

SEBPS v1.0: Designing a Secure and Maintainable End-User Platform

Semesterarbeit N. Perrenoud

August 9, 2010



Materialität

Titel	Semesterarbeit N. Perrenoud: SEBPS v1.0: Designing a Secure and Maintainable End-User Platform
Bestellnummer	16393
Lieferant	N. Perrenoud
Beginn	05.02.2010
Ende	01.04.2010
Student	Semesterarbeit N. Perrenoud
Projektleiter	Dr. Lukas Ruf, <Lukas.Ruf@consecom.com>, Dr. Arno Wagner, <Arno.Wagner@consecom.com>
Professor	Prof. Dr. Bernhard Plattner, <Bernhard.Plattner@tik.ee.ethz.ch>
Auftragnehmer	Consecom AG Bellariastr. 12 CH-8002 Zürich Dr. Lukas Ruf, Dr. Arno Wagner, Consecom AG, <Lukas.Ruf@consecom.com>, Tel. +41-79-557-20-20

Dokumentenkontrolle

Ver.	Datum	Autor	Stand, Änderungen
v0.5	11.02.2010	Dr. Lukas Ruf, Dr. Arno Wagner	Draft für Initialmeeting
v1.0	9.08.2010	Dr. Lukas Ruf, Dr. Arno Wagner	Releaseversion

Involvierte Personen

Name	eMail	Funktion
Perrenoud, Nicolas	<Nicolas.Perrenoud@consecom.org>	Student D-ITET, ETH Zürich
Ruf, Lukas	<Lukas.Ruf@consecom.com>	Senior Security Consultant Consecom AG
Wagner, Arno	<Arno.Wagner@consecom.com>	Senior Security Consultant Consecom AG

Verteiler

Name	eMail	Funktion
Ruf, Lukas	<Lukas.Ruf@consecom.com>	Senior Security Consultant Consecom AG
Wagner, Arno	<Arno.Wagner@consecom.com>	Senior Security Consultant Consecom AG
Plattner, Bernhard	<Bernhard.Plattner@tik.ee.ethz.ch>	Professor ETH Zürich



Management Summary

The web browser is a common target for cybercrime operations because people use it regularly to do eCom-merce and online banking. This report describes the development of a maintainable end-user platform for secure browsing. It is implemented as a hardened operating system in a virtual machine providing a confined browser. The layered approach significantly raises the bar for manipulating the user's system through malware and should make attacks uneconomic.



Contents

Management Summary	2
Contents	5
List of Figures	6
List of Tables	7
1 Introduction	8
1.1 Related work	8
2 Requirements	10
2.1 Use cases	10
2.1.1 E-Banking and Stock Trading	10
2.1.2 Online Shopping	11
2.1.3 Secure Browsing	11
2.2 Technical Requirements	12
2.2.1 Security	12
2.2.2 Usability	12
2.2.3 Maintainability	13
3 Design	15
3.1 System Layers	15
3.1.1 Host OS	15
3.1.2 Virtual Machine	15
3.1.3 Guest OS Kernel	15
3.1.4 Network Stack	16
3.1.5 File System	16
3.1.6 Operating System	17
3.1.7 Distribution Packages	17
3.1.8 User installed software	17
3.2 Hardening principles and consequences for our platform	18
3.2.1 Review of kernel features	18
3.2.2 Mandatory Access Control	18
3.2.3 Firewall	20
3.2.4 Package selection and cleanup	20
3.3 Software selection	21
3.3.1 Virtual Machine	21
3.3.2 Kernel	21
3.3.3 Boot Manager	21
3.3.4 Firewall	21
3.3.5 Mandatory Access Control	21
3.3.6 Distribution and package management	22
3.3.7 Window Manager and Desktop Environment	22
3.3.8 Web browser	22
3.3.9 Flash Player	22
3.3.10 PDF Reader	22
3.4 Limitations	22



4	Implementation	24
4.1	Implementation Guide	24
4.1.1	Initial Setup	24
4.1.2	Packages	24
4.1.3	Kernel	25
4.1.4	Guest additions	25
4.1.5	Cleanup	25
4.1.6	Security: Mandatory Access Control	26
4.1.7	Security: Firewall	26
4.1.8	Update-Mechanism	27
4.2	Lessons learned	27
5	Deployment and Maintenance	28
5.1	Distribution	28
5.2	Usage	28
5.3	Maintenance	28
6	Security Analysis	30
6.1	Attacks	30
6.2	Virtualized system	30
6.3	Host system	30
6.4	Limitations	31
7	Future work	32
8	Conclusion	33
	References	34
A	Installation Guide	35
A.1	Step-by-Step Setup	35
A.1.1	Bootstrapping	35
A.1.2	Kernel	35
A.1.3	Package installation	36
A.1.4	VMware Tools	36
A.1.5	Applications	37
A.1.6	Window Manager / Desktop Environment	37
A.1.7	Miscellaneous	38
A.1.8	Firewall	38
A.1.9	SELinux	38
B	Files	40
C	SELinux in a Nutshell	42
C.1	MAC vs DAC	43
C.2	Security context	43
C.3	Type Enforcement	44
C.4	Installation	45
C.4.1	Prerequisites	45
C.4.2	Installation of required packages	46
C.4.3	User management	47
C.5	SELinux Utilities	47
C.5.1	Core tools	47
C.5.2	Management Tools	48
C.5.3	File system tools	49
C.5.4	Policy Tools	50
C.6	Writing policies	51
C.6.1	Example using the passwd program	52
C.6.2	Creating a policy module	53
C.6.3	Using auditing information	54
C.6.4	Boolean switches	54



C.7 SELinux Reference	54
C.7.1 Common rule statements	54
C.7.2 Policy Module Skeleton	55
C.7.3 Common modules of the reference policy	56



List of Figures

3.1	System layers	16
3.2	IPTABLES filter chains	20
4.1	Linux kernel configuration	26
5.1	Maintenance concept	29
8.1	The result: Hardened browser in virtualized Linux	33
C.1	SELinux mechanism	42
C.2	Security context example	43
C.3	Type enforcement example	45



List of Tables

3.1	Filesystem hierarchy standard	17
3.2	Linux hardening techniques	19
B.1	SEBPS related config files	40
B.2	SEBPS related scripts	41
C.1	MAC vs DAC	43
C.2	SELinux core tools	48
C.3	semanage elements	49
C.4	semanage options	49
C.5	SELinux filesystem tools	50
C.6	SELinux policy tools	50
C.7	Common modules of the reference policy	56



Chapter 1

Introduction

The goal of this work is to design a platform for secure eCommerce and web browsing. We define eCommerce as any online transactions involving money done using a web-browser; this can for example be online-banking, shopping or stock trading. Perfect security is not possible as skilled hackers may always find ways to break a system if given enough time, money and access. Additionally, the more one tries to secure a system, the more effort and knowledge from the user is needed to operate such a system. With our platform, we want to raise the bar such that attacks became less economic, but we also keep a strong focus on the usability. Our approach is to deliver to the user a complete hardened operating system (OS) designed for eCommerce. This system is deployed as a virtual machine (VM) image on a portable device (USB-Stick, CDROM, Live-CD...) and accompanied by a free version of a virtual machine software (VMWare Player) for running the image. The platform confines software (e.g. a browser) run inside the virtual machine such that it cannot alter files it does not have the explicit permission to do so. The virtual machine adds a layer of security between processes on the host and processes inside it. This significantly raises the bar for malware to manipulate the browser of the virtual machine guest or processes in the guest system to break out and affect the user's files.

We choose Linux as the guest operating system because it is completely open source, is well documented, a lot of research has been done concerning Linux in general and its security in particular, it has been proven to be enough secure for most purposes (if well maintained, properly configured and hardened) because it's the basis of a lot of high-availability services in the internet, and it has a simple software management.

With that, we hope to deliver an easy to use solution with reasonable security enhancements. This report will show what ideas lead to the design of the platform, which compromises we took, how we implemented the system and how we want to distribute it. At the end, we will draw a conclusion on how we reached our goals and highlight some ideas for future work.

1.1 Related work

Because this is an active field in security business and research, a lot of projects with a likewise focus have emerged during the last years.

One well known solution, especially in the german-speaking parts of Europe, is the ct'Bankix system. It is the approach of a read-only system¹. It relies on a Linux Live-CD which boots a system whose files cannot be compromised because they are all readonly. User preferences and downloaded files may be stored on an additional USB stick. Updates are not possible without burning a CD or saving the Updates on the USB stick and load them on boot.

