Extending SELinux to track memory-pages accesses

Martin Peres, supervised by Jérémy Briffaut
ENSI de Bourges

February 11, 2011

Contents

List of Figures

Chapter 1

Introduction

There are several Mandatory Access Control systems for Linux, but none of them actually brings MAC into the memory pages.

Would it be possible to extend SELinux to add support for memory pages access control on the x86 architecture? What would be the performance hit?

Chapter 2

State of the art

2.1 Security

Security is always led by the principle of least privilege[1]. It means that a process should only be able to perform its task and nothing more.

Following this principle results in a greater system reliability and security as sub-systems are not able to interact with each others unless they are supposed to. The thus simplified system can then be audited and proved more easily.

We can distinguish three different security threats[2]:

- Confidentiality: An unauthorized personnel read confidential data
- Integrity: An unauthorized personnel modified data
- Availability: An unauthorized personnel managed to prevent the system from working

2.2 Access control models

Access control models allow to control users' file accesses according to the user's privilege. All access control models fall into two categories. The discretionary (DAC) and the mandatory (MAC) access control models.

DAC lets users manage their files and define what operations other users can perform on them. This is a security threat as applications launched by the user have the right to change the availability of a potentially-confidential file to others.

The MAC policy is set by system administrators. This access control is usually set as a second layer of protection after DAC. This extra layer allows system administrators to enforce some security properties. It enhances security as users can't do everything they want with their files anymore.

2.2.1 Multilevel security

Multilevel security (MLS[3]) is a data classification model proposed by Brotzman in 1985 for the USA's Department Of Defence(DoD) as "Guidance for Applying the Department of Defense Trusted Computer System Evaluation Criteria in Specific Environments".

This paper introduced different levels of accreditation such as:

- Top Secret
- Secret
- Confidential
- Unclassified

Labeling data with an accreditation level eases data access by authorized personnel while making it harder for the unauthorized ones to break its confidentiality or integrity.

MLS is a requirement for the Bell-LaPadula and Biba models.

Bell-LaPadula

The Bell-LaPadula[4] model can be considered as a MAC model. This model defines entities(subjects) requesting to access objects (files, data). In an MLS environment, this model can be summed-up as no read-up, no write-down. See figure 2.1.

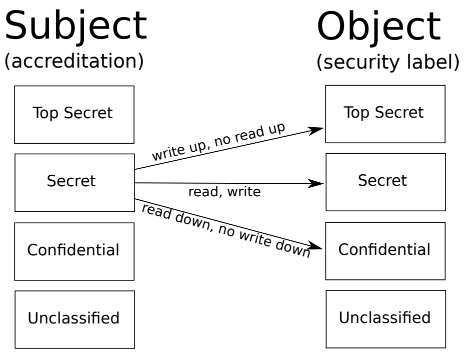


Figure 2.1: The Bell-LaPadula security model

This model preserves confidentiality but fails at preserving integrity. Also, when applied, this model tends to over-classify information which finally leads to having all the files in the top secret level. This is because there is no way for applications to de-classify data automatically.

Biba

The Biba[5] model is a clone of the Bell-LaPadula model that focuses on integrity instead of confidentiality. In an MLS environment, this model can be summed-up as no write-up, no read-down. This is the exact opposite of the Bell-LaPadula model. See figure 2.2.

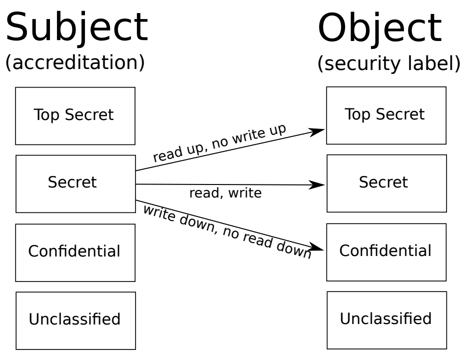


Figure 2.2: The Biba security model

The Biba model also suffers the same problems as the Bell-LaPadula model, but the other way around.

2.2.2 Domain and Type Enforcement

The major problem of the MLS approach of security is that not all systems are that hierarchical. Actually, most of them are not.

With the MLS approach, it means that people working on a top secret project get clearance to every top secret project when they only needed to get clearance relative to their work. This is a direct violation of the principle of least privilege and this is what the Domain and Type Enforcement[6] is all about.

Using DTE, independent components of the system are isolated from each others in what are called domains or sandboxes. See figure 2.3.

Subjects, Objects and Actions

The DTE model formalizes system interactions and gives names to different entities:

- Subject: The initiator of the interaction (may be a person or a process)
- Action: The kind of action done by the subject on the object
- Object: The object on which the action is performed by the subject.

Then the mandatory access control checks whether the interactions on the system are legal or not and applies the decision as whether the interaction should be pursued or not.

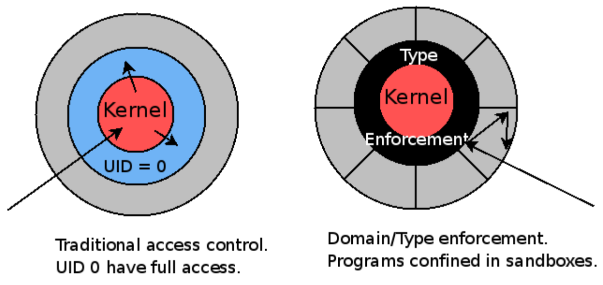


Figure 2.3: The DTE and MLS models compared

2.2.3 Role-Based Access Control

The Bell-LaPadula or Biba models being too difficult to implement for industries, most of them sticked to DAC. This lead to the Role-Based Access Control (RBAC)[7] model.

RBAC is a MAC model which describes allowed interactions in terms of roles instead of users. This eases the administrators' work as they only need to write the policies once and then assign users a role. As industries already have role-based tasks, this makes sense to keep these roles for the data access control.

2.3 Hardware overview

Now that the major security models have been described, it is time to look at how the hardware works.

As you can see on the figure 2.4, the northbridge interconnects everything. Its task is to provide access to the memory to the system. There are two ways to access the central memory, either the CPU accesses it or a device knowing how to do Direct Memory Access(DMA). Given that memory accesses are performance critical, the northbridge needs to answer requests in a timely fashion.

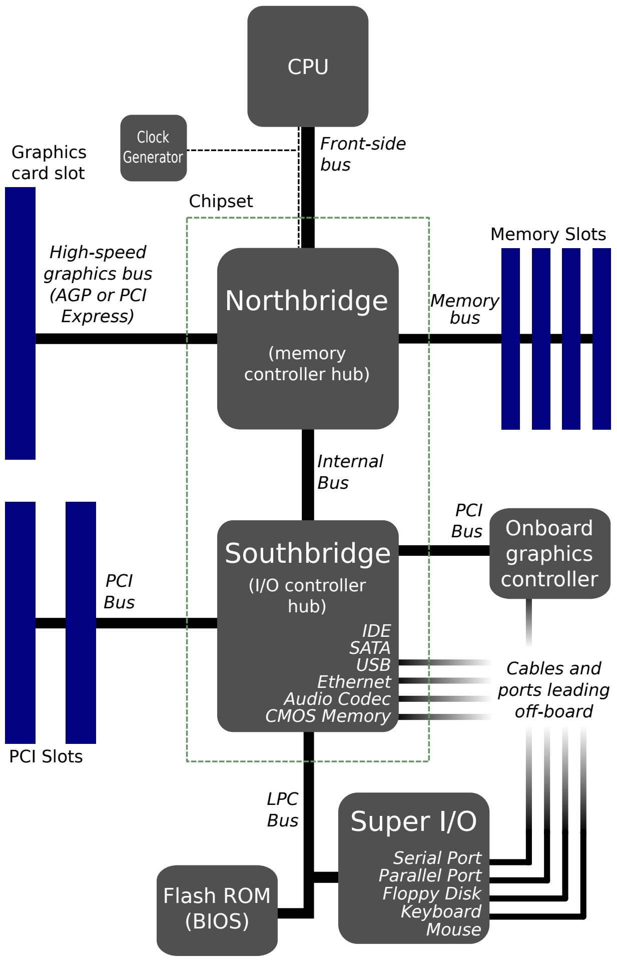


Figure 2.4: General overview of the main components of a motherboard

The southbridge is dedicated to connecting low-bandwidth peripherals to the faster quatuor that is the CPU, the northbridge, the memory and the graphic card.

