A New Approach to Enforce the Security Properties of a Clustered High-Interaction Honeypot

J. Briffaut, J. Rouzaud-Cornabas, C. Toinard, Y. Zemali
ENSI de Bourges
LIFO — EA 4022
88 Bld Lahitolle
18020 Bourges Cedex, France
Email: {jeremy.briffaut,jonathan.rouzaud-cornabas,christian.toinard,yacine.zemali}@ensi-bourges.fr

ABSTRACT

This paper enlarges previous works of the authors related to the security of a high-interaction honeypot. The challenge is to have a Security Property Language (SPL) for defining the required properties for controlling the activities between processes and resources. That language must authorize the definition of security properties related to confidentiality, integrity and availability. Moreover, that SPL must be able to enforce the security of target Operating Systems. It is an open problem not only regarding the security of Operating Systems but also regarding the security of high-interaction honeypots. That paper shows that existing approaches really fail to manage a large range of security properties. The first reason is that a SPL is really missing to express and enforce a large set of security properties. The second reason is that protection and detection approaches fail to manage a large set of security properties. Our paper proposes PIGA-Protect a new approach to control the system calls in order to guarantee the requested security properties. It goes much further than existing systems, since a language is proposed for expressing a large range of security properties. That language authorizes to enforce security properties that cannot be easily managed with other approaches. Finally, security properties are proposed for a high interaction honeypot. The goal is to capture information about attacks without compromising the system and without restricting too much the attacker activities. In this case, the attacker must get sufficient privileges without corrupting the system. For that purpose, the formalized properties are taken into account by 1) PIGA-Protect to prevent from corruption and by 2) PIGA-IDS to detect malicious activities and to ease the attack analysis.


1. INTRODUCTION

A high interaction honeypot is required to capture host based attacks and to offer the vulnerabilities of the target system. As each operating system, distribution, service, software can contain different vulnerabilities according to their version, the deployment of a large heterogeneous cluster of honeypots is required to increase the number of possible attacks.

As attackers have a direct access to a real system and can exploit the vulnerabilities of the target system, they can gain administrator privileges and compromise easily the system.

The main problems related to the administration of high interaction honeypots are:

- an attacker must get sufficient privileges to conduct attacks
- the operating system corruption must be prevented
- the administrator must get a precise control of the security properties that are required to secure the honeypot
- the requested security properties must be automatically enforced on the target Operating Systems
This paper proposes an advanced SPL for securing high-interaction honeypot. The main objective is to provide an open platform for expressing security properties and to use it for securing the different Operating Systems of our honeypot. The difficulty is to have security properties that prevent from system corruption while authorizing a large variety of attacks. Deploying high-interaction honeypot is a challenging research activity and few works address the SPL that could fit to secure them. Classical approaches for enforcing the security of the Operating Systems present several limitations. First of all, an SPL is missing to ease the definition of the requested security properties. Second, the support for a large range of security properties is poorly addressed. Finally, security enforcement does not deal with the security requirements of high-interaction honeypots.

2. STATE OF THE ART

2.1. SECURITY PROPERTIES

Currently, one cannot find any operating system able to harden a large range of security properties. Under Linux, there are at least four security models available to ensure a mandatory access control policy: SELinux [1], GRSecurity [2], SMACK and RSBAC. But none of these solutions can ensure a large set of security properties. In the best case, they can ensure one or two limited properties such as the Bell and Lapadula [3] confidentiality or the Biba [4] integrity. Under the BSD family, solutions such as Trusted BSD (available within the following Operating Systems: FreeBSD, OpenBSD, MacOSX, NetBSD) provide more or less the same kind of a Mandatory Access Control than SELinux. But, again they fail to ensure the large majority of requested security properties.

The major limitation is associated to a transitive closure of allowed system calls that enables to violate a security property. All the existing approaches fail to manage causal relationships between system calls, authorizing thus illegal activities.

