CORRELATION OF SYSTEM EVENTS: HIGH PERFORMANCE CLASSIFICATION OF SELINUX ACTIVITIES AND SCENARIOS

J. Rouzaud-Cornabas, P. Clemente, C. Toinard
Laboratoire d'Informatique Fondamentale d'Orléans
University of Orléans, Bâtiment IIIA, Rue Léonard de Vinci
45067 Orléans, France
Email: {jonathan.rouzaud-cornabas,patrice.clemente,christian.toinard}@univ-orleans.fr

KEYWORDS

SELinux sessions and scenarios, correlation, detection.

ABSTRACT

This paper presents an architecture for the characterization and the classification of activities occurring in a computer. These activities are considered from a system point of view, currently dealing with information coming from SELinux system logs.

Starting from system events, and following an incremental approach, this paper shows how to characterize high-level and macro activities occuring on the system and how to classify those activities. It gives the formal basics of the approach and presents our implementation. The results of experiments uses real samples taken from our honeypot. Correlation results are obtained using a *grid* computation. Our high performance architecture enables to compute a large amount of events captured during one year on a high interaction honeypot.

INTRODUCTION

The correlation of data generally aims at exploring heterogeneous information to highlight related data regarding particular aspects (e.g., chronology, semantics) among these different sources.

The core of the works presented here is the reconstruction of sessions composed of system activites, assuming that their elementary system operations (i.e., SELinux interactions or 'system calls') are all catched by the SELinux logs.

Following an incremental approach, our solution assembles system events together to represent high-level system activities (linux commands, execution of processes). We group them to represent macro activities including malicious ones, such as connections to hosts, island hopping between machines, massive activities, etc. The combination of these macro activities allow us to charaterize complete sessions on a SELinux system. Each macro activity can be classified. Complete sessions can also be reconstructed combining different types of macro-activities.

The paper gives the fundations of our approach. It defines events, high-level events, "meta-events", sessions, activities and gives the fundations for the algorithms that compute those data. The experimental results show the efficiency of the classification for SELinux activities. As far as we know, it is the first solution that enables to compute SELinux logs. It is interesting since SELinux provides a very secured kernel but produces large amout of events in the log file. Moreover, SELinux events are much more complicated than classical Unix traces.

STATE OF THE ART

The advantages of correlating information from multiple sources, are presented in (Valeur et al., 2004; Kruegel et al., 2005). Those authors propose a generic framework to correlate any kind of information coming from multiple network sources. But they only use data collected from networks and not at the operating system level. (Qin, 2005) combines several correlation algorithms. But again, the data come only from network tools and sensors.

(Chari and Cheng, 2003) studied system activities for specific system services. (Bowen et al., 2000) uses code analysis to find the authorized system calls. (Molina et al., 2007) uses strace to monitor system calls. All these approaches monitor partially the occuring system calls, so none of them is able to monitor all the system calls in order to reconstruct complete sessions. Moreover, complex sessions such as Island Hopping cannot be analyzed. In (Briffaut et al., 2006), scenarios are seen in terms of sequences of legal operations.

Finally, there is currently no solution at all supporting the reconstruction for SELinux sessions.

SOFTWARE ARCHITECTURE

The novelty of our approach deals with the reconstruction of 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 informations 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 have some tools to log all the events that are created on the computer or its local network (network connections).

Proceedings of the 2008 High Performance
Computing & Simulation Conference ©ECMS
Waleed W. Smari (Ed.)
ISBN: 978-0-9553018-7-2 / ISBN: 978-0-9553018-6-5 (CD)

The current implementation uses only system calls coming from the SELinux logs.

Adding Meta Knowledge to SELinux Events

In this first part, our correlation architecture analyzes each elementary operation reported by SELinux in order to highlight the similarity(ies) between them. Indeed, a lot of events are related to the same linux commands or process execution.

This first step adds meta-knowledges to raw events. That improves the effectiveness of the high-level classification modules.

Definitions

Event: Intuitively, an event is closed to the notion of 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 $l \in L$, $ATTR = \{attr_l; l \in L\}$

Table 1 gives a subset of L including the labels for the major attributes.

