The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

BUILDING A FRAMEWORK FOR INFORMATION FLOW AWARE WEB APPLICATIONS

A Thesis in

Computer Science and Engineering

by

Yogesh Raju Sreenivasan

© 2008 Yogesh Raju Sreenivasan

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2008

The thesis of Yogesh Raju Sreenivasan was reviewed and approved* by the following.

Trent Jaeger
Associate Professor of Computer Science and Engineering
Thesis Adviser

Patrick McDaniel
Associate Professor of Computer Science and Engineering

Mahmut Kandemir
Associate Professor of Computer Science and Engineering
Graduate Program Chair of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

Web has profoundly changed the way people share information and is being used extensively by organizations to share sensitive information. For such distributed web-based environments, multilevel classification of information has become an essential requirement.

Information flow policies that ensure multilevel classification of information are currently being enforced independently at different layers in a distributed system using technologies such as mandatory access control based operating systems and security-typed languages. In this thesis, we focus on designing a framework to unify the enforcement of information flow policies across different layers leveraging security-enhanced linux, xen virtual machine monitor and labeled IPsec mechanisms. We use selinux and labeled IPsec mechanisms to convey and enforce information flow policies at different layers and xen virtual machines to sandbox browser instances to isolate different web applications. Each virtual machine instance serves a single web-application at a pre-defined secrecy/integrity range. Our work focuses on bootstrapping the virtual machines with necessary policies and enforcing these policies during the run-time. In addition, we also propose an approach based on pre-loaded virtual machines to reduce the browser start-up latency.

Our analysis demonstrates that despite the overhead of virtualization, IPsec processing and policy enforcement, the proposed approach achieves throughput and latency that is reasonable for most web applications used for sharing sensitive information. Through the above-mentioned mechanisms, we build a distributed web-based system that can provide strict information flow guarantees.

Table of Contents

List of Tables vi

List of Figures vii

Acknowledgments viii

Chapter 1. Introduction 1

Chapter 2. Background 6
- 2.1 Information Flow Security 6
- 2.2 Leveraging Security Typed Langugages 7
- 2.3 MAC Operating Systems 7
  - 2.3.1 Network Mandatory Access Control 10
- 2.4 Integrating SELinux and JIF 11
- 2.5 Virtual Machine Monitor 12
  - 2.5.1 Virtual Machine Security 13

Chapter 3. Related Work 15
- 3.1 Information Flow Policies 15
- 3.2 Sandboxing Mechanism 17
- 3.3 Secure Browser Architectures 18
- 3.4 Distributed Access Control 20

Chapter 4. Architecture 22
  - 4.0.1 Configuration Phase 24
  - 4.0.2 Enforcement Phase 25
- 4.1 Web Application 25
- 4.2 VMLoad Service 28
- 4.3 Label Mapper 30
- 4.4 Policy Store 31
- 4.5 Compliance 32
  - 4.5.1 PIDSI Approach 32
  - 4.5.2 SIESTA 32

Chapter 5. Implementation 34
- 5.1 System Implementation 34
- 5.2 Enforcement 35
- 5.3 Supporting Services 38
  - 5.3.1 VMLoad Daemon 38
  - 5.3.2 Policy Store 39
- 5.4 Network Address Translation 41

5.4.1 Interaction with Labeled IPsec 42
5.5 Pre-Loaded VMs 43
5.5.1 Supporting Services 45

Chapter 6. System Evaluation 46
6.1 System Setup 46
6.2 Browser Startup Latency 47
6.3 Network Performance 50
6.3.1 Summary 52
6.4 Web Application Mock-up 53
6.4.1 System Design 53
6.4.2 Security Guarantees 55
6.4.3 Summary 56

Chapter 7. Conclusion and Future Work 57

Appendix A . Interaction Between Labeled IPsec and NAT in Linux Kernel 59
A.1 Details 59
A.2 Patch 60

References 62

List of Tables

6.1 Browser Startup Latency 47
6.2 Overhead due to Network Address Translation 52

List of Figures

1.1 High Level System Design 2

2.1 Xen Hypervisor Architecture 12

4.1 Virtual Machine System Architecture applied to the client-side of a web application. 23

4.2 Web Application 27

6.1 TCP Throughput values for measuring IPsec Processing overhead 51

Acknowledgments

First and Foremost I would like to thank my advisor, Dr. Trent Jaeger, for all the guidance, motivation and encouragement he has shown me during my time here at Penn State. I am deeply grateful to him for giving me the opportunity to pursue research in the field of system security and be part of the SIIS Laboratory. I also would like to thank my co-advisor, Dr. Patrick McDaniel for his invaluable support and advise during my thesis research.

