Linux Enhanced Security

Reference Manual

v0.2

(rev 20040103)

By: Pedro Hortas & Artur D'Assumpção

TABLE OF CONTENTS

1

Linux Enhanced Security Overview

CHAPTER 1
LINUX ENHANCED SECURITY OVERVIEW

1 CHAPTER 1: THE LINUX ENHANCED SECURITY OVERVIEW

1.1 THE LINUX ENHANCED SECURITY PROJECT

The `Linux Enhanced Security` is a free software project, released under the `GNU General Public License`, which aims at the development of `Linux` based security patches as well as user-land tools meant to aid security and provide `GNU/Linux` based system administrators with extended abilities and control over their hosts and networks.

→ Visit our web site at `http://www.lesecurity.org`.

1.2 ARCHITECTURE SUPPORT

The `Linux Enhanced Security` is meant to support multiple architectures. Although current development releases supports only `x86`, future releases will support a broader set of architectures.

1.2.1 Supported Architectures and Architecture Dependent Features

Since multiple architectures are supported, it is expectable to have certain architecture specific options available. We can't forget also, that each architecture has its unique behaviour and this will have direct influence in the behaviour of `Linux Enhanced Security` features.

1.2.1.1 x86 Architecture (32 bits) (i386)

This architecture is currently supported. All features were designed to work with it.

1.2.1.2 IA-64 Architecture

Not currently supported (support will be probably added on series 1.x).

1.2.1.3 x86-64 Architecture (EM64T based chipsets)

Not currently supported (support will be probably added on series 1.x).

1.2.1.4 Sparc Architecture

Not currently supported (support will be probably added on series 1.x).

1.2.1.5 Alpha Architecture

Not currently supported (support will be probably added on series 1.x).

1.2.1.6 PPC Architecture

Not currently supported (support will be probably added on series 1.x).

1.2.2 SMP Support

When `Symmetric Multi-Processing` (`SMP`) support is compiled, the `Kernel` is prepared to handle more than one processor at the same time. This kind of support will need locking mechanisms that allow concurrent accesses to the same shared memory region work with data integrity guarantees.

1.2.2.1 SMP Safety

All features provided by `Linux Enhanced Security` are `SMP` safe and should work as expected.

1.3 THE LINUX ENHANCED SECURITY INTERNALS

Whenever a `Linux Enhanced Security` features is triggered within the `Kernel`, a well defined `API` is used to access important structures and execute those functionalities.

1.3.1 The lesec() System Call

The `lesec()` system call was designed to allow the interaction between `user-space` applications and `kernel-space` features implemented by `Linux Enhanced Security`.

This system call can only be executed by the `super-user` and has the following prototype:

```
int lesec(int call, void *data, int op);
```

The `call` argument expects an integer value that specifies which feature we're invoking. For instance, if we wish to call the `chpown` (→ *see section 5.1*) operation, then we'll need to specify the `call` argument as `"LESEC_CHPOWN_CALL"`.

The `data` argument expects a pointer to a structure needed by call handlers to perform the requested operation. For instance, the `"LESEC_CHPOWN_CALL"` has a handler, the `chpown_call()` system call, which expects a structure containing an `uid`, `gid` and `pid` values whenever the `"CHPOWN_WRITE"` operation is requested.

The `op` argument specifies the operation that will be executed. For instance, if we wish to write data into the `Kernel`, to be handled by the `"LESEC_CHPOWN_CALL"` handler, then we'll specify the operation as `"CHPOWN_WRITE"`.

These interactions are made by all implemented features that need access to `kernel-space`.

Related Sections

1.3.2 Accessing Data Structures
5.1 Change Process Owner (chpown)

1.3.2 Accessing Data Structures

Whenever a pointer to a data structure is passed as an argument to the `lesec()` system call, the `kernel-space` implementation needs to know what kind of structure it is. We identify it trough the `call` and `op` arguments, which will then allow us to cast the void pointer to a known data structure.

After the type cast we copy the entire data structure from the `user-space` memory to `kernel-space` memory, avoiding direct interactions with `user-space` memory. This is done using the `copy_from_user()` system call. Data access is only available when all `user-space` memory is mapped in `Kernel` memory.

1.4 THE LSM IMPLEMENTATION OVERVIEW

The `Linux Security Modules` (`LSM`) implementation was designed to allow programmers to write modules which can be loaded and binded to special security hooks provided by the `Kernel`. This allows the creation of security enhancements in the `Kernel` layer just by loading one or more modules avoiding the trouble of patching the source and recompiling it all over again.

1.4.1 Problem

At first sight the `LSM` framework seems to be very useful and innovative, avoiding the patching of the `Kernel` source, as it was already said. But its implementation is so flawed and incomplete that simply fails to accomplish its main purposes. Some points of view:

➔ **It's not possible to make any security enhancements that are dependent of data structure restructuring**

So if you're coding a `LSM` and at the same time need to patch the `Kernel` to restructure data structures, there's no sense in coding the `LSM`. It makes much more sense to patch everything over the `Kernel` and recompile it, since you'll be doing it any way.

➔ **If you're using more than one security module, you can't guarantee their actions**

If a module returns immediately on a special security hook, there isn't any type of recursion that will lead him into others.

➔ **There are lots of places in the Kernel that doesn't have any special security hooks**

If you need to trig in any of these places you'll need to patch over the `Kernel` source. Once more, if you need to patch, recompile the `Kernel` and reboot, then it makes much more sense to patch everything, avoiding the complexity of the security module implementation.

1.4.2 Solution

To solve this problem, we've decided that the `Linux Security Modules` implementation isn't suitable for the `Linux Enhanced Security` development. Therefore there will be no support or expected compatibility when enabling the `Linux Security Modules` feature within the `Kernel`.

1.5 DEVELOPMENT RULES

To maintain code and functionality integrity, a couple of basic rules must be followed while developing new features for `Linux Enhanced Security`. This section will list them, divided by categories:

➔ Indentation Rules

- Code indentation should be similar to the used in the `Kernel`.

➔ Variable/Constant Name Rules

- **Call Identifiers**
  - A call is an identifier for a handler function. For instance, "`LESEC_CHPOWN_CALL`" is a reference to invoke the handler `chpown_call()` system call.
  - Call identifier names are always prefixed by the "`LESEC_`" word and suffixed by the "`_CALL`" word. The middle word uses capitalized characters and it specifies a feature. For instance "`LESEC_CHPOWN_CALL`".
- **Op Identifiers**
  - An op is an identifier for the operation that handler should perform. For instance, "`CHPOWN_WRITE`".
  - Op identifier names are always prefixed by the feature's name and suffixed by the operation's name. For instance, "`CHPOWN_WRITE`".
  - Defined operations are always handled inside the correspondent call's handler function.
- Calls, Operations, Flags and similar items should be always declared with `enum`.

