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

Umbrella

Security for Consumer Electronics

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

Project Report
Aalborg University
January 2005

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
Security for Consumer Electronics
We can't prevent the rain ...
– But we don't get wet!

PROJECT PERIOD:

DAT7,
September –
December 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: 16

REPORT PAGES: 94

APPENDIX PAGES: 32

TOTAL PAGES: 126

SYNOPSIS:

This project report describes the *Umbrella* security mechanism designed for consumer electronic devices running Linux. Umbrella implements a combination of process based mandatory access control and authentication of binaries.

Umbrella is implemented on top of the Linux Security Modules framework in Linux kernel 2.6.

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 set of 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 provide 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 tampered with, and take appropriate action.

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

Authors

Søren Nøhr Christensen

Kristian Sørensen

Michel Thrysøe

Contents

1 Introduction

This report is documentation for the Umbrella Project, which is a security mechanism for Linux on consumer electronic devices like ranging from mobile phones and handheld computers to settop boxes, etc, henceforth called consumer electronic devices or CE devices. Umbrella implements a combination of process based mandatory access control and authentication of binaries. The idea for Umbrella emerged when investigating ways to enforce security on CE devices.

The Umbrella Project is developed as our master's thesis at Department of Computer Science, Aalborg University, Denmark and the work has been continued as a part of our semesters on Industrial Computer Science.

Background research in security for CE devices was performed in the autumn semester 2003 together with the initial design of Umbrella. The design and implementation of Umbrella was done in the spring of 2004, where we in June completed our master's thesis.

The following two semesters is at Industrial Computer Science at Aalborg University. In autumn 2004, most of the implementation was completed. Furthermore, much time was spent on discussing Umbrella with various partners outside the university. During this semester, we have cooperated with TDC[1], for making a proof-of-concept implementation of Umbrella for their embedded Linux-based alarm box. During a week in October, we attended a meeting of the Security Working Group of the Consumer Electronics Linux Forum[2] in Princeton, New Jersey. At the meeting we presented Umbrella and participated in a discussion on security issues for CE devices.

1.1 Linux on CE Devices

As the Umbrella project started it was aimed at developing a security mechanism for the Symbian[3] operating system, since this is the leading operating system for handhelds and mobile phones [16]. It was, however, not possible to gain access to the Symbian source code, which is why Linux was chosen instead. After working with the Umbrella project for 12 months, it is clear that the Linux approach is becoming increasingly more interesting as Motorola, NEC, Samsung, Panasonic and others have released smart phones based on Linux

[1] www.tdc.com
[2] www.celinux.org
[3] www.symbian.com

[7, 4]. Linux for embedded systems is in rapid development along with support for the ARM architecture which is used in many CE devices. An example is HP, who is actively supporting this development.

The interest in Linux on CE devices is shown clearly in the work of the Consumer Electronics Linux Forum (CELF), which have been formed to work with promoting and researching into Linux in consumer electronics.

There are several reasons for this interest in using Linux on handhelds and in other kinds of consumer electronics, including reduced time to market [5, 39], openness and the fact that it is free. Umbrella provides the feature of protecting such consumer electronics, including its devices and files against various attacks.

In a commercial operating system (OS) for CE devices, if a company requests new feature, the owner of the OS might use this feature in their own products and delay the release of new versions of the OS. This might result a situation where the company with the idea is unable to make money from their idea. This situation will not occur with open source OSs like Linux. Everyone can add code to the operating system suiting their own requirements, without having to disclose ideas to potential competitors.

1.2 Umbrella Security Improvements

CE 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 CE devices, which raises a demand for security measures within the operating system. Umbrella implements a combination of process based mandatory access control and authentication of binaries 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.

A main area for improvement of security 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 [34]. By enforcing MAC on processes, 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 the principle of least privilege. This limits the harm of e.g. process hijacking.

Another area for improvement is the integrity of executables, where digitally signed binaries (DSB) are introduced. The vendor signs the file with his private key, making it possible to verify the origin and integrity with the corresponding public key. A checksum 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 CE devices.

The basics of the above ideas for improvement is illustrated in Figure 1.1. When

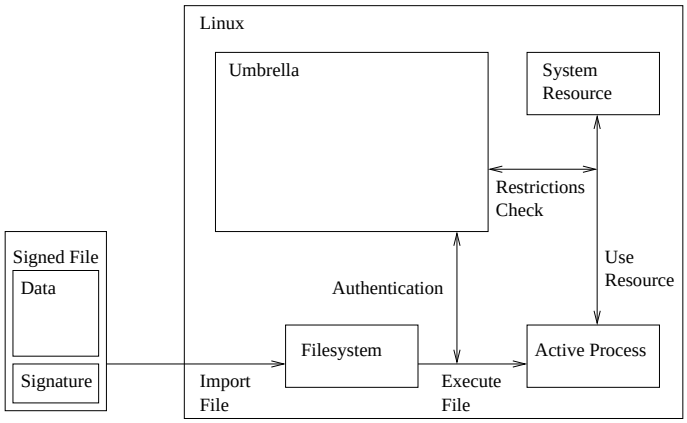


Figure 1.1: The basics of Umbrella.

a file is executed Umbrella authenticates the file. If the authentication is positive, the binary is allowed execution and given the set of restrictions which was included within the signature. If the authentication is negative the process is either rejected or sandboxed. When an active process tries to access a system resource, Umbrella 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 consumer electronics 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 CE Devices

Two main security threats against CE 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, performing a denial of service attack. 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[4] 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 [42] and [48].

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 will overflow into adjacent buffers, thereby corrupting 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 privileges 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.

Another form for overflow attacks are heap overflows [54].

Format String Attacks

Format string attacks is another way of tampering with the contents of the stack, done by exploiting unchecked format string input [41].

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

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

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

```
1 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.

The danger in this, is that the programmer passed a string to `printf` 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

[4] www.cert.org

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 [41].

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 CE devices that is becoming increasingly relevant, is that of malicious software, such as viruses, worms, trojans, backdoors, etc. [15, 1, 18, 17]. Recently the viruses Capir [30] and Skulls [29] have been spread targeting Symbian based devices. With the increasing use of email on CE devices, handhelds and mobile phones, and the fact that such 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 privileges 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 sandbox.

1.4 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 [22], 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 43.000 visits since the public launch February 3rd 2004. Access to stable releases are available from the Umbrella project site[6]. Since the first version was released February 3rd, more than 750 downloads have been performed.

[5] umbrella.sourceforge.net
[6] www.sourceforge.net/projects/umbrella

1.5 Report Overview

Chapter 2 contains a description 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 and issues regarding possible optimizations of Umbrella.

Chapter 5 includes preliminary benchmarks of Umbrella and the Umbrella API including examples. 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.

Appendix A contains information regarding practical issues involved in installing and using Linux on a HP iPAQ.

Appendix B gives descriptions of Linux distributions and projects for consumer electronics.

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

Appendix D lists 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 [8].

- 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 $\{read, write\}$. This means that subject S_2 can perform a *write* operation on object O_1 if and only if $write \in M[S_2, O_1]$. Figure 2.1 depicts this example and the general approach to mandatory access control.

A presentation of several security project follows.

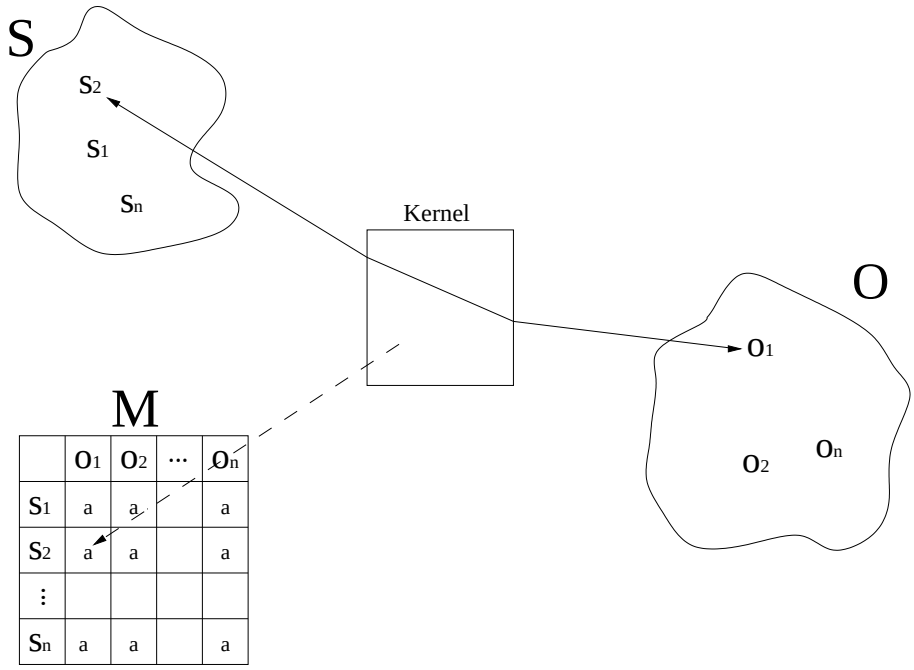


Figure 2.1: The general MAC approach.

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 [57], 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

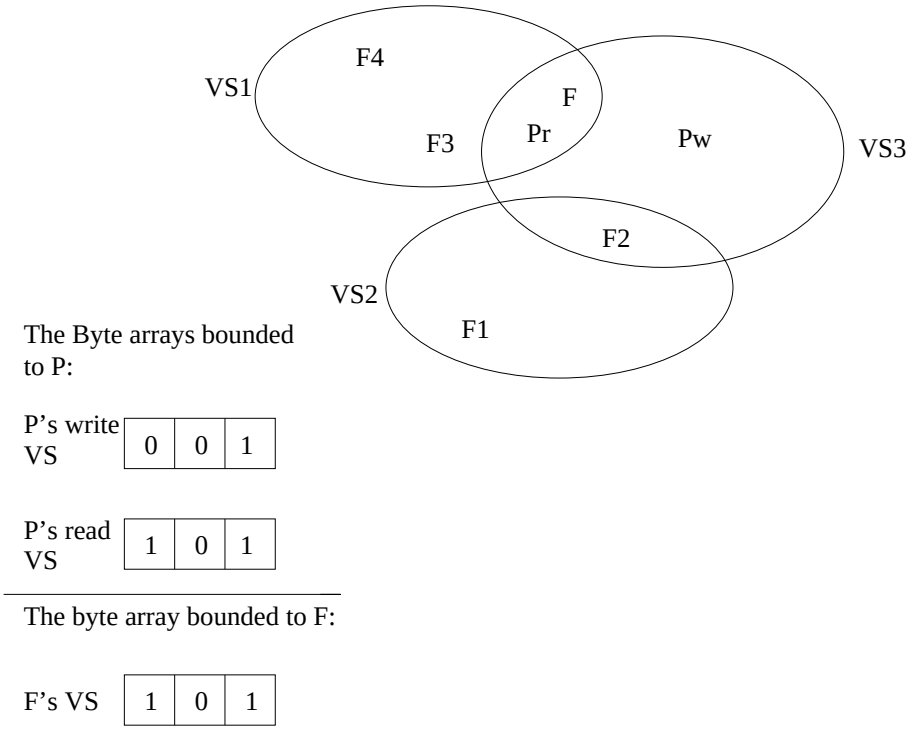


Figure 2.2: Example of Virtual Spaces.

$A = read, write$. 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

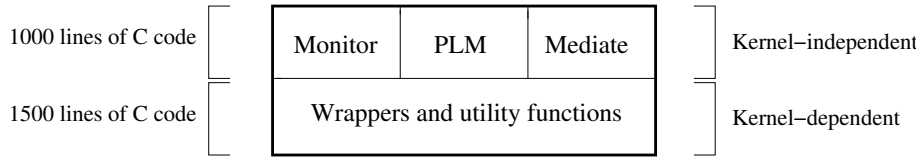


Figure 2.3: Loadable kernel module architecture.

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 [33] 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 [32] 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.

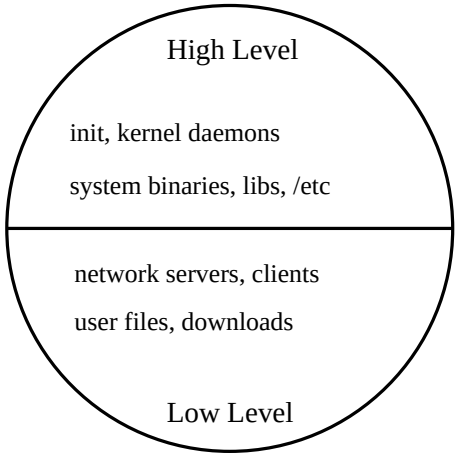


Figure 2.4: The two levels in LOMAC.

A description of the LOMAC wrapper functions can be found in [33], 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 [33] 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) [47] is based on the Flask security architecture [49], 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.

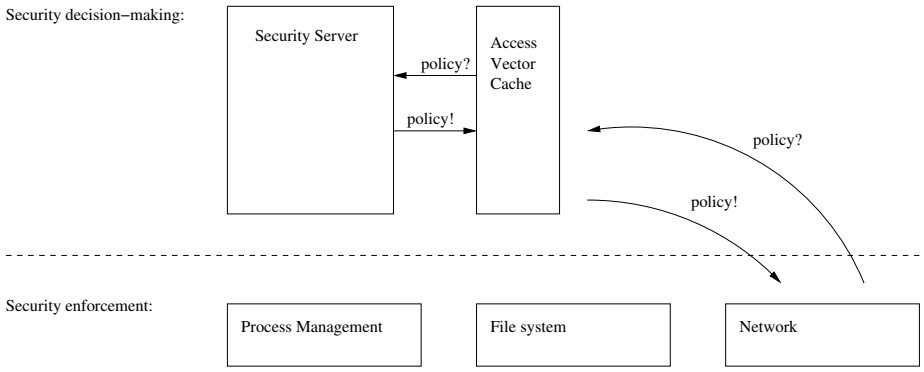


Figure 2.5: Component overview of Security-Enhanced Linux.

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 [46].

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 im-

plementation of SELinux can be used in designing a security measure optimized for Linux on consumer electronic 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 [35].

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 BSign

BSign embeds secure hashes and digital signatures into files for verification and authentication. Currently, BSign works with target file types that are of the ELF format: executables, kernel modules, shared and static link libraries.

BSign calculates a SHA1 value of the binary and encrypts this using a private key. This ensures both integrity and authenticity. The encrypted hash is stored on the file, by adding another section to the files ELF header.

BSign is a program which uses SHA1 [3, 21] for hashing of files and it uses GPG [21] for asynchronous cryptography; the project is hosted at the Debian Linux web site.

2.7 DigSig – Security Project

The DigSig project is a part of the Distributed Security Infrastructure [10] project, which is a project that is supported by Ericsson Research Canada. The DigSig project makes it possible to sign binaries and thereby ensure integrity of

these. This approach is taken to counter the class of attacks where malicious code is hidden in code with a legitimate purpose. DigSig is using the BSign project to sign the binaries and a kernel module to verify signatures.

In the DigSig approach, binaries are not signed by vendors, but control is handed over to the local system administrator. She is responsible to sign all binaries that she trusts with her private key. Then, those binaries are verified with the corresponding public key. This means that it is still possible to use favorite binaries, meaning no change in habits. Basically, DigSig only guarantees two things.

1. If you signed a binary, nobody else than you can modify that binary without being detected.
2. Nobody can run a binary which is not signed or badly signed.

One should be extremely careful not to sign untrusted code; if malicious code is signed, all security benefits are lost. More information on the DigSig project can be found in [20].

2.8 Immunix SubDomain

Immunix SubDomain is an operating system extension designed to prevent vulnerability misuse. This is done by providing a least privilege mechanism, that acts on programs as opposed to users. SubDomain complements the systems original security, making it possible to secure programs independent of the users running the program. Furthermore SubDomain can confine software components at a finer granularity than processes, such as modules for a webserver. The security policy of SubDomain is gathered in a single file and the policies themselves are fairly easy to understand and write.

