Department of Computer Science
Aalborg University
Fredrik Bajersvej 7E
DK–9220 Aalborg Øst
Denmark

Umbrella

We can't prevent the rain . . .

– But we don't get wet!

Master's Thesis
Aalborg University
June 2004

Group members:	Søren Nøhr Christensen	snc@cs.aau.dk
	Kristian Sørensen	ks@cs.aau.dk
	Michel Thrysøe	mthrysoe@cs.aau.dk

Faculty of Engineering and Science
Aalborg University

Department of Computer Science

TITLE:
Umbrella
We can't prevent the rain ...
– But we don't get wet!

PROJECT PERIOD:
DAT6,
February –
June 2004

PROJECT GROUP:
umbrella@cs.aau.dk
umbrella.sourceforge.net

GROUP MEMBERS:
Søren Nøhr Christensen
Kristian Sørensen
Michel Thrysøe

SUPERVISOR:
Emmanuel Fleury

NUMBER OF COPIES: 12

REPORT PAGES: 76

APPENDIX PAGES: 35

TOTAL PAGES: 127

SYNOPSIS:

This master's thesis describes the *Umbrella* security mechanism for Linux on handhelds. Umbrella implements a combination of process based mandatory access control and authentication of files.

Umbrella is implemented on top of the Linux Security Modules framework in Linux kernel 2.6. A HP iPAQ PDA has been used for implementation and testing purposes.

The mandatory access control scheme is enforced at process level, by a set of restrictions for each process, where every process has at least the restrictions of its parent. When a process spawns a new child process, it is possible for the programmer to specify a more restrictive context for this child. Thus, it is possible for the programmer to enforce the principle of *least privilege* for possibly dangerous child processes.

Vendors provides signed executables by means of public key cryptography. The signature consists of a set of restrictions to be set on time of execution and a hash value of the executable. The latter enables Umbrella to check if the file has been altered.

The process based MAC part of Umbrella have been successfully implemented, and file system relevant implementation is pending work. Furthermore, Umbrella have been benchmarked for performance and methods for verifying LSM have been investigated.

Authors

Søren Nøhr Christensen

Kristian Sørensen

Michel Thrysøe

Contents

1 Introduction

This report describes the a security mechanism for Linux on handhelds, implementing a combination of process based mandatory access control and authentication of files. This mechanism is named Umbrella. The idea for Umbrella emerged when investigating ways to enforce security on handheld devices. The Umbrella project is developed as our master's thesis at Department of Computer Science, Aalborg University, Denmark.

1.1 Linux on Handhelds

As the Umbrella project started it was aimed at developing a security mechanism for the Symbian OS[1], since this is the leading operating system for handhelds [18]. It was not possible to gain access to the Symbian source code, which is why Linux was chosen instead. After working with the Umbrella project for a period of nine months, the Linux approach is becoming increasingly more interesting as Motorola, Samsung, Panasonic and others have released smart phones based on Linux [6, 3]. Linux for embedded systems is in rapid development along with support for the ARM architecture which is used in many handheld devices. An example is HP, who is currently supporting this development.

There are several reasons for using Linux on handhelds, including reduced time to market [4, 33], openness and the fact that it is free. Umbrella provides the feature of protecting the handheld, including its devices and files against various attacks.

1.2 What is Umbrella?

Handheld devices are in rapid and constant development and they perform an increasing amount of tasks in everyday life. The increasing popularity and wide spread use introduces more and more threats to handhelds, which raises a demand for security measures within the operating system. Umbrella for handhelds implements a combination of process based mandatory access control and authentication of files for Linux based on the Linux Security Modules framework (LSM). The mandatory access control (MAC) scheme is enforced by sets of restrictions on individual processes.

[1]http://www.symbian.com

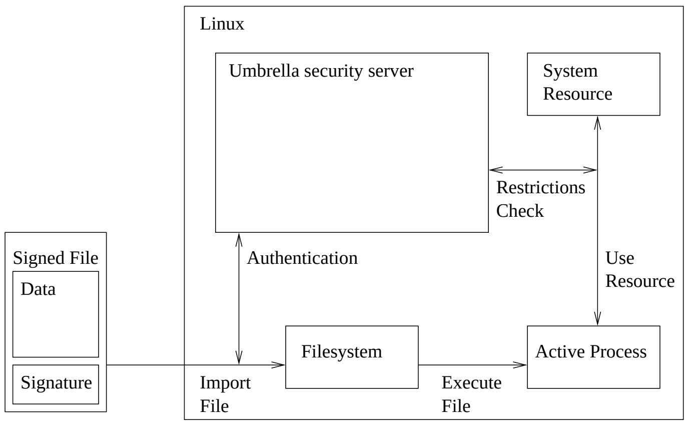

Figure 1.1: The Umbrella idea.

1.2.1 Security Improvements Provided by Umbrella

A main area for improvement is the access control measures of the operating system. The standard discretionary access control (DAC) mechanism can be supplemented by introducing the concept of mandatory access control. By enforcing MAC on processes as opposed to subject and objects, two advantages emerge. First, it is possible to view the access control structure as a tree, where children have at least all the restrictions of its parent. Secondly, this avoids the need for manual setting of restrictions for all programs in the system. The security mechanism deals with the problem of malicious software, using e.g. the principle of least privilege. This limits the harm of process hijacking.

Another area for improvement is the integrity of executables. This must be improved by introducing vendor-signing of executable files. A hash value is used to ensure that the file has not been tampered with. The set of restrictions that the program must run with, is included to ease the introduction of restrictions on the system. This is important because the system should be transparent to the user; user interaction regarding restrictions and other security questions is very unwanted on a handheld.

The basics of the above ideas for improvement is illustrated in Figure 1.1. When a file is imported to the system, the security server is asked for authentication. If the authentication is positive, the file is added to file system and given the set of restrictions which was included within the signature. If the authentication is negative, a predefined set of restrictions is given. When an active process tries to access a system resource, the security server checks if the process is restricted from this. If so, the process is denied access to the resource.

Implementing Umbrella in the operating system of a handheld device makes it possible to control and restrict access to resources on the device. By adapting existing programs, it is possible to restrict processes to least privilege, thereby limiting damage done by e.g. malicious email attachments as well as preventing viruses from spreading.

1.3 Threats to Handhelds

The two main security threats against handheld devices are imposed through the exploit of errors in existing software and through software with malicious content, such as a virus attached to an email. Both threats are elaborated below, along with a introduction to how Umbrella seeks to counter threats of these types.

1.3.1 Errors in Existing Software

Errors in software can be exploited to gain unauthorized access to a system or crash processes. Different classes of attacks exists and two are briefly described below, namely buffer overflow attacks and format string attacks.

Buffer Overflow Attacks

Buffer overflows are one of the most common exploits today; the international Computer Emergency Response Team[2] have sent out more than a hundred security warnings concerning buffer overflows within the last year. A short introduction is given here, but further details can be found in [36] and [41].

A buffer overflow occurs when a process tries to store more data in a buffer, than it is intended to hold and no bounds checking is done. Since buffers are created to contain a finite amount of data, the extra information can overflow into adjacent buffers, thereby corrupting or overwriting valid data held in them.

In buffer overflow attacks, the extra data may contain code designed to trigger specific actions, in effect sending new instructions to the attacked computer. This could be used to crash or hijack a process. A hijacked process can be used to gain further access to the system, as it has the rights of the user that started the original process. From this it is obvious to see that hijacking of a process owned by root, is very dangerous to the system. Crashing a process via a buffer overflow is also a potential attack which can be used to bring down the system, i.e. a denial of service attack. Providing a solution for this type of attacks is difficult, since they can occur in any process at any time, and new exploits are discovered continuously.

Format String Attacks

Format string attacks is another way of tampering with the contents of the stack, done by exploiting unchecked format string input [35].

Format string bugs come from the same dark corner as many other security holes; the laziness of programmer's. The concept is best explained using an example. A programmers task is to print out a string or copy it to some buffer. What he means to write is something like:

```
printf("%s", str);
```

[2]http://www.cert.org

Instead he decides that he can save time, effort and 6 bytes of source code by writing

```
printf(str);
```

Why bother with the extra `printf` argument and the time it takes to parse through that format? The first argument to `printf` is a string to be printed anyway! Because the programmer has unknowingly opened a security hole that allows an attacker to control the execution of the program.

What did the programmer do that was so wrong? He passed in a string that he wanted printed verbatim. Instead, the string is interpreted by the `printf` function as a format string. It is scanned for special format characters such as `%d`. As formats are encountered, a variable number of argument values are retrieved from the stack. From this it is obvious that an attacker can peek into the memory of the program by printing out these values stored on the stack. What may not be as obvious is that this simple mistake can give away enough control to allow an arbitrary value to be written into the memory of the running program. This involves the use of other format characters and a detailed explanation can be found in [35].

The approach taken by Umbrella ensures that it is not important how an attack is performed, since Umbrella does not attempt to prevent the process hijacking itself, but instead seeks to limit the potential damage done by a hijacked process. The reason for this decision is that the number and variety of new attacks makes it virtually impossible to counter all of them.

1.3.2 Malicious Software

Another threat to handhelds that is becoming increasingly relevant, is that of malicious software, such as viruses, worms, etc. [17, 1, 20, 19]. With the increasing use of email on handhelds and the fact that handheld devices include several communication features, this opens for various attacks through these. Another way of getting malicious code executed on a system, is to hide it in software with a legitimate purpose.

The attacks that can be imposed through execution of malicious code range over a variety of attacks, such as a virus that sends it self to all contacts in the address book or a Trojan horse that installs a back door on the system.

The threat of malicious code executed through email can be countered by limiting the rights of an attachment, executed from the email client. The threat from malicious code concealed in legitimate programs can be countered by introducing an authentication of executable files. Files from untrusted sources should be executed in a very restricted environment.

1.4 Report Overview

Chapter 2 contains a description of a selection of existing security projects, which have been investigated as inspiration for Umbrella.

Chapter 3 contains the design document for Umbrella, including design of the process based MAC along with the design of the vendor-signed files.

Chapter 4 presents implementation details along with issues regarding possible optimizations of Umbrella.

Chapter 5 includes preliminary benchmarks of Umbrella and the Umbrella API including an example. Finally suggestions of how to circumvent the security provided by Umbrella is presented along with two real life examples.

Chapter 6 documents verification of Umbrella, which consists of verifying the Linux Security Modules framework. An exploit in LSM has been investigated and discussed.

Chapter 7 concludes the project.

Appendices A and B contains information regarding practical issues involved in installing and using Linux on the iPAQ, as well as a description of Linux distributions for handhelds.

Appendix C explains the Linux Security Modules framework which is the base of Umbrella.

Appendix D lists all LSM security hooks.

Appendix E presents the roadmap for the Umbrella implementation.

2 Analysis of Existing Security Projects

This chapter covers a selection of security projects involving some sort of MAC mechanism for Linux. Several projects have been investigated to gain an overview of the field and seek inspiration and ideas for the design of Umbrella.

2.1 Mandatory Access Control Principles

In general terms, mandatory security policy represents any security policy that is defined strictly by system security policy administrator along with any policy attributes associated. MAC policy specifies how certain subjects and objects can access operating system objects and services. There are two fundamental implications of the MAC approach [7].

- Users cannot manipulate access control attributes of the objects they own at their own discretion.
- Privileges associated with a process are determined by appropriate MAC mechanisms, based on relevant mandatory security policy settings, on a per task basis.

The general access control model, that many MAC implementations are based on, describes the system's state using three entities (S, O, M), where S is a set of subjects, O is a set of objects and M is an access matrix which has one row for each subject and one column for each object. The cell $M[S_2, O_1]$ contains the set, a, of access rights that subject S_2 has for object O_1. These access rights are taken from a finite set A, that could be e.g $\{\textit{read}, \textit{write}\}$. This means that subject S_2 can perform a *write* operation on object O_1 if and only if $\textit{write} \in M[S_2, O_1]$. Figure 2.1 depicts this example and the general approach to mandatory access control.

A presentation of several security project follows.

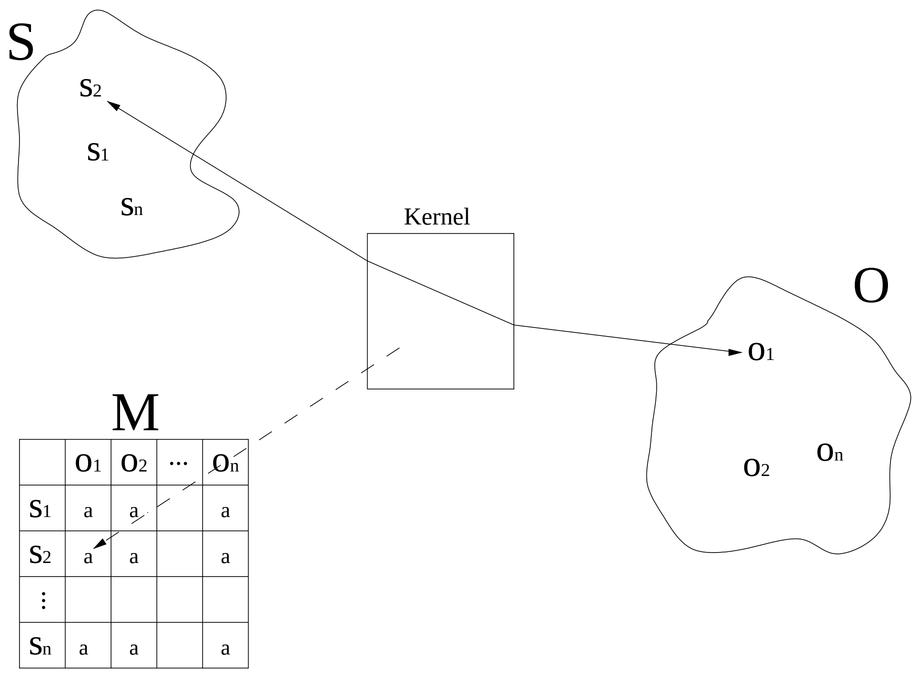

Figure 2.1: The general MAC approach.

2.2 The Medusa DS9 Security Project

Medusa is a security project currently implemented for Linux, but implementations for other platforms are planned. Its approach to security is a Virtual Space model [48], which separates the objects and subjects of the system into a finite number of domains called Virtual Spaces (VS). The access matrix is divided into properties for subjects and properties for objects. Subjects and objects must share Virtual Space in order for operations to be permitted.

Second part of the Medusa security framework, is the Security Decision Center (SDC). The SDC is responsible for updating the Virtual Space sets, as well as allowing or denying access to objects.

In the following the example shown in Figure 2.2, goes through authorization of operations to demonstrate how this is done.

When subject P wishes to do a *write* operation on object F, P's "write-VS's", Pw, are checked against the Virtual Space that F belongs to, by the SDC, i.e. Pw and F have to share one or more Virtual Space's. Figure 2.2 shows how Pw and F share $VS3$, and thereby making it possible for P to write to F.

The byte arrays depicted in Figure 2.2 shows how the implementation of the Virtual Space's is done. In the example there are three Virtual Space's, $VS1$, $VS2$, $VS3$. The object F belongs to $VS1$ and $VS3$, which can be seen by the high bits in the first and third place in the array. Every subject has a number of byte arrays corresponding to the number of access rights in A. In the example the subject P contains two byte arrays, one for each access right in

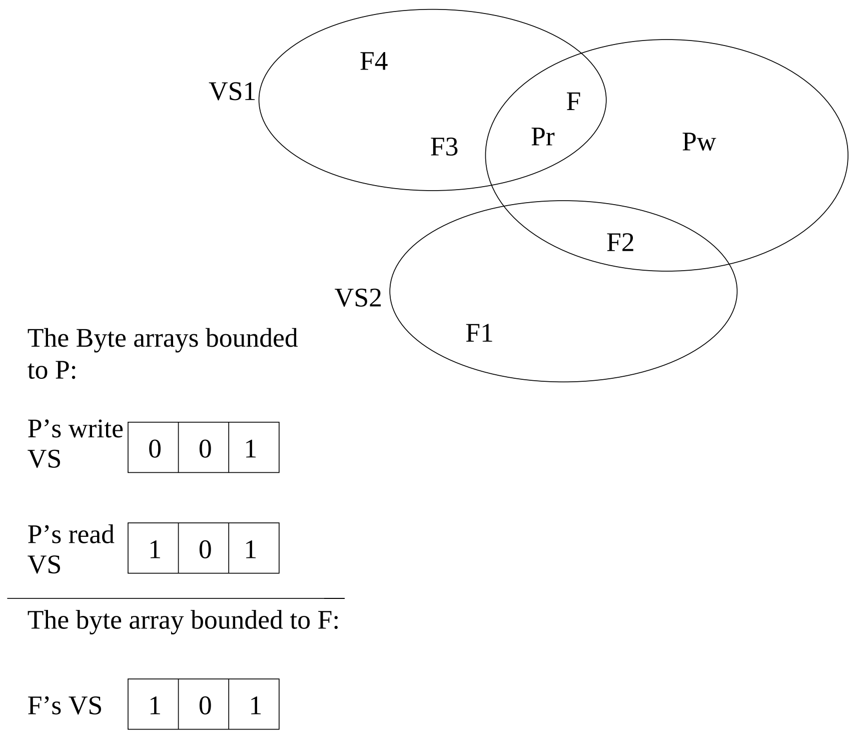

Figure 2.2: Example of Virtual Spaces.

$A = read, write$. In the Figure Pw belongs to $VS3$, meaning that P is allowed to do a *write* operation on all objects $\in VS3$.

The SDC is implemented as a user space security daemon called Constable. Constable is the current implementation of an authorization server. The user space implementation allows kernel changes to be simpler and smaller and thus easier to port to new versions of the Linux kernel and to be more flexible, so improvements to the authorization server should not require changes in the kernel.

The Virtual Spaces are implemented as a small patch to the kernel, which adds the arrays of bits in each subject and object, that indicate to which Virtual Spaces these belong. Each subject has such an array, for each possible action it can perform (*read* and *write* in the above example). Each object also has a bit array of which operations require Medusa's confirmation. The implementation as bit arrays, makes it possible to perform fast calculations when authorizing an operation. To check whether P has write access to F, is an *and*-operation on the numbers 001 and 101, seen on Figure 2.2.

2.2.1 Summary

The first impression of Medusa Project shows several advantages, mainly flexibility. This is achieved by means of the Virtual Space model, which allows the use of advanced security policies. Furthermore the implementation of the

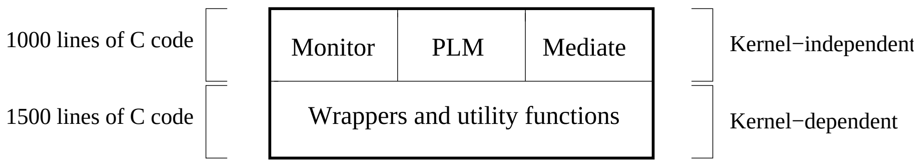

Figure 2.3: Loadable kernel module architecture.

Security Decision Center as a user space daemon, makes it possible to update policies, without changing the kernel. However, the Medusa project is not mature enough to be considered for use in production. The amount of available documentation is very limited, and available versions of Medusa exists only to deprecated versions of the Linux kernel. Further information on the Medusa Project can be found at: `http://medusa.fornax.sk.`

2.3 LOMAC - MAC You Can Live With

The LOMAC system was described in [27] by Timothy Fraser and the following is a description of LOMAC based on this article.

LOMAC was developed to make it possible to apply mandatory access control mechanisms to standard Linux kernels already in use. One of the key features of LOMAC is the fact that it uses an approach, which emphasizes compatibility and transparency to the user. This is achieved by using a simple but useful MAC integrity protection to Linux which:

- Is applicable to standard kernels.
- Is compatible with existing applications.
- Requires no site-specific configuration.
- Is largely invisible to the user.

Using these design criteria LOMAC was developed to solve some of the problems which have been found with other MAC implementations which include; incompatibility with existing kernel and application software, increased administrative overhead and disruption of traditional usage patterns.

2.3.1 Getting control

LOMAC is an implementation of a Low Water Mark MAC protocol [26] in a loadable kernel module, as seen in Figure 2.3. This implementation can be deployed to standard off the shelf Linux distributions. This is accomplished by interposing LOMAC between the kernel and the processes, at the kernels system call interface. This is done at initialization time, where the kernel's system call vector is traversed and the addresses of the security-relevant system calls are replaced with addresses of the corresponding wrapper functions in LOMAC.

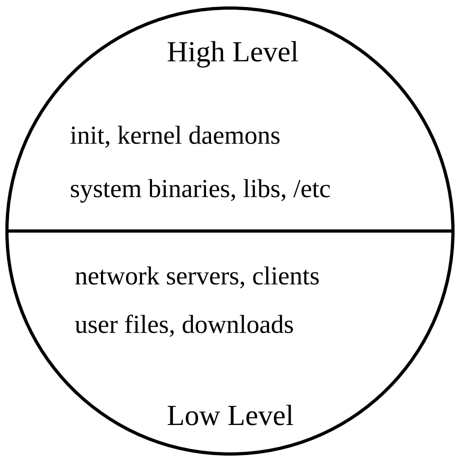

Figure 2.4: The two levels in LOMAC.

A description of the LOMAC wrapper functions can be found in [27], where examples of such functions are given.

2.3.2 Security Levels

LOMAC divides the system into two integrity levels; high and low, both seen in Figure 2.4. The high level contains the critical system components such as the init process, kernel daemons, system binaries etc. The low level contains client and server processes that read from the network, local user processes and their files. Once the integrity level is assigned to a file it is never changed, but high-level processes can be demoted on run-time, if they read low-integrity data. It is however not possible to increase the integrity level of a process once it has been demoted.

LOMAC uses this division of integrity levels to provide protection in two ways. First, it prevents low-level processes from modifying or signaling high-level files and processes. Secondly, LOMAC prevents migration of low-level data to the high-integrity level by demoting any process reading low-level data.

The assignment of integrity levels is achieved using a small set of rules to determine the integrity level of the different parts of the file system. These rules are specified at compile-time, since the goal of the current implementation is to use a single generic configuration, that provide some protection on all systems. This is done with a simple form of implicit attribute mapping where the path names are stored in an a list of records, which contain the path name, a child-of flag and the security level. This list is ordered by the length of the paths. When the integrity level for an object has to be decided, the list is traversed in a linearly fashion until a match is found. It is proposed in [27] that the use of a hash table, will give quicker lookups. The child-of flag is used to set the integrity-level of all sub-directories in a path, hence if a directory is low-integrity all sub-directories in the directory is also given a low-integrity label.

2.3.3 Exceptions and known problems

To maintain compatibility with some important parts of the operating system, it is necessary to allow some processes to modify local `/etc` configuration files. An example of such a process is the pump client side DHCP agent which modifies local configuration files like `/etc/resolv.conf`, on behalf of a remote DHCP server.

Known problems and issues with LOMAC include problems with running high-level processes which are using temporary files. This is due to the nature of the `/tmp` directory has to be a low-integrity level directory. The result of this is that all high level processes using temporary files will be demoted to a low-integrity process.

LOMAC does not prevent malicious low-level processes from harming the integrity of other low-level parts of the system. This includes the possibility of modifying, deleting files or sending kill signals to other low-level processes.

The use of "trusted" programs provide an inherent security risk, since there is a possibility of bugs in all programs. An example of this is the OpenSSH server, which is a trusted process, in which vulnerabilities have been found a number of times.

2.3.4 Summary

LOMAC provides a generic protection to standard Linux systems, by providing a default configuration. Security policies are specified at compile time, which prevents dynamic changes of security configuration. The use of only two integrity levels does not provide sufficient flexibility to device a detailed policy of access rights for individual processes. Further information on the LOMAC project can be found at: `http://opensource.nailabs.com/lomac.`

2.4 Security-Enhanced Linux

Security-Enhanced Linux (SELinux) [40] is based on the Flask security architecture [42], which provides a clean separation between the policy enforcement code and the policy decision-making code.

The Flask architecture provides two policy-independent data types for security labels, security context and security identifier (SID). SIDs are mapped to a security context by a Security Server.

Figure 2.5 depicts the components included in SELinux. As the Flask architecture devices, the policy decision-making and policy enforcement components are separated. Below, these components are briefly elaborated.

2.4.1 Policy decision-making

The security server includes all the policy decision-making code. The security server is completely independent from the rest of the system, and can thus be substituted by another implementation.

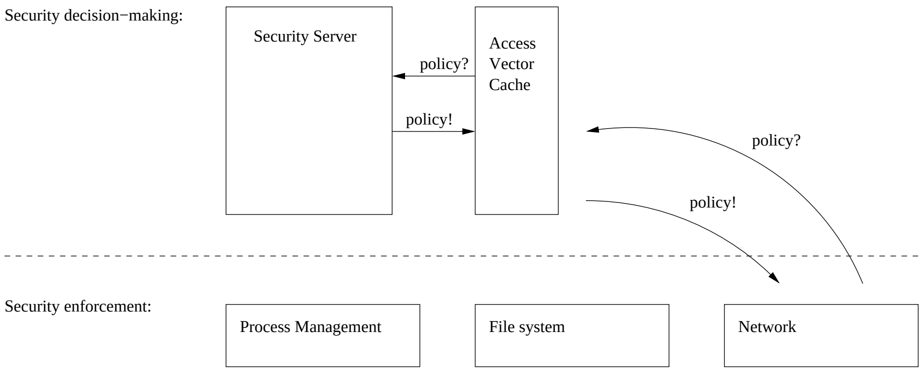

Figure 2.5: Component overview of Security-Enhanced Linux.

To maximize performance, an access vector cache (AVC) is implemented. The AVC caches decisions made by the security server. Furthermore the AVC also includes a number of security classes, that is used for easy classification of subjects and objects. A security class is an access vector, that is shared among a number of files. When the AVC is consulted it returns a reference to the entry in the cache that contains the policy in issue. Details on configuring SELinux are found in [39].