[5] counts more than 150 publications related to information-flow security. The majority deals with non-interference. The other part aims at enforcing information-flow policies using program analysis techniques. Information flow analysis in a program language does not fit to ensure security policies between several processes that are using the services of an Operating System.

Several works address how to enforce security properties within an Operating System. [6] considers only protection policies and enforcement mechanisms such as the ones considered in MAC systems. Thus, he does not deal with information flows and more generally does not take care of confidentiality nor of integrity but of safety. Solutions such as [7][8] consider how to detect flows between privileged and unprivileged entities. Such solutions can enforce both integrity and confidentiality between those two classes of entities. However, they cannot formalize other classes of security properties such as preventing flows between privileged entities. Solutions such as [9][10] present several limitations since definition of security properties is not possible. This explains why say that security properties are not really supported in practice. In such solutions, security relies on the ability of administrators to define good protection policies without any mechanism to administrate the required properties. Moreover, the current approaches such as [9][10][11], address the information flows but they cannot take into account complex combinations of a multiple flows. Finally, none of those approaches provides any SPL to ease the formalization of the requested security properties.

Many solutions deal with detecting violations of security properties. Solutions such as [12] detect violation of both confidentiality and integrity between privileged and unprivileged entities. Again, those solutions consider only non-interference and cannot enforce other classes of security properties. [13] claims to enforce both integrity and confidentiality properties, but they do not give any formalization of the considered properties. In practice, [13] ensures only the access consistency which is only a limited case of integrity or confidentiality property.

2.2. HONEYPOTS

High-interaction honeypots deploy real operating systems [14]. Generally high-interaction honeypots use vmware to emulate the host [15] but hiding [16] and securing the hypervisor is very hard to achieve. Moreover, the enforcement of the security of the target Operating System is not addressed.

In contrast with recent generation of honeypots described in [17], our work goes one step further. We improve the administration and the security enforcement to get a new generation of honeypot using a new approach to guarantee the required security properties of the target Operating Systems.

Paper [18] presents how to configure honeypots dynamically based on network scans. It improves the initial work
of Hieb and Graham. (the opened ports) to the detected attacks. In [19] the authors tries to build an honeypot that can be plugged into a local network and that find automatically unused IP. These works allow a system administrator to deploy an honeypot according to his network configuration and to let the honeypot evolves and adapts the services with the requests of attackers. These goals are crucial when deploying some honeypots on a large scale network with thousands of hosts but the choice of the operating system and the security of the operating systems remains an opened question that we want to address with this paper.

Some papers address the techniques to detect that a host is an honeypot [20, 21]. These techniques are more or less related to kernel analysis that allows to detect a User-mode Linux or a VMware host. This is precisely these papers that shows that deploying real operating systems with real services is better for capturing attackers, but is possibly more dangerous for other hosts on the network.

In [22], the authors propose an advanced hybrid type of honeypots: shadow honeypots. They analyze the network traffic in order to redirect suspicious packets to a shadow honeypot. This shadow honeypot is a replication of the protected software that can be used to analyze the received attacks. These types of solution are complex to deploy and our purpose is to collect more information by letting intruders attack directly the target host while enforcing their security.

3. THE SECURITY PROPERTY LANGUAGE

Our SPL enables the definition of security functions that can be seen as macros formalizing generic security properties.

Our language also includes general collections such as $SC$ (the whole set of security contexts, i.e. all the entities of the system), $SSC$ (the whole set of subject security contexts i.e. all the subjects of the system), $OSC$ (the whole set of object security contexts i.e. all the objects of the system) and $IS$ (the whole set of possible operations, i.e all the system calls available on the system).