```
foo {
 /etc/readme      r ,
 /etc/writeme     w ,
 /usr/bin/bar     x ,
}
```

The above rule specifies a subdomain where the program `foo` has read access to the file `/etc/readme`, write access to the file `/etc/writeme` and the possibility of executing the file `/usr/bin/bar`.

SubDomain is implemented as a loadable kernel module [36] and it is deeply based on the Linux Security Modules framework. SubDomain is based on the work done by Crispin Cowan et. a.l in [26] and it is a closed source project

2.9 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 [19]. 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.

2.10 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 CE 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 of consumer electronics. In Medusa the Security Decision Center is implemented as a user space daemon, to ease the implementation. It is unclear how Medusa protects the user space security daemon from other user space applications, but it is clear that it would be better protected in the kernel.

LOMAC was designed as a transparent MAC implementation for use in existing systems. For use in consumer electronic devices the design of the LOMAC system will often be 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, 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 consumer electronic device intended for normal users.

SubDomain is much simpler to configure and understand than e.g. SELinux, however it is still too complex for CE devices. Furthermore, the system offers no protection against malicious binaries and finally it is closed source.

2.11 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 consumer electronic devices 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 permissions in 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 before they are executed.

The security policy of Umbrella is stored decentralized. Within programs, a *restricted fork* is available for developers, that enables restricting dangerous children to least privilege. Furthermore, the policy is stored in digitally signed binaries, which contain restrictions to be applied on time of execution.

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.

By sandboxing 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.

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 based on the LSM security framework, to provide independence of different versions of the 2.6 Linux kernel series.

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. Digitally signed binaries are used for authentication and distributing the security policy with the binary.

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 Consumer Electronic Devices

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

New handhelds and mobile phones 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. CE devices are more and more integrated into everyday life and users can store personal information and confidential data on their device. 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.

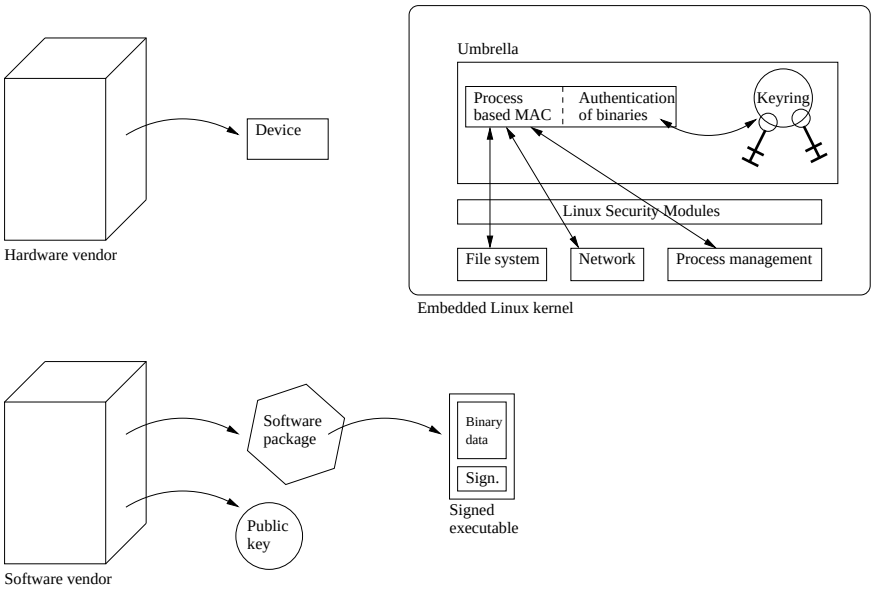


Figure 3.1: Top level design of Umbrella. In the upper left corner is a hardware vendor who produces embedded devices. These run embedded Linux with Umbrella. The embedded kernel is depicted in the upper right corner. Umbrella consists of two things, namely process based mandatory access control and authentication of binaries. The process based MAC controls access to objects in the kernel such as file system, network and process management. This is done through the Linux Security Modules. In the lower left corner a software provider produces packages. In these, the executable file is signed. When this file is executed on the system the public key of the vendor is looked up in the keyring. If the program can be authenticated, the resulting process of the binary may run. If not, the program may be denied execution or the process may be sand-boxed.

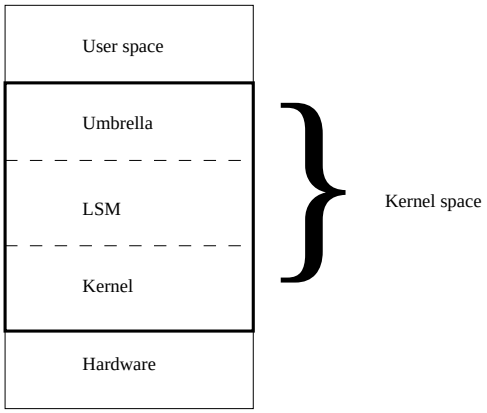


Figure 3.2: Placement of Umbrella in the Linux kernel.

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; every object and subject still need to be accounted for.

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*, where the access control measure is enforced by restrictions only. Thus, every running process have access to the entire system, except a given set of restrictions. The discretionary access control still applies, i.e. Umbrella co-exists with DAC. If either Umbrella or DAC denies access to a given resource, access is denied.

The motivation for using restrictions as opposed to permissions, like traditional MAC systems, is that the number of elements that a process may not access is very limited, as opposed to what the process may access. Also, prohibited group of elements tend not to vary between different versions of the programs, where the group of elements that the program may access mostly expands through out the releases of new versions.

Umbrella simplifies the making of security policy in general. Making restrictions access to files in e.g. `/usr/bin` will cause a high number of restrictions, as most programs do not need access to more than a few of these binaries. However, the idea of restrictions is to set restrictions on what the program *may not, under any circumstances, access*. If the given process is owned by *root* and no restriction on `/usr/bin` is present, the process has indeed capabilities of tampering with the binaries. However, as these are digitally signed Umbrella will detect this when executing these files, and then either simply deny execution or sandbox processes created, this is elaborated in Section 3.3.

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`, has 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 through a restricted fork, which allows enforcement of the principle of least privilege.

3.2.2 Restrictions

Restrictions are conceptually 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.

We denote the static restrictions, i.e. the bit-vector, non-file system restrictions (NFSR) and the dynamic restrictions, i.e. the paths in the file system, file system restrictions (FSR).

When the process tries to access a resource the kernel first consults the discretionary access control and if access is denied there, this decision is returned to the process. If not, the NFSR and FSR are checked. If either denies access to the resource, access is denied.

3.2.3 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 *any given 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 immediately 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 if any exists. Execute restrictions and signed binaries 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 for execution before all restrictions are set.

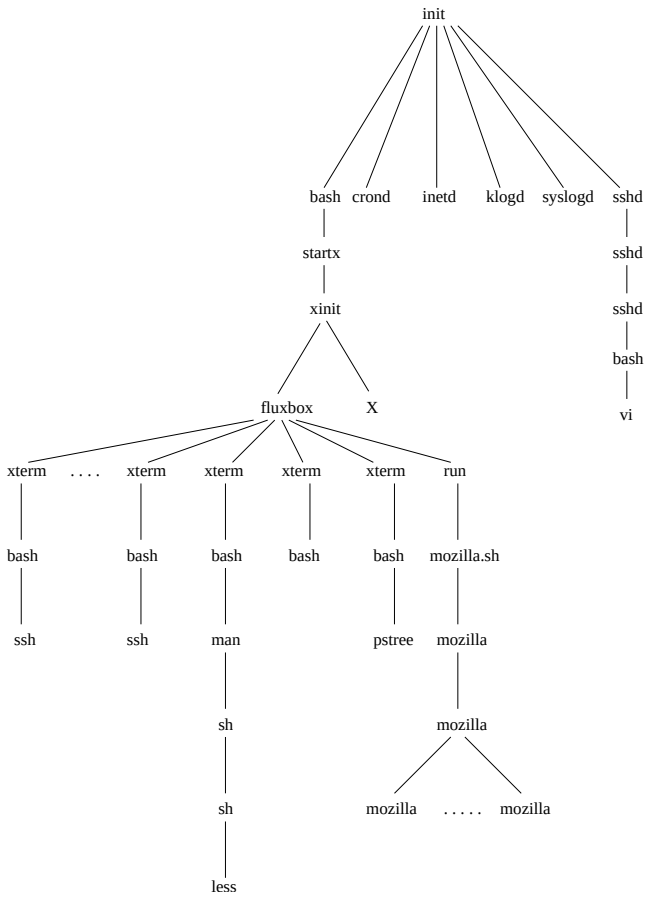


Figure 3.3: Example of the Linux process structure. When a Linux system boots and the kernel is loaded, the first process created is the `init` process, which is executed from `/sbin/init`. The root node is thus the `init` process. This process spawns a number of daemons. The Secure Shell daemon (`sshd`) 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 (`xterm`) running a shell (`bash`). In the right branch of the children of `fluxbox`, a mozilla browser is started, which also has a number of children.

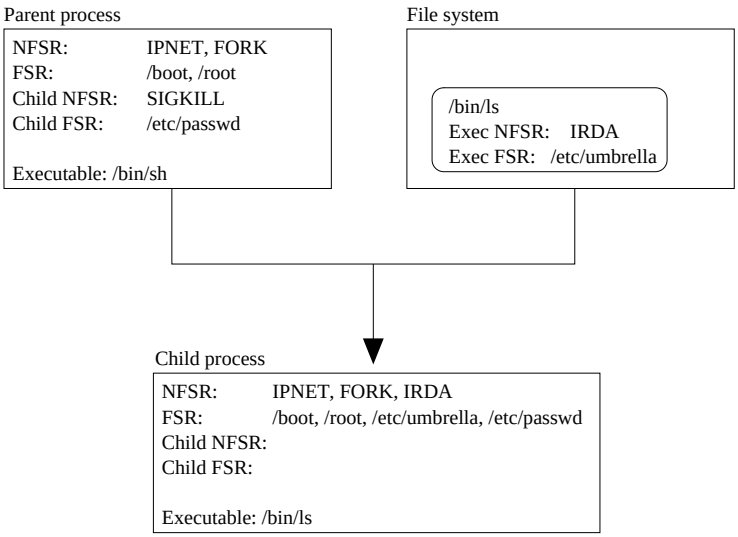


Figure 3.4: The procedure of creating a new child process and setting its restrictions.

3.2.4 Data Structures

The restrictions in the bit-vector holding the non-file system restrictions for each process is set directly in the vector. To ease this each index is bound to a string by `#define` statements. These can be found in the second column of Table 3.1 on the next page.

The file system restrictions are stored in a tree structure [24], consisting of elements of a path. An example is depicted in Figure 3.5 on the facing page, where restrictions are on the paths `/bin/ls`, `/home`, `/usr/bin` and `/usr/lib`. This is fixed for a running process. Thus, there is no way of removing restrictions from a running process but new restrictions may be added to a child, when this is forked.

The following pseudo describes the algorithm for adding a new restriction. The path has the form `/name_1/name_2/name_3/.../name_n`. The parameters to the function is a pointer to the root node and the restriction. In line 11, the situation where a more general restriction is added. E.g. in Figure 3.5, if the restriction `/usr` is added, the subtree containing `bin` and `lib` shall be removed.

```
1  if (successors != NULL) {
2      extract the first name_i and remove it from the path;
3      compare name_i to each the successors;
4
5      if (none of the successor match name_i) {
6          add name_i to successors;
7          create the rest of the path in the fsr;
8      }
9  } else return;
10
11 free all the successors recursively; replace successors by NULL;
```

The algorithm for checking restrictions is outlined in the below pseudo code.

```
1  while (path is not empty) {
```

Index	Restriction	Comment
1	SIGKILL	Ability to send kill signal (signal 9)
2	SIGTERM	Ability to send termination signal (signal 15)
3	SIGQUIT	Ability to send quit signal (signal 3)
4	SIGHUP	Ability to send hangup signal (signal 1)
5	SIGTRAP	Ability to send trap signal (signal 5)
6	SIGALRM	Ability to send alarm signal (signal 14)
7	SIGCHLD	Ability to send child stopped signal (signal 20)
8	IPNET	All networking through IP sockets
9	IRDA	All networking through infrared devices
10	BLUETOOTH	All networking through bluetooth devices
11	FORK	Ability to fork new processes

Table 3.1: Non-file system restrictions in Umbrella.

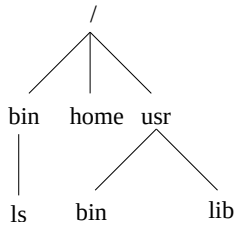


Figure 3.5: Tree structure for storing FSR.

