each one) in order to detect a complex scenario. Actually, this combination is implemented as a self organizing

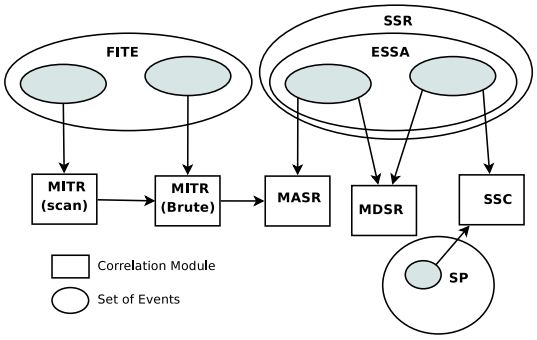

Figure 5. Complex Scenario Detection

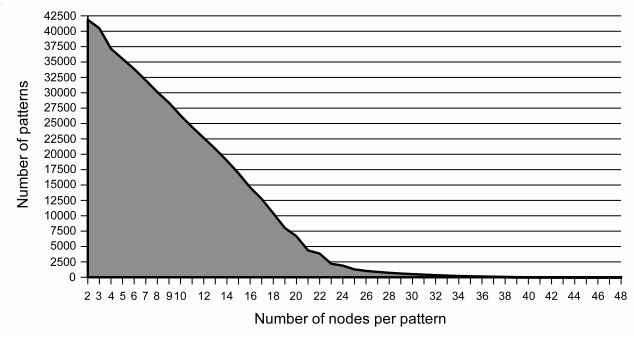

Figure 6. Number of learned patterns

algorithm that orders the use of each module in order to fit with the current scenario being computed.

Figure 5 represents an example of a Complex Scenario detection. It starts on the left with a scan of a monitored computer (i.e. a MITR meta-event) followed by a bruteforce attempt (i.e. a MITR meta-event). Then, the bruteforce intrusion attempt (i.e. the MITR meta-event) is linked (as an MASR) to the ESSA meta-events associated with the activities caused by the bruteforce (cf. MASR). On the right side, the SSC module classifies another ESSA event using a system pattern (SP). This ESSA event represents the system session caused by that successful bruteforce intrusion. The SSC event is linked to the MASR meta-event as a new MDSR meta-event.

V. System Pattern Learning (SPL)

Currently, a learning module is proposed to compute automatically the System Patterns (used in SSC and CSD). For this purpose, each system automaton is compared with all the other system automata, according to a similarity level. The comparison is done on the nodes (i.e. the filtered events or factorized events). If at least two successive nodes are equal, a new pattern is created. This pattern contains only the equal attributes corresponding to the considered similarity level. If the pattern already exists, its frequency is simply increased. The algorithm compares all the sessions one by one in order to have correctness. In terms of theoretical complexity, each session is compared with all the others. Thus, the complexity is $\mathcal{C}_{spl} = \Theta((n-1)!)$ in number of comparisons between the sessions, where n is the number of reconstructed sessions.

Actually, this first approach is really time consuming.

We compute a first pool of sessions in order to learn the patterns. The computation was manually stopped after 2 days. The results (cf. figure 6) show that the obtained patterns with less than 20 nodes are numerous. But on the other hand, very few patterns with more than 30 nodes were learned. After those two days, none over 48 nodes were learned. That real experience shows that this algorithm is not suitable for all the data. Indeed, we have sessions with more than 1,000,000 events that we can't compute. However, for the events that are hugely factorized, very small patterns can fit. For example, the following pattern, made of less than 10 factorized nodes is much more interesting than a simple pattern made of 10 events. This pattern is made of the following linux commands: $ssh, bash, ls, wget, bzip2, pscan2$. This gives good hope for the future if we consider a limited initial set of patterns for the linux commands. For example, if we could build a pattern for each of the above commands, we can hopefully be able to build complex patterns using those basic ones.

VI. Massive Activities Pattern Learning (MAPL)

Using a demographic clustering algorithm that regroups the events by clusters based on the similarity level between a subset of the attributes, we are able to detect massive activities without requiring in advance the corresponding pattern. This learning module is able to work because of the nature of a massive activity: many events with similar attributes and values in a time window. Moreover, this unsupervised algorithm permits to work out-of-box without configuration.