Another approach is the hardened browser on a portable device. One example is the KOBIL² m-identity stick of Migros Bank³. When it is inserted, it launches a tool which checks for updates, tries to search for some common hacking techniques (like mirror drivers) and then starts a hardened and readonly version of the Firefox browser from the stick, directly opening the login page of the eBanking application. The stick has a slot for inserting a smart card which holds the users client SSL certificate. The browser uses this certificate automatically when

¹<http://www.heise.de/ct/projekte/Sicheres-Online-Banking-mit-Bankix-284099.html>

²<http://www.insinova.ch/produkte/midentity/index.html>

³<http://www.migrosbank.ch/de/MBancNet/MIDentity/default.htm>



opening the banking website. Despite the browser stored as readonly file, its process is a normal application process. It might be possible, though difficult, for malware who has access to the process tree to manipulate it. Keyloggers can always record the login credentials. It is not publicly known how the SSL certificate is transferred to the browser, but it might also possible for a malware replacing the USB device driver to read all data passing through it.

A technology becoming popular during the last years is application virtualization. A single application is encapsulated in a small virtual machine which emulates for example config directories, the registry or libraries. This allows an application to run in an environment that does suit the native application. The security is improved because the applications is isolated from the operating system and poorly written or buggy code may not harm it. This approach is also less resource consuming than a full virtual machine. Commercial examples are VMware ThinApp or Microsoft Application Virtualization. A limitation is that not all software can be virtualized because they require special integration into or support from the underlying operating system.

IBM Research recently developed another approach. Their idea is to provide a pass-through proxy run from a portable USB device⁴. This Zone Trusted Information Channel [1] (ZTIC) stick launches the web browser and connects with pre-configured banking websites when inserted into the computer. It handles the SSL connection for the browser. Encryption/decryption happens in hardware on the stick itself. It looks for sensitive information like account numbers in the data stream (this has to be pre-configured by the issuing bank) and stops the connection when it sees a payment order. The user is then asked to verify the correctness of the account number shown on the sticks internal display by pressing a yes -or-no button. This can prevent from man-in-the-browser or man-in-the-middle attacks. As it needs configuration by the service provider (e.g. bank) and perhaps some modification on the server side, it is not a general solution for eCommerce.

⁴<http://www.zurich.ibm.com/ztic/>



Chapter 2

Requirements

The requirements describe what characteristics the end product should offer. They identify capabilities and qualities of the system and are a guideline throughout the whole design and implementation process. We start from our primary goal that we want a secure system for eCommerce and online banking and then ask ourselves what users exactly want to. We give use cases which describe the system from the user's point of view, where we also look what components need to be protected and how they might be attacked. We also think about our needs as producers of the platform, i.e. how to maintain and deploy it. After gathering the needs, we specify the requirements and categorize them in three categories: usability, security and maintainability.

2.1 Use cases

We begin with giving some use cases that show the possible attacks and implications for the requirements of the system.

2.1.1 E-Banking and Stock Trading

Scenario: This user regularly uses the internet to do financial transactions. He uses the browser to visit the website of his bank, logs in to e-banking/e-trading application check his balance and to make some payments. He sometimes wants to download or print payment confirmation. When he wants to communicate with the bank, he uses the messaging system inside the e-banking application or another channel (phone, direct contact) as banks not often use regular email because of phishing problem. **What needs to be protected:** Login credentials, account number and balance, financial transactions. **Possible attacks:**

- Phishing/Pharming: An attack on DNS System; the user is led to a fake website resembling his bank.
- Keylogger: The attacker reads the users credentials to impersonates him and logs himself into the banking application to transfer money to his (or his mules) accounts.
- Trojans: Manipulate the user's browser as a Man-in-the-Browser attack (cache/extensions/output) to take over transactions.
- XSS/XSRF: Abuse of the user's session through errors in the web application. This out of our scope for our work because it would need action from the application developer. The risk should be negligible for banking websites.

Implications:

- The user needs the possibility to view PDF files
- The User needs a way to save his documents (account summary or payment confirmations, often PDF's) to an external device (USB stick) or a local shared folder.

- We need a hardened system and browser to mitigate Man-in-the-Browser and Man-in-the-Middle attacks. This can be done by confining the browser, update it regularly and disable the installation of new extensions.
- We also should have some phishing protection. Most modern browsers support that with blacklists and tests for letters looking similar to other letters in domain names which are used to fool users.

Considerations:

- An idea would be to allow only SSL traffic (this requires that all banks support that for their regular websites too) or use SSL blacklists to block fake bank websites.
- Another idea for protection against fake websites would be to allow only specific sites to be visited (using local firewall/proxy), a list of banking sites has to be maintained. This may be tricky to handle as the user needs to trust us and we get some sort of censorship control.
- Do we want to allow printing? IT is practical for the user, but does he really need it? Getting printing to work correctly with most printers through a virtual machine may need a lot of effort.

2.1.2 Online Shopping

Scenario: The User visits websites to shop various products online. The products may be for example food, electronic devices, event tickets, pharmacy stuff or some kind of service. He uses his credit card to pay, submits account data of his bank or leaves his address data (e-mail or real address) to get a bill. He also visits search engines or recommendation sites to get information about products. Such sites are often full of advertisements and not always trustworthy. Not all shops use SSL to transfer the final order from the users browser to their servers. What needs to be protected: Credit card number, user name and address, e-mail, visited pages, orders

Possible attacks:

- Drive-by download: Installation of malware through ad banner or wrong configured web application (XSS)
- Trojan/Keylogger: Used to get Credit Card data and/or e-mail address
- Phishing

Implications:

- We need a hardened system and an up-to-date browser to conquer trojans, drive-by downloads and other malware.
- We should warn the user when he wants to visit suspicious sites (one may use a site recommendation/warning extension inside the browser). But this has to be compatible to privacy laws.
- When registering with an online shop, users are often requested to confirm their account by email. To do that, the user needs to check his email by visiting a webmail page if his provider has one, or use a local email client like Thunderbird. If he wants to use his email client, there should be a possibility to copy-paste data to the browser running inside our platform.

2.1.3 Secure Browsing

Scenario: The user wants to browse the internet securely, he is aware that he may visit sites which are NOT trustworthy. Today even serious sites like the UN website spread viruses due to bugs in the webapplication. His system needs to be protected from malware which will install directly through the browser using vulnerability or by opening downloaded files. **Implications:**

- The user wants to use the Flash Player to watch movies (eg. Youtube), play flash games etc.
- He does not want to be infected with malware when he downloads some files; he wants to be sure that downloaded files can not be executed and do some damage to his system.
- We may offer the possibility to clean private browsing data should on exit (cookies/forms/cache/history).

2.2 Technical Requirements

The requirements can be divided into three categories: Security, Usability and Maintainability. The requirements are listed with the layer model in mind from the bottom layer (kernel) to the top layer (user installed applications).

2.2.1 Security

- **Firewall:** The firewall settings should block any external network connection requests to make network scanning harder. It should also only allow connections from local programs like the browser to HTTP and DNS servers and block everything else.
- **Root access:** A normal system user has very limited power over the system (especially on Linux). If he (or a program he executes) becomes root, he gains power over the system. Because we deliver a tailored platform for our use cases, the user need not install additional software or make changes to the system. We thus can disable the possibility to gain root access for the user. Special care has to be taken for programs with SETUID functions. These are programs that run with root privileges, because they need it (for example passwd program which allows a user to change his password).
- **Mandatory Access Control:** This approach to access control requires that a subject (a user or process) doing some action on an object (file, process, port) has to have explicit permission to do that, everything else is denied by default. It allows a fine grained set of policies that clearly specify what a program may do. With that we can confine for example a browser to only access files which are relevant for it. This implements the need-to-know principle. This principle invented by government organizations restricts that even if one has all the necessary official approvals (such as a security clearance) to access certain information, one would not be given access to such information unless one has a specific need to know; that is, access to the information must be necessary for the conduct of one's task. This requirement requires that we check for every program which files it uses in which way and will form one of the foundations of our secure system.
- **Secure Browser:** The most common action of the user using the platform is browsing the web. Because of that, the browser is one of the most attacked targets. We thus need some kind of secure Browser. Browsers are very complex software today because the web offers a lot of different possibilities which browsers have to deal with. It needs to handle connections securely, play flash movies, run java or web 2.0 applications, open document viewers or download files. The question arises how do we rank how secure a browser is? Also important is the acceptance of the user; it should be a browser people generally know well. The browser should have phishing protection and warn about bad websites. One should consider using a SSL blacklisting service to improve phishing protection, if that service guarantees anonymity. An example of such a browser extension is the Firefox SSL Blacklist Addon¹.
- **Browser Extensions:** An attack method which became popular recently especially for attacking e-banking transactions is the Man-in-the-browser attack. This works by adding a browser extension unnoticed by the user. The extension then has full control over the contents the user sees on a website. This is supported by a rich API of the browser. The simplest solution to this threat would be to disable the installation of additional browser extensions.

2.2.2 Usability

- **Fast boot:** The boot process should be optimized because the user does not want to wait long until he can start browsing the web. This can be done by analyzing what is loaded and removing/disabling everything we do not explicitly need. Parallelization of the boot process would give a great boost. This feature, currently developed in the upstart project, is considered experimental at the moment (it is in the testing phase as of June 2010²), therefore we do currently do not require that. Using the suspend abilities of the virtual machine, if offered, can also be used to speed up the startup of the platform.
- **File exchange:** The platform is not intended to be used as permanent storage; the user should rather save his files on the host system or on a portable device. The shared folder feature of virtualization software and pass-through of USB devices can achieve this goal.