I am very grateful to all my colleagues at the *Systems and Internet Infrastructure Security* lab for providing me with an excellent research environment and helping me understand my research area better. I am especially indebted to my colleagues, Sandra, Dave and Boniface who worked with me in this project. I would also like to thank Venkat Yekkirala for helping us with linux kernel related problems in a timely manner.

I would also like to express my deepest gratitude to my family. I would like to thank my parents for their constant support, encouragement and love throughout my life, without which I would not have had a chance to be at Penn State.

I would like to use this opportunity to acknowledge all my friends who have helped me stay focused on my goals and supported me through these difficult years.

Chapter 1

Introduction

The World Wide Web has evolved from being used primarily for accessing static documents over the network to a powerful platform for deploying complex applications. Web technologies such as wikis, blogs, and social-networking present a new model for organizing and sharing information among large number of users. These technologies, because of their ease-of-use and collaborative nature, are being adopted by various communities and used regularly in more sensitive environments for sharing information [42]. Web-based applications deployed in such environments require strong isolation and security guarantees.

Current browsers are not designed adequately for their new role and environment. The original model of usage where all the web pages are treated as a single application provides little isolation or security between distinct web applications hosted on the same browser. Efforts to retrofit web browsers to help improve security guarantees have not been very successful as these mechanisms are only as secure as the browsers they run within, which currently are designed for old usage model. Browser Architectures, such as Tahoma [9], isolate different web applications by sandboxing browser instances inside virtual machines and controls browser communication based on a network policy specified by the web service. Sandboxing only at the application level can be too coarse grained, as they don't provide any isolation among different objects within the web application. And also they rely on commercial off-the-shelf web browsers for processing their web applications and does not control information flows across web pages of

different secrecy and integrity classes within the same web application. As mentioned above, Tahoma architecture controls browser communication based on the policy downloaded from the server; these may be incompatible with the client system and may not fully satisfy the security requirements.

Recent advances, in the field of operating systems and programming languages, are bringing flexible mandatory access control, to commercial systems. But currently, information flow enforcement is performed independently at distinct layers of a system (e.g., application, operating system, or virtual machines), resulting in a restrictive and inflexible enforcement, and we lack a coherent way to unify these systems to satisfy the security guarantees of distributed environments. This thesis forms a part of the overall system whose goal is to unify the enforcement of information flow policies across virtual machine, operating system and application layers to achieve more effective and flexible enforcement. We focus mainly on browser-based web applications but could be extended to other distributed applications as well.

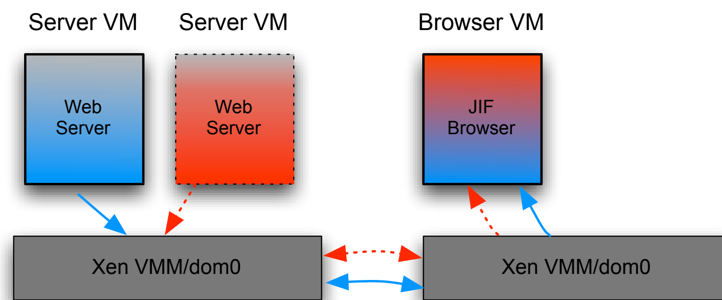

Fig. 1.1. High Level System Design

Figure 1.1 gives a high level overview of the overall system. First, an information flow-aware application determines the security label for output communications. Information

flow-aware browser is developed using a security-typed language. JIF is an example of one such language, which we use to build the browser that can guarantee information flow enforcement. Secondly, the guest operating system in the virtual machine authorizes the application and creates a secure channel using labeled IPsec, for transmitting the data. The proposed system runs SELinux operating system in the guest VMs. And lastly, the virtual machine monitor authorizes VM's request and extends the IPsec tunnel to convey the security label. In our prototype, guest VMs are hosted on a Xen Virtual Machine Monitor.