```
    if (successors != NULL) {
        extract the first name_i and remove it from the path;
        compare name_i to each the successors;

        if (none of the successor match name_i)
            return allowed;
    }
    else
        return denied;
}
return allowed;
```

3.2.5 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, networking protocol or signal. If so, access is denied. In the following how restrictions are set on processes is elaborated.

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. Also it is possible for a vendor to assign restrictions to a binary (called execute restrictions), which are applied to the process created when this binary is executed. Execute restrictions will be elaborated later in this section. Below two examples of restrictions for new processes are given.

If a thread is only rendering a picture, it should have all NFSR restrictions set from Table 3.1 plus a restriction on root of the file system from `/`. If the process is hijacked, the hijacker have no access to the file system, nor the network, sending signals 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 email attachments. 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.

Involving the Vendors

When a program is to be distributed to Umbrella protected systems, the software vendor can specify execute restrictions for this binary. Execute restrictions consist of 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. The authentication is elaborated in Section 3.3 on the next page.

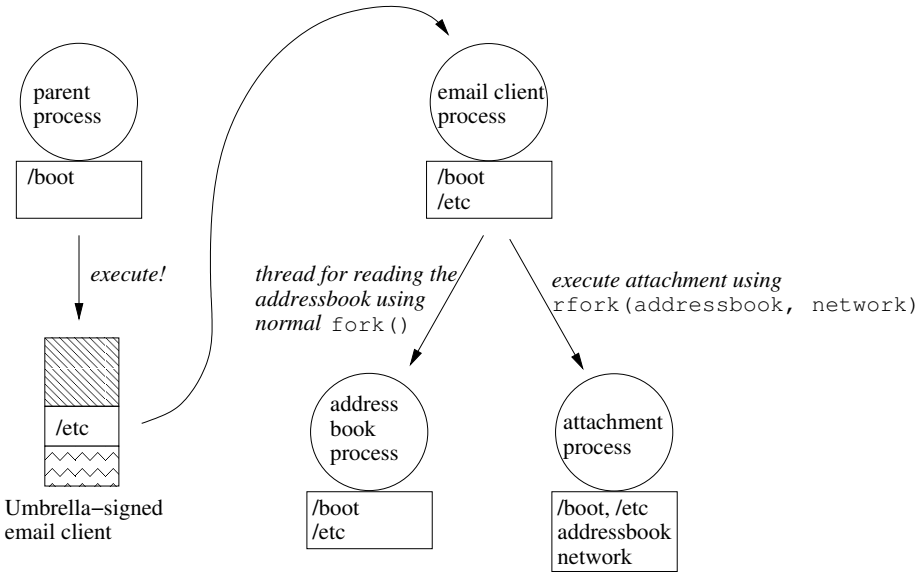


Figure 3.6: Restrictions applied in practice.

On Figure 3.6 the execute restrictions, inheritance and restricted fork is illustrated. A given process is running with the restriction `/boot`. This process executes a vendor signed email client with the execute restriction `/etc`. Thus, the resulting process inherits the restriction from the parent process and gets assigned the execute restriction, depicted in the upper right corner of the figure. The developer of the email client writes the code for executing attachments. As the attachment may possibly be dangerous the developer chooses to execute it using the `rfork`, where new restrictions are set on the `addressbook` and the `network`. The main email client process also executes a new thread for reading the address book. This is done using a regular `fork` and thus this process simply inherits the restrictions of its parent (the email client process).

3.3 Digitally Signed Binaries

Now the focus shifts from the process based MAC to the authentication of binaries. Binaries entering a system must be authenticated. This serves a dual purpose; the authentication of binaries provides protection from executables that may be harmful to the system and the digital signatures provide a way of importing restrictions to the system. The following sections elaborate on the design of the digitally signed binaries, how these are utilized to prevent possibly malicious executables from compromising the system, and how restrictions are introduced through these.

In the following, a *signature* denote a hash value and a set of execute restrictions that is encrypted with a private key. A *digitally signed binary* or DSB denote a binary and its associated signature, as shown in Figure 3.7 on page 38. The *security information* associated with an executable on an Umbrella file system is

the associated execute restrictions. This information is stored in the executables LSM *security field*.

3.3.1 Signature Design

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

- Signatures must include restrictions for use on time of execution.
- The use of signatures must be transparent to the user.
- Security information needs a persistent and secure storage.

In the process of investigating possible models for DSB, several possible solutions are discussed. A solution where the signature is kept in a separate file, another possibility is to modify the executable linkable format (ELF) [52] to include the signature and finally the option of simply appending the contents of the signature to the end of the binary. Each of these possibilities as well as some combinations are discussed below.

Signature Models

This and the following section discus various designs of introducing DSBs in Umbrella. The actual solution can be found in Section 3.3.2 on the facing page.

The first discussed solution to the DSB problem, is to transfer the signature to the device in a separate file. The implementation of such a solution would not require any changes to ELF, which clearly is an advantage. However, several problems exists with transferring the signature to the device 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 binary. 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 binary enters the file system and if an associated signature was found, the signature would be copied to the binary's secure extended attribute in the file system. If an associated signature is not found, the binary will be sandboxed. Alternatively, the transfer of the signatures could wait until the binary is executed for the first time. The major drawback of this two file solution is the maintenance of the signature cache as well as the lack of transparency.

The problems found with the two files solution, suggested that solution where the signature is included in the binary is preferable. A possibility was to modify ELF like done in the BSign project and the work described in [53]. This solution would definitely offer a simple way of transferring signatures for executables to the 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 the need for modifying these. However, virtually all binaries today follow the ELF format. Regarding scripts, one

argument is that scripts are a way to execute other binaries, and if these binaries are properly restricted through their signatures, no further restrictions are needed on the scripts.

A third solution is to simply 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 could be a problem regarding scripts, since the tag must be enclosed in something that makes all scripting languages of interest ignore it.

Signature Storage

Common to the proposed solutions above is the need for considering a secure and persistent storage for the security information. The checksum and signing with a private key ensures that it is not possible to tamper with a binary without detection. In the solution where the signature is embedded in the ELF header, both persistent and secure storage is achieved. There is however a problem with performance in this scenario. Whenever a DSB is executed the signature must be fetched from disk and decrypted, for binaries that are executed often this overhead is not acceptable, a faster storage option is needed.

One option is use the extended attributes in the file system. The extended attributes are a void field in the file system that can be used to persistently store attributes related to files. This storage is also secure, since it is only directly accessible through the kernel. The extended attributes can be used to cache the security information of a file, and after the first execution of the file the security information can be fetched from this field, without the need for decryption. One major drawback exists, the extended attributes feature is *not* implemented for the journalized flash file system, which is the common file system on consumer electronic devices [55].

The solution selected for Umbrella is appending the signature to the end of the binary and utilizing BSign to insert a singed hash in the ELF header. The storage issue is solved by using a LSM field as cache. The approach is transparent and flexible. This is elaborated below along with details on how security information is transferred from the signature to the security field in the file system.

3.3.2 Signatures in Umbrella

The rest of this chapter describes the actual design of DSBs in Umbrella.

The layout of a DSB can be seen in Figure 3.7. The signature consist of a vendor ID, a set of execute restrictions and a hash of the original binary including ID and restrictions. The hash is encrypted with the private key of the vendor. The vendor ID is used to lookup the corresponding public key. The hash value is used for ensuring that the binary or restrictions has not been tampered with.

Figure 3.8 elaborates on the creation and layout of a DSB. The restrictions associated with a binary is appended. After this BSign is used, first it creates a

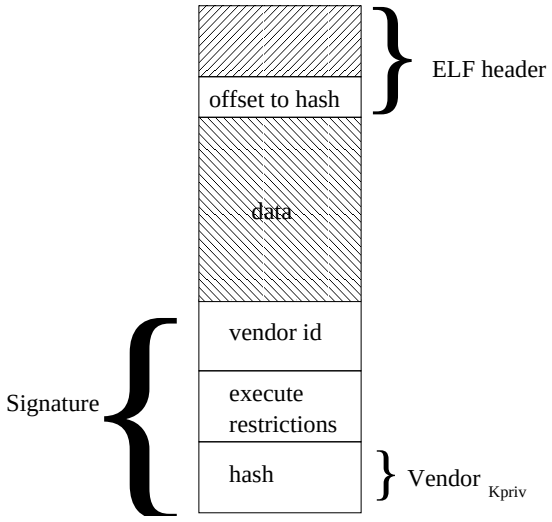


Figure 3.7: Layout of a digitally signed binary.

SHA1 checksum of the binary and the appended restrictions. This checksum is signed with the vendors private key, to provide an authentication mechanism. This is appended to the binary with restrictions.

In a system protected by Umbrella, two types of binaries can enter the file system, namely signed and unsigned. Both are stored on the file system and no further action is taken until the binary is executed. If an unsigned binary is executed, a default action is taken. This default action can either be execution in a sandboxed environment or simply denied execution. The default action can be specified by the vendor of the CE device.

Figure 3.9 shows the process for verifying a binary prior to execution. The LSM hook `bprm_check_security` is called before a binary is executed, and the implementation of this hook is used for verifying the binary. First step is decryption using the vendors public key, to authenticate the origin of the binary. Next step is the integrity check, using the checksum of the binary with restrictions. If either fails, the default action is taken, reject execution or sandbox the process. If both succeeds the binary is executed with the associated restrictions.

The storage of the signatures on the binaries is both secure and persistent. Secure because of the checksum, and persistent since binaries are stored on the disk. This solution, however, lacks in performance since the decryption and checksum calculation must be performed on each execution of a DSB. This is solved by using the LSM security field that is associated with each file in the system. This field is protected by the kernel and therefore secure. It is not persistent, so it is only used as a cache. On Figure 3.10 it is shown how this optimization works.

First execution of a DSB is shown on the left side of Figure 3.10 and is identical to the procedure seen in Figure 3.9 except that the restrictions of the binary is stored in its associated LSM security field before execution. On the right side

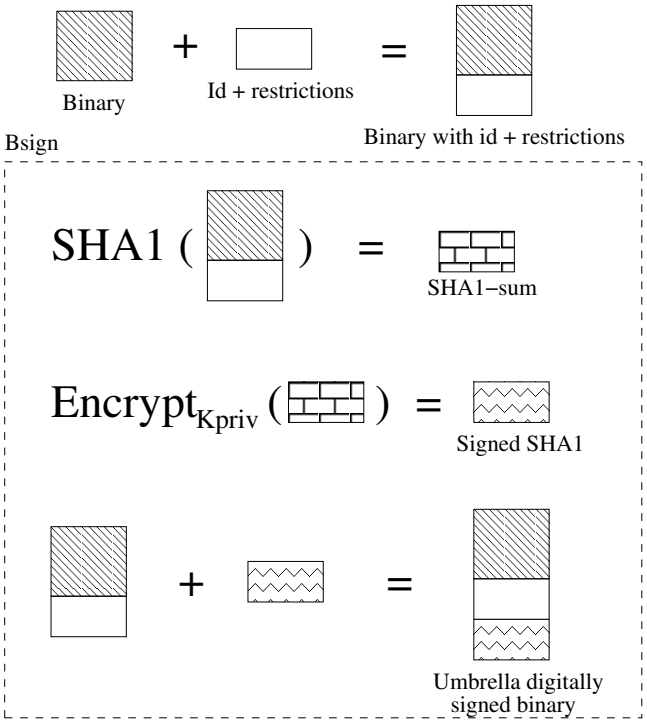


Figure 3.8: Creating a DSB.

the procedure for following executions is shown. The binary is fetched from disk, the restrictions are fetched from the LSM field of the binary and applied to the process. After a reboot the security fields are reset and the procedure for first time execution must be taken again. This scheme is especially efficient for mobile phones and PDA's since they do not reboot often.

The scheme described above gives Umbrella a secure and convenient way of handling both signed and unsigned binary files. The origin and integrity of the files are checked and the restrictions associated with the DSB are imported to the system transparent to the user. Below details will be given regarding handling of public keys on an Umbrella system.

3.3.3 Key Handling

The DSB scheme requires that only public keys from trusted vendors is in the system.

Static Approach

One way of handling this is to compile the public keys of trusted vendors into the kernel itself, making it impossible to tamper with them nor add new keys. This approach is feasible if the provider of the device knows what vendors should deliver software to the device at time of roll out. This approach is very easy to

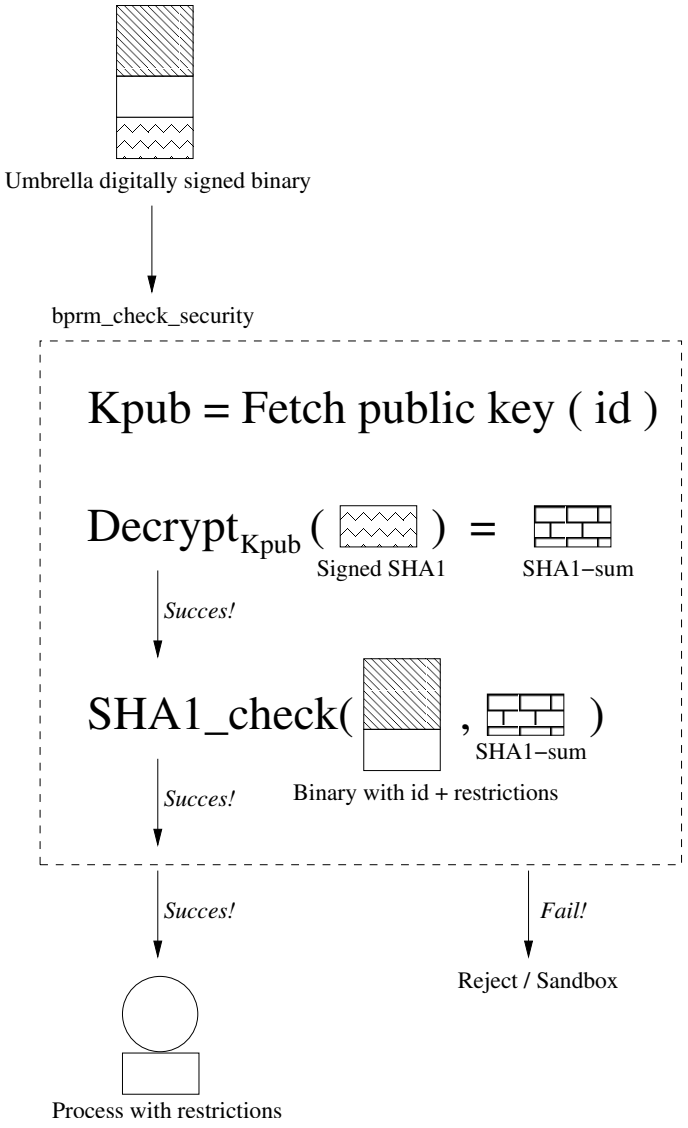


Figure 3.9: Authenticating and executing a DSB.

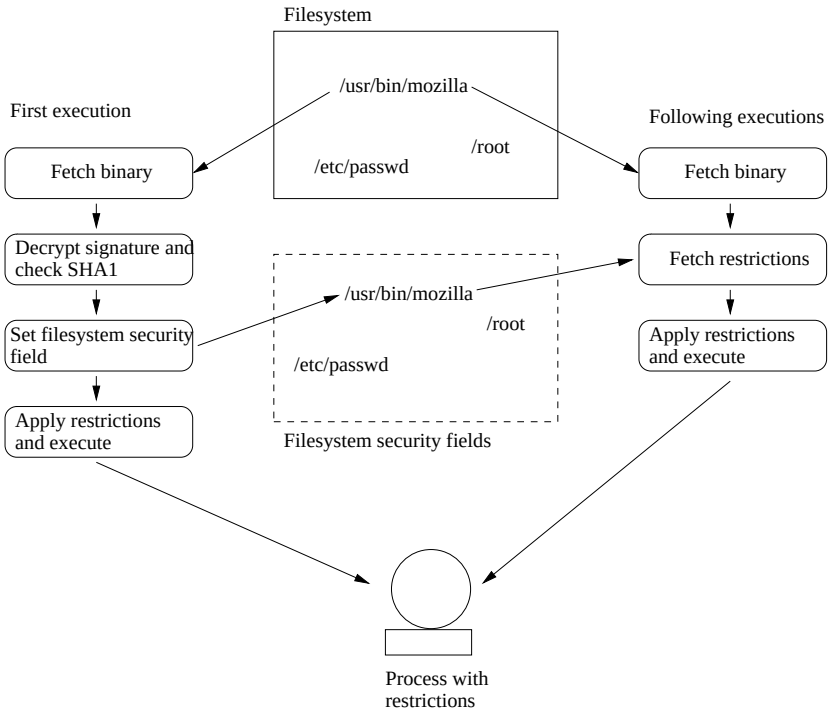


Figure 3.10: Using LSM security fields as cache for restrictions.

implement, but also very inflexible. Another issue is that if the private key of one or more vendors is exposed, it will be possible for an attacker to become "trusted", and keys can only be changed by changing the entire kernel.

Using a Trusted Key Server

A more flexible solution that allows users to add various keys is distribution of public keys is via a trusted key server[45, 21]. A scheme that can be compared to that of Kerberos [25].

The Keyring

On the system public keys most be protected from tampering and substitution. This is done by storing them in a kernel ring protected by the kernel. The kernel writes the key ring to the directory `/etc/umbrella/keyring`. To protect the public keys from unauthorized access on disk, *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.

From Linux 2.6.10-rc3 a option for including digital keys in the kernel was added. This feature can be used to build a keyring by a searchable sequence of keys. By default each process is equipped with access to five standard keyrings: UID-

specific, GID-specific, session, process and thread. We believe that this new feature can be used to implement the Umbrella keyring.

3.4 Conclusion

The goals achieved in the design of Umbrella is to eliminate the access matrix, provide transparency to the user and distribute the security policy with the programs to avoid the global system policy.

The design 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 are the union of restrictions from the parent and the restrictions specified for the child. To ensure integrity of software and ease the configuration of Umbrella digitally signed binaries are introduced. These provide both authentication and means for setting restrictions for processes without user interaction.

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 file names 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 DSB. Lastly optimizations to the current implementation are presented.

4.1 Process Based MAC

The process control block [50] in Linux is defined in the structure `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 a process.

The security field for a process is specified by the structure `security_struct` in `security/umbrella/include/umb_types.h`. The non-file system restrictions and the file system restrictions for the process are specified in `nfsr` and `fsr`. NFSR for the next child process are given in the bit-vector `child_nfsr` and the list of FSR are available in the `child_fsr` array. The next child's FSR are stored in an array of char pointers like the data submitted from user space. When the new child is created, the data in the array is added to the child's FSR tree. The maximum number of additional FSR, that a child can be assigned is defined in `MAX_FSR` as 256. This was found to be a reasonable upper bound on the number of restrictions, given the amount of resources found on most consumer electronic devices.