2.4.2 Policy enforcement

The policy enforcement is implemented on top of the Linux Security Modules (LSM) framework. This framework provides a set of hook functions in the Linux kernel, which supports modularity for implementing a security system. More information on LSM, can be found in Appendix C.

The policy enforcement code opaquely handles security contexts and SIDs, which can only be interpreted by the security server. The policy enforcement code also binds SIDs to active processes and objects, consulting the security server when a SID needs to be computed for a new subject or object.

The policy enforcement code obtains policy decisions from the security server, through the AVC. These decisions are used to assign security labels to processes and objects, and to control operations based on these security labels. This is done by passing a pair of SIDs and a security class.

The policy enforcement code in the file system, maintains a persistent mapping in each file system, that maps inodes to persistent SIDs and maps these persistent SIDs to security contexts.

2.4.3 Summary

The main advantage of SELinux is the clear separation of policy decision-making and policy enforcement. This especially makes the policy decision-making adaptive to changing domains. Furthermore, the policy enforcement is implemented on top of the general kernel security framework LSM. Parts of the design and implementation of SELinux can be used in designing a security measure optimized for Linux on handheld devices.

2.5 Linux Intrusion Detection System

Linux Intrusion Detection System (LIDS) is a kernel patch and administration tool for strengthening Linux kernel security. LIDS implements a reference monitor and MAC in the kernel. Furthermore it implements a port scan detector, a file protection system and a mechanism for protection of processes.

LIDS protects the kernel from tampering by inserting a security check into system calls. The file protection is based on the fact that access to files is also handled through system calls. Process protection is implemented in the same manner. LIDS store access restrictions in an access control list, one for files and one for restrictions on processes. Restrictions are based on object and subjects. The access control lists are integrated into the virtual file system, making the access control lists independent on the underlaying file system.

To further protect the kernel, LIDS provides a feature that makes it possible to seal the kernel. Once the kernel is sealed, it is no longer possible to load or unload loadable kernel modules. This feature is implemented to ensure the integrity of the LIDS kernel [29].

2.5.1 Summary

LIDS provides features for restricting access and protecting the system. These features are based on an static ACLs. The idea of sealing the kernel is interesting to protect the kernel from malicious loadable kernel modules from intruders. However, the LIDS implementation seems weak in that the kernel can be unsealed by a user space program. More information regarding the LIDS project can be found at the project web site `http://www.lids.org.`

2.6 Related Projects

Other related projects that deal with security issues include the following.

- Janus – `http://www.cs.berkeley.edu/~daw/janus`
  - Janus is a security tool for sandboxing untrusted applications within a restricted execution environment.
  - Latest version is for Linux 2.2.x kernels running on the x86 architecture.
- GrSecurity – `http://www.grsecurity.net`
  - GrSecurity implements role-based access control, and has further made efforts for “change root” hardening, race prevention of temporary files, extensive auditing, additional randomness in the TCP/IP stack and allows users to view personal processes only.

  - Latest versions are for Linux 2.4.x and 2.6.x kernels.

- RSBAC – `http://www.rsbac.de`
  - The RSBAC framework is based on the Generalized Framework for Access Control by Abrams and LaPadula [21]. All security relevant system calls are extended by security enforcement code. This code calls the central decision component, which in turn calls all active decision modules and generates a combined decision. This decision is then enforced by the system call extensions.

    Decisions are based on the type of access, the access target and on the values of attributes attached to the subject calling and to the target to be accessed. All attributes are stored in fully protected directories, one on each mounted device. Thus changes to attributes require special system calls provided.

    All types of network accesses can be controlled individually for all users and programs, which gives full control over network behavior and makes unintended network accesses easier to prevent and detect.
  - Latest development versions are for Linux 2.4.x and 2.6.x kernels.

- VXE – `http://www.intes.odessa.ua/vxe`
  - Virtual eXecuting Environment, VXE, mainly offers daemon protection and restrictions on shells, where ACLs specify lists of files which are available to different users.
  - Latest version is for Linux 2.4.16 kernel.

- Openwall Project – `http://www.openwall.com`
  - Memory protection patch. Dismissed by Linus Torvalds.

- The PAX Project – `http://pageexec.virtualave.net`
  - Stack protection patch, that prevents introduction and execution of arbitrary code.

- LinSEC – `http://www.linsec.org`
  - The main aim of LinSec is to introduce mandatory access control into Linux. LinSec consists of four parts. Capabilities, file system access domains, IP labeling lists and socket access control.

    As for Capabilities, LinSec heavily extends the Linux native capability model to allow fine grained delegation of individual capabilities to both users and programs on the system.

    The file system access domain sub-system allows restriction of accessible file system parts for both individual users and programs. Now you can restrict user activities to only its home directory, mailbox, etc.

    IP labeling lists enable restriction on allowed network connections on per program basis. The policy may be configured so that no one except e.g. the mail transfer agent can connect to remote port 25.

The socket access control model enables fine grained socket access control by associating, with each socket, a set of capabilities required for a local process to connect to the socket.

– Latest version is for Linux 2.4.18 kernel.

2.7 Discussion on Existing Projects

The investigated projects all contain different features of more or less interest. The following discussion will evaluate these features.

The lack of support for newer Linux kernels and the limited amount of documentation is a drawback of the Medusa Project. The flexibility of the Virtual Space model is interesting, it does however, require configuration from a system administrator, which is not desirable on a handheld device. Furthermore Medusa is designed to operate in a multiuser environment, where all subjects and objects are assigned individual security contexts, which does not fit normal usage for a handheld device. In Medusa the Security Decision Center is implemented as a user space daemon, to ease the implementation. It is unclear how the Medusa developers intents to protect the user space security daemon from other user space applications, but it is clear that the SDC would be better protected in the kernel.

LOMAC was designed as a transparent MAC implementation for use in existing systems. For use in handheld devices the design of the LOMAC system is simply to inflexible due to the two level model and the compile time configuration of policies. Although the two security levels would in theory allow a separation of important system processes and configurations from the rest of the system. This does however create a range of rather problematic issues. Although, the compile time configuration eliminates user interaction, which is useful on handheld devices, it creates a completely static list of rules.

SELinux offers clear separation of policy decision-making and policy enforcement code, as devised by the Flask model. This is a very interesting due to the high flexibility it provides. The LSM framework is also very interesting, since it provides an interface on which a security system can be based. By basing a security module on LSM, the module will require much less maintaining across different versions of the Linux kernel. However like Medusa, SELinux is designed for a different purpose, and include e.g. mechanisms to prevent migration of information, making the framework rather complex. Furthermore SELinux is based on assigning properties to objects and subjects, thereby requiring a policy to be made for all new objects and subjects in the system. This type of policy management requires active participation of a security administrator, which is not desirable on a handheld device intended for normal users.

2.8 The Idea of Umbrella

The MAC implementations investigated, use object lists to describe the privileges assigned to the objects in the system. However this results in a static list of objects and privileges, where any changes would require the intervention of a security administrator and possibly a reboot. On a handheld device this is not convenient and should be replaced by a scheme, where privileges are assigned dynamically, with minimal user interaction.

Umbrella will operate by means restrictions as opposed to access control lists. These restrictions will be effective on processes only. This way, the security system does not need to consider a policy for programs that are not executed.

The process based restrictions will be inherited from the parent process to its children. Children will always be as restricted as their parent. The parent process has furthermore the ability to specify additional restrictions for its children. This naturally has to involve the programmer. Umbrella will therefore specify a very simple API for setting restrictions for next sub-process, i.e. next child.

By introducing heavy restrictions on untrusted programs, the Umbrella MAC scheme can be used to build a virtual sandbox for these programs. In this sandbox, access to network, file system or process signaling, could be restricted.

To make it possible to assign restrictions to programs when executed, the process based MAC will be combined with the possibility for developers to specify execute restrictions for their programs. Thus, when a program is executed, the process will get the restrictions specified by the developer and the restrictions it inherits from its parent.

The concept of trusted vendors will be introduced by means of public key cryptography. The public key of a vendor must be present within Umbrella. When the vendor distributes his program, he will have to include a signature that includes execute restrictions, and a hash of the executable file. Umbrella can verify, that the binary and the execute restrictions have not been tampered with during transfer to the system.

Umbrella will be implemented by the concept of the Flask model and based on LSM.

3 Design

Umbrella is a security system that implements process based mandatory access control. Umbrella implements MAC using restrictions on processes to control and restrict access to resources. Vendor-signed files are used to introduce restrictions to the system and to authenticate the origin of files.

This chapter explains how Umbrella is designed, beginning with a view of the different components and how they are connected. After this the different components are described in further detail. This chapter can be read separately to give a description of Umbrella. Full details on the implementation can be found in Chapter 4.

3.1 Top Level Design

Before going into the details of the design of Umbrella, an overview of is given. Figure 3.1 on the following page explains the different components of Umbrella and how they are connected.

3.1.1 Resources on a Handheld

Umbrella can enforce restrictions on processes, but what would it be interesting to restrict processes from doing?

New handhelds like the iPAQ 5550 are packed with devices for communication like bluetooth, infrared and wireless network. It is important to be able to control and restrict access to these devices. Handheld devices are more and more integrated into everyday life and users can store personal information and confidential data on their handheld. It is vital to have the possibility to protect these data.

3.1.2 Placement of Umbrella

As seen in Figure 3.2 the kernel runs on top of and protects the hardware. Conceptually LSM is placed on top of the kernel and can thus be used a foundation to build security modules for Linux. Umbrella is placed on top of LSM. All of this is kernel space, and therefore protected by hardware. Umbrella can from its position mediate calls from user space to hardware resources.

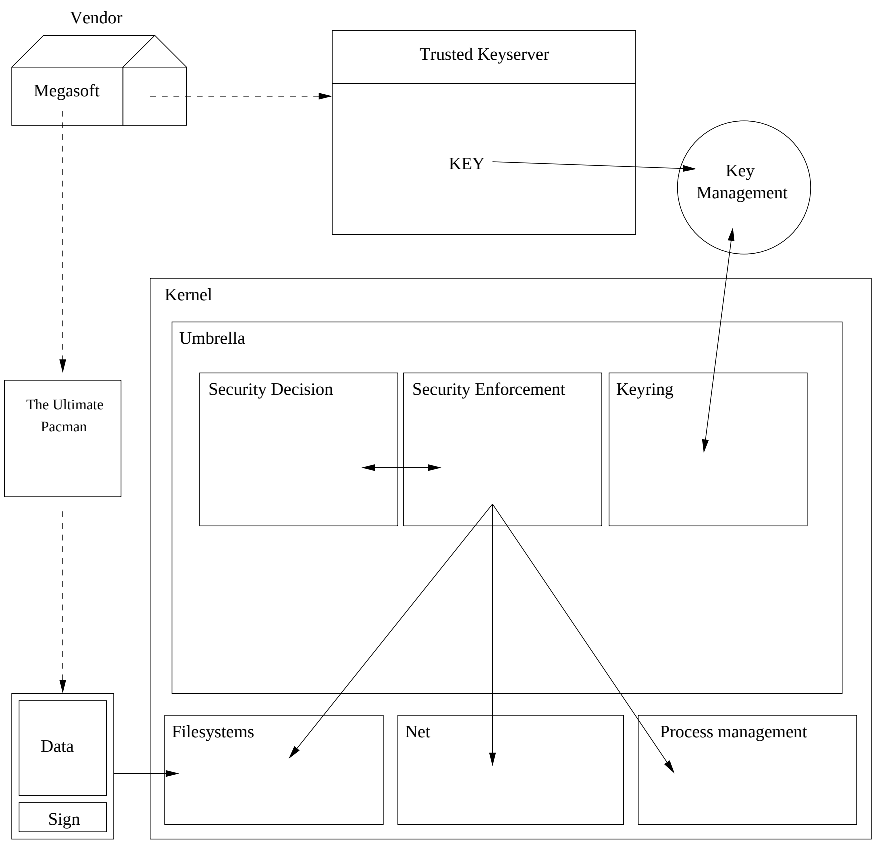

Figure 3.1: Top level design of Umbrella. In the upper left corner, a vendor is depicted. This vendor have made his public key available on a trusted key server. A user downloads the product “The Ultimate Pacman” to his handheld. Since the vendor’s public key is available from the key server, it can be inserted into the systems key ring, using the key management part of Umbrella. From there it can be used to authenticate the origin of “The Ultimate Pacman”. The signature contains a number of restrictions that “The Ultimate Pacman” must be executed with. When the executable is transfered to the file system, the signature is verified and the restrictions are stored along with the executable. Processes are monitored by the security enforcement component to make sure restrictions are enforced. The security decision component is asked for security decisions, given a process and its associated set of restrictions.

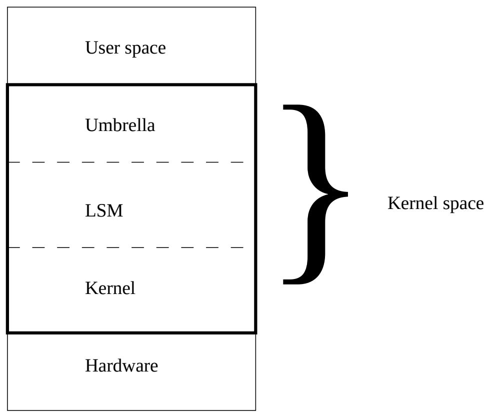

Figure 3.2: The placement of Umbrella.

The security enforcement part of Umbrella is to be implemented using the LSM framework. The use of LSM in the implementation is detailed in Chapter 4 and the framework itself is explained in Appendix C. Read this Appendix to gain knowledge of LSM, which is needed in the report.

3.2 Process Based MAC

The major drawback of all the existing MAC implementations is mainly maintenance of the access matrix, as described in Chapter 2. Even though this matrix may be modeled by means of e.g. type-enforcement, the basic problem exists; we still need to account for every object and subject in the whole system.

Whenever new subjects or objects are added to the system, these must be added to the access matrix. If 1000 objects exist in the system and a new subject is added, decision will have to be made for all objects if this new subject will have access or not.

Measures like type-enforcement does indeed make the access matrix a bit more flexible, but new objects and subjects, will still have to be assigned or added to the right domain or type.

Umbrella takes a new approach to solve this problem by introducing process based mandatory access control. Furthermore, to eliminate the access matrix, the access control measure is enforced by restrictions only. Thus, every running process have access to the whole system, except a given set of restrictions. Discretionary access control still apply, i.e. Umbrella co-exists with DAC.

To justify, if process based restrictions is enough to protect a system, consider which elements of any given computer system, at a given point in time, that possibly can do harm or access confidential material. The *only* item that can do harm is the process currently executing on the CPU. Everything else is not important, at this specific point in time. If the executing process is restricted appropriately, it can be guaranteed, that if it malfunctions, then the possible harm done will not affect the resources from which the process is restricted.

3.2.1 The Linux Process Tree

The idea of processes based mandatory access control is strongly supported by the way processes are structured in Linux. Processes in Linux are ordered in a tree structure. On Figure 3.3, an example from a running Linux system is depicted.

Every process, except `init`, have a parent. Security is enforced by ensuring that every process will be at least as restricted as its parent. This is to be done by inheriting restrictions from parents to children. Parent processes have the possibility of specifying additional restrictions for its children, which allows enforcement of the principle of least privilege.

3.2.2 Restrictions

Restrictions are very simple. Besides a few special restrictions, *all* restrictions are defined as a path in the file system (devices are files on a Linux system). If a restriction to a given path is set, the path and everything below is unaccessible for the process. The special restrictions are listed in the lower part of Table 3.1 on page 36.

By introducing restrictions, we need to consider how these will be modeled. Three approaches were considered.

The Different Approaches

A static model, where a fixed number of restrictions are defined, will be very easy to implement, very small, and easy to optimize for minimizing overhead. However, this approach would make the system highly inflexible. The restrictions that developers would be able to set, would be those in the static list defined in Umbrella.

To maximize performance, the static model could be implemented as a bit-vector, associated with each process and file found on the system. Each bit in the vector will represent a specific restriction, thus, if a bit is set, then the corresponding restriction applies to the process. The use of a bit-vector can minimize the CPU instructions used for e.g. inheritance of restrictions, since copying may be done with only one cycle (for a 32 bit vector).

The first approach was to use a bit-vector only, making it very long, so that new restrictions could be added. To find the appropriate index in the vector, a global hash table should map restrictions to indexes in the bit-vector. This did not work, as the size of the bit-vector in any case would be critical. Furthermore, removing restrictions from the system, would require updating every file and process in the system, which is unacceptable.

A dynamic model will include the ability for adding an unlimited number of restrictions, storing the restrictions in e.g. structures, mapped in a hash table. This would inevitably be slow, since several memory allocations are needed for each restriction. Also inheritance of restrictions from parent to children would be slow due to this.

The dynamic model, would model each restriction as a structure, and store all

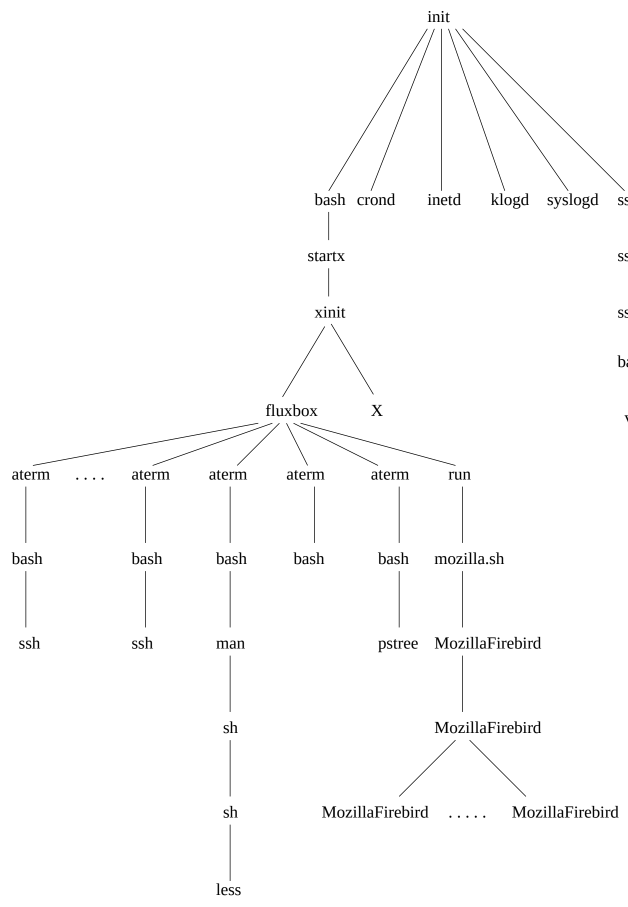

Figure 3.3: Example of the Linux process structure. When a Linux system boots and the kernel is loaded, the first process created is the `init` process, which is executed from `/sbin/init`. The root node is thus the `init` process. This process spawns a number of daemons. The Secure Shell daemon (`sshd`) has spawned a child process to handle an incoming request. User interaction starts with a Bourne Again shell from which the graphic interface (`X`) is launched. The running window manager is the `fluxbox` process, from which the user has spawned a number of terminal programs (`aterm`) running a shell (`bash`). In the right branch of the children of `fluxbox`, a Mozilla Firebird browser is started, which also has a number of children.

these in a local hash table for each process. This solution would provide a very high degree of flexibility, but unfortunately it would also be very slow, as the entire hash table would be copied, when inheriting restrictions from parents to children on creation of new processes.

3.2.3 The Umbrella Approach

The solution chosen for Umbrella is a combination of the static and the dynamic model. A bit-vector will be used to store a small amount of static restrictions, that will be specified according to the specific platform.

For the HP iPAQ, restrictions for devices such as network, bluetooth, infrared, wireless network, etc. will be added, together with a small list of vital paths from the file system. Index zero in the bit-vector, will be reserved to indicate if the process has assigned a number of dynamic restrictions.

We denote the static restrictions, i.e. the bit-vector, *level 1 restrictions* and the dynamic restrictions, i.e. the structures in the hash tables, *level 2 restrictions.*

When the process tries to access a resource, the level 1 restrictions are checked. If this denies access to the resource, access is denied immediately. If not, the level 2 restrictions are checked accordingly.

This combination of level 1 and level 2 restrictions gives Umbrella the desired performance and flexibility.

Inheritance

When a process forks a new child process, the restrictions will be inherited from the parent to the child. We utilize the Linux process tree to ensure that *a process is always as restricted or more restricted than its parent.* To ensure this, it is thus not possible for a process to change its own restrictions.

When a process forks a child, it has the possibility of setting additional restrictions for this child. When the child process is forked it inherits all restrictions from its parent, and any additional restrictions specified by the parent. If the new process is an executed binary, then execute restrictions are added. Execute restrictions and signed files are elaborated in Section 3.3. This procedure is depicted in Figure 3.4. A consequence of this is that children is always at least as restricted as their parent.

Restrictions 3.1 *Given that p_1 and p_2 are nodes in the process tree P and p_1 has the restriction set r_1 and p_2 has the restriction set r_2, where r_1 and r_2 are sub-sets of R which is the set of all possible restrictions.*

If p_1 is a descendant of p_2 then r_1 is a superset of r_2.

The inheritance of restrictions happens before the process is created, and thus the new process is not scheduled before all restrictions are set.

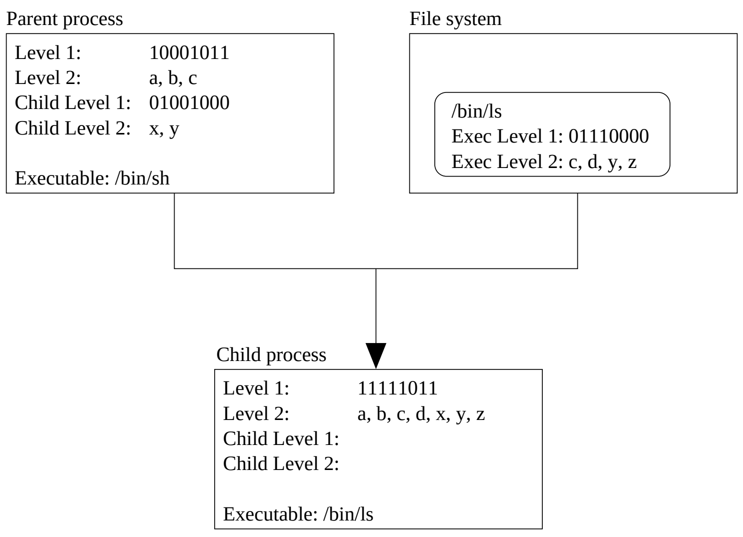

Figure 3.4: The procedure of creating a new child process and setting its restrictions.

Data Structures

As stated above, Umbrella uses a two level design for the restrictions. The data structure for holding the level 1 restrictions is a bit-vector, which contains a fixed set of restrictions for the entire system. The restrictions are mapped to a specific index by a small hash table. The level 2 restrictions are fixed for a running process, but new processes must have the ability to have an arbitrary number of restrictions. These restrictions are stored in structures, which are mapped in a hash table. At top level, these structures are connected as depicted in Figure 3.5.

The bit-vector contains a global fixed set of basic and frequently used restrictions. If a process should be restricted from other resources than those located in the bit-vector, these are placed in the hash table. The bit-vector contains a bit that indicates if further restrictions should be looked up in the level 2 hash table. Given the limited size of the level 1 hash table, it could possibly fit into the CPU cache, increasing speed of lookups.

The level 1 restrictions are listed and commented in Table 3.1. This list is made specific to the devices and file system on the HP iPAQ running Familiar Linux and should be customized for use on other systems.

The purpose of the level 2 hash table is to provide a flexible data structure for further file system restrictions that applies only to specific processes. The restrictions in level 2 are only evaluated if the *secondlevel* bit in the bit-vector is set, as defined in Table 3.1. The hash table is individual for all processes, which makes it slower to perform a lookup in this, compared to the bit-vector. The restrictions in level 2 are *only* paths from the file system, i.e. this could include, personal files such as an address book, specific configuration files, programs, home banking keys etc.

Index	**Restriction**	**Comment**
0	secondlevel	Flag for lookup level 2 restrictions
1	/bin	Directory for system-wide binaries
2	/boot	Kernel and boot loader configurations
3	/dev	All devices (except network)
4	/etc	System-wide configurations
5	/home	Users' home directories
6	/lib	Libraries
7	/mnt	Mount point for external file systems
8	/proc	Run-time kernel information and configuration
9	/root	Home directory of administrator account
10	/sbin	Programs primarily for administrator
11	/tmp	Temporary files
12	/usr	System-wide applications, libraries, header files etc.
13	/var	Cache, locks, logs, web, databases etc.
14	/etc/X11	Configuration of X – graphical interface
15	/etc/bluetooth	Bluetooth configuration
16	/etc/fstab	Configuration of device mount points
17	/etc/init.d	Boot-time init-scripts
18	/etc/modules	Configuration of kernel modules
19	/etc/modules.conf	Configuration of kernel modules
20	/etc/passwd	User of the system
21	/etc/services	Xinet daemon configuration
22	/etc/shadow	Users' encrypted passwords
23	/etc/umbrella	Umbrella configurations and vendor keys
24	nofiles	Access to all file systems
25	noip	All networking through IP sockets
26	noirda	Infrared devices
27	nobluetooth	Bluetooth devices
28	nofork	Ability to fork new processes

Table 3.1: Level 1 restrictions in Umbrella for the HP iPAQ 5550.