¹<http://codefromthe70s.org/sslblacklist.aspx>

²<http://packages.debian.org/squeeze/upstart>

- **Lightweight Desktop Environment:** Common Linux Desktop Environments are feature-rich but depend on hundreds of packages. In our setting where our main purpose is browsing the web securely, we need a lightweight desktop where we can remove everything that adds complexity or distracts the user (by showing irrelevant popup messages, offering too much configuration options etc). A desktop environment with a small footprint also helps saving disk space.
- **System Language selection:** The user needs the possibility to choose the language of the Desktop Applications. At least the swiss national languages should be supported, but it would be even better if we also can offer a bigger set of languages like English, Spain, Turkish etc.
- **Web browser:** The web browser has to be familiar to the user. It should have common browser features such as bookmarks, form auto-completion, history, search and the possibility to set a custom homepage.
- **PDF Reader:** Banking Applications offer balance reports or payment confirmations and online shops offer order confirmations as downloadable PDF document. The user must have the possibility to view these files either directly in the browser or download and open them with a dedicated viewer.
- **File extraction:** A lot of files one wants to download from the internet are packed with the popular ZIP, RAR or TAR/GZIP formats. A graphical archive extraction utility would be preferable.
- **Adobe Flash Plugin:** The flash player is one of the most installed browser extensions, because flash movies or applications are quite popular today, even on serious eCommerce websites (to enhance product previews) or banking applications (for diagrams, statistics or as part of the security system). Because of that we require our browser to have this plugin. On the other hand, the flash player is known to have a lot of security bugs and its security model is regularly criticized in the news. We need to design a mechanism to confine the flash player somehow such that exploited vulnerabilities cannot attack the browser or the underlying operating system.
- **Graphical enhancements:** The boot process should be enhanced with a boot screen, because it adds a professional look to the product and normal users can get confused if they see a lot of messages they do not understand. Also the wallpaper of the desktop should show some sort of product or company logo and the overall GUI design should reflect the company's corporate identity.

2.2.3 Maintainability

- **Disk Size:** As we wish to distribute the platform on a removable device, the size is limited by the chosen medium. The space needed by our system can roughly be calculated by the formula: Base System + Desktop Environment + Apps + Swap + about 20% free space used by the update process (package downloads temporary files and extraction of archived files). We can either use CD-/DVD-ROMS or portable USB devices (USB sticks). As we want to allow the user to make some changes to the system and want to apply updates, USB sticks are a better choice; they also have normally faster reading speed. Our goal is to not get a bigger image than 2 GB so it fits on cheap USB devices.
- **RAM:** 256 MB of virtual memory should be sufficient for a lightweight desktop environment and browser. With that setting, the platform can also be run on older computers which have only 1 GB of RAM.
- **Virtual Machine:** The virtual machine software needs to be easy to install for the common user. It needs to run under Windows, MacOS and Linux as desktop application. It needs support for Linux guests. We require a VM that is free to use (Open-Source or Freeware).
- **Distribution:** We choose Debian Linux (5.0, Lenny) as our distribution. This choice comes for a number of reasons: It is considered stable, because new features are only added after long discussion and testing phases. Its package management system offers a huge quantity of additional software, and it's easy to operate a custom Debian package server to deliver self made packages or mirror the official repository. Its processes, structure and config files are well understood and documented. Its development is completely open and democratic; there are no business interests and no license fees. Furthermore our previous experience with that distribution led to the decision to choose Debian. An alternative would have been CentOS, a fork of RedHat Enterprise Linux, because the SELinux Mandatory Access Control mechanism is neatly implemented right from the beginning in CentOS.



- **Updates:** Applications, especially browsers, are quite often updated. We need some sort of update mechanism which checks for new updates every day (using a task scheduler) or at boot time. This process has to run as root user and fulfill his task automatically without asking for additional information. It is important that all packages have the correct verification signatures so that no attacker may offer fake-packets to the system.
- **Configuration management:** Config files such as firewall rules or policies also need regular updates. One can create a set of related Debian package files and distribute them with the package management, so that no additional steps are required to apply changes in configuration.



Chapter 3

Design

We now describe how we want to implement of our requirements. The questions that arise are how we can raise the security of each of our layers; how we can encapsulate sensitive applications like a browser and what the impacts on the base system are. This chapter will first describe how we split our system in various layers, then cover the security aspect of each layer in greater depth as most work of the project has been done here. Then we explain which software we choose to fulfill our needs best and show the conceptual limitations of our system.

3.1 System Layers

We divide our system in a set of layers to better understand where which requirements apply, where we communicate with the outside world, where we have threat vectors and how we can secure the respective part of the system.

3.1.1 Host OS

The host operating system is maintained by user (or its computer support service). The installed software may be out of date and have unfixed security vulnerabilities. Browsing the web may not be secure because the browser normally runs with the users privileges. The system may be even infected by some malware (keylogger, backdoor, remote control software). The user is encouraged to keep the operating system up-to date and use an anti-virus solution.

3.1.2 Virtual Machine

The virtual machine is produced and maintained by its vendor (VMWare in our case). We assume that it works correctly. Security bugs may be discovered, but patching these bugs depends on the vendor. Most virtual machine solutions offer security features like disk encryption, protection from keyloggers (VMWare Workstation on Windows for example catches keyboard inputs before they can be read by the keyboard hook WH_KEYBOARD¹; this can be tested by creating a simple software keylogger in your favorite programming language²) or an API for third party security products³.

3.1.3 Guest OS Kernel

The guest OS kernel (in our case Linux $\geq 2.6.32$) is the interface between the (virtualized) hardware and the hardened applications. It does all the memory management, process scheduling and driver (Input devices, network cards, storage drives, graphics cards) handling. Processes running in the kernel context (the kernel space) have full privileges over the system. The kernel is loaded by the boot manager. It starts the init process after initializing all devices.

¹ Windows Global Hooks, <http://msdn.microsoft.com/en-us/library/ms644959%28v=VS.85%29.aspx>, 21.6.2010

² Keylogger Source Code, <http://www.codeproject.com/KB/cs/globalhook.aspx>, 21.6.2010

³ VMSafe, <http://www.vmware.com/technical-resources/security/vmsafe.html>, 22.6.2010

