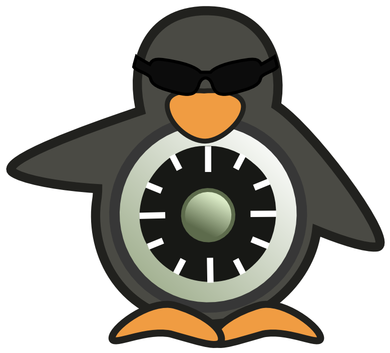
Integrating PIGA-MAC into the Linux kernel

3rd year STI research project
2011 – 2012

Teacher: Jérémy Briffaut
Student: Timothée Ravier



Acknowledgments

I would like to thank Jérémy Briffaut for the many advices he gave me. This project is based on his thesis and would not have been possible without his work.

I would also like to thank Martin Peres as he introduced me to the PIGA concept and kernel development.

Contents

List of Figures

Glossary

AppArmor	*Application Armor:* a MAC mechanism included in the Linux kernel
DAC	*Discretionary Access Control*
HRU	Theorical model representing DAC systems
IDS	*Intrusion Detection System:* a device or software that monitors network and/or system activities for malicious activities
IPS	*Intrusion Prevention System:* an extension of an IDS that also tries to block or prevent intrusions that are detected
LSM	*Linux Security Modules:* Hooks inserted in the Linux kernel in order to implement modular access control
MAC	*Mandatory Access Control*
MCS	*Multi Categories Security*
MLS	*Multilevel Security*
PaX	Linux kernel patch including support for non-executable memory pages
PIGA	*Policy Interaction Graph Analysis*
SELinux	*Security Enhanced Linux:* a MAC mechanism included in the Linux kernel
SMACK	*Simplified Mandatory Access Control Kernel:* a MAC mechanism included in the Linux kernel
TOMOYO Linux	A MAC mechanism included in the Linux kernel

Introduction

This project is an integral part of my formation at the ENSI de Bourges, a computer science and security french engineering school.

PIGA (Policy Interaction Graph Analysis) is an advanced Mandatory Access Control mechanism designed to enforce security policy on a system. Just like SELinux, PIGA does not directly prevent vulnerability exploitation, but prevents any further action an attacker might attempt to perform on the system.

The current PIGA design faced unexpected problems and some of them are related to the fact that the decision making code is located in userspace. That's the main difficulty that we are trying to address with this kernel port. Among the other problems that I will have to solve, they are multi-core and multi-processor related issues. I will also try to improve performance and ease of use while trying to port the complete PIGA feature set, which is an ambitious goal.

I will begin with a short description of all currently available protections for the main open source operating systems such as Linux distributions and some from the BSD family. Then I will proceed to some PIGA reminders, kernel/userspace implementation design. Finally, I will discuss some of the problems I faced during those six months and potential solutions and improvements for further development.

1 State of the art

1.1 Context and study limits

There is a wide range of computer security problems, from hardware specific ones to sotfware related flaws. New vulnerabilities are discovered every day and they occur in every part of modern and complex operating systems. Current mainstream systems are built around a core, or kernel, which controls and abstracts hardware interactions that are provided to the applications. The kernel has full control on the hardware and is often trusted to ensure that each application running in userspace is separated from the others.

In this project, we will not focus on kernel security or kernel hardening (despite the name PIGA-Linux). The main goal of PIGA is to go further than SELinux, in order to enforce security properties on an operating system. So I'll focus my study on mechanisms which allow a user or an administrator to control permissions given to a program running on a computer. It could be summarized as an advanced in-kernel control of userspace applications behaviours.

Thus, I will use several restrictions to further constrain this study:

- We are limited to the operating system; we do not take into account interactions with an other system on the network for example;
- We want the solution to be as architecture independant as possible; we should not require specific hardware design to work properly;
- As we do not take into account the underlying hardware, we do not focus on hardware flaws either and therefore do not study hardware based attacks (defective network card or driver...);
- We should focus on systems which have the source code available for study.

Most of current operating system are using an access control design to ensure minimal security, seperation and control on user's activities. They were introduced to limit the effects of an attack against the operating system. But this limited control does not prevent complex attack scenario, such as the compromission of a service which was given adminstrative rights on the system. [1] describes in details most of the problems current operating systems are still facing nowadays.

Generally speaking, we lack a global, truly enforced, system wide security mechanism. More than simple checks in interactions between kernel and userspace, it's the whole interaction set on a system that need to be controlled.

First I'm going to detail the objectives an operating system has to achieve. Then I will discuss theorical models and their implementations in modern systems.

In this study, I will refer to files (and others ressources such as sockets) as objects, and to program as subjects, whether or not they are directly controlled by a real human user on the system.

1.2 Generality

1.2.1 Security objectives

There are three main objectives generally defined in computer security articles and governmental processes [2]. An operating system has to provide means to ensure they can be enforced:

Confidentiality: Information must only be available to users who need it and have the corresponding privilege. Example: a user should not have access to other users passwords, files or secrets; classified information should not be available for unclassified users to read.

Integrity: The system must remain in coherent state. Example: important information such as passwords, which programs are installed and their content, configuration files, should not be alterable by unprivilege users.

Availability: The system must remain stable, reactive and usable. Programs have to function properly. Added security controls should limit as little as possible legitimate system functionnality and programs.

I will focus on the first two properties, which are confidentiality and integrity. Attack scenario such as Denial of Service will therefore not be taken into account as their goal is to limit a system disponibility. Future references to security objectives will include only those two.

However, a security solution should not turn a working system to an unsable one and thus it can not have a high ressource footprint. I'll try to limit the impact on performance of the chosen solution.

1.2.2 The principle of least privilege

The process to make the first step toward the previously discussed objectives is based on the principle of least privilege. It involves two steps:

1. make sure each major program functionnality is divided into smaller program (generally depending on their ressource needs);
2. restrict the rights of each program to the smallest set of permissions needed for it to function well.

The most common examples are the postfix and qmail mailer daemon. They are divided in several programs and communicate using files or SystemV IPC. There is a network facing program, a user facing program and the code running under a privilege user is therefore severely limited, which, as a side effect, improves it's auditability. This way a vulnerability will only impact one program which has a limited permission set. It would require several flaws in each different part to gain enough privilege to affect the system.

Generally speaking, a program should not have acces to a ressource if it doesn't use it, even if it isn't a confidential ressource.

On modern operating system, this separation is often enforced using different user accounts with different capabilities. There is an administrator which has all rights on the system, several unprivileged account for regular users, and network enabled accounts for services such as the web server.

However, few programs are well designed enough to benefit from this rule. It also doesn't provide a solution in case of program compromission so this isn't enough to enforce security objectives.

1.2.3 Who/what do I need to trust?

Every design made to secure an operating system has to rely, at some point, on one of it's components (either a hardware one, a software one, or both). This component is trusted to be safe and hardened against attacks. This is summarized in this list:

- Trust all the software to abide by a security policy while being aware that the software is not trustworthy;
- Trust all the software to abide by a security policy and the software is validated as trustworthy (by static analysis, unit testing, branch analysis);
- Trust no software but enforce a security policy with mechanisms that are not trustworthy;
- Trust no software but enforce a security policy with trustworthy hardware mechanisms.