Label	***Description***
$class$	event/alarm classification
$scontext$	source security context
$tcontext$	target security context
sec_perms	security permission (e.g., r, w)
sec_class	security class (e.g., file, dir)
$hostname$	where the event appeared
pid	process number
$ppid$	parent process number
id	unique event number
$date$	date (in millisecond) of the event
$name$	name of the file involved
$inode$	inode number of the event
$priority$	priority level of the event/alarm
$idsession$	unique system session number
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
$count$	number of occurences of the event

Table 1: Event Attributes

Meta-event (*ME*): The notion of meta-event is also introduced. A meta-event ME is a set of attributes and/or collections of attributes. For example, the meta-event $ME = (\{scontext\}, tcontext, \{s\_addr\}, d\_addr, \{s\_port\}, d\_port)$ represents a meta-event with multiple source contexts, source IP and source ports (e.g. such as a meta-event that modelizes a DDoS attack).

Raw events (*RAW*): A RAW E^{raw} event is composed by the following attributes: id, $class$, $scontext$, $tcontext$, sec_perms, sec_class, $hostname$, pid, $ppid$, id, $date$, $name$, s_addr, d_addr, s_port, d_port.

Event filtering (*EF*): This module is a pre-processing for the correlation phase. If the events can not be normalized, they will not be used by the correlation processus, see figure 1.

Formally, an E^{ef} event has exactly the same attributes as a RAW event.

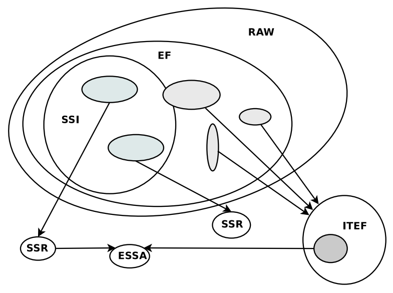

Figure 1: Overall Process of correlation

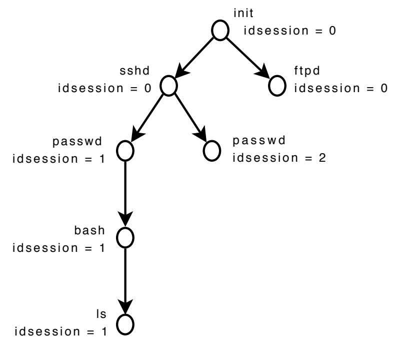

Figure 2: System Session Identification

For example, the EF module will exclude every events that have not been correctly generated by SELinux, i.e. missing or invalid attributes (e.g. no/incomplete date).

System Session Identification (*SSI*): SSI adds meta-knowledges to specific subsets of EF. It builds the tree of PID and PPID of each session.

Each new E^{ef} event of EF is added in real time to an existing tree that represents the (P)PID link or it creates a new tree where the 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, are taken into account. Ongoing works also allow to take into account sessions for services or system scripts.

Formally, a SSI E^{ssi} event is a EF event with an extra attribute $idsession$.

The initialization of the $idsession$ attribute uses an $entry\_point\_set$ that describes the interesting transitions between processes. For example, the transition from the ‘sshd’ context to the ‘user’ context enables to compute a new $idsession$ attribute.

Figure 2 shows that two branches below the ‘$sshd'$ event have different $idsessions$.

InTeraction Event Factorizing (*ITEF*):

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 (execution of a single process) can produce a very large number of system calls. For example, the reading of a

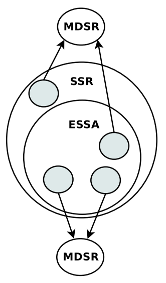

Figure 3: Migrating/Distributed Session Reconstruction

file can generate thousands of events for the same activity. So ITEF generates a single "meta-event' instead of thousands of events. This module is largely based on (Valeur et al., 2004; Kruegel et al., 2005; Qin, 2005).

An ITEF meta-event ME^{itef} is composed by single attributes: id, $class$, $scontext$, $tcontext$, sec_class, sec_perms, $hostname$, $dateBegin$, $dateEnd$ and $count$. Also, it has several sets of attributes for all included events: id, pid, $ppid$, $name$, s_port, d_port, s_addr, d_addr and optionally $idsession$.