With this module, we can study and detect the new massive activities in almost real-time with the current implementation. Moreover, we are able to generate on the fly the pattern to detect this type of massive activity. The complexity of this module is linear with the number of events required for the generated clusters.

VII. Experimentation

A. Physical Architecture for the Correlation Process

The decentralized architecture presented in [13], did not fit our needs because we want to limit the overhead on the monitored hosts.Moreover, it is not safe to run the correlation on the monitored computers since they can be compromised. Finally, we need a scalable architecture for the various steps of our correlation process. Thus, we propose a *grid* architecture for the whole correlation process, supporting a large amount of RAW events, and also providing almost real-time classification.

The *grid* is composed of one correlation manager that handles the user requests and transfers them to the grid controller. The grid controller transfers each request on computing nodes using load-balancing algorithms. There are 10 computation nodes having 3Ghz processors and

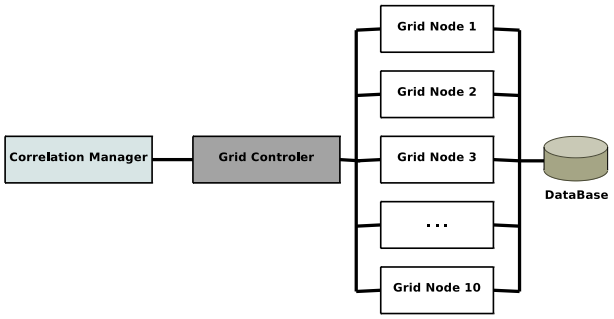

Figure 7. Grid Architecture for Correlation

512MB of memory. We used an IBM DB2 centralized database for the data retention. Each module of the global correlation process is implemented as an agent that can be instantiated on multiple nodes of the *grid* to compute the various data. Thus, a scalable architecture is available, enabling to add new computers on the fly, using a PXE boot, to increase the computing power.

B. Experiment Results

As said before, even if our conceptual architecture can correlate multiple information coming from multiple sources and sensors, at this stage of our works, our solution has been experimented using SELinux events.

Those events come from a High-Interaction HoneyPot including 4 hosts using GNU/SELinux over Gentoo and Debian. One year of events are processed using our *grid* implementation.

1) Event Filtering: One host of our honeypot generated around 164,000,000 raw events i.e. SELinux system calls. The Event Filtering module is important because syslog makes many errors when it reports the events. EF detects about 3 percents of wrong events but this rate can increase with an important activity of the SELinux host. During our experiments, EF produces 160,000,000 valid events.

The Factorizing InTeraction Event module reduces the number of events that has to be stored in the database. Starting from 160,000,000 EF events, FITE succeeded to produce only 8,000,000 meta-events.

2) System Session Identification: The System Session Identification module SSI computed the data coming from the four SELinux hosts of our honeypot. The SSI module was set to use only SSH, MySQL and FTP services as entry points (other services, such as HTTP SMTP IMAP, have not been considered by those experiments). Table XII shows the number of sessions for each of the four SELinux hosts. $Gentoo1$, $Gentoo2$ and $Debian$ are connected directly to the Internet while $Gentoo3$ is reachable through one of those three gateways. Differences between $Gentoo$ and $Debian$ is due to the SELinux policies that are more precise on $Debian$ than on $Gentoo$. Starting from a $Gentoo$ host with 160,000,000 EF events, SSI identifies an average of 40,000 sessions.

3) System Session Reconstruction: The System Session Reconstruction module associates each session with all the system activities. Many sessions are almost empty i.e. the user only connects but does not enter any commands. A typical example can be when an attacker only tests a password and disconnects (even when the password is the good one). For one year of logs, corresponding to 40,000 SSI sessions, SSR build only 8,000 sessions where an activity continues after a successful login. On average, the computing time was 1500 ms per session, using our *grid* architecture. Computing 40,000 SSI sessions took about 17 hours. Thus, if we consider an average of 40,000/365 = 109 sessions per day i.e. less than 5 sessions per hour, those sessions are computed in almost real-time (5*1,5=7,5s).