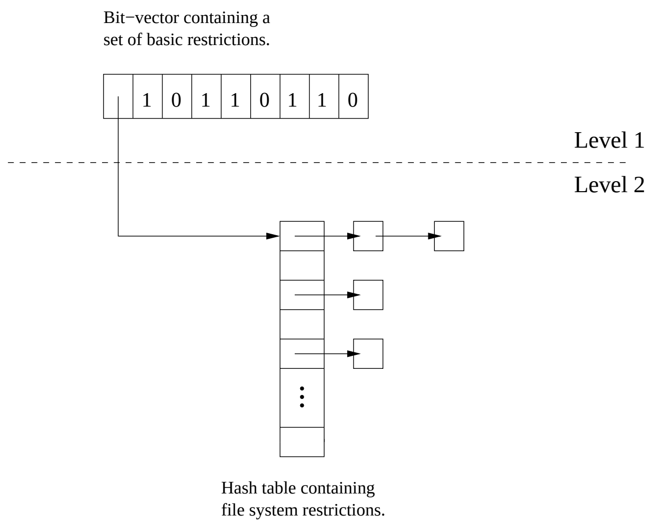

Figure 3.5: Umbrella uses two levels of restrictions.

Applying Restrictions

When a process tries to access any given resource, Umbrella mediates this and examines if a restriction is set for that specific path. If so, access is denied. Also access to network devices and other resources are mediated to examine if the process is restricted from these.

3.2.4 Involving the Developers

Developers of software must be involved to make secure programs. For Umbrella it is vital that when a process forks a new child, a suitable set of restrictions is specified. This is the *only* effort required by the developers. Below two examples of restrictions for new processes are given.

If a thread is only rendering a picture, this process should have level 1 restrictions `{24, 25, 26, 27, 28}` from Table 3.1. If the process is hijacked, the hijacker have no access to the file system, nor the network or even the ability to fork a new process. This in effect sandboxes the process, making it impossible to do harm, besides crashing the process.

Another example is execution of attachments from email clients. When the attachment is executed from within the email client, a good restriction for the developer to set is access to the address book. In this way, if the attachment is a virus, it is unable to forward itself to everyone in the address book. Possibly a restriction from network access for the attachment would also be desirable, to completely prevent abuse of the network.

When a developer has made a program for an Umbrella protected system, the developer can specify execute restrictions, which is a set of restrictions the program is assigned, when executed. These restrictions are stored along with the file, in a signature that also contains means of authentication.

3.3 Signed Files

Umbrella relies on a scheme where files entering the system must be authenticated. This serves a dual purpose; the authentication of files provides protection from executables that may be harmful to the system and the signatures provide a way of importing restrictions to the system. The following sections elaborate on the design of the signatures attached to files, how these are utilized to prevent possibly malicious executables from compromising the system, and how they are combined with the use of trusted key servers.

In the following, a *signature* denote a hash value and a set of execute restrictions that is encrypted with a private key. A *signed file* denote a file and its associated signature, as shown in Figure 3.6 on page 40. The *security information* associated with an executable on an Umbrella file system is the associated execute restrictions. This information is stored in the executables *security field*.

3.3.1 File Signatures

One of the central elements in Umbrella is the signatures which are used to ensure the integrity of files entering the system and provide the set of restrictions that each file is to be executed with. The requirements of the signatures identified for Umbrella are:

- Signatures must include restrictions for use on time of execution.
- Umbrella must be able to sign all types of files, including executable scripts and binary executables.
- The use of signatures must be transparent to the user.

Possible Signature Models

In the process of investigating possible models for signed files, three possible solutions is discussed. A solution where the signature is kept in a separate file, another possibility is to modify the executable file format to include the signature and finally the option of simply appending the contents of the signature to the end of the file. Each of these possibilities are discussed below.

The first possible solution to the signed file problem, was to transfer the signature to the handheld device in a separate file. The implementation of such a solution would not require any changes to existing file formats, which clearly is an advantage. However, several problems exists with transferring the signature to the handheld separately. To be able to assign restrictions to an executable as soon as it enters the file system, the signature would have to be transferred to the system before the associated file. This could be implemented using a signature cache where the signature is copied to before the executable enters the system. The cache would then be checked when a new file enters the file system and if an associated signature was found, the signature would be copied to the files security field. If an associated signature was not found, the file would be assigned a default set of restrictions[1]. Alternatively the transfer of the signa-

[1]This default set of restrictions is specified by the vendor of the handheld device.

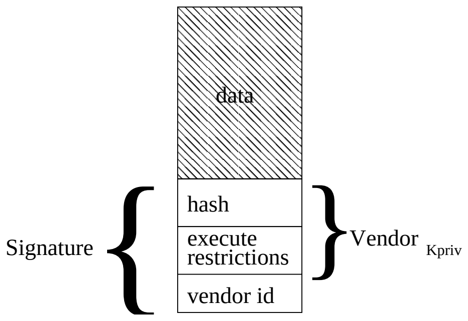

Figure 3.6: A signed file.

tures could wait until the file is executed for the first time. The major drawback of this solution is the maintenance of the signature cache as well as the lacking transparency, due to the two file solution.

The problems found with the two files solution, suggested that solution where the signature is included in the file is preferable. For executable files a possibility was to modify the ELF file format like done in the DigSig project [9] and the work described in [45]. This solution would definitely offer a simple way of transferring signatures for executables to the handheld device. This solution does not however offer any way of using signatures with scripts and other executable formats. As a result of this any support of other file types than ELF would result in modifying additional file types, which is undesirable.

The chosen solution is to append the signature to the end of the executable. Like the two file solution, no changes are needed to existing file formats. This fact also makes it possible create signatures for scripts and other non executable files. Appending the signature to files, does require that the signature is encapsulated in a tag, that makes it easy to separate from the original file. This is the solution selected for Umbrella, since it is believed to be a good solution for handhelds. The approach is transparent easy to implement and flexible. This is elaborated below along with details on how the information in signatures is transferred to the security field in the file system.

Signatures in Umbrella

The layout of a signed file can be seen in Figure 3.6. The signature consist of a hash of the original file and a set of execute restrictions, both are encrypted with a private key. A vendor id will ease the lookup of the corresponding public key.

The hash value is used for ensuring that the file has not been tampered with. When the signature is created the hash and the execute restrictions are signed with the vendors private key to prevent tampering with the contents, as well as for authentication purposes. The vendor id is used to identify which public key to use for decryption of the file signature. The public key itself can be used as the vendor id, since it is unique.

In a system protected by Umbrella, two types of files can enter the file system,

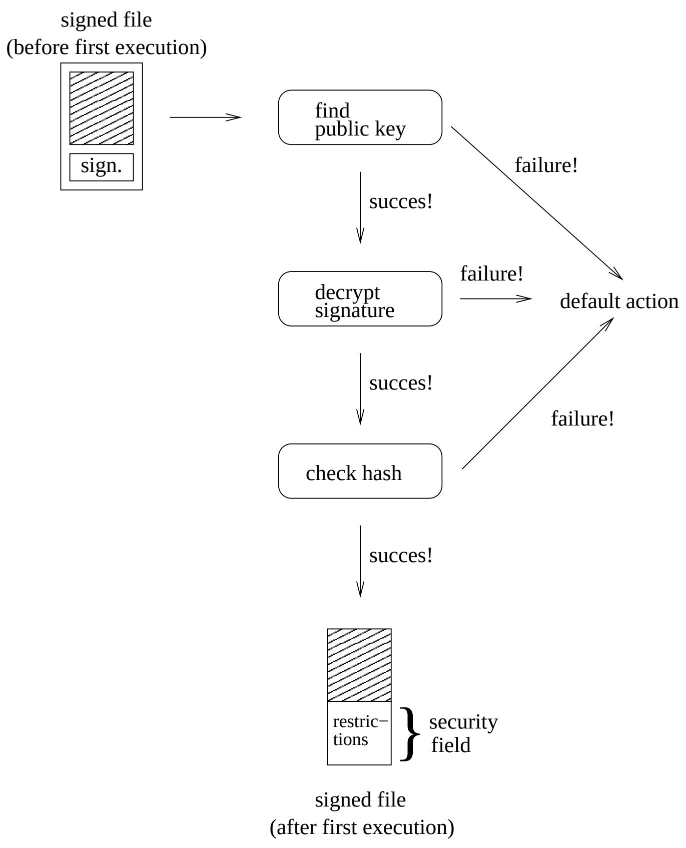

Figure 3.7: Procedure for first execution of a signed file.

namely signed and unsigned. Both are stored on the file system and no further action is taken until the file is executed. When a file is executed, the signature is decrypted using the corresponding public key and the hash of the data is checked, to ensure that the file has not been tampered with. The execute restrictions are stored in the file's security field. If the file has no signature, if the decrypting of the signature or the hash check fails or if no public key for decrypting the signature exists, the *default action* is performed. This default action is a set of restrictions specified by the vendor of the handheld device, and could be e.g. used to create a low level sandbox. Figure 3.7 shows the procedure before the first execution of a file.

The features described above gives Umbrella a secure and convenient method for importing new files to an existing system. The origin of the files is checked and the restrictions associated with the file is imported to the system transparent to the user. Below details will be given regarding handling of keys.

3.3.2 Public Keys in Umbrella

Since Umbrella relies on validation of file signatures for incoming files, a way of verifying the origin of signed files is needed. To achieve this Umbrella uses public

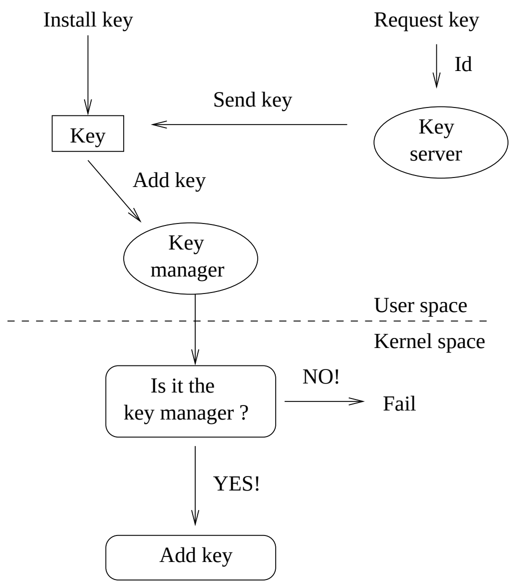

Figure 3.8: Procedure for adding a key to Umbrella.

key cryptography, where all signed files are signed with the vendors private key, as elaborated earlier. This ensures that only the correct public key can decrypt the signature, thereby verifying the origin of the file.

Distributing and Storing Keys

The distribution of public keys is one of the central security issues in Umbrella, which requires a scheme where the distribution is handled in a secure manner. Umbrella uses a trusted key server as basis to distribute public keys, because a trusted server ensures the authenticity of the public keys installed on the device.

Figure 3.8 illustrates the process of adding public keys to Umbrella. The key can enter the system in two ways. The user can request to have a key added directly or through a key server. The key is passed to the key management along with a request to install it. When the key management receives a request for installing a key, it uses a system call to insert the key into the key ring. To ensure that the key master is the only process, that can successfully make add and remove keys, a check of the current process is performed. In this way the keys can be protected, even though they are placed on storage accessible from user space.

The key ring contains public keys used for decrypting files from vendors. Persistent storage is needed to store the key ring, this storage must be protected to prevent tampering with keys and injection of unwanted keys. Management of the key ring is handled through system calls that are only accessible to a key management program. The kernel is then responsible for updating the key ring on the persistent storage.

The kernel writes the key ring to the directory `/etc/umbrella/keyring`. To

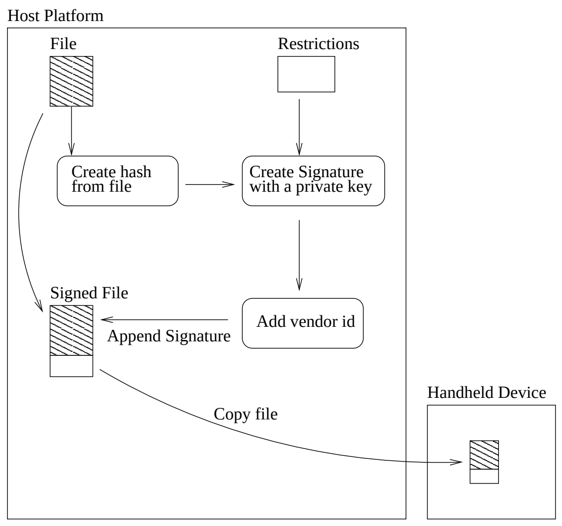

Figure 3.9: Creating a Signed File.

protect the public keys from unauthorized access, *all* processes will be restricted from this directory. A nice feature of inheriting restrictions from parent processes to children, now shows its strength: It is enough to restrict the `init` process from accessing the key ring directory, and this will automatically be inherited by all other processes.

Umbrella relies on a scheme using time stamps for the public keys, to maintain the contents of the key ring. This serves a dual purpose; it prevents keys from being cracked using a brute force type attack, and secondly it prevents outdated keys on the system. This scheme is standard in public key cryptography [38, 22].

3.3.3 Creating Signed Files

Development of programs for handheld devices are not done on the device itself, but rather on a host platform and then cross-compiled for the intended target platform. Umbrella uses this fact, to create the signed files on the host platform using a set of user space tools and then transferring the complete signed file to the device. The process of creating a signed file is illustrated in Figure 3.9. When the executable have been created, a set of restrictions are created for it. A hash is then created of the original file. The hash and the signature are then merged into a signature and encrypted with the vendors private key. When the signature is created it is appended to the executable.

3.4 Extra Features

This section describes design features that has been discussed during the development of Umbrella.

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25		Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

Table 3.2: Signals in the Linux kernel.

3.4.1 Controlling Signals

In Linux it is possible for processes to signal each other. One of the signals is used to kill a process. This means that an exploit in a process can be used to send kill signal to other processes owned by the same user, performing a denial of service attack. By mediating signals and restricting the use of these, Umbrella could effectively prevent attacks of this kind.

Linux supports the standard signals listed in Table 3.2, which are described in the original POSIX.1e standard [30]. Several of the signal numbers are architecture dependent, as indicated in the *Value* column. Three values are given, the first one is usually valid for Alpha and SPARC, the middle one for i386, PPC and SH, and the last one for MIPS. If only a single value is present, this is generic for all architectures. The entries in the *Action* column specify the default action for the signal, as follows:

- *Term:* Default action is to terminate the process.
- *Ign:* Default action is to ignore the signal.
- *Core:* Default action is to terminate the process and dump core.
- *Stop:* Default action is to stop the process.

In default kernels, the signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored, however the LSM framework extends the kernels capabilities to mediate signaling between processes. The hook `task_kill` can be used to control signals sent to processes. This would allow Umbrella to mediate signaling, thereby gaining the ability to restrict processes from sending kill signals to other processes. Preventing this type of signaling could help protect the system from malicious processes attempting to kill important system processes. This could be achieved by creating a *level 1* restriction called e.g. *nokill*.

This simple, yet very powerful feature is of high priority on the list of pending implementation issues.

3.4.2 Integrity of Files

Although Umbrella uses restrictions, to prevent processes from accessing specific files and directories, no mechanism exist for preventing modification of files outside the restricted areas. This creates a potential security risk, since it would allow a malicious process to modify executables located in unrestricted areas if DAC allows it. This would introduce the possibility of having non-trusted code executed with few restrictions.

If a trusted process is successfully attacked and used to launch a virus similar to the file viruses [10] from the DOS era, then although the virus is launched with the restrictions of the trusted process and DAC still applies, a situation emerges were any local unprotected files found on the system could be infected. The infected executables, with various sets of restrictions, could then potentially be used to further spread the virus on the system.

To prevent the situation described above, a scheme where changes to files containing restrictions are detected has been discussed. By extending Umbrella to monitor any changes to files with extended attributes attached via the LSM framework, it is possible to detect any attempts to modify existing files.

If an attempt to modify a file is detected, a check is performed to determine if a security field exists. If so, the file is marked as being dirty. The dirty field is checked before execution to determine if the file can be trusted. If the file is dirty, the restrictions in the security field are disregarded and the default set of restrictions are applied.

Another advantage of implementing this mechanism at kernel level is, that the dirty field can be protected because all access to it is mediated by Umbrella. This would protect the field from any attempts to unset it from user space. Any attempt to replace a file is also impossible, because it involves deleting the original file. When the file is deleted, so is the security field attached to it, and unless the replacement file contains a valid file signature it is only assigned the default set of restrictions.

3.5 Conclusion

This chapter presented the design of Umbrella, which is a process based mandatory access control mechanism for Linux. The idea behind Umbrella is to create a mechanism which is not based on the traditional subject/object model. The result is a design that uses the Linux process tree as a base to control the assignment of restrictions to processes, where restrictions are inherited from parents to children. The restrictions for a given process is the union of restrictions from the parent and the restrictions specified for the child. To ease the configuration of Umbrella the concept of signed files is introduced to allow importing restrictions for programs without user interaction. Furthermore two extensions to Umbrella has been discussed, namely the ability to control signaling between processes and preventing modification of executables containing restrictions.

It is our belief that the combination of process based mandatory access control together with authentication of files presented in the design of Umbrella is a step in the right direction for creating a flexible security scheme for handheld devices.

4 Implementation

This chapter contain details of the Umbrella implementation. The implementation of Umbrella is based upon Linux Security Modules, which is explained in detail in Appendix C. To fully understand the contents of this chapter, knowledge of LSM is needed.

All paths and filenames given in this chapter are relative to the root of the Linux 2.6 kernel source tree, except paths denoting restrictions.

Like the design chapter, the implementation starts with the process based mandatory access control, and then moves on to the signed files. Lastly optimizations to the current implementation are presented.

4.1 Process Based MAC

The process control block [43] in Linux is defined in the `task_struct` in `include/linux/sched.c`. LSM provides a void security field in this structure, which is used to hold the Umbrella security information for the process.

The security field for a process is specified by the `security_struct` in `security/umbrella/include/umb_types.h`. The level 1 and level 2 restrictions for the process are specified in `level1` and `level2`. Level 1 restrictions for the next child process are given in the bit-vector `child_level1` and the list of level 2 restrictions are available in the `child_level2` array. The maximum number of level 2 child restrictions are defined in `MAX_L2` as 256. This was found to be a reasonable upper bound on the number of restrictions, given the amount of resources found on a handheld.

```
struct security_struct {
    bitvector level1;
    hashtable level2;
    bitvector child_level1;
    char *child_level2[MAX_L2];
};
```

4.1.1 Level 1 Restrictions

The level 1 restrictions are implemented as a bit-vector. This implementation is adopted from a bit-vector library implemented by Steffen Beyer[1]. Special functions for Umbrella purposes are added, such as logical *or* and *and* functions, which are used when restrictions are inherited from parents to children.

Level 1 restrictions may only be two directories deep, as Table 3.1 on page 36 shows. This limitation is introduced minimize the time of a lookup. A few optimizations are still pending for the bit-vector implementation, which Section 4.3 presents.

4.1.2 Level 2 Restrictions

The level 2 restrictions are implemented in hash tables. These tables map the restrictions, which are stored in `restriction` structures, defined in `security/umbrella/include/umb_hash.h`.

```
struct restriction {
    unsigned long hash;
    char *path;
    struct restriction *next;
};
```

The hash function used for the hash table, does not implement perfect hashing, and can therefore not guarantee that no collisions on the hash value will occur. Therefore the `restriction` structure includes pointers to the next structure, that have identical modulated hash value. Due to the possibility of collisions, the restriction itself, i.e. the path, is also stored in the structure. Thus, when the appropriate `restriction` is found, a string compare is necessary. Details on the hash table implementation are given below.

Hash Functions: Jhash and KShash

In the Linux kernel the hash function *jhash* is available from `include/linux/jhash.h`. This function can hash everything, producing an unsigned long. The function was to be used for implementing hashing of paths, but unfortunately it produced different output for pointers to identical strings. Whether jhash can be tweaked to only hash the intended strings, is still unresolved. To achieve the primary goal of getting the Umbrella unfolded, before optimizations, a small kernel string hash function *kshash* was implemented as follows.

```
for (i = 0; i < string_length; i++) {
    if (i % 2)
        hashval = hashval * (int)string[i];
    else
        hashval = hashval + (int)string[i];
}
```

[1] http://freshmeat.net/projects/bitvector

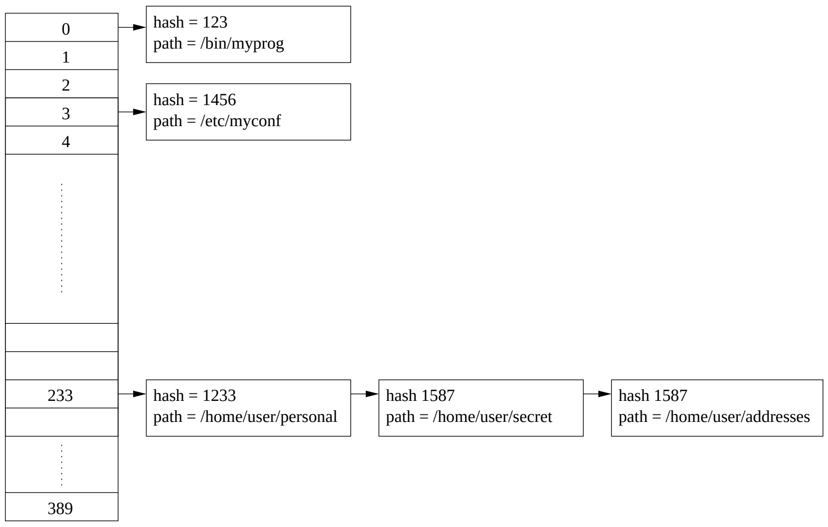

Figure 4.1: Hash table example.

This hash function produces an unsigned long, on which the modulus operation can be applied to find the appropriate index in the hash table.

Hash Table Implementation

The procedure for looking up a restriction in a hash table, first locates all structures which have identical hash value. These must be subject to a string compare with the path looked up. If a match is found, a negative decision is returned.

The restrictions are mapped into the hash table by calculating modulus 389 of the hash value of the path. The number for modulus is chosen to 389 because it is a prime, it is slightly less than a power of 2 and it is as far as possible from the nearest two powers of 2 [32]. Having a hash table, at around double size of the number of allowed restrictions, reduce the clustering in the hash table.

The restrictions in the hash table are copied from a parent to its children, when the child is created. This makes it very easy to clean up, when a child dies.

A great optimization to this would be to implement the copying using the *copy on write* principle [44]. In the case where a parent and its child have the exact same set of restrictions, the child could have a pointer to the parents restrictions instead of its own copy. When the child have further restrictions than the parent it will get a copy of the restrictions. This optimization will eliminate a lot of memory allocation and string copying.

With a perfect hashing algorithm, all the code for handling collisions could be omitted, however this is not possible since it would require that all keys are known in advance [12, 23].

An example of an Umbrella hash table containing collisions is depicted in Figure 4.1.

4.1.3 Security Server

The security server is implemented in `security/umbrella/umb_ss.c`. It basically has only one function for making decisions, namely the `ss_get_decision`, which takes a `path_array` as its only argument. The full path is divided by each slash, for example the path `/etc/passwd` is represented in the array `{"/etc", "/passwd"}`. The path array is implemented as such, to minimize string manipulations. Since the path is to be used in portions of one directory at a time, a regular string would be a bad solution. The first code listing shows the level 1 part of `ss_get_decision`.

```
1   /* LEVEL 1 - global restrictions */
2
3   if(path_array[0] != NULL) check1 = get_index(*path_array);
4
5   if((path_array[0] != NULL) && (path_array[1] != NULL)) {
6        second_dir = (char *)kmalloc(sizeof(char)*(strlen(*
             path_array) + strlen(*(path_array+1))),GFP_ATOMIC);
7        strcpy(second_dir, path_array[0]);
8        strcat(second_dir, *(path_array + 1));
9        check2 = get_index(second_dir);
10       kfree(second_dir);
11  }
12
13  if (check1 != -1) bit1 = bv_testbit(security->level1, check1);
14
15  if (check2 != -1) bit2 = bv_testbit(security->level1, check2);
16
17  if (bit1 || bit2) decision = 1;
18
19  if (bit1 == -1 || bit2 == -1) printk(KERN_INFO "Umbrella: umb_ss.c:
20      get_decision: bitvector overflow in checkbit!\n");
21
22  ......
23
24  return decision;
```

The path is first looked up in the level 1 restrictions. This lookup is done by comparing paths from level 1 restrictions (see Table 3.1 on page 36) with the first two elements in the path array. This is done by the `get_index` function in line 3 and in line 9. A position in the bit-vector is returned from the `get_index` function, which performs a simple string comparison, that easily can be optimized, as discussed in Section 4.3. The call in line 3 performs a lookup on the first element in the path array, and the call in line 9 performs a lookup on the concatenation of the two first elements. The function `bv_testbit` simply checks whether a given bit is set, and returns 1 if so and 0 if not. If the bit for the specific path is set, then the decision returned is negative. Otherwise, if level 2 restrictions are present, the path is looked up in the hash table.

```
/* LEVEL 2 - process specific restrictions */

if((!decision) && (bv_testbit(security->level1, SECONDLEVEL)) ){
    path_length = 0;
    i = 0;
    while(path_array[i] != NULL) {
        path_length += strlen(path_array[i]);
        i++;
    }
10      array_length = i;
11
12      temp = (char *)kmalloc(sizeof(char)*path_length, GFP_ATOMIC);
13          for (i = 0; i < path_length; i++)
14              temp[i] = '\0';
15
16          i = 0;
17          while ((array_length-- > 0) && !decision) {
18              strcat(temp, path_array[i]);
19              decision = hash_exists(security->level2, temp);
20              i++;
21          }
22          kfree(temp);
23  }
24
25  while(*path_array != NULL) kfree(*path_array++);
26
27  return decision;
```

Level 2 restrictions are also looked up in steps of directories. The test in line 3 checks if a decision was made when looking up level 1 restrictions, and whether level 2 restrictions exists. In lines 17-21 the elements of the path array are looked up, and when a decision is taken, temporary variables are freed and the decision is returned. The lookup is elaborated in the example below. The example contain many steps, but the individual steps are performed fast. All the level 2 lookups are hash table lookups which are performed in constant time. Furthermore the function returns as soon a decision is made.