```
1 struct security_struct {
2     bitvector nfsr;
3     struct fsr fsr;
4     bitvector child_nfsr;
5     char *child_fsr[MAX_FSR];
6 };
```

4.1.1 Non-File System Restrictions

The non-file system restrictions are implemented as a bit-vector. This implementation is very simple and written to suit the specific needs of Umbrella. The complete interface of Umbrellas bit-vector library is found in the header file `security/umbrella/include/umb_bitvector.h` and it is shown below. The complete implementation is found in `security/umbrella/umb_bitvector.c`.

```
1 typedef uint32_t *bitvector;
2
3 void      bv_bit_on  (bitvector vector, int index);
4 bitvector bv_create  (void);
5 void      bv_destroy (bitvector vector);
6 void      bv_or      (bitvector result, bitvector a, bitvector b);
7 void      bv_reset   (bitvector vector);
8 void      bv_print   (bitvector vector);
9 int       bv_testbit (bitvector vector, int index);
```

The functions `bv_create` and `bv_destroy` handles memory allocation and deallocation for bit-vectors. The function `bv_reset` is used to empty the bit-vector `child_nfsr`, when a new child is created and `bv_print` is available for debugging purposes only.

Using the `bv_bit_on` function it is possible to specify any of the restrictions in Table 3.1 on page 33 to a given process. For security checks on a process the `bv_testbit` function is used. Time for lookup of these restrictions are crucial to the performance of a system, as e.g. a very large number of sockets are created during runtime.

In the following a closer look at the two most important functions in the bit-vector library are presented. The function `bv_testbit` is used for fast lookups of NFSR and `bv_or` for inheritance.

The implementation of `bv_testbit` is simple. Using a shift operation the appropriate bit is located from the index and the test is performed using a logical and-operation, returning if the bit is set or not.

```
1 int bv_testbit(const bitvector vector, int index) {
2
3     int tmp = *vector;
4
5     tmp >>= index;
6     return tmp & 1;
7
8 }
```

Looking at the function in assembler.

```
1  bv_testbit:
2      pushl   %ebp
3      movl    %esp, %ebp
4      subl    $8, %esp
5      movl    8(%ebp), %eax
6      movl    (%eax), %eax
7      movl    %eax, -8(%ebp)
8      movzbl  12(%ebp), %ecx
9      leal    -8(%ebp), %eax
10     sarl    %cl, (%eax)
11     movl    -8(%ebp), %eax
12     andl    $1, %eax
```

```
13      leave
14      ret
```

We see that this function is very simple, thirteen machine instructions and no conditional jumps. Performance of this function is important for fast lookups.

Another important function in the bit-vector library is `bv_or`, which is used for inheritance of restrictions and is therefore called every time a new process is created. This function simply makes a bit-wise OR between two bit-vectors, storing the result in a third bit-vector.

```
1  void bv_or(bitvector result, bitvector a, bitvector b) {
2
3      *result = *a | *b;
4
5  }
```

As seen from the below assembler language snippet it only takes 10 machine instructions to create the new bit-vector.

```
1  bv_or:
2      pushl   %ebp
3      movl    %esp, %ebp
4      movl    8(%ebp), %ecx
5      movl    12(%ebp), %edx
6      movl    16(%ebp), %eax
7      movl    (%eax), %eax
8      orl     (%edx), %eax
9      movl    %eax, (%ecx)
10     popl    %ebp
11     ret
```

4.1.2 File System Restrictions

As stated in Section 3.2.4 the file system restrictions are arranged in trees. The actual implementation is done using linked lists as standard trees [24]. The tree structure comes because of the fact that each node have both a `next` node, and a `succ` node. The pointer `next` points to the next node on the same level in the tree, if any. The `succ` pointer is used to move down through the levels of the tree. The `name` is the directory associated with this node in the list. The list are composed of a number of `fsr` structures, defined in `security/umbrella/include/umb_types.h`.

```
1  struct fsr {
2      char *name;
3      struct fsr *succ;
4      struct fsr *next;
5  };
```

The interface to the FSR is found in `security/umbrella/include/umb_types.h` and seen below.

```
1  struct fsr *umb_fsr_init           (void);
2  int    umb_fsr_insert              (struct fsr *root, char **path);
3  int    umb_fsr_check               (struct fsr *root, char **path);
4  struct fsr *umb_fsr_create_subtree (char **path);
5  struct fsr *umb_fsr_copy           (struct fsr *source);
6  void   umb_fsr_destroy             (struct fsr *target);
```

The implementation of these functions is in `security/umbrella/umb_types.c` and some examples are given here.

The function for looking up an restriction traverses the linked lists to find a path that matches the one given in the second argument. It walks through the input path one directory at a time, and calls itself recursively if a match is found. If at some point a FSR node has no more successors, this means that all subdirectories of the current is restricted, and a `DENIED` is returned.

```
1  int umb_fsr_check(struct fsr *root, char **path) {
2      struct fsr *fsr = root;
3      int retval = -1;
4
5      if (*path == NULL)
6          return ALLOWED;
7
8      if (fsr->name == NULL && fsr->succ == NULL)
9          fsr = fsr->next;
10
11     if (!fsr)
12         return ALLOWED;
13
14     if (retval == -1 && !strcmp(*path, fsr->name)) {
15
16         if (fsr->succ == NULL)
17             retval = DENIED;
18         else {
19             if (++path != NULL)
20                 retval = umb_fsr_check(fsr->succ, path);
21             else
22                 retval = ALLOWED;
23         }
24
25     } else if (retval == -1 && strcmp(*path, fsr->name)) {
26
27         if (fsr->next != NULL)
28             retval = umb_fsr_check(fsr->next, path);
29         else
30             retval = ALLOWED;
31     }
32     return retval;
33 }
```

The variable `retval` is used to return a decision value back through the stack of recursive calls, and whenever the value is different from -1, a decision has been made, and the function falls through to the return. This must ensure that nothing further is done once a decision is made.

When a new process is created, this process receives a copy of its parents FSR tree, along with any additional restrictions specified by the parent process. The function `umb_fsr_copy` copies the parents tree. If the child created does not have more restrictions than the parent, there is no need to copy the restrictions, but making the inheritance by means of the *copy on write* principle [51], i.e. simply making the child having a reference to the parent's FSR tree. By this scheme many memory allocations can be avoided, which will speed up inheritance dramatically. The copy on write principle is not yet implemented.

The function `umb_fsr_insert` is used to assign any additional restrictions from an array of strings. The implementation of this function and the rest of the functions in the FSR interface can be found in `security/umbrella/umb_types.c`.

4.1.3 LSM Hook Implementations

The LSM hooks used by 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 the kernel header file `include/linux/security.h` and in Appendix D. Appendix C describes how the hooks are placed in the kernel.

File Hooks

The hooks implemented for protecting the file system include `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 that makes a lookup in the processes FSR tree is done in line 13.

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`. Once this `dentry` is found it is given to the `file_hook_wrapper`.

```
1  static inline int file_hook_wrapper(struct dentry *dentry) {
2      int ss_decision, i = 0;
3      char *path[MAX_PATH_DEPTH];
4      struct security_struct *cur_security = current->security;
5
6      if (parse_path(dentry, path) == -EOVERFLOW)
7          ss_decision = -EOVERFLOW;
8
9      else {
10         ss_decision = umb_fsr_check(cur_security->fsr, path);
11         if (ss_decision)
12             ss_decision = -EPERM;
13     }
14     return ss_decision;
15 }
```

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 restrictions are stored in the bit-vector, making a security lookup simply consists of testing one bit in NFSR. The implementation of the `socket_create` is shown below.

```
1  static int umb_socket_create(int family, int type, int protocol) {
2      int decision = 0;
3      struct security_struct *security = current->security;
4
5      switch (family) {
6      case 2:
7          decision = bv_testbit(security->level1, IPNET);
```

```
8           break;
9
10      case 23:
11          decision = bv_testbit(security->level1, IRDA);
12          break;
13
14      case 31:
15          decision = bv_testbit(security->level1, BLUETOOTH);
16          break;
17      }
18
19      if (decision)
20          return -EPERM;
21      else
22          return ALLOWED;
23  }
```

The very fast lookups for sockets are important, since socket are used all the time for communication between processes.

Process Hooks

Controlling the creation of new processes is performed by `task_create`. This hook implements the NFSR on creation of new processes, and like the hook for controlling network, it simply checks this bit in the bit-vector.

```
1   static int umb_task_create(unsigned long clone_flags) {
2       struct security_struct *security = current->security;
3
4       if (current->security == NULL)
5           return ALLOWED;
6
7       if (bv_testbit(security->nfsr, FORK))
8           return -EPERM;
9       else
10          return ALLOWED;
11  }
```

4.1.4 Umbrella System Call

To set restrictions, information must be propagated from user space to kernel space, and this can be done via the `sys_umb_set_child_restrictions` system call shown below. This system call is wrapped in the Umbrella library, and can from user space programs be called as the restricted fork `rfork`. This system call can be called prior to forking a new child process, to specify some additional restrictions the new process should have, besides those inherited from the parent process.

The parameter list of the system call is an array of integers, used for NFSR, and an array of char pointers for the FSR. The check in line 13 ensures that the handling of file system restrictions is skipped if none exists.

In line 10 `bv_bit_on` is called to set the NFSR by taking a bit-vector and an index as arguments. Lines 15-30 handles the FSR. The FSR has to be terminated by a `NULL`, else it is impossible to find out how many restrictions are given as argument. The *kernel* does not crash by making a system call without

the terminating NULL. However, the calling user space program does, as the memory traversed is located in user space, which causes a segmentation fault.

```
asmlinkage void sys_umb_set_child_restrictions(int nfsr[],
                                               char **fsr) {
    int i = 0;
    struct security_struct *security = current->security;

    /* -10 is termination symbol of nfsr */
    for (i=0; i<NUMBER_OF_NFSR; i++) {
        if (nfsr[i] == -10)
            break;
        bv_bit_on(security->child_nfsr, nfsr[i]);
    }

    if((fsr != NULL) && (*fsr != NULL)) {
        bv_bit_on(security->child_nfsr, 0);
        for (i=0; i<MAX_L2; i++) {
            if (fsr[i] == NULL) break;
            security->child_fsr[i] =
                (char *)kmalloc(sizeof(char) *
                                (strlen(fsr[i]) + 1), GFP_ATOMIC);

            /* handle not enough memory situation */
            if (!security->child_fsr[i]) {
                while (i>0) {
                    kfree(security->child_fsr[i]);
                    i--;
                }
                return;
            }
            strcpy(security->child_fsr[i], fsr[i]);
        }
    }
    return;
}
```

Once the system call has set the appropriate fields the new process can be created. During creation of the process the task_alloc_security hook is called. Umbrella uses this hook to transfer restrictions to the new process.