Graphic cards are a central element on today's desktop computers. As they communicate a lot with the CPU (through Memory-Mapped IO(MMIO) and DMA), they have a dedicated high-speed PCI Express or AGP bus directly connected to the northbridge. This is the only peripheral to have access direct access to the northbridge.

The rest is pretty much self-explanatory or doesn't pose any security threat that I try to address in this paper.

2.3.1 Virtual Memory

Overview

Virtual memory has always been a feature provided by any x86 processor but support failed to appear to the general public until Windows 3.0.

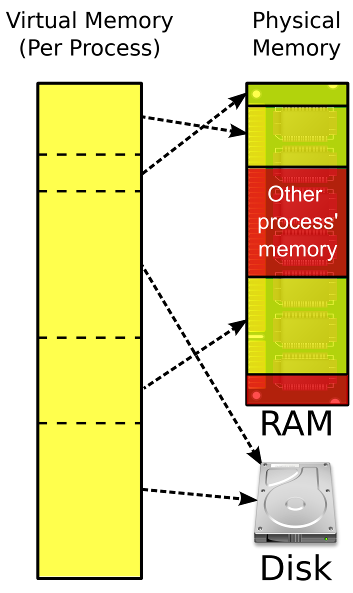


Figure 2.5: General overview of what is virtual memory

As shown on the figure 2.5, the goal of virtual memory is to add an indirection layer between the addresses processed by a processus and the physical RAM. This technique makes it possible for processes to allocate big chunck of contiguous memory in their virtual memory space even though there is no such contiguous space in the physical memory space. Also, it made possible for the OS to emulate a bigger physical memory by dumping some rarely-used memory areas to the hard disk drive. On Unix, this is the role of the swap partition.

The introduction of this indirection layer is also a nice security feature as processes are now jailed into their virtual memory space and cannot access others' memory space.

Paging

To avoid fragmentation and increase the speed of virtual to physical memory address translation, memory pages have been introduced. A page is the smallest memory space an Operating System can allocate. On x86 processors, the typical memory page is 4KB.

As can be seen on figure 2.6, a 32-bits physical memory address is split in 3 parts:

- Offset: Composed of 12 bits, it can address 2^{12} bits = 4KB, the size of a memory page. This means the lower 12 bits are used to address every byte of a memory page.
- Page: Composed of 10 bits. This can address 1000(2^{10}) memory pages.
- Directory: Composed of 10 bits. This can address 1000 page directories.

In the end, a 32-bits processors can address one million(2^{10+10}) 4KB memory pages.

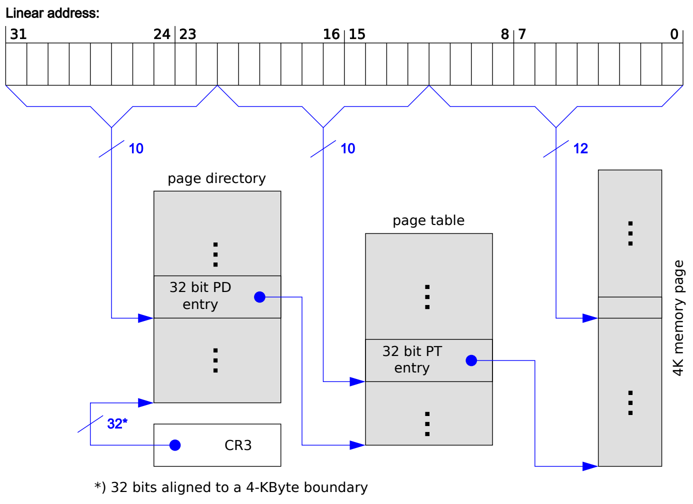


Figure 2.6: Paging on x86 processors

Address translation

When the processor wants to access a memory address, it goes through the Memory-Management Unit(MMU) that will handle the request for him.

As the processor usually deals with virtual addresses, it is the MMU's task to translate those virtual addresses into physical ones. To do so, the MMU first looks into the Transition Look-aside Buffer(TLB) for an already known correspondence. If there are no matches, the MMU calls for help to the OS by issuing an Interruption Request(IRQ), this IRQ is called page fault.

It is possible to flush the content of the entire TLB, just a specific range of linear address or a single page. This list is not comprehensive, there are several other possibilities.

There are in fact at least two independent TLBs in the MMU. The Instruction-fetch Transition Look-aside Buffer(ITLB) is used for fetching instructions while the Data Transition look-aside Buffer(DTLB) is used for data (read/write) transferts.

See figure 2.7 for a visual representation of the virtual-to-physical translation of the memory address.

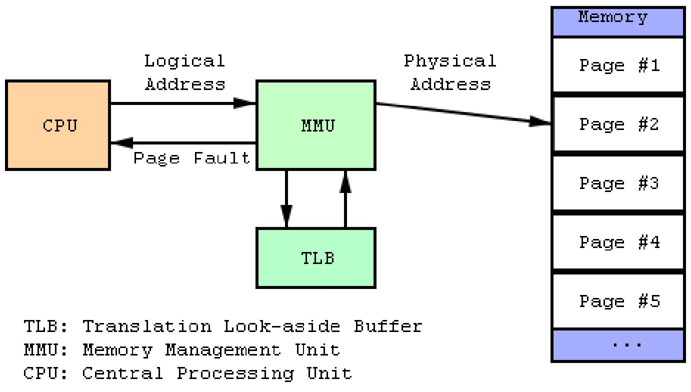


Figure 2.7: General overview of the address translation process

Page fault

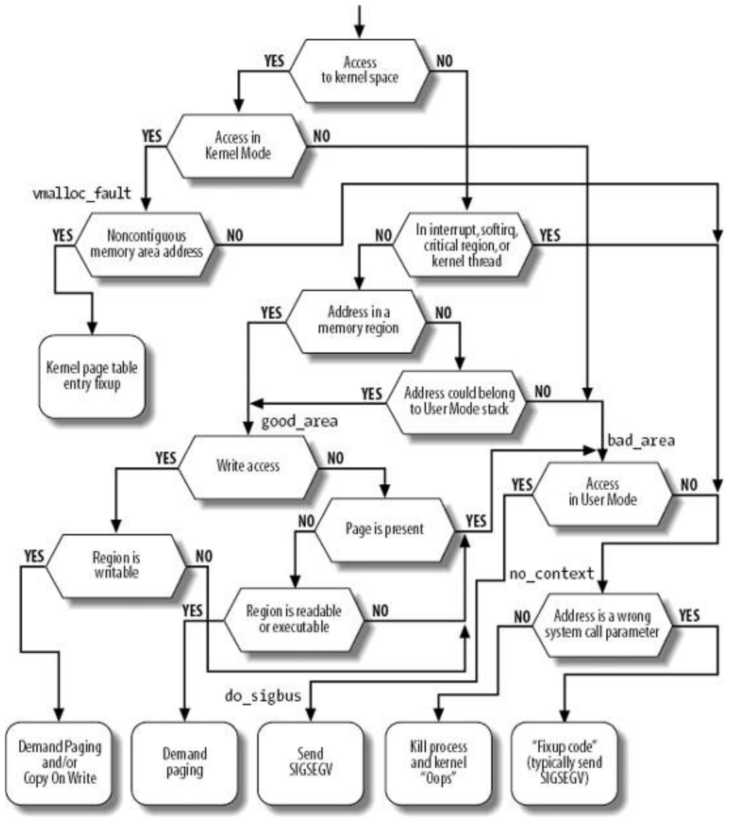
There are multiple reasons for an OS to receive a page fault. This goes from the simplest case of the MMU asking for help on virtual-to-physical memory address to the implementation of Linux's famous Copy-on-Write system. See figure 2.8 for a detailed flow-chart of Linux's page fault handler.

x86 processors don't make any difference between reading a memory page or executing code from it. We'll talk about it a bit later.