Example

The security server's `get_decision` function is called with the path `/usr/share/myapp/bin/hello_world`.

The level 1 restrictions are checked as follows:

1. Lookup index for `/usr` in level 1 restrictions.
2. If bit is set, then return negative decision.
3. Lookup index for `/usr/share` in level 1 restrictions.
4. If bit is set, then return negative decision.

If no negative decision is made, the level 2 restrictions will be checked as follows:

1. Lookup `/usr` in level 2 hash table, and return a negative decision if it exists.
2. Lookup `/usr/share` in level 2 hash table, and return a negative decision if it exists.
3. Lookup `/usr/share/myapp` in level 2 hash table, and return a negative decision if it exists.
4. Lookup `/usr/share/myapp/bin` in level 2 hash table, and return a negative decision if it exists.

5. Lookup /usr/share/myapp/hello_world in level 2 hash table, and return a negative decision if it exists.

6. Return a positive decision.

4.1.4 LSM Hook Implementations

The LSM hooks used Umbrella are divided into three different classes of hooks: File hooks, network hooks and process hooks. In this section these different classes of hooks are described.

A full list of LSM hooks together with descriptions can be found in include/linux/security.h and in Appendix D. In Appendix C describes how the hooks are placed in the kernel.

File Hooks

The hooks implemented for protecting the file system are inode_create, inode_permission, inode_link, inode_unlink, inode_rename, inode_setattr and inode_mkdir.

All the LSM hooks intercepting calls to the file system are implemented in one simple and generic file_hook_wrapper, which is listed below. The call to the security server is performed in line 15.

The difference between the hook implementations for controlling the access to the file system, is mainly checking for NULL and digging out the right dentry. The dentry structure is short for *directory entry* and it is defined in include/linux/dcache.h.

```
1  static inline int file_hook_wrapper(struct dentry *dentry) {
2      int ss_decision = 0;
3      int i = 0;
4      char *path[MAX_PATH_DEPTH];
5
6      for (i = 0; i < MAX_PATH_DEPTH; i++)
7          path[i] = NULL;
8
9      if (extract_path(dentry, path) == -EOVERFLOW) {
10         printk(KERN_INFO "Umbrella file_hook_wrapper: ");
11         printk(KERN_INFO "Overflow from extract_path!\n");
12         return 1;
13     }
14
15     ss_decision = ss_get_decision(path);
16
17     if (ss_decision)
18         ss_decision = -EPERM;
19
20     return ss_decision;
21 }
```

Network Hooks

Only one hook for controlling network is necessary, namely the one for creating sockets. The important part of this hook implementation is locating the family of network socket being created. Since the network is part of the special level 1 restrictions, only one check for the appropriate field in the bit-vector is necessary. As devised by the Flask model, this check should be performed by the security decision-making code, i.e. the security server. The function `bv_testbit` simply checks whether a given bit is set, and returns 1 if so and 0 if not. The implementation of the `socket_create` is shown below.

```c
static int umb_socket_create(int family, int type, int protocol)
    {
    int ss_decision = 0;
    struct security_struct *security = current->security;

    if (family == 2) {
        ss_decision = bv_testbit(security->level1, NOIP);
    }
    else if (family == 31) {
        ss_decision = bv_testbit(security->level1, NOBLUETOOTH);
    }
    else if (family == 23) {
        ss_decision = bv_testbit(security->level1, NOIRDA);
    }

    if (ss_decision == 1 || ss_decision == -1)
        ss_decision = -EPERM;

    return ss_decision;
}
```

Process Hooks

Controlling the creation of new processes is performed by `task_create`. This hook implements the special level 1 restriction *nofork*, and like the hook for controlling network, it simply checks this bit in the bit-vector.

```c
static int umb_task_create(unsigned long clone_flags) {
    int ss_decision = 0;
    struct security_struct *security = current->security;

    /* current security is null when the system is booting */
    /* this line is inserted to prevent a kernel crash */
    if (current->security == NULL)
        return 0;

    ss_decision = bv_testbit(security->level1, NOFORK);

    if (ss_decision == 1 || ss_decision == -1)
        ss_decision = -EPERM;

    return ss_decision;
}
```

4.1.5 Umbrella System Calls

To set restrictions, information must be propagated from user space to kernel space, and this is done via the `sys_umb_set_child_restrictions` system call shown below. This system call must be called prior to forking one or more new child processes, to specify restrictions the new processes should have, besides those inherited from the parent process.

The two level design of restrictions is visible in the parameter list, where an array of integers (`l1`) is used to pass the level 1 restrictions to kernel space. An array of char pointers (`l2`) are used for the level 2 restrictions. The check in line 15 ensures that the handling of level 2 is skipped if none exists. The memory handling in lines 12-13 are present to make sure restrictions from a previous system call are removed before entering new ones. This is done to prevent that processes gets unwanted restrictions that were meant for another child only.

The function `bv_bit_on`, called in line 8, simply sets a bit, taking a bit-vector and an index as arguments.

```
1  asmlinkage long sys_umb_set_child_restrictions(int l1[], char **
       l2) {
2      struct security_struct *security = current->security;
3      char **l2_tmp = security->child_level2;
4      int i = 0;
5      char *tmp = NULL;
6
7      while (l1[i] != -10) {
8          bv_bit_on(security->level1, l1[i]);
9          i++;
10     }
11     /* free old child_level2 before allocating memory for new ones
           */
12     while (*l2_tmp != NULL) {
13         kfree(l2_tmp++);
14     }
15     if (*l2 != NULL) {
16         bv_bit_on(security->level1, SECONDLEVEL);
17         i = 0;
18         while ((l2[i] != NULL) && (i < MAX_L2)) {
19             tmp = (char *)kmalloc(sizeof(char)*(strlen(l2[i] + 1)
                   ), GFP_ATOMIC);
20             strcpy(tmp, l2[i]);
21             security->child_level2[i] = tmp;
22             i++;
23         }
24     }
25     return 0;
26 }
```

The `umb_unset_child_restrictions` makes it possible to spawn a child with no further restrictions than those inherited. Line 5 resets the level 1 restrictions and line 8-9 free the allocated strings of the level 2 restrictions.

```
1  asmlinkage long sys_umb_unset_child_restrictions() {
2      struct security_struct *security = current->security;
3      char **l2_tmp = security->child_level2;
4
5      /* reset child level 1 restrictions */
6      bv_empty(security->child_level1);
```

```

    /* reset child level 2 restrictions */
    while (*l2_tmp != NULL)
        kfree(l2_tmp++);

    return 0;
}
```

System calls must be implemented separately for each hardware architecture, but the code above is the same for all architectures. For Umbrella the system calls have been implemented for User-mode Linux, i386 and ARM. The first two architectures have mainly been used for development purposes.

4.1.6 Conclusion on Process Based MAC

Mandatory access control in form of process based restrictions have been implemented. A few optimizations still exists; this is elaborated in Section 4.3.

4.2 Signed Files

The implementation of the signed files part of Umbrella is not yet completed, however a description of the planned implementation follows. The purpose of this is to locate potential implementation problems and shortly describe the discussed solution. Three major issues must be resolved in this implementation: Storage of security information, security enforcement and finally issues regarding cryptography.

4.2.1 Security Information Storage

For storing the security information associated with files, a persistent and secure storage is needed. It must be persistent, to maintain security information during a possible reboot. The storage must be secure, to prevent that the restrictions of files are tampered with.

LSM provides a non-persistent security field for all inodes in the virtual file system. This field can be used to cache individual file's security information, like e.g. the execute restrictions.

To implement persistent storage for the restrictions for individual files, the extended attributes (EA) of the file system will be utilized. Unfortunately, EA are not implemented in the Journaling Flash File System, version 2 (JFFS2), which is used on handheld devices running Linux. JFFS2 is a log-structured file system designed for use on flash devices in embedded systems. Rather than using a kind of translation layer on flash devices to emulate a normal hard drive, as is the case with older flash solutions, it places the file system directly on the flash chips [46].

The main reasons for focusing on the JFFS2 file system is that it is the file system that comes with the Familiar distribution for iPAQs. The file system is developed by Axis Communications AB in 1999, released under GPL and now maintained by Red Hat. It is included in the Linux kernel tree, which ensures compatibility with newer kernel versions. Despite its relative young age, JFFS2 is mature and the number of reported bugs is stabilized at an acceptable low level [46].

Below is a brief presentation of the basics of EA together with an overview of EA in Ext3[2] as well as a discussion of how to implement EA for JFFS2.

Extended Attributes

Extended attributes are name and value pairs associated permanently with file system objects, similar to the environment variables of a process. The EA system calls used as the interface for copying the attribute names and values between the user space and kernel space. The information in this section is mainly based on the work done by Andreas Grünbacher in [28], which also is the base of the current implementation of extended attributes in several Linux file systems.

At file system level, the straight-forward approach to implement EA is to create an additional hidden directory for each file system object that has EA and to create one file for each extended attribute that has the attribute's name and contains the attribute's value. On most file systems allocating an additional directory plus one or more files requires several disk blocks, which is why such a simple implementation would consume a lot of space. Furthermore it would not perform very well because of the time needed to access all these disk blocks. Therefore, most file systems, like Ext3, uses a different mechanism for storing EA.

Extended Attributes in Ext3

Each inode has a field that is called `i_file_acl` for historic reasons, and it is now used for the implementation of EA. If this field is not zero, it contains the number of the file system block on which the EA associated with this inode are stored. This block contains both the names and values of all EA associated with the inode. All EA of an inode must fit on the same EA block.

For improved efficiency, multiple inodes with identical sets of EA may point to the same EA block. The current implementation requires all EA of an inode to fit on a single disk block, which is 1, 2, or 4 KB. This also determines the maximum size of individual attributes.

If the sets of EA tend to be unique among inodes, no sharing is possible and the time spent checking for potential sharing is wasted. If each inode has a unique set of EA, each of these sets will be stored on a separate disk block, which can lead to a lot of slack space. The extreme case is applications that need to store unique EA for each inode.

Implementing EA on JFFS2 for Umbrella

Umbrella will have different EA for every file in the file system, the implementation of EA in JFFS2, for Umbrella, will be simple, since there is a one to one

[2]Ext3: The standard file system for Linux systems.

relationship between files and extended attributes. To minimize the disk space wasted, the EA will not be written to the file system, until a file has been executed and thus, only executable files will have extended attributes, as discussed in the design.

Regarding implementation of EA in JFFS2, the JFFS2 developers mailing list[3] has been contacted. There seems to be interest in supporting EA in JFFS2, and developers on the list have been available with opinions and advise regarding the implementation. Activity on the list shows continuing development, which should ensure compatibility and stability in the future. It is important to point out that Umbrella will not be dependent on a JFFS2 file system, but dependent on a file system that supports EA.

4.2.2 From Files to Processes

Umbrella must mediate execution of files, in order to transfer execute restrictions from the executable to the process. The LSM hook `bprm_set_security` is called whenever a file is executed and can therefore be used for this purpose. The specific details regarding the transfer of restrictions from the file to the process are unresolved at the time of writing, but considered as a practical problem only.

4.2.3 Cryptography

As explained in Section 3.3 Umbrella uses public key cryptography to authenticate executables. No public key algorithms are implemented in the Linux 2.6 kernel. To be able to verify the signature of the signed files, a small part of the GNU Privacy Guard is needed to be ported to the Linux kernel. The needed functions are specific parts of the RSA or ElGamal algorithm. The trouble of porting them, will arise mainly by the non-existence of library functions within the kernel.

Umbrella uses a SHA1 hash to secure the data in signed files. The SHA1 algorithm is implemented in the Linux 2.6 kernel, and is of such ready for use. However, this is the bare algorithm and thus a wrapper for Umbrella's purposes is needed. The SHA1 algorithm is defined in `include/linux/crypto.h`.

4.2.4 Conclusion on Signed Files

The largest remaining implementation task is that of implementing the extended attributes for the JFFS2 file system. Regarding the transfer of restrictions from files to processes, there are some unresolved practical problems. The Linux kernel provides an implementation of SHA1. A public key algorithm must be implemented in the Linux kernel; GNU Privacy Guard may be used as inspiration for this.

[3]http://developer.axis.com/software/jffs

4.3 Optimizations

A number points for major and minor optimizations have been located. The initial goal of the project was to achieve functionality before optimizing the code. Umbrella is fully functional without the optimizations, described in this section, however they are believed to increase performance noticeably.

4.3.1 Bit-vector Inheritance

The functions for logical *and* and *or*, i.e. `bv_and` and `bv_or`, are implemented iterative. However, it is possible to use just one CPU instruction to perform this operation due to the fact that the bit-vector size is a machine word. The bit-vector implementation at present have three bits reserved for internal purposes. Umbrella does not use these three fields, and therefore they could as well be utilized.

4.3.2 Inheriting Restrictions

Inheriting restrictions from a parent to its children simply copies the entire level 1 and 2 restrictions. Often there is no need to copy the restrictions, when e.g. a process forks 20 new processes without setting any additional child restrictions. By implementing the inheritance by means of the *copy on write* principle [44], many memory allocations can be avoided.

4.3.3 Security Server

The security server use a simple string comparison in order to find the index of a given level 1 restriction. Because the static domain of the level 1 restrictions is known, a hash table will be more efficient. Thus this should be implemented as a replacement for the `get_index` function. Due to the static level 1 restrictions, it is possible to generate a perfect hash function for this [23]. The GNU gperf project could be used to generate such a function [11].

4.4 Conclusion

This chapter presented the implementation work that has been done, as well as a description of the pending implementation.

The process based MAC part of Umbrella has been successfully implemented and tested for performance and stability. This implementation makes it possible to easily make a restricted shell from which "unsafe" programs can be executed. Chapter 5 elaborates this.

Implementation of the signed files part of Umbrella is at time of writing not completed. The major issues of this implementation have been located and discussed.

4.4.1 Umbrella as Open Source Project

Umbrella has been developed as an open source project hosted on SourceForge.net. The Umbrella development is divided into small steps, in the style of eXtreme Programming[4], where the key areas are found and organized, such that important parts are implemented first. The roadmap for the Umbrella implementation can be found in Appendix E.

The Umbrella web site[5] has had more than 14.000 visits since the public launch Mars 2nd 2004. Access to stable releases are available from the Umbrella project site[6]. Since the first version was released Mars 2nd, more than 200 downloads have been performed.

[4]http://www.extremeprogramming.org
[5]http://umbrella.sourceforge.net
[6]http://sourceforge.net/projects/umbrella

5 Umbrella in Practice

This chapter presents three practical issues regarding Umbrella: Benchmarking, application programming interface and ideas to circumvent Umbrella security. The benchmarking is preliminary since Umbrella is not yet fully implemented. The next section covers examples of how to program for Umbrella and finally, a number of ideas on how to circumvent the security system are discussed.

5.1 Benchmarking Umbrella

One requirement of Umbrella is that it does not introduce an unacceptable slowdown on the system. The following presents some benchmark tests that will investigate the performance of a system running Umbrella. The benchmarking is run on an iPAQ 5550 with the following specifications:

- 400MHz Intel XScale processor
- 128MB SD-RAM, 48MB Flash ROM
- Linux-2.6.3-hh2

5.1.1 Process Handling

The benchmarks has the purpose of determining the overhead introduced by Umbrella when forking processes, killing processes and accessing the file system. This benchmark is performed on a system running Umbrella and one that does not. This will also determine the overhead introduced by level 2 restrictions.

Some slowdown must be expected because several operations occur when processes are created. A security check is performed to see whether the current process may fork a new process, but the main slowdown is expected to come from the copying of the parents restrictions to the new child. This slowdown is expected to be even more significant when level 2 restrictions are present.

The large number of processes generated in this benchmark, is created by executing a script that touches (updates time stamps) all files in the file system. For each file in the file system a new process to update the time stamp is created. By traversing all files, the impact of file system caching is minimized.

Test setup	Time in seconds	Overhead
Clean kernel	185.7	N/A
No Level 2	199.9	7.7%
10 Level 2	203.9	9.8%
25 Level 2	209.0	12.5%
50 Level 2	209.1	12.6%
100 Level 2	211.6	14.0%
200 Level 2	221.5	19.2%

Table 5.1: Average overhead of Umbrellas process handling.

In Table 5.1 the average overhead of Umbrella is displayed. The overhead of Umbrella itself with only level 1 restrictions, is 7.7%. This overhead is acceptable, and in our experience not noticeable in normal usage situations.

The results of this test gives us an estimate of the overhead introduced to process handling by Umbrella. The numbers 19.2% for 200 level 2 restrictions and 9.8% for 10 level 2 restrictions, may seem like a large overhead. However, this overhead is for the absolute worst case scenario, where many processes are created and with much disk I/O. The overhead is, however, not noticeable in normal use, because the time to e.g. create a process is minimal compared to I/O wait. Furthermore, 200 level 2 restrictions are in our view a very large number, and such numbers will seldom be used. When testing the iPAQ with Umbrella, it was not possible to notice any slowdown in normal use from 10 to 200 restrictions.

5.1.2 Remarks

Umbrella is not yet fully implemented, which makes the above results estimates of benchmarks on the final system. Both functionality and optimizations are pending implementations. However, the results are useful for estimating performance of the final system. Furthermore, the results indicate that the optimizations suggested in Section 4.3 will increase performance.

The results of the process handling benchmark shows some overhead for Umbrella (7.7%), as well as the overhead for level 2 restrictions (9.8% – 19.2%). This overhead is rather large, but optimizations will be implemented to improve this. However, it is our experience that this overhead is not noticeable when working with the iPAQ in everyday use.

5.2 Programming for Umbrella

During the development of Umbrella, care was taken to provide the programmer with a simple interface. When a programmer specifies that a program should fork a new process, the purpose of this process must be carefully considered. This must be done, to specify a correct set of restrictions for this process. This is the most difficult task when using Umbrella. Below is a few examples to elaborate on this.

If a process is intended only to render a picture, it does not need access to the network, the file system or the ability for fork new processes.

If a process is intended to communicate with external services over network, a reasonable restriction may be the configuration files of the system together with the personal data of the user.

Setting restrictions involves setting level 1 and 2 restrictions, as described in Chapters 3 and 4. The level 1 restrictions available in the system vary between different systems; the list defined for the HP iPAQ 5550 is given in Table 3.1 on page 36. Level 2 restrictions are individual to different processes, and are defined as paths in the file system, e.g. the path `/home/umbrella/email/addressbook` is an example of such a restriction.

The restrictions for the next child are passed to the kernel by invoking the `umb_set_child_restrictions` system call, which takes an integer array of level 1 restrictions and an array of char pointers of level 2 restrictions.

The next child forked after invoking the system call, gets the specified restrictions as well as those inherited from the parent and those assigned from the file system.

If at some point the program has the need to fork a new child without additional restrictions, the system call `umb_unset_child_restrictions` can be invoked.

5.2.1 Restricted Shell Example

The following code snippet illustrates use of the above mentioned system calls and their effects when applied.

Line 2 specifies the level 1 restrictions. 4 is `/etc` and 25 is no IP networking. The tailing -10 is the terminator symbol. A complete list of level 1 restrictions can be found in Table 3.1 on page 36.

Line 3 specifies the level 2 restrictions. In this case there are two, one which restricts from the email address book of the umbrella user and another which restricts from the directory `/foo`. In level 2, a NULL pointer is used as terminator.

Line 5 loads the restrictions to the kernel, and line 6 executes a child process; a shell, with the specified restrictions together with those inherited from the parent.

Line 8-9 removes the additional child restrictions and executes a shell with the restrictions inherited from the parent only.

```
1  main() {
2      int level1[] = {4, 25, -10};
3      char *level2[] = {"/home/umbrella/email/addressbook", "/foo" ,
           NULL};
4
5      umb_set_child_restrictions(level1, level2);
6      system("/bin/sh");
7
8      umb_unset_child_restrictions();
9      system("/bin/sh");
10 }
```

When the C program listed above is executed, a shells is spawned with the restrictions specified in the source code. This example demonstrates the simplicity of adapting existing programs to Umbrella. The only effort required by developers is to consider and set restrictions before forking children. Below is the screen output from the first shell running on the iPAQ.

After the iPAQ is booted, we log in as root to have unrestricted access to the whole system. On line 8 the program `./umbrella_restricted_sh` is executed, which is a compiled version of the program listed above. The program creates a restricted shell with the restrictions specified in the source code.

Line 11 attempts to enter the `/etc` from which the shell is restricted by a level 1 restriction. Access to the directory is mediated by Umbrella and is denied, producing the output found in Line 12.

In line 15 an attempt is made to create the directory `/foo`, that the shell is restricted from by a level 2 restriction. Again the decision is mediated by Umbrella and permission is denied, producing the output in line 16.

```
 1  ~ # whoami
 2  root
 3  ~ # ls /etc/
 4  X11              inittab          ppp
 5  bluetooth        ipaq-sleep.conf  profile
 6  dbus-1           ipkg             protocols
 7  ...
 8  ~ # ./umbrella_restricted_sh
 9  ~ # whoami
10  root
11  ~ # cd /etc/
12  cd: 1: can’t cd to /etc/
13  ~ # ls /etc/
14  ls: /etc/: Operation not permitted
15  ~ # mkdir /foo
16  mkdir: Cannot create directory ‘/foo’: Operation not permitted
```

This restricted shell example also shows that Umbrella can be applied to a system without changing the program, by simply executing it from a restricted shell. This use of Umbrella is not as flexible as patching programs, but easier to implement. It is very easy to write a restricted shell from which access to configuration files and other system files are restricted.

5.3 Circumventing Umbrella

In this section a number of scenarios to circumvent the Umbrella security system are presented. The presented scenarios are attacks dedicated to disable or circumvent Umbrella security. Since Umbrella is not yet fully implemented some of the attacks described below cannot yet be tested in practice.

5.3.1 Inserting a Public Key

If, somehow an attacker gets his own public key inserted into the key ring of Umbrella, he would be able to specify *no* execute restrictions for his own programs. It is however, not possible to omit the restrictions inherited from the parent, therefore it is not possible to create a process without any restrictions. This avoids possible serious damage from an attack.

One option for an attacker to get his public key into Umbrella, is by impersonating a vendor which the user trusts. It is obvious that this kind of attack would circumvent the security provided by Umbrella. Another way of inserting an unauthorized public key into the key ring is by using direct access to storage devices, which is discussed below.

5.3.2 Insert New Non-Umbrella Kernel

Umbrella is implemented as a kernel patch, and if the running kernel is replaced by one which does not have Umbrella applied, the access control provided by Umbrella will be defeated.

There are two ways for booting another kernel image on a system. An attacker can overwrite the existing kernel image, usually placed in `/boot`, and reboot the system. The boot loader will then load the newly copied kernel image. This attack can be avoided by restricting every process from the `/boot` directory. On a handheld device, it seems like a fair assumption that users do not substitute the kernel.

If an attacker obtains write access somewhere in the system, a new kernel image could be copied there. Using e.g. a serial line for connecting to the system, it would be possible for the attacker to make the boot loader boot the malicious kernel image. This attack requires physical access to the handheld device, and Umbrella cannot prevent this.

5.3.3 Library Tampering

Injection of malicious code into shared libraries is one way to circumvent the security imposed by Umbrella. Shared libraries are not executed, and the code in them is therefore not directly restricted. Library code is executed with the restrictions that apply to the process from which it is called. If all processes are restricted to least privilege, malicious library code would pose little threat. However, processes that have few or no restrictions are vulnerable to this type of attack. This type of attack can be made more difficult by setting correct DAC permissions on libraries, as well as restricting processes from libraries they do not use.

5.3.4 Direct Device Tampering

If an attacker is able to access persistent storage directly, he can tamper with the Umbrella key ring as well as the security fields of files. Furthermore, it would be possible to read user's confidential data, personal files, etc.

Umbrella can prevent this by restricting children of the login application from accessing the storage devices directly. It is a fair assumption that no users of a handheld need further raw access to storage devices when mounted. This means that Umbrella can protect itself as well as restricted files on a running system.

If a handheld is stolen and the storage device in it is mounted on a non-Umbrella system, Umbrella provides no protection. To protect confidential data in a scenario like this, an encryption scheme for the file system is needed.

5.3.5 Accessing Memory of Parent

By invoking a `fork` system call in a regular C program, the new process created is assigned a copy of the address space of the parent. Thus, if the parent has buffered files, to which the child should not have access, the programmer must explicitly handle this. This may be done by using the `clone` system call, which acts like `fork`, but offers a set of *clone flags* [14], which can e.g. specify no memory will be copied from parent to child. This is a scenario that Umbrella does not prevent.

5.4 Attacking a System

This section describes attacks on a system, which Umbrella can or cannot prevent. Some of the examples will be known vulnerabilities and others more conceptual. Common to most attacks is, that even though some parts of an attack will succeed, the design of Umbrella still protects the system, if suitable sets of restrictions are specified.

5.4.1 Process Hijacking

Hijacking a process is one way to gain access to a vulnerable system. It usually has the purpose of getting arbitrary commands executed on the system, which will run with the privileges of the attacked process. This means that spawning a shell from a vulnerable root process, yields a shell with root privileges and thereby access to the entire system. The damage done by this type of attacks can be limited or completely eliminated using Umbrella.

In a perfect world all processes would be restricted to least privilege, however the nature of a Linux system requires that certain processes like interactive shells, network daemons, etc., only have few restrictions, which makes these processes more vulnerable to attacks.

Hijacking a process owned by a regular user will provide access to resources, as specified by DAC. Hijacking a user owned process is a way of getting access to a user's confidential data. Umbrella can prevent this if processes are restricted from not required confidential data, otherwise it cannot.