Standard security models are in the first category whereas most of the one I will present later are in the third and forth ones.

Testing the whole feature set of an operating system is a tedious task. This is even harder if we consider the various architectures,hardware diversity and the complexity involved in the design of core pieces of an operating system such as the kernel. Some software, such as the QNX operating system [3], have had intensive testing, a controlled design and are certified as conforming to strict criteria like the Common Criteria [4]. Those systems are part of the second category. However, the source code for those operating systems are not freely available so I will not detail them much further.

The Figure 1 [5] shows an example of hardware based integrity control, with corresponds to the fourth category. Its goal is to verify the signature of critical components in the system to make sure that they have not been tampered with before boot. If the check fails, the system is unable to boot.

The Linux Integrity Subsystem [6] is an other example of hardware assisted integrity control, involving a Trusted Platform Module (TPM) which stores cryptographic keys. However this approach is rather complex and protects only from some off-line based attacks which are not detailled in this study.

It should be noted that both of those mechanisms only provide detection and can not prevent an intrusion or corruption from happening.

1.3 Access control models

Several Access control models have been implemented in modern operating systems to achieve the objectives mentioned previously. I'll detail those models in this section.

1.3.1 Discretionary Access Control (DAC)

The Discretionary Access Control model give each user full control over the files and other system objects it owns. The control is left at the discretion of the user, and most of the time this is also the identity impersonated by an attacker.

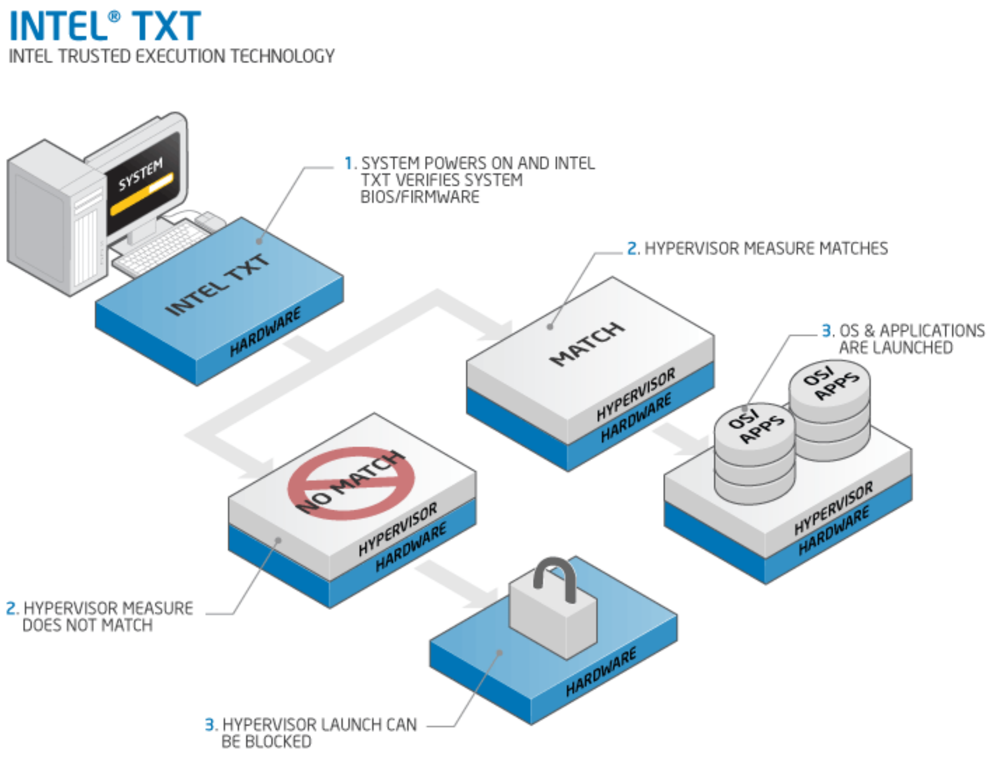

INTEL® TXT
INTEL TRUSTED EXECUTION TECHNOLOGY
SYSTEM
INTEL TXT
HARDWARE
1. SYSTEM POWERS ON AND INTEL TXT VERIFIES SYSTEM BIOS/FIRMWARE
2. HYPERVISOR MEASURE MATCHES
MATCH
HYPERVISOR
HARDWARE
3. OS & APPLICATIONS ARE LAUNCHED
OS/ APPS
OS/ APPS
HYPERVISOR
HARDWARE
NO MATCH
HYPERVISOR
HARDWARE
2. HYPERVISOR MEASURE DOES NOT MATCH
HARDWARE
3. HYPERVISOR LAUNCH CAN BE BLOCKED

Figure 1: Intel Trusted Execution Technology

Applications ran by this user are also given the same control, and most of the time they have too many privileges. For example, Firefox can access any file stored in the user home directory, including the user ssh private key.

DAC based operating systems are also often vulnerable to the confused deputy problem [7]. In this scenario, a privileged application is tricked by an underprivileged one into revealing information or acting in a particular way which wasn't intended in the first place.

This model can not ensure confidentiality or any other complex property as for example a user give grant access to other users on confidential information he has access to. DAC is to be considered as being part of the first category, in which no clear policy is enforced on the system. [8] further describes why a DAC based system will not be able to ensure security properties.

1.3.2 Mandatory Access Control (MAC)

The reasons behind the increassing necessity to use Mandatory Access Control are fully described in [1]. They are partialy summarized in this example: Browers (and related plugins) are considered as the most used attack vector for conventionnal desktop computers. With classic DAC, a user's Firefox process would have the same privilege as any other program ran by this user. So Firefox would be able to read the user's private ssh keys generally stored in their base directory and send it to a remote attacker.

The MAC philosophy is to enforce the principle of least privilege to every program on an operating system. Firefox would then not have the required permissions to acces those keys, as only the ssh daemon would be able to do so. The consequences of users making bad choices or mistakes, while managing the permissions given to his files for example, are also severly limited in a MAC environment.

MAC mechanisms are based on clear distinctions between the user (its applications) and the process responsible for grating or denying permissions. This “process” can be implemented as an other application but generally require cooperation from the kernel. Its goal is to ensure that a set of rules, the security policy, are enforced on a system.

Using MAC, we can build a policy enforing the least privilege design by starting with a base policy denying acces to everything, and progressively allowing applications to use the ressources they need. In terms of trust, MAC can be categorized in the third and forth ones which means not trusting applications to behave properly and ensuring that another mechanism, software or hardware based, will control them.

1.3.3 Capability-based security

Capability-based security is a concept which gives each subject or process a restricted set of permissions on critical ressources. A process will never be able to gain more permissions, and it can only give the permissions it already owns to its children. As with those restrictions every process on the system would only lose capabilities, such operating systems implements capability exchange to allow granting of selected capabilities. There are only a few operating systems implementing such design [9].

The Linux kernel implements a similar but very basic concept of capabilities to limit the need of binaries with the setuid flag. This flag allows binaries to be run under the uid or identity of the binary owner and is often used to allow users to run programs which require root privileges. In this case, capabilities allow the administrator to grant a process the access to a restricted set of system ressources such as binding on TCP port under 1024 while keeping the process runnning under an underprivileged user. Those capabilities can not be transfered to other processes.