When getting a page fault on an x86 processor, Linux gets the address of the page that needs to be addressed and some flags (bitfield):

- bit 0 - 0: no page found 1: protection fault
- bit 1 - 0: read access 1: write access
- bit 2 - 0: kernel-mode access 1: user-mode access
- ...

If bit 0 is set, it means the processor tried to access a memory page (read/execute or write) without having the right to do so.



Access to kernel space
YES
NO
Access in Kernel Mode
vmalloc_fault
Noncontiguous memory area address
In interrupt, softirq, critical region, or kernel thread
Address in a memory region
Kernel page table entry fixup
Address could belong to User Mode stack
good_area
bad_area
Write access
Access in User Mode
Page is present
Region is writable
no_context
Region is readable or executable
Address is a wrong system call parameter
do_sigbus
Demand Paging and/or Copy On Write
Demand paging
Send SIGSEGV
Kill process and kernel "Oops"
"Fixup code" (typically send SIGSEGV)

Figure 2.8: Flow diagram of Linux's page fault handler. Copyright "Understanding the Linux Kernel"[10]

2.4 Security mechanisms in hardware

Now that we have roughly seen how the hardware works, let's see what security features are available to us.

2.4.1 Rings: The principle of least privilege applied to the CPU

The most famous hardware security feature is the privilege separation provided by the CPU's rings. On an x86 processor, there are 4 rings, going from the most privileged to the least privileged ($0 \mapsto 3$).

Virtualization-related ring -1 put aside, the level 0 is the most privileged level. It basically allows full access to the hardware.

On the contrary, the level 3 is only suitable for applications. Indeed, this level of privilege doesn't let the executed program exit its virtual memory space. If it tries to, the MMU would trap the call which would result in the OS killing

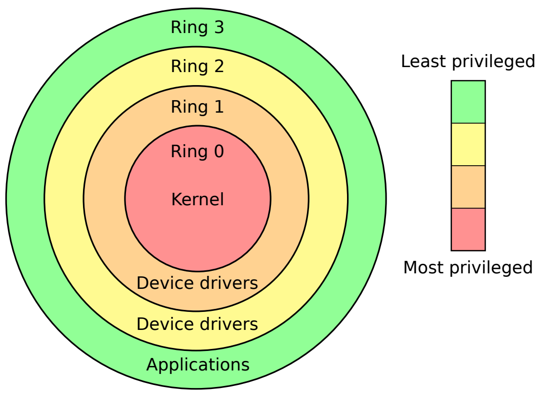


Figure 2.9: CPU's rings

the application. Though, the level 3 allows shared-memory access across applications, but it has to be set-up by the OS.

Typically, you'll find that only two rings are effectively used by the Operating Systems, the more privileged for the kernel-space and the least privileged for the user-space.

On a side note, changing the ring is really expensive. On Linux, it takes generally ∼1000 to ∼1500 CPU cycles for a round trip from the level 3 to the level 0 and back to the level 3 again. This is to be compared to the ∼100 CPU cycles it takes to perform a context-switch (switch from a process to an other, see scheduling).

2.4.2 The No eXecute bit

Initially introduced by AMD in his AMD64 architecture, the NX bit works around the security gap introduced by not differencing executions from reads in x86's memory pages access protocol.

When set on a memory page, this bit prevents the execution of code from this page. This allows the physical separation of the code from the data (stack, heap) of a program. This make applications less vulnerable to the exploitation of Buffer Overflows which are one of the primary vector of attack on a system where physical access is impossible.

Intel copied this extension in their architecture but named it the eXecute Disable(XD) bit.

2.4.3 Controlling direct memory accesses

Now that we have seen that applications are jailed into a virtual memory space, there are other devices which can access the memory.

Basically, we've already seen that any PCI, SATA, IDE or SCSI peripheral are able to access memory directly through the Southbridge and then the Northbridge This allows malicious devices to copy sensitive information from a process to another (malicious) process which could then send this data to anyone.

The problem is that DMA eludes the control of the Kernel which was supposed to be the only one to be able to access the memory directly. To address the problem, the Input/Ouput Memory Management Units(IOMMU) have been introduced.

2.4.4 IOMMU

The IOMMU's role is to restrict the different peripherals from accessing memory spaces that have not been specifically mapped for them.

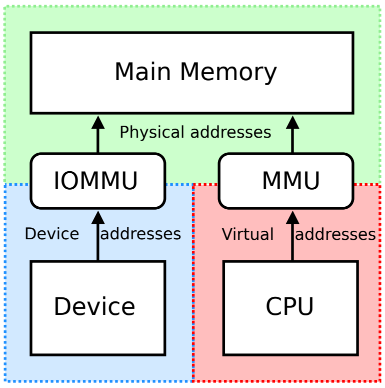


Figure 2.10: IOMMU versus MMU

The mapping process can only be done by the CPU (in ring 0, of course), this is the key to the security of the IOMMU.

The IOMMU is located in the Northbridge.

2.4.5 Memory access control

Just like the IOMMU can control DMA, we can use the TLB for some kind of access control.

x86 hardware support

x86 memory pages can be tagged with 2 security-related flags:

- write-protection: The memory page is read only. It can also be executed.
- supervision-mode: The memory page can only be accessed in ring 0

x86 memory pages can also be tagged with other flags:

- present: Is the page in the main memory?
- dirty: Has the page been modified?

Emulating full access control on Userspace programs

As explained in the PageExec[8] documentation, there is a trick to get a full access-control.

If we set the supervision-mode bit to every memory page, we will get protection faults traps from the MMU for every memory access made from the userspace. But if we also clear the present bit, we end up in the page fault handler with all the information we need to determine whether the access was a read, a write or an execution access.

2.5 Linux Security Systems

In the purpose of this article, we will now restrict our choice of Operating System to Linux only. The main reasons behind this choice are:

- Free/Libre and Open Source: We can see, edit, recompile and share our work on Linux
- Numerous alreadyimplemented Mandatory Access Controls: We can study them down to the source code
- LSM: A unique security framework that makes it simple to implement new MAC
- Very good Documentation: No need to reverse engineer everything
- Portability: It is meant to work on many architectures

2.5.1 Linux Security Module

LSM[9] is a unique feature among Operating Systems. It allows security modules targeted for Linux to be compiled built-in without having to manually hook all the security-related system calls. Instead, security modules would just need to fill a structure of function pointers so as LSM calls the security module by itself.

One strong advantage of the use of LSM over just letting developers hook the code by themselves is that LSM provides a stabler interface.

LSM was created when the NSA tried to merge SELinux mainline. Linus complained of the lack of consensus between security researchers on what is the right approach. This is why he asked the NSA developer to make SELinux as a module. In response, Linus got a proposition to split SELinux into two parts, the hook code (that is renamed LSM) and SELinux.

2.5.2 GrSecurity

GrSecurity[11] is a security suite that is not based on LSM. It is composed of:

- Pax: Memory-related security improvements
- RBAC: GrSec's implementation of the Role Based Access Control and the Domain and Type Enforcement models.
- Chroot-hardening: Makes it harder for a process to escape from a chroot.

Pax

Pax[12, 19, 20, 21, 22, 23, 24, 25, 26, 27] aims to increase the security of applications by randomizing the allocated/mmaped/stack and heap addresses to make it harder for attackers to exploit buffer overflows[13]. Pax also implements the emulation of the NX Bit on the x86_32 architecture.

A common problem of forbidding execution on writable pages is that it is incompatible with trampoline functions[29, 30]. This problem has been worked-around by introducing EmuTramp[28] in Pax.

Another solution is to compile everything with gcc's flag -fPIC (Position Independent Code).

RBAC

GrSecurity's RBAC implementation aims for efficiency, feature completeness and ease of use. It features a learning mode that allow system administrators to execute the program in a controlled environment in order to automatically learn the MAC policy for an application.