Our language includes structures of control such as iteration associated to the $\text{Foreach}$ keyword. It also includes logical operators such as $\text{and}$, $\text{or}$, $\text{not}$ and $\circ$ (composition), statistical operators such as $\text{average}$ and $\text{deviation}$ or temporal operators such as $\text{within}_{\text{time}}$ and $\text{at}$. Finally, it provides assertions such as $\text{SuchThat}$ or $\text{Different}$ to evaluate conditions involving different relationships between contexts. Several types of relationships are supported such as $\rightarrow$, $\Rightarrow$, $>$, $\gg$, $\rightarrow\text{trans}$ for expressing an interaction, a causal dependency, a direct information flow, a transitive information flow between two contexts or a transition from a context to another, respectively.

Since our language can take into account all the system calls but also the causal sequences, it can prevent from covert channels. For example, a sequence of signals can be used by several processes to transfer information between them. Obviously, there are remaining covert channels, such as the ones that cannot be observed at the system call level (e.g. CPU channels or Direct Memory Access).

In our language, the definition of a security function begins with the keyword $\text{define}$. Such a function requires a signature using parameters which are prefixed by a $\$ and a piece of code associated with that signature. The code of a security function uses the different language constructors i.e. operators, local variables, relationships between two contexts and collections.

3.1. DEFINITION OF SECURITY PROPERTIES

Usually, a piece of language starts with the definition of a generic property as a function including a prototype and the corresponding code.

Each function that defines a security property, contains three parts:

- the function signature: the definition of the required parameters which are usually a set of security contexts;
- eventually, a section of code: a sequence of contexts using our language structures of control i.e. iterations, logical, statistical or temporal operator;
- assertions: conditions involving several relationships between contexts in order to compute the associated illegal activities.

For each call of a function, the set of the activities in contradiction with the considered security property, i.e. the computed assertions, can thus be deduced starting from the existing MAC policy. It is the task of our compiler to compute each function call in order to add into the kernel database all the corresponding illegal activities.

Listing 1 gives two examples of security functions expressed using our language. First of all, a function $\text{(integrity)}$ guarantees the integrity of an object context $sc2$ (file, directory, socket, etc.) with respect to a subject context $sc1$ (process). This property is made up of two language constructions:
• any operation $\$eo$, associated to an interaction $\$sc1 \rightarrow \$sc2$ between two contexts $\$sc1$ and $\$sc2$, must not be a writing like operation (is_write_like);

• any operation $\$eo$, corresponding to the last operation of a causal transitive closure $\$sc1 \Rightarrow \$sc2$ between two contexts $\$sc1$ and $\$sc2$, must not be a writing like operation (is_write_like).

The second security function, confidentiality, corresponds to the confidentiality of an object $\$sc2$ regarding to subject $\$sc1$. This property allows to check that $\$sc1$ will not be able to obtain information directly, for example via a reading, or indirectly, via a causal transitive closure combining readings and writings, in order to form an information flow. This property is made up of two assertions:

• any activity $\$sc2 > $\$sc1$, corresponding to an information transfer from $\$sc2$ to $\$sc1$ i.e. a direct transfer between those two contexts, must not exist;

• any activity $\$sc2 \gg \$sc1$, corresponding to an information flow from $\$sc2$ to $\$sc1$ i.e. a causal transitive transfer between those two contexts, must not exist;

Let us note that these two prototypes must be called with relevant parameters in order to define the spectrum of corresponding activities. Indeed, the prototype does not permit to compute illegal activities. It requires a concrete instantiation to deduce the activities using the contexts that are passed as parameter values.

Listing 1. Examples of security properties

```
define integrity( $\$sc1 IN SSC, $\$sc2 IN OSC ) {  
   Foreach $\$eo IN IS  
   suchThat ( $\$sc1 \rightarrow ( $\$eo ) $\$sc2 )  
   { not(is_write_like($\$eo)) };

   Foreach $\$eo IN IS  
   suchThat ( $\$sc1 \Rightarrow ( $\$eo ) $\$sc2 )  
   { not(is_write_like($\$eo)) };

}

define confidentiality( $\$sc1 IN SSC, $\$sc2 IN OSC ) {  
   suchThat ( $\$sc2 > $\$sc1 ),  
   { not(exist()) };

   suchThat ( $\$sc2 \gg \$sc1 ),  
   { not(exist()) };

}
```