For example, a "meta-event" factorizing all the read events of $/etc/rc.conf$ during the session 1142 on the host $selinux\_computer$ is in particular composed of: $scontext = user\_u : user\_r : user\_t$, $tcontext = system\_u : object\_r : etc\_t$, $hostname = selinux\_computer$, $idession = 1142$, $sec\_class = file$, $sec\_perms = read$, $name = rc.conf$

System Session Reconstruction (SSR)
As shown on the 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: id, $idsession$, $hostname$, $dateBegin$, $dateEnd$, $count$ and $class$.

For example, SSR merges all the events with the same identification session (e.g. idsession = '1051') and the same hostname (e.g. hostname = 'www-server'). It's worth mentioning that this module is an original proposition of this paper.

***Enhanced System Session Activities (*ESSA*)*: In contrast to (Valeur et al., 2004) (Kruegel et al., 2005), this module supports Inter Process Communication (like pipes, unix sockets). A ESSA meta-event ME^{essa} is composed by single: id, $idsession$, $hostname$, $dateBegin$, $dateEnd$, $count$ and $class$ and a set of attributes: $\{id\}$.

For example, ESSA produces a "meta-event" for a transmission between two processes through a Unix socket. Thus, a relationship is proposed between a $idsession$ and all the associated IPC events (using their id). A meta-event links a session SSR to all the events coming from the other sessions. The association between the sessions is achieved by the following module.

***Migrating/Distributed Session Reconstruction (*MDSR*)*: In contrast with (Valeur et al., 2004), MDSR authorizes multiple relationships. The MDSR meta-event ME^{mdsr}contains the following unique attributes: $dateBegin$, $dateEnd$, $exchange\_type$, $named\_socket$, and the following multiple attributes: $idsession_i$, $hostname_i$, s_addr_i, d_addr_i, $scontext_i$ and $tcontext_i$, with $i \in [1..n]$, where n equals the number of computers involved in the session.

Let a first information flow exist between session $s1$ and $s2$ and a second one between $s2$ and $s3$. Then MDSR produces a "meta-events" for the transitive information flow. The first information flow is associated to a ESSA "meta-event" $ME^{essa}_{s1,s2}$ between $s1$ and $s2$ events and the second information flow provides a ESSA meta-event $ME^{essa}_{s2,s3}$ between $s2$ and $s3$ events. The MDSR produces a first "meta-event" $ME^{mdsr}_{ME^{essa}_{s1,s2}}$ relating $s1$ and $s2$ and a second one $ME^{mdsr}_{ME^{essa}_{s2,s3}}$ relating $s2$ and $s3$. Finally, MDSR produces a meta-event $ME^{mdsr}_{ME^{mdsr}_{ME^{essa}_{s1,s2}}, ME^{mdsr}_{ME^{essa}_{s2,s3}}}$ for the transitive flow relating $s1$ and $s3$.

Meta Events Recognition
***System Pattern Recognition (*SPR*)*: This module allows the correlation processus to classify SSR and ESSA.

Prerequisites and consequences (Ning et al., 2001) (Cuppens and Miège, 2002) do not feet for automatic learning of sessions. Prerequisites and consequences must be written by end users. It is a difficult task. Our approach is totally different. SPR creates an automaton that represents a session in order to compare it with System Patterns (SP) also represented as automata. Thoses SP are learnt by different processus (see the System Pattern Learning section). SPR generates a SPR meta-event that classifies the session with those patterns.

SPR generate a SPR "meta-event" with the following attributes: $idsession$, $hostname$, $id\_pattern$ and $level$.

For example, as seen on figure 4, SPR allows to compare a session represented as an automaton (on the left side) with a system pattern learnt before (on the right side). They are both real world 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 one, (3) the opening of a virtual console, (4) some interactions, i.e. read and write, on this console. The pattern (on the right side) represents: (1) the detection of a connection through SSH (transition from SSHD to user context), (2) then the opening of a virtual console.

To allow a better recognition, this module compares the two automata using a similarity level (i.e. the number of event attributes that have to be equal between the two compared nodes).

For example, as seen on the figure 4, the lowest similarity level corresponds to the comparison between the session and the pattern session using only two predefined security context values. Different level of classification can be used with the same pattern. At the similarity level 4, the sub-session (in the rectangle) on the left cannot be recognized.