```
static int umb_task_alloc_security(struct task_struct *p) {
    ...
    /* nfsr */
    bv_or((bitvector)security->nfsr,
          (bitvector)current_security->child_nfsr,
          (bitvector)current_security->nfsr);
    bv_reset(current_security->child_nfsr);

    /* fsr inheritance */
    fsr = umb_fsr_copy(current_security->fsr);

    /* additional fsr  */
    tmp_fsr = current_security->child_fsr;
    i = 0;

    while (tmp_fsr[i] != NULL && i < MAX_FSR) {
        chop_string(tmp, tmp_fsr[i]);
        umb_fsr_insert(fsr, tmp)

        kfree(tmp_fsr[i]);
        tmp_fsr[i] = NULL;
        i++;
    }
    ...
}
```

Lines 4-6 is the call to `bv_or` to set the new process' non-file system restrictions. Line 5 fetches the restrictions previously set using the system call. Line 10 copies the parents file system restrictions to the child and the rest of the functions inserts possible additional file system restrictions. To prevent siblings being assigned the same restrictions the `child_nfsr` is reset in line 7 and the `tmp_fsr` is freed in line 20-21.

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. Another solution has been implemented as well, namely a proc file system interface [43].

The proc file system interface makes it possible to specify child restrictions by writing to the file `/proc/umbrella`. The interface has been implemented thanks to Magnus Therning, Philips Research, Eindhoven.

4.2 Digitally Signed Binaries

The implementation of the digitally signed binaries, is the second major task in the implementation of Umbrella. To simplify the implementation this task have been divided into three major areas; storage of security information, security enforcement and issues regarding implementation of public key cryptography in the kernel.

4.2.1 Security Information Storage

As described in Section 3.3 the Umbrella signature is divided into several parts. In the actual implementation the ID and execute restrictions are appended to the binary file and a digital signature of this is created using BSign. This digital signature is then appended after the ID and execute restrictions. To automate this process a Ruby script has been developed and resides in the Umbrella library; when installed located in `/usr/bin/sign_file.rb`. Below is an example of how the script is used.

```
1 sign_file.rb --id=UmbrellaInc --nfsr=IPNET:BLUETOOTH
2              --fsr=/etc:/tmp --file=<Program to be bsigned>
```

The Ruby script `sign_files.rb` is run with four parameters. The `id` is the ID of the vendor signing the file. Secondly, the `nfsr` is a colon separated list of NFSR. The third parameter `fsr` is a list of colon separated FSR paths. Finally the `file` specifies the actual file to sign.

In the example above the program is assigned the NFSR restrictions `IPNET` and `BLUETOOTH` and FSR on the directories `/tmp` and `/etc`. The script then calls BSign to create a signed hash of the binary data and the restrictions, as shown in Figure 3.8 on page 39.

4.2.2 Security Enforcement

Umbrella must mediate execution of binaries in order to perform two security related tasks; integrity checking of the executable and the transfer of ex-

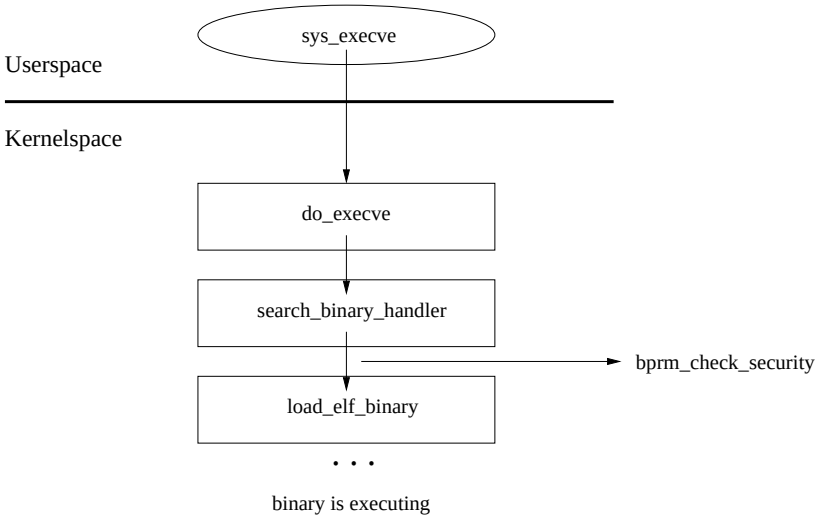


Figure 4.1: The control flow of a binary execution on an Umbrella enabled system.

ecute restrictions from the executable to the new process. The LSM hook `bprm_check_security` is called whenever a file is executed and is therefore used for this purpose. More precisely this hook is called whenever the `execve` system call is searching for a binary handler. If Umbrella's security checks succeeds, the `load_elf_binary` function is called to load the binary into memory, see Figure 4.1.

The hook `bprm_check_security` calls `umb_handle_signature` to handle authentication and execute restrictions.

```
1  int umb_bprm_check_security (struct linux_binprm * bprm) {
2      return umb_handle_signature(bprm);
3  }
```

The function `umb_handle_signature` is the entry point for the signature part of Umbrella and is called from the `bprm_check_security hook`. All other functions in this part of Umbrella, will only be evoked as a result of statements in this function. Because the function's size the following description will be divided into several parts.

```
1
2  int umb_handle_signature(struct linux_binprm * bprm) {
3      ...
4      struct file *file = NULL;
5
6      file = bprm->file;
7
8      if ( file == NULL )
9          return -EPERM;
10
11     elf_hdr = umb_read_elf_header(file);
12     if (elf_hdr != NULL) {
13         size = elf_hdr->e_shnum * sizeof(Elf32_Shdr);
14         elf_shdata = umb_read_section_header(file, size,
15                                              elf_hdr->e_shoff);
16         if (elf_shdata == NULL) {
17             printk("Umbrella: Unable to find elf_shdata\n ");
```

```
            kfree(elf_hdr);
            return -EPERM;
        }
    ...
```

In the first part of the function the ELF data is read into memory. The function `umb_read_elf_header` returns an `elfhdr` struct containing the actual ELF header, that is placed in the variable `elf_hdr`. Then the function `umb_read_section_header` is called to transfer all entries in the section header table of the ELF to the variable `elf_shdata`. If `umb_read_section_header` fails, the ELF is invalid and `-EPERM` is returned to deny execution.

Next part of the function extracts the signature from the binary and stores the BSign and Umbrella part separately in the struct `dsig` below.

```
struct dsig {
    char *bsig;
    char *usig;
};
```

```
...
/* Let's find the signature section */
file_sig = umb_find_signature(elf_hdr, elf_shdata,
                              file, &signature_offset);
if (file_sig == NULL) {
    printk("Umbrella: Signature not found for the binary: %s !\n",
            bprm->filename);
    ...
    return -EPERM;
}
...
```

The function `umb_find_signature` is called to extract the signature. The function takes four parameters; `elf_hdr` is an ELF header, `elf_shdata` holds all entries in the section header table of the ELF, `file` is the file handler of the binary and `sh_offset` is the offset of the signature section in the ELF binary. The return type is a `dsig` struct, which holds the BSign and Umbrella signature.

```
...
retval = umb_check_signature(elf_shdata, file_sig,
                             file, signature_offset);

if ( retval == 0 ) {
    parsed_usig = umb_parse_usig(bprm,file_sig->usig);
    if (parsed_usig == NULL) {
        printk("Umbrella: Error parsing the Umbrella Signature\n");
       return -EPERM;
    }
    ...
```

The function `umb_check_signature` checks the BSign signature using the appropriate public key. This part of Umbrella is elaborated in Section 4.2.3. If the verification of the BSign signature is successful, the Umbrella specific signature parts (ID and execute restrictions) are extracted using the function `umb_parse_signature`. The function returns a `parsed_sig` struct containing the Umbrella signature. If the function returns `NULL`, `-EPERM` is returned because that there was a problem parsing the signature.

```
...
if (security != NULL) {

```

```
4       bv_or((wordptr)security->child_nfsr,
5       (wordptr)security->child_nfsr,
6       (wordptr)parsed_usig->nfsr);
7
8       /* Setting child fsr */
9       fsr = parsed_usig->fsr;
10      if(fsr && *fsr != NULL) {
11          bv_bit_on(security->child_nfsr, 0);
12          i = 0;
13          while ((fsr[i] != NULL) && (i < UMBRELLA_SIG_NO_FSR)) {
14              tmp = (char *)kmalloc(sizeof(char) *
15                                    (strlen(fsr[i]) + 1), GFP_ATOMIC)
                                        ;
16              strcpy(tmp, fsr[i]);
17              security->child_fsr[i] = tmp;
18              i++;
19          }
20      }
21  ...
22  return retval;
```

If the signature was parsed successfully and the current process has a security field the execute restrictions found in the signature are transferred to the current processes' child `nfsr` and `fsr` fields. Currently the implementation does not include the caching of restrictions in the LSM security fields on inode structures.

4.2.3 Cryptography

No public key algorithms currently exist in the Linux kernel. To be able to verify the DSBs, e.g. the RSA algorithm is needed. This can be obtained by porting a small part of the GNU Privacy Guard (GPG) to the kernel. Recently a thread on the Linux kernel mailinglist[1] has discussed introducing a RSA library natively in the kernel.

The Digsig project already ported the RSA part of GPG, for verifying digital signatures, to kernel space [10]. Digsig only ported the multi precision integer (MPI) library and the RSA source code to kernel space. This reduced the amount of code imported into the kernel to about $\frac{1}{10}$ of the original size. Because of the availability of the of the ported GPG code, it was decided to use this for Umbrella.

Umbrella uses a SHA1 hash to secure the data in DSBs. The SHA1 algorithm is implemented in the Linux 2.6 kernel, and is ready for use. The Digsig has implemented three wrapper functions for the SHA1 functions available in the kernel. These can be used directly in Umbrella.

```
1   static int  digsig\_sha1\_init   (SIGCTX * ctx);
2   static void digsig\_sha1\_update (SIGCTX * ctx, char *buf,
3                                     int buflen);
4   static int  digsig\_sha1\_final  (SIGCTX * ctx, char *digest);
```

The first `digsig_sha1_init` allocates the necessary data structures to hold the SHA1 hash values. The functions `digsig_sha1_update` hashes the data and the function `digsig_sha1_final` finalizes the hashing. The SHA1 algorithm is defined in `include/linux/crypto.h`.

[1] www.lkml.org/lkml/2004/6/14/209

4.2.4 Authentication

In this section, the authentication of signatures is elaborated. As with the porting of the GPG library, the current implementation is based on the Digsig module, as it offers the features needed. The checking of binaries serves three main purposes; ensuring the origin of the binary, mediating execution of binaries and ensuring that the execute restrictions has not been tampered with. The authentication step consist of two parts; importing the public key into the kernel and checking the actual signature.

The Entry Point

The authentication of signatures is invoked from a call to `umb_check_signature` from `umb_handle_signature`. This function is located in the source code file `security/umbrella/umb_signature.c`.

```
1  int umb_check_signature(Elf32_Shdr *elf_shdata, struct dsig *sig,
2                          struct file *file, int sh_offset) {
3      int retval = 0;
4      if (key_loaded == 0) {
5
6          pkey_n = (char*)kmalloc(DIGSIG_MPI_MAX_SIZE_N, GFP_ATOMIC);
7          ...
8
9          pkey_e = (char*)kmalloc(DIGSIG_MPI_MAX_SIZE_E, GFP_ATOMIC);
10         ...
11
12         /*alloc ok, now lets get the key */
13         retval = umb_get_pkey(pkey_n, pkey_e, pfile);
14         ...
15
16         key_loaded = 1;
17     }
18
19     retval = umb_verify_signature(elf_shdata, sig->bsig,
20                                   file, sh_offset);
21
22     return retval;
23 }
```

The first time this function is called memory is allocated for the modulus and the exponent in line 6 and 9. These are then loaded using the function `umb_get_pkey`. The keys are stored in the variables `pkey_n` and `pkey_e` and then moved to the key variable `umbrella_public_key`. If the key was loaded successfully the `booted` variable is set to 1 to indicate that the key is loaded. When the key is loaded the actual authentication is initiated using the function `umb_verify_signature` and the return value is returned to the function `umb_handle_signature`.

Obtaining the Public Key

The loading of public keys in Umbrella is handled by the function `umb_get_pkey`. The function stores the modulus and the exponent of the key in the pointer arguments `raw_public_key_n` and `raw_public_key_e`. The last argument to the function is the file containing the public key.

```
1  int umb_get_pkey(unsigned char *raw_public_key_n,
2                   unsigned char *raw_public_key_e,
3                   char *pkey_file) {
4      ...
5
6      fd = filp_open(pkey_file, O_RDONLY, 0400);
7      ...
8
9      kernel_read(fd, offset++, &c, 1);
10     raw_public_key_n[0] = c;
11
12     len = c << 8;
13
14     kernel_read(fd, offset++, &c, 1);
15     raw_public_key_n[1] = c;
16
17     len |= c;
18     len = (len + 7) / 8;
19
20     if (len > DSI_ELF_SIG_SIZE) {
21         ...
22         return -1;
23     }
24
25     for (i = 0; i < len; i++) {
26         kernel_read(fd, offset++, &c, 1);
27         raw_public_key_n[i + 2] = c;
28     }
29 ...
```

The first part of the function is used to obtain the file struct associated with the key file, using the kernel function `flip_open`. After checking if the file struct was found and the necessary number of file operations exist, the length of the modulus part of the key is calculated and stored in the `len` variable. This length is calculated from the first two bits in the key. If this length is longer than 512 bytes, as defined in `DSI_ELF_SIG_SIZE`, the function returns `-1` meaning the key is corrupted. After the key has been checked, the key is read from disk one character at the time using `kernel_read`, in the loop in line 25, and stored in `raw_public_key_n`.

After calculating the modulus part of the key, the procedure is repeated to calculate the exponent part of the key.

Checking the Signature

After the key has been imported into the system, the check of the signature is started, by using the function `umb_verify_signature`. The function is called with three parameters; `elf_shdata` is the data which where signed, `sig->bsig` is the original signature of the binary and finally `sh_offset` which is the offset of signature section in the ELF header.

```
1  int umb_check_signature(Elf32_Shdr *elf_shdata, struct dsig *sig,
2                          struct file *file, int sh_offset) {
3      ...
4      retval = umb_verify_signature(elf_shdata,sig->bsig,file,
           sh_offset);
5      ...
```

The `umb_verify_signature` function verifies if BSign signature matches binary's signature.