Chroot-hardening

Even though chroot is not a security feature, the GrSecurity developers decided to build on it by extending its capacities:

- Prevent chrooted application from communicating with other applications (only non-unix sockets are allowed)
- Do not allow chrooted application to connect to a Unix socket made by a non-chrooted application
- Do not allow applications to exit the chroot (no double chroot, hardened chdir("/"))
- ...

The goal is to provide a system like FreeBSD's light jails.

Trusted Path Execution

The goal of TPE[14] is to prevent users from executing programs that are not trusted by the system administrator. Technically, this boils down to preventing the user the execution of programs that are not stored into a root-owned directory such as /usr/bin and /bin.

Others

GrSecurity has a lot of features, it is impossible to even summarize them here. To get a complete list, please visit `http://grsecurity.net/features.php`.

2.5.3 SELinux

Security-Enhanced Linux[16] is a Mandatory Access Control following the RBAC, MLS and TDE. It has been developed by the USA's National Security Agency and has been released under the GNU General Public License on December 22, 2000.

SELinux is composed of a kernel part and a userspace tools and patched applications.

Kernel Inclusion, the birth of LSM

SELinux has been merged and made available in Linux 2.6.0 in August 2003. It took more than three years before the SELinux and Linux developers (and especially Linus) finally agree on making SELinux a module dependent of LSM.

Linus's main objection to the merge was that security people are incapable of agreeing on a single design. That is why he asked for more modularity. This lead to the creation of LSM, a set of hooks presenting an interface security module could link to.

The decision process

Contrary to most MAC systems in the kernel that uses filepath, SELinux bases his decision on security label that have been tagged into the ressources SELinux controls access to. For instance, contrary to AppArmor, when you hardlink a file on SELinux, the resulting file will share the same security label. This wouldn't be the case with AppArmor as the security label depends on the filepath of the new hardlink. The advantage of this technique is that it is compatible with any filesystem, may it be local or distant without any modification of the filesystem drivers.

Once an interaction has been retrieved by SELinux through LSM, SELinux needs to take a decision whether the said interaction is legal or not. To do so, SELinux uses its internal decision maker, the AVC.

To give a decision, the AVC needs:

- The source security ID(SID): The domain that initiated the interaction
- The target SID: The domain on which the action is performed
- The class: The class of action that is performed (security, process, system, capabilities, filesystem, file, dir, socket, ...)
- The action performed (read, write, delete, unlock, fork, ...)

The output of the AVC is a binary decision. In the case the decision was "denied", it is likely that the event got logged.

The AVC's decisions are based upon policies that have been written, compiled and sent from the userspace.

The AVC can be turned on and off by the setenforce (0|1) in userspace. You will need the sysadm_r role to do that.

Kernel traces

Unless asked otherwise, the AVC will log any denied interaction. These logs are available in the kernel logs.

You can see some kernel logs in the appendix A.2.

Chapter 3

Extending SELinux to track memory-pages accesses

This report is about my Master research project. It is about using the SELinux's AVC to implement a mandatory access control on memory pages.

3.1 Installing an SELinux Distribution

My first goal was to install an SELinux-enabled Linux distribution. I wanted to install it myself to be able to understand SELinux from the ground up.

There are two major Linux distributions that provide SELinux.

3.1.1 Fedora

Overview

Fedora is a mainstream Linux-based Operating System developed by the Fedora-project and sponsored by Red Hat. There is a new release every 6 months and the support of each version lasts 13 months.

By default, Fedora comes with SELinux installed in targeted mode. It means that only applications for which an SELinux has been written are properly sandboxed.

Another particularity of Fedora's SELinux base policy is that it uses both DTE and MLS for its security labels.

Installation

Just like Ubuntu, Fedora is very easy to install. You just need to insert the latest Fedora's CD into your computer and follow the steps on the screen.

If you are looking for ease of use, Fedora is a great fit.

Fedora SELinux's documentation is well-written, easy to find and understand. Fedora has some tutorials to get you started in SELinux.

Comments

Fedora looks good and is simple to use. On the other hand, I didn't get the possibility to easily do all the installation process by myself.

As I wanted to learn from the ground up how works an SELinux-enabled OS, Fedora wasn't what I needed.

3.1.2 Gentoo Hardened

Overview

Gentoo is an highly-customizable Linux distribution. Gentoo is often referred as a meta-distribution because of this.

Contrary to most Linux distributions, Gentoo doesn't provide Binary packages, it is the user's task to compile the package with the options he wants.

Another notable difference is that Gentoo is a rolling release. There are no version numbers, the distribution is updated every day and tries as much as possible to keep away from patching applications themselves.

Gentoo proposes an highly-secure version of itself called "Gentoo-Hardened". The typical Gentoo-Hardened distribution involves Pax, GrSecurity (without RBAC), SELinux and an hardened GCC.

Installation

The installation of a Gentoo Hardened is explained in Gentoo's SELinux Handbook available at the address `http://www.gentoo.org/proj/en/hardened/selinux/selinux-handbook.xml`.

It isn't as easy to follow as Fedora, but at least, you can understand what is done.

The kernel compilation and the installation of the hardened gcc went smoothly but when I tried to install Gentoo SELinux's base policy, I ran into multiple errors that I couldn't solve myself.

In the end, I have decided to use an image of an already installed Gentoo Hardened as understanding the Kernel part was more important to me than the Userspace one.

Comments

The outdated installation guide is a pity but other than that, Gentoo Hardened works as expected and probably provides the most hardened Linux OS possible.

This is the distribution I choose to work with.

3.2 Action Plan

Now that we have a working SELinux-enabled OS, it is time to start working on implementing the memory access control system.

The plan is to tag memory pages when they are created with the SID of the currently-running process.

Then, we rely on Pax's PAGEEXEC feature to generate page_faults whenever a process accesses a memory page.

The last step is to give the information to SELinux's AVC and enforce its decision (allow or deny).

3.2.1 Adding ourselves in KBuild

Before we start introducing code in the kernel, it is important to register our (not-yet-existing) feature to KBuild.

KBuild is the build system of the Linux kernel. It allows users to select what features they would like to have in the kernel.

Our feature being an SELinux memory access control, we'll modify the SELinux's KBuild file which is located at *security/selinux/KBuild* and add the following:

```
config SECURITY_SELINUX_MEMORY
    bool "NSA SELinux control userspace memory accesses"
    depends on SECURITY_SELINUX && EXPERIMENTAL
    help
    This option adds control over the userspace memory
    accesses.

    If you are unsure on how to answer this question,
    answer no.
```

config Give a unique name to the feature

bool Tell this feature is either activated or not (Boolean) and give a user-friendly name to it

depends on Tell that this feature can only be activated if SELinux and the experimental switch are activated

help The help text explaining what the feature is

We can now surround all our code related to this feature with *#ifdef CONFIG_SECURITY_SELINUX_MEMORY*.

3.2.2 Tag the memory pages

The first thing we need to do is to associate an SELinux SID to a memory page. This is done by adding a u32 in the *struct page* structure.

```
/*
 * Each physical page in the system has a struct page
 * associated with it to keep track of whatever it is
 * we are using the page for at the moment.
 * Note that we have no way to track which tasks are
 * using a page, though if it is a pagecache page, rmap
 * structures can tell us who is mapping it.
 */
struct page {
  unsigned long flags;   /* Atomic flags, some possibly
                          * updated asynchronously */
  atomic_t _count;       /* Usage count, see below. */

...

+ #ifdef CONFIG_SECURITY
+    u32 sid;
+ #endif
};
```

By default, SID will be 0. We now need to hook the page creation function in order to initialize the SID with the SID of the process which requested the allocation of the page.

3.2.3 Finding the functions to hook

According to what we saw earlier, the page fault handler and the page allocator seem to be places to hook.