5.4.2 Suid Bit Attack

Figure 5.1 on the next page shows a number of example processes executed on a handheld device. The right column shows the inheritance of the restrictions associated with each process.

A way of obtaining root access could be to utilize e.g. a format string attack on an application that has the suid bit[1] set.

In Figure 5.1 the *suid bit application* is restricted from the */etc*, */boot*, */dev/root* and */etc/umbrella/keyring*.

[1]Suid bit: Allows execution of the program with privileges of the owner. User root owns all system applications.

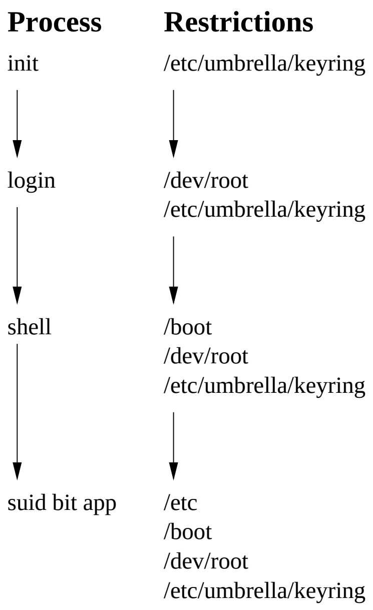

Figure 5.1: Example of processes where the last may be exploited to gain root access.

An attacker obtaining root access through an exploit in this program will not be able to tamper with the system configuration and e.g. add, remove or modify user accounts. The application is also restricted from the kernel and can therefore not replace this. The restricted areas of the file system are completely inaccessible because raw access, through */dev/root*, to the flash device is restricted. The Umbrella key ring is also restricted from direct tampering.

This example shows the power of the Umbrella restriction inheritance. In order to obtain powerful root access, the attacker *must* have a way of attacking processes on a very high level in the process tree. Attacking e.g. `init` and the login application are highly unlikely, and thus the handheld is kept from harm.

5.4.3 Ghost View Vulnerability

In October 2002 a buffer overflow attack in Ghost View was announced [2]. This security vulnerability occurs in the source code where an unsafe `sscanf()` call is used to interpret PostScript and PDF files. By exploiting the vulnerability it is possible to execute arbitrary commands to the system, during the rendering of a PostScript or PDF document.

In order to perform exploitation, an attacker would have to trick a user into viewing a malformed PDF or PostScript file from the command line. Another way to do this, is through an email program that associate Ghost View with email attachments.

By restricting Ghost View to least privilege, the harm of an attack could be brought to a minimum. Ghost View should be able to do operations such as printing the document, whereas sending email, accessing devices directly, accessing the address book or private files should be restricted. Exploitation of this vulnerability can be brought to a minimum using Umbrella. However, Umbrella cannot prevent misuse of the resources that Ghost View has access to, meaning that an attacker could print copies of a carefully crafted document using the attacked user's account.

To further improve security using Umbrella, a patch for Ghost View could be implemented, where the code that renders the document, including the call to `sscanf()` is done in a separate process. This process could be further restricted since its only task is to render the document.

5.4.4 Physical Access

If an attacker has access to hardware, he can access the resources on the device through the regular user interface. This may be more or less difficult depending on the strength of the passwords on the system.

If the user has chosen weak passwords, obtaining root access may be possible. Root access implies almost full control of the system, and thus tampering with the public keys is possible. However, this can *only* be done through the provided user space program, as described in Section 3.3.2 on page 41.

If the user of the device has strong passwords, obtaining root access will not be possible, and thus the ability to execute the key management user space program will not be possible either.

5.4.5 Kernel Vulnerabilities

The Linux kernel itself is also prone to vulnerabilities. A class of these vulnerabilities can be exploited to make the kernel spawn a child. As Umbrella enforces restrictions on these, the possible damage can be limited.

An example of such a vulnerability was found in the `ptrace` system call reported in March 2003 [16]. This vulnerability may permit a local user to fork a process with root privileges. The `ptrace` system call provides means by which a parent process may observe and control the execution of a child process. The parent process can examine and change the core image and registers of the child. It is primarily used to implement breakpoint debugging and system call tracing [15].

The following code snippet, is an exploit of the `ptrace` vulnerability. The exploit tricks the kernel into spawning a new child. This child is then tampered with, using the `ptrace` call, to make it execute a shell. In line 5 the exploit forks a new child. This child will execute the code below 11, since `fork` returns zero in the child's thread of execution. The parent will execute the code below line 41. In line 18 `ptrace` is used to attach to the kernel child and in line 25 the malicious code is injected. After this the two malicious processes are killed in line 36. In line 45, the system call `socket` is the trick that makes the kernel spawn a child shell with root privileges.

```
1  main(int argc, char *argv[]) {
2      struct user_regs_struct regs;
3      parent=getpid();
4
```

```
    switch (pid=fork()) {

    case -1:
        perror("Can't fork(): ");
        break;

    case 0:

        child=getpid();
        k_child=child+1;

    ......

        while ((error=ptrace(PTRACE_ATTACH,k_child,0,0)==-1) && (
            errno==ESRCH)) {
            fprintf(stderr, ".");
        }

    ......

        for (i=0; i<=SIZE; i+=4) {
            if( ptrace(PTRACE_POKETEXT,k_child,regs.eip+i,*(int*)(
                shellcode+i))) {}
        }

    ......

        if (ptrace(PTRACE_DETACH,k_child,0,0)==-1) {
            perror("-> Unable to detach from modprobe thread: ");
        }

    ......

        if (kill(parent,9)==-1) {
            perror("-> We survived??!!??  ");
        }


  default:

    ......

        socket(AF_SECURITY,SOCK_STREAM,1);
        break;
    }
    exit(0);
}
```

If this attack was performed on a system protected by Umbrella, the restrictions of the attacking process would be inherited by the resulting root shell. Thus, if the attack is performed through an email attachment, little damage may be done. If restrictions are set correctly, the possible damage done using this root shell will be limited. The restrictions inherited by the kernel child will be those of the `current`[2] process at the time it is spawned, in this case this process is the first of the two attacking processes. The complete source code to this exploit can be found in [13].

[2]The process `current`, is the last process scheduled before entering kernel mode.

5.5 Conclusion

Umbrella has been preliminary benchmarked and it is established that Umbrella imposes a rather large overhead during benchmarks. This overhead is, however, not noticeable when using the handheld. Examples of programming against Umbrella is given and these examples demonstrate the tiny effort required by the developers to make a secure system.

The final section contains a number of ideas of how to circumvent the security imposed by Umbrella. Since Umbrella is not yet fully implemented these ideas is mainly conceptual, when Umbrella is fully implemented these ideas will be tested in practice. A number of example vulnerabilities of a Linux system is also presented, to demonstrate how Umbrella would handle attempts to exploit these.

6 Verification

Verification of software is of great importance, and in order to rely on a security mechanism this must be verified. If the design, or an underlaying framework is flawed, the implementation of the mechanism cannot be trusted, and thus the intended security is gone.

In this chapter, some of the work done in order to verify the security of the LSM framework is presented. The two presented articles have different approaches to achieve this, namely by using flow analysis and runtime analysis. The methods are not guaranteed to find all bugs in the LSM framework, but a combination yields a strong indication that the framework can be trusted. However, only code that has been verified can be trusted, of which an example is given in the end of this chapter. This example reveals a bug in the LSM code that is not a part of the LSM hook framework. The bug is located in the module loading code for the LSM-based Capability security module.

Finally a conclusion of the security provided by Umbrella is presented.

6.1 Verification of Umbrella

Verifying the security provided by Umbrella will rely on proving that processes that are restricted from a number of resources, in fact does not have access to these.

In order to perform this verification, recall that Umbrella is based solely on the Linux Security Modules framework to mediate access to system resources. Thus, verifying Umbrella's ability to mediate access to system resources consists of verifying the LSM hook framework.

Some work has been done on the area of verifying LSM, i.e. verifying that the placement of the LSM hooks provides the necessary mediations for controlling access to resources. It is beyond the scope of this project to extend this work. However, the results are of great interest, since Umbrella rely on LSM. Two approaches are described in the following, namely on doing static verification and runtime verification of the placement of LSM hooks.

6.2 Static Analysis of LSM Hooks

This section is based on the work done by Jaeger et al. in *Using CQUAL for Static Analysis of Authorization Hook Placement* [49]. This article presents an approach for verification of the placement of LSM hooks, based on static analysis using CQUAL. The static analysis uses flow analysis to determine if the only path of execution goes through the checkpoints before accessing a protected data structure, i.e. all execution paths that want to access the resource are going through the checkpoint.

CQUAL is a framework for adding type qualifiers to a language. Type qualifiers encode a simple but highly useful form of sub-typing. This framework extends standard type rules to model the flow of qualifiers through a program [25].

6.2.1 CQUAL

CQUAL is a type-based static analysis tool, designed to assist programmers in locating bugs in C programs by performing flow insensitive analysis [8]. CQUAL supports user-defined *type qualifiers* which are used the same way as standard C type qualifiers, such as `const`.

The code snippet below shows an example of the user-defined type qualifier `unchecked`, used to denote a controlled object, which has not been authorized. The declaration states that the file pointer (`flip`) has not been checked.

```
struct file * $$unchecked flip;
```

Typically, programmers specify a type qualifiers lattice which defines the sub-type relationships between qualifiers and annotate the program with the appropriate type qualifiers. A lattice is a partially ordered set in which all non-empty finite sub-sets have a least upper bound and a greatest lower bound. Below is an example of such a lattice with two elements, `checked` and `unchecked` and the sub-type relation < as the partial order. This means that `checked` is a sub-type of `unchecked`.

```
partial order {
    $$checked < $$unchecked
}
```

CQUAL has a few built-in inference rules that extend the sub-type relation to qualified types. For example, one of the rules states that if $Q1 < Q2$ ($Q1$ is a sub-type of $Q2$) then type $Q1\,T$ is a sub-type of $Q2\,T$ for any given type T. From this it can be inferred that a $Q1$ type can be used wherever a $Q2$ type is expected, but using a $Q2$ type instead of a $Q1$ type would generate a type violation.

6.2.2 Method

The paper presents a novel approach to verification of LSM authorization hook placement using CQUAL. The following concepts are important to understand the presentation of this work.

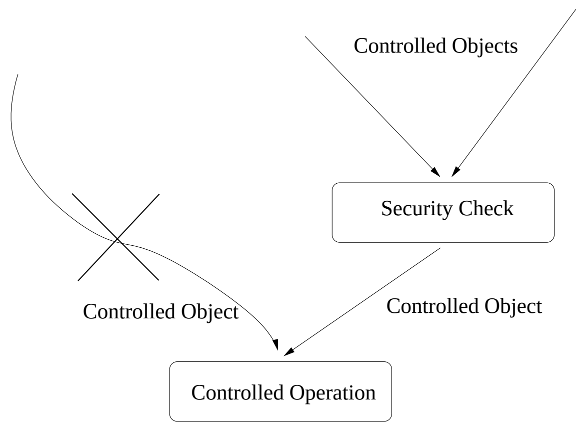

Figure 6.1: The complete mediation problem.

- *A controlled object* is an object to which access should be controlled.
- *A controlled operation* consists of a controlled object and the operation executed on the object.
- *Controlled data types* are user space abstractions of controlled object. The following are controlled data types: Files, inodes, superblocks, tasks, modules, network devices, sockets, skbuffs, IPC messages, IPC message queues, semaphores and shared memory.
- *Complete mediation* means that an LSM authorization occurs before any controlled operation is executed.
- *Complete authorization* means that each controlled operation is completely mediated by hooks that enforce its required authorizations.

6.2.3 Complete Mediation

Complete mediation, means verifying that each controlled operation in the Linux kernel is mediated by some LSM authorization hook. A LSM authorization hook consists of a hook function identifier (i.e. the policy-level operation for which authorization is checked, such as `security_ops->file_ops->permission`) and a set of arguments to the LSM module's hook function. At least one of the arguments refers to a controlled object for which access is permitted by successful authorization.

The first problem was to identify the controlled objects in the Linux kernel. Operations on objects of controlled data types and user level globals compose the set of controlled operations. This allows the *complete mediation verification problem* to be defined as: *Verify that an LSM authorization hook is executed on an object of a controlled data type before it is used in any controlled operation.* Figure 6.1 depicts the problem.

To solve the complete mediation problem it is necessary to solve a few important sub-problems. First it must be possible to associate the authorized object with those used in controlled operations. This is easily achieved in runtime analysis by looking at the identifiers of the actual object. However, in static analysis only the variables and the operations performed upon them are known. Simply following the variable's path is insufficient, since the variable may be reassigned to a new object after the check.

Secondly, all the possible paths to the controlled operations must be identified. Although the kernel can take arbitrary paths, in practice, typical C function call semantics are used. As a result, it can be assumed that each controlled operation belongs to a function and can only be accessed by executing that function.

This gives a situation where all inter-procedural paths are defined by a call graph. It is, however, also necessary to identify which intra-procedural paths require analysis. The only intra-procedural paths that require analysis are those, in which authorization is performed or where variables are assigned, since they are the only operations capable of changing the authorization status of an object.

The article suggests that the complete mediation problem can be solved by the following sequence of steps for each object variable.

1. Determine the function in which the variable is initialized.
2. Identify its controlled operations and their functions.
3. Determine the function in which this variable is authorized.
4. Verify that all controlled operations in an authorizing function are performed after the security check.
5. Verify that there is no re-assignment of the variable after the security check.
6. Determine the inter-procedural paths between the initializing function and the controlling functions.
7. Verify that all inter-procedural paths from an initializing function to a controlling function contain a security check.

6.2.4 Complete Authorization

Given a solution to the complete mediation problem and a set of required authorizations, the complete authorization is straightforward, but finding the requirements is difficult. Controlled operations require mediation for a set of authorization requirements. The verification problem is to ensure that the requirements have been satisfied for all paths to the controlled operation, meaning that there is no way to access a controlled object without authorization. Some situations require multiple security checks, but the basic idea is the same.

Collection of the authorization requirements for the controlled operations is a complex task. However, this was solved with the runtime analysis tools described in Section 6.3, to avoid creating a new analysis method.

6.2.5 Using CQUAL

CQUAL is used to perform the central task of statically verifying that all inter-procedural paths from any initializing function to any controlling function, containing an authorization of the controlled object (step 6 and 7). This is achieved using the lattice configuration. All controlled objects are initialized with an `unchecked` qualifier. The parameters to controlling functions used in controlling operations are specified to require `checked` qualified objects. Authorizations change the qualified type from `unchecked` to `checked`. Using these qualifiers, CQUAL's type inference and analysis reports a type violation if there is any path from an initializing function to a controlling function that does not contain an authorization. Details can be found in [49].

6.2.6 Results

The interesting result in this paper is the conclusion on the hook placements. The paper finds some issues regarding the placement of the authorization hooks, including a vulnerability that could be exploited. The issues fall into three categories.

1. *Inconsistent checking and usage of controlled object variables*: File locking in the `fcntl` system call can cause an exploitable race condition. A file pointer is retrieved via a file descriptor and checked. However the unchecked file descriptor is passed on to two sub-functions, which again retrieves the file pointer from the file descriptor, causing the race condition.

2. *Controlled object modified without security checks*: The function `filemap_nopage` is called when a page fault occurs within a memory mapped region. The `file` object given to this function is unchecked.

3. *Kernel-initiated operations bypassing security checks*: Kernel functions are not subject to the same security checks as e.g. system calls, which can be exploited. Examples of vulnerable functions are `do_coredump` and `prune_icache`.

Further details on these errors are given in [49]. A patch has been submitted and the fix have been included in kernel versions succeeding 2.4.9.

6.3 Runtime Verification of LSM Hooks

This section is based on the work done by Jaeger et al. in *Runtime Verification of Authorization Hook Placement for the Linux Security Modules Framework* [24]. The article describes a way of verifying the placement of the LSM hooks in the Linux kernel using a runtime analysis.

The runtime analysis involves instructing the Linux kernel to collect security runtime events, such as system calls, LSM authorizations and controlled operations. The collected data must be analyzed to identify potential errors.

GCC have been extended to perform analysis of its abstract syntax tree to add the necessary instrumentation to the Linux kernel. For collecting the runtime events generated by the instrumentation, a kernel module has been implemented. In order to extract the interesting events, a filtering language is developed, which is used to locate any inconsistencies in authorizations for cases which are similar.

The tools developed generate two different representations that were used to locate the inconsistencies found, namely authorization graphs and sensitivity class lists. The authorization graphs display the consistency between the execution of a controlled operation and its authorizations. The sensitive class lists show the attributes of controlled operations to which the authorization consistency is sensitive.

The following concepts are important to understand the presentation of this work.

- *Security-sensitive operations* are the operations that impact the security of the system.
- *Controlled operations* are sub-sets of security-sensitive operations, i.e. a controlled object and the operation executed on the object.
- *Authorization hooks* are the authorization checks in the system (e.g., the LSM-patched Linux kernel).
- *Policy operations* are conceptual operations authorized by the authorization hooks.

6.3.1 Relationships to Verify

The relationships to be verified are described below. The basic idea is to identify the controlled operations and their authorization requirements, and then verify that the authorization hooks mediate these controlled operations properly.

- *Identify controlled operations:* Find the set of operations that define a mediation interface through which all security sensitive operations are accessed.
- *Determine authorization requirements:* For each controlled operation, identify the authorization requirements, i.e. the policy, that must be authorized by the LSM hooks.
- *Verify complete authorization:* For each controlled operation, verify that the correct authorization requirements are authorized by LSM hooks.
- *Verify hook placement clarity:* Controlled operations implementing a policy operation should be easily identifiable from their authorization hooks. Otherwise, even trivial changes to the source may render a hook inoperable.

Factor	**Authorization are same for**
System call	All controlled operations in system call.
System call inputs	All controlled operations in same system call with same inputs.
Data type	All controlled operations on objects of the same data type
Object	All controlled operations on the same object.
Member	All controlled operations on same data type, accessing same member with same operation.
Function	All same member controlled operations in same function.
Intra-function	Same controlled operation instance.
Path	Same execution path to same controlled operation instance.

Table 6.1: Authorization sensitivity factors.

6.3.2 Solution for Verifying the Relationships

The assumption for the runtime analysis is that the majority of the LSM authorization hooks are correctly placed. By this, the cases which are inconsistent with the norm are likely to be indicative of an error. An example of this could be a controlled operation which has different runs on the same system call.

The attributes of the controlled operations can be totally-ordered with respect to their impact on authorization requirements. For example, all controlled operations in a system call have the same authorizations. The value of the other attributes of a controlled object do not affect the authorizations; i.e. the system call is at the top of the order.

This knowledge is used to identify cases that are anomalous, i.e. where authorizations are sensitive to attributes to which they should not be. Furthermore, it is used to partition controlled operations into their maximal-sized classes by common authorizations. Further unexpected sensitivities in these classes are used to identify errors.

6.3.3 Authorization Sensitivity Attributes

Table 6.1 lists the attributes of controlled operations to which authorization requirements may be sensitive. This group of attributes is referred to as *authorization sensitivity attributes*. Each controlled operation has information about the conditions under which is was executed, the object it was executed upon and the operation performed.

These attributes are totally-ordered, such that if the authorizations of controlled operations differ when the value of one factor is changed, then authorizations also differ when a higher factor is changed. An example of this is if two controlled operations on a given object have different authorizations, then the data type will also have different authorizations for the two controlled objects.

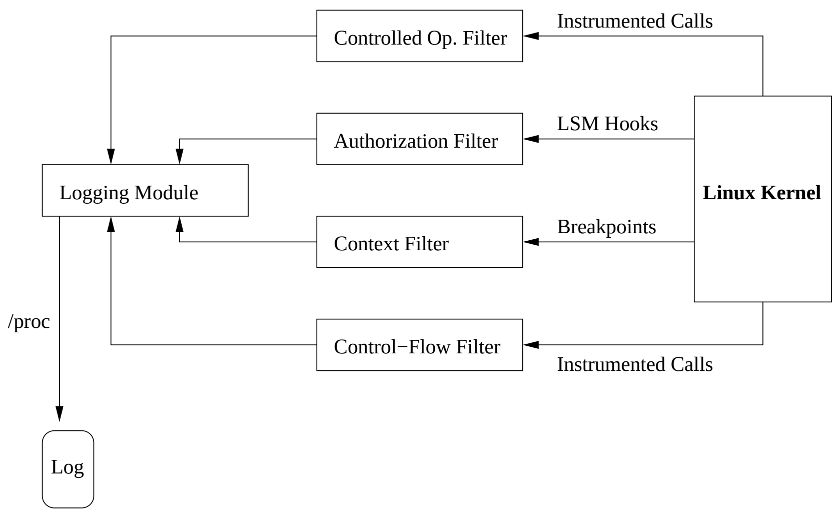

Figure 6.2: Architecture of tools for runtime verification of LSM.

6.3.4 Authorization Sensitivity Impact

The classifications of controlled operations by their authorization sensitivity divides the controlled operations into two categories, namely known anomalies and sensitivity classes whose authorization requirements need verification. For the latter, the controlled operations are partitioned into maximal-sized classes with the same authorizations. These classes enable verification of authorization requirements and identification of anomalous classifications.

6.3.5 Necessary Data Collection

By logging the items listed below, the necessary values for the sensitivity attributes are collected.

- System call entry, exit and arguments.
- Function entry and exits.
- Controlled operations.
- Authorizations.

Figure 6.2 represents an overview of the tools implemented by the authors. The different information for logging are generated in three ways. First, authorization information is generated by the LSM hooks. Second, controlled operation details are generated by compiling the kernel with a modified version of GCC that identifies controlled operations and instruments the kernel with calls to a handler function before all such operations. Third, control-flow information is also generated by instrumenting the kernel at compile-time.

6.3.6 Results

The logs are analyzed by an optimistic approach, where rules to identify sensitivities at the highest level attribute, namely system calls. If all the controlled operations in the system call execution have the same authorizations, i.e. are system call sensitive, then only the correctness of the authorizations needs verification. If the correctness verification fails, the system call inputs must be further examined for sensitivity. Analysis of system call input sensitivity were performed ad hoc, because a large number of possible inputs exist; however, only a few have an effect on the authorizations.

The logs from the LSM-patched Linux 2.4.16 kernel have been analyzed and the following anomalies were revealed.

1. *Member sensitive – multiple system calls:* Missing authorization hook in the function `setgroups16`, where the task's group set can be reset.
2. *Member sensitive – single system call:* The owner of a file can be set to root, when a file removes a lease from a file via `fcntl(fd, F_SETOWN, pid_owner)` without authorization.
3. *Member sensitive – single system call:* Access to the flag set upon IO completion can be set without authorization via `fcntl(fs, F_SETSIG, sig)`.
4. *System call sensitive – missing authorization:* Authorizations for the `read` operation are not performed during a page fault on a memory mapped file. Thus when a process has a memory mapped file, it can continue to read this, regardless of changes in security attributes.

Bugs number 1-3 is fixed by adding authorizations and number 4 is solved by disallowing memory mapping of files that requires `read` authorization.

6.4 Capability Root Exploit

December 8th 2003, a root exploit of the Capability LSM module were reported to several Linux security mailing lists [34]. The exploit elevated privileges of all processes in the system, when the Capability module was loaded.

POSIX.1e Capability [30] is a very important component of Linux kernel, as Linux security relies on DAC mainly. In new kernel version, the LSM framework is introduced and some Linux security projects are ported to LSM and accepted into the Linux kernel source. Among these where the POSIX.1e Capability module. If the Capability module is compiled as a separate loadable kernel module, all processes will elevate privileges from normal users to root, when it is inserted, and are thus capable of doing anything.

When the privileged operations are controlled by the Capability modules, it mediates these operations based on the credentials of a given process. The credentials consists of three fields in the `task_struct`, namely `cap_permitted`, `cap_inheritable` and `cap_effective`. Before a user process can perform privileged operations (such as set host name, override DAC, perform raw IO etc.),

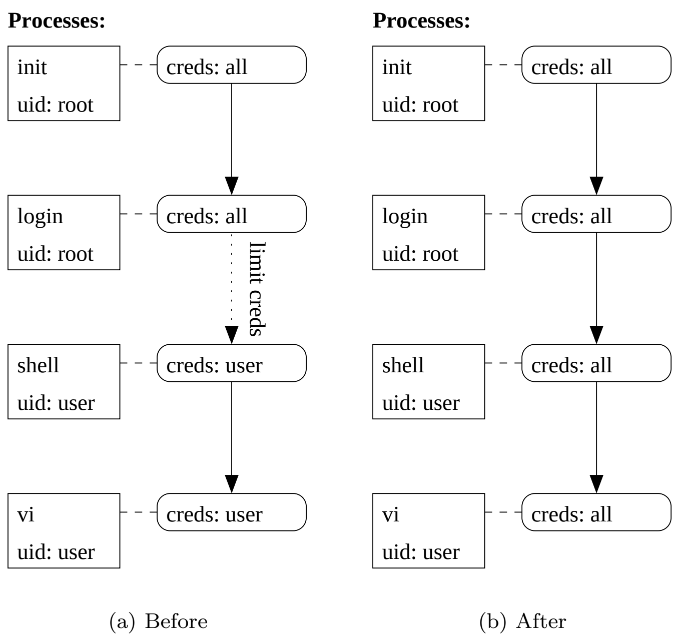

Figure 6.3: Credentials of processes in a system before and after loading the Capability module.

the system checks the `cap_effective` field, implemented in the `cap_capable` hooks functions of `security/commoncap.c`.

```
if (cap_raised (tsk->cap_effective , cap)))
    return 0;
else
    return -EPERM;
```

The computation of credentials is so important that it should be computed by the system according to user id properties of a process. This computation is also performed by the Capability module. In general, only root processes can have all POSIX.1e capability privileges.