```
1  static int umb_verify_signature(Elf32_Shdr *elf_shdata,
2                                  char *sig_orig,
3                                  struct file *file, int sh_offset) {
4      ...
5
6      ctx = umb_sign_verify_init(HASH_SHA1, SIGN_RSA);
7      ...
8
9      /* continue verification loop */
10     retval = digsig_sign_verify_update(ctx, read_blocks, retval);
11     ...
12
13     retval = digsig_sign_verify_final(ctx, sig_result,
14                                       DIGSIG_ELF_SIG_SIZE,
15                                       sig_orig + DIGSIG_BSIGN_INFOS
                                             );
16     ...
17     return retval;
18 }
```

Although this function is not explained in detail, the important parts is the calls to the three functions handling authentication. The function `umb_sign_verify`, checks if the selected hashing and cryptographic algorithm are supported and allocates space for the SHA1 data structures. The `digsig_sign_verify_update` calls `digsig_sha1_update` to create a new hash of the data. When the new hash is created, it is passed to the function `digsig_sign_verify_final` that performs the actual authentication.

4.3 Conclusion

The process based MAC has been successfully implemented. The data structure is undergoing changes from hash tables to the FSR trees. This transformation has not completed yet, but is expected to be working soon.

The implementation of the DSB part of Umbrella is progressing and is showing good promise of giving the desired result. At time of writing most of the code needed to perform verification of the BSign signatures is ported from the Digsig kernel module. However, some final testing and bug fixing still remain, before the implementation is completed.

Pending work is the implementation of the keyring, to which data structures are available from Linux 2.6.10-rc3. Furthermore, the optimization of using the LSM fields as cache for security information, is work to be done.

5 Umbrella in Practice

This chapter presents three practical issues regarding Umbrella, namely benchmarking, application programming interface and ideas to circumvent Umbrella. The benchmarking is preliminary since Umbrella is not yet fully implemented.

5.1 Umbrella Benchmarks

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 a machine with the following specifications.

- Intel Pentium 4 1.8 GHz
- CPU cache 512 KB
- 512 MB RAM
- Red Hat Linux 9
- Linux-2.6.9

It is important to note that the Umbrella that this benchmark is performed on is *not* coherent with the Umbrella described in Chapters 3 and 4. The difference is mainly in the implementation of the file system restrictions. In the benchmarked Umbrella these are implemented using hash tables. It was not possible to complete a benchmark of the new system to include in this report, but the comparison is interesting so the new benchmarks will be done as soon as the FSR is believed to be stable.

5.1.1 Benchmark Details

Three different benchmarks has been performed, each ran five times, on a system running an Umbrella patched Linux and one that did not.

1. Process creation – 40k processes
2. File system access – unpacking Linux-2.6.9

3. Interactive simulation – compilation of Linux-2.6.9

The process creation benchmark (1) will stress the functions involved in assigning security information, namely allocation of memory for security structure and inheritance of restrictions. The benchmark is done by timing the execution of a script that creates 40.000 processes. This is done for different sets of restrictions. The results can be found in Table 5.1 and Figure 5.1.

The file system access benchmark (2) is aimed at the performance of the functions involved in checking file system restrictions. This benchmark will also be performed for different settings of restrictions. The benchmark is performed by unpacking a Linux kernel tree, which will create approximately 19.000 files. The results can be found in Table 5.2 and Figure 5.3.

The interactive simulation benchmark (3) is a combination of benchmark 1 and 2. This benchmark will compile the Linux 2.6.9 kernel, which will create a large number of processes and access a large part of the 19.000 files in the kernel tree. This benchmark simulates interactive behavior, where files, process and I/O wait is involved. The results can be found in Table 5.3 and Figure 5.3.

5.1.2 Results

# of Restrictions	Clean	Umbrella	Overhead
None	15.5s	16.5s	6.7%
10 NFSR	15.5s	16.5s	6.5%
10 NFSR and 5 FSR	15.5s	16.9s	9.3%
10 NFSR and 50 FSR	15.5s	18.8s	21.4%

Table 5.1: Results from the process creation benchmark (1).

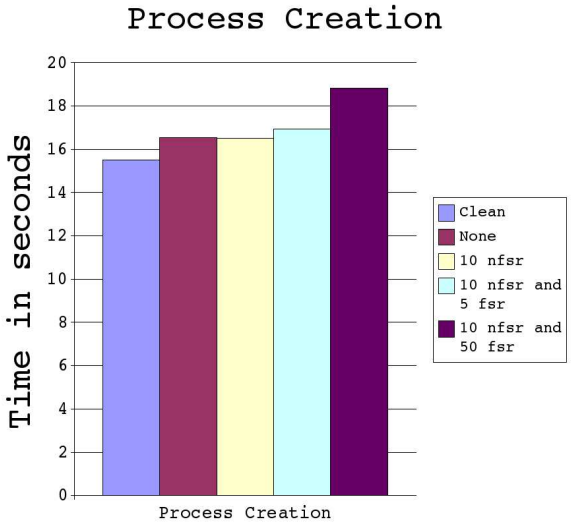


Figure 5.1: Results of the process creation benchmark (1).

# of Restrictions	Clean	Umbrella	Overhead
None	45.7s	49.9s	9.2%
10 NFSR	45.7s	47.9s	4.8%
10 NFSR and 5 FSR	45.7s	50.9s	11.4%
10 NFSR and 50 FSR	45.7s	52.9s	15.7%

Table 5.2: Results from the file system access benchmark (2).

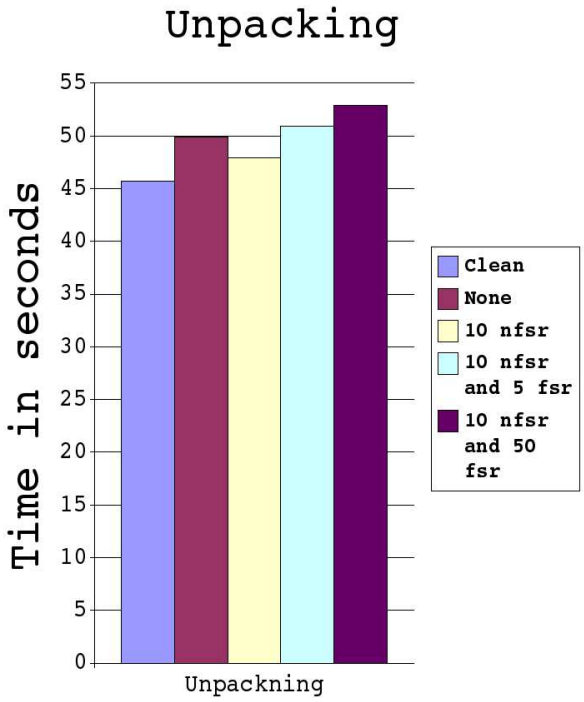


Figure 5.2: Results of the file system access benchmark (2).

# of Restrictions	Clean	Umbrella	Overhead
None	362.4s	365.6s	0.9%
10 NFSR	362.4s	365.8s	1.0%
10 NFSR and 5 FSR	362.4s	368.4s	1.7%
10 NFSR and 50 FSR	362.4s	369.2s	1.9%

Table 5.3: Results of the interactive simulation benchmark (3).

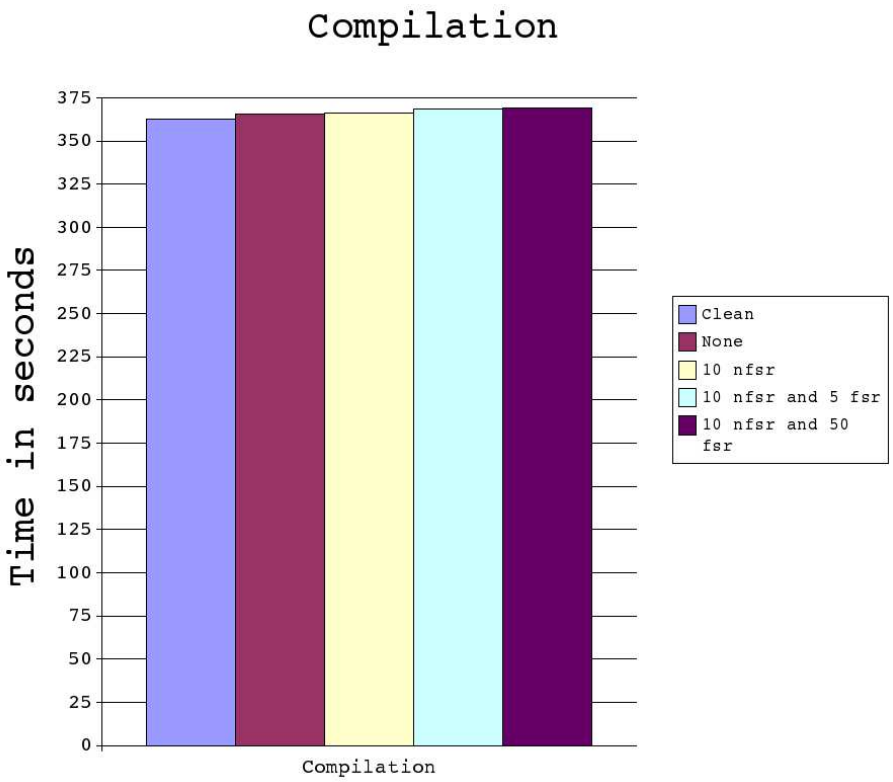


Figure 5.3: Results of the interactive simulation benchmark (3).

5.1.3 Discussion

Umbrella is not yet fully implemented, which makes the above results estimates of benchmarks on the final system. The majority of the planned functionality is implemented, however, no optimizations are performed yet. However, the results are useful for estimating performance of the final system. The results indicate that Umbrella will not suffer from major performance issues. The particular benchmarks are aimed at the process based MAC only, and do not reflect the new design of FSR from Chapter 3. The benchmarks for the DSBs are pending work when this feature is believed to be stable.

Instead of going through the results from one end to the other, the most interesting results will be discussed in the following.

In the process creation benchmark the performance of the inheritance of restrictions is put to the test. Creating such a large number of processes that does nothing, stresses the algorithms for inheritance and inserting restrictions. The results in Table 5.1 and Figure 5.1 clearly shows the overhead imposed by these algorithms. As the number of restrictions is enlarged the overhead rises to more than 20 %. This overhead is rather large, but with normal system use, this amount of processes will never be created without any intermediate computation or I/O-wait. Furthermore, the implementation of the copy on write principle for inheritance will reduce the overhead significantly.

The overhead introduced by the access control functions is investigated in the second benchmark. As Table 5.2 and Figure 5.3 shows the overhead rises with the number of restrictions, except for one result, the benchmark of a Umbrella system, where no restrictions are set. This result is surprising, since it takes longer time, than the benchmark with 10 NFSR. The expected result was, that the no restrictions benchmark would be slightly faster since no lookups are made. All the benchmarks where done five times and an average was calculated to avoid uncertainties, so this is not the explanation.

The rest of the file system access benchmarks shows that more restrictions yields more overhead. It was assumed that increasing the number of file system restrictions would not yield more overhead since lookups in a hash table takes constant time. The conclusion is that collisions in the hash table cause this overhead. The design and implementation reflected in this documentation uses less string manipulation than the implementation benchmarked, which is based on hash tables. This makes us confident that the FSR solution will bring down the overhead of file system access control.

In the final benchmark the overhead of Umbrella is minimal. The main reason for this, is that once the compiler is working, no files are accessed and no further processes are forked. We believe that this benchmark is a good approximation to ordinary use of the system, since both process creation, file system access and a plain computation is performed.

Regarding all the benchmarks it will be very interesting to see what impact the new implementation of file system restrictions will have. As mentioned we believe that it will increase performance on file system access control. As soon as the implementation is completed, the benchmarks will be updated.

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 re-

striction may be the configuration files of the system together with the personal data of the user.

5.2.1 Getting Started

To set up a system for experimenting with Umbrella the following steps must be performed.

1. Linux kernel minimum 2.6.6 and either of these architectures: i386, ARM or User-Mode Linux.
2. The newest stable Umbrella, which can be downloaded from: *www.sourceforge.net/projects/umbrella*
3. The Umbrella kernel patch in the tarball is applied by changing directory to the vanilla kernel source and doing a `bzcat umbrella.patch.bz2 | patch -p1`
4. The kernel should then be configured as usual, with the additional options `CONFIG_SECURITY=y`, `CONFIG_SECURITY_NETWORK=y` and `CONFIG_SECURITY_UMBRELLA=y`. Support for setting restrictions through the `proc` filesystem can be obtained by setting `CONFIG_SECURITY_UMBRELLA_PROCFS=y`.
5. Compile and boot the kernel
6. Install `libumbrella`
7. Compile your programs for Umbrella, sign your programs for Umbrella ... in short: *Hack away!*

5.2.2 Setting Restrictions

Setting restrictions involves setting the static restrictions (NFSR), as defined in Table 3.1 on page 33, and suitable restrictions for the file system (FSR).

The restrictions are introduced by the restricted fork (`rfork`), which acts like a normal `fork` but takes two additional arguments, namely an integer array of the static restrictions and an array of char pointers to filesystem restrictions.

```
1 rfork(int[] nfsr, char *fsr[]);
```

The `rfork` is a wrapper for the system call `umb_set_child_restrictions`, which sets restrictions for the *next* child this process created and then `rfork` calls the standard `fork`.

When the child of the process is executed, the child restrictions are removed from the parent, and thus if this process afterward forks a new process, the resulting process will only have the restrictions inherited from the parent.

The restrictions for the next child are passed to the kernel by invoking the system call, which takes an integer array of NFSR and an array of char pointers of FSR.

5.2.3 The Umbrella Library

The Umbrella library `libumbrella` is a small library that implements the `rfork` wrapper for the system call. To compile the library the `C_INCLUDE_PATH` in the shell environment must be set to the include directory of the Linux kernel source with the Umbrella patch applied.