➔ Kernel-Land Developing Rules

- For bit operations always use the `Kernel bitops API`. This `API` is defined at "`asm/bitops.h`" `Linux` header.
- **SMP Support**
  - All features must be `SMP` safe. You can omit this if it's clearly unnecessary.
  - When developing `SMP` safe code, always use the `Kernel SMP API`.

➔ User-Land Developing Rules

- `User-space` tools must always interact with the `Linux Enhanced Security` features, using the `lesec()` system call.

➔ General Developing Rules

- **Standard Issues**
  - You should never break the standards. This can happen exceptionally and must always be well documented and should be always optional.
- **License Issues**
  - Each file should have a license Header. You can find this header in "`doc/`" directory and never forget to update the file name and directory at the top of the header.

1.6 TESTING LINUX ENHANCED SECURITY FEATURES

Currently and due the lack of implemented features, only a few tests have been provided within the `Linux Enhanced Security Testing Toolkit`. With these tests you can test your system against almost all implemented security features. In the future a broader set of tests will be available.

→ You can download the `Linux Enhanced Security Testing Toolkit` from our website at `http://www.lesecurity.org`.

2

Memory Protections

CHAPTER 2
MEMORY PROTECTIONS

2 CHAPTER 2: MEMORY PROTECTIONS

2.1 NON-EXECUTABLE MAPS

The `Non-Executable Maps` implementations attempt to emulate trough software, the behaviour of the execution bit in the `CPU's` pagination mechanism, in the architectures that don't support it. Depending on the operating system's segmentation design, the lack of the execution bit in the pagination mechanism, can lead a set of instructions to be executed over non-executable mapped memory regions. This happens because there is no physical way for the operation system to relate a non-executable map with a non-executable page, therefore the `CPU` won't be able to protect pages that are mapped has non-executable memory against code executions.

These implementations can significantly slow a system's performance if not taken seriously while in a developing stage. Normally these are very resource consuming, since they are always performing all sorts of checks, every time an application is interacting with certain memory regions.

Nowadays concerns, while being a services provider, are not only in data integrity and trust trough security implementations. There is also a major concern in the availability and accessibility of a service using Quality of Service. This last item is invariably related with a system's processing capacity, we can have an optimized link and a good traffic shaper, but what's all that good for if we're wasting our system's resources with other unrelated tasks?

We think that performance in security systems is very important and these should always try to minimize its impact, finding a good balance between them. Equally, we should never give away security for a highly performed system, we must be reasonable and choose an acceptable security/performance level. Mainly under this subject, the `Linux Enhanced Security` searches this balance and tries to offer reasonable solutions implementing different ideas for this problem.

2.1.1 Non-Executable Maps Implementations

There are many different techniques to inject and execute arbitrary code in a running process. If an attacker accomplishes to use one of these techniques to change a process's execution flow, he won't be able to change the system, if he isn't able to execute system calls.

While most implementations try to prevent any execution attempts in non-executable maps, this one has a little different approach. A process has the legitimacy to execute instructions in his address space, even if it his non-executable mapped memory region. This sounds a little controversy, but we'll see that interrupts can helps us to prevent specific code executions from happen.

While in `user-mode` the `Kernel` hasn't any possible way to verify what a process is doing, but when an interrupt occurs (`All-Interrupts Checking behaviour`) or a system call (`System Call Checking behaviour`) is executed a context switch happens and the possibility to evaluate the process's condition before the system call execution is gained. Already in `kernel-mode` the process is checked and if the `CPU's eip` is over a non-executable map region, he's forced to terminate. Disabling the possibility to execute system calls under these conditions prevent almost all sorts of attacks by denying any system privilege and resource requests.

These checks can be triggered differently using two available behaviours: `All-Interrupts Checking behaviour` and `System Call Checking behaviour.`

While using the `All-Interrupts Checking behaviour` a non-executable map is executable until an interrupt occurs. This means that every time an interrupt occurs, the `Kernel` verifies if the process has the acceptable conditions to continue its execution. This behaviour is purely academic and has some disadvantages:

- High resource consuming implementation, since interrupts are always happening.
- Can terminate legit code execution, for instance, `GCC trampolines`.

While using the `System Call Checking behaviour` a non-executable map is executable until a system call is executed. This means that only when a context switch is triggered by a system call, the `Kernel` verifies if the process has the acceptable conditions to continue its execution. This is the most advisable behaviour since it has some valuable advantages:

- Very low resource consuming.
- Allow legit code execution, for instance, `GCC trampoline` compatibility.

Anyway, these are not full proof solutions and there are a few situations where they fail to accomplish security:

- In the `System Call Checking behaviour` it's possible to create a loop that will starve the `CPU's` resources. But this situation is also valid in a local environment, where users are allowed to execute their own code. So, this is a job for a resources limit tool and not a non-executable map implementation.

- Under certain specific circumstances, it's possible to avoid (→ see `section 2.1.3.8`) both behaviours. But, it's also easier to implement a legit return into `libc` attack. This can be prevented using a `GCC Stackguard` like patch.

As we've said, covering these issues here, would lead to a very slow implementation and since there are alternative solutions that can be used to complement it without harming the system's performance, we've decided to implement it this way.

→ A `Randomized Stack` protection may difficult issues like these from happening, please see `section 2.2`.

→ A `StackGuard` like protection can successfully prevent `stack-smash` attacks. Please see in `section 2.1.3.2` why these mechanisms can be also a good security solution in complement with `kernel-side` protections.

If you still don't know which behaviour you should choose for your system please see section 2.1.3 for more information.

Related Sections

2.1.3.2	The GCC Executable Stacks and StackGuard
2.1.3.8	Avoiding the Non-Executable Maps Protection
2.2	Randomized Stack

2.1.1.1 i386 Compatible Processor Behaviours

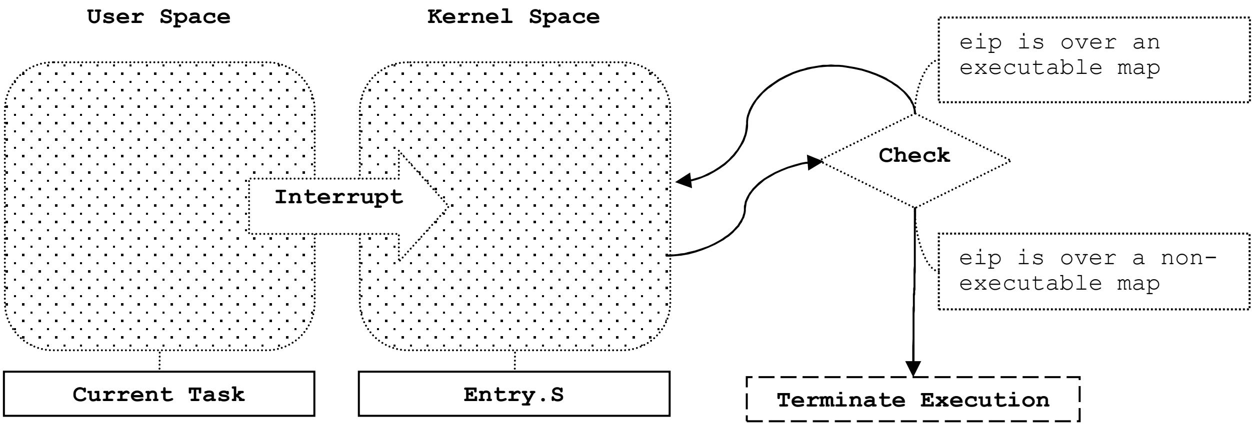
In `section 2.1.1` we've overviewed how these behaviours worked. In this section we'll try to go further in explanations being a little more technical.