3.2. INSTANTIATION OF SECURITY PROPERTIES

A security property instantiation corresponds to the call of a security function. Calling a security function instantiates a property through the definition of the required security contexts as the parameters values. Each function call makes it possible to extract, from the SELinux policy of the target system, the whole set of the illegal activities in contradiction with the requested security property. PIGA-Protect uses then the set of illegal activities, i.e. the database of the activity patterns, in order to prevent the violations of that security property.

Listing 2 formalizes three security properties using the two security functions from listing 1. The first property allows to guarantee the integrity of the executable files, i.e. the Linux binary applications, regarding to the end-users. Thus, an end-user will not be able to modify a binary file either directly or indirectly by using a causal sequence (for example via an intermediate su aiming at modifying a binary file).

The second property aims at guaranteeing the confidentiality of files containing passwords (/etc/shadow) regarding to unprivileged users. Thus, an ordinary user cannot obtain, directly (via a reading) or indirectly (via an information flow), information from this file. This is an advanced property since usually confidentiality is guaranteed only against direct access but not against a causal transitive closure involving possibly several processes but leading to an information flow. Thus, information flows are prevented.

4. PIGA-PROTECT

4.1. PROTECTION ARCHITECTURE

Our control model is divided into two stages as depicted in figure 1. The first one allows the definition of security properties through our SPL. A compiler, PIGA-CC, analyzes a mandatory policy (SELinux or GRSecurity) in order to extract all the forbidden activities associated with requested security properties. The formalized security properties are used by the PIGA-CC compiler that will analyze them in order to generate the list of forbidden activities. For each security property instantiation, PIGA-CC enumerates paths into various graphs associated with the mandatory policy. Each path corresponds to a possible violation of a required security property. Thus, the compiler enumerates all the forbidden activities i.e. all the sequences of system calls connecting a source and a target context. The compiler sends those illegal activities to PIGA-Protect in order to prevent the violation of the requested security properties.

Then, a second part, called PIGA-Protec, enforces those
security properties at the operating system level. PIGA-Protect captures the related system calls. Each system call is thus suspended and waits for an authorization or a deny response. PIGA-Protect allows or denies the considered system call aiming at preventing the occurence of the forbidden activities. Thus, the system call fails if its execution could lead to the violation of the security properties. For that purpose, PIGA-Protect reconstructs the system activities and compares them with the precomputed database including all the possible illegal activities.

In our solution, the system call is controlled regarding first the classical DAC policy, then the MAC policy (e.g. SELinux), and finally the requested security properties. This latter control is performed by PIGA-Protect. So, our approach allows to combine existing controls such as DAC and MAC with our PIGA-Protect approach.

4.2. EXAMPLE OF PREVENTION

Listing 3 gives an example of a session trying to violate the confidentiality property of listing 2. An application exploit enables to exploit a Buffer-Overflow on the application passwd in order to force it to copy the file /etc/shadow (which is readable by this program) in /tmp/ and to fix the correct rights for the user. When this program is called, line 1, the user could try to obtain a copy of this file in /tmp (line 3). But, when the user tries to access to the file copy-shadow, the operation of reading is refused by PIGA-Protect. A trace corresponding to this violation attempt is thus generated by PIGA-Protect (see listing 4). This trace shows that the activity, which could cause the violation, corresponds to an information flow from the file shadow towards the application passwd (via a reading), then the program passwd wrote the contents of this file into a temporary user file. But, when the user tries to read this file, PIGA-Protect refuses this last system call. It should be noticed that all legal activities are permitted. It is only the last system call, corresponding to the end of an illegal activity, that fails. In other words, legal activities could not end with an illegal system call.