```
1 export C_INCLUDE_PATH=/usr/src/linux-2.6.9-umbrella/include
```

The library is then compiled and installed by typing `make` and `make install`. The installation also installes script for signing files to `/usr/bin/sign_file.rb`. Uninstalling is invoked by a `make uninstall`.

The Umbrella library can be obtained either from the stable tarball on the Umbrella website or fetched from the development CVS.

5.2.4 Example Program

The following code snippet illustrates use of the restricted fork.

We need to import `umbrella.h` in line 3 to be able to call `libumbrella`. We define the non filesystem restrictions in line 8 and the file system restrictions in line 9. Now instead of making a normal `fork` in the `switch` statement at line 11 we call `rfork`.

The child process created by the restricted fork is now restricted as specified. When it breaks out of the `switch` structure and execute a shell, these restrictions are inherited to the shell executed in line 23.

In this example the parent process simply dies in line 20, but the child lives on to execute `xterm` in line 23.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<umbrella.h>
4
5  int main() {
6
7      int pid;
8      int  nfsr[] = {IPNET, BLUETOOTH};
9      char *fsr[] = {"/boot", "/foo", NULL};
10
11     switch (pid = rfork(nfsr, fsr)) {
12         case 0: /* child */
13             printf("child pid: %i\n", getpid());
14             break;
15         case -1:
16             printf("rfork ERROR\n");
17             return -1;
18         default: /* parent */
19             printf("parent pid: %i\n", getpid());
20             exit (0);
21     }
22
23     system("/usr/bin/xterm");
24
25     printf("Restricted child died\n");
26
27     return 0;
28 }
```

When this C program is executed, a shell 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.

```
~ # whoami
root
~ # ls /boot
ls /boot/
config-2.4.21-15.EL             initrd-2.4.21-20.ELsmp.img
vmlinux-2.4.21-15               kernel-2.6.9
...
~ # ./umbrella_restricted_sh
~ # whoami
root
~ # ls /boot
ls: /boot/: Operation not permitted
~ # mkdir /foo/bar
mkdir: Cannot create directory ‘/foo/bar’: 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.

To prevent e.g. kernel tampering, researchers at IBM, which is member of the Trusted Computing Group[1] (TCG), has developed a system that creates a chain of trust in the boot process for Linux. The simple overview is that a chip (TCG standard) verifies the boot loader, which verifies the Linux kernel, which in turn verifies the init process, and so on. All executable content that is loaded onto the Linux system is measured before execution and these measurements are protected by the Trusted Platform Module that is part of the TCG standards [44].

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.

Library tampering can be avoided by signing shared libraries. These are like executable files in ELF format, and can be signed directly. However, execute restrictions does not apply to libraries. The current implementation which use the LSM hook `bprm_check_security` does not provide this feature, however the Digsig project uses the hook `file_mmap` hook to achieve this.

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

[1] www.trustedcomputinggroup.org

system. Beside restricted access to devices in `/dev`, restrictions on making new devices should be restricted. This restriction is not yet implemented, but the hook `inode_mknod` can be used for this purpose.

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. This is native supported in Linux and details are presented in [27, 23].

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* [12], 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 different attack scenarios, and the ability of Umbrella to limit the harm of these. First, a discussion on the harm of process hijacking is presented. Then a concrete example of how a suid-bit attack can be performed and contained by Umbrella. Then a buffer overflow example in Ghost View is examined. Finally the harm of a concrete kernel vulnerability in Linux process trace system call is discussed.

5.4.1 Process Hijacking

Hijacking a process is one way to break into a vulnerable system. Hijacking a privileged root process and making it spawn e.g. a shell, yields a shell with root privileges and thereby access to the entire system. Common ways of hijacking a process is described in Section 1.3 on page 11. The damage done by this type of attacks can be limited or completely eliminated using Umbrella.

Hijacking a process owned by a regular user will provide access to resources, as specified by DAC. Hijacking a user owned process, e.g. a browser or email client, 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.

5.4.2 Suid-Bit Attack

A program with the suid-bit set execution with privileges of the owner. Suid-bit programs owned by root, thus enable regular users to execute the program with privileges of root. If such a program can be successfully attacked, a attacker can elevate his privileges to that of root.

To demonstrate that suid-bit vulnerabilities are a real threat to system security, a couple of examples of the concept have been found on the Gentoo Security Advisories[2].

- GLSA 200409-18 – cdrtools: Local root vulnerability in cdrecord if suid-bit is set.
- GLSA 200409-11 – star: Suid root vulnerability

In both of the above examples, a local user can elevate his privileges to that of root, by exploiting bugs in suid-bit programs.

Umbrella Suid-Bit Demonstration Program

In the following a proof of concept suid-bit attack is performed on a Umbrella patched Linux. The example is based on the stable patch 0.5.2 of Umbrella. The code can be fetched from CVS in `umbrella-devel/demo_programs`.

Two example programs where developed. The first, `vulnerable_program.c` is a very simple program listed below.

```
1 #include<stdio.h>
2 #include<stdlib.h>
3
4 int main(int argc, char *argv[]) {
5     char buff[7];
6     strcpy(buff, argv[1]);
7     return 0;
8 }
```

The program contain a small static buffer with a size of 7 and the command line argument is copied directly into it, without checking the length. This creates the possibility of a buffer overflow, which can be used to override the return address of the program.

The second program, `signed_vulnerable.c` is similar to the first one, except that it has been modified to use Umbrella for protecting the system, when handling dangerous input.

```
1  int main(int argc, char *argv[]) {
2      int  nfsr[] = {};
3      char *fsr[] = {"/root", NULL};
4
5      umb_set_child_restrictions(fsr, nfsr);
6
7      char buff[7];
8      strcpy(buff, argv[1]);
9      return 0;
10 }
```

This example adds lines 2-5 to the above vulnerable program. These lines calls the Umbrella system call in order to restrict the next created process from `/root`. Copying the command line argument into the buffer creates the possibility of executing of arbitrary code. This arbitrary code will be restricted according to the restrictions set by the system call.

[2] www.gentoo.org/security/en/glsa

To compile the demo example, a special `exploit` section have been added to the `Makefile` of the demo package.

```
exploit: checkenv src/eggshell src/vulnerable_program \
         src/signed_vulnerable tobin strip

    #Signing the umbrella protected exploitable program
    sign_file.rb --id=UmbrellaInc \
                 --l1=IPNET --file=bin/signed_vulnerable

    #Set suid-bit to create the chance of getting a root shell
    chown root:root bin/vulnerable_program \
                    bin/signed_vulnerable
    chmod u+s       bin/vulnerable_program \
                    bin/signed_vulnerable
```

The `exploit` section is responsible for compiling three small programs. The first is `eggshell.c`, which places some shellcode in the environment of a new shell. This shellcode can then be used to attack the programs `vulnerable_program.c` and `signed_vulnerable.c` which are listed above. After compiling the file `signed_vulnerable`, it is signed using `sign_file.rb` and sets a single restriction on the network. Finally the owner of the files is set to root and the suid-bit is set. The binary files can be found in the `demo_programs/bin`.

```
$ ./eggshell
Using address: 0xbffff2c8
$ whoami
john
$ ls /root/
ls: /root/: Permission denied
$ ping localhost
PING localhost (127.0.0.1) 56(84) bytes of data.
64 bytes from (127.0.0.1): icmp_seq=1 ttl=64 time=0.056 ms
```

The first step in the exploit is to execute the program `eggshell` to prepare the attack. The program retuns an address containing the shellcode and launches a new shell owned by john to prevent the memory of the shell code to be deallocated. Because the new shell is owned by john, it is prohibited from accessing the areas reserved for the root account, but it is still able access the network.

```
# ./vulnerable_program `perl -e 'print "\xc8\xf2\xff\xbf"x132'`
# whoami
root
# ls /root/
dead.letter  testme
# cat /root/testme
Hello World
# echo "Inserting stuff" >> /root/testme
# cat /root/testme
Hello World
Inserting stuff
# ping localhost
PING localhost (127.0.0.1) 56(84) bytes of data.
64 bytes from (127.0.0.1): icmp_seq=1 ttl=64 time=0.058 ms
```

The next step is to run the vulnerable program with the address of the shellcode 132 times. Perl is used to print this. The return address is the overwritten by the new memory address of the shell code, see Figure 5.4 on the next page. When the program returns, it returns to the shellcode, which executes a new shell. The new shell spawned is a root shell, which gives unrestricted access

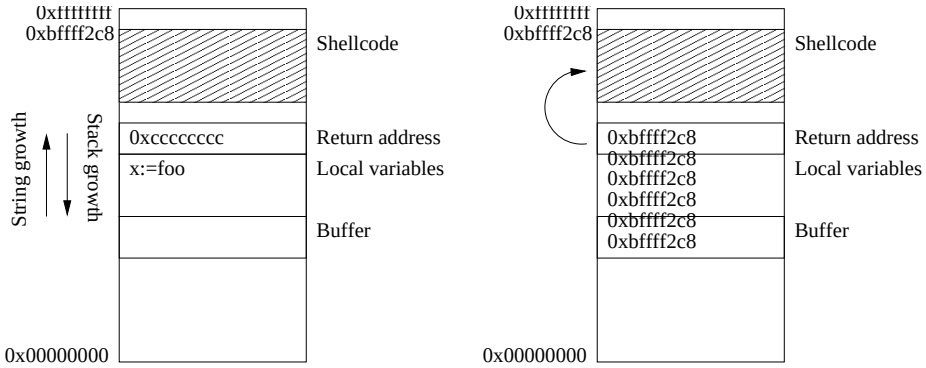


Figure 5.4: Buffer overflow example.

to the entire system. An example of this is the `/root/testme` file used in this example, that contain a single line of text, namely `Hello World`.

As shown in line 8 it is possible to modify the file, and because this process is owned by root, full access to all files in the system is possible. Also network access is possible as shown in line 12.

```
1  # ./eggshell
2  Using address: 0xbffff2c8
3  # whoami
4  john
5  # ls /root/
6  ls: /root/: Permission denied
7  # ping localhost
8  PING localhost (127.0.0.1) 56(84) bytes of data.
9  64 bytes from (127.0.0.1): icmp_seq=1 ttl=64 time=0.068 ms
10 ...
11 # ./signed_vulnerable ‘perl -e ’print "\xc8\xf2\xff\xbf"x132’‘
12 # ls /root/
13 ls: /root/: Operation not permitted
14 # ping localhost
15 socket: Operation not permitted
```

This final example shows how the attack demonstrated above is contained when the attacked program is signed using Umbrella. The operations in line 12 and 14 are now not permitted.

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 signing the Ghost View binary, it can be restricted and the harm of an

attack could be contained. 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.

This type of exploit is a real threat to CE devices. Recently exploits like this was found in TIFF library, which is included in e.g. the PDF library, and the image library `imlib`, which is used in many programs[3].

5.4.4 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 [14]. 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 [13].

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
5      switch (pid=fork()) {
6
7      case -1:
8          perror("Can't fork(): ");
9          break;
10
11     case 0:
12
13         child=getpid();
14         k_child=child+1;
```

[3]Exploits elaborated in GLSA 200412-02 and GLSA 200412-03.

```
15
16      ...
17
18          while ((error=ptrace(PTRACE_ATTACH,k_child,0,0)==-1) && (
                errno==ESRCH)) {
19              fprintf(stderr, ".");
20          }
21
22      ...
23
24          for (i=0; i<=SIZE; i+=4) {
25              if( ptrace(PTRACE_POKETEXT,k_child,regs.eip+i,*(int*)(
                    shellcode+i))) {}
26          }
27
28      ...
29
30          if (ptrace(PTRACE_DETACH,k_child,0,0)==-1) {
31              perror("-> Unable to detach from modprobe thread: ");
32          }
33
34      ...
35
36          if (kill(parent,9)==-1) {
37              perror("-> We survived??!!??  ");
38          }
39
40
41     default:
42
43      ...
44
45          socket(AF_SECURITY,SOCK_STREAM,1);
46          break;
47      }
48      exit(0);
49  }
```

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`[4] 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 [11].

5.5 Conclusion

Umbrella has been preliminary benchmarked and the results shows that Umbrella imposes an overhead, which hopefully can be minimized implementing the new file system restrictions design. The overhead of Umbrella is, however, not noticeable when interactively using a system running an Umbrella patched Linux.

Examples of programming against Umbrella is given and these examples demonstrate the tiny effort required by the developers to make a secure system.

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

A number of example vulnerabilities, like exploitation of suid-bit programs, in a Linux system are also presented to demonstrate how Umbrella in a very simple manner can be utilized to prevent serious damage of such attacks.

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* [58]. 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 [31].

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 [9]. 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.

```
1 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`.

```
1 partial order {
2     $$checked < $$unchecked
3 }
```

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.

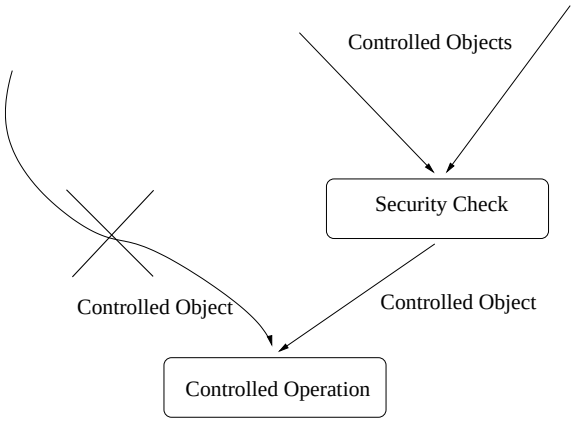


Figure 6.1: The complete mediation problem.

- *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 interprocedural 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 [58].

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 [58]. 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* [28]. 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.

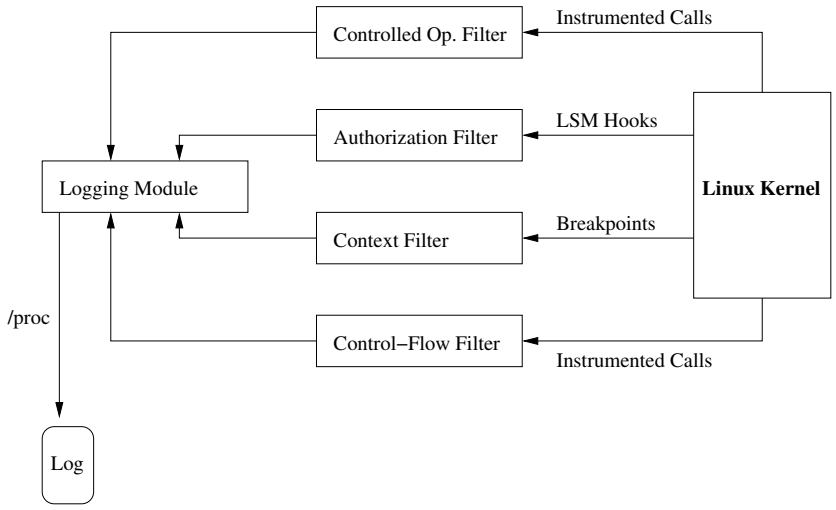


Figure 6.2: Architecture of tools for runtime verification of LSM.

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.

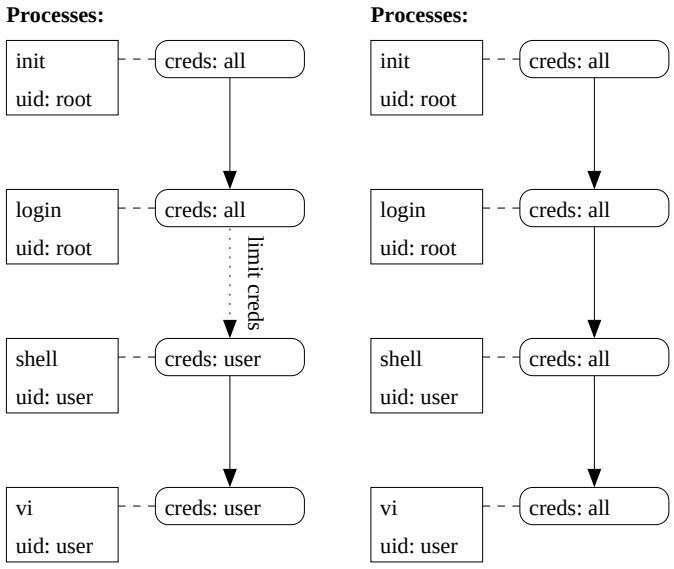
Bugs number 1-3 is fixed by adding authorizations and number 4 is solved by disallowing memory mapping of files that requires `read` authorization.

6.4 Capability Root Exploit

December 8th 2003, a root exploit of the Capability LSM module were reported to several Linux security mailing lists [40]. The exploit elevated privileges of all processes in the system, when the Capability module was loaded.

POSIX.1e Capability [37] 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.),



(a) Before (b) After

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`.