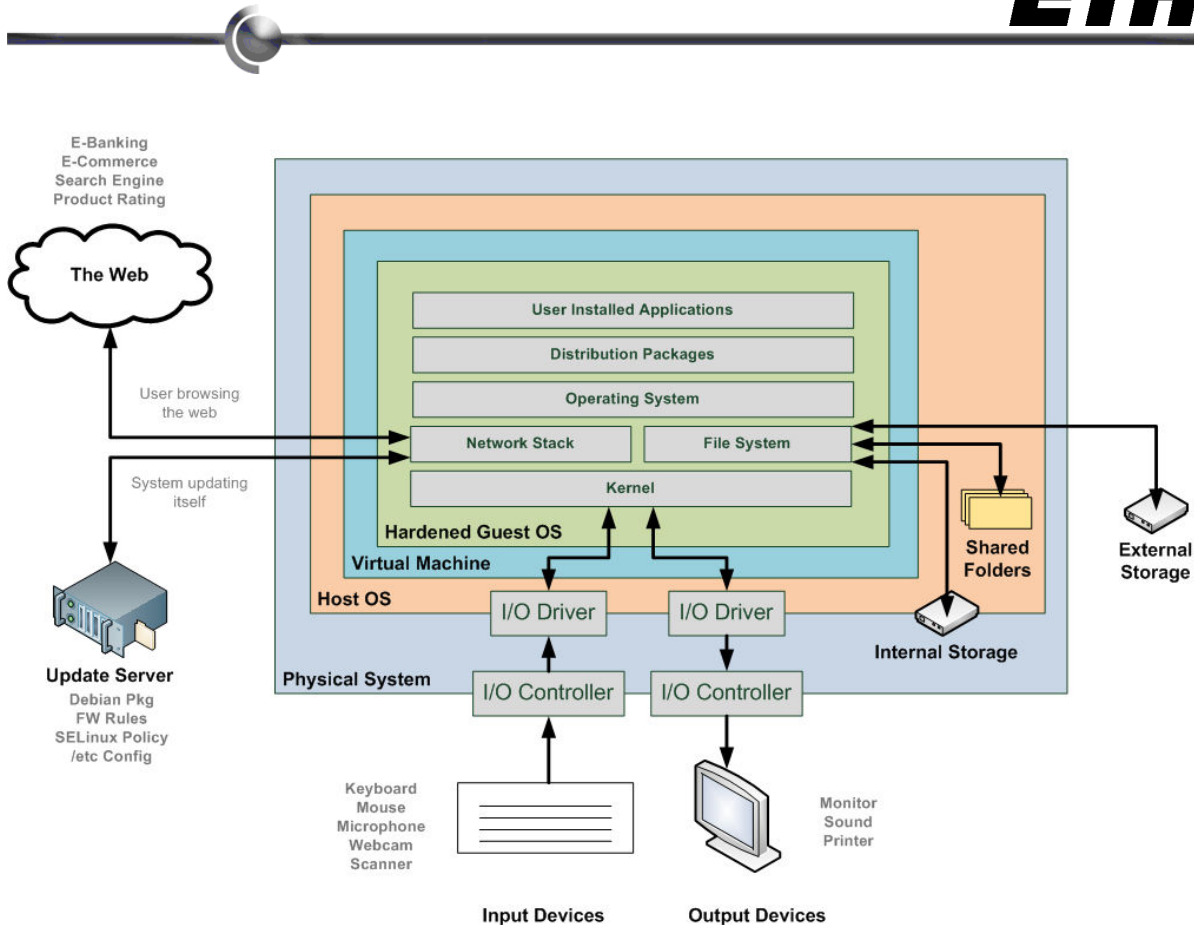


Figure 3.1: System layers

3.1.4 Network Stack

The network stack is a kernel subsystem and covers most of the OSI-Layers. Every network packet passes through it. It offers low-level network services like ICMP messages and ARP. The Linux firewall Netfilter (which can be configured by its interface iptables) is directly integrated into the network subsystem.

3.1.5 File System

Access to the file system is integrated as another subsystem into the kernel. Support for specific types (eg. Ext3) has to be enabled in the configuration before building the kernel. Besides the kernel support we see the following components as part of the file system layer:

- The file system table (configured in `/etc/fstab`).
- The Root file system according to the filesystem hierarchy standard⁴. A list of the directories under the system root is shown in table 3.1.5.
- The swap space for swapping out processes that are not in use when there is no free memory left.
- Mount points for external storage devices.

⁴<http://www.pathname.com/fhs, 21.6.2010>

Table 3.1: Filesystem hierarchy standard

etc	Configuration files and scripts
bin	Essential command line binaries
lib	Essential system libraries and kernel modules
usr	Unix system resources (libraries, binaries, shared files, source code)
home	The user homes
var	Variable data files (logs, mail queue, databases)
boot	Boot loader and kernel
mnt	Mount points
tmp	Temporary data

3.1.6 Operating System

The operating system is the base system provided by the Linux distribution vendor. It has the following components:

- Standard Unix Toolchain (ls, cd, cat, touch, rm, ps and many more)
- User management tools (passwd, useradd, groupadd, chown, chmod, chroot)
- Shell (bash)
- System libraries in /lib, /usr/lib, /usr/local/lib

3.1.7 Distribution Packages

Nearly every Linux distribution has some sort of package management to ease the installation of new software and keep the system up to date. This layer covers all the various software installed by the package management:

- The package management itself
- Window Manager (The X-Windows System) & Desktop Environment with X-Login Manager (xdm/kdm/gdm)
- Console and desktop applications
- Configuration files
- Languages and regional settings
- Virtual Machine Client Tools (Hardware Hotplugging, Notifications, Power Management, Shared Folders, Shared Clipboard)

3.1.8 User installed software

Users could install some software regardless of a package management system (which only the root user may use). They can download executable binaries, scripts or some source code to their home directory and run it if no security rules prevent them to do so. Some applications like web browsers have a custom package management like an extension manager which stores its files also in the user home directory.



3.2 Hardening principles and consequences for our platform

The most important task after setting up the system is to harden it. Hardening means to modify existing objects of the system in a way that they are harder to attack. The security of a Linux distribution after installation is often too lax because the focus is on being able to run all programs the user wants to use without any problems. A lot of techniques have evolved over the years that help to harden the system. There are different techniques for every layer of the system. The table 3.2 will list the most common hardening techniques along with the layer they affect.

Most of the techniques listed in the table were applied to our system in some way. The foundation of our hardening approach lies in four points: Review of kernel features, enabling Mandatory Access Control, firewalling and package management.

3.2.1 Review of kernel features

An important security principle is that complexity is our worst enemy. The Linux kernel is a really complex system with millions of lines of code. Despite the thousands of programmers that work on and review the code, bugs are not avoidable. To make the kernel more secure, we have to remove everything that we do not need to reduce the attack vector. In our case, where we use a clearly defined environment (a virtual machine), this task is not too difficult, nevertheless the process is time consuming as it involves some try-and-error cycles. In principal a lot of drivers and file systems and support for special architectures can be removed. For our security measures, some other features need to be enabled which are inactive by default (Auditing, Firewall, Mandatory Access Control).

3.2.2 Mandatory Access Control

Most available software is not programmed with security concepts in mind. Even in operating systems vulnerabilities are discovered on a daily basis. The sheer complexity and quantity of today's software makes it very hard to test all possible states of a software. Therefore design flaws and bugs will always exist. In recent years a lot of research has been done in this field and various mechanisms have evolved that try to minimize the impact of an exploited security hole[2]. One such mechanism is Mandatory Access Control (MAC) where the Operating system can confine programs and allow only those actions that are considered "good". Mandatory access control sees active processes as subjects and files, ports, shared memory and other processes as objects. All subjects and objects have security attributes. A subject wishing to perform an action on an object requires special permission to do so. This permission is stored as an authorization rule and enforced by the OS kernel by examining the security attributes. The set of rules is called a policy. By default, every action is forbidden. Thus the policy can grow quite large to cover all available applications. A rule has the form allow subject object permissions. There exist numerous MAC solutions for Linux, among which SELinux, AppArmor and Tomoyo are the most common.

- SELinux⁵ was developed as a research project by the National Security Agency (NSA). It added MAC directly into the kernel and was integrated in the mainline version in 2003. It uses security labels on files and processes to enforce its rules. RedHat (RHEL, CentoOS, Fedora) is one of the drivers of SELinux. They develop it further, provide extensive documentation and integrate it in all their products. Stable ports to various other distributions like Debian or Gentoo exist.
- AppArmor⁶ was maintained and advertised by Novell and is not part of the kernel as of today. It works with file paths (every application identified by its file path has its own security profile) and can be used to confine single applications, but it is not capable of restricting access to the whole system. AppArmor is considered to be simpler to maintain and it does not need explicit support from the file system. On the other hand one cannot enforce a strict "deny everything except what we allow" policy and the path-based identification can be fooled with hard- or symlinks. AppArmor is now increasingly used within Ubuntu Linux distributions.
- TOMOYO⁷ tries to achieve the same goal as AppArmor. It is lightweight and suitable for embedded Linux and has been recently (June 2009) merged into the kernel.

⁵<http://www.nsa.gov/research/selinux>

⁶<http://de.opensuse.org/AppArmor>

⁷<http://tomoyo.sourceforge.jp>

Table 3.2: Linux hardening techniques

Technique	Explanation	Attack/Impact
Minimizing installed software	Remove all software that will not explicitly be used. Minimizes the overall complexity of the system and the number of open doors for attacks.	Less systems complexity = less vulnerabilities
Patching the system	Update system regularly to fix recently discovered bugs.	Discovered security holes get closed
Securing filesystem permissions and S*ID binaries	Check the permissions of application, log and temp directories. Search for binaries allowing to be executed as root user.	S*ID would binaries allow root access if they have vulnerabilities that allow opening a shell.
Improving login and user security	Remove unused users and groups. Enable a password policy. Review the groups the user is in. Decide if user may become root.	Reduces the chance that an attacker can become root
Setting some physical and boot security controls	Secure BIOS and boot loader. If we use real hardware also consider securing it (physical data security, locks)	Sets the bar higher for attacks on the real machine (e.g. running a Live CD to gain access to the harddisk).
Securing the daemons via network access controls	Enable firewall policy and/or network intrusion detection systems.	Mitigate direct attacks over the network. Reduce the possibility that downloaded malware may communicate to the outside world.
Increasing logging and audit information	Collect all logs in a central place and consult them regularly. Install a log management and alerting system.	Allow forensics if system has been compromised
Remove unused kernel features	Remove all features from the kernel that are not used (e.g. drivers, server features, special filesystems). This also reduces boot time and kernel size in the RAM.	Reduce set of attack vectors by minimizing the kernels complexity.
Tuning kernel security features	The kernel has various runtime options to make program execution more secure, e.g. by turning on stack randomization or disabling executable code on the stack.	Makes buffer-overflow attacks harder
Tuning kernel network features	The kernel has various runtime options to limit the vulnerability of the network stack.	May enhance network speed and protect from various network based attacks (Spoofingm, Flooding, Scanning)
Enabling Mandatory Access Control	Mandatory access control allows user or processes to perform only actions they are absolutely allowed to.	Processor are only allowed to do actions on files that are explicitly allowed.
Use stable software	Use only software packages that are marked stable and have proven their reliability in long term use.	Reduce the risk that new vulnerabilities are discovered because many eyes have already checked the code
Check config files	Check the config files along with the manual of the installed software to see if they offer options to enhance security.	Lax configuration may open doors into the system (e.g. default enabled anonymous ftp user).
Turn of unused services	Turn off or even uninstall system services that are not or only sporadically used.	Reduce attack vectors by lowering number of open ports and running processes.