	Gentoo 1	Gentoo 2	Gentoo 3	Debian
Sessions	58,163	30,825	79	139,859

TABLE XII.
NUMBER OF SESSIONS DETECTED FOR EACH COMPUTER

Table XIII shows that SSR consumes between 800 and 3,000 milliseconds to reconstruct a session according to the number of SELinux events included in that session. The computation includes a constant time of 700ms for launching the agent. Many sessions requires 800ms i.e. uses only 100ms for the algorithm processing. The worst case was reached by a SSH session with a local bruteforce on the ftp server. In that case, the machine spent time for swapping because the required memory was too large, thus increasing abnormally the computation for this session.

With the latest implementation of our JavaGrid (based on JPPF framework), we are able to reduce the launching time to only 20ms. That high capacity enables to manage large datasets.

	Small	Average	Large	Huge
Duration	800	1500	3000	15700

TABLE XIII.
SSR COMPUTATION TIME (IN MILLISECONDS)

4) Massive IT *Reconstruction:* The MITR module has been able to detect one bruteforce on the FTP service. Due to the configuration of SELinux, some relevant system calls have not been audited. Those missing events prevented to efficiently detect the SSH bruteforces.

5) System Pattern Learning: As we said in the System Pattern Learning section, the rough complexity for the construction of the system patterns is uncomputable. However, our experimentation showed some directions we could exploit.

Complexity. In average a session includes about 600 nodes. The comparison between those sessions takes between 1 seconds for the lowest similarity level $a = 2$ and about 200 seconds for the highest similarity level $a = 4$. About $40,000$ sessions have been collected during one year. That means at least $\mathcal{O}(40000! \times 1) \simeq \infty$ seconds to compute all the sessions at the lowest similarity level. Of

course, the comparison at higher similarity level are less numerous, and typically only few sessions are compared at the highest level. To minimize the computation, only big sessions have been considered, stating that those sessions contain also smaller ones. Among these $40,000$ sessions, only 72 had above the average number (i.e. 600 nodes per session). Our first experiments build more than 40,000 patterns after a computation of 2 days that was manually ended.

6) System Pattern Recognition: The SPR module used the System Patterns created with the SPL module (see System Pattern Learning section).

The right part of figure 3 shows a really learned system pattern representing the connection of a user through SSH i.e. a migration from the SSH context to the user context, then the opening of a virtual console and finally an interaction between the user and the virtual console.

3 different levels of similarity have been considered for the recognition of 40,000 sessions (see the System Pattern Learning section above). As seen in table XIV, 13,040 sessions contain this pattern for a similarity level of 2 i.e. the comparison is based only on two security contexts (source and target). For level 3 (addition of one variable), 13,000 sessions respect this pattern, and only 6,680 sessions for level 4 (addition of one variable). We are currently working on isomorphisms detection of subgraphs, in order to benefit from complexity results given in IV-G.1. It should allow more complex patterns and the maximum level of similarity.

Similarity	**OK**	**Not OK**
2	13040	26960
3	13000	27000
4	6680	33320

TABLE XIV.
CLASSIFIED SYSTEM PATTERNS

7) Migrating/Distributed Session Reconstruction: MDSR recognized 4 IslandHopping (3 using network connections, 1 using a Unix socket). MASR was able to link 7 sessions that took part of a massive attack. It is worth saying that the limited numbers of complex sessions (such as MDSR, MITR and MASR) is due to the efficient protection provided by the SELinux kernel. This MAC protection limits really the possibilities of ordinary users and controls all the interactions between a process and the system resources. So, it is really hard to conduct advanced attacks on such systems. However, one can see that possibilities to violate the security still exist.

8) Massive Activities Pattern Learning: Using our module, we are able to detect very large massive attacks in our dataset. One of these massive attacks is linked to almost 15% of all the events generated during one year and spreads on 3 days. But, we have also detected many smaller massive attacks. All those ones are a bruteforce attack on SSH launched from our honeypot. Finally, we have been able to validate completely our module through the detection of two local massive attacks, one launched against our ftp service on the local loopback and the second one launched against our local $MySQL$ service using a Unix socket.