These behaviours end up being very simple, but let us explain first how the `Kernel` handles interrupts.

There are a few interrupts that forces a context switch from user-space to `kernel-space`, for instance, a time interrupt triggered by the `real time clock` or a `task-switch` interrupt triggered by a system call. Whenever one of these interrupts occurs the `Kernel` starts executing the `entry.S` code. The `entry.S` code is a `Kernel` section that has various handlers to redirect the `Kernel's` execution flow to a specific section, depending on the context that has switched into `kernel-space`.

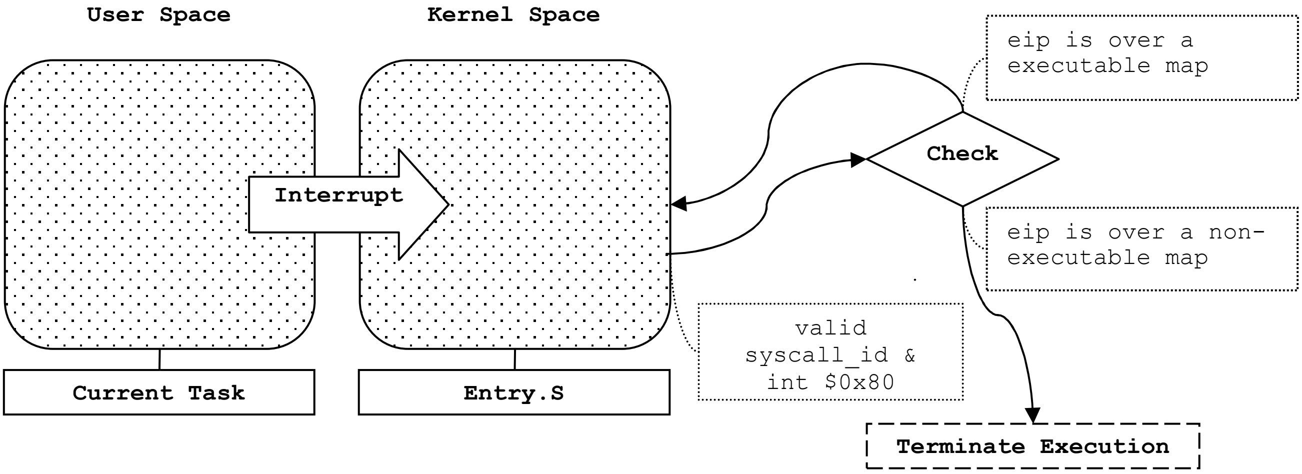
If we place code before the system_call checks in the `entry.S` code, it will be executed every time a context switch interrupt occurs; this is the `All-Interrupts Checking behaviour`.

Diagram for the All-Interrupts Checking behaviour



In the other hand, if we place code along with the system call checks, it will be executed only when a system call is executed; this is the `System Call Checking behaviour`.

Diagram for the System Call Checking behaviour



It’s always preformed the same check to validate the current task, either when an interrupt occurs, `All-Interrupts behaviour`, or either when a `task-switch` is triggered by system call, `System Call Checking behaviour`. Validating a process consists only in verifying which map the `CPU’s eip` is. Then, if the `“VM_EXEC”` flag on that map is unset, a `“SIGSEGV”` is sent to the current process forcing its termination.

Although it isn’t a full proof protection, it can still be very successful while stopping almost all attacks and has an incredible performance.

Related Sections

2.1.1 Non-Executable Maps Implementations

2.1.1.2 The Future

We’re not sure of the actual usefulness in performing these checks each time an interrupt occurs with the `All-Interrupts Checking behaviour`. The time that a process can take in a tick depends on the processor. Execution times on a `80386` processor are smaller than on a `PIV` processor. A `PIV` processor is able to execute much more instructions per tick, making this behaviour faster but less interactive with the routine. In the other hand a `80386` processor will interact much more times with the routine, slowing down the scheduling process.

While a final version hasn’t been release, we’ll decide what we’ll do with this protection. Probably we’ll only trig on system calls, leaving the `All-Interrupts Checking behaviour` behind.

Aside from our decision, we maintain both behaviours for you to choose when configuring the `Kernel`.

2.1.2 The modify_ldt() Compatibility (i386 only)

Some applications might need some control over the memory segmentation of their process space, this is common between operating systems emulators that need to reproduce that specific system’s segmentation design. The `Linux Kernel` allow the applications to define new segments trough `modify_ldt()` system call.

Since `Non-Executable Maps` would interfere with these applications, a `modify_ldt()` compatibility was also implemented. With this compatibility, the applications running under these conditions will still be able to have their maps verified without having problems.

2.1.2.1 How Does modify_ldt() Compatibility Works

When a process executes `modify_ldt()` to define new segments, the `Kernel` will assign a new `Local Descriptor Table` (`LDT`) and obviously, it will no longer share its own with other processes.

According to IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture, Section 3, Page 19: *`"(...) The following default segment selections cannot be overridden: Instruction fetches must be made from the code segment. (...)"`*, so, when calculating the `eip` position on the `Linear Address Space`, we only need to take care with the `Code Segment Selector` (`CSS`) because we won't be able to call a far pointer using any of the other segment selectors (`SS, DS, ES, FS or GS`).

When a process performs a far call to execute instructions on a newly created segment, the `eip` won't receive a `Linear Address`, instead it will receive an `offset` value for that segment starting at address `zero`. While we're checking if the `eip` position is within map boundaries, we'll add the base address that's inside the `Segment Descriptor` defining the actual code segment. We can identify this `Segment Descriptor` by reading the actual `Code Segment` (`CS)` index that's used by the far pointer. This check is only performed if the current `CS` and the original `CS` defined by the `Kernel` under `Current Privilege Level` (`CPL) 3`, don't match up.

2.1.3 Warnings and Suggestions

Before starting to choose your `Linux Enhanced Security` options, you should be aware of some important details that must be taken seriously, jeopardizing your system's security if you do not do so.

2.1.3.1 Selecting A Non-Executable Maps Behaviour

As we said above in `section 2.1.1.2` the `All-Interrupts Checking behaviour` doesn't make sense on slower processors so if your machine has a low processing capacity you probably want to check maps only when a system call is executed. The only advantage in choosing the `All-Interrupts Checking behaviour` is that a process won't loop for long if there's arbitrary code forcing it to do so. Although if this happens, no other resources will be compromised except `CPU` time.