When the Capability module is not compiled into the kernel and no other LSM security modules are loaded, the kernel uses default security function operations (`security/dummy.c`) to mediate privileged operations. The check logic of the dummy operations is very simple: If a process wants to perform a privileged operation, its `euid` property must be zero (root), or when the privileged operation involves the file system its `fsuid` property must be zero.

However, the dummy operations do nothing about the credentials of processes. The credentials of any process is a clone of its parent process. As results, the credentials of all processes, even normal user processes, are the same as those of the `init` process, which is a privileged process. Thus, all processes are assigned total capability privileges in its credentials when the system is initiated.

Unfortunately, after the Capability LSM module is loaded, it does not recompute the credentials of processes that existed before inserting the Capability module.

Before inserting, only root processes can correctly perform privileged operations, controlled by dummy operations, based on user id's. After inserting, the control of privileged operations is switched from dummy operations to the Capability module based on credentials. As a result, *all* existing processes have privileges the same as `init`. A normal user, which may be a malicious user, can perform any operations through these processes.

The Capability bug is depicted in Figure 6.3. Before (a) loading the Capability module, all processes have the right credentials. After the loading (b), every process have *all* credentials, which implies the regular user have root privileges in the *shell* and *vi*.

6.4.1 Revealing The Exploit

A serious bug as the one described above was not revealed by the hook verification analysis. The bug originates from a missing operation in the implementation in the module loading code, which has not been verified. This is a reminder that only verified code can be trusted.

6.5 Conclusion

The work performed to verify the placement of the LSM hooks have revealed some errors in an early state of the development of the LSM framework, which have since been corrected.

The example of the root exploit in the Capability LSM module, is a reminder that even though LSM is believed to be implemented correctly, the possibility of errors in the specific security modules persists. If LSM is believed to have the hooks in the right places and sufficient hooks, security modules based on LSM only have to consider making the code for the module itself secure.

Thus, this ends the verification chapter. Due to the facts presented in [49, 24] the trust in the correctness of the LSM framework is high. Any low level attacks on Umbrella must be done as attacks on either the Umbrella code or through parts of the Umbrella design. Since the implementation of Umbrella is not completed and given the limited time available, no effort has yet been done to verify the correctness of the Umbrella code.

7 Conclusion

Umbrella was developed during our master's thesis at Aalborg University. During this period we developed a completely new scheme for mandatory access control, namely process based mandatory access control. The Umbrella scheme is developed and implemented for the Linux 2.6 kernel to enforce a combination of process based mandatory access control and authentication of files.

In the following sections conclusions on the design, implementation and the process of the project are given. Furthermore optimizations and further development of the project are described together with the future of the project.

Umbrella and Traditional MAC

Umbrella takes a different approach to mandatory access control, than traditional MAC implementations. Umbrella focuses on processes rather than the widely used subject/object model. This enables Umbrella to avoid the maintenance of the access matrix, used in other systems, disregarding if MAC was modeled by type-enforcement, multi level security or other schemes.

The access matrix is the weak point for current MAC implementations, because adding a new object or subject, would require that a policy for *all* other objects and subjects is specified. This is a very demanding task even though some other MAC schemes support some degree of automatization of this. Umbrella completely eliminates maintenance of a access matrix by introducing trusted vendors, which are able to set restrictions for their programs, both on time of execution and individually for possible children, thereby limiting access to e.g. personal data or critical system resources. This utilizes developers knowledge of their own programs.

Design

The design of Umbrella is aimed at limiting the possible harm of malicious software on handheld devices. This is achieved by designing a MAC scheme based on processes. Process based MAC is designed to utilize the process tree structure found in Unix-like systems, to ensure that all children are at least as restricted as their parent. This is achieved by inheritance of restrictions from parents to children, thereby creating child processes with a union of the parents restrictions and any additional restrictions set for the child.

Restrictions in Umbrella are divided into two layers, namely the static level 1 restrictions and the dynamic level 2 restrictions. Level 1 restrictions represent a static list of restrictions specific to the hardware platform. These restrictions are represented in a bit-vector, which is mapped by a hash table, to increase the speed of both inheritance of restrictions and lookups in the vector. Level 2 restrictions are individual to each process and contains only restrictions on the file system. Because of the dynamic nature of level 2 restrictions, they are stored in a hash table for each process, which allows up to 256 restrictions to be set.

The combination of level 1 and level 2 restrictions allow Umbrella to perform well because the level 1 hash table is small enough to fit into the cache of the CPU and still Umbrella is highly flexible due to the individual level 2 restrictions.

One of the philosophies of Umbrella is that, to make a secure system, the developers must be involved. Umbrella encourages programmers to restrict processes from accessing specific parts of the file system, network and critical system resources. We believe that this moving of responsibilities, from a security administrator to the developers, is necessary to obtain secure computer systems. The developers can specify suitable restrictions for a process, much more accurate than a security administrator who has never seen the source code.

Umbrella is designed to be transparent to the user. This is obtained by importing restriction through vendor-signed files. The file signatures are created using public key cryptography, where the public keys are stored on the device in a protected key ring. When a signed file enters the system, the signature is verified and the program is assigned the restrictions specified.

Two extra features for Umbrella has been evaluated. One for controlling the ability for processes to send signals and one for preventing injection of code into existing executables.

Implementation

Umbrella has been developed as an open source project hosted on SourceForge.net. The Umbrella web site have had more than 14.000 visits since the public launch on March 2nd 2004 and since the first version was released more than 200 downloads have been performed.

The Umbrella development process is divided into small steps, in the style of eXtreme Programming with short release cycles and implementing functionality before considering optimizations. The main goal of the spring semester, was to have an implementation, which could show the principle of Umbrella; this has been achieved.

The completed parts of the implementation covers restrictions on processes, which are implemented together with the intended functionality, i.e. inheritance and the ability to set child restrictions. The base for this is the Linux Security Modules framework, which provides hooks for controlling access to data structures in the kernel. The data structures for holding restrictions, i.e. bit-vector, hash function and hash table, are implemented specifically for suiting the needs of Umbrella; high flexibility and good performance.

For the pending major implementation parts, the integration into the file system, i.e. signed files, is of high priority. This requires a persistent and protected storage in the file system, access to public key cryptography within the kernel, and a number of user space tools for signing files.

Protected and persistent storage will be achieved by implementing extended attributes to the JFFS2 file system, which is the file system used for flash memory on Linux. The community developing JFFS2 is interested in having extended attributes added to the file system, thus there is a good chance that such an implementation will make it into the kernel source tree. However, implementing extensions to a file system is difficult, and the Umbrella team has no experience in this field yet. The task is thus estimated to require a considerable amount of time.

Public key cryptography within the kernel will consist of porting selected parts of the GNU Privacy Guard to the kernel, and merge it into the Umbrella source tree.

For minor pending implementation parts, the controlling of process signaling is of highest priority, due to the impact of the ability to signal processes. This is a practical issue, which is believed to be easily resolved.

Besides the implementation, we spent time investigating work in the field of verification of the Linux Security Modules framework. Since Umbrella is completely based on LSM the security provided, relies on this. The work of two articles is presented and their results were a small number of errors in the LSM framework, which have been corrected. Verifying the Umbrella source code is a pending task.

Performance of Umbrella

Umbrella is not fully implemented and thus it is not possible to perform a comprehensive benchmark of the entire system. However the current implementation does allow a benchmark of process based MAC.

The performance measures indicate an overhead around 8% for level 1 restrictions only. Adding level 2 restrictions raise the overhead to between 10% and 19%. However the 19% only occur when assigning a huge and unrealistic high number of level 2 restrictions. The current implementation of level 1 and 2 restrictions lacks some optimization, which is believed to increase performance considerably.

Although the overhead of 10% seems rather large, it is not noticeable when using the iPAQ in everyday use.

Future of Umbrella

The next steps of the development of Umbrella are to finish up the implementation and to prepare Umbrella for use in “real-world” products. At the time of writing, several companies have been contacted to find interest for a cooperation during the next two semesters. A number of companies have replied with interest, and initial contacts have been established.

Cooperation with a company is an essential part of the next two semesters at Industrial Computer Science at Aalborg University. The goals for the first semester is to complete the implementation, write and submit an article on Umbrella to a relevant conference. Our current supervisor, Emmanuel Fleury, has agreed to continue to supervise and work with us for this period.

Umbrella in Other Operating Systems

The design of Umbrella is aimed at handhelds and it is tested on a HP iPAQ running Linux. However, the design is applicable to any given operating system if it meets the following requirements.

- It must be possible to determine the restrictions of any process creating a child in order to perform the inheritance of restrictions.
- The file system should support extended attributes or similar properties for storing security information on the file system.
- It must be possible to mediate calls to the kernel.
- It must be possible to store information on active processes.

It would be interesting to port Umbrella to other operating systems for handhelds, like Symbian and Microsoft PocketPC. It has not, however, been possible to find detailed information regarding the above mentioned requirements on these operating systems.

Implementing Umbrella in other operating systems would be eased, if the processes are organized in a tree-like structure. Unix-like operating systems, such as the various versions of BSD fulfills this requirement. The porting of Umbrella will be a matter of implementing a LSM-like framework for mediating various calls in the kernel along with adding security fields in the kernel data structures. After this the Umbrella code is easily portable.

Getting Umbrella in Linux to run on other hardware architectures, simply requires the implementation of the system calls for that platform. The rest of Umbrella is architecture independent, and even independent of the sub-version of the Linux 2.6 kernel. The current Umbrella implementation is currently ported to i386, User-Mode Linux and ARM for the iPAQ.

Final Remarks

The project has spanned the last nine months and during that period many thoughts and ideas have been discussed, tried, and discarded. The project teams effort, together with great supervision and commitment from Emmanuel Fleury have resulted in a completely new scheme for mandatory access control for handheld devices.

The current implementation of Umbrella consists of the functionality concerning process based MAC. The implementation of signed files will be attended in the autumn semester, where an article on Umbrella will also be written. Cooperating with a company for adapting Umbrella to real products is also of great interest and the initial contact has already been established.

We believe that the idea for process based restrictions and inheritance of these is a step in the right direction for implementing a flexible scheme for mandatory access control on handheld devices. The idea of signing files enhances the trustworthiness of software from different vendors, and improves the general level of data integrity. Furthermore, we believe that the combination of these two strong schemes enables Umbrella to provide a secure and flexible environment for the next generation of Linux powered handheld devices.

A Tools and Howtos

A.1 Installing Linux on the iPAQ

In this section, we will give a short introduction on how to get started using Linux on the HP iPAQ 5550. The introduction will include what devices, peripherals and distributions are required for successfully installing Linux of the iPAQ.

Installing Linux is a three step process, which is listed below:

1. Use ActiveSync or network to copy the file `bootldr` and `BootBlaster3900-2.6.exe` to the iPAQ.
2. Use `BootBlaster3900-2.6.exe` to install `bootldr`.
3. Install Linux distribution via serial port (in this example the Familiar distribution is used).

A.1.1 Ad. 1: Copying Files

The first step is copying the `BootBlaster3900-2.6.exe` program and the file `bootldr` to the iPAQ using ActiveSync. This can be done by following the steps listed below:

1. Download the following files.
   - `http://familiar.handhelds.org/releases/v0.7/install/files/BootBlaster3900-2.6.exe`
   - `http://handhelds.org/download/bootldr/pxa/bootldr-pxa-2.21.10.bin.gz`
2. If ActiveSync is not already installed on the host PC, install it from the CD-ROM that followed the iPAQ.
3. Copy `BootBlaster3900-2.6.exe` to the default folder on the iPAQ by clicking Explore in ActiveSync and dragging their icons there. Ignore any ”may need to convert” messages.
4. Do the same thing for `bootldr-pxa-2.21.10.bin.gz`.

A.1.2 Ad. 2: Installing the Boot Loader

Next step is to replace the default boot loader, before doing this the following should be done.

1. Start BootBlaster by
   - Select "Start → Programs" on the iPAQ touchscreen.
   - Tap on File Explorer.
   - Tap on the Bootblaster file.
2. Backup existing OS by:
   - Execute "Flash → Save Bootldr .gz Format" in BootBlaster to save the boot loader in file `\My Documents\saved_bootldr.gz` on the iPAQ.
   - Execute "Flash → Save Wince .gz Format" in BootBlaster to save the PocketPC image in files `\My Documents\wince_image.gz` and `\My Documents\assets_image.gz` on the iPAQ. This takes about five minutes and the iPAQ may seem frozen during this period.
   - The first two steps produce the following files which should be copied to the PC for safe keeping.
     - `asset_image.gz`
     - `saved_bootldr.gz`
     - `wince_image.gz`
   - Before continuing, be sure that the iPAQ is plugged into external power, and that the battery is charged, to protect against the small chance of power failure during the very limited period the iPAQ is reprogramming the boot loader flash. Do *not* touch the power button or reset button on your iPAQ until you have performed the "Verify" step below. To install the boot loader follow the steps below:
     - Execute "Flash → Program".
     - Select `bootldr-pxa-2.21.10.bin.gz.`
     - Wait patiently. It takes about 15 seconds to program the boot loader. Do *not* interrupt this process, or the iPAQ may be left in an *unusable state*.
   - Execute "Flash → Verify".
     - If it says that the boot loader is not valid, then do *not* reset or turn off the iPAQ. Instead try programming the flash again.
     - If that does not work, program your flash with your saved boot loader.

A.1.3 Ad. 3: Installing Linux

You will need to use a terminal program such as Minicom, Kermit, or Hyperterminal. If you use Minicom or Kermit, you will need to use an external ymodem program such as sb, which is available in the Linux lrzsz package. To install Linux follow the steps below:

1. Download the latest Familiar distribution from `www.handhelds.org.` At time of writing this is the file `bootgpe2-0.7.2+unstable3-h3900.jffs2.`
2. Configure the terminal emulator using these settings: 115200 8N1 serial configuration, no flow control, no hardware handshaking.
3. Test that the terminal emulator is properly interacting with the boot loader, by issuing the command `help` , which should write a list of possible commands.
4. At the `boot>` prompt, issue the command `load root`.
5. Proceed to upload the JFFS2 file with ymodem, using the terminal emulator. This could take awhile so be patient.
6. At the `boot>` prompt, issue the command `boot`.
7. Linux should now start booting.

A.1.4 Restoring PocketPC 2003

When the BootBlaster3900-2.6.exe backups PocketPC, it seems to backup 32 + 16MB of ROM, and hence creates an image file of 49,807,360 bytes (uncompressed size). Restoring this file (for restoring PocketPC) to the iPAQ with boot loader version 2.20.1 seems impossible, due to its size and the fact that the root partition only can hold 32MB. Trying to restore PocketPC 2003 the normal way produces the following error message from the iPAQ

```
boot> load root
partition root is a jffs2 partition:
 expecting .jffs2 or wince_image.gz.
 After receiving file, will automatically uncompress .gz images
loading flash region root
using ymodem
ready for YMODEM transfer...
C4E130C3C8926E4097FDAD7E74AE1B19D
00F79874 bytes loaded to A0000400
Looks like a gzipped image, let's verify it...
Looks like a gzipped image, let's verify it...
Verifying gzipped image
.................................. (etc etc)
verifyGZipImage: calculated CRC = 0x8DFC748A
verifyGZipImage: read CRC = 0x8DFC748A
img_size is too large for region: 01F80000
img_size is too large for region: 01F80000
img_size is too large for region: 01F80000
boot>
```

It seems that the PocketPC image contains 2 rom images. One located in the first 32MB and a backup image located in the remaining 16MB of the PocketPC image. The second rom image is removed by using the command:

```
$ dd bs=1k count=32256 if=wince_image of=wince_image.new
32256+0 records in
32256+0 records out
```

This produces a new image file with 33,030,144 bytes. This file is then gzipped and transferred to the iPAQ via Minicom and the `load root` command, in the above example. The boot loader is able to erase and write the new flash and automatic verification is also successful. After the transfer is completed the iPAQ is booted with the `boot wince` command. When PocketPC have been restored it is optional to reinstall the original boot loader, but this can be done by reseting the iPAQ while holding down the cursor to activate the boot loader again. Using Minicom again to send the command `load bootldr`.

A final word of warning ... Reinstalling the boot loader is dangerous and could potentially turn the iPAQ into a very expensive paperweight!

A.2 Building a Cross Compiler for the iPAQ

There are two options for getting a working cross compiler and standard C library (together called a tool chain) for the iPAQ. Several binary versions are available from the familiar website[1]. The second option is to compile your own, which we describe how to do here. To cross compile the Intel XScale processor, we cross compile for the ARM architecture.

A cross compiler is dependent upon the kernel header files, so if using another version of the kernel there *might* be problems – but in most cases everything should work smoothly. The pre-compiled cross compiler have the version numbers of the GCC compiler version used. The latest releases in GCC-3 is a good choice.

Next follows an overview of how to build a cross compiler for the iPAQ.

1. Get the kernel source.
2. Get the binutils, gcc and glibc source.
3. Build binutils.
4. Build compiler.
5. Cross compile glibc.
6. Rebuild compiler with new glibc.

If there is no need to cross compile applications, one can skip step 5 and 6. It requires some work to get glib to compile and when that is finally done, new problems arise getting GCC to accept the new glibc.

A small shell script is available for building the tool-chain, it can be found at: `http://handhelds.org/download/toolchain/gcc-build-cross-3.3.`

A.3 Using the 2.6 Kernel on the iPAQ

In the following we describe our work with configuring the 2.6 kernel for running on the iPAQ.

[1]http://handhelds.org/download/toolchain.

Using the cross-compiler described in Appendix A.2 and the kernel source from `http://www.familiar.org` we build a 2.6 kernel for the iPAQ. Next we present a step-by-step "howto" on configuring the kernel.

The below paths, filenames and commands are all relative to the root of Linux kernel source tree.

A.3.1 Configuring the Kernel

First step is to copy the default iPAQ configuration file:

```
cp arch/arm/configs/ipaqpxa_defconfig .config
```

We have edited the `.config` manually because we have experienced that menu-config somehow removed some options that were set in the default configuration file. In the following we present the options that we have changed in the default configuration.

In order to output debug information, while the kernel is loading, to the serial port, the following options must be set in the "Character Device" section:

```
CONFIG_VT=y
CONFIG_VT_CONSOLE=y
CONFIG_HW_CONSOLE=y
```

And in the "Serial Drivers" section:

```
CONFIG_SERIAL_PXA=y
CONFIG_SERIAL_PXA_CONSOLE=y
CONFIG_SERIAL_CORE=y
CONFIG_SERIAL_CORE_CONSOLE=y
CONFIG_UNIX98_PTYS=y
CONFIG_UNIX98_PTY_COUNT=32
```

We must enable JFFS2 file system support, to make it is possible to mount the file system on the iPAQ.

```
CONFIG_JFFS2_FS=y
CONFIG_JFFS2_FS_DEBUG=2
CONFIG_JFFS2_FS_NAND=y
```

A working a complete configuration file can be found at the Umbrella projects SourceForge website at `http://sourceforge.net/projects/umbrella`.

We were not able to compile the kernel with sleeve support enabled, therefore we set `CONFIG_IPAQ_SLEEVE=n` in the "XScale-based iPAQ" section.

To enable the support for LSM and Umbrella the `CONFIG_SECURITY=y` and the `CONFIG_SECURITY_UMBRELLA=y` options must be set.

Compiling the kernel

When configuration is complete the kernel is compiled with the cross-compiler described in Appendix A.2 using the command "make zImage". Remember to include the cross-compiler in your path. If using the CVS tree is necessary to merge the Umbrella source code with the kernel. This is done using the Ruby script `merge_kernel.rb` located at `umbrella-devel/scripts` directory. The script copies the source files to the correct places in the source tree.

After compilation the kernel image is located in `arch/arm/boot/zImage` and can be copied using to the iPAQ using a terminal program like Minicom.

A.3.2 Booting the new kernel

On the iPAQ the new kernel image should be copied to `/boot/zImage2.6` and the new kernel is booted with the command:

```
boot jffs2 /boot/zImage2.6 console=ttyS0,115200
```

The *console* parameter sends kernel messages to Minicom through the serial port *ttyS0*, with a speed of 115200bps. Make sure that Minicom is configured accordingly.

A.4 User-mode Linux

User-mode Linux (UML) is a safe way of running Linux. With UML you are able to run buggy software, experiment with new Linux kernels or distributions, and poke around in the internals of Linux, all without risking the main Linux setup.

User-mode Linux gives you a virtual machine that may have more hardware and software virtual resources than the actual, physical computer. Disk storage for the virtual machine is entirely contained inside a single file on the physical machine. It is possible e.g. to assign a virtual machine only a limited hardware access. With properly limited access, nothing you do on the virtual machine can change or damage the host computer, or its software.

The purpose of setting up User-mode Linux for this project, was to have an easy way of testing our changes to the kernel code without risking changes in the host-computers. Further more it saves boot-time for each new attempt to boot the kernel. Testing this way is much easier than uploading a new kernel image to the iPAQ or simply boot the local host all the time.

The User-mode Linux download site, can be found at:
`http://user-mode-linux.sf.net/dl-sf.html`

Following a small guide for setting up UML for kernel 2.6-test9.

1. Download the kernel source from kernel.org, or directly
   `ftp://sunsite.dk/pub/os/linux/kernel.org/kernel/v2.6/linux-2.6.0-test9.tar.gz`
2. Download the appropriate User-mode Linux patch from the UML download site, or directly
   `http://unc.dl.sourceforge.net/sourceforge/user-mode-linux/uml-patch-2.6.0-test9-1.bz2`
3. Extract the kernel source and enter the directory.
4. Patch the kernel source:
   `bzcat ../uml-patch-2.6.0-test9-1.bz2 | patch -p1`
5. Install the User-mode Linux utilities. If these are not installed some parts of the kernel, e.g. network, will not work or even compile. The utilities can be downloaded from the UML download site, or directly
   `http://heanet.dl.sourceforge.net/sourceforge/user-mode-linux/uml_utilities_20030903.tar.bz2`
6. To adapt the User-mode Linux to compile with Linux Security Modules, the file `include/asm-um/common.lds.S` must have added the line `SECURITY_INIT` before the `__exitcall_begin`, currently line 81.
7. Configure the kernel for the `um` architecture:
   `make menuconfig ARCH=um`
8. Build the kernel uncompressed:
   `make linux ARCH=um`
9. Download a file system for the virtual machine from UML download site, or directly
   `http://prdownloads.sourceforge.net/user-mode-linux/Debian-3.0r0.ext2.bz2`
10. Uncompress the file system and rename it to `root_fs`.
11. Make a directory for the virtual machine, e.g. called `uml`.
12. Copy the compiled kernel `linux-2.6.0-test9/linux` and `root_fs` to the `uml` directory.
13. Give the `linux` binary execute permissions.
14. Execute the kernel, `./linux`, and your virtual machine boots.

For more information on running UML, see the User-mode Linux how-to:
`http://user-mode-linux.sourceforge.net/UserModeLinux-HOWTO.html`

B Linux for Handhelds

There exist several projects for stripping the Linux kernel for use on handhelds. Among these projects the following were considered for the purpose of running on the HP iPAQ.

- Familiar Linux – `http://familiar.handhelds.org`

  The Familiar project aims for creating the next generation of PDA operating system. Currently, most of the development time is being put to wards producing a stable, and full featured Linux distribution for the HP iPAQ series of handheld computers, as well as applications to run on top of the distribution.

- Intimate Linux – `http://intimate.handhelds.org`

  The Intimate project is a fully blown Debian based Linux distribution for the HP iPAQ. Taking the work being done by the Familiar project and combining it with fully blown Debian package management, and access to the thousands of existing Debian packages for the ARM architecture. The distribution will thus not fit within the limited amount of RAM the iPAQ has built-in. The minimum requirements are around 140MB of storage for the base image. The storage solution is based upon micro hard drives, that you can connect to the iPAQ.

- Etlinux – `http://www.etlinux.org`

  Etlinux is a complete Linux-based system designed to run on very small x86-based industrial computers. It has been designed to be small, modular, flexible and complete. It has reduced the usage memory and disk requirements to 4MB in total.

- OpenZaurus – `http://www.openzaurus.org`

  The OpenZaurus project was created as an alternative ROM (kernel and root file system) image for the Sharp Zaurus Personal Mobile Tool. OpenZaurus is a Debian based embedded distribution built from source, from the ground up. Given its Debian roots, it is quite similar to other embedded Debian-based distributions, such as the Familiar project.

C Linux Security Modules

A prerequisite for implementation of Umbrella is the ability to mediate calls to kernel space. This ability is implemented in the Linux Security Modules framework as hook functions. In this chapter the LSM framework is investigated regarding implementation of resource access control. There are several advantages to this approach. The LSM framework is a part of the Linux kernel from version 2.6, which ensures modules dependent upon LSM will also work in future versions of the Linux kernel. Furthermore LSM offers stackable modules, which makes it possible to run several security modules at the same time. Detailed information on LSM can be found in [5, 47].

C.1 The Becoming of LSM

In March 2001, the National Security Agency (NSA) gave a presentation about Security-Enhanced Linux (SELinux) at the 2.5 Linux Kernel Summit. SELinux is an implementation of flexible and fine-grained non-discretionary access controls in the Linux kernel, originally implemented as its own particular kernel patch.

In response to the NSA presentation, Linus Torvalds made a set of remarks that described a security framework he would be willing to consider for inclusion in the Linux kernel. He described a general framework that would provide a set of security hooks to control operations on kernel objects and a set of opaque security fields in kernel data structures for maintaining security attributes. This framework could then be used by loadable kernel modules to implement any desired model of security. Linus also suggested the possibility of migrating the Linux capabilities code into such a module.