% of all events	s_con	t_con
15	sshd	passwd
4.265	sshd	passwd
3.4587	bash	ftpd
1.458	bash	mysqlUnix

TABLE XV.
MASSIVE ACTIVITIES PATTERNS

Table XVI shows other results, related with the similarity between the sources and destinations security contexts, but also with the $comm$ value i.e. the shell command that generates the event. More than a half of all the events can be linked to the usage of two commands $find$ and $pscan2$. Look at the honeypot, we found that $find$ and $pscan2$ are exactly the same application: a bruteforce for ssh. With this observation, we can link more than 80% of all events to ssh bruteforce. Finally the two other lines can be linked to a legal activity. The first one is our script that backups all the logs: it connects through ssh, compress the data using $bzip2$ and sends it back. The second one is another script that keeps up-to-date our packages: it connects through ssh, authenticates from a $user$ account to a $root$ account and then downloads and upgrades the packages.

VIII. CONCLUSION AND PERSPECTIVES

This paper presents a method to classify SELinux activities starting from complete sessions. Complex sessions can be reconstructed such as distributed, migrating sessions using several connections. Currently, it is the only solution able to analyze SELinux logs. The problem is complex due to the large amount of events generated by a complete system. The solution has been experimented during one year using several high-interaction honeypots. More than 160,000,000 events have been analyzed for each honeypot. Thus, 8,000 sessions, with relevant activities, including several distributed sessions have been completely reconstructed. Some of these activities have been classified according to various system patterns. Classification requires system patterns in order to associate an class of activity to the reconstructed activity. Using malicious patterns, we can also classify malicious activities.

Our classification method presents a linear complexity. Using a *grid* approach, a real-time classification of the system sessions has been proposed. A learning module is proposed to compute automatically the (legal or malicious) system patterns starting from the reconstructed sessions. We show that rough pattern learning is a NP-Complete problem. However, malicious patterns have been learnt.

The process of learning detects and classifies massive activities using an unsupervised solution. We are thus able to detect and classify never seen and already seen

% of all events	s_con	t_con	$comm$
52.45	user	ssh_port	find
29.13	user	ssh_port	pscan2
6.30	user / root	user_home / root_home / etc / ssh / tmp	bzip2 / bash
3.47	ssh / user / root / portage	user_home / tmp / http_port / etc	wget / touch / tar

TABLE XVI.
MASSIVE ACTIVITIES PATTERNS

patterns of massive activities without the need of any configuration.

In order to have real patterns that the classification can use, we are investigating methods to manually build patterns. The method is pretty simple, we consider a pattern learning from the Unix commands (i.e. executing the same type of activity). For example, a set of n sessions executing ls should lead to the construction of good patterns for the ls activity. With those elementary patterns, the system should be able to reconstruct complex patterns including simple ones.

REFERENCES

[1] S. N. Chari and P.-C. Cheng, "Bluebox: A policy-driven, host-based Intrusion Detection System," *ACM Transaction on Information and System Security*, vol. 6, no. 2, 2003.

[2] B. McCarty, *SELinux: NSA's Open Source Security Enhanced Linux*. O'Reilly Media, Inc., 2004.

[3] R. Bowen, D. Chee, M. Segal, R. Sekar, P. Uppuluri, and T. Shanbag, "Building survivable systems: An integrated approach based on intrusion detection and confinement," in *DARPA Information Survivability Symposium*. IEEE Computer Society, January 2000. [Online]. Available: citeseer.ist.psu.edu/bowen00building.html

[4] J. Molina, X. Chorin, and M. Cukier, "Filesystem activity following a ssh compromise: An empirical study of file sequences," in *ICISC*, 2007, pp. 144–155.

[5] J. Briffaut, J.-F. Lalande, C. Toinard, and M. Blanc, "Collaboration between mac policies and ids based on a meta-policy approach," in *Workshop on Collaboration and Security (COLSEC'06)*, W. Smari, W. W. et McQuay, Ed. Las Vegas, USA: IEEE Computer Society, May 2006, p. 48–55.