In this thesis, we focus of four key issues,

- *Flexible sandboxing of Web browsers*. Although SELinux and JIF guarantee enforcement of information flow policies, the level of assurance provided by these certified products is not suitable for being deployed in environments that require strict security guarantees. In our proposed design, each browser instance is run inside a virtual machine and serves a distinct web application hence providing the required isolation.

- *Comprehensive Policy Construction*. We construct the SELinux and IPsec policies for the virtual machine and the management domain (or dom0), required to support the web application. We also ensure that the installed application policy is compliant with the overall system policy during the virtual machine boot-up.

- *Secure/Authorized Browser Communication*. We establish labeled IPsec tunnels to convey the data and its security label to the remote end. And we authorize the setup of these tunnels using the network policies defined in the virtual machine and the dom0.

- *Construction of distinct Web Applications*. In our design, each browser virtual machine serves a single web application at a pre-defined secrecy/integrity range. We define a web application in a fine-grained manner based on the security label of the URLs.

This work focuses on building a framework for bootstrapping virtual machines used for sandboxing browser instances. SELinux and JIF based browser instances enforce the required information flow policies at the operating system layer and at the application layer inside the virtual machine. We have implemented services in the management domain of Xen hypervisor (dom0) which, in addition to loading a new browser VM, are also responsible for constructing and installing the SELinux and IPsec policies required to support the web application. These services also ensure that the generated application policy is compliant with the overall system policy. We have also implemented other supporting services inside the virtual machines to maintain/update the security policies.

We use labeled IPsec tunnels between the virtual machine and its dom0 and also between two dom0s to convey the security label of the browser communications during run-time and use SELinux network modules inside the virtual machines and dom0 to authorize the creation of these tunnels. In order to generate the policy required to establish the IPsec tunnel between dom0s we need to retrieve the address of the destination dom0. In this thesis, we also propose an approach based on Network Address Translation to establish the IPsec tunnel with destination dom0, given the ip address of the web server VM. In our framework, we propose mechanisms to define and construct distinct web applications based on the security label of the objects served.

The contributions of this thesis are as follows:

- We present the design and implementation of a new browser architecture that unifies the enforcement of information flow policies at all layers.

- We optimize the prototype implementation using Pre-loaded VMs to reduce the browser startup latency. In this approach, we maintain a pool of pre-loaded VMs and the service running in the dom0 selects the VM from the pool, installs the necessary policies and forwards the web page request.

- We also measure the system performance to demonstrate that despite the overhead of virtualization and IPsec processing, the proposed approach can achieve throughput and latency that is reasonable for most web applications.

- We demonstrate the working of our implementation by applying it to an analyst-driven web application model commonly used in sensitive environments for sharing information.

The rest of this thesis is organized as follows. Chapter 2 provides background motivation for the problem and Chapter 3 presents the related work. Chapter 4 defines system's high-level architecture and description of various components. Chapter 5 describes the implementation, while Chapter 6 presents an experimental evaluation of our prototype implementation. Finally, Chapter 7 presents our conclusion.

Chapter 2

Background

In this section, we briefly describe different technologies and previous research work in the are in our system design. Our description of their architecture is limited to the aspects that area relevant to our discussion.

2.1 Information Flow Security

Information flow security at different layers has long been a focus of the security community. An information flow policy defines the different classes of information that can exist in a system and how data flows between these classes. Information flow model for secrecy, called as multi- level security (MLS), was formalized by Bell and La-Padula [6]. In the MLS model, the subjects and objects are given a *sensitivity level*, or *security clearance*, and the MLS policies restrict how information flows between them. For example, consider an application with four sensitivity levels: Unclassified (UC), Confidential (CO), Secret (S) and Top Secret (TS). In this case, TS dominates S.

MLS model defines information flow based on two properties: the simple security property or ”no read up” meaning that subject’s sensitivity level should be higher than the object’s sensitivity level for read access, and star (*) property or ”no write down” meaning the object’s sensitivity level should be higher than the subject’s sensitivity level for write access. To provide

more granularity, MLS levels are further divided into categories. A subject must have a superset of the object's categories to dominate the object.

This thesis forms a part of an overall system whose goal is to unify the information flow guarantees provided at different layers of the system to achieve stronger information flow guarantees. In this chapter we will examine the information flow guarantees achieved at different layers, starting at the application layer and also the technologies used to extend and unify these guarantees.