```
1 if (cap_raised (tsk->cap_effective, cap)))
2     return 0;
3 else
4     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 [58, 28] 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 is a completely new scheme for security for CE devices, which implements a combination of process based mandatory access control and authentication of binaries, where the MAC policy is embedded in the binary. The MAC is based on restrictions processes, where all processes are at least as restricted as their parent. The Umbrella scheme is developed and implemented for the Linux 2.6 kernel series.

Conclusions on the design, implementation and the development process of the project are presented in the following. Furthermore, optimizations and the future of the project are described.

Umbrella and Traditional MAC

Umbrella takes a different approach to mandatory access control, than traditional implementations. Umbrella focuses on individual processes rather than the widely used global subject/object model. This enables Umbrella to avoid the maintenance of an 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, requires a policy for *all* other objects and subjects. This is a very demanding task even though some MAC schemes support some degree of automatization of this. Umbrella completely eliminates maintenance of an access matrix by integrating security policy in signed binaries. Binaries are signed by trusted vendors, who are able to set restrictions for programs, both on time of execution and for sub-processes individually. Restrictions can limit access to e.g. personal data or system resources.

Design

The design of Umbrella is aimed at limiting the possible harm of malicious software on consumer electronic 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 Linux 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 parts, namely static non-file system restrictions, on signals, networking and process creation, and dynamic file system restrictions on paths in the file system. Non-file system restrictions are represented in a bit-vector, which provides good performance of both inheritance and lookups. The file system restrictions are stored in a tree structure, which matches the structure of the file system in Linux.

The combination of non-file system restrictions and file system restrictions allow Umbrella to perform well and be highly flexible at the same time.

One of the philosophies of Umbrella is that, *to make a secure system, the software developers must be involved*. Umbrella encourages programmers to restrict processes from accessing not needed parts of the file system, network and other system resources. We believe that transferring responsibilities, from the security administrator to the developers, is necessary to obtain secure computer systems. The developers can specify suitable restrictions for sub-processes of programs, much more fine grained and accurate than a security administrator who *cannot* be familiar with *every* aspect of *all* programs in the system.

Umbrella is designed to be transparent to the user. This is achieved by importing security policies through digitally signed binaries. The file signatures are handled by means of public key cryptography, where the public keys are stored on the device in a protected key ring. When a digitally signed binary is executed, it is authenticated and the resulting process is assigned the embedded restrictions.

Implementation

Umbrella has been developed as an open source project hosted on SourceForge.net. The Umbrella web site have had more than 43.000 visits since the public launch on February 3rd 2004 and since the first version was released more than 750 downloads have been performed.

The completed parts of the implementation covers restrictions on processes, which are implemented together with the intended functionality, i.e. inheritance, the restricted fork and setting restrictions from digitally signed binaries.

The base for the Umbrella implementation is the Linux Security Modules framework, which provides hooks for controlling access to data structures in the kernel. Since Umbrella is completely based on LSM the security provided, relies on this. The work of two articles on verification of LSM are presented. The results were a small number of errors in the LSM framework, which have been corrected. Verifying the Umbrella source code is a pending task.

The data structures for holding restrictions, i.e. bit-vector and the file system restriction trees, are implemented specifically for suiting the needs of Umbrella; high flexibility and good performance.

Implementation of the integration with GNU Privacy Guard to verify digitally signed binaries is almost completed. Finishing implementation of the redesigned file system restrictions is also near completion. The next task is the implemen-

tation of a kernel space keyring.

Development Process

The Umbrella development process has been divided into small steps, in the style of extreme programming with short release cycles and implementing functionality before considering optimizations.

One of the goals of the autumn semester 2004, was to complete the implementation of Umbrella, to show the principle of the combination of process based MAC and digitally signed binaries. Another goal was to make a proof of concept implementation for TDC's Linux-based alarm box.

Besides these practical activities, a great deal of effort has been invested in presenting and discussing Umbrella with contacts outside Aalborg University. That has lead to contacts with persons from Panasonic, Phillips and IBM and many others.

In October Panasonic invited us to participate in a meeting with the Security Working Group of Consumer Electronics Linux Forum in Princeton, New Jersey. Besides great feedback on Umbrella and expansion of our personal networks, the trip to Princeton has lead to a contract with Panasonic for the spring semester and an invitation to work in Tokyo in the summer 2005.

January 2005 will be spent on writing a paper on Umbrella and participating in a new meeting with Consumer Electronics Linux Forum, which is held in San Jose, California.

The goals for the spring semester are not completely determined. It could be interesting to perform formal verification of Umbrella and one of the courses next semester is *Test and Verification of Software*. Besides that, Panasonic indicated that they have several projects regarding security, that could fit our spring semester. Finally we would like to finish the implementation of Umbrella, to prove the strength of the combination of process based MAC and digitally signed binaries.

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 between 1% and 6% for non-file system only. Adding file system restrictions raise the overhead to between 2% and 20%. The benchmarked implementation of the file system restrictions lacks some optimization, which is believed to increase performance of this part.

Using a computer with Umbrella running it is, in our experience, not possible to feel and overhead.

Umbrella in Other Operating Systems

The design of Umbrella is aimed at consumer electronics and it is tested on a HP iPAQ, an embedded alarm box running Linux and several i386 systems.

Porting of Umbrella to other operating systems is 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.

It would be interesting to port Umbrella to other operating systems for CE devices like Symbian or 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.

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 the ARM architectures.

Final Remarks

The project has spanned for almost 15 months and during that period many thoughts and ideas have been discussed, tried, and discarded. The project team effort, together with great supervision and commitment from Emmanuel Fleury have resulted in a completely new security scheme for consumer electronics.

The combination of process based mandatory access control and digitally signed binaries has proven to be a very powerful concept for protecting Linux based CE devices against an increasing rain of attacks.

We can't prevent the rain, – but we don't get wet!

The Umbrella Team.

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 menuconfig 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:

```
1 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.

B Linux for Handhelds

There exist several projects and forums that supports and is working on the use of Linux on consumer electronic devices. The following is a number of these.

- 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.

- uCLinux – `http://uclinux.org`

  uClinux is a set of patches for Linux that supports MMUless processors. It brings a full featured operating system onto platforms that would otherwise run less advanced, simpler operating systems. uClinux gives the

programmer a Linux API with remarkably few concessions to the lack of MMU (Memory Management Unit), and in terms of code size and efficiency it has an advantage over standard Linux.

- Consumer Electronics Linux Forum (CELF) – `http://celinux.org`

  CELF is an industry group that is focused on the advancement of Linux as an open source platform for consumer electronic (CE) devices. The CELF intends to operate completely within the letter and the spirit of the open source community. The CELF is a place to come and discuss various issues that are of particular importance to the CE industry. Through an open process, the CELF members will clarify and codify certain requirements to be addressed in open source software. Thereafter, the CELF will evaluate any open source submissions as to their effectiveness and responsiveness to the requirements.

- Flash Linux – `http://sourceforge.net/projects/flashlinux`

  Flash Linux is a compact distribution designed to run off 256Mb USB keys. It includes hardware detection, auto configuration, a fairly complete Gnome 2.8 desktop, and associated office tools. Flash Linux is in the very first development cycle, and therefore still unstable.

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 [6, 56].

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

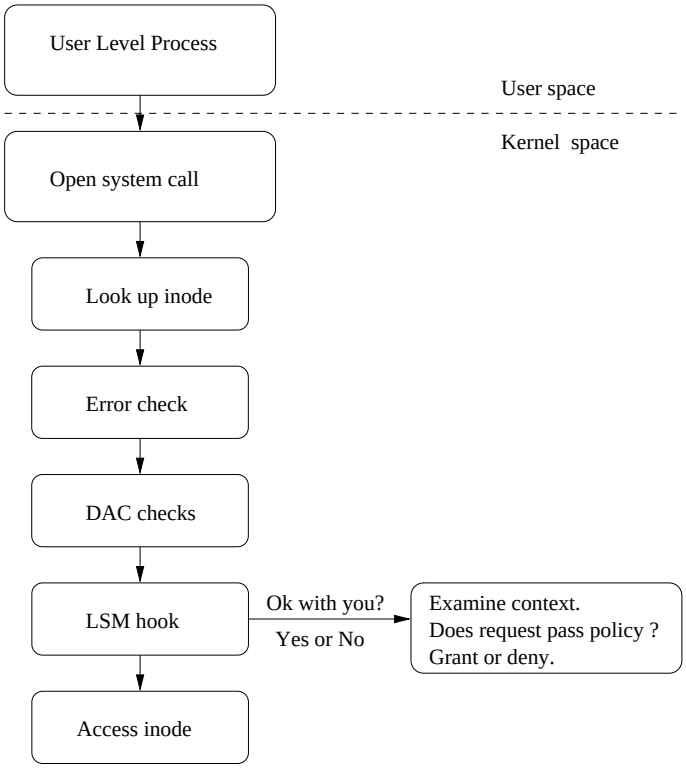


Figure C.1: LSM hook architecture.

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 [56]. 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`.

```
1  struct task_struct {
2      /* -1 unrunnable, 0 runnable, >0 stopped */
3      volatile long state;
4      struct thread_info *thread_info;
5      ...
6      // Security field added by LSM
7      void *security;
8      ...
9      siginfo_t *last_siginfo; /* For ptrace use.  */
10 };
```

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:

```
1 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.

```
1  int vfs_mkdir(struct inode *dir, struct dentry *dentry, int mode) {
2      int error = may_create(dir, dentry, NULL);
3
4      if (error) return error;
5
6      if (!dir->i_op || !dir->i_op->mkdir)
7          return -EPERM;
8
9      mode &= (S_IRWXUGO|S_ISVTX);
10     error = security_inode_mkdir(dir, dentry, mode); /* hook */
11     if (error) return error;
12
13     DQUOT_INIT(dir);
14     error = dir->i_op->mkdir(dir, dentry, mode);
15     if (!error) {
16         inode_dir_notify(dir, DN_CREATE);
17         security_inode_post_mkdir(dir,dentry, mode);  /* hook */
18     }
19     return error;
20 }
```

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 [56]. 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 [38]

The module is structured like a regular Linux kernel module. Information about this may be found in [43]. 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.

```
1 static int dirmonitor_inode_mkdir (struct inode *dir, struct dentry
      *dentry, int mode) {
2     printk(KERN_INFO "Directory created\n");
3     return 0;
4 }
```

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

```
1 static struct security_operations dirmonitor_security_ops = {
2     .inode_mkdir = dirmonitor_inode_mkdir,
3 };
```

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

This appendix provides information on all the security hooks in Linux 2.6. This information can be found updated for a specific kernel in the kernel source tree in `include/linux/security.h`. The @-names refers to parameter names of the hook functions.

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.

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.

Inode Operations

inode_alloc_security Allocate and attach a security structure to @inode → 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 → 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.

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.

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.

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.

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.

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 → inode → i_security. This hook may be used to update the sock → inode → 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 → sk_security field, which is used to copy security attributes between local stream sockets.

sk_free_security Deallocate security structure.

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.0.1 Individual Messages Held In System V IPC Message Queues

msg_msg_alloc_security Allocate and attach a security structure to the msg → 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.0.2 System V IPC Message Queues

msg_queue_alloc_security Allocate and attach a security structure to the msq → 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.0.3 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.0.4 System V Semaphores

sem_alloc_security Allocate and attach a security structure to the sma → 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.

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.

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. (**RELEASED 2004-09-23**) The forth release.
   - Update Umbrella for latest kernel.
   - Hash tables moved to separate files.
   - Support for procfs from Magnus Therning applied.
   - Static NFSR hash table implemented (reduced overhead by 50
5. (**RELEASED 2004-10-07**) Implement execute-restrictions on digitally signed binaries.
   - It is now possible to append execute restrictions to binaries.
   - The bit-vector library has been completely reimplemented to suit the specific needs of Umbrella.
6. (**WORK IN PROGRESS**) Implement integration with GNU Privacy Guard.
   - Ability to verify the signature, and thereby be sure the software is from a known vendor, that it has not been altered and the execute-restrictions have not been altered.
   - Implement the new file system restrictions (FSR) design, according to the design in Chapter 3.
7. (**PLANNED FOR SPRING 2005**) Implement keyring in the kernel.
   - Implementation of a kernel space keyring for holding public keys of various vendors.
   - User space tools to manage these keys.

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] An Overview of Cryptography.
http://www.garykessler.net/library/crypto.html♯intro, May 2003.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[20] A. Apvrille, D. Gordon, S. Hallyn, M. Pourzandi, and V. Roy. The DigSig Project. March 2004.

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

[22] Kent Beck. *Extreme Programming Explained*. Addison Wesley, 1999.

[23] David Braun. Disk Encryption HOWTO.
http://www.tldp.org/HOWTO/Disk-Encryption-HOWTO, November 2004.

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

[25] George Coulouri, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison Wesley, third edition, 2000.

[26] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor. SubDomain: Parsimonious Server Security. 14th USENIX Systems Administration Conference (LISA 2000), December 2000.

[27] Christophe Devine. Encrypted Root Filesystem HOWTO.
http://tldp.org/HOWTO/Encrypted-Root-Filesystem-HOWTO, October 2004.

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

[29] eWeek. Symbian says skulls may not be malware.
http://www.eweek.com/article2/0,1759,1731525,00.asp, November 2004.

[30] eWeek. This time, cell phone virus is for real.
http://www.eweek.com/article2/0,1759,1614794,00.asp, October 2004.

[31] 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.

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

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

[34] Simson Garfinkel and Gene Spafford. *Pratical UNIX and Internet Security*. O'Reilly, second edition, 1996.

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

[36] Immunix. The Immunix SubDomain Suite. http://immunix.org/pdfs/ImmunixSubDomainsSuite.pdf, September 2004.

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

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

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

[40] 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.

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

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

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

[44] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert Van Doorn. Design and implementation of a tcg-based integrity measurement architecture. Technical report, IBM Research, 2004.

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

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

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

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

[49] 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.

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

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

[52] Jeroen Ruigrok van der Werven. *BSD File Formats Manual*. FreeBSD Organization, July 1999. man 5 elf.

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

[54] John Viega and Matt Messier. *Secure Programming Cookbook for C and C++*. O'Reilly, 2003.

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

[56] 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.

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

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