The Linux Security Modules project was started by WireX[1] to develop such a framework. LSM is a joint development effort by several security projects, including Immunix, SELinux and Janus and several individuals, including Greg Kroah-Hartman and James Morris, to develop a Linux kernel patch that implements this framework. The patch is currently tracking the 2.4 series and is an

[1]http://www.wirex.com

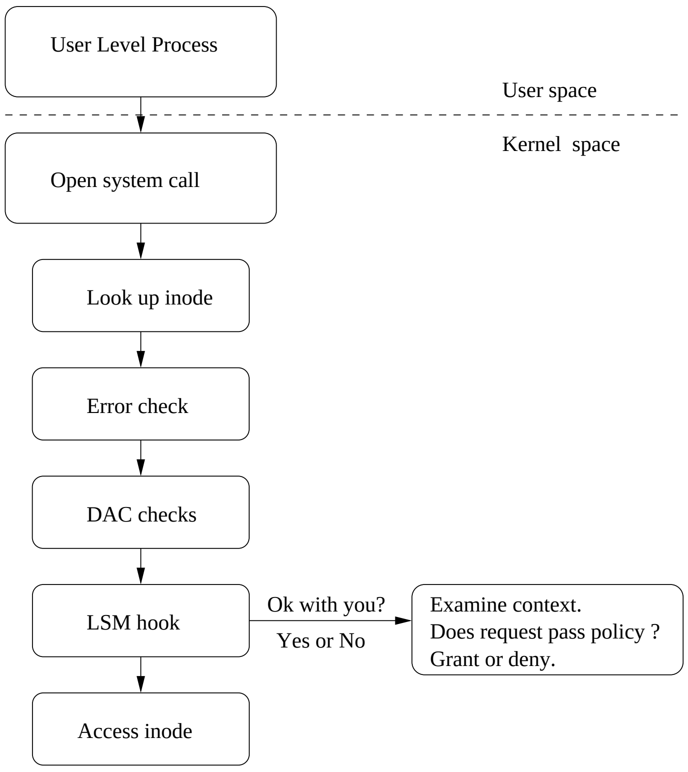

Figure C.1: LSM hook architecture.

integrated part of the 2.6 test series. This chapter provides an overview of the framework and the example capabilities security module provided by the LSM kernel patch. This is followed by an example of how to use the LSM framework to implement an actual security module.

C.2 The LSM Framework

LSM is a general framework aimed at supporting security modules in the kernel. In particular, the LSM framework is primarily focused on supporting access control modules, although future development is likely to address other security needs such as auditing [47]. By itself, the framework does not provide any additional security; it merely provides the infrastructure to support security modules. The LSM kernel patch also moves most of the capabilities logic into an optional security module, with the system using the traditional superuser logic as default. This capabilities module is discussed further in the section C.3.

The basic abstraction of the LSM interface is to mediate access to internal kernel objects. LSM seeks to allow modules to answer the question “May a given subject perform a kernel operation on an internal kernel object?”.

The mediation of access to kernel objects is achieved by placing hooks in the kernel code, just before the kernel would have accessed an internal kernel object,

Structure	Object
task_struct	Task (Process)
linux_binprm	Program
super_block	File-system
inode	Pipe, File or Socket
file	Open File
sk_buff	Network Buffer (packet)
net_device	Network Device
kern_ipc_perm	Semaphore, Shared Memory Segment or Message Queue
msg_msg	Individual Message

Table C.1: Kernel data structures modified by LSM.

as shown in figure C.1. A hook makes a call to a function that the LSM module must provide. As seen in C.1 the hook is placed immediately after the DAC checks, so the LSM module can override decisions made by the DAC. Table C.1 shows the kernel data structures modified by the LSM kernel patch and the corresponding abstract object.

The LSM kernel patch adds security fields to kernel data structures and inserts calls to hook functions at critical points in the kernel code to manage the security fields and to perform access control. It also adds functions for registering and unregistering security modules, and adds a general security system call to support new system calls for security-aware applications.

C.2.1 Security Fields

The LSM security fields are void pointers. For process and program execution security fields were added to the structures `task_struct` and `linux_binprm`. The listing below shows such a security field in `task_struct`.

```
struct task_struct {
    /* -1 unrunnable, 0 runnable, >0 stopped */
    volatile long state;
    struct thread_info *thread_info;
    ...
    // Security field added by LSM
    void *security;
    ...
    siginfo_t *last_siginfo; /* For ptrace use.  */
};
```

For file system security information, a security field was added to the structure `super_block`. For pipe, file, and socket security information, security fields were added to the structures `inode` and `file`. For packet and network device security information, security fields were added to the structures `sk_buff` and `net_device`. For System V IPC security information, security fields were added to the structures `kern_ipc_perm` and `msg_msg`. Additionally, the definitions for the structures `msg_msg`, `msg_queue`, and `shmid_kernel` were moved to the header files `include/linux/msg.h` and `include/linux/shm.h` to allow the security modules to use these definitions.

The setting of these security fields and the management of the associated security data is handled by the security modules. LSM merely provides the fields and a set of calls to security hooks, that can be implemented by the module. For most kinds of objects, an `alloc_security` and a `free_security` hook are defined, to permit the security module allocate and free security data when the corresponding kernel data structures is allocated and freed. An other set of security hooks are provided to permit the security module to update the security data as necessary, e.g. a `post_lookup` hook used to set security data for an inode after a successful lookup operation. Furthermore LSM does not provide any locking mechanism for the security fields, and it becomes the responsibility of the security module.

C.2.2 Hooks

Each LSM hook is a function pointer in a global table, `security_ops`. This table is a `security_operations` structure defined in `include/linux/security.h`. Detailed documentation for each hook is included in this header file, but can also be found at: `http://lsm.immunix.org/docs/2.5/lsm_interface.html`.

At present, the `security_operations` structure consists of a collection of substructures that group related hooks based on the kernel objects seen in table C.1, as well as some top-level hook function pointers for system operations. Hook calls can also be found in the kernel code by looking for the string `security_ops->`. An example of such a hook function is `inode_mkdir`, which is defined in the structure `security_operations` as:

```
int(*inode_mkdir)(struct inode *dir, struct dentry *dentry, int
    mode)
```

Below is the kernel function `vfs_mkdir` with one security hook call to mediate access and one security hook call to manage the security field. The security hooks are marked by a small comment.

```
int vfs_mkdir(struct inode *dir, struct dentry *dentry, int mode
    ) {
    int error = may_create(dir, dentry, NULL);

    if (error) return error;

    if (!dir->i_op || !dir->i_op->mkdir)
        return -EPERM;

    mode &= (S_IRWXUGO|S_ISVTX);
    error = security_inode_mkdir(dir, dentry, mode); /* hook */
    if (error) return error;

    DQUOT_INIT(dir);
    error = dir->i_op->mkdir(dir, dentry, mode);
    if (!error) {
        inode_dir_notify(dir, DN_CREATE);
        security_inode_post_mkdir(dir,dentry, mode);  /* hook */
    }
    return error;
}
```

This hook function is used to implement the security module example found in Section C.4.

Although the LSM hooks are organized into sub-structures based on kernel object, all of the hooks can be viewed as falling into two major categories: hooks that are used to manage the security fields and hooks that are used to perform access control. Examples of the first category of hooks include the `alloc_security` and `free_security` hooks defined for each kernel data structure that has a security field. These hooks are used to allocate and free security structures for kernel objects. The first category of hooks also includes hooks that set information in the security field after allocation, such as the `post_lookup` hook in the structure `inode_security_ops`. This hook is used to set security information for inodes after successful lookup operations. An example of the second category of hooks is the permission hook in the structure `inode_security_ops`, that checks permissions when accessing an inode.

C.2.3 Per-process Security Hooks

Linus Torvalds mentioned per-process security hooks in his original remarks as a possible alternative to global security hooks. However, if LSM were to start from the perspective of per-process hooks, then the base framework would have to deal with how to handle operations, like `kill`, that involve multiple processes, since each process might have its own hook for controlling the operation. This would require a general mechanism for composing hooks in the base framework. Additionally, LSM would still need global hooks for operations that have no process context like e.g. network input operations. Consequently, LSM provides global security hooks, but a security module is free to implement per-process hooks by storing a `security_ops` table in each process' security field and then invoking these per-process hooks from the global hooks. The problem of composition is thus deferred to the module.

C.2.4 Global Security Operations Table

The global `security_ops` table is initialized to a set of hook functions provided by a dummy security module that provides traditional superuser logic. A `register_security` function (in `security/security.c`) is provided to allow a security module to set `security_ops` to refer to its own hook functions, and an `unregister_security` function is provided to revert `security_ops` to the dummy module hooks. This mechanism is used to set the primary security module, which is responsible for making the final decision for each hook.

C.2.5 Security Module Stacking

LSM also provides a simple mechanism for stacking additional security modules with the primary security module. It defines `register_security` and `unregister_security` hooks in the `security_operations` structure and provides `mod_reg_security` and `mod_unreg_security` functions that invoke these hooks after performing some sanity checking. A security module can call these functions in order to stack with other modules. However, the actual details of how this stacking is handled are deferred to the module, which can implement these hooks in any way it wishes; including always returning an error if it does not wish to support stacking. In this manner, LSM again defers the problem of composition to the module.

C.2.6 Expanding the framework

LSM adds a general security system call that simply invokes the `sys_security` hook. This system call and hook permits security modules to implement new system calls for security-aware applications. The interface is similar to socket call, but also has an id to help identify the security module whose call is being invoked.

The next section describe how to implement a security module using the LSM hooks. This is done to explore the possibilities for using the basic LSM functionality.

C.3 The LSM Capabilities Module

The Linux kernel currently provides support for at sub-set of POSIX.1e capabilities [47]. One of the requirements for the LSM project was to move this functionality to an optional security module. POSIX.1e capabilities provides a mechanism for partitioning traditional superuser privileges and assigning them to particular processes.

By nature, privilege granting is a permissive form of access control, since it grants an access that would normally be denied. As a consequence, the LSM framework must provide a permissive interface with at least the same granularity of the Linux capabilities implementation. LSM retains the existing `capable` interface used within the kernel for performing capability checks. However the `capable` function have been reduced to a simple wrapper for a LSM hook, thereby allowing any desired logic to be implemented in a security module. This allows LSM to leverage the numerous existing kernel calls to `capable` and to avoid intrusive changes to the kernel. LSM also defines hooks to allow the logic for other forms of capability checking and capability computations to be encapsulated within the security module.

A process capability set, a bit-vector, is stored in the `task_struct` structure. Because LSM adds an opaque security field to this structure and hooks to manage the field, it would be possible to move the existing bit-vector into the field. Such a change would be logical in the LSM framework, but have been not been implemented to ease stacking with other modules. One of the difficulties of stacking security modules in the LSM framework is the need to share the opaque security fields. Many security modules would want to stack with the capabilities module, since its logic have been integrated into the kernel for some time and it is relied upon by some applications such as `named` and `sendmail`. Leaving the capability bit-vector in the `task_struct` structure eases this composition, at the cost of wasted space for modules does not use it.

The Linux kernel support for capabilities also include two system calls: `capset` and capget. To ensure compatibility with existing applications, these system calls are retained by LSM, but the core capabilities logic for these functions has been replaced by calls to LSM hooks.

C.4 Example Security Module

To demonstrate the Linux Security Modules, a small security module was implemented. The module reacts every time a directory is created, and prints a message to the kernel log. Another example module can be found in [31]

The module is structured like a regular Linux kernel module. Information about this may be found in [37]. In the following, some code snippets from the module are presented and commented.

This is the implementation of the LSM hook function inode_mkdir. To follow the normal guidelines for modules, the function is prefixed with the module name. Returning 0 means that creation of the directory is allowed. In a more advanced example, some checks on the parameters or other parts of the system could be performed.

```
static int dirmonitor_inode_mkdir (struct inode *dir, struct
    dentry *dentry, int mode) {
   printk(KERN_INFO "Directory created\n");
   return 0;
}
```

Next, we need to state the use of the capability function for the inode_mkdir hook.

```
static struct security_operations dirmonitor_security_ops = {
   .inode_mkdir = dirmonitor_inode_mkdir,
};
```

The init and exit functions register the dirmonitor module with the security framework by calling the register_security and unregister_security functions with the above struct as parameter.

C.5 Discussion

The Linux Security Modules framework provide general interface for implementing security modules for Linux, where it is possible to use several LSM security modules at the same time. LSM is an integrated part of the Linux kernel 2.6, which ensures that modules dependent on LSM will run on future kernels. The main part of the LSM hooks is placed in the kernel where access to resources are handled. These facts these are the main reason for choosing LSM as framework for implementing Umbrella.

D LSM Hooks in Linux 2.6.3

This appendix provides information on all the security hooks in Linux 2.6.3. This information can be found in the kernel source tree in `include/linux/security.h`. The @-names refers to parameter names of the hook functions.

D.1 Program Execution Operations

bprm_alloc_security Allocate and attach a security structure to the @bprm → security field. The security field is initialized to NULL when the bprm structure is allocated. @bprm contains the linux_binprm structure to be modified. Return 0 if operation was successful.

bprm_free_security @bprm contains the linux_binprm structure to be modified. Deallocate and clear the @bprm → security field.

bprm_compute_creds Compute and set the security attributes of a process being transformed by an execve operation based on the old attributes (current → security) and the information saved in @bprm → security by the set_security hook. Since this hook function (and its caller) are void, this hook can not return an error. However, it can leave the security attributes of the process unchanged if an access failure occurs at this point. It can also perform other state changes on the process (e.g. closing open file descriptors to which access is no longer granted if the attributes were changed). @bprm contains the linux_binprm structure.

bprm_set_security Save security information in the bprm → security field, typically based on information about the bprm → file, for later use by the compute_creds hook. This hook may also optionally check permissions (e.g. for transitions between security domains). This hook may be called multiple times during a single execve, e.g. for interpreters. The hook can tell whether it has already been called by checking to see if @bprm → security is non-NULL. If so, then the hook may decide either to retain the security information saved earlier or to replace it. @bprm contains the linux_binprm structure. Return 0 if the hook is successful and permission is granted.

bprm_check_security This hook mediates the point when a search for a binary handler will begin. It allows a check the @bprm → security value which is set in the preceding set_security call. The primary difference from set_security is that the argv list and envp list are reliably available in @bprm. This hook may be called multiple times during a single execve; and in each pass set_security is called first. @bprm contains the linux_binprm structure. Return 0 if the hook is successful and permission is granted.

bprm_secureexec Return a boolean value (0 or 1) indicating whether a "secure exec" is required. The flag is passed in the auxiliary table on the initial stack to the ELF interpreter to indicate whether libc should enable secure mode. @bprm contains the linux_binprm structure.

D.2 File System Operations

sb_alloc_security Allocate and attach a security structure to the sb → s_security field. The s_security field is initialized to NULL when the structure is allocated. @sb contains the super_block structure to be modified. Return 0 if operation was successful.

sb_free_security Deallocate and clear the sb → s_security field. @sb contains the super_block structure to be modified.

sb_statfs Check permission before obtaining file system statistics for the @sb file system. @sb contains the super_block structure for the file system. Return 0 if permission is granted.

sb_mount Check permission before an object specified by @dev_name is mounted on the mount point named by @nd. For an ordinary mount, @dev_name identifies a device if the file system type requires a device. For a remount (@flags & MS_REMOUNT), @dev_name is irrelevant. For a loopback-/bind mount (@flags & MS_BIND), @dev_name identifies the pathname of the object being mounted. @dev_name contains the name for object being mounted. @nd contains the nameidata structure for mount point object. @type contains the file system type. @flags contains the mount flags. @data contains the file system-specific data. Return 0 if permission is granted.

sb_copy_data Allow mount option data to be copied prior to parsing by the file system, so that the security module can extract security-specific mount options cleanly (a file system may modify the data e.g. with strsep()). This also allows the original mount data to be stripped of security- specific options to avoid having to make file systems aware of them. @fstype the type of file system being mounted. @orig the original mount data copied from user space. @copy copied data which will be passed to the security module. Returns 0 if the copy was successful.

sb_check_sb Check permission before the device with superblock @mnt → sb is mounted on the mount point named by @nd. @mnt contains the vfsmount for device being mounted. @nd contains the nameidata object for the mount point. Return 0 if permission is granted.

sb_umount Check permission before the @mnt file system is unmounted. @mnt contains the mounted file system. @flags contains the unmount flags, e.g. MNT_FORCE. Return 0 if permission is granted.

sb_umount_close Close any files in the @mnt mounted file system that are held open by the security module. This hook is called during an umount operation prior to checking whether the file system is still busy. @mnt contains the mounted file system.

sb_umount_busy Handle a failed umount of the @mnt mounted file system, e.g. re-opening any files that were closed by umount_close. This hook is called during an umount operation if the umount fails after a call to the umount_close hook. @mnt contains the mounted file system.

sb_post_remount Update the security module's state when a file system is remounted. This hook is only called if the remount was successful. @mnt contains the mounted file system. @flags contains the new file system flags. @data contains the file system-specific data.

sb_post_mountroot Update the security module's state when the root file system is mounted. This hook is only called if the mount was successful.

sb_post_addmount Update the security module's state when a file system is mounted. This hook is called any time a mount is successfully grafetd to the tree. @mnt contains the mounted file system. @mountpoint_nd contains the nameidata structure for the mount point.

sb_pivotroot Check permission before pivoting the root file system. @old_nd contains the nameidata structure for the new location of the current root (put_old). @new_nd contains the nameidata structure for the new root (new_root). Return 0 if permission is granted.

sb_post_pivotroot Update module state after a successful pivot. @old_nd contains the nameidata structure for the old root. @new_nd contains the nameidata structure for the new root.

D.3 Inode Operations

inode_alloc_security Allocate and attach a security structure to @inode $\rightarrow$ i_security. The i_security field is initialized to NULL when the inode structure is allocated. @inode contains the inode structure. Return 0 if operation was successful.

inode_free_security @inode contains the inode structure. Deallocate the inode security structure and set @inode $\rightarrow$ i_security to NULL.

inode_create Check permission to create a regular file. @dir contains inode structure of the parent of the new file. @dentry contains the dentry structure for the file to be created. @mode contains the file mode of the file to be created. Return 0 if permission is granted.

inode_post_create Set the security attributes on a newly created regular file. This hook is called after a file has been successfully created. @dir contains the inode structure of the parent directory of the new file. @dentry contains the the dentry structure for the newly created file. @mode contains the file mode.

inode_link Check permission before creating a new hard link to a file. @old_dentry contains the dentry structure for an existing link to the file. @dir contains the inode structure of the parent directory of the new link. @new_dentry contains the dentry structure for the new link. Return 0 if permission is granted.

inode_post_link Set security attributes for a new hard link to a file. @old_dentry contains the dentry structure for the existing link. @dir contains the inode structure of the parent directory of the new file. @new_dentry contains the dentry structure for the new file link.

inode_unlink Check the permission to remove a hard link to a file. @dir contains the inode structure of parent directory of the file. @dentry contains the dentry structure for file to be unlinked. Return 0 if permission is granted.

inode_symlink Check the permission to create a symbolic link to a file. @dir contains the inode structure of parent directory of the symbolic link. @dentry contains the dentry structure of the symbolic link. @old_name contains the pathname of file. Return 0 if permission is granted.

inode_post_symlink @dir contains the inode structure of the parent directory of the new link. @dentry contains the dentry structure of new symbolic link. @old_name contains the pathname of file. Set security attributes for a newly created symbolic link. Note that @dentry → d_inode may be NULL, since the file system might not instantiate the dentry (e.g. NFS).

inode_mkdir Check permissions to create a new directory in the existing directory associated with inode strcture @dir. @dir containst the inode structure of parent of the directory to be created. @dentry contains the dentry structure of new directory. @mode contains the mode of new directory. Return 0 if permission is granted.

inode_post_mkdir Set security attributes on a newly created directory. @dir contains the inode structure of parent of the directory to be created. @dentry contains the dentry structure of new directory. @mode contains the mode of new directory.

inode_rmdir Check the permission to remove a directory. @dir contains the inode structure of parent of the directory to be removed. @dentry contains the dentry structure of directory to be removed. Return 0 if permission is granted.

inode_mknod Check permissions when creating a special file (or a socket or a fifo file created via the mknod system call). Note that if mknod operation is being done for a regular file, then the create hook will be called and not this hook. @dir contains the inode structure of parent of the new file.

@dentry contains the dentry structure of the new file. @mode contains the mode of the new file. @dev contains the the device number. Return 0 if permission is granted.

inode_post_mknod Set security attributes on a newly created special file (or socket or fifo file created via the mknod system call). @dir contains the inode structure of parent of the new node. @dentry contains the dentry structure of the new node. @mode contains the mode of the new node. @dev contains the the device number.

inode_rename Check for permission to rename a file or directory. @old_dir contains the inode structure for parent of the old link. @old_dentry contains the dentry structure of the old link. @new_dir contains the inode structure for parent of the new link. @new_dentry contains the dentry structure of the new link. Return 0 if permission is granted.

inode_post_rename Set security attributes on a renamed file or directory. @old_dir contains the inode structure for parent of the old link. @old_dentry contains the dentry structure of the old link. @new_dir contains the inode structure for parent of the new link. @new_dentry contains the dentry structure of the new link.

inode_readlink Check the permission to read the symbolic link. @dentry contains the dentry structure for the file link. Return 0 if permission is granted.

inode_follow_link Check permission to follow a symbolic link when looking up a pathname. @dentry contains the dentry structure for the link. @nd contains the nameidata structure for the parent directory. Return 0 if permission is granted.

inode_permission Check permission before accessing an inode. This hook is called by the existing Linux permission function, so a security module can use it to provide additional checking for existing Linux permission checks. Notice that this hook is called when a file is opened (as well as many other operations), whereas the file_security_ops permission hook is called when the actual read/write operations are performed. @inode contains the inode structure to check. @mask contains the permission mask. @nd contains the nameidata (may be NULL). Return 0 if permission is granted.

inode_setattr Check permission before setting file attributes. Note that the kernel call to notify_change is performed from several locations, whenever file attributes change (such as when a file is truncated, chown/chmod operations, transferring disk quotas, etc). @dentry contains the dentry structure for the file. @attr is the iattr structure containing the new file attributes. Return 0 if permission is granted.

inode_getattr Check permission before obtaining file attributes. @mnt is the vfsmount where the dentry was looked up @dentry contains the dentry structure for the file. Return 0 if permission is granted.

inode_delete @inode contains the inode structure for deleted inode. This hook is called when a deleted inode is released (i.e. an inode with no hard links has its use count drop to zero). A security module can use this hook to release any persistent label associated with the inode.

inode_setxattr Check permission before setting the extended attributes @value identified by @name for @dentry. Return 0 if permission is granted.

inode_post_setxattr Update inode security field after successful setxattr operation. @value identified by @name for @dentry.

inode_getxattr Check permission before obtaining the extended attributes identified by @name for @dentry. Return 0 if permission is granted.

inode_listxattr Check permission before obtaining the list of extended attribute names for @dentry. Return 0 if permission is granted.

inode_removexattr Check permission before removing the extended attribute identified by @name for @dentry. Return 0 if permission is granted.

inode_getsecurity Copy the extended attribute representation of the security label associated with @name for @dentry into @buffer. @buffer may be NULL to request the size of the buffer required. @size indicates the size of @buffer in bytes. Note that @name is the remainder of the attribute name after the security. prefix has been removed. Return number of bytes used/required on success.

inode_setsecurity Set the security label associated with @name for @dentry from the extended attribute value @value. @size indicates the size of the @value in bytes. @flags may be XATTR_CREATE, XATTR_REPLACE, or 0. Note that @name is the remainder of the attribute name after the security. prefix has been removed. Return 0 on success.

inode_listsecurity Copy the extended attribute names for the security labels associated with @dentry into @buffer. @buffer may be NULL to request the size of the buffer required. Returns number of bytes used/required on success.

D.4 File Operations

file_permission Check file permissions before accessing an open file. This hook is called by various operations that read or write files. A security module can use this hook to perform additional checking on these operations, e.g. to revalidate permissions on use to support privilege bracketing or policy changes. Notice that this hook is used when the actual read/write operations are performed, whereas the inode_security_ops hook is called when a file is opened (as well as many other operations). Caveat: Although this hook can be used to revalidate permissions for various system call operations that read or write files, it does not address the revalidation of permissions for memory-mapped files. Security modules must handle this separately if they need such revalidation. @file contains the file structure being accessed. @mask contains the requested permissions. Return 0 if permission is granted.

file_alloc_security Allocate and attach a security structure to the file → f_security field. The security field is initialized to NULL when the structure is first created. @file contains the file structure to secure. Return 0 if the hook is successful and permission is granted.

file_free_security Deallocate and free any security structures stored in file → f_security. @file contains the file structure being modified.

file_ioctl @file contains the file structure. @cmd contains the operation to perform. @arg contains the operational arguments. Check permission for an ioctl operation on @file. Note that @arg can sometimes represents a user space pointer; in other cases, it may be a simple integer value. When @arg represents a user space pointer, it should never be used by the security module. Return 0 if permission is granted.

file_mmap Check permissions for a mmap operation. The @file may be NULL, e.g. if mapping anonymous memory. @file contains the file structure for file to map (may be NULL). @prot contains the requested permissions. @flags contains the operational flags. Return 0 if permission is granted.

file_mprotect Check permissions before changing memory access permissions. @vma contains the memory region to modify. @prot contains the requested permissions. Return 0 if permission is granted.

file_lock Check permission before performing file locking operations. Note: this hook mediates both flock and fcntl style locks. @file contains the file structure. @cmd contains the posix-translated lock operation to perform (e.g. F_RDLCK, F_WRLCK). Return 0 if permission is granted.

file_fcntl Check permission before allowing the file operation specified by @cmd from being performed on the file @file. Note that @arg can sometimes represents a user space pointer; in other cases, it may be a simple integer value. When @arg represents a user space pointer, it should never be used by the security module. @file contains the file structure. @cmd contains the operation to be performed. @arg contains the operational arguments. Return 0 if permission is granted.

file_set_fowner Save owner security information (typically from current → security) in file → f_security for later use by the send_sigiotask hook. @file contains the file structure to update. Return 0 on success.

file_send_sigiotask Check permission for the file owner @fown to send SIGIO to the process @tsk. Note that this hook is always called from interrupt. Note that the fown_struct, @fown, is never outside the context of a struct file, so the file structure (and associated security information) can always be obtained: (struct file *)((long)fown - offsetof(struct file,f_owner)); @tsk contains the structure of task receiving signal. @fown contains the file owner information. @fd contains the file descriptor. @reason contains the operational flags. Return 0 if permission is granted.

file_receive This hook allows security modules to control the ability of a process to receive an open file descriptor via socket IPC. @file contains the file structure being received. Return 0 if permission is granted.