2.2 Leveraging Security Typed Langugages

Recent advances in the field of security-typed languages allow us to construct applications that enforce information flow policies with provable security guarantees. Security-typed languages extend the type system of the language to include security label. The language compiler ensures that applications written using the language enforce information flows specified by the language's security policy.

JIF [27] extends Java Programming language with security typing and provides essential features specific to information flow enforcement. In our proposed design, we use the information-flow aware browser developed using JIF programming language. However, in this thesis, we will not be discussing about the implementation of the browser.

2.3 MAC Operating Systems

Below the application layer is the operating system layer. The concept of multi-level security (or mandatory access control) has been around since 1960s, and the first operating system to include MLS capabilities is Multics [30]. Multics was a large, long-term operating system

project where many of the security fundamentals such as reference monitor, protection systems, protection domains, and multilevel security policies were included. Multics protection system consists of three different models,

1. **Access Control List:** Each object is associated with its own *access control list* (ACL) which specifies who has what access on the object.

2. **Rings and Brackets:** Multics also limits access based on the protection ring. Each *segment* is associated with a *ring bracket*, which specifies the permission the process has over the segment.

3. **Multi-Level Security:** Multics also stores the secrecy level of each segment and each process in a list and authorizes the request based on the simple and * security properties.

In multics, the request to access a segment is granted only if all the three policies, discussed above, authorize the request.

Mandatory Access Control is becoming available in commercial operating systems recently. Operating systems with MAC security includes Trusted Solaris [26], TrustedBSD [11] and SEDarwin [40]. SELinux [29] is one of the most popular and active MAC frameworks for Linux. The basis for SELinux can be found in the Flask (FLux Advanced Security Kernel) Architecture [38]. The major goals of FLASK architecture, inherited by the SELinux framework, are to provide fine-grained access control and clean separation of policy specification and enforcement mechanisms.

NSA's SELinux has been integrated into Linux through the Linux Security Module Framework [37]. LSM provides a general-purpose access control framework for Linux kernel that

enables implementation of Mandatory Access Controls in individual systems. The LSM Framework adds security fields to kernel data structures and inserts hooks at special points in the kernel code to perform a module-specific access-control check. LSM is available in mainstream Linux kernel version 2.6.

SELinux architecture has provision to enforce three security models: the Type Enforcement (TE) model, Role-Based Access Control (RBAC) model and Multi Level Security (MLS model. In a SELinux system, every element is associated with a *class* (file, dir, process, socket, etc.) and each *class* is associated with a list of security sensitive operations or *modes* of access (read, write, open, polmatch, etc.). In addition to these classifications, the TE model further associates all subjects and objects with a security type and manages an access control matrix specifying which subject types can access which object types.

The MLS model supported by SELinux is largely orthogonal to the TE and RBAC models and there is no interaction between them. It associates an MLS level (s0-s15) and an MLS category (c0-c255) to all subjects and objects in the system. On every security sensitive operation, a set of MLS constrains defined by the SELinux policy is checked based on the MLS level of the subject and the object as well as the object class and mode of access. SELinux seeks to implement the MLS policy in accordance with the definitions by Bell-LaPadula model.

Red Hat Enterprise Linux (RHEL) version 4 (and later versions) comes with SELinux-enabled kernel. The RHEL 5 version contains additional MLS enhancements and currently in evaluation under LSPP/EAL4. The Evaluation Assurance Level (EAL1 through EAL7) is a numerical grade assigned following the completion of a Common Criteria security evaluation. Due to functional requirements these emerging systems are highly unlikely to have high level of formal assurance.

We use SELinux as the guest operating system inside the virtual machine to enforce information flow guarantees. Unlike traditional operating system environments, each virtual machine, and in-turn each guest operating system, is devoted to a single application, the JIF web browser. We currently use the MLS labels for secrecy and envision using TE labels for integrity.

2.3.1 Network Mandatory Access Control

Mandatory Access Controls for network communications offered by the original LSM framework are very limited and focused on a single machine.