1.3.4 Multilevel Security and Multi Categories Security (MLS & MCS)

The multi-level and multi-category security model enable us to achieve goals such as integrity and confidentiality. Objects and subjects are separated in several category or security level and rules are set up to limit interactions between those levels or category.

Bell-LaPaluda: This model [10] tends to preserve confidentiality of information as it only allows a subject to write information to a superior or equal level of confidentiality, and read information from an equal or inferior level. The several disadvantages in this model are the lack of integrity control and distinctions between too subjects with the same confidentiality level.

Biba: As opposed to the previously described model, it’s integrity control that’s being enforced by the Biba design [11]. Only modifications in an inferior or equal level and reads from a superior or equal level are allowed.

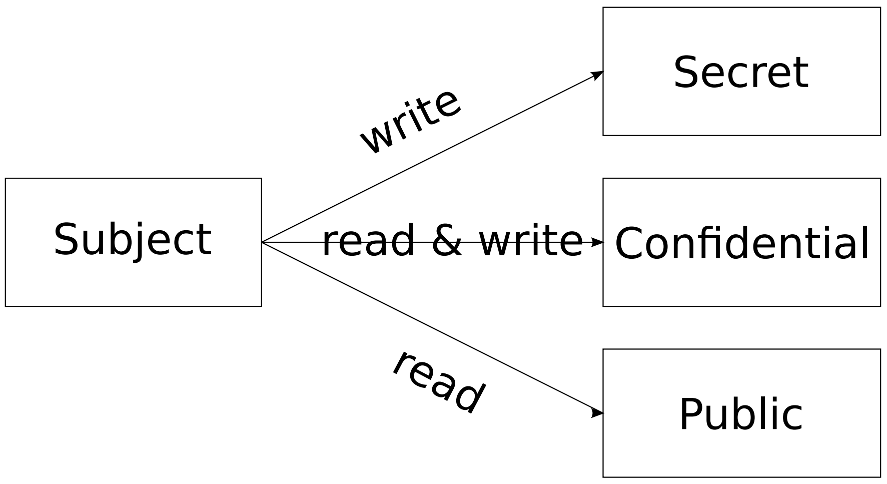


Figure 2: The Bell-LaPadula Model

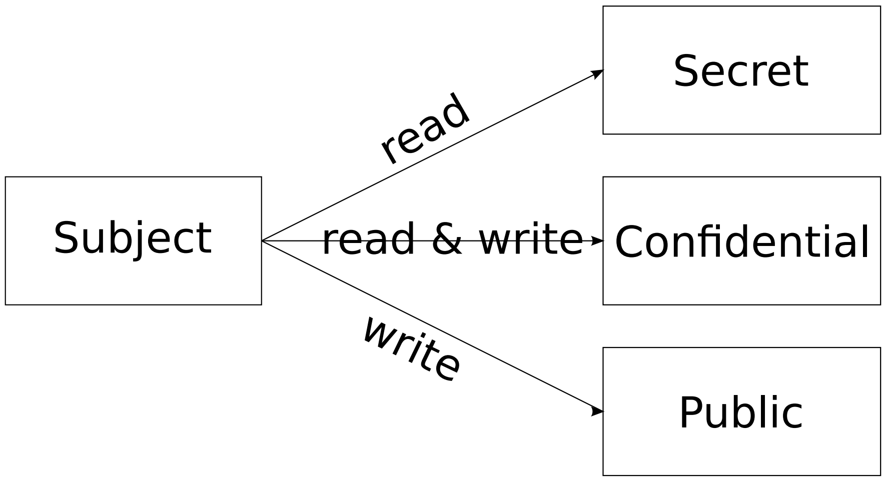


Figure 3: The Biba Integrity Model

Those relatively simple security model can not enforce more than a single security property at the same time. They also raise the problem of over classification as in the Bell-LaPadula model, information can only get more classified, and its declassification require the intervention of a trusted user. This partially compromise the model as it can not function without external intervention. In fact, those models can be applied to reduced sections of an operating system to provide local integrity and confidentiality checks.

1.3.5 Role Based Access Control (RBAC)

Role Base Access Control is an administration model to simplify the management of MAC enabled operating systems. It allows to regroup desired set of permissions into several roles and assign roles to users. For example, the user, admin, web_admin roles could be defined so an user could be given control over the web server without requiring full acces as in root or admin access to the server. This also allows the administrators to easily grant and remove a large group of various permissions to an user, thus improving management.

1.3.6 Type Enforcement

The Type Enforcement model is based on a reference monitor and full labelling of system ressources. A security context is associated to each object and subject and every interaction between any subject or by a subject on an object is controlled by the reference monitor. Most of the time, decisions are based on a policy which is enforced on the system by a trusted element, which is often sotfware based such as the operating system kernel.

The only allowed interactions are those defined in the policy and any other access is denied by the reference monitor. This model require the administrator to grant only the required permissions to selected subjects and to chose a security context for every file on every storage used by a system.

1.4 Available solutions to enforce Mandatory Access Control

This section will detail the different implementations of security models in some operating systems. I decided to concentrate my study on systems where the source code is freely available for study: Linux and the BSD family. This section is based on the detailled summary on page 86 of [12] and [13]. As of today, the first four solutions are available in the upstream Linux kernel as Linux Security Modules.

1.4.1 Simplified Mandatory Access Control Kernel (SMACK)

The SMACK model [14] is explicitly focused on easy administration and easy setup. It is based on security labels that are given to every objects and subjects on a system. Access control is enforced by storing rules inside the security labels. Several special labels allow read only or write only support for example.

Userspace support in applications is not required to function properly, but some scripts are provided to simplify setup and management.

Constrains for the administrator: No centralized policy or label management. Must carefully setup every label on the system to avoid mis-labelling. Does not advertise audit features.

Advantages: Easy to setup as there isn't any particular tool required. Presented as simple, according to the author, as compared to SELinux for example.

1.4.2 TOMOYO Linux

The TOMOYO Linux tool [15] [16] [17] is a project enabling users and administrators to easily build a security policy by a "try and learn" mechanism. It focuses on clear policy generation, path base control and process history. It also features per application firewall, application behaviour audit, confined environment for remote users for example and a simple RBAC management. However, the current version integrated into the Linux kernel is not complete and cannot provide the full functionnality.

Constrains for the administrator: No initial policy provided, require manual tweaks to be really efficient

Advantages: Learning mode available, audit tools, progressive setup.

1.4.3 Application Armor (AppArmor)

The AppArmor security module provide application centered control. It enforces capabilities and access restrictions and features a learning mode which logs access and simplify profile configuration. It was designed to be easier to setup than SELinux.

Constrains for the administrator: No global policy. Must setup each program separately.

Advantages: Learning mode available, some profile are provided by distributions for widely used softwares.