We decided to use SELinux because it is integrated into the kernel. It is stable since many years and therefore quite common. Its popularity has grown over the last years and its available in most distribution (especially). AppArmor or TOMOYO are as of now in the unstable version of Debian and its not recommended to use them

for production systems. For more information about SELinux consider the chapter "SELinux in a Nutshell" in the appendix.

3.2.3 Firewall

A software firewall permits authorized network connections and blocks unauthorized access. It operates on the network stack in the kernel and guards both outgoing and incoming network streams. The decisions are made based on a set of rules. The rules consists of IP-addresses, port numbers, application names, user ID's, data patterns etc. There exist several firewalls techniques like packet filtering, application layer firewalls or proxies. (Stateful) packet filters use only the information contained in a packet itself. They are easy to configure shipped with most operating systems (for example Netfilter/IPTABLES in Linux). Application firewalls are more powerful because they understand protocols (for example HTTP), but they are more resource consuming, need very careful configuration and are mostly build for server operations[3]. Proxies are dedicated gateways in the network which act as a server for the user and as a client to the outside real server. They keep clients behind them anonymous and offer content auditing, caching and filtering.

The firewall concept for our platform is rather simple. We use Netfilter/IPTABLES (the default Linux firewall) and no application layer filtering because it is not feasible for our use case, as it needs a lot of maintenance and complex rules and we do not want to impose restrictions on which websites a user may visit. A proxy is also not suitable because we do not need any of the features it provides. There are only three applications that are allowed to connect to the internet:

- The web browser (and his plugins like Flash Player or Adobe Reader).
- The network subsystem to do DNS requests.
- An update-mechanism (custom shell scripts and distribution package management).

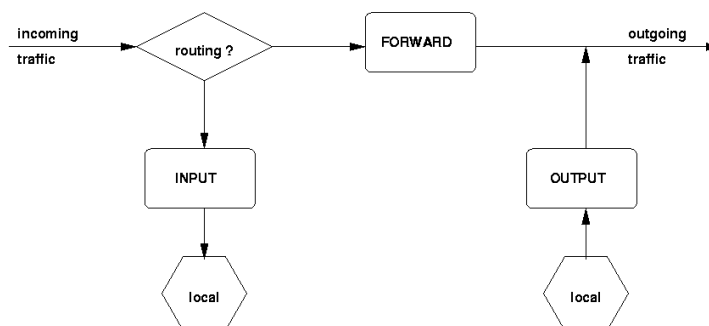


Figure 3.2: IPTABLES filter chains

As there are no server applications running, any connection attempts from the outside world must be blocked; only packets belonging to existing sessions are allowed through. Outgoing connections are only allowed to web servers (http and https ports 80 and 443) and only from the browser by the normal user or from the update tool run by root. Allowing only connections to DNS and HTTP ports does not mean that only browsing the web is allowed. There are various projects that try to use these channels for any kind of data by using tunneling strategies⁸ ⁹. Because we also employ mandatory access control to specify which programs a user may run and in what context, this risk can be minimized.

3.2.4 Package selection and cleanup

The more software is available on a system, the more attack vectors exist. By removing all applications that we do not explicitly need, we make a hackers work more difficult, so for example if we remove the GNU compiler collection, he cannot create some C programs on the fly. Also by stopping all server services that are not really needed, we reduce the amount of open ports, shared memory segments, log files and temporary data. There

⁸<http://www.dnstunnel.de>

⁹<http://www.nocrew.org/software/httpunnel.html>

is also another reason for removing as much packets as possible and this is disk space. Because we want to offer the system on a removable USB drive our goal is to not have a system larger than 1 GB. The following list will show some of the packages we would remove:

- All Xorg drivers except the one needed (VMware graphics, keyboard, mouse).
- X utilities like terminals, screensavers, graphical config tools.
- Network tools like ftp, telnet, traceroute or portmap.
- Server software like tcpd, ntpd, intetd.
- Loggers like syslog.
- Building tools like gcc and bindutils.
- Man pages, documentation, shared data files (fonts, graphics) and languages.

3.3 Software selection

In this section we describe which software we choose in the end to do the tasks specified in the requirements. The list is again ordered according to our layers.

3.3.1 Virtual Machine

We choose VMWare (Workstation for development, Player for deployment) as our virtual machine as it fulfills all our requirements (Desktop application, multi-platform support). VMWare's MacOS variant is called VMWare Fusion. Additional reasons for choosing VMWare are that it is widely deployed, well known, has proven its reliability, the guest tools offer a lot of features (shared folders, resolution scaling, Unity) and the snapshot manager of VMWare Workstation helps a lot in developing the platform. Alternatives would have been Virtual Box (Open Source) with which most things work quite similar, but its guest tools are not as powerful and also a bit unstable (by personal experience). It was also a business decision because VMWare Player or Workstation is widespread in use in enterprise environment so users will not have to install another virtual machine application.

3.3.2 Kernel

We choose Linux (currently version 2.6.32) because it can be customized in many ways, is well documented and understood and it is considered quite secure when properly configured and hardened, because lots of security researchers and volunteers review the code regularly.

3.3.3 Boot Manager

We choose lilo over grub because it is easy to configure, has a smaller footprint and loads faster than grub. The drawback is that one has to call lilo after every modification to the config or after a kernel update (this can be automated).

3.3.4 Firewall

We choose the Linux Netfilter/IPTABLES system to do this job because its simple to implement (it is already contained in the kernel), does not use a lot of resources, well documented and reliable.

3.3.5 Mandatory Access Control

We choose SELinux because it is the most widely used MAC technique for Linux. A lot of research has been done about it, there are tutorials and documentations available on numerous websites and it allows a deny everything default policy (which other solutions like AppArmor or Tomoyo do not offer).



3.3.6 Distribution and package management

We use Debian Linux 5 (Lenny) because it has no business interests coupled with it, is well documented, the whole development process is open and we already have experience in setting up and maintaining Debian systems. It is possible to set up a dedicated package server/mirror to distribute updates of our configuration and script files. Debian has a conservative package update policy which means that one may not find the newest cutting-edge software in its repositories, but rather stable and well proven packages. Some software we want to use needs additional package sources because they are not available in the stable tree. An alternative would have been a Red Hat influenced distribution like CentOS or Fedora.

3.3.7 Window Manager and Desktop Environment

The foundation of the graphical environment is the Xorg Server (there is really no reliable alternative as of today) and we use LXDE as desktop environment. There are various other lightweight desktop environments like FVWM, EDE, XFCE or Fluxbox but LXDE seems to use the fastest one and the design of its user interface keeps up with the time, which means that the alternatives often have a old ugly design that cannot be compared to what a user knows from using Windows or MacOS.

3.3.8 Web browser

The requirements for the browser are that it is well known to the user, runs stable on Linux, regularly updated and considered more secure than its alternatives. The two primary options are Firefox (called Icedove in Debian) and Google Chrome. Firefox can be extended with lots of plugins. Chrome has a short startup time and tries to implement new security techniques faster than its competitors¹⁰ (for example every tab and every plugin runs as a dedicated process). When checking vulnerability databases^{11 12}, both browsers have around the same amount of security related bugs. The reason why we finally choose Firefox over Chrome is that Chrome is just too young and because of that not all its behaviors are tested or well understood. Firefox adds support for plugins as dedicated processes in June 2010, so it will then be possible to confine the plugin process even more than the browser process with mandatory access controls. Alternatives to Firefox/Chrome would have been the browsers of the various desktop environments like Konqueror or Epiphany but they are not known to most users and depend on a lot of libraries (QT, GTK) which we do not want to install because they eat up a lot of disk space and increase the set of attack vectors.

3.3.9 Flash Player

There is no reliable alternative to the official player which is well known for having a lot of bugs. But we can decrease the risk for exploits if we manage to confine the player.

3.3.10 PDF Reader

An official version of the Adobe Reader is available for Debian Linux. There exist also some open source readers like xpdf, evince, ePDFView. Choosing the Adobe Reader was mainly a business decision but also a decision of usability, because it integrates well into the browser and most user know how to use it. A drawback is that we need to manually add it to the Debian package management and that the Adobe Reader needs a lot more disk space than its competitors.