One of the earliest solutions for MAC control for network communications is the IPSO (IP Security Options) [13]. In IPSO, the MLS sensitivity levels and categories are encoded in the IP packet header and decoded and authorized by the receiving system. Netlabel is a labeled networking implementation for the Linux kernel that implements the IPSO standard partially. The Netlabel approach has some inherent disadvantages; it can encode only the MLS levels and not the other parts of the SELinux label and also it adds significant packet processing overhead especially when the IP packets are fragmented.

Secmark is another implementation of labeled networking for the Linux kernel that uses netfilter/iptables rules to select and label network packets and authorizes them using SELinux policies. This approach has many disadvantages and also very coarse-grained.

We will focus mainly on the recent extension to LSM framework to support fine-grained control of Linux network communications by Labeled IPsec tunnels, [16] which was incorporated into the Linux Kernel version 2.6.19. This approach enables MAC controls by labeling the IPsec Security Associations (SAs) and associating IP packets with the corresponding SAs. The idea is that IPsec can be used to negotiate security labels in the same manner as it does for the

SAs and SELinux can authorize which processes can access these IPsec SAs. The Labeled IPsec mechanism has been further extended to support MLS levels supported by the SELinux system. This mechanism in addition to conveying the labels also inherently leverages the IPsec security mechanisms.

In our system design, we use Labeled IPsec mechanism to label all network communications and authorize the browser sockets. We use three orthogonal Labeled IPsec tunnels, described in later sections, to convey the security label of the browser socket.

2.4 Integrating SELinux and JIF

Boniface *et al.* [15] propose an approach to integrate information flow enforcement of applications written in a security-typed language (*jif*) with SELinux. By integrating the enforcement guarantees of security-typed languages and operating systems, SELinux can build a comprehensive guarantee of system security. Authors propose an architecture and implementation of an operating system service, called SIESTA [14] that integrate security-typed language with operating system MAC services.

SIESTA's main goal is to guarantee that a given security-typed application is executed on a given operating system only if the system and application security policies and requirements are consistent. SIESTA ensures that the current system state fulfills the application security requirements and application security policies are compliant with the system security policies.

In our proposed design, we use the *siesta* service to ensure compliance between operating system and application layers. SIESTA service is loaded in each of the guest operating systems, which in-turn loads the JIF browser. Before loading the JIF browser, *siesta* checks for compliance between the SELinux and JIF policies.

2.5 Virtual Machine Monitor

Virtualization technology is quickly gaining popularity. In this technology, a software layer called a virtual machine monitor creates multiple virtual machines over one physical machine by multiplexing multiple virtual resources onto a single physical resource. Recent advances in the field has allowed us to aggregate multiple virtual machines, each running its own operating system, on to a single physical machine. Xen [5] is an x86 virtual machine monitor or hypervisor, which allows multiple commodity operating systems to share conventional hardware in a safe manner. Xen uses an approach called *para-virtualization* where the hypervisor presents a virtual machine abstraction that is similar but not identical to the actual physical hardware. This approach promises improved performance and strong resource isolation, although it does require modifications to the guest OS.

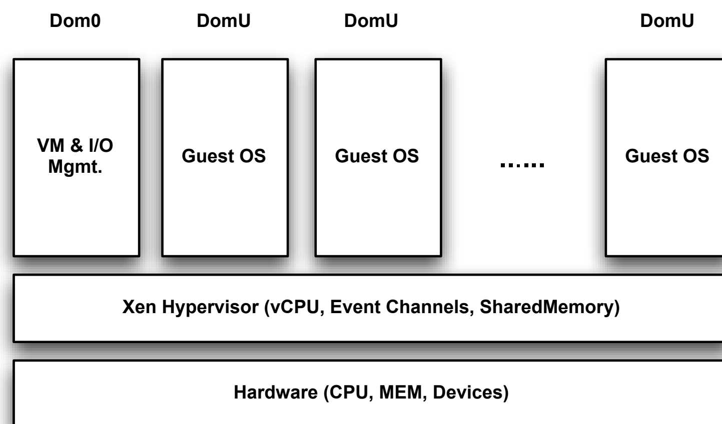

Fig. 2.1. Xen Hypervisor Architecture

Virtual Machines, knows as *domain* (or *guests*) in Xen, are built on top of the Xen hypervisor. A special VM, called dom0 (domain zero) is created first during the system boot-up.