1.4.4 Security Enhanced Linux (SELinux)

SELinux is MAC mechanism implemented in the Linux kernel as a Linux Security Module [18]. Originally developed by the National Security Agency (NSA), it features type enforcement based MAC, RBAC management, MLS and MCS support. It is based on security contexts associated with each objects on the system, and a reference monitor implemented in the Linux kernel. Just like any other LSM based access control, SELinux checks every system call for compliance to a set of previously loaded policy. Every choice is made independantly, ensuring SELinux is stateless thus ignoring covert channels and indirect interactions.

Constrains for the administrator: The full set of allowed interactions as to be described in a specified language to create the policy. Proper testing must ensure that every application can access the ressources they need. File systems must be able to store security context as an extended attribute, preventing the use of old or exotic ones. Network operations (NFS shares) are challenging and require external control.

Advantages: Fine-grained confinement of subjects on a system; available in several mainstream distributions and well-tested in the Linux kernel; a reference policy is available.

1.4.5 Policy Interaction Graph Analysis - Mandatory Access Control (PIGA-MAC)

PIGA is a concept with a collection of associated tools made by Jérémy Briffaut as a proof of concept for his PhD thesis and further enhanced later with the help of the SDS team from the LIFO laboratory and ENSIB students to produce PIGA-OS. The PIGA-OS operating system won the Sec&Si challenge organized by the french National Research Agency (ANR) [19] [20]. It's goal is to enforce security properties on a system thanks to an activity description language and a compiler to obtain sequences of SELinux interaction that are infringing those properties. This improvement could be simplified as being a statefull SELinux implementation and is able to detect covert channel exploits.

Constrains for the administrator: Knowledge of the activity description language and SELinux is required. The PIGA policy generator is performance hungry, but it is a one time operation.

Advantages: Fine-grained control over on going interaction. SELinux advantages.

1.4.6 PaX & grsecurity

grsecurity [21] is another implementation of the MAC and RBAC models. It also features Access Control Lists and trusted path execution, ensuring the binaries on the system can not be modified by untrusted users. Often associated with PaX witch is a Linux kernel patch, it brings heavy restrictions to the rights given to standard users and access to memory pages.

Constrains for the administrator: Several program do not work with the PaX restrictions. The trusted execution of programs require carefull administration and policy design.

Advantages: Advanced memory protection, rule learning feature, simplier design compared to SELinux.

1.4.7 WˆX & TrustedBSD

The Trusted BSD project [22] has added a feature similiar to PaX called WˆX to enforce separation between executable and writable memory pages on FreeBSD. Audit capabilities, Access Control Lists and several MAC mechanisms were also added. As I'm not familiar with *BSD operating systems, I will not detail this section much further.

2 Development and test environment

PIGA features are intrinsically linked to SELinux features so the choice of development environment was limited. The main GNU/Linux distributions supporting SELinux are:

- Red Hat Enterprise Linux [23], CentOS [24], Scientific Linux [25], Fedora [26];
- Gentoo [27] with the Hardened Gentoo project [28].

As PIGA-OS is based on Hardened Gentoo, I chose to stick with this distribution as the primary development environment.

2.1 Hardened Gentoo

The Hardened Gentoo Project has taken upon the goal of building a completly secure operating system from the ground up. To achieve this goal, several particularities have been selected:

- a linux kernel with the PaX & grsecurity patch;
- a carefully crafted configuration and several restrictive compilation options;
- a modular SELinux policy based on the reference policy.

The first step of this project was to setup a virtual Hardened Gentoo development environment. I used the well written Gentoo SELinux Handbook available at [29].

The Gentoo choice was also driven by the fact that I had to understand most of the building process in order to make it work properly. This constrain was actualy a benefit as I needed a clear view on how SELinux works and how do I manage it.

2.2 QEMU, KVM, libvirtd and the Virtual Machine Manager

In order to ease the development process, I used some KVM and QEMU special features. The Virtual Machine Manager also provided me with all the tools required to deploy QEMU/KVM virtual machines more easily.

2.2.1 Using an host compiled kernel

QEMU enabled me to use an externaly (as in: not on a virtual drive) compiled kernel. It saved me a lot of time in kernel compilation as I was able to do it on the host system instead of the virtual one. As the host and the virtual machine architecture matchs (x86_64), there was no cross-compilation involved and I simply used this kernel into the Gentoo virtual machine.

This option is enabled with the corresponding arguments added to the qemu command line:

```
qemu−kvm ... −kernel path/to/bzImage −append root=/dev/sda1 ro quiet
```

2.2.2 Debuging a kernel with QEMU and gdbserver

One of the many useful feature in the QEMU emulator is the ability to debug a kernel running inside a virtual machine thanks to the gdbserver program. In order to use it, you have to add this argument to the qemu command line, where 1234 is the port you want it to listen to:

```
qemu-kvm ... -gdb tcp::1234
```

2.2.3 libvirtd XML configuration

To simplify network setup and management for virtual machines, I choosed to use libvirtd and the graphical user interface called virt-manager which stands for Virtual Machine Manager. The libvirtd daemon uses XML files to store virtual machines configurations. This is the example configuration I used, including the previously discussed QEMU features:

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu↩
   /1.0'>
  <name>Gentoo_projet</name>
  <memory>2097152</memory>
  <vcpu>1</vcpu>
  <os>
    <type arch='x86_64' machine='pc-0.14'>hvm</type>
    <kernel>linux-2.6.39-hardened-r8/arch/x86_64/boot/zImage</kernel>
    <cmdline>root=/dev/sda1 ro quiet</cmdline>
    <boot dev='hd'/>
  </os>
  ...
  <devices>
    <emulator>/usr/bin/qemu-kvm</emulator>
    <disk type='file' device='disk'>
      <driver name='qemu' type='raw'/>
      <source file='gentoo_projet.img'/>
      <target dev='hda' bus='ide'/>
      <address type='drive' controller='0' bus='0' unit='0'/>
    </disk>
    <interface type='network'>
      <model type='e1000'/>
      ...
    </interface>
    ...
  </devices>
  <qemu:commandline>
    <qemu:arg value='-gdb'/>
    <qemu:arg value='tcp::1234'/>
  </qemu:commandline>
</domain>
```

3 PIGA-Linux

3.1 What does PIGA do?

PIGA's first objective was to detect any malicious activity on a system. To achieve this goal, PIGA is built on top of an other MAC (only grsecurity and SELinux in its current form).

Each MAC implementation defines operations or interactions performed on a system by userspace programs (read, write...). PIGA builds a graph with the MAC defined interactions in order to obtain an exhaustive view of all the interactions and sequence of interactions that are allowed on a system.

The language defined in [12] is designed help administrators represent security properties. Administrators can use this language to define which properties they want to ensure on a selected system (program binaries integrity for example). Then, the PIGA-pol tool will generate the full list of sequence of interactions infringing the security properties from the interaction graph previously generated.

This will ensure that no cover-channel or alternative way can be used to exploit a weakness in the MAC policy. [12] contains a full description of PIGA features.

The Figure 4 details the process to generate the signature database from both an access control policy and a detection policy. Those signatures can then be used to enforce access control on a system and report access errors or warnings.

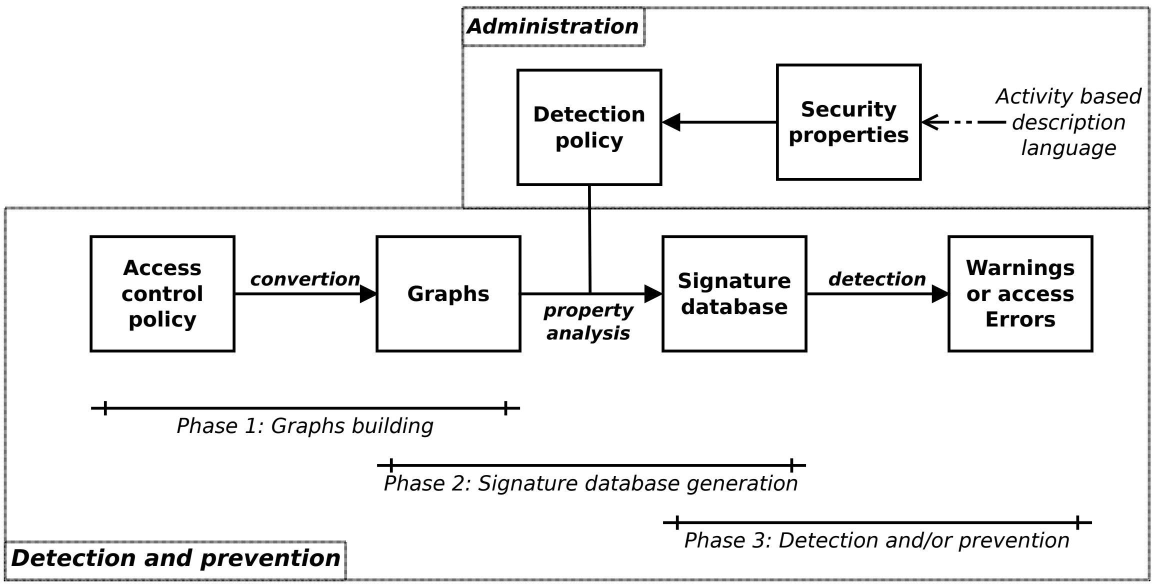


Figure 4: The process to enable PIGA control

3.2 Why porting PIGA to Linux?

The PIGA concept gave birth to several components such as the PIGA-IDS/IPS, which is an userspace daemon written in Java. It was developped as a proof of concept and turned into a fully working

IPS for the PIGA-OS project [19]. Its userspace design led to some limits and raised unexpected problems for network related interactions. It also introduced several additionnal context switches for each system call and had a limited but mesurable impact on the system performance.

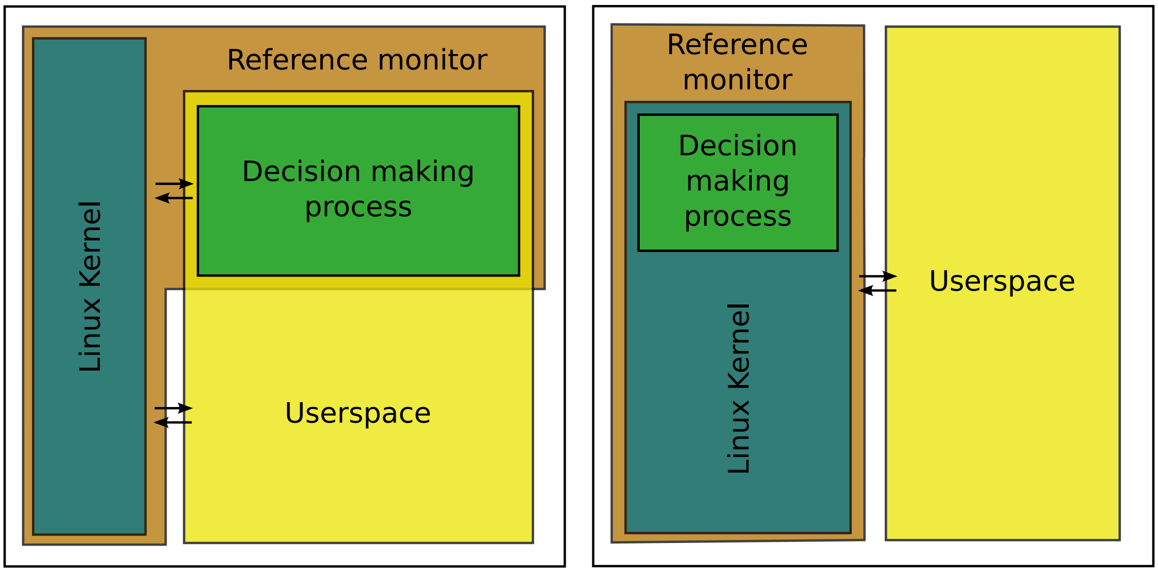


Figure 5: From PIGA-IDS/IPS to PIGA-Linux

From a security point of view, the process making the decission to allow or deny access what located in userspace and thus was more vulnerable to conventionnal attacks.

Several of those problems could be solved more easily by moving the decision making process into the kernel.

3.3 Objectives

The following objectives (and implementation order) were layed out with Jérémy Briffaut at the begining of the project:

1. Implement a basic decision making algorithm in the kernel
2. Create userspace tools to load a PIGA policy into the kernel
3. Optimize the policy storage into the kernel to handle large policy (more than three hundred thousand signatures) in order to make this useful in the real world
4. Implement remaining PIGA-IDS and PIGA-IPS fonctionnality (conditionnal policy, process based signatures, better audit features, rule learning, state save and restore...)

The previously discussed objectives and the development environment I have chosen add several constrains:

1. As PIGA will likely be developped and used inside a virtual machine, we need to make sure the modifications we bring into the kernel are not affecting virtualization.

2. The implementation has to be fast enough to work in a limited environment such as a virtual machine.
3. To make PIGA as architecture independant as possible, we should not use processor specific features for example.

3.4 Architecture

PIGA-Linux is divided between a kernel patch, a userspace tool, and “standard” PIGA tools. This project focused only on the kernel patch and the policy loader tool. I chose to keep the PIGA signatures text format as I don’t need to change those steps in the PIGA process. The “standard” tools functionnality will be briefly explained here for convenience.

3.4.1 PIGA tools

As detailled previously, several tools are required to enable PIGA based control on an operating system. First, we need a simple tool to generate the interaction graph from a MAC policy by listing every possible iteraction. This graph can then be processed by the PIGA-pol tool which looks for sequences that could that would match security properties defined by an administrator in the PIGA activity based description language. The PIGA-pol tool can generate a signature database for grsecurity or SELinux based MAC, but I will only use the SELinux version. This choice was based on the fact that PIGA-OS is SELinux based and that SELinux provides a fine-grained access control.

A PIGA policy obtained from an SELinux policy should follow those simple guidelines :

```
# Lines starting with an '#' are comments. Blank lines and spaces are
# ignored
#
# The first two fields should be ignored for now as they correspond to
# not yet implemented PIGA features
#
# Each sequence should be kept on one line for now. Future
# improvements should allow spreading of a sequence on several lines.
#
# Description:
# <ignored> - <ignored> : <source_selinux_security_context>
# -( <interaction_class> { <interaction_type> } )->
# <target_selinux_security_context> ; ...