3.4 Limitations

Our design should significantly reduce the risk of a user getting his system infected by malware if he browses the web with the secure platform. What it does not cover is the security of the host system. A rootkit on the host may control all inputs and outputs and may even try to manipulate the virtual machine if he is aware of it (but

¹⁰Ars Technica: <http://arstechnica.com/security/news/2009/03/chrome-is-the-only-browser-left-standing-in-pwn2own-contest.ars>

¹¹Secunia Firefox 3.6: <http://secunia.com/advisories/product/28698>

¹²Secunia Chrome: <http://secunia.com/advisories/product/28713>

that needs great efforts and may not be economic). More on that subject can be found in the security analysis chapter.

We do not offer support for smart-card readers which need to communicate with some system components or the browser, as drivers in Linux may not be available for this mostly proprietary solutions. Also the virtual machine needs to route the data correctly from the USB connector on the host to the internal virtual USB connector; this does not always work correctly (as of personal experience) and may need several tries which frustrates the user.

Solutions like the ZTIC or KOBIL stick (described in 1.1) may not work either, because the ZTIC stick provides a proxy (which is implemented as a library and a browser-plugin) and the KOBIL stick a hardened browser; both require installation of additional software and special configuration of our policies and it is not guaranteed that everything will work together. If client SSL certificates are required (KOBIL), the user would have to import them manually in our browser. ZTIC would need to install a browser plugin and some sort of proxy driver either in the host or guest system. It must be evaluated if that would work (we currently do not have a ZTIC stick in our possession to test this), but in our opinion it will not be possible as our browser already has established a SSL connection when connecting to an eBanking website and it will not automatically search for a proxy on the host system.

Printing to real printers is not planned because of lack of availability of drivers and corresponding support of the virtual machine for USB and parallel port printers. Network printing may work if printers support the ability to print documents that are simply sent as PS or PDF data streams. It is possible to save a document/website as PDF using the embedded printing dialog of Firefox and the desktop environment and moving the output file to the shared folder to print it later using the host system printing mechanisms.



Chapter 4

Implementation

This chapter describes the actual implementation of our secure platform. What we want to achieve is a hardened Linux system inside a virtual machine to add several layers between a secure browser a user uses for eCommerce and his normal operating system where he stores his data and runs various third party programs with unknown impact on security. It has been elaborated in the chapter design which techniques we want to apply to make the system more secure and what software we want to use.

The general steps are as follows: We begin with setting up an initial Debian system inside the virtual machine. We add all required software like the virtual machine guest tools, a desktop environment, the browser and document viewers and verify that everything works. In a next step we customize, harden and shrink down the kernel. After we verify that everything still works we remove unused software packages and cleanup the system. The next step is the most important: Hardening the system using the techniques described in the design section. This consists primarily of setting up mandatory access control, enabling the firewall and checking configuration files. We finish with setting up an update mechanism.

4.1 Implementation Guide

We will now describe the process in greater depth. A complete summary of what has been done can be found in the appendix.

4.1.1 Initial Setup

We start by creating a virtual machine image in VMware workstation. Initially we set the disk size to 2 GB to have enough space for temporary data; the goal is to use less than 1 GB in the end. Then we start the virtual machine and install Debian 5.0 Lenny. We create three partitions: The root partition, the swap partition (we reserve 256 MB RAM for the virtual machine and create swap space of 128 MB) and a partition for variable data where we store the directories /var and /home. This partition can then be mounted with the noexec flag to prevent that a user may execute something from his home directory. After the installation has finished, we mount that partition at /mnt/varhome and alter the fstab accordingly, move the directories /var and /home to it and create symlinks at the old locations. A local user should be created during installation, we call it sebbs. It is important to check that he has the correct permissions to read and write his home directory.

4.1.2 Packages

Now we install all additional packages we need and remove others that were preinstalled and we do not need (A list has been created using `dpkg --get-selections` do automate this step if one wants retry the implementation process). Some packages like the Acrobat Reader or Flash need to be downloaded directly from their vendors' websites or special Debian repositories dedicated to provide proprietary packages. We also install the Xorg Server and the LXDE desktop environment (no special configuration is needed).



4.1.3 Kernel

The kernel customization is a rather time-intensive try-and-error process. We use a separate Debian system for building the kernel (faster and we do not need all the build tools and sources in our disk-space-limited system). The process involved three steps and was rather practical in nature:

1. Decide which core kernel features we need
2. Identify what devices we have on the system (using the documentation of the virtual machine and specialized tools like lspci)
3. Start with the standard kernel shipped with the distribution. Remove everything we have not listed until the system does not work. This process works in iterations of removing a feature, compiling the kernel, installing it, rebooting and testing if everything still works as expected (devices, network, graphics and applications).

The following list will name the most important changes to the kernel:

Added features:

- Auditing and SELinux support: Needed for mandatory access control
- Security labels for filesystems: Needed for mandatory access control
- Netfilter and IPTABLES: Needed for firewalling

Removed features:

- Initrd and loadable module support: We want a small compact kernel
- Multicore and other platforms than x86: As we are in a VM, we do not need this
- Power management: Same reason as above
- A lot of drivers: PCI Express, ISA, ATA, Parallelport, Sound, Multimedia, USB
- All filesystems except the ones we use (ext2/ext3)

4.1.4 Guest additions

The VMware guest additions require a manual install. They should be installed after the custom kernel has been set-up because they need the correct kernel headers. The virtual machine provides the setup on a virtual CD-ROM. After unpacking, the setup script must be called which does all necessary modifications. The kernel headers and the build tools (make, binutils, gcc) are needed to compile some of the modules. After that one has support for shared folders (can be found under /mnt/hgfs), autofitting of the screen resolution, seamless cursor and keyboard and enhanced networking support.

4.1.5 Cleanup

We remove all packages and other data we do not explicitly need. We do this cleanup process by printing out a list of all installed packages and check each one carefully (with the aid of package directory documentation and the tool aptitude which shows the package dependencies graphically). Additional tools which help in this process are deborphan which identifies unused packages (for example libraries that are no longer in use) and localepurge which removes all locale and language support for locales that are not enabled.

```

.config - Linux Kernel v2.6.32.10 Configuration

Security options
Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted letters are hotkeys.
Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for
Help, </> for Search. Legend: [*] built-in [ ] excluded <M> module < > module capable

[ ] Enable access key retention support
[*] Enable different security models
[*] Enable the securityfs filesystem
[*] Socket and Networking Security Hooks
[ ] XFRM (IPSec) Networking Security Hooks
[ ] Security hooks for pathname based access control
[*] File POSIX Capabilities
(65536) Low address space for LSM to protect from user allocation
[*] NSA SELinux Support
[*] NSA SELinux boot parameter
[*] NSA SELinux boot parameter default value
[ ] NSA SELinux runtime disable
[*] NSA SELinux Development Support
[*] NSA SELinux AVC Statistics
(1) NSA SELinux checkreqprot default value
[ ] NSA SELinux maximum supported policy format version
[ ] TOMOYO Linux Support
[ ] Integrity Measurement Architecture(IMA)

<Select> < Exit > < Help >

```

Figure 4.1: Linux kernel configuration


4.1.6 Security: Mandatory Access Control

This is the most time consuming part of the implementation, because the correct rules for MAC have to be found by try-and-error and understanding how applications work which involves reading a lot of documentation. We use SELinux to enforce the MAC mechanism. A general introduction and tutorial to SELinux can be found in the appendix. SELinux is bundled with a reference policy originally developed by the NSA and now maintained by Tresys and the community. This reference policy has reasonable rules for all the default system and kernel processes and all well-known server services. After SELinux has been installed, we need to add custom rules to get it to work correctly with our set of applications. We set its mode to permissive, which means that violations are logged in the kernel log (can be displayed with dmesg) but actions are always allowed. We use now our 'clean' system in the manner a user would use it (starting up the graphical environment, launching the browser, opening documents, using the shared folder etc) and review the log after that. From that we can develop our rules. We create a custom policy module for our base system (called sebps) and one for the browser (firefox) and the adobe reader (acroread). To ease the development, the tool audit2allow can be used, which takes the error log and creates rules out of it.

It's important to review and rethink each rule, because not everyone is really needed. This process works in many (really many) cycles because in permissive mode somehow not all violations are shown at once; some rules need to be set and applied to show the next violations (we automated this process with some bash scripts). When everything works without throwing errors, we can enable the enforcing mode and test the functionality again. The browser, flash player and acrobat reader need to run in special domains; for that we need to write some rules by ourselves according to the guide in the appendix. When finished, we can be sure that each process runs in its own domain and is only allowed to do what is stated in the policy. As new versions of Firefox (since version 3.6.4) run every plugin in a special container (a process called plugin-container), we are able to confine plugins further and create dedicated domains for the Adobe Flash and Acrobat plugins. With that, we can hinder malicious Flash movies or Acrobat documents to contaminate the running browser or access files to which the browser itself has access.

4.1.7 Security: Firewall

We use IPTABLES to setup firewall rules. We create a bash script that contains all our rules according to the section in the design phase. To load this rules initially we execute this script. After that, we add calls to iptables-



load and iptables-save to the network hooks in /etc/network such that the rules are saved after the network goes down and loaded before the network goes up.