Listing 3. Prevention of a flaw from a password file

```
briffaut@pigaos ~/$ ./exploit passwd
/etc/shadow copied in /tmp
briffaut@pigaos ~/$ ls -lh /tmp/copied-shadow
-rw-r--r-- 1 briffaut users 313K 28 dec 21:57 copied-shadow
```

Listing 4. Prevented violation of a confidentiality property

```
Denied confidentiality3$110 system_u:object_r:shadow_t--->user_u:object_r:passwd_t--->user_u:
object_r:unknown_policy
```

5. EXPERIMENTATION

5.1. EXPERIMENTATION ON A HONEYPOT

This section presents a usage of our solution for securing a high-interaction honeypot. The security properties defined by the administrator are then used either by PIGA-Protect that is described in this paper, and the other ones are used by PIGA-IDS such as described in [24]. But, that paper presented only the analysis of the malicious activities but neither the language nor the security properties permitting that analysis. So, this section will present only the major security properties that have been formalized, which ones are used by PIGA-Protect and which ones are used by PIGA-IDS. The analysis of the malicious activities can
be found in [24] and are not presented hereafter. For securing our honeypot during 2 years, we defined several security properties such as those of listing1 in order to 1) highly protect all the nodes of our distributed honeypot and 2) detect the malicious activities of the intruders. Several advanced security properties have been formalized using our language. However, for briefly only some of them will be detailed. Finally, the number of the possible violations for the requested security properties, allowed by the associated SELinux Policies, are presented.

5.1.1. Monitoring Security Properties

Listing 5 shows some security functions required for highly monitoring our honeypot. The first property, domain_integrity (line 1) expresses that any logically ‘ch-rooted’ context can not interact with any other context outside its own domain. Typically, that first property will be used by PIGA-IDS but not by PIGA-Protect. Otherwise, PIGA-Protect would prevent any modification from the user context to another context, which is a non sense in a honeypot since it limits too much the intruder privileges.

The second property, duties_separation expresses that a first context can not be executed by a second one if this first context has just been modified by this second one. Again, that property is too wide to be used as a protection property. But, it will help to observe any installation of a new program.

The next listing 6 shows some properties instanciated for monitoring our honeypot i.e. for use of PIGA-IDS.

The first one monitors the activities of the users outside their own context. The second one captures each user attempt to get information from system (e.g. configuration). The third one permits to detect any installation and execution of programs.

Listing 5. Security properties

```
define domain_integrity( $Chroot IN SC ) [ 2
  Foreach $eo IN IS, Foreach $sc1 IN $Chroot, Foreach $sc2 IN SC
  suchThat { ( $sc1 -> { $eo } $sc2 ) } ];

define duties_separation( $sc1 IN CS ) [ 9
  suchThat $sc1 IN IS, Foreach $eo IN IS
  Foreach $sc2 IN $Chroot
  suchThat { ( $sc1 -> { $eo } $sc2 ) } ];
```

5.1.2. Protection of Security Properties

Listing 7 shows some security functions required for enforcing the security of our honeypot. Those security functions will be used to implement security properties that will be enforced by PIGA-Protect. The goal is to prevent from the corruption of the system by the attackers otherwise re-installation is needed and monitoring of illegal activities can be compromised.

The first security function aims at enforcing that each execution request deals with a binary object coming from a trusted path. The second one ensures integrity of all the based system binaries with an exception for local login of a system administrator. This property is one of the major security property that will prevent from a system corruption without dissallowing the local updates of the system by the regular administrator. The third functions guarantees that an indirect access (i.e. associated to a causal sequence of system calls) is possible only if the corresponding direct access is allowed.