WARN - 0$4028 : root:sysadm_r:sysadm_t -( file { read } )-> system_u:
object_r:locale_t ; root:sysadm_r:sysadm_t -( file { write } )-> root:
object_r:user_tmp_t ; root:sysadm_r:sysadm_t -( file { read } )->
root:object_r:user_tmp_t
```

3.4.2 Linux kernel implementation

As of today, the kernel implementation is severly limited in terms of performance, but fullfils the basic purpose of PIGA: blocking sequences of interactions.

A standard PIGA policy may involve 1 000 to 300 000 rules containing 1 to 13 SELinux interactions. In order to have a low memory consumption and a fast load time from userspace to the kernel, the PIGA policy is first stored in a basic array of 'struct sequence' in the kernel and the links composing the sequences are stored alongside in an other array of struct link.

```
struct sequence {
    unsigned int length;
    unsigned int current_position;
    unsigned int link_offset;
};

struct link {
    unsigned int cs;
    unsigned int cc;
    unsigned short tclass;
    unsigned int requested;
};

// The number of sequences and links
static unsigned int s_len = 0;
static unsigned int l_len = 0;

// Sequence and link 'array storage'
static struct sequence * seq = NULL;
static struct link * link = NULL;
```

The sequences are placed in order in the 'seq' array, and their corresponding links are added to the 'link array', at the offset 'link_offset'. For example, to obtain the third sequence:

```
// Retrieve the sequence:
struct sequence * seq2 = seq[2];

// Retrieve the associated `seq[2].length` links:
for(i = 0; i < seq2->length; ++i) {
    struct link * l = link.[seq2->link_offset + i];
    // Use the link l here
}
```

This simple implementation allows us to load the PIGA policy easily from userspace as it only requires the copy of two memory regions, which is a fast process.

In order to reduce seeking time during normal kernel operation, I attempted to add a second dynamic storage based on the kernel implementation of linked list. It stores interactions in an array indexed by the tclass field:

```
/* Defines the maximum tclass value used in SELinux + 1 for ↩
   convenience */
#define PIGA_SELINUX_MAX_TCLASS 50

static struct list_head * tclass_tab[PIGA_SELINUX_MAX_TCLASS];
```

The statistics obtained from the policy containing 300 000 sequences are backing this simple design as signatures appears to be well distributed among the different SELinux "tclass" type. The impact on the algorithm complexity will be discussed further down.

The kernel modifications remain as less intrusive as possible as only two major files need minor modifications. The decision taking algorithm for SELinux regroups every checks in a single function, allowing us to easily add the PIGA control after the SELinux one. Every system call controled by a LSM hook has to pass this test. This is an extract from security/selinux/avc.c:

```
...
#ifdef CONFIG_SECURITY_PIGA
#include "piga/include/piga.h"
#endif
...
int avc_has_perm_flags(u32 ssid, u32 tsid, u16 tclass,
               u32 requested, struct common_audit_data *auditdata,
               unsigned flags)
{
    struct av_decision avd;
    int rc, rc2;

    rc = avc_has_perm_noaudit(ssid, tsid, tclass, requested, 0, &avd);

#ifdef CONFIG_SECURITY_PIGA
    if (likely(rc != -EACCES)) {
        rc = piga_has_perm(ssid, tsid, tclass, requested, auditdata,
                           rc, &avd);
    }
#endif
    rc2 = avc_audit(ssid, tsid, tclass, requested, &avd, rc,
                    auditdata, flags);
    if (rc2)
        return rc2;
    return rc;
}
...
```

This ensures that already denied access are not checked by PIGA-Linux. As all SELinux behaviours remain unchanged, PIGA denials currently appear as SELinux denials. For now, we can keep the SELinux auditing interface to audit PIGA denials. In future development, we should include a flag or a special field specifying that this particular denial is related to PIGA.

The core algorithm taking decisions in PIGA-Linux is located in security/selinux/piga/piga.c:

```
...
int piga_has_perm(u32 ssid, u32 tsid, u16 tclass, u32 requested,
                  struct common_audit_data *auditdata, int rc, struct
                  av_decision * avd)
{
    struct sig_list * seqs = NULL;
    struct sequence * s = NULL;
    u32 denied = 0, audited = 0;
    struct sig_list *sg;
...
```

First, we check if PIGA is enabled and if it should check every access or only SELinux audited access. Each vector in a PIGA signature/seqeunce should be added to a special SELinux module loaded before enabling PIGA in order to watch only the interactions that are part of a signature and skip everything else. This is the first step to improve performance.

```
    if (piga_status_enabled) {
        ...
            if (piga_audit_only_mode) {...}
        }
```

The access control part was designed to be as generic as possible. The piga_get_sequence_at() function has just enough information about the current interaction to retrieve every sequence that could potentially match. Any performance improvement and design change will be implemened in this function and will not impact the remaining code. The first two versions of the piga_get_sequence_at() function have a linear complexity and depend only on the number of loaded signatures.

```
        rc = PIGA_ALLOW;
        read_lock(&tclass_lock);
        seqs = piga_get_sequence_at(ssid, tsid, tclass);
        if (unlikely(list_empty(&(seqs->list)))) {
            read_unlock(&tclass_lock);
            return rc;
        }

        // Iterate over the list to find if the signature is maching
        list_for_each_entry(sg, &(seqs->list), list) {
            s = sg->seq;
            if (unlikely(
                piga_seq_get_cs(s) == ssid
                && piga_seq_get_cc(s) == tsid
                && piga_seq_get_tclass(s) == tclass
                && (piga_seq_get_requested(s) & requested) > 0)) {

                print_vector(ssid, tsid, tclass, requested, auditdata,↩
                    rc, avd);

                if (unlikely(piga_seq_end(s) == true)) {
                    printk(KERN_INFO "PIGA: End of sequence. Access ↩
                       denied\n");
                    rc = PIGA_DENY;
                } else {
                    read_unlock(&tclass_lock);
                    write_lock(&tclass_lock);
                    piga_seq_next(sg);
                    write_unlock(&tclass_lock);
                    read_lock(&tclass_lock);
                }
            }
        }
        read_unlock(&tclass_lock);
    }

    return rc;
}
...
```

To prevent concurrent access to the kernel stored signatures, I used a simple read/write locking mechanism, which will only enable one process to modify the state of a signature at a precise time. This is also the first step towards a multi-cpu and multi-core architecture. This design will allow the processing of a group of sequences affected by an interaction as a whole rather than one after an other. Fine-grained locking could also be implemented in future work in order to allow parallel interaction checks.

3.4.3 Piga Policy Parser

The PIGA Policy Parser (PPP) is a userspace tool built with Flex and Bison (the GNU Lex & Yacc equivalent). It parses and checks the correctness of a set of signatures contained in a policy file. Those signatures were previously generated with PIGA related tools from a set of PIGA rules and a SELinux policy. This is an extract from the Yacc parser (ppp.y) detailling the grammar used to parse PIGA policies:

```
%token <str> TYPE
%token <str> BOOL
%token <str> SID
%token EOL ENDF AVPERMSTART AVPERMEND
```