In terms of complexity, the SPR module compares each

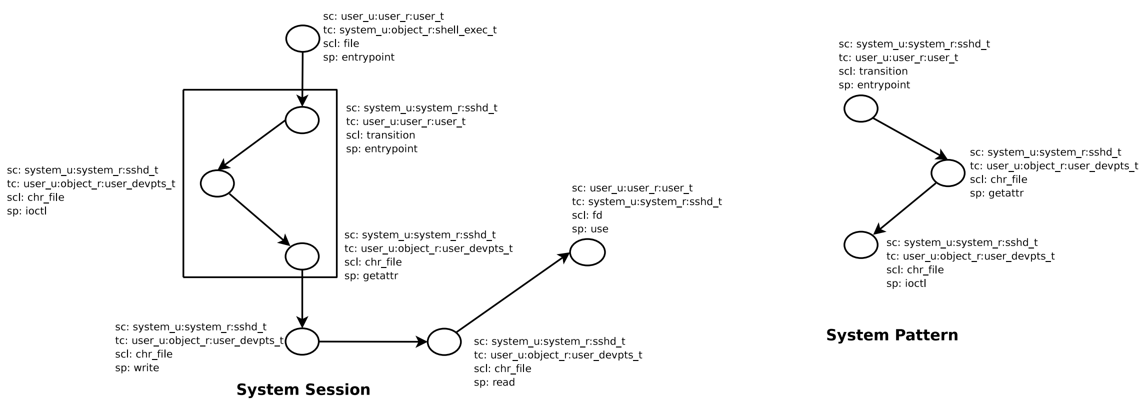
sc: user_u:user_r:user_t
tc: system_u:object_r:shell_exec_t
scl: file
sp: entrypoint
sc: system_u:system_r:sshd_t
tc: user_u:user_r:user_t
scl: transition
sp: entrypoint
sc: system_u:system_r:sshd_t
tc: user_u:object_r:user_devpts_t
scl: chr_file
sp: ioctl
sc: system_u:system_r:sshd_t
tc: user_u:object_r:user_devpts_t
scl: chr_file
sp: getattr
sc: user_u:user_r:user_t
tc: system_u:system_r:sshd_t
scl: fd
sp: use
sc: system_u:system_r:sshd_t
tc: user_u:object_r:user_devpts_t
scl: chr_file
sp: write
sc: system_u:system_r:sshd_t
tc: user_u:object_r:user_devpts_t
scl: chr_file
sp: read
System Session
sc: system_u:system_r:sshd_t
tc: user_u:user_r:user_t
scl: transition
sp: entrypoint
sc: system_u:system_r:sshd_t
tc: user_u:object_r:user_devpts_t
scl: chr_file
sp: getattr
sc: system_u:system_r:sshd_t
tc: user_u:object_r:user_devpts_t
scl: chr_file
sp: ioctl
System Pattern

Figure 4: System Pattern Recognition

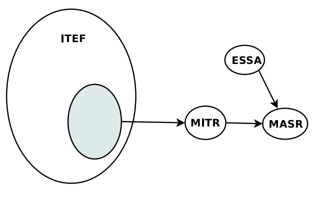

Figure 5: Massive Activity Detection

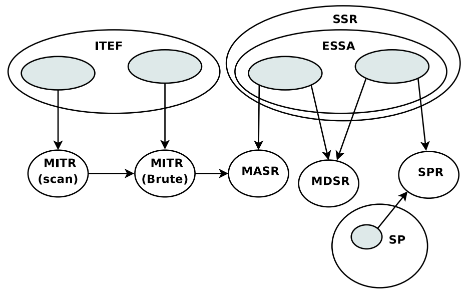

Figure 6: Complex Scenario Detection

unclassified (i.e. unrecognized) session with each System Pattern. Let n be the number of session to classify and p the number of patterns. The complexity $\mathcal{C}_{spr} = \Theta(np)$ in number of comparisons. Each comparison is actually quite complicated. Indeed, it adresses the problem of counting sub-graphs isomorphisms, which in general is at least in $\Omega(e!)$ where e is the number of event (nodes) of the graph (i.e. the System Pattern). Hopefully, recent works (Eppstein, 2000) have provided interesting theoritical 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 nodes (which is also the case here, where $2 < a < 7$), the counting of the sub-graphs, that are isomorph to another a fixed graph P, (i.e. a Session Pattern here) can be done linearly to e. With those results, the SPR complexity is reduced to $\mathcal{C}_{spr} = \Theta(np \times ke_G)$, 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 correlation using the recognition of session patterns.