This domain is responsible for managing (create, destroy, migrate) other VMs and also controls the assignment of I/O devices (virtual disks, virtual ethernet cards) to VMs. VMs started by Dom0 are called DomUs (user or guest domains) and they can run any para-virtualized operating system. The guest OSs running on Xen are modified minimally, for example some privileged hardware instruction are replaced by calls to the hypervisor.

2.5.1 Virtual Machine Security

In Xen, each operating system and its applications run in a single VM that is typically isolated from the other VMs, except through network communications or storage. However, MAC enforcements for VM systems have emerged. MAC enforcement on VM systems control inter-VM communications and it is stronger than traditional VM isolation. The Xen Mandatory access control framework is an implementation of the sHype Hypervisor Security Architecture [33]. sHype is a hypervisor *reference monitor* that enforces comprehensive, mandatory access control policies on inter-VM operations. To control sharing between domains, sHype mediates all inter-VM communication (shared memory, event channels) as well as the access of domains to resources such as storage disks and authorizes based on the sHype security policy.

The current sHype model implements two formal security policies and both policies work on their own set of types (CW- or TE-types).

- **Chinese Wall Policy**: It defines conflict sets and ensures that VMs that are assigned CW-types in the same conflict set never run on the same physical machine. This is especially useful to mitigate covert channels.

- **Type Enforcement**: It assigned TE types to VMs and resources and ensures that VMs only share virtual resources if they have a TE type in common.

Our goal is to leverage the Xen/sHype architecture to enforce information flow guarantees wherever possible and the guarantee strong isolation among different web applications. In our proposed design, the JIF browsers and web servers are run inside the Xen virtual machine and Labeled IPsec tunnels are used to convey security labels between them. Each browser VM is associated with a single web application, serving pages within the pre-defined secrecy/integrity range. The main aim of this thesis is to extend and enforce the information flows originating from the JIF browser VM. For example, if the JIF browser sends secret and public requests, the virtual machine monitor layer must be able to differentiate these requests and deliver the secret requests only to the authorized VMs.

Chapter 3

Related Work

In this chapter, we summarize other related work, in particular, work that investigates the use of information flow policies and distributed access control mechanisms in a web-based environment. This chapter also discusses about previous works on different sandboxing mechanisms.

3.1 Information Flow Policies

The need for information security for web-based applications can be attributed to several factors, including the availability of high volumes of sensitive information maintained by corporations and government agencies and also the way in which web browsers have evolved from a simple document-rendering engine to an environment for complex, dynamic applications. Modeling security policies in web-based application is a well-studied topic [10].

Several models, including discretionary access control (DAC), mandatory access control (MAC) and role-based access control (RBAC) have been proposed to address the information security requirements of the web application [20]. Unlike other models, MAC can be used to model information flow control policies such as multi-level security and integrity policies and can also provide high level of assurance. For Web-based applications, Multilevel classification of information could be an important requirement; it can be used to classify users and the type of information being accessed.

Niemeyer [28] presents an analysis of the use of HTTP and other web technologies in multilevel secure environments. The analysis focuses on two multilevel secure systems referred to as High-to-Low (HTL), with an MLS database server accessed by a web browser software and Low-to-High, with MLS infrastructure supporting web browsing, where the security classification of information on one side of the system differs from that of the other side. Authors argue that in a well configured environment using HTTP in MLS mode does not create additional risk, but at the same time generic HTTP vulnerabilities are not ameliorated by they use of traditional technologies. In this thesis, we propose a framework for the use of web-based applications in MLS environments leveraging SELinux MAC policies and Xen hypervisor technologies.

Recently, there have been efforts to implement multi-level security for hypervisors or virtual machine monitors. Karger [22] discusses about the requirements of MLS systems and its implications on the design of multi-level secure hypervisors. The hypervisor is being used to separate multiple instances of untrusted/trusted operating systems, running at different security levels. In NetTop [25], a Virtual Machine Monitor application, VMWare, is installed on a SELinux host, with additional features to guarantee system integrity. The Virtual Machines and the virtual disks are assigned different SELinux types and the access between them is controlled by SELinux MAC policies.

Flexible mandatory access control (MAC) enforcement is now available for virtual machine systems. For example, the sHype MAC system for Xen VMM is part of the mainline Xen distribution. The semantics of sHype allows us to express different policy models, including the Chinese Wall Policy, Multi-Level security and Caernarvon Policy, a