```
%token <str> WORD

%%

wholefile:  file ENDF {...} | file EOL ENDF {...};
file:       line {...} | file EOL line {...};

line:       TYPE '-' BOOL ':' signature {...} |
            BOOL ':' signature {...};

signature:  transition {...} | signature ';' transition {...};
transition: SID AVPERMSTART vector AVPERMEND SID {...};

vector:     WORD '{' request '}' {...};
request:    WORD {...} | request WORD {...};

%%
```

The PPP currently offers only the first basic features and options:

- -h: prints the help
- -v: turn verbose mode on, printing warnings for secrity contexts found in the PIGA policy but not in the loaded SELinux policy
- -s: display some statistics from the PIGA policy, such as the count of each security context

Some possible, not implemented, improvements and ideas are listed here:

- -e: enable PIGA once if the policy loaded successfully
- -n: do not enable PIGA after loading the policy (default)
- -l: load the given SELinux module before loading the PIGA policy
- -f: use full check mode for PIGA
- -a: use SELinux audit mode for PIGA (requires a custom SELinux module which can be loaded with -l)
- -r file: restore previous PIGA state from file
- -o file: store current PIGA state in file.
- -c: enable conditionnal signatures (PIGA feature not yet available)
- -w: enable warning/audit mode (PIGA feature not yet available)

In order to load a policy into the kernel, you need to give a policy file to the tool standard input as PPP is based on Flex & Yacc. Classic input from a file as an argument will be implemented soon. Usage: ppp [-hvsenfacw] [-l selinux_module] [-r state_file] [-o state_file] < PIGA_policy.pol):

```
$ ppp -s < piga_policy.pol
PPP: Beggining PIGA policy parsing and loading...
PPP: Policy successfully parsed. Setting up in the kernel...
PPP: PIGA policy loadded:
    4 signatures have been loaded
    0 signatures have been ignored
Policy stats: '<name> (<sid>): <count>'
    Security context stats:
            root:object_r:user_tmp_t (114): 2
            root:sysadm_r:sysadm_t (167): 3
            sysadm_u:sysadm_r:dhcpc_t (219): 2
            sysadm_u:sysadm_r:groupadd_t (215): 2
            sysadm_u:sysadm_r:nscd_t (216): 3
            sysadm_u:sysadm_r:sysadm_t (214): 4
            sysadm_u:sysadm_r:useradd_t (218): 2
            system_u:object_r:locale_t (86): 1
            system_u:system_r:init_t (47): 3
            system_u:system_r:initrc_t (56): 2
            system_u:system_r:sulogin_t (217): 2
    tclass stats:
            file (6): 3
            process (2): 10
    requested stats:
            read: 2
            transition: 10
            write: 1
```

Parsing the whole policy containing 300 000 signatures takes approximately 13 seconds inside the virtual machine. The loading time inside the kernel depends on the algorithm used for the dynamic policy storage.

In this example, we used the statistics flag to dispaly some information about the policy loaded. First, PPP tells us that 4 out of 4 signatures have been loaded as none has been ignored. The SID (Security ID) field correspond to the current number associated with the security context in the kernel. Those numbers change at each reboot, so this display has no further purpose. The SIDs are retreived throught the kernel interface described in the next section.

3.4.4 Interface between the kernel and userspace

For now, the PPP tool is using two custom system call. One which enables it to quickly load the policy:

```
/** Userspace code **/

#ifdef __x86_64
#define __NR_sys_piga_add_sequence 307
```

```
#endif /* __x86_64 */

/**
 *  s_len: the number of 'struct sequence' in the 's' vector
 *  l_len: the number of 'struct link' in the 'l' vector
 *  s: the 'struct sequence' vector
 *  l: the 'struct link' vector
**/
syscall(__NR_sys_piga_add_sequence, s_len, l_len, s, l);
```

And an other one needed to retieve the corresponding SID for each SELinux security context.This was designed in order to do as much parsing as possible in userspace:

```
/** Userspace code **/

#ifdef __x86_64
#define __NR_sys_piga_get_sid 308
#endif /* __x86_64 */

/**
 * scontext: the security context
 * len: security context string length
 * sid: the corresponding sid returned by the kernel
**/
syscall(__NR_sys_piga_add_sequence, scontext, len, sid);
```

As I'm going to improve the kernel storage of policy, I'll probably change this interface to make it more compliant with current kernel design policy (we should not create any new system call, but use ioctl, or other interfaces like sockets...).

PIGA control from userspace is also a work in progress, but is correctly implemented. In order to be consistent with the recent move of the SELinux filesystem to /sys/fs/selinux, I created a directory alongside for piga. There are three pseudo files available in /sys/fs/piga:

```
$ ls /sys/fs/piga/
mode  stats  status
```

- mode: This controls the mode in which PIGA is running. If disabled, PIGA runs in full check mode, checking every SELinux interaction. If enabled, PIGA checks only selinux audited interactions, thus improving performance.

```
    $ cat /sys/fs/piga/mode
    Disabled: Full check
    $ echo 1 > /sys/fs/piga/mode
    $ cat /proc/piga/mode
    Enabled: SELinux audited interactions only
```

- stats: This contains some statistics about the number of sequences and links available, and the number of those that have been loaded into the tclass lists.

```
$ cat /sys/fs/piga/stats
PIGA statistics:
    4 signatures available
    12 links available

    4 signatures stored in lists
```

- status: The current status of PIGA-Linux, either enabled or disabled.