→ If you'd like to know more about the incompatibilities of the `All-Interrupts Checking behaviour`, please refer to `section 2.1.3.3`.

Related Sections

2.1.1.2	The Future
2.1.3.3	Trampolines Compatibility

2.1.3.2 The GCC Executable Stacks and StackGuard

Since the first appearance of the newly `AMD` processors with the executable bit support in the pagination mechanism that the `GNU Compiler Collection` (`GCC`) started to force executable stack maps in the early `3.3.x` releases. This decision was taken by the `GCC` developer team due the fact that certain code wasn't running in these processors anymore. This was happening because the enforcement of the non-executable pages in the processor's pagination mechanism was faulting the `GCC's` nested function handlers, often called `trampolines`.

Trampolines are small pieces of code generated on-the-fly that are placed on the process's stack map and then executed. If the stack map is non-executable, then a process that uses nested functions will simply fail its execution and most probably will get killed with a `Page Fault`.

This seems a little controversial, finally that we have an execution bit support in `x86` architectures that solves the non-executable maps problem, the `GCC` now introduces a technical solution that is not compatible and deprecates this support. However, the `GCC` developer team plans to support natively in future releases the `StackGuard` patch has a solution for this problem. Obviously this is a sloppy solution covering the real problem.

The `StackGuard` project is a `GCC` patch that prevents stack-smashing attacks. Placing a token (`canary`) before the `return address` it's possible to know if it has been modified, checking if the token has been also modified. Normally this token can be either a random, null or terminate value. This solution doesn't prevent against stack writings, it only prevents against execution flow changes, by manipulating the `return address` value.

The `StackGuard` protection doesn't deprecate the `Non-Executable Maps protection` since it doesn't prevent against heap attacks.

When using the `All-Interrupts Checking behaviour` only binaries compiled with `GCC` versions prior to `3.3.0` will be `stack-smash` safe and also `trampoline` incompatible (→ see `section 2.1.3.3`). When using the `System Call Checking behaviour` there will be a new option that forces a check in executable stack maps too.

Related Sections

2.1.3.3 Trampolines Compatibility

2.1.3.3 Trampolines Compatibility

As explained above (→ see `section 2.1.3.2`), `trampolines` are small pieces of code generated on-the-fly that are placed on the process's stack map and then executed. `Trampolines` only need to handle addressing values and execute a `call` instruction, excluding system call execution. Therefore, if we're only checking the `eip` value each time a system call is executed (`System Call Checking behaviour`), there won't be any `trampoline` incompatibilities. However, this doesn't happen when checking the `eip` value each time an interrupt occurs (`All-Interrupts Checking behaviour`). If an interrupt occurs while a `trampoline` is being executed, the `eip` will be over a non-executable map and the process will be forced to terminate. This last behaviour is not trampoline compatible, turning a process's execution unpredictable.

Related Sections

2.1.3.2 The GCC Executable Stacks and StackGuard

2.1.3.4 When Should modify_ldt() Compatibility Be Used

You should use this option if you're working with programs that depends the `modify_ldt()` system call to work properly. This should be the case of some emulators or programs that where designed to work on a specific architecture. If you're not one of these cases, unless you really need it for any other reason, you can leave this option disabled.

2.1.3.5 The AMD64 Architecture

According to AMD64 Architecture Programmer's Manual, Volume 2, System Programming, revision 3.09, Chapter 5, Page Translation and Protection, Page 174: `"`*`(...) The AMD64 architecture introduces a third form of protection that prevents software from attempting to execute data pages as instructions. (...)`*`"`. If you're using this processor you won't need to use `Non-Executable Maps protection`.

→ `Please be warned for the GCC` executable stack implementation in `section 2.1.3.2`.

Related Sections

2.1.3.2 The GCC Executable Stacks and StackGuard

2.1.3.6 The Intel Itanium 2 Architecture

This architecture contains a `Non-eXecutable` (`NX`) bit on page permissions which enables non-executable pages. If you're using this processor you won't need to use `Non-Executable Maps protection`.

→ `Please be warned for the GCC` executable stack implementation in `section 2.1.3.2`.

Related Sections

2.1.3.2 The GCC Executable Stacks and StackGuard

2.1.3.7 The Intel LaGrande Technology (LT)

This technology will be implemented on future `PIV` processors (as well has the `VanderPool Technology`). The `LT Technology` supports many new hardware security features including the `NX` bit on page permissions (→ see `section 2.1.3.6`). If you're using a processor with this technology you won't need to use `Non-Executable Maps protection`.

→ `Please be warned for the GCC` executable stack implementation in `section 2.1.3.2`.

Related Sections

2.1.3.2 The GCC Executable Stacks and StackGuard

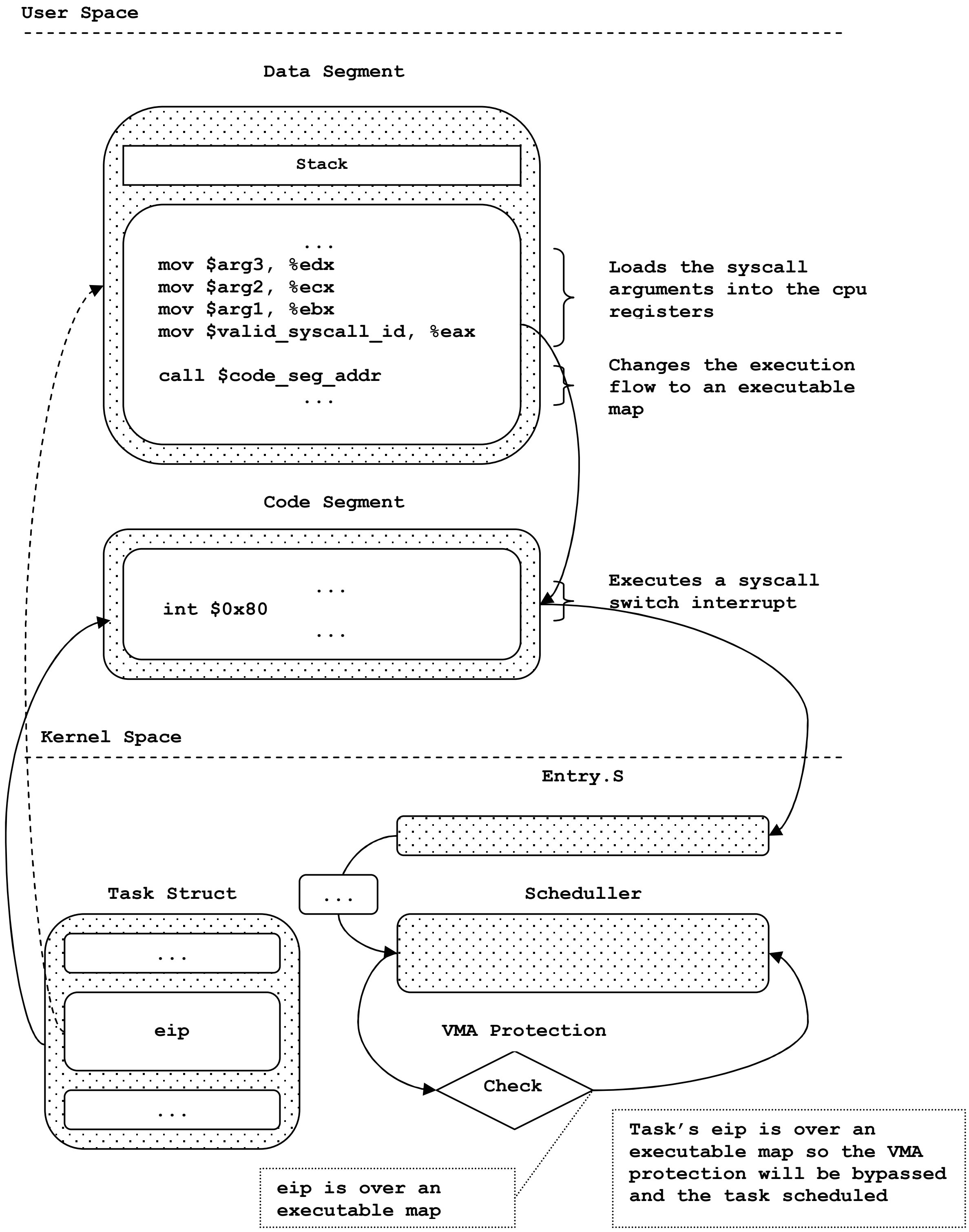
2.1.3.8 Avoiding the Non-Executable Maps Protection