4.1.8 Update-Mechanism

We only implement a simple update mechanism using the already available package management tools, leaving a powerful maintenance solution as a task for future work. Updates can either be checked at boot or on regular basis using anacron (not cron, because our system runs not the whole day). For that we create a bash script that calls the update, upgrade and cleanup mechanisms of the apt package manager and add it to the runlevel system.

4.2 Lessons learned

In the implementation process we discovered some technological problems and limitations, For example when configuring the firewall, we saw a drawback of the Linux Netfilter; one cannot match rules for a process name, so we cannot just allow the web-browser to connect to some port, we have to allow all applications a given user may run. There existed a process-name matching mechanism but that was removed in the current kernel version, because it provided no real security as any process may alter its command name with some tricks. We have to solve this issue with mandatory access control.

A drawback of the MAC mechanism of SELinux is that it can only add security contexts to files if the file system has support for extended attributes (xattrs). When using shared folders to copy downloaded files to the Windows Documents folder, the copied files have an unlabeled context as NTFS does not support labeling. One can of course add rules for handling unlabeled files, but all files in all shared folders have the same context, so we do have to take care which shared folders in which locations we set up.



Chapter 5

Deployment and Maintenance

This is a short chapter because it was not the main focus of the work. Distribution is rather simple as we already specified in the beginning that we will implement the platform on a virtual machine image. Everyone can use that image because the virtual machine player runs on all major desktop platforms. Maintenance is described as a concept and the real implementation is left as future work.

5.1 Distribution

We designed our secure platform as a VMware virtual machine image, which requires either VMware Workstation (needs to be purchased) or the free VMware player to run. We intend to pack the image and a distributable version of the VMware Player on a portable media (USB Stick or CD/DVD-ROM). The user would then have to run the VMware Player setup, copy the image to his local hard drive and load it into the player. He also has to configure a shared directory so that he can exchange files between his operating system and the system inside the virtual machine. To make the setup more user-friendly, we will create a custom setup application that asks for the location of the image, sets up the shared directory and creates a link on the desktop.

5.2 Usage

To use the platform, the user opens the virtual machine image in the VMware Player and boots the system. After the desktop environment has been loaded he can start browsing the web. We set the browser to start automatically when the LXDE desktop manager has finished loading. The user can use the embedded PDF viewer to read documents and he can save them to a Downloads folder, which has the shared directory as a subfolder. Every file he copies into the shared folder is immediately available on the host system.

5.3 Maintenance

Maintenance from our point of view is mainly auto-updating the system. The user does not have to do anything to maintain the system. We use the Debian package management to download and install new updates. Our modifications (kernel, policies and applications from outside the package tree) are packed into Debian packages. At boot, a script checks the package server if any updates are available and installs them if necessary. The SELinux MAC policies have to be extended to allow this operation. Sometimes a package update can create conflicts or even break a system. To avoid this, we create a dedicated Debian mirror server and the system pulls all packets only from this server. We can verify with a testbed of our system that the update works correctly and only after that unlock the packets for downloading. Every packet's Debian signature has to be checked, then we need to sign them with the signature of our package server. The public key of this signature has to be loaded into the package management of our hardened system and all other custom keys removed. Thus we can make sure that only packages signed and verified by ourselves are downloaded onto the system.

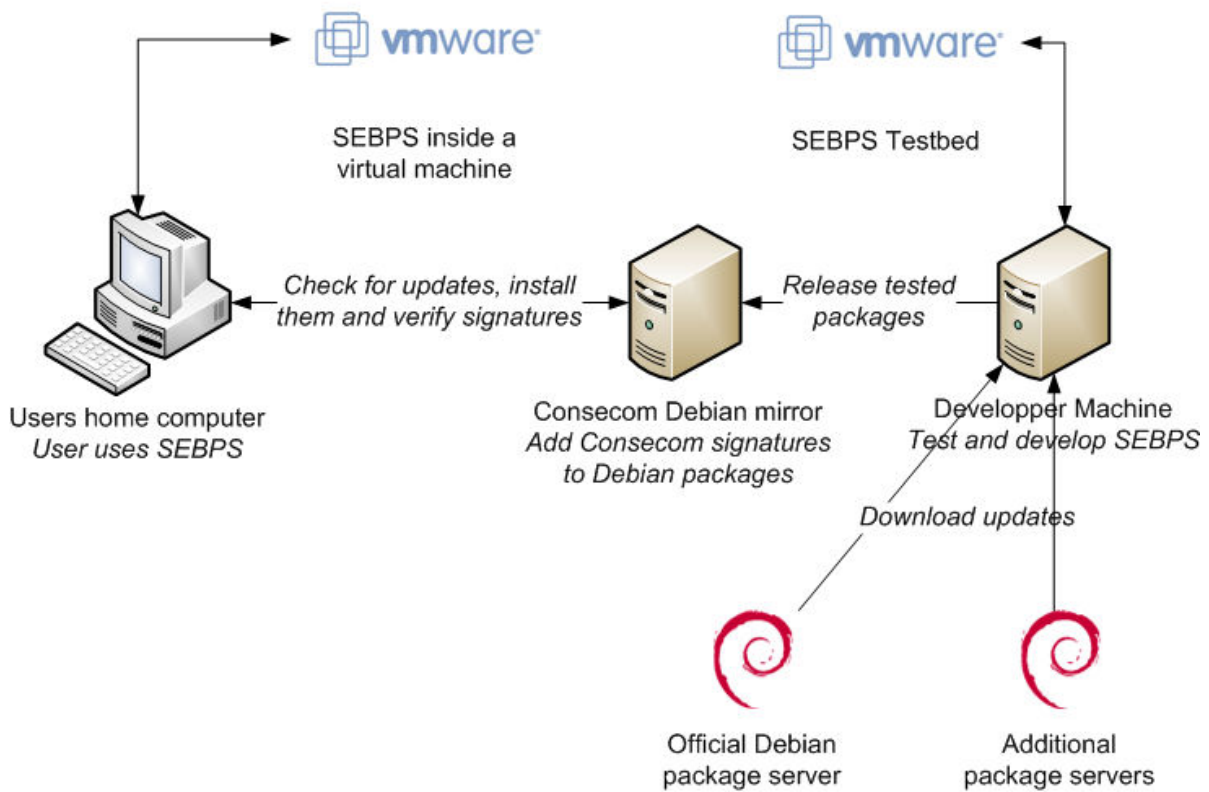


Figure 5.1: Maintenance concept



Chapter 6

Security Analysis

The security analysis will describe which security goals we reached and what we do not achieve. Security is often a compromise between being more secure and imposing some usability restrictions onto the user, for example more actions are needed to reach a desired result; more knowledge about security topics is needed or more time is consumed because the system's performance is lower. If a cybercriminal is given enough time, money and resources, he can successfully attack almost any system. Our goal is to augment the security of the system to a level where attacks become uneconomic but the usability is not reduced too much (see section 3.4).

6.1 Attacks

As we use a virtualized system, we distinguish two possible attack methods. Firstly, malware on the host system could read or manipulate the inputs (keyboard, mouse) and outputs (display) of the system or hijack the internet connection. It also can try to attack the virtual machine image when it is not used or try to break into a running virtual machine. Secondly, malware could be downloaded and executed in the virtual machine unnoticed to the user. The malware could then infect the virtual machine and its applications, not knowing that it runs in a virtual environment. Virtualization aware malware could try to break out and infect the host. Both is dangerous, because malware inside the VM can act as a man-in-the-middle or man-in-the-browser to read sensitive data (e.g. credit card number, account balances) or actively manipulate transactions. When it is able to break out, it is a threat to the documents of a user or even other machines in the network.

6.2 Virtualized system

Virtualization is not strictly counted a security feature, but in reality it makes it quite hard for malware inside the VM to break out. It is the duty of VM vendors to ensure that this might not happen. Users of virtualization software need to check regularly for updates (today most virtualization solutions do this automatically) because vulnerabilities appear in all virtualization products. With the security model of Linux and additional mandatory access control using SELinux, we can ensure that applications only do what they're explicitly allowed to do. SELinux has been proven to be quite secure in many environments and it is supported and developed further by major Linux companies and the NSA. Our concept of hardening every layer of the system raises the bar even higher, as many different techniques need to be broken to finally manipulate the system in some way.

6.3 Host system

Breaking into the VM from the operating system is considered easier, because if the image is not somehow encrypted, a specially tailored malware can read and write the image file. Even so, a malware author needs broad knowledge of the inner layout of the file systems, the operating system and the applications running in the virtual environment. It is quite possible that some offline modifications to a VM image can render the virtualized system unusable and therefore minimize the benefit for the attacker to get some information from the user when he wants to use the VM. He then also risks to be detected by the user if suddenly some things do not work as



they should. One advance that plays into our hands is that our solution is for now a niche product and thus it will likely be that it is not economic to develop a virtualization and SEBPS aware malware that does exactly know the layout of our system and can manipulate the image at the correct locations without breaking something. Malware on the host that tries to hijack the connection coming from the virtualized and going into the internet will not be successful if the user browses websites with SSL encryption (which is the normal case for banking and eCommerce websites), but the user should know the concept of a SSL certificate and know the basic rules (check the domain name, read the certificate, not answer to mails requesting login data) how he can verify that he is indeed communicating with the banking servers, as phishing may still be possible. We help the user to go to valid login pages by providing a maintained home page for the browser (<http://swissbanks.sebps.net>) with direct links to all major swiss e-banking sites.