```
$ cat /sys/fs/piga/status
Disabled
$ echo 1 > /sys/fs/piga/status
$ cat /proc/piga/status
Enabled
```

4 Results and problems encountered

4.1 Performance

Even in this crude form, PIGA-Linux isn't performing so bad. It takes notably longer to complete syscalls (4 seconds for a simple "ls -alh" in a folder with ten files) but the system is still working. The audit only mode might also improve performance a lot.

Further work was attempted to increase runtime performance and it should be noted that it involves preprocessing of sequences and thus adds delay to the policy loading step.

The focus on performance improvements was not successfully as of today due to the difficulty of debuging kernel space algorithm. Towards the end, I found how to debug a kernel running inside QEMU, which proved to be useful.

Performance improvements are a key task in the success of this project, and could be achieved by implementing and testing algorithms in userspace before porting it to the kernel.

Once it will have reach acceptable speed, we should compare it to vanilla, SELinux, PaX and grsecurity kernels, probably using the the Phoronix Test Suite. I also created a script enabling simple regression testing and helping to find memory leaks.

4.2 Design and implementation security

4.2.1 Implementing PIGA interactions

The Linux Security Model is based on hooks implemented as function pointers [18]. These hooks are sometimes considered as being arbitrary placed in the kernel code [30] but they also ensure that the structure, which will be used in the security functions, are correctly locked down and concurrency safe. They should provide just enough information in order to make an access control decision. Finally, their design is fondamentaly ponctual.

However, the PIGA definition of interactions is based on system call start and end time (see page 64 and 65 in [12]) and thus isn't ponctual. For example, a blocking read syscall, waiting for content from a pipe, would escape PIGA sequence checking, as the control might happened before any data is written into the pipe.

We need to modify the PIGA design to include new LSM hooks, which should be placed right after the portion of code that deals with blocking state in the kernel if this is possible (this will require a deep knowledge of the kernel subsystems and might just be impossible as it could require hardware specific modifications). "exit" hooks might be needed if we would like to ensure full detections (but not full prevention) of malicious interactions. Those hooks would ensure time based detection which is as close as possible to the PIGA definition, but they won't be able to undo what a syscall just did and thus will be limited to detection. A carefully crafted succession of system calls could be able to evade the test and would not be blocked, but only detected. Any further attempt will be denied.

4.2.2 /sys/fs/piga pseudo filesystem

Access to the PIGA pseudo filesystem is restrited to sysadm_u/root with the classic SELinux policy as it is labelled system_u:object_r:sysfs_t. We could make a new SELinux module in order to further restrict the control from userspace. We could for example completly disable access once the policy is loaded.

4.2.3 Syscalls used by PPP

The policy loader syscalls are not carefully controlled. They should be “transformed” into ioctl calls as it seems to be the accepted way of adding such control into the kernel.

4.3 Missing and expected features

Those features are still missing and should be part of the project:

- conditionnal PIGA policy, preventing access to certain ressources once a sequence based boolean expression is true;
- “process-linked” signatures, linking sequence progress to process to further precise with operations are to be denied;
- better audit features to make the distinction between SELinux and PIGA denials clear;
- rule learning, advanced grsecurity-like feature, involving step by step, deny by default learning, to ease the learning curve for administrators;
- current signature state save and restore, for continuity accross system reboot/halt.

Conclusion

Thanks to this project and the helpful supervision of Jérémy Briffaut, I was able to learn a lot about the SELinux implementation in the Linux kernel, the current PIGA implementation and all of its features.

The main challenge I faced was kernel debuging and I did not solve it soon engouh. An other option would have been to implement the full algorithm in C, in userspace first, using only limited functionnality from the glibc and to optimise it, before porting it to the kernel.

I'm thankful to Jérémy Briffaut for letting me work on this topic under his supervision. I will probably keep working on this project.

References

[1] P.A. Loscocco, S.D. Smalley, P.A. Muckelbauer, R.C. Taylor, S.J. Turner, and J.F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proceedings of the 21st National Information Systems Security Conference*, volume 10, pages 303–314, 1998.

[2] D.C. Latham. Department of Defense Trusted Computer System Evaluation Criteria. *Department of Defense*, 1986.

[3] QNX Realtime Operating System. `http://www.qnx.com`.

[4] Common Criteria for Information Technology Security Evaluation. `http://www.commoncriteriaportal.org`.

[5] Intel Trusted Execution Technology (TXT). `http://www.intel.com/technology/malwarereduction/index.htm`.

[6] Linux Integrity Subsystem and Integrity Measurement Architecture. `http://linux-ima.sourceforge.net`.

[7] The confused deputy problem. `http://www.cis.upenn.edu/~KeyKOS/ConfusedDeputy.html`.

[8] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, 1976. Available at: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.106.7226&rep=rep1&type=pdf`.

[9] Capsicum: practical capabilities for UNIX. `http://www.cl.cam.ac.uk/research/security/capsicum/`.

[10] D.E. Bell. Secure computer systems: Mathematical foundations. Technical report, DTIC Document, 1973.

[11] K.J. Biba. Integrity considerations for secure computer systems. Technical report, DTIC Document, 1977.

[12] Jérémy Briffaut. *Formalisation et garantie de propriétés de sécurité système: application à la détection d'intrusions*. PhD thesis, ENSI de Bourges, 2007. Available at: `http://tel.archives-ouvertes.fr/tel-00261613`.

[13] How is TOMOYO linux different from SELinux and AppArmor? Secure OS comparison at a glance. `http://tomoyo.sourceforge.jp/wiki-e/?WhatIs#comparison`.

[14] Casey Schaufler. The Simplified Mandatory Access Control Kernel, 2008. Available at: `http://linuxplumbersconf.org/2009/slides/Casey-SmackPlumbers2010.pdf`.

[15] T.H.T.H.K. TANAKA. Access policy generation system based on process execution history. 2003. Available at: `http://en.sourceforge.jp/projects/tomoyo/docs/nsf2003-en.pdf/en/2/nsf2003-en.pdf.pdf`.

[16] T. Harada, T. HORIE, and K. TANAKA. Task Oriented Management Obviates Your Onus on Linux. In *Linux Conference*, 2004. Available at: http://sourceforge.jp/projects/tomoyo/document/lc2004-en.pdf.

[17] T. HARADA, T. HORIE, and K. TANAKA. Towards a manageable Linux security. In *Linux Conference*, volume 2005, 2005. Available at: http://ko.sourceforge.jp/projects/tomoyo/docs/lc2005-en.pdf/en/2/lc2005-en.pdf.pdf.

[18] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, volume 2. San Francisco, CA, 2002. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.84.6867&rep=rep1&type=pdf.

[19] Jérémy Briffaut, Martin Peres, Jonathan Rouzaud-Cornabas, Jigar Solanki, Christian Toinard, and Benjamin Venelle. PIGA-OS: retour sur le système d'exploitation vainqueur du défi sécurité. In *RenPar'20 / SympA'14/ CFSE 8*, 2011. Available at: http://renpar.irisa.fr/cfse8/cfse8_16.pdf.

[20] Jérémy Briffaut, Jean-François. Lalande, and Christian Toinard. Formalization of security properties: enforcement for mac operating systems and verification of dynamic mac policies. *International journal on advances in security*, 2(4):325–343, 2010.

[21] grsecurity website. http://grsecurity.net.

[22] The TrustedBSD project. http://www.trustedbsd.org.

[23] Red Hat Enterprise Linux. http://www.redhat.com/rhel.

[24] CentOS. http://www.centos.org.

[25] Scientific Linux. http://www.scientificlinux.org.

[26] Fedora. http://fedoraproject.org.

[27] Gentoo. http://www.gentoo.org.

[28] Gentoo Hardened Project. http://www.gentoo.org/proj/en/hardened.

[29] Gentoo SELinux Handbook. Available at: http://www.gentoo.org/proj/en/hardened/selinux/selinux-handbook.xml.

[30] Why doesn't grsecurity use LSM? http://grsecurity.net/lsm.php.

[31] Robert Love. *Linux kernel development, third edition*. Addison-Wesley Professional, 2010.

[32] Frank Mayer, Karl MacMillan, and David Caplan. *SELinux by Example: Using Security Enhanced Linux (Prentice Hall Open Source Software Development Series)*. Prentice Hall PTR, 2006.

[33] Richard Haines. *The SELinux Notebook - The Foundations - 2nd edition*. Richard Haines, 2009. Available at: http://www.freetechbooks.com/the-selinux-notebook-the-foundations-t785.html.

[34] Daniel J. Walsh and Karl MacMillan. Managing Red Hat Enterprise Linux 5. In *SELinux Symposium*, 2007. Available at: `http://people.redhat.com/dwalsh/SELinux/Presentations/ManageRHEL5.pdf`.

[35] Bruce Schneier. *Secrets and lies: digital security in a networked world*. Wiley, 2011.

Appendix

Notice about the Tux logo displayed on the first and last page

Notice about the SELinux logo on the first page