Listing 6. Security rules instancing security properties

```
domain_integrity( $Chroot:=".*::.*::user.*" );
confidentiality($sc1:=user_u;user_r;user_t, $sc2 :="system_u:object_r:=");
duties_separation($sc1:=".*");
```

Listing 7. Security properties

```
define trustedpathexecution( $TPE IN CS ) [ 7
  Foreach $sc1 IN CSS, Foreach $sc2 IN CS,
  Foreach $eo IN IS
  suchThat { ( $sc1 -> { $eo } $sc2 ) } ];

define executable_integrity($EXEC IN CS) [ 14
  Foreach $sc1 IN CSS, Foreach $sc2 IN CSS,
  Foreach $eo IN IS
  suchThat { ( $sc1 -> { $eo } $sc2 ) } ];

define access_consistency($sc1 IN CS, $sc2 IN CS) [ 17
  suchThat { ( $sc1 >>> $sc2 )
    { exist($sc2 > $sc1) } ];
```
The next listing 8 shows some properties instanciated for preventing our honeypot from corruption i.e. for use of PIGA-Protect. The first property guarantees a trusted path execution for all the object files associated with the bin,t, exec,t, lib,f or ld_so,t contexts. The second property prevents any modification of the executable files except if the modification comes from a local privileged login i.e. the modification is forbidden when it is associated with a causal sequences starting from ssh. The third property guarantees access consistency for the any subject associated with ordinary users. So, covert channels are prevented for ordinary users. The fourth and fifth properties prevent from information flows coming from shadow password files or ossim files to ordinary users. The fourth property uses a flow level that is less sensitive. Indeed, a flow level of 8 limits the system call to the read and write like operations. The last property uses a flow level with a greater sensitivity. A flow level of 1 takes into account all the system calls including the system calls associated to the transmission of Unix signals.

Listing 8. Sample patterns compiled by PIGA-CC

<table>
<thead>
<tr>
<th>Line</th>
<th>Pattern Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>trustedpathexecution</td>
</tr>
<tr>
<td>2</td>
<td>executable_integ..</td>
</tr>
<tr>
<td>3</td>
<td>access_consistency</td>
</tr>
<tr>
<td>4</td>
<td>FLOW_LEVEL=8;</td>
</tr>
</tbody>
</table>

5.1.3. EXAMPLES OF PREVENTED ATTACKS

Figure 2 describes a limited subset of the interaction graph associated with the SELinux access control policy. This limited subgraph aims at illustrating only some scenarios of attacks associated with the proposed security properties.

Let us consider an attacker that gets an illegitimate login from a user account. That attacker creates a directory with corrupted executables that will open a back door when they are executed by the legitimate user. For this purpose, he adds that fake directory to the user path variable. Without the trustedpathexecution property, the legitimate user will be fooled and the attack succeeds. In contrast with the other properties, it should be noticed that this property can be removed to open the honeypot to illegitimate actions. So, one can see that the honeypot is easily managed using our security management language.

The executable_integ...y is a mandatory property. It authorizes a sequence of actions starting with the login_d context and passing through the admin_d context to modify a bin,t object. In contrast, a sequence of actions beginning with the ssh_d context or passing through the user_d context cannot modify a bin,t object. That property enables a local administrator to add new binaries but prevent illegal updates.

The confidentiality with flow level 8 authorizes a signal between passwd_d and ssh_d. It is mandatory since a successful connexion from ssh requires a signal transmission between passwd_d and ssh_d. That property is not completely safe since it authorizes a covert channel using the signal between those two contexts. However, that covert channel is formally permitted. So, it is not an unknown or illegal covert channel. Moreover, PIGA – IDS is able to generate an event associated to the signal and thus that remaining covert channel can be detected.

The confidentiality with flow level 1 prevents an information flow between ossim_file,t and user,t. This flow must be fully prevented since the ossim_file,t contains a clear password for the ossim database. So, that property prevents from covert channel by taking into account all the direct or indirect flows. The consequence is a possible failover for illegitimate ossim usage. But, it is mandatory to limit the ossim application to a logging facility.