Hooking the page allocator

Finding the real “Page Allocator” wasn’t an easy task. In the end, it seems like all the functions call the function *get_page_from_freelist*.

```
/*
 * get_page_from_freelist goes through the zonelist
 * trying to allocate a page.
 */
static struct page *
get_page_from_freelist(gfp_t gfp_mask,
        nodemask_t *nodemask, unsigned int order,
        struct zonelist *zonelist, int high_zoneidx,
        int alloc_flags, struct zone *preferred_zone,
        int migratetype)
{
        struct zoneref *z;
        struct page *page = NULL;
...
+       if (page)
+               security_page_alloc(page);

        return page;
}
```

This hook is special as we don't need to prevent the allocation. LSM just needs to know when a memory page has been allocated because

Hook the page fault handler

From what we saw earlier, the page fault handler is the place where we can do the memory access control. As we want to be good Linux citizens, we create an LSM hook for it.

As seen on figure 2.8, following the page fault handler's flow of execution is a bit difficult. Anyway, we only want SELinux to do the access control when we are sure the page is valid. So, no need to look before the *good_area* label.

I found out that trying to access the page before the call to *handle_mm_fault* was *really* painful. This is why I accepted that the page was mapped *before* I do any kind of access control. This shouldn't matter as I don't let the program execute any further (SIGSEGV) if SELinux denies access.

Here is the code:

```
/*
 * This routine handles page faults.  It determines the
 * address, and the problem, and then passes it off to
 * one of the appropriate routines.
 */
dotraplinkage void __kprobes
do_page_fault(struct pt_regs *regs,
              unsigned long error_code)
{

[...]

good_area:

[...]

  /*
```

```
     * If for any reason at all we couldn't handle
     * the fault, make sure we exit gracefully rather
     * than endlessly redo the fault:
     */
    fault = handle_mm_fault(mm, vma, address,
                            write ? FAULT_FLAG_WRITE : 0);
    if (unlikely(fault & VM_FAULT_ERROR)) {
      mm_fault_error(regs, error_code, address, fault);
      return;
    }

+   /* LSM HOOK */
+   page = follow_page(vma, address, FOLL_GET);
+   if (likely(error_code & PF_USER)) {
+     if (unlikely(security_page_fault(page, access))) {
+       /*bad_area_access_error(regs, error_code,
+                                    address);
+       return;*/
+     }
+   }

  [...]

}
```

3.2.4 Adding the LSM hooks

LSM is an abstraction layer, its sole purpose is to be called by the hooks and forward to the right security module. To do so, LSM uses a giant structure in which it stores function pointers. These pointers are then initialized by the LSM module that have been chosen by the Linux user.

Add the function pointers

The structure we need to hack is *struct security_operations* located in *include/linux/security.h*. Let's add the two new function pointers we need at the end of *struct security_operations*'s definition:

```
struct security_operations {
    char name[SECURITY_NAME_MAX + 1];
  [...]
+ #ifdef CONFIG_SECURITY_SELINUX_MEMORY
+   int (*page_fault) (struct page *page,
+                      enum page_access access);
+   int (*page_alloc) (struct page *page);
+ #endif /* CONFIG_SECURITY_SELINUX_MEMORY */
};
```

Creating hook functions

To create new hook functions, you need to define them in *security/security.c*. In our case, we add at the end of the file this code:

```
+ int security_page_fault(struct page *page,
+                         enum page_access access)
+ {
+   /* Test that security_ops has been initialised
+    * before jumping to the right function.
+    */
+   if (security_ops)
+     return security_ops->page_fault(page, access);
+   else
+     return 0;
+ }
+
+ int security_page_alloc(struct page *page)
+ {
+   /* We test for security_ops->page_alloc because
+    * page_alloc shouldn't be called if SELinux
+    * has not started yet. This means that at the
+    * end of SELinux's starting process, we need to
+    * set the security_ops->page_alloc function
+    * pointer to the right place
+    * (see selinux_complete_init() in
+    * security/selinux/hooks.c).
+    */
+   if (security_ops && security_ops->page_alloc)
+     return security_ops->page_alloc(page);
+   else
+     return 0;
+ }
```

Once we have defined the functions, we need to make them public by adding them at the end of *include/linux/security.h*.

```
static inline void free_secdata(void *secdata)
{ }
#endif /* CONFIG_SECURITY */
+ int security_page_fault(struct page *page,
+                         enum page_access access);
+ int security_page_alloc(struct page *page);
        #endif /* ! __LINUX_SECURITY_H */
```

3.2.5 From LSM to the SELinux's AVC

The LSM work is now done but we still need to implement SELinux's side.

Let's first implement the code that will receive the interaction from LSM and transmit it to SELinux's AVC:

```
#ifdef CONFIG_SECURITY_SELINUX_MEMORY
int selinux_page_fault(struct page *page,
                       enum page_access access)
{
        u32 sid = current_sid();
        u32 tsid = 0;
        int memory_class = MEMORY_READ;

        if (!page) {
                return 1;
        }
        tsid = page->sid;

        switch (access)
        {
        case page_read:
                memory_class = MEMORY_READ;
                break;
        case page_write:
                memory_class = MEMORY_WRITE;
                break;
        case page_exec:
                memory_class = MEMORY_EXEC;
                break;
        }

        return avc_has_perm(sid, tsid, SECCLASS_MEMORY,
                            memory_class, NULL);
}

int selinux_page_alloc(struct page *page)
{
        u32 sid = current_sid();

        if (sid == SECINITSID_UNLABELED)
                sid = SECINITSID_KERNEL;

        /* Label the memory pages */
        page->sid = sid;

        return 0;
}
#endif
```

Then, we need to register these functions into the SELinux's *struct security_operations*:

```
static
struct security_operations selinux_ops __read_only = {
  .name =        "selinux",

 [...]

#ifdef CONFIG_SECURITY_SELINUX_MEMORY
  .page_fault = selinux_page_fault ,
  .page_alloc = NULL,
#endif
};
```

As you can see, page_alloc is initialized to NULL. This is because this function shouldn't be called before SELinux has started. The initialization of the page_alloc function pointer is then done in the function selinux_complete_init():

```
void selinux_complete_init(void)
{
  printk(KERN_DEBUG "SELinux:  "
         "Completing initialization.\n");

  /* Set up any superblocks initialized prior
   * to the policy load.
   */
     printk(KERN_DEBUG "SELinux:  "
            "Setting up existing superblocks.\n");
     iterate_supers(delayed_superblock_init , NULL);

     printk(KERN_DEBUG "SELinux:  "
            "Starting memory tagging\n");
     selinux_ops.page_alloc = selinux_page_alloc;
}
```

3.3 Analysis

Recompiling the kernel and booting it leads to literally hundreds of thousands of lines in the kernel logs.

Looking at them revealed only a few untagged pages that could have been allocated before SELinux started as they were all related to the kernel logs.

Coverage

Given how low-level the memory protection we have implemented is, it is very difficult to test if it works in all the situations.

This is why I created a simple test program that you can see in the appendix B.3. The goal of this program is to first generate a read and a write access on a global and local variable and then on the heap. Secondly, it tries to execute a shellcode (to test PAX). The output of the program is available in the appendix B.3.2.

Analyzing the SELinux logs shows that all the read pages are untagged. This shouldn't be the case and means that the SID stored in the struct page is not permanent. There is still room for improvement.

Performance

When I first thought about doing this kind of memory access control, I thought the performance hit would be tremendous but, when I tried it, I couldn't feel any difference in speed. I then looked at some old benchmarks of the PAGEEXEC implementation[18] and realized how light was the performance hit.

In our case, the performance is roughly the same as we just add the cost of SELinux's AVC to the PAGEEXEC cost.

Of course, we need proper benchmarks to really put a boundary to the performance hit.

3.4 Conclusion

According to the results so far, it seems possible to implement an efficient Mandatory Access Control on memory pages. The method has been known for at least 10 years, yet, tried to implement it.

The reason for that may lie in the complexity of tagging memory and writing policies for it.

3.5 Perspectives

When fixed, this research project could be used by the Userspace to tag the memory it allocates through malloc to achieve a greater granularity of memory sharing. With such a system, some applications like the x-server could no longer need to access all the memory but only some memory pages.

Another perspective is the ability to write a system-wide information flow control by feeding the information gathered by SELinux's AVC to PIGA[31]. PIGA could use the information from this research project to look for hidden communication channels without always being pessimistic.