[6] F. Valeur, G. Vigna, C. Kruegel, and R. A. Kemmerer, "A comprehensive approach to intrusion detection alert correlation," *IEEE Transactions on dependable and secure computing*, vol. 1, no. 3, July-September 2004.

[7] C. Kruegel, F. Valeur, and G. Vigna, *Intrusion Detection and Correlation: Challenges and Solutions*. Springer, 2005.

[8] S. Eckmann, G. Vigna, and R. Kemmerer, "STATL: An Attack Language for State-based Intrusion Detection," *Journal of Computer Security*, vol. 10, no. 1/2, pp. 71–104, 2002.

[9] X. Qin, "A probabilistic-based framework for infosec alert correlation," Ph.D. dissertation, Georgia Institute of Technology, 2005.

[10] P. Ning, D. Reeves, and Y. Cui, "Correlating alerts using prerequisites of intrusions," North Carolina State University, Tech. Rep. TR-2001-13, 12 2001. [Online]. Available: citeseer.ist.psu.edu/ning01correlating.html

[11] F. Cuppens and A. Miège, "Alert correlation in a cooperative intrusion detection framework," in *IEEE Symposium on Security and Privacy*. Oakland, USA: IEEE, May 2002.

[12] D. Eppstein, "Diameter and treewidth in minor-closed graph families," *Algorithmica*, vol. 27, pp. 275–291, 2000. [Online]. Available: http://dx.doi.org/10.1007/s004530010020

[13] C. Krugel, T. Toth, and C. Kerer, "Decentralized event correlation for intrusion detection," in *Information Security and Cryptology*, 2001, pp. 114–131. [Online]. Available: citeseer.ist.psu.edu/krugel01decentralized.html

BIOGRAPHIES

Mathieu Blanc is currently a researcher at the French Commissariat à l'Énergie Atomique (CEA). Mathieu Blanc made a PhD in Computer Sciences at the University of Orleans in 2006, as part of the 'Security of Distributed Systems' team, on the topics of the security of largely distributed systems and clusters, including security policies and intrusion detection. He still works on these topics at the CEA.

Patrice Clemente is currently an associate professor at the École Nationale Supérieure d'Ingénieurs de Bourges, France, and a member of the team 'Security of Distributed Systems' of Laboratoire d'Informatique Fondamentale d'Orléans (LIFO). He obtained a PhD in Computer Sciences in 2004 at the University of Franche-Comté, and made his PhD at France Télécom R&D (CNET). He joined the SDS team at Bourges in 2003. His main research at SDS is focused on meta-policies of security, correlation for intrusion detection, intrusion detection for Mandatory Access Control and Discretionary Access Control mechanisms.

Jonathan Rouzaud-Cornabas is a PhD student at the École Nationale Supérieure d'Ingénieurs de Bourges, France, and a member of the team 'Security of Distributed Systems' of Laboratoire d'Informatique Fondamentale d'Orléans (LIFO). He obtained his Master's Degrees in Computer Sciences in 2007 at the University of Orléans, as a member of the SDS team. His main PhD's research at SDS is focused on intrusion detection for Mandatory Access Control and Discretionary Access Control mechanisms.

Christian Toinard is a Full Professor in Computer Sciences. He leads the team dealing with the Security of Distributed Systems i.e. one of the four teams at LIFO, a joint laboratory between the University of Orléans and ENSI of Bourges. He got an engineering degree in Computer Sciences from ENSIMAG Grenoble in 1986. He work as software developper, in the field of embedded and distributed systems, for several french compagnies from 1986 to 1990. Then, he obtained a PhD in Computer Sciences at the University of Paris VI in 1992. Since 2002, he is Professor in Computer Sciences at ENSI of Bourges, an engineering school with a program in Computer Sciences. Since 2002, he developed a research activity towards a close cooperation with CEA for securing Linux OS.

Since 2006 Patrice and Christian organize the COLlaboration and SECurity workshop (COLSEC), held in parallel of Collaborative Technologies and Systems (CTS), in the U.SA.