Malware running on the host might also not be capable of logging what the user types inside the virtual machine, as VMware captures keyboard inputs before they are processed by the Microsoft Windows keyboard-event broadcast system. A simple example written in C#¹ can prove how easy it is to record keyboard inputs in windows, but also shows that VMware captures them and does not let Windows process them further. There may exist sophisticated malware that sits deeper in the system, for example as a driver that still can read every keystroke, but its functionality could conflict with the virtualization software (this topic would need further research on the availability of such malware). Hardware key-loggers are of course out of scope and there exist no user friendly techniques to detect them, but they are also harder to install.

6.4 Limitations

What we can not prevent are attacks using remote control software where the attacker could distract the user somehow and then control the virtualized application to manipulate an ongoing financial transaction. This form of attack would need a real person with knowledge of the banking application he wants to attack and he needs to be fast and have a good distraction technique (for example simulation a short freeze/flicker of the desktop which perhaps might not alarm the user). However this attack may not be economic because it is difficult to automate. Also because our system runs inside a virtual machine, he might have difficulties to automate his actions with macros because he cannot get window/cursor positioning information out of our virtualized system without image processing software. Thus we think that attacks may only be tried on a small volume basis and this would not yield big profits for the attacker.

SELinux limits allowed actions for a given program. Attacks may still be possible if a vulnerability inside that program is discovered and exploited, but they can only do what is granted to the program. Software that requires a broad range of SELinux permissions can do more damage if it gets hacked. For example the Adobe Reader needs permission to execute something on the stack, otherwise it will not start up. If now an exploit uses a buffer overflow vulnerability to execute code on the stack, it still can do that. Thus we have to decide what is more important: Use a more secure PDF reader or offer the user a PDF reader he already knows. SELinux nevertheless reduces the impact of such attacks because even if a hacker can execute malicious code on the stack, he cannot for example execute any program, as the SELinux domain Acrobat Reader is running in does not have that permission. But he can read or write data to the user's documents folder because Acrobat Reader has that permission to allow the user to save files there.

¹<http://www.colinneller.com/blog/GlobalWindowsHooksWithNET.aspx>



Chapter 7

Future work

The actual work described in this report consisted of specifying the requirements, designing and implementing a basic platform that allows the user to do the tasks described in the use cases and that fulfills the security requirements. We will now propose some ideas for future work on the platform mainly in the areas of usability and maintenance.

First of all, some sort of setup mechanism should be implemented according to the description in section 5.1. The goal is that the user needs only some few clicks until he can start using the system. The setup should run on all three major operating systems Windows, MacOS and Linux. It should create the shared directory and take care that it is created inside the users documents folder (selecting a system folder or the root device of the system hard disk would create a security risk as the browser in the secure platform is allowed to do any write and read operation on shared folders).

The GUI of the desktop environment could be enhanced with the possibility to choose a language and a keyboard layout (this perhaps needs a small graphical application to be programmed which will ask the user to choose his location and language at the first boot). Further we could create a logo and corporate design for SEBPS and integrate that logo as a boot screen and wallpaper. VMware has the Unity feature, which hides the desktop environment and lets application windows appear as if they are windows in the host system. It has to be tested if that feature works correctly and if it will enhance the usability.

Future versions of Debian will possibly replace¹ the classic init boot system with upstart² which will decrease boot time by parallelizing the start of tasks. If that feature becomes stable, we should implement it to enhance the usability as users are more likely to use the platform if it is booting faster.

¹<http://article.gmane.org/gmane.linux.debian.devel.announce/1395>

²<http://upstart.ubuntu.com>

Chapter 8

Conclusion

In this report, we described the design and implementation of a secure system for eCommerce and online banking. We raise the bar for malicious hackers to attack a user browsing the web with malware by adding several layers between his data and the browser. The browser is confined using mandatory access control. The user may download files and copy them to the documents folder of his host operating system. We deploy the platform on a removable media coupled with a free version of a virtual machine player. We hope that this approach is an economic solution to make eCommerce more secure without greatly reducing the usability or the dependency on special hardware or operating systems.

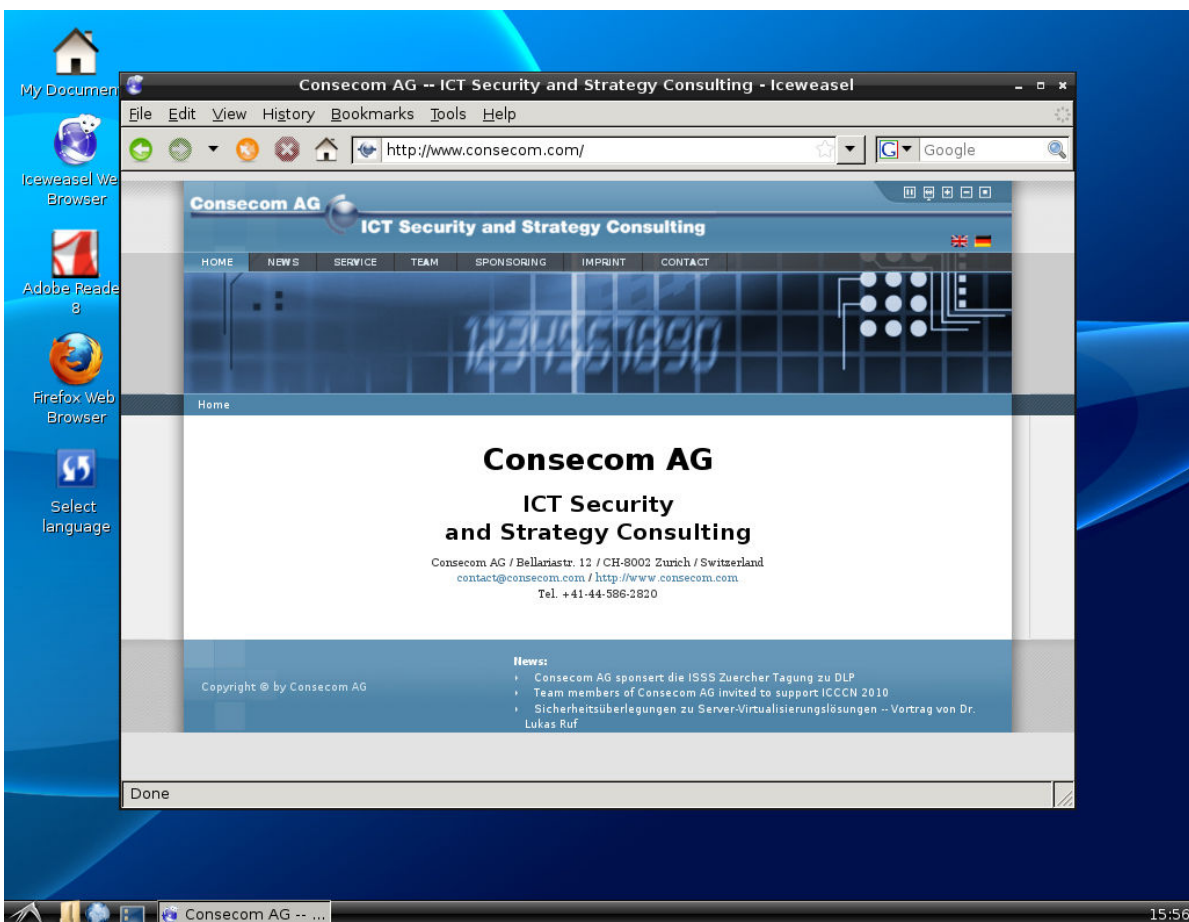


Figure 8.1: The result: Hardened browser in virtualized Linux

A horizontal line spans the width of the page, starting from the left edge. On the left side of this line, there is a small, three-dimensional sphere with a metallic, reflective surface, partially overlapping the line.

References

- [1] Lipp et al: *The Zurich Trusted Information Channel: An efficient Defence against Man-in-the-Middle and Malicious Software Attacks*, 2008.
- [2] National Security Agency: *The Inevitability of Failure* <http://csrc.nist.gov/nissc/1998/proceedings/paperF1.pdf>, 2008.
- [3] Al-Tawil & Al-Kaltham: *Evaluation and Testing of Internet Firewalls*, 1999
- [4] Trent Jaeger: *Reference Monitor*, Pennsylvania State University, <http://www.cse.psu.edu/~tjaeger/cse544-s10/papers/refmon.pdf>