The `Non-Executable Maps protection` performs only one check to see if the `eip` is over a non-executable map region. Under certain conditions it's possible to bypass it jumping to an executable map before the actual context switch happens.

A system call context switch happens whenever an interrupt `0x80` is executed. The `Kernel` will then load the specific system call arguments directly from the `CPU's` registers.

If we can control the process's stack we're able to execute instructions to load the `CPU's` registers with a specific system call's argument values. Then, if we perform a `far jump` to an interrupt `0x80` instruction already existent in a code map, the `Non-Executable Maps protection` will see the `eip` over a legit executable map allowing the process execution. It's very probable to find system call interrupts in a code map since a process can't do much without system calls and these can be often found within the `libc` code. Actually this is most similar to `return into libc` attacks but with some disadvantages that difficult the whole process. Since you're performing a `far jump` into a read-only map, you won't be able to control the execution flow when the system call returns, almost certainly leading to a process crash. Although we're limited to only one system call, if the process has *`real uid`* `0`, executing the `execve()` system call is enough to compromise the system, but in most cases we'll need to `set*id()` first. Avoiding this protection can be simpler if we use a `return into libc` technique which doesn't have these disadvantages and was never meant to be covered by this protection.

Diagram for the Non-Executable Maps Bypass (reproduces a stack-smash attack)



As we've said in the beginning of this chapter (→ see `section 2.1`) we invest in solutions that offer a good balance between security and performance. This is a really fast implementation that gives not the best but a very acceptable security level, therefore we've decided to leave it this way instead of loosing performance with a more complex solution.

We're already working on a `GCC` based protection to complement this protection without loosing performance and provide a wider secure solution against these issues.

→ We also warn you for the use of a Randomized Stack protection altogether with this protection

Related Sections

2.1 Non-Executable Maps
2.2 Randomized Stack

2.2 RANDOMIZED STACK

Stack randomization techniques appeared has an effective solution against the well known `stack-smashing` attacks. Although it self doesn't serve as a full proof security replacement, its simplicity and effeteness made it a big trump in nowadays security schemes.

2.2.1 How Does Randomized Stack Works

Each time a binary is executed, multiple code and data maps are requested to the operating system. One of them is an expand-down data map, also known as stack, which will be placed at the top of the process's memory. Later, a random value is subtracted from the pointer that points to the top of the process's memory, this way selecting a random memory region. A different random region is selected between executions, statistically reducing the chances, closer to 0%, that a `stack-smash` attack has to be successful.

2.2.2 Randomized Stack and StackGuard

There are many implementations that prevent `stack-smash` attacks but all of them have their pros and cons. Sometimes we need to use more than one protection or choose one that best fits our system in order to increase effeteness in preventing these attacks.

For instance, when using the `StackGuard` with `GCC`, the use of `Randomized Stack protection` may be omitted but the `StackGuard` protection, in some cases, can be avoidable with some exploiting techniques that are based on a previous stack analysis to retrieve the `canary` value and craft it into the string which contains the `shellcode` and `return address` value. This kind of exploiting is generally used when using `random canaries`, because these are generated with a random value `XORed` along with the `return address` of the current stack frame.

If security is really important on your system, then you should use `StackGuard` and `Randomized Stack protection`.

2.2.3 The Future

With the actual evolution of compile time security enhancements and processor protections, this feature may become deprecated soon as well as the `Non-Executable Maps protection`. But for now, this enhancement it's justifiable.

2.3 VMA PROTECTIONS

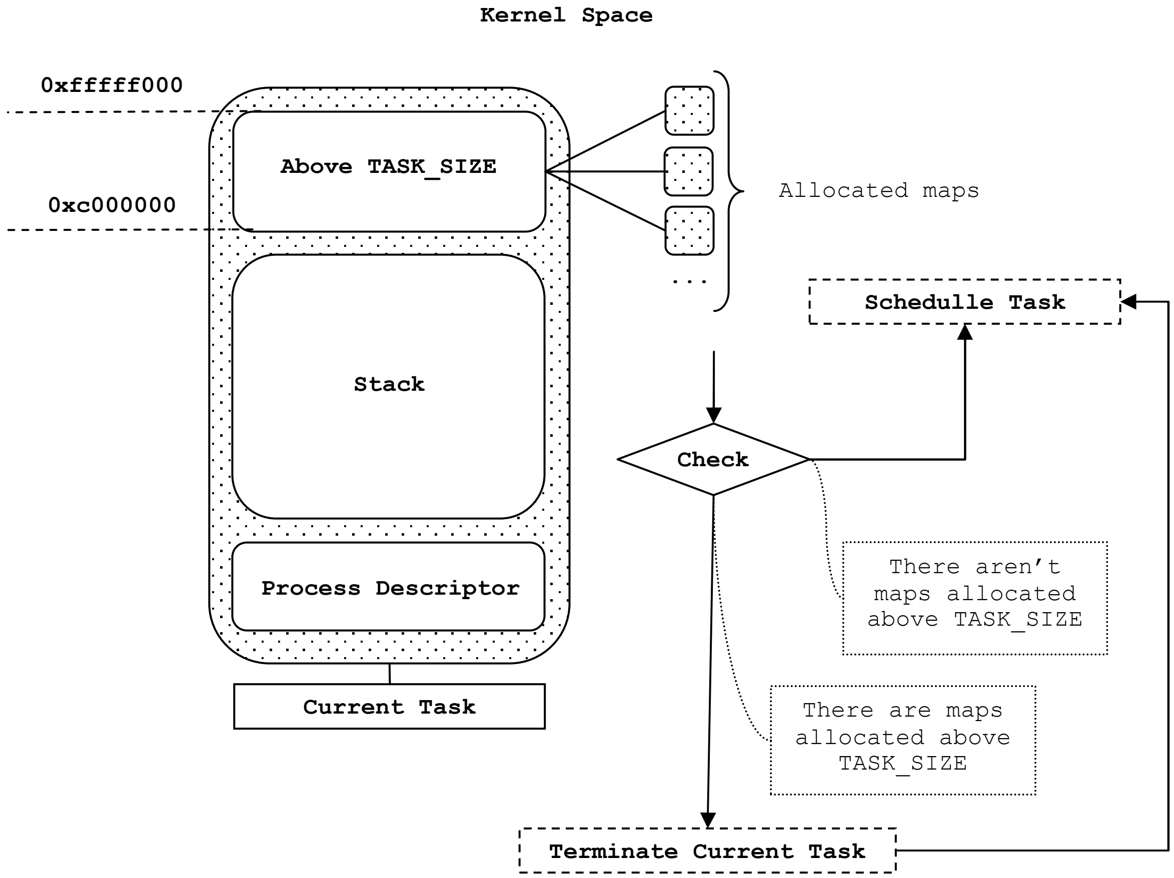
Lately we've been assisting the uncovering of multiple flaws in the `Linux Kernel` that could lead into a locally compromised system. Most of these flaws were in boundary checks preformed on values passed to system calls. Good examples of this flaws appeared in `munmap()`, `mremap()` and `brk()` system calls that allowed an `user-space` process to map `Kernel` memory as a consequence. Once this memory was mapped in `user-space`, the only thing left to do was to change specific values in specific `Kernel` structures, the trickiest part, but how this was accomplished is another story.