Appendix A

SELinux

A.1 Writing an SELinux policy

The best way to write a policy for an application is to write an selinux module which is composed of three files:

- my_module.te: The type-enforcement file. Defines the allowed interactions.
- my_module.fc: The file-constraint file. Defines the security label that will be given to a file.
- my_module.if: The interface file. This file is generally useless.

my_module.te The type-enforcement file Let's see an example of a template of a .te file:

```
policy_module(${module},1.0.0)

require { type ${user_t}, tmp_t, var_log_t;}

type ${module_exec_t};
type ${module_domain_t};
type ${module_tmp_domain_t};
type ${module_log_domain_t};

#transition
role ${user_r} types ${module_domain_t};
domain_type(${module_domain_t});
domain_entry_file(${module_domain_t}, ${module_exec_t})
domtrans_pattern(${user_t}, ${module_exec_t}, \
        ${module_domain_t});

type_transition ${module_domain_t} tmp_t:file \
        ${module_tmp_domain_t};
type_transition ${module_domain_t} var_log_t:file \
        ${module_log_domain_t};
```

Syntax The .te file syntax explained a bit

- policy_module: Defines the name of the policy module and the version number
- require: The external domain dependencies of the policy module
- type: Type declarated in this module
- #transition: The minimum set of rules to confine an application into a domain at boot time
- role: Define a domain as a valid role for a domain
- domain_type: A macro to define a new domain
- domain_entry_file: A macro to define the domain in which an app should be executed in
- domtrans_pattern: A macro to allow $user_t to execute $module_exec_t in the $module_domain_t domain
- type_transition: *If domain $module_domain_t writes a file that would take the tmp_t as a domain, change its security label to $module_tmp_domain_t*

Template variables The meaning of the template variables

- module: The name of the module
- user_t: The domain from which the user will launch the application
- user_r: The role of the user that will launch the application
- module_exec_t: The domain that will be given to the app's binary (file label)
- module_domain_t: The domain in which the application will be sandboxed
- module_tmp_domain_t: The domain that will hold tmp files (file label)
- module_log_domain_t: The domain that will hold log files (file label)

my_module.fc The file-constraint file

```
###################################
############            ############
############ WARNING ############
############            ############
###################################
# If you update this file ,
# please remember to add the types in the te file
#

# Store ${app_path} under the
# security label system_u:object_r:${module_exec_t}
${app_path}      --      system_u:object_r:${module_exec_t}
```

```

# Example of tagging a complete directory or anything
# else if you write the right regexp
#HOME_DIR/\.myapp(/.*)? user_u:object_r:myapp_home_t
```

A.2 Example of SELinux kernel logs

```
type=1405 audit(1297364131.091:3): bool=
    xserver_object_manager val=1 old_val=0 auid=4294967295
     ses=4294967295

type=1404 audit(1297364131.275:4): enforcing=1
    old_enforcing=0 auid=4294967295 ses=4294967295

type=1400 audit(1297364132.331:5): avc:  denied  { bind }
     for  pid=1765 comm="agetty" ppid=1 scontext=system_u:
    system_r:getty_t tcontext=system_u:system_r:getty_t
    tclass=netlink_route_socket

type=1400 audit(1297364132.336:6): avc:  denied  { getcap
     } for  pid=1649 comm="syslog-ng" ppid=1648 scontext=
    system_u:system_r:syslogd_t tcontext=system_u:system_r
    :syslogd_t tclass=process

type=1400 audit(1297364132.336:9): avc:  denied  { bind }
     for  pid=1764 comm="agetty" ppid=1 scontext=system_u:
    system_r:getty_t tcontext=system_u:system_r:getty_t
    tclass=netlink_route_socket

type=1400 audit(1297364360.545:14): avc:  denied  {
    module_request } for  pid=1770 comm="sshd" ppid=1727
    kmod="net-pf-10" scontext=system_u:system_r:sshd_t
    tcontext=system_u:system_r:kernel_t tclass=system

type=1400 audit(1297364361.477:15): avc:  denied  {
    search } for  pid=1770 comm="sshd" ppid=1727 name="
    dbus" dev=sdb1 ino=73300 scontext=system_u:system_r:
    sshd_t tcontext=system_u:object_r:
    system_dbusd_var_run_t tclass=dir
```

Appendix B

Useful Pax documentation

B.1 PageExec

B.1.1 Design

The goal of PAGEEXEC is to implement the non-executable page feature using the paging logic of IA-32 based CPUs.

Traditionally page protection is implemented by using the features of the CPU Memory Management Unit. Unfortunately IA-32 lacks the hardware support for execution protection, i.e., it is not possible to directly mark a page as executable/non-executable in the paging related structures (the page directory (pde) and table entries (pte)). What still makes it possible to implement non-executable pages is the fact that from the Pentium series on the Intel CPUs have a split Translation Lookaside Buffer for code and data (AMD CPUs have a split TLB since the K5 series however due to its organization it is usable for our purposes only since the K7 core based CPUs).

The role of the TLB is to act as a cache for virtual/physical address translations that the CPU has to perform for every single memory access (be that instruction fetch or data read/write). Without the TLB the CPU would have to perform an expensive page table walk operation for every such memory access and obviously that would be detrimental to performance.

The TLB operates in a simple manner: whenever the CPU wants to access a given virtual address, it will first check whether the TLB has a cached translation or not. On a TLB hit it will take the physical address directly from the TLB, otherwise it will perform a page table walk to look up the required translation and cache the result in the TLB as well (if the page table walk is unable to find the translation or the result is in conflict with the access type, e.g., a write to a read-only page, then the CPU will instead raise a page fault exception). Note that hardware assisted page table walking and automatic TLB loading are features specific to IA-32, other CPUs may have or need software assistance in this operation. Since the TLB has a finite size, sooner or later it becomes full and the CPU will have to purge entries to make room for new translations (on IA-32 this is again automatically done in hardware). Software can also purge TLB entries by either removing all translations (e.g., whenever a userland context switch happens) or those corresponding to a specific virtual address.

As mentioned already, from the Pentium on Intel CPUs have a split TLB, that is, virtual/physical translations are cached in two independent TLBs depending on the access type: instruction fetch related memory accesses will load the ITLB, everything else loads the DTLB (if both kinds of accesses are made to a page then both TLBs will have an entry). TLB entry replacement works also on a per TLB basis except for the software initiated purges which act on both.

The above described TLB behaviour means that software has explicit control over ITLB/DTLB loading: it can get notified on hardware TLB load attempts if it sets up the page tables so that such attempts will fail and trigger a page fault exception, and it can also initiate a TLB load by making the appropriate memory access to have the CPU walk the page tables and load one of the TLBs. This in turn is the key to implement non-executable pages: such pages can be marked either as non-present or requiring supervisor level access in the page tables hence userland memory accesses would raise a page fault. The page fault handler can then decide whether it was an instruction fetch attempt (by comparing the fault address to that of the instruction that raised the fault) or a legitimate data access. In the former case we will have detected an execution attempt in a non-executable page and can act accordingly (terminate the task), in the latter case we can just change the affected page table entry temporarily to allow user level access and have the CPU load it into the DTLB (we will of course have to restore the page table entry to the old state so that further page table walks will again raise a page fault).

The decision between using non-present or supervisor mode page table entries for marking a page as non-executable comes down to performance in the end, the latter being less intrusive because kernel initiated data accesses to userland pages will not raise a page fault.

To sum it up, PAGEEXEC as implemented in PaX overloads the meaning of the User/Supervisor bit in the ptes to mean the executable/non-executable status and also makes sure that data accesses to non-executable pages still work as before.

B.1.2 Implementation

PAGEEXEC requires two sets of changes in Linux: the kernel has to be taught that the i386 architecture can do the proper non-executable semantics, and next we have to deal with the special page faults that require kernel assisted DTLB loading.