5.1.4. COMPILATION RESULTS

The table 1 shows the number of illegal activities computed by PIGA-CC. More precisely, we can see the number of illegal activities for three different hosts (i.e. an Internet Gateway, a honeypot with X server, a honeypot with System Services). For example, we can see that the MAC
policy on the Gentoo hosts leaded to 29,510 possible violations of the confidentiality property. The big difference between Gentoo and Debian hosts was because the Debian system includes a graphical interface with multiple graphical applications. We see that the computation of the illegal activities took only about 50s for the Gentoo policies while taking more than 10 minutes for Debian. Indeed, more than 1,300,000 activities are compiled on for that host.

One can see that a very large number of possible violations exists for a given SELinux system. So, PIGA-Protect is able to enforce those security properties in order to prevent the possible violations of the SELinux system. Indeed, SELinux protects efficiently against direct accesses but cannot manage flows or complex security properties. Moreover, PIGA-IDS is very efficient in reporting the violations of the considered security properties. [24] shows that PIGA-IDS eases the analysis of the illegal activities. Indeed, since each violation is associated to a well formalized security property, the PIGA-IDS events ease forensic and analysis of attacks.

<table>
<thead>
<tr>
<th>Graph</th>
<th>Gateway (Gentoo)</th>
<th>Workstation (Debian)</th>
<th>Server (Gentoo)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security Contexts (nodes)</td>
<td>577</td>
<td>3,017</td>
<td>595</td>
</tr>
<tr>
<td>Interactions (edges)</td>
<td>17,684</td>
<td>314,582</td>
<td>18,215</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Security Property</th>
<th>Rules</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>executable_integrity</td>
<td>137</td>
<td>9,461</td>
</tr>
<tr>
<td>domain_integrity</td>
<td>16,283</td>
<td>510,215</td>
</tr>
<tr>
<td>confidentiality</td>
<td>29,510</td>
<td>726,842</td>
</tr>
<tr>
<td>duties_separation</td>
<td>243</td>
<td>16,405</td>
</tr>
<tr>
<td>access_consistency</td>
<td>171</td>
<td>260</td>
</tr>
<tr>
<td>trustedpathexecution</td>
<td>362</td>
<td>675</td>
</tr>
<tr>
<td>Size of the database</td>
<td>1,1 Mo</td>
<td>3,6 Mo</td>
</tr>
<tr>
<td>Nb of audit</td>
<td>1,064</td>
<td>44,303</td>
</tr>
<tr>
<td>Computation time</td>
<td>47s</td>
<td>10 min 31s</td>
</tr>
</tbody>
</table>

Table 1. Database of illegal activities produced by PIGA-CC

6. CONCLUSION

The paper presents a new Security Property Language that enables to formalize advanced security properties related to integrity, confidentiality or availability. That language has been experimented in real conditions to develop an advanced high-interaction honeypot.

In previous works of the authors, analysis of the attacks occurring during two years of experimentation has been detailed. That analysis was easier due to PIGA-IDS that detects the violations of any security property that can be formalized using our language.

That paper gives the inner mechanisms related to the security of our honeypot since it describes 1) our Security Property Language, 2) the security properties required for our honeypot, 3) PIGA-Protect, a control mechanism implemented within the SELinux Kernel in order to guarantee the security properties i.e. to prevent the violations of the requested security properties and 4) the security properties required for PIGA-Protect and PIGA-IDS.

PIGA-Protect is a powerful mechanism for preventing the violations of a wide range of security properties. It has been validated during two years of experimentation since despite more than 2 millions of attacks the target systems has never been corrupted. In contrast with other approaches that cannot take into account combinations of activities between processes and ressources, our SPL enforces easily a large variety of properties. It enables to formalize all the security properties from the literature and support the definition of new properties.

Finally, our SPL provides a open method to deal with complex security properties. It supports heterogeneous target technologies. Despite an application is presented for SELinux, our SPL can enforce security of other infrastructures such as XACML. Indeed, it can reuse any protection policy in order to compute all the activities that can violate the requested properties. New usage of our SPL can be defined in the future to improve the security of GRID infrastructures.

REFERENCES