D.5 Process Operations

task_create Check permission before creating a child process. See the clone(2) manual page for definitions of the @clone_flags. @clone_flags contains the flags indicating what should be shared. Return 0 if permission is granted.

task_alloc_security @p contains the task_struct for child process. Allocate and attach a security structure to the p → security field. The security field is initialized to NULL when the task structure is allocated. Return 0 if operation was successful.

task_free_security @p contains the task_struct for process. Deallocate and clear the p → security field.

task_setuid Check permission before setting one or more of the user identity attributes of the current process. The @flags parameter indicates which of the set*uid system calls invoked this hook and how to interpret the @id0, @id1, and @id2 parameters. See the LSM_SETID definitions at the beginning of this file for the @flags values and their meanings. @id0 contains a uid. @id1 contains a uid. @id2 contains a uid. @flags contains one of the LSM_SETID_* values. Return 0 if permission is granted.

task_post_setuid Update the module's state after setting one or more of the user identity attributes of the current process. The @flags parameter indicates which of the set*uid system calls invoked this hook. If @flags is LSM_SETID_FS, then @old_ruid is the old fs uid and the other parameters are not used. @old_ruid contains the old real uid (or fs uid if LSM_SETID_FS). @old_euid contains the old effective uid (or -1 if LSM_SETID_FS). @old_suid contains the old saved uid (or -1 if LSM_SETID_FS). @flags contains one of the LSM_SETID_* values. Return 0 on success.

task_setgid Check permission before setting one or more of the group identity attributes of the current process. The @flags parameter indicates which of the set*gid system calls invoked this hook and how to interpret the @id0, @id1, and @id2 parameters. See the LSM_SETID definitions at the beginning of this file for the @flags values and their meanings. @id0 contains a gid. @id1 contains a gid. @id2 contains a gid. @flags contains one of the LSM_SETID_* values. Return 0 if permission is granted.

task_setpgid Check permission before setting the process group identifier of the process @p to @pgid. @p contains the task_struct for process being modified. @pgid contains the new pgid. Return 0 if permission is granted.

task_getpgid Check permission before getting the process group identifier of the process @p. @p contains the task_struct for the process. Return 0 if permission is granted.

task_getsid Check permission before getting the session identifier of the process @p. @p contains the task_struct for the process. Return 0 if permission is granted.

task_setgroups Check permission before setting the supplementary group set of the current process to @grouplist. @gidsetsize contains the number of elements in @grouplist. @grouplist contains the array of gids. Return 0 if permission is granted.

task_setnice Check permission before setting the nice value of @p to @nice. @p contains the task_struct of process. @nice contains the new nice value. Return 0 if permission is granted.

task_setrlimit Check permission before setting the resource limits of the current process for @resource to @new_rlim. The old resource limit values can be examined by dereferencing (current → rlim + resource). @resource contains the resource whose limit is being set. @new_rlim contains the new limits for @resource. Return 0 if permission is granted.

task_setscheduler Check permission before setting scheduling policy and/or parameters of process @p based on @policy and @lp. @p contains the task_struct for process. @policy contains the scheduling policy. @lp contains the scheduling parameters. Return 0 if permission is granted.

task_getscheduler Check permission before obtaining scheduling information for process @p. @p contains the task_struct for process. Return 0 if permission is granted.

task_kill Check permission before sending signal @sig to @p. @info can be NULL, the constant 1, or a pointer to a siginfo structure. If @info is 1 or SI_FROMKERNEL(info) is true, then the signal should be viewed as coming from the kernel and should typically be permitted. SIGIO signals are handled separately by the send_sigiotask hook in file_security_ops. @p contains the task_struct for process. @info contains the signal information. @sig contains the signal value. Return 0 if permission is granted.

task_wait Check permission before allowing a process to reap a child process @p and collect its status information. @p contains the task_struct for process. Return 0 if permission is granted.

task_prctl Check permission before performing a process control operation on the current process. @option contains the operation. @arg2 contains a argument. @arg3 contains a argument. @arg4 contains a argument. @arg5 contains a argument. Return 0 if permission is granted.

task_reparent_to_init Set the security attributes in @p → security for a kernel thread that is being reparented to the init task. @p contains the task_struct for the kernel thread.

task_to_inode Set the security attributes for an inode based on an associated task's security attributes, e.g. for /proc/pid inodes. @p contains the task_struct for the task. @inode contains the inode structure for the inode.

D.6 Netlink Messaging

netlink_send Save security information for a netlink message so that permission checking can be performed when the message is processed. The security information can be saved using the eff_cap field of the netlink_skb_parms structure. @skb contains the sk_buff structure for the netlink message. Return 0 if the information was successfully saved.

netlink_recv Check permission before processing the received netlink message in @skb. @skb contains the sk_buff structure for the netlink message. Return 0 if permission is granted.

D.7 Unix Domain Networking

unix_stream_connect Check permissions before establishing a Unix domain stream connection between @sock and @other. @sock contains the socket structure. @other contains the peer socket structure. Return 0 if permission is granted.

unix_may_send Check permissions before connecting or sending datagrams from @sock to @other. @sock contains the socket structure. @sock contains the peer socket structure. Return 0 if permission is granted. The @unix_stream_connect and @unix_may_send hooks were necessary because Linux provides an alternative to the conventional file name space for Unix domain sockets. Whereas binding and connecting to sockets in the file name space is mediated by the typical file permissions (and caught by the mknod and permission hooks in inode_security_ops), binding and connecting to sockets in the abstract name space is completely unmediated. Sufficient control of Unix domain sockets in the abstract name space isn't possible using only the socket layer hooks, since we need to know the actual target socket, which is not looked up until we are inside the af_unix code.

D.8 Socket Operations

socket_create Check permissions prior to creating a new socket. @family contains the requested protocol family. @type contains the requested communications type. @protocol contains the requested protocol. Return 0 if permission is granted.

socket_post_create This hook allows a module to update or allocate a per-socket security structure. Note that the security field was not added directly to the socket structure, but rather, the socket security information is stored in the associated inode. Typically, the inode alloc_security hook will allocate and and attach security information to sock $\rightarrow$ inode $\rightarrow$ i_security. This hook may be used to update the sock $\rightarrow$ inode $\rightarrow$ i_security field with additional information that wasn't available when the inode was allocated. @sock contains the newly created socket structure. @family contains the requested protocol family. @type contains the requested communications type. @protocol contains the requested protocol.

socket_bind Check permission before socket protocol layer bind operation is performed and the socket @sock is bound to the address specified in the

@address parameter. @sock contains the socket structure. @address contains the address to bind to. @addrlen contains the length of address. Return 0 if permission is granted.

socket_connect Check permission before socket protocol layer connect operation attempts to connect socket @sock to a remote address, @address. @sock contains the socket structure. @address contains the address of remote endpoint. @addrlen contains the length of address. Return 0 if permission is granted.

socket_listen Check permission before socket protocol layer listen operation. @sock contains the socket structure. @backlog contains the maximum length for the pending connection queue. Return 0 if permission is granted.

socket_accept Check permission before accepting a new connection. Note that the new socket, @newsock, has been created and some information copied to it, but the accept operation has not actually been performed. @sock contains the listening socket structure. @newsock contains the newly created server socket for connection. Return 0 if permission is granted.

socket_post_accept This hook allows a security module to copy security information into the newly created socket's inode. @sock contains the listening socket structure. @newsock contains the newly created server socket for connection.

socket_sendmsg Check permission before transmitting a message to another socket. @sock contains the socket structure. @msg contains the message to be transmitted. @size contains the size of message. Return 0 if permission is granted.

socket_recvmsg Check permission before receiving a message from a socket. @sock contains the socket structure. @msg contains the message structure. @size contains the size of message structure. @flags contains the operational flags. Return 0 if permission is granted.

socket_getsockname Check permission before the local address (name) of the socket object @sock is retrieved. @sock contains the socket structure. Return 0 if permission is granted.

socket_getpeername Check permission before the remote address (name) of a socket object @sock is retrieved. @sock contains the socket structure. Return 0 if permission is granted.

socket_getsockopt Check permissions before retrieving the options associated with socket @sock. @sock contains the socket structure. @level contains the protocol level to retrieve option from. @optname contains the name of option to retrieve. Return 0 if permission is granted.

socket_setsockopt Check permissions before setting the options associated with socket @sock. @sock contains the socket structure. @level contains the protocol level to set options for. @optname contains the name of the option to set. Return 0 if permission is granted.

socket_shutdown Checks permission before all or part of a connection on the socket @sock is shut down. @sock contains the socket structure. @how contains the flag indicating how future sends and receives are handled. Return 0 if permission is granted.

socket_sock_rcv_skb Check permissions on incoming network packets. This hook is distinct from Netfilter's IP input hooks since it is the first time that the incoming sk_buff @skb has been associated with a particular socket, @sk. @sk contains the sock (not socket) associated with the incoming sk_buff. @skb contains the incoming network data.

socket_getpeersec This hook allows the security module to provide peer socket security state to user space via getsockopt SO_GETPEERSEC. @sock is the local socket. @optval user space memory where the security state is to be copied. @optlen user space int where the module should copy the actual length of the security state. @len as input is the maximum length to copy to user space provided by the caller. Return 0 if all is well, otherwise, typical getsockopt return values.

sk_alloc_security Allocate and attach a security structure to the sk $\rightarrow$ sk_security field, which is used to copy security attributes between local stream sockets.

sk_free_security Deallocate security structure.

D.9 System V IPC Operations

ipc_permission Check permissions for access to IPC @ipcp contains the kernel IPC permission structure @flag contains the desired (requested) permission set Return 0 if permission is granted.

D.9.1 Individual Messages Held In System V IPC Message Queues

msg_msg_alloc_security Allocate and attach a security structure to the msg $\rightarrow$ security field. The security field is initialized to NULL when the structure is first created. @msg contains the message structure to be modified. Return 0 if operation was successful and permission is granted.

msg_msg_free_security Deallocate the security structure for this message. @msg contains the message structure to be modified.

D.9.2 System V IPC Message Queues

msg_queue_alloc_security Allocate and attach a security structure to the msq $\rightarrow$ q_perm.security field. The security field is initialized to NULL when the structure is first created. @msq contains the message queue structure to be modified. Return 0 if operation was successful and permission is granted.

msg_queue_free_security Deallocate security structure for this message queue. @msq contains the message queue structure to be modified.

msg_queue_associate Check permission when a message queue is requested through the msgget system call. This hook is only called when returning the message queue identifier for an existing message queue, not when a new message queue is created. @msq contains the message queue to act upon. @msqflg contains the operation control flags. Return 0 if permission is granted.

msg_queue_msgctl Check permission when a message control operation specified by @cmd is to be performed on the message queue @msq. The @msq may be NULL, e.g. for IPC_INFO or MSG_INFO. @msq contains the message queue to act upon. May be NULL. @cmd contains the operation to be performed. Return 0 if permission is granted.

msg_queue_msgsnd Check permission before a message, @msg, is enqueued on the message queue, @msq. @msq contains the message queue to send message to. @msg contains the message to be enqueued. @msqflg contains operational flags. Return 0 if permission is granted.

msg_queue_msgrcv Check permission before a message, @msg, is removed from the message queue, @msq. The @target task structure contains a pointer to the process that will be receiving the message (not equal to the current process when inline receives are being performed). @msq contains the message queue to retrieve message from. @msg contains the message destination. @target contains the task structure for recipient process. @type contains the type of message requested. @mode contains the operational flags. Return 0 if permission is granted.

D.9.3 System V Shared Memory Segments

shm_alloc_security Allocate and attach a security structure to the shp → shm_perm.security field. The security field is initialized to NULL when the structure is first created. @shp contains the shared memory structure to be modified. Return 0 if operation was successful and permission is granted.

shm_free_security Deallocate the security struct for this memory segment. @shp contains the shared memory structure to be modified.

shm_associate Check permission when a shared memory region is requested through the shmget system call. This hook is only called when returning the shared memory region identifier for an existing region, not when a new shared memory region is created. @shp contains the shared memory structure to be modified. @shmflg contains the operation control flags. Return 0 if permission is granted.

shm_shmctl Check permission when a shared memory control operation specified by @cmd is to be performed on the shared memory region @shp. The @shp may be NULL, e.g. for IPC_INFO or SHM_INFO. @shp contains shared memory structure to be modified. @cmd contains the operation to be performed. Return 0 if permission is granted.

shm_shmat Check permissions prior to allowing the shmat system call to attach the shared memory segment @shp to the data segment of the calling process. The attaching address is specified by @shmaddr. @shp contains the shared memory structure to be modified. @shmaddr contains the address to attach memory region to. @shmflg contains the operational flags. Return 0 if permission is granted.

D.9.4 System V Semaphores

sem_alloc_security Allocate and attach a security structure to the sma $\rightarrow$ sem_perm.security field. The security field is initialized to NULL when the structure is first created. @sma contains the semaphore structure Return 0 if operation was successful and permission is granted.

sem_free_security Deallocate security struct for this semaphore @sma contains the semaphore structure.

sem_associate Check permission when a semaphore is requested through the semget system call. This hook is only called when returning the semaphore identifier for an existing semaphore, not when a new one must be created. @sma contains the semaphore structure. @semflg contains the operation control flags. Return 0 if permission is granted.

sem_semctl Check permission when a semaphore operation specified by @cmd is to be performed on the semaphore @sma. The @sma may be NULL, e.g. for IPC_INFO or SEM_INFO. @sma contains the semaphore structure. May be NULL. @cmd contains the operation to be performed. Return 0 if permission is granted. @sem_semop Check permissions before performing operations on members of the semaphore set @sma. If the @alter flag is nonzero, the semaphore set may be modified. @sma contains the semaphore structure. @sops contains the operations to perform. @nsops contains the number of operations to perform. @alter contains the flag indicating whether changes are to be made. Return 0 if permission is granted.

D.10 Capabilities and Different System Calls

ptrace Check permission before allowing the @parent process to trace the @child process. Security modules may also want to perform a process tracing check during an execve in the set_security or compute_creds hooks of binprm_security_ops if the process is being traced and its security attributes would be changed by the execve. @parent contains the task_struct structure for parent process. @child contains the task_struct structure for child process. Return 0 if permission is granted.

capget Get the @effective, @inheritable, and @permitted capability sets for the @target process. The hook may also perform permission checking to determine if the current process is allowed to see the capability sets of the @target process. @target contains the task_struct structure for target process. @effective contains the effective capability set. @inheritable contains the inheritable capability set. @permitted contains the permitted capability set. Return 0 if the capability sets were successfully obtained.

capset_check Check permission before setting the @effective, @inheritable, and @permitted capability sets for the @target process. Caveat: @target is also set to current if a set of processes is specified (i.e. all processes other than current and init or a particular process group). Hence, the capset_set hook may need to revalidate permission to the actual target process. @target contains the task_struct structure for target process. @effective contains the effective capability set. @inheritable contains the inheritable capability set. @permitted contains the permitted capability set. Return 0 if permission is granted.

capset_set Set the @effective, @inheritable, and @permitted capability sets for the @target process. Since capset_check cannot always check permission to the real @target process, this hook may also perform permission checking to determine if the current process is allowed to set the capability sets of the @target process. However, this hook has no way of returning an error due to the structure of the sys_capset code. @target contains the task_struct structure for target process. @effective contains the effective capability set. @inheritable contains the inheritable capability set. @permitted contains the permitted capability set.

acct Check permission before enabling or disabling process accounting. If accounting is being enabled, then @file refers to the open file used to store accounting records. If accounting is being disabled, then @file is NULL. @file contains the file structure for the accounting file (may be NULL). Return 0 if permission is granted.

sysctl Check permission before accessing the @table sysctl variable in the manner specified by @op. @table contains the ctl_table structure for the sysctl variable. @op contains the operation (001 = search, 002 = write, 004 = read). Return 0 if permission is granted.

capable Check whether the @tsk process has the @cap capability. @tsk contains the task_struct for the process. @cap contains the capability <include/linux/capability.h>. Return 0 if the capability is granted for @tsk.

syslog Check permission before accessing the kernel message ring or changing logging to the console. See the syslog(2) manual page for an explanation of the @type values. @type contains the type of action. Return 0 if permission is granted.

vm_enough_memory Check permissions for allocating a new virtual mapping. @pages contains the number of pages. Return 0 if permission is granted.

D.11 Registering and Unregistering Modules

register_security Allow module stacking. @name contains the name of the security module being stacked. @ops contains a pointer to the struct security_operations of the module to stack.

unregister_security Remove a stacked module. @name contains the name of the security module being unstacked. @ops contains a pointer to the struct security_operations of the module to unstack.

E Roadmap of Umbrella

The development of Umbrella is done in small steps in style of eXtreme Programming, with many sub-releases. In this section you will find the overall roadmap for the project.

1. (**RELEASED 2004-03-01**) The primary goal of the first release is to begin implementing the kernel module and getting a feeling for coding into the Linux kernel and LSM. Besides that, the ability to make simple umbrella system calls from user space is to be addressed.
   - Ability to set child restrictions for restricting the next child from accessing the directory `/tmp`.
   - Simple restriction-vector on processes.
   - Micro user space library for coding for Umbrella (setting child restrictions).
2. (**RELEASED 2004-03-08**) In this release the goal is to make it possible to restrict processes from more than one resource.
   - Implement a bit-vector to hold restrictions and bind it to the security field of processes.
   - Rewrite Security Server to make it able to handle multiple restrictions, by looking up in a list.
   - Rewrite `umbrella_scr` system call to set multiple restrictions from user-space.
   - Add the possibility to restrict network usage.
3. (**RELEASED 2004-03-31**) The third release will mainly concentrate on code cleanup, implementing a dynamic data structure for the bit-vector and writing more documentation.
   - Code cleanup will prefix the umbrella files by `umb_` and function names will be truncated to something understandable and of writable lengths. Furthermore, comments the code will be better structured with the comments in the .h files and only internal comments in the .c files.

- The documentation will mainly include some examples of coding, i386 vs. iPAQ implementation, how restrictions work and how to apply User-mode Linux for testing.
- Ported Umbrella to the iPAQ 5550

4. Implement restrictions for process signaling.
5. Implement extended attributes in the JFFS2 file system. This major task is believed to take a considerable amount of time.
6. Begin the integration of execute restrictions from signed files.
   - Data structures to hold the appropriate security fields will be implemented, and placed at the security field of the file system.
   - For test purposes a system-call will be implemented in order to set this security field from user space.
7. Start the integration of public key encryption into the kernel.
   - The ability for the kernel to catch incoming files and, if existing, decrypt its security fields and assign these to the actual files in the file system.
   - By this simple step we are able to transfer a signed file to the file system. This automatically get the right restrictions set, and when executed, these restrictions will apply for the process.
8. Release user space tools.
   - The Umbrella tool box will provide a set of programs for automatically generating a correct signed file from any given executable file. The signed files are described in detail here.

Bibliography

[1] Handhelds Will Get Hammered.
http://www.pcworld.com/news/article/0,aid,17526,00.asp, July 2000.

[2] CERT Advisory: gv contains buffer overflow in sscanf() function.
http://www.kb.cert.org/vuls/id/600777, October 2002.

[3] Device Profile: Samsung SCH-i519 smartphone.
http://linuxdevices.com/articles/AT4481058519.html, December 2003.

[4] Linux to power most Motorola phones.
http://news.com.com/2100-1001-984424.html, February 2003.

[5] LSM Documentation. http://lsm.immunix.org/docs/, November 2003.

[6] Motorola's first Linux Smartphone, the A760.
http://www.mobileburn.com/news.jsp?Id=234, February 2003.

[7] OS Security Background: Mandatory Access Control.
http://www.linsec.org/doc/final/node20.html, December 2003.

[8] Definition of Insensitive Flow analysis.
http://www.cl.cam.ac.uk/ jds31/useful/dfgloss.html, May 2004.

[9] DSI - Distributed Security Infrastructure. http://disec.sourceforge.net/, May 2004.

[10] File Viruses. http://www.viruslist.com/eng/viruslistbooks.html?id=25, May 2004.

[11] Gnu gperf. http://www.gnu.org/software/gperf, May 2004.

[12] Hash Functions and Block Ciphers.
http://burtleburtle.net/bob/hash/index.html, May 2004.

[13] Linux Kernel Privileged Process Hijacking Vulnerability.
http://www.securityfocus.com/bid/7112/info, May 2004.

[14] Man page for clone(2).
http://www.die.net/doc/linux/man/man2/clone.2.html, May 2004.

[15] Man page for ptrace(2).
http://www.die.net/doc/linux/man/man2/ptrace.2.html, May 2004.

[16] Ptrace contains vulnerability allowing for local root compromise.
http://www.kb.cert.org/vuls/id/628849, May 2004.

[17] Report: Threats Coming from all Sides.
http://itmanagement.earthweb.com/secu/article.php/3326731, 2004.

[18] Symbian leads smartphone market.
http://www.symbian.com/press-office/2004/pr040322b.html, March 2004.

[19] W32.Beagle.A@mm.
http://securityresponse.symantec.com/avcenter/venc/data/w32.beagle.a@mm.html, May 2004.

[20] W32.Netsky.C@mm.
http://securityresponse.symantec.com/avcenter/venc/data/w32.netsky.c@mm.html?Open, May 2004.

[21] Marshall D. Abrams, Leonard J. LaPadula, and Ingrid M. Olson. Building Generalized Access Control on UNIX. pages 65–70. MITRE, USENIX, August 1990.

[22] Mike Ashley. The GNU Privacy Handbook.
http://www.gnupg.org/gph/en/manual.html, May 2004.

[23] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[24] Antony Edwards, Trent Jaeger, and Xiaolan Zhang. Runtime Verification of Authorization Hook Placement for the Linux Security Modules Framework. November 2002.

[25] Jeffrey S. Foster, Manuel Fahndrich, and Alexander Aiken. A theory of type qualifiers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 192–203, 1999.

[26] Timothy Fraser. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. pages 230–245, May 2000.

[27] Timothy Fraser. LOMAC: MAC You Can Live With. Boston, Massachusetts, USA, 2001. USENIX.

[28] Andreas Grünbacher. POSIX Access Control Lists on Linux. Usenix Annual Technical Conference, June 2003.

[29] Brian Hatch. An Overview of LIDS.
http://www.securityfocus.com/infocus/1496, November 2003.

[30] Lowell Johnson, Berry Needham, Charles Severence, Lynne Ambuel, and Casey Schaufler. Ieee standard 1003.1e. 1999.

[31] Greg Kroah-Hartman. Using the Kernel Security Module Interface. *Linux Journal*, November 2002.

[32] Aaron Krowne. Good hash table primes.
http://planetmath.org/encyclopedia/GoodHashTablePrimes.html, May 2004.

[33] Rick Lehrbaum. Sneak preview: A Linux powered wireless phone.
http://linuxdevices.com/articles/AT5512478189.html, June 2003.

[34] Bin Liang. Linux kernel: Problem: A 2.6.0-test11 capability lsm module serious bug. http://seclists.org/lists/linux-kernel/2003/Dec/1680.html, May 2004.

[35] Tim Newshan. Format String Attacks. Technical report, Guardent, Inc., September 2000.

[36] Aleph One. Smashing The Stack For Fun And Profit. http://www.cs.ucsb.edu/ jzhou/security/overflow.html, May 2004.

[37] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers.* O'Reilly, 2 edition, 2001.

[38] Bruce Schneier. *Applied Cryptography.* John Wiely & Sons, Inc, 1994.

[39] Stephen Smalley. Configuring the SELinux Policy. Technical report, NSA, February 2002.

[40] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing SELinux as a Linux Security Module. Technical report, NAI Labs, May 2002.

[41] Eugene H. Spafford. The internet worm program: An analysis. Technical Report Purdue Technical Report CSD-TR-823, West Lafayette, IN 47907-2004, 1988.

[42] Ray Spencer, Peter Loscocco, Stephen Smalley, Mike Hilbler, David Andersen, and Jay Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. Technical report, Secure Computing Corporation and NSA and University of Utah, 1998.

[43] William Stallings. *Operating Systems - Internals and Design Principles.* Prentice Hall, fourth edition, 2000.

[44] TheFreeDictionary.com. Copy-on-write. http://encyclopedia.thefreedictionary.com/Copy-on-write, May 2004.

[45] Leendert van Doorn, Gerco Ballintijn, and William A. Arbaugh. Signed Executable for Linux. June 2001.

[46] David Woodhouse. JFFS : The Journalling Flash File System. Technical report, Red Hat, Inc., 2001.

[47] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. USENIX Security Symposium, 2002.

[48] Marek Zelem and Milan Pikula. ZP Security Framework. Technical report, Faculty of Electrical Engineering and Information Technology Slovak University of Technology in Bratislava.

[49] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using CQUAL for Static Analysis of Authorization Hook Placement. June 2002.