The lowlevel definitions of the capabilities of the paging logic are in include/asm-i386/pgtable.h. Here we simply redefine the constants that are used for creating the ptes of non-executable pages. One such use of these constants is the protection_map[] array defined in mm/mmap.c which is referenced whenever the kernel sets up a pte for a userland mapping. Since PAGEEXEC can be disabled on a per task basis we have to modify all code that accesses this array so that we provide an executable pte even if it was not explicitly requested. Affected files include fs/exec.c (where the stack pages are set up), mm/mprotect.c, mm/filemap.c and mm/mmap.c. The changes in the latter two cooperate in a nontrivial way: do_mmap_pgoff() creates executable nonanonymous mappings by default and it is the job of generic_file_mmap() to turn it into a non-executable one (as the mapping turned out to be a file mapping). This logic ensures that non-anonymous mappings of devices remain executable regardless of PAGE-

EXEC. We opted for this approach to remain as compatible as possible (by not affecting all non-anonymous mappings) yet still make use of the non-executable feature in the most frequently encountered case.

The kernel assisted DTLB loading logic is in the IA32 specific page fault handler which in Linux is do_page_fault() in arch/i386/mm/fault.c. For easier code maintenance we created our own page fault entry point called pax_do_page_fault() which gets called first from the lowlevel page fault exception handler page_fault found in arch/i386/kernel/entry.S.

First we verify that the given page fault is ours by checking for a userland fault caused by access conflict (vs. not present page). Next we pay special attention to faults caused by an instruction fetch since this means an attempt of code execution in a non-executable page. Such faults are easily identified by checking for a read access where the target address of the page fault is equal to the userland instruction pointer (which is saved by the CPU at the time of the fault for us). The default action is of course a task termination along with a log message, only EMUTRAMP when enabled can change it (see separate document).

Next we prepare the mask for setting up the special pte for loading the DTLB and then we acquire the spinlock that guards MMU state changes (since we are about to cause such a change ourselves). Holding the spinlock is also necessary for looking up the target pte that we will modify and load into the DTLB. If the pte state we looked up no longer corresponds to the fault type then we must have raced with other MMU state changing code and pass down the fault to the original fault handler. It is also the time when we can identify (and pass down) copy-on-write page faults that have the same fault type but a different pte state than what is caused by the PAGEEXEC logic.

Finally we change the pte to allow userland accesses to the given page then perform a dummy read memory access that will have the CPU page table walk logic load it into the DTLB and then we change the state back to be in supervisor mode. There is a trick in this part of the code that is worth a few words. If the TLB already has an entry for a given virtual/physical translation then initiating a memory access will not cause a page table walk, that is, for our DTLB loading to work we would have to ensure that the DTLB has no entries for our virtual address. It turns out that different members of the Intel IA-32 family have a different behaviour when the CPU raises a page fault during a page table walk (which is our case): the old Pentium (but not the MMX version) CPUs would still cache the translation if it described a present mapping but had an access conflict (which is our case since we have a supervisor mode pte that is accessed while executing in user mode) whereas newer CPUs (P6 core based ones, P4 and probably future CPUs as well) would not cache them at all. This means that in the second case we can be sure that the DTLB has no translations for our target virtual address and can omit a very expensive 'invlpg' instruction (it sped up the fast path by some 20% on a P3).

B.2 SegMem

B.2.1 Design

The goal of SEGMEXEC is to implement the non-executable page feature using the segmentation logic of IA-32 based CPUs.

On IA-32 Linux runs in protected mode with paging enabled. This means that for every memory access (be that instruction fetch or normal data access) the CPU will perform a two step address translation. In the first step the logical address decoded from the instruction is translated into a linear (or in another terminology, virtual) address. This translation is done by the segmentation logic whose details are explained in a separate document.

While Linux effectively does not use segmentation by creating 0 based and 4 GB limited segments for both code and data accesses (therefore logical addresses are the same as linear addresses), it is possible to set up segments that allow to implement non-executable pages.

The basic idea is that we divide the 3 GB userland linear address space into two equal halves and use one to store mappings meant for data access (that is, we define a data segment descriptor to cover the 0-1.5 GB linear address range) and the other for storing mappings for execution (that is, we define a code segment descriptor to cover the 1.5-3 GB linear address range). Since an executable mapping can be used for data accesses as well, we will have to ensure that such mappings are visible in both segments and mirror each other. This setup will then separate data accesses from instruction fetches in the sense that they will hit different linear addresses and therefore allow for control/intervention based on the access type. In particular, if a data-only (and therefore non-executable) mapping is present only in the 0-1.5 GB linear address range, then instruction fetches to the same logical addresses will end up in the 1.5-3 GB linear address range and will raise a page fault hence allow detecting such execution attempts.

B.2.2 Implementation

The core of SEGMEXEC is vma mirroring which is discussed in a separate document. The mirrors for executable file mappings are set up in do_mmap() (an inline function defined in include/linux/mm.h) except for a special case with RANDEXEC (see separate document). do_mmap() is the one common function called by both userland and kernel originated mapping requests.

The special code and data segment descriptors are placed into a new GDT called gdt_table2 in arch/i386/kernel/head.S. The separate GDT is needed for two reasons: first it simplifies the implementation in that the CS/SS selectors used for userland do not have to change, and second, this setup prevents a simple attack that a single GDT setup would be subject to (the retf and other instructions could be abused to break out of the restricted code segment used for SEGMEXEC tasks). Since the GDT stores the userland code/data descriptors which are different for SEGMEXEC tasks, we have to modify the lowlevel context switching code called __switch_to() in arch/i386/kernel/process.c and the last steps of load_elf_binary() in fs/binfmt_elf.c (where the task is first prepared to execute in userland).

The GDT also has APM specific descriptors which are set up at runtime and must be propagated to the second GDT as well (in arch/i386/kernel/apm.c).

Finally the GDT stores also the per CPU TSS and LDT descriptors whose content must be synchronized between the two GDTs (in set_tss_desc() and set_ldt_desc() in arch/i386/kernel/traps.c).

Since the kernel allows userland to define its own code segment descriptors in the LDT, we have to disallow it since it could be used to break out of the SEGMEXEC specific restricted code segment (the extra checks are in write_ldt() in arch/i386/kernel/ldt.c).

B.3 The coverage test

B.3.1 The test itself

```
#include <stdio.h>
#include <malloc.h>

char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

int global;

void test_value(const char *var_name, int *value)
{
        printf("Test variable '%s' (@=%p)\n", var_name, value);
        printf("        write: ", value);
        *value = 0x42;
        printf("  ok\n");

        printf("        read : ", value);
        printf("0x%x\n\n", *value);
}

int main(int argc, char **argv)
{
        int local;

        test_value("local", &local);
        test_value("global", &global);

        printf("Will create an heap variable: ");
        int *heap = (int*)malloc(sizeof(int));
        printf("ok (@=%p)\n", heap);

        test_value("heap", heap);

        printf("Will now execute the shellcode\n");
        (*(void(*)()) shellcode)();

        return 0;
}
```

B.3.2 Output of the program on the hardened Kernel

```
pigaos ~ # ./a.out
Test variable 'local' (@=0x5b45c2d8)
        write:  ok
        read : 0x42

Test variable 'global' (@=0x804a078)
        write:  ok
        read : 0x42

Will create an heap variable: ok (@=0x80548b8)
Test variable 'heap' (@=0x80548b8)
        write:  ok
        read : 0x42

Will now execute the shellcode
Processus arrete
```