2.3.1 How Does VMA Protections Works

Every task has a region in its address space that is reserved to `Kernel` data. This region is between `0xc0000000` and `0xffffffff`, therefore `Kernel` memory will always be mapped here. Once we already know that the address space reserved to the `Kernel` is above the `3GB`, we also know that the task's data must be under the `3GB`. The `Linux Kernel` has the "`TASK_SIZE`" macro that we can use to know exactly where the task's memory ends.

The main idea for this protection mechanism is to check, every time the `Kernel` is returning into `user-space` after a system call, if there is any memory mapped above the "`TASK_SIZE`" value. If this happens, we know that kernel memory is mapped, therefore a `SIGSEGV` is sent, forcing the task to terminate.

Diagram for the VMA Protections



This protection is trivial to implement in the Linux Kernel 2.4.x series, since there are no mapped regions above "TASK_SIZE" available to user-space. However, in the newest Linux Kernel 2.6.x series, every task has a memory region above "TASK_SIZE" mapped from 0xffffe000 to 0xfffff000 (at least on i386 architectures). This region is used by the Dynamic Shared Object (DSO) map (→ see section 2.3.2.4) and can be ignored while performing normal map checks without great impact on performance (→ see section 2.3.2.2). Since maps cannot be overlapped by other maps, it's safe to ignore these reserved mapped regions.

Related Sections

2.3.2.2 The Impact on Performance
2.3.2.4 Dynamic Shared Objects Map

2.3.2 Warnings and Suggestions

Before starting to choose your Linux Enhanced Security options, you should be aware of some important details that must be taken seriously, jeopardizing your system's security if you do not do so.

2.3.2.1 When Should VMA Protections Be Used

You may find this a little paranoid, however, security holes like those present in `munmap()`, `mremap()` and `brk()`, may still happen. We can never be too sure about the system calls safety therefore, if security is most important to your system, it's advisable to select this option.

2.3.2.2 The Impact on Performance

If you select this option, the impact on the performance of your system will be very low, since the algorithm used to perform the VMA checks is optimized with caching mechanisms that speeds up the entire process. The first time that VMA pools are verified, the stack map pointer is cached and since this map is always the last one before reaching "TASK_SIZE", future verifications use directly the cached pointer, ignoring all maps below.

2.3.2.3 The Persistent Kernel Map (PK Map)

The Persistent Kernel Map (PK Map) is a memory pool that contains, for short periods of time, Page Table Entries (PTE) that are used to map High Memory Region pages into Normal Memory Region and vice-versa. This map behaves like a memory bouncer.

This memory region isn't new in Linux Kernel 2.6.x series and exists in older Kernel versions since High Memory Management support first appeared. The difference between older and current series is the size of this map that isn't constant anymore and has now a variable range between "PKMAP_BASE" and "FIXADDR_SIZE".

For the x86 compatible architectures, when the number of CPUs is less than or equal to 32 units, the "PKMAP_BASE" constant holds the 0xff800000 value and the "FIXADDR_SIZE" is a compile time defined constant. This constant value depends on the Kernel configuration, therefore we can only say that PKMap begins on "PKMAP_BASE" and ends on "FIXADDR_SIZE".

2.3.2.4 Dynamic Shared Objects Map (DSO Map)

The `Dynamic Shared Objects Map` (`DSO Map`) was first introduced in the recent `Linux Kernel 2.6.x` series and it's used to load an `ELF` binary containing, has its name says, `Dynamic Shared Objects`. These objects are used to speed up system calls, `sigtrampoline` and `sigreturn` purposes.

For the `x86` compatible architectures, the `DSOs` are called `Virtual System Calls` and for `IA-64` these are called `Fast System Calls`, because system call's virtualization isn't supported by this architecture.

> → See "`linux/Documentation/ia64/fsys.txt`" for more information about `Fast System Calls`.

2.4 DISABLED /DEV/MEM AND /DEV/KMEM

Nowadays, many backdoor systems are installed into the kernel space directly through "`/dev/mem`" or "`/dev/kmem`" devices even if the `Kernel` hasn't compiled the module support. The only way to prevent this kind of backdoors is preventing those devices from being opened.

2.4.1 How Does Disable /dev/mem and /dev/kmem Work

These are character devices that are handled by special routines called, device operations. There are many operations available to character devices, but the most common amongst them are `open()`, `write()`, `read()` and `close()`. Disabling the `open()` operation for these devices will leave them inaccessible and any `open()` attempt on the device will return an "`EPERM`".

2.4.2 Conclusion

Since it's not possible to open these devices, there's no way to install backdoor code into the `Kernel` space. Although, as a side effect, loading `Kernel` modules will be impossible, neither running `Klog` nor `X Server`.

Related Sections	
2.4.3.1	Incompatibility with Loadable Kernel Modules (LKMs)
2.4.3.2	Incompatibility with Kernel Logger Daemon (klogd)
2.4.3.3	Incompatiblity with X Servers

2.4.3 Warnings and Suggestions

Before starting to choose your Linux Enhanced Security options, you should be aware of some important details that must be taken seriously, jeopardizing your system's security if you do not do so.

2.4.3.1 Incompatibility with Loadable Kernel Modules (LKMs)

The `Loadable Kernel Modules` (`LKMs`) are loaded trough "`/dev/kmem`" using a set of `user-land` tools called `modutils`. If this device is disabled, there's no way to load a module.

2.4.3.2 Incompatibility with Kernel Logger Daemon (klogd)

The `Kernel Logger Daemon` is used to log events generated by the `Kernel` and depends `"/dev/kmem"` to work properly. Therefore, if this device is disabled, `klogd` will fail its initialization.

2.4.3.3 Incompatibility with X Servers

Some `X Servers` like `XOrg` and `XFree86`, use the `"/dev/kmem"` to access directly to the `Kernel` memory. If this device is disabled then `X Servers` like these won't be able to run.

2.4.3.4 How Does Backdoors Works

There are multiple ways to load backdoor code into `Kernel` space, but they will always need to open `"/dev/mem"` or `"/dev/kmem"` to access the `Kernel` memory. This happens because an attacker needs to know the exact location of some important `Kernel` pointers in order to change and point them to the backdoor code. Loading code into `Kernel` space can be a simple process when you have `modutils`, but very painful when these aren’t supported, since writing portable ways to load it in different systems is always a difficult to accomplish. Without these devices, such thing isn’t possible anymore.

Process Protections

3

CHAPTER 3
PROCESS PROTECTIONS

3 CHAPTER 3: PROCESS PROTECTIONS

3.1 RANDOMIZED PIDS

There are flaws that can be exploited by guessing the `pid` value of a process that hasn't been yet launched. This type of attack is based on the sequential `pid` attribution. The `pid` randomization comes has a solution for this problem.

3.1.1 How Does Randomized PIDs Works

When a new process is created, the `Kernel` attributes a unique `pid` that will distinct it from all the others. Normally the `pid` value is attributed adding `1` to the previous attributed `pid`, but when randomization is enabled this will be randomly generated value between `0x300` and `0x7fff`. Case happens to be generated an already attributed `pid` then the algorithm will enter a loop, adding `1` to the randomly generated `pid` until a free one is found.

3.1.2 Conclusion

If you use `pid` randomization together with `Proc File System Protections` (→ see `section 4.1`), will be almost impossible to retrieve the `pid` of a process.

Related Sections

4.1 Proc File System Protections

3.2 HIDDEN MAPS

There are attacks that need to consult "`/proc/<pid>/maps`" to access a task's map information and locate pointers references needed to successfully exploit an existent flaw. Since this file is only used information/debugging issues and the current task doesn't depend from it, it's safe to omit all map information in it.

3.2.1 How Does Hidden Maps Works

Every time a read operation is called for this file, the `Proc` file system handlers are modified in such way that instead of returning real `VMA` pointer information, each map will have a null pointer has a reference.

3.2.2 Conclusion

Placing null pointers in each map reference, there's no way for an attacker to know the process's memory map regions using "`/proc/<pid>/maps`".

File System Protections

4

CHAPTER 4
FILE SYSTEM PROTECTIONS

4 CHAPTER 4: FILE SYSTEM PROTECTIONS

4.1 PROC FILE SYSTEM PROTECTIONS

The `Proc` file system gathers various files with constantly updated system and process information. In systems that have hostile local environments, for instance shell providers, it may be useful to deny or restrict access to this information. The `Proc File System Protections` enable you to select different access restrictions to system and process information trough the `Proc` file system.

> → See `Appendix A` for a complete reference list of options and files where these restrictions are applied.

4.1.1 How Does Proc File System Protections Works

For system files that lay at the `Proc's` root directory the only option available is enable or disable and for the process's information you can select between `user` and `group level` restrictions.

> → See `Appendix A` for a complete reference of the modified permissions and disabled files.

4.1.2 Conclusion

There is certain information that isn't supposed to be seen by users on a system. Occulting important information may difficult the disclosure or even exploit process of a certain security flaw.

Has an alternative to the system files protection you can also change their permission using `chmod()` to restrict access to the system users. However there are some files that shouldn't be seen, not only by users, but even by `super-user`. For instance, `kallsyms` and `kcore` files could be used by a successful attacker to retrieve sensitive information as memory offsets and user passwords respectively. Therefore, we advise you to disable of these files.

5

Administration Tools

CHAPTER 5
ADMINISTRATION TOOLS

5 CHAPTER 5: ADMINISTRATION TOOLS

5.1 CHANGE PROCESS OWNER (CHPOWN)

Sometimes there are application daemons that only need certain higher privileges while they're starting up. For instance, if you're binding `Apache` into privileged service ports (`<1024`), you'll need to run it with `super-user` privileges, otherwise the `bind()` system call will return an "`EPERM`". However after this initialization, it's very possible that `super-user` privileges won't be needed anymore, and if they are, you can easily create an environment where they won't.

You may think that `Apache` isn't a very good example because, if it's well configured, its children processes, which actually process the user input data, are running with `local-user` privileges and at most compromising a local user account. Well, that's not a wised thought, since history tells us that many shared memory flaws allowed, what appeared a `local-user` compromise, to be a `super-user` compromise, executing code in a shared memory space with `super-user` privileges. Like `Apache`, many other applications will have this sort of security flaws.

As we've seen, this can be a security problem and we should never trust the application's privilege separation mechanism, in the worst case scenario this should be always guaranteed by the operating system.

> → As a solution for these issues, `chpown` enables you to change on-the-fly a process real user and group.

5.1.1 How Does Chpown Works

Whenever `chpown` is executed to change a process owner, it will interact with `kernel-space`, trough the `lesec()` system call, and update the process's task structure fields; `suid`, `fsuid`, `rgid`, `egid`, `sgid` and `fsgid`, to the requested owner privileges. If `chpown` has requested changes to an invalid `pid` value, the `lesec()` system call returns "`EINVAL`".

> → See the `Appendix B` for the `chpown` manual.

5.1.2 Conclusion

Having services running with `local-user` privileges reduces the chances of a compromised system and at most you'll have a compromised service.

Sometimes an attacker would use signals to kill a service, for instance if an apache process child is running with the same parent `uid`, he will be able to kill it. However, `Signal Protections` (`sigp`) should work for theses cases (→ see `section 5.2`).

Related Sections

5.2 Signal Protection (SIGP)

5.2 SIGNAL PROTECTION (SIGP)

If an attacker can successfully exploit a flaw present in a child process and if it's running with the same parent privileges, he's able to send signals to the parent process. For instance, he could use this feature to send a kill signal and force the parent process to terminate execution. With `Signal Protection` you can deny certain signals from being delivered to a given process, even the kill signal.

5.2.1 How Does Sigp Works

Within the `Kernel` each task is discriminated by a task structure. Each task structure has a special `32` bit mask that identifies at most `32` inhibit signals. When `sigp` is executed, it will interact with `kernel-space`, trough the `lesec()` system call, and mask set/unset the correspondent bit. Whenever the `Kernel` delivers a signal to a process it will then check its bit mask first and if the correspondent bit is cleared, the signal is delivered, otherwise it is discarded.

→ See the `Appendix C` for the `chpown` manual.

5.2.2 Conclusion

Inhibiting certain signals may difficult attacks that depends this feature to work properly, fortifying your services availability and reducing the possibilities of successful `Denial of Service` attacks only to client instances.

Audit Options

6

CHAPTER 6
AUDIT OPTIONS

6 CHAPTER 6: AUDIT OPTIONS

6.1 LOG LINUX ENHANCED SECURITY KERNEL EVENTS

The `Audit Options` goal is to log system’s relevant information. This feature isn’t yet developed and presently you can only log certain features. In future releases this option should be vastly explored in order to offer a power set of system crucial information.

Proc File System Restricted Files

A

APPENDIX A
PROC FILESYSTEM RESTRICTED FILES

APPENDIX A: PROC FILESYSTEM RESTRICTED FILES

OPTIONS

Restriction options for accessing "`/proc/pid/`" data:

Option	Comment
`LESEC_PROC_FS_PROT_OPT_USR`	Restrict access on a user basis
`LESEC_PROC_FS_PROT_OPT_GRP`	Restrict access on a group basis

Default option is "`LESEC_PROC_FS_PROT_OPT_USR`".

CONFIGURATION OPTIONS

Directory "`/proc/<pid>`" restriction modes for "`LESEC_PROC_FS_PROT_OPT_PID`" option:

Mode	Option
`S_IFDIR|S_IRUSR|S_IXUSR`	`LESEC_PROC_FS_PROT_OPT_USR`
`S_IFDIR|S_IRUSR|S_IXUSR|S_IRGRP|S_IXGRP`	`LESEC_PROC_FS_PROT_OPT_GRP`

Options to disable correspondent "`/proc`" files:

Option	File
`LESEC_PROC_FS_PROT_MEMINFO`	`/proc/meminfo`
`LESEC_PROC_FS_PROT_CPUINFO`	`/proc/cpuinfo`
`LESEC_PROC_FS_PROT_HW`	`/proc/hardware`
`LESEC_PROC_FS_PROT_STRAM`	`/proc/stram`
`LESEC_PROC_FS_PROT_DEV`	`/proc/devices`
`LESEC_PROC_FS_PROT_FS`	`/proc/filesystems`
`LESEC_PROC_FS_PROT_CMDLINE`	`/proc/cmdline`
`LESEC_PROC_FS_PROT_LOCKS`	`/proc/locks`
`LESEC_PROC_FS_PROT_XDOM`	`/proc/execdomains`
`LESEC_PROC_FS_PROT_PART`	`/proc/partitions`
`LESEC_PROC_FS_PROT_STAT`	`/proc/stat`
`LESEC_PROC_FS_PROT_DISKSTAT`	`/proc/diskstats`
`LESEC_PROC_FS_PROT_INT`	`/proc/interrupts`
`LESEC_PROC_FS_PROT_MODULES`	`/proc/modules`
`LESEC_PROC_FS_PROT_SSTAT`	`/proc/schedstat`
`LESEC_PROC_FS_PROT_VMSTAT`	`/proc/vmstat`
`LESEC_PROC_FS_PROT_BUDINFO`	`/proc/buddyinfo`
`LESEC_PROC_FS_PROT_KCORE`	`/proc/kcore`
`LESEC_PROC_FS_PROT_KASYMS`	`/proc/kallsyms`

Administration Tools: Chpown

B

APPENDIX B
ADMINISTRATION TOOLS: CHPOWN

APPENDIX B: CHPOWN

Usage

```
chpown <user>[:<group>] <pid>
```

Description

Change the user and/or group ownership for a given process.

Options

Argument	Description
user	The new user-name or the uid value that will be set for the process.
group	The new group-name or gid value that will be set for the process. This argument is optional.
pid	The process id value.

Example

```
# chpown apache:apache 1234
```

This will modify the process's user/group, identified by 2321, to user and group `apache`.

PROTOCOL SPECIFICATION

Call identifier

LESEC_CHPOWN_CALL

Operation identifiers

CHPOWN_WRITE

Data specification

Value	Size (bits)	Description
uid	32	Specifies the uid value
gid	32	Specifies the gid value
pid	16	Specifies the pid value

Data alignment of 64 bits

Administration Tools: Sigp

C

APPENDIX C
ADMINISTRATION TOOLS: SIGP

APPENDIX C: SIGP

Usage

```
sigp <option> [args]
```

Description

Inhibit certain signals in a process.

Options

Arguments	**Description**
`-s <pid> +-signal [... +-signal]`	Set/unset a list of inhibited signals in a process identified by the argument pid. To set you must concatenate the character '+' and to unset the character '-'.
`-p <pid>`	Prints the list for inhibited signals for the process identified by pid.
`-l`	Prints the list of valid signals.
`-h`	Prints the help output.

Example

```
# sigp -s 1234 +SIGKILL +SIGSEGV -SIGTERM
```

This will set inhibited signals "`SIGKILL`", "`SIGSEGV`" and unset "`SIGTERM`" for the process with `pid 1234`.

PROTOCOL SPECIFICATION

Call identifier

```
LESEC_SIGP_CALL
```

Operation identifiers

```
SIGP_WRITE
SIGP_READ
```

Data specification

Value	Size (bits)	Description
isig_set	32	Set signals mask
isig_unset	32	unset signals mask

Bibliography

BIBLIOGRAPHY

BIBLIOGRAPHY

Bibliographic references that where used to support this manual a listed here.

I. AMD64 ARCHITECTURE: PROGRAMMERS MANUALS

`http://www.amd.com/us-en/Processors/TechnicalResources/0,,30_182_739_7203,00.html`

II. INTEL ARCHITECTURE: SOFTWARE DEVELOPERS MANUALS

`http://www.intel.com/design/Pentium4/documentation.htm?iid=ipp_dlc_procp4f+tech_doc&#manuals`

III. INTEL EXTENDED MEMORY 64 TECHNOLOGY: SOFTWARE DEVELOPER'S

http://www.intel.com/design/pentium4/manuals/index_new.htm#em64_doc_ch

IV. INTEL LAGRAND AND VANDERPOOL TECHNOLOGY

http://www.intel.com/pressroom/archive/releases/20040218corp.htm

V. GNU COMPILER COLLECTION (GCC) INTERNALS

http://gcc.gnu.org/onlinegocs/gccint/

VI. IMMUNUX STACKGUARD MECHANISM: STACK INTEGRITY CHECKING

http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/mechanism.html

Credits

CREDITS

CREDITS

Has you can imagine, this project demands a lot of work and writing good documentation it’s not an easy task. Therefore all contributions have proven themselves to be a very important issue while writing this manual.

We’d like to dedicate this manual section to the people that have contributed significantly with text and corrections. Here is a list of them:

- João Santos
- Bruno Vieira
- Luis Pedrosa