Massive Activity Reconstruction:

Massive IT *Reconstruction (*MITR*)*: In constrast to (Valeur et al., 2004) (Kruegel et al., 2005), this module classifies massive system activities. It merges and counts all the events (i.e. interactions or IT) that are the same (all attributes with an equal value) in a given time window to create a MITR meta-event. If they are too many above a given threshold, it creates a meta-event MITR. The definition of these thresholds remains outside the scope of this work. The meta-event contains relevant informations 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 contains: $scontext_{i1}$, $tcontext_{i2}$, sec_class_{i3}, sec_perms_{i4}, $dateBegin$, $dateEnd$, $exchange\_type_{i5}$, $named\_socket_{i6}$, s_addr_{i7}, d_addr_{i8}, s_port_{i9}, d_port_{i10}, $count$ and id_mitr, where each ik equals 1 or n (with n the number of machines involved in the massive interaction).

*Massive Activity Session Reconstruction (*MASR*)*: This module is another original proposition. It ables to link a massive activity with sessions that created it or have been created by it. When a link is found, it creates a meta-event MASR (see figure 5).

This module generates a MASR meta-event with the following attributes: id_mitr, $idsession$, $hostname$ and $direction$.

For example, MASR links a massive activity, like a DDoS that has been launched from a monitored computer, with the session that have launched it (using meta-knowledges extracted from MITR meta-event and ESSA/SSR meta-event).

Complex Scenario Detection (CSD)

In other correlation approaches, a special language enables to express the links between "meta-events" (Cuppens and Miège, 2002; Ning et al., 2001; Valeur et al., 2004; Kruegel et al., 2005; Eckmann et al., 2002).

Our approach is different. CSD is not really a module like the other ones, but a combination of modules that represents the way to link the previously seen modules all together (with none, one or multiple implementations of each one) in order to detect a complex scenario. Actually, this combination is implemented by the connexion of automata representing each module.

The abstract figure 6 represents 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 representing activities on the system caused by the bruteforce (cf. MASR).

On the right side, the SPR module classifies another ESSA using a system pattern (SP). This ESSA represents the system session caused by the successful intrusion following the bruteforce attempt.

This SPR is linked with the MASR meta-event as a new MDSR meta-event.

SYSTEM PATTERN LEARNING (SPL)

Currently, a learning module is proposed to compute automatically the System Patterns (used in SPD 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. filtered events or factorized filtered events). If at least two successive nodes are equals, a new pattern is created. This pattern contains only the equal nodes with the attributes corresponding to the similarity level used to create the pattern. If the pattern already exists, its frequency is increased.

The main advantage of this approach is that it is totally unsupervised. The algorithm compares all the sessions one by one in order to have correctness. In terms of theoritical complexity, as each single session is compared with all the others. Thus, the complexity is $\mathcal{C}_{spl} = \Theta((n-1)!)$ in number of comparisons between sessions, where n is the number of reconstructed sessions. Specific DNA sequence alignment algorithms, like the BLAST family (Altschul et al., 1990), enable to provide a lower complexity. Thus, this complexity problem can be addressed. Our first results provide automation for computation of the System Patterns (on the next section).

EXPERIMENTATION

Physical Architecture for the Correlation Process

The decentralized architecture presented in (Krugel et al., 2001), did not fit our needs because we wanted to limit the overhead on the monitored hosts due to security monitoring. Also, it seems too dangerous for us to let the correlation process take place on the same computers where scenarios could happen. In addition, we needed a scalable architecture for the various steps of the correlation process.

Thus, we proposed a *grid* architecture for the whole correlation process, supporting a large amout of RAW events, and also providing almost real-time recognition.

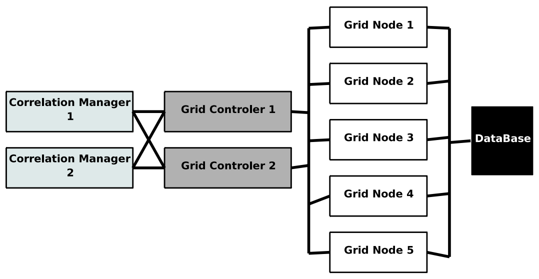

Figure 7: Correlation Architecture on Grid

The *grid* is composed of 3 computation nodes of 3Ghz and 512MB of RAM). We introduced a centralized database for the data retention but also to use a dedicated computing *grid* for our correlation process as shown in the figure 7. The dedicated computing *grid* also brings us the ability to easily implement our modular architecture: each module is implemented as an agent that can be instantiated on multiple nodes of the *grid* to compute various data. Thus, we have an easier scalable processus: we just need to add new computers on the fly, using PXE boot to increase the computing power.

Experiment Results

As said before, even if our conceptual architecture can correlate multiple informations 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 with 4 hosts using GNU/SELinux over Gentoo and Debian. One year of events are processed using our *grid* implementation. An IBM DB2 database is used to store those logs.

Event Filtering: One host of our honeypot generated around 164,000,000 of raw events i.e. SELinux system calls. The Event Filtering EF module is important because syslog makes many errors when it reports 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 of valid events.

160,000,000 events cannot be processed using a classical database such as MySQL because of memory and CPU overhead. So, the IT Event Factorizing module is required in order to reduce the number of events that has to be stored in the database. Starting from 160,000,000 EF events, ITEF succeeded to produce only 8,000,000 meta-events (as shown on table 2).

System Session Identification: The System Session Identification module SSI was applied to the four SELinux hosts of our honeypot. The SSI module was set to use only SSH and FTP services as entry point (other services, such as HTTP SMTP IMAP, have not been

Without ITEF	With ITEF
160,000,000	8,000,000

Table 2: Events number for 1 Year on 1 SELinux Host

considered by those experiments). Table 3 shows the number of sessions for each of the four SELinux hosts. *Gentoo*1, *Gentoo*2 and *Debian* are connected directly to the Internet while *Gentoo*3 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.

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

Table 3: Number of Sessions Detected for Each Computer

System Session Reconstruction: The System Session Reconstruction module associates each session with all the system activities. Many sessions are almost empty. A typical example can be when an attacker only tests a password and disconnects (even when the password was the good one). Starting from 40,000 SSI sessions, SSR generates only 8,000 sessions where the activities continue after the login attempts. Table 4 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 duration includes a constant time of 700ms for launching the agent. Many sessions requires 800ms i.e. uses only 100ms for the algorithm run. The largest session takes, a longer time, up to 2300ms. This 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 big. This increases abnormally the computation time for this session.

	Small	Average	Large	Huge
Length	800	1500	3000	15700

Table 4: SSR Computation Time (in milliseconds)

For one year of experiments, the average time is 1500ms with our *grid* configuration. With 40,000 identified sessions, it took on average 190 hours to analyze one year of logs. So, it takes half an hour to analyze the logs of the day and 20ms for 1mn of logs. As one can see, it is possible to reconstruct sessions in real-time with our architecture.

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

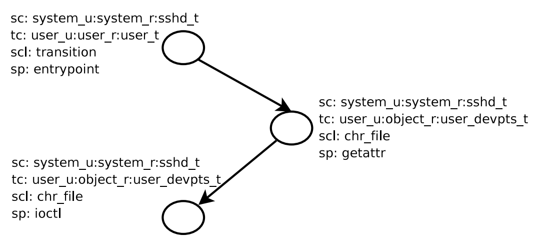
sc: system_u:system_r:sshd_t
tc: user_u:user_r:user_t
scl: transition
sp: entrypoint
sc: system_u:system_r:sshd_t
tc: user_u:object_r:user_devpts_t
scl: chr_file
sp: getattr
sc: system_u:system_r:sshd_t
tc: user_u:object_r:user_devpts_t
scl: chr_file
sp: ioctl

Figure 8: Example of a Learned (Basic) System Pattern

System Pattern Learning: As we said in the System Pattern Learning section, the rough complexity of the construction of the system patterns is uncomputable. However, the study of the data from our honeypot showed us some directions we could exploit. 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 $a = 7$. 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 40000 sessions, only 72 had above the average number(i.e. 600 nodes per session). Our first experiments build 30 patterns after a computation of 3 weeks manually ended.

As previously said, we are currently investigating bioinformatics algorithms (like BLAST) and also unsupervised classification and learning technics to overcome this (sub-)graph clustering problem.

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

The figure 8 shows a 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.

6 different levels of similarity have been considered for the recognition of 40,000 sessions (see the System Pattern Learning section). As seen in table 5, 13040 sessions contain this pattern for a similarity level 2 i.e. the comparaison is based only on security contexts (source and target). For level 3 (addition of one variable), 13000 sessions respect this pattern, and only 6680 sessions for level 4 (addition of one variable). Table 5 shows also that the recognition did not find any session respecting the pattern for higher levels of similarity. The maximum level of similarity can be increased using other patterns. Actually, the learning module is too slow to compute

enough and accurate patterns, that's why we are currently working on bioinformatics algorithms.

Similarity	**OK**	**Not OK**
2	13040	26960
3	13000	27000
4	6680	33320
5	0	0
6	0	0
7	0	0

Table 5: Classified System Patterns

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 ressources. So, it is really hard to conduct advanced attacks on such systems. However, one can see that possibilities to violate the security exist.

CONCLUSION

This paper presents several modules that cooperate to analyze complete sessions of SELinux system activities. Complex sessions can be reconstructed such as distributed, migrating sessions using several connections. Currenlty, 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, have been recognized. Moreover several complex sessions have been completly reconstructed. Using a *grid* approach, a real-time classification of system sessions has been proposed. The solution uses System Patterns in order to analyze the logs. A learning module is proposed to compute automatically the relevant patterns starting from the reconstructed sessions. The paper shows that rough Pattern learning is a NP-Complete problem. However, heuristics have been proposed and real constructed patterns have been presented. Future works will address how to use the systems sessions to defined advanced security properties such as integrity models or to detect the violation of those properties. Applications will be proposed either for protecting a system of for detecting the intrusions. Moreover, several improvements will be proposed. First, newer parallel processing could reduce the required computation time. Second, newer heuristics will be defined to compute the Activity Patterns.

References

Altschul, S., Gish, W., Miller, Myers, E., and Lipman, D. (1990). Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410.

Bowen, R., Chee, D., Segal, M., Sekar, R., Uppuluri, P., and Shanbag, T. (2000). Building survivable systems: An integrated approach based on intrusion detection and confinement. In *DARPA Information Survivability Symposium*. IEEE Computer Society.

Briffaut, J., Lalande, J.-F., Toinard, C., and Blanc, M. (2006). Collaboration between mac policies and ids based on a meta-policy approach. In Smari, W. W. et McQuay, W., editor, *Workshop on Collaboration and Security (COLSEC'06)*, page 48–55, Las Vegas, USA. IEEE Computer Society.

Chari, S. N. and Cheng, P.-C. (2003). Bluebox: A policy-driven, host-based Intrusion Detection System. *ACM Transaction on Information and System Security*, 6(2).

Cuppens, F. and Miège, A. (2002). Alert correlation in a cooperative intrusion detection framework. In *IEEE Symposium on Security and Privacy*, Oakland, USA. IEEE.

Eckmann, S., Vigna, G., and Kemmerer, R. (2002). STATL: An Attack Language for State-based Intrusion Detection. *Journal of Computer Security*, 10(1/2):71–104.

Eppstein, D. (2000). Diameter and treewidth in minor-closed graph families. *Algorithmica*, 27:275–291.

Kruegel, C., Valeur, F., and Vigna, G. (2005). *Intrusion Detection and Correlation: Challenges and Solutions*. Springer.

Krugel, C., Toth, T., and Kerer, C. (2001). Decentralized event correlation for intrusion detection. In *Information Security and Cryptology*, pages 114–131.

Molina, J., Chorin, X., and Cukier, M. (2007). Filesystem activity following a ssh compromise: An empirical study of file sequences. In *ICISC*, pages 144–155.

Ning, P., Reeves, D., and Cui, Y. (2001). Correlating alerts using prerequisites of intrusions. Technical Report TR-2001-13, North Carolina State University.

Qin, X. (2005). *A Probabilistic-Based Framework for INFOSEC Alert Correlation*. PhD thesis, Georgia Institute of Technology.

Valeur, F., Vigna, G., Kruegel, C., and Kemmerer, R. A. (2004). A comprehensive approach to intrusion detection alert correlation. *IEEE Transactions on dependable and secure computing*, 1(3).