B.3.3 The generated SELinux trace

```
[ 5408.342034] avc:  granted  { execute } for  pid=1836
   comm="bash" name="a.out" dev=sda3 ino=130496 scontext=
   root:sysadm_r:sysadm_t tcontext=root:object_r:
   user_home_t tclass=file

[ 5408.342034] avc:  granted  { execute_no_trans } for
   pid=1836 comm="bash" path="/root/a.out" dev=sda3 ino
   =130496 scontext=root:sysadm_r:sysadm_t tcontext=root:
   object_r:user_home_t tclass=file

[ 5408.344663] avc:  granted  { execute } for  pid=1836
   comm="a.out" path="/root/a.out" dev=sda3 ino=130496
   scontext=root:sysadm_r:sysadm_t tcontext=root:object_r
   :user_home_t tclass=file

[ 5408.344900] avc:  denied  { read } for  pid=1836 comm
   ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
   system_u:object_r:unlabeled_t tclass=memory

[ 5408.344928] avc:  denied  { read } for  pid=1836 comm
   ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
   system_u:object_r:unlabeled_t tclass=memory

[ 5408.344951] avc:  denied  { read } for  pid=1836 comm
   ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
   system_u:object_r:unlabeled_t tclass=memory

[ 5408.344974] avc:  denied  { write } for  pid=1836 comm
   ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=root
```

```
:sysadm_r:sysadm_t tclass=memory

[ 5408.345003] avc: denied { read } for pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.345022] avc: denied { read } for pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.345042] avc: denied { read } for pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.345057] avc: denied { read } for pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.345072] avc: denied { read } for pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.345086] avc: denied { read } for pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.345101] avc: denied { read } for pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.345126] avc: denied { read } for pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=root:sysadm_r:sysadm_t tclass=memory

[ 5408.345146] avc: denied { read } for pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=root:sysadm_r:sysadm_t tclass=memory

[ 5408.345160] avc: denied { read } for pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.345175] avc: denied { read } for pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.345190] avc: denied { read } for pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.345210] avc:  denied  { write } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=root:sysadm_r:sysadm_t tclass=memory

[ 5408.345226] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.345256] avc:  denied  { write } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=root:sysadm_r:sysadm_t tclass=memory

[ 5408.345270] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.345274] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.345274] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.345274] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.345274] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.345274] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.345274] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.345274] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.345274] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.345274] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.345274] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.345274] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.345274] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.345274] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.345274] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.345274] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.345274] avc:  denied  { write } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=root
    :sysadm_r:sysadm_t tclass=memory

[ 5408.345274] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.345274] avc:  denied  { write } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=root
    :sysadm_r:sysadm_t tclass=memory

[ 5408.347922] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.347947] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.347970] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

```

```
[ 5408.347996] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.348227] avc:  denied  { write } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=root:sysadm_r:sysadm_t tclass=memory

[ 5408.348255] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.348280] avc:  denied  { write } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=root:sysadm_r:sysadm_t tclass=memory

[ 5408.348299] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.348320] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.348340] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.348361] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.348376] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.348391] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.348405] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.348419] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.348433] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.348448] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.348462] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.348477] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.348492] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.348506] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.348521] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.348535] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.348550] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.348570] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.348585] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.348606] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.348921] avc:  denied  { write } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=root
    :sysadm_r:sysadm_t tclass=memory

```

```
[ 5408.348942] avc:  denied  { write } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=root:sysadm_r:sysadm_t tclass=memory

[ 5408.348966] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.348989] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.349012] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
        system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
      ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
       system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
      ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
       system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
      ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
       system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
      ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
       system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
      ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
       system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
      ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
       system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
      ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
       system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
      ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
       system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
      ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
       system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { write } for  pid=1836 comm
      ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=root
       :sysadm_r:sysadm_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
      ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
       system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
      ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
       system_u:object_r:unlabeled_t tclass=memory

```

```
[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { read } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=
    system_u:object_r:unlabeled_t tclass=memory

[ 5408.349027] avc:  denied  { write } for  pid=1836 comm
    ="a.out" scontext=root:sysadm_r:sysadm_t tcontext=root
    :sysadm_r:sysadm_t tclass=memory

[ 5408.349800] attempt in: /root/a.out, 0804a000-0804b000
     00001000

[ 5408.349800] task: /root/a.out(a.out):1836, uid/euid:
    0/0, PC: 0804a040, SP: 5e2aeb6c

[ 5408.349800] at PC: eb 1f 5e 89 76 08 31 c0 88 46 07 89
     46 0c b0 0b 89 f3 8d 4e

[ 5408.351088] at SP-4: 5e2aeb98 080485d8 08048738
    08055188 080485f0 5e2aeb98 5234a605 52484b80 00000042
    08055188 080485f0 00000000 5e2aec18 52333bd6 00000001
    5e2aec44 5e2aec4c 523296a0 524743d0 524743d0 ffffffff
```

Bibliography

[1] Roger Needham, *Protection systems and protection implementations*, Proc. 1972 Fall Joint Computer Conference, AFIPS Conf. Proc., vol. 41, pt. 1, pp. 571-578

[2] James P. Anderson, *Computer Security Threat Monitoring and Surveillance*, James P. Anderson Co., Fort Washington, PA (Apr. 1980)

[3] Robert L. Brotzman, CSC-STD-004-85: *Computer Security Requirements - Guidance For Applying The Department Of Defense Trusted Computer System Evaluation Criteria In Specific Environments* (June 25, 1985)

[4] D. E. Bell and L. J. La Padula. *Secure computer systems : Mathematical foundations and model.* Technical Report M74-244, The MITRE Corporation, Bedford, MA, May 1973.

[5] Biba, K. J. *Integrity Considerations for Secure Computer Systems*, MTR-3153, The Mitre Corporation, April 1977.

[6] Boebert, W. E. and Kain, R. Y. . *A practical alternative to hierarchical integrity policies.* In The 8th National Computer Security Conference, pp. 18–27, Gaithersburg, MD, USA. (1985)

[7] Ferraiolo, D.F. and Kuhn, D.R. (October 1992). *Role-Based Access Control.* 15th National Computer Security Conference. pp. 554–563

[8] `http://pax.grsecurity.net/docs/pageexec.txt`

[9] Chris Wright and Crispin Cowan. *Linux Security Modules: General Security Support for the Linux Kernel.* August 2002

[10] Daniel P. Bovet, Marco Cesati. *Understanding the Linux Kernel.* O'Reilly, Third Edition. November 2005

[11] Spengler, B. (2002). *Detection, prevention, and containment : A study of grsecurity.* In Libre Software Meeting 2002 (LSM2002), Bordeaux, France. `http://www.grsecurity.net/papers.php`.

[12] `http://pax.grsecurity.net/docs/pax.txt`

[13] Elias Levy. *Smashing the Stack for Fun and Profit.* August 1996

[14] Niki A. Rahimi. *Trusted path execution for the linux 2.6 kernel as a linux security module.* In Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '04). USENIX Association, Berkeley, CA, USA, 34-34.

[15] Kernel Trap. *Abusing Chroot.* 2007. `http://kerneltrap.org/Linux/Abusing\_chroot`

[16] Bill Mccarty, *SELinux: NSA's Open Source Security Enhanced Linux.* O'Reilly, First Edition, October 1994.

[17] Frank Mayer, Karl MacMillan, David Caplan, *SELinux By Example.* Prentice Hall, First Edition (August 6, 2006).

[18] `http://www.pjvenda.net/linux/doc/pax-performance/`

[19] `http://pax.grsecurity.net/docs/noexec.txt`

[20] `http://pax.grsecurity.net/docs/aslr.txt`

[21] `http://pax.grsecurity.net/docs/segmexec.txt`

[22] `http://pax.grsecurity.net/docs/mprotect.txt`

[23] `http://pax.grsecurity.net/docs/randustack.txt`

[24] `http://pax.grsecurity.net/docs/randkstack.txt`

[25] `http://pax.grsecurity.net/docs/randmmap.txt`

[26] `http://pax.grsecurity.net/docs/randexec.txt`

[27] `http://pax.grsecurity.net/docs/vmmirror.txt`

[28] `http://pax.grsecurity.net/docs/emutramp.txt`

[29] Thiago Macieira. *Moving code around.* December 2010. `http://labs.qt.nokia.com/2010/12/04/moving-code-around/`

[30] Thiago Macieira. *Moving code around more easily.* December 2010. `http://labs.qt.nokia.com/2010/12/05/moving-code-around-more-easily/`

[31] M. Blanc, J. Briffaut, J.-F. Lalande, and C. Toinard. *Enforcement of security properties for dynamic mac policies.* In IARIA, editor, Third International Conference on Emerging Security Information, Systems and Technologies, pages 114–120, Athens/Vouliagmeni, Greece, June 2009. IEEE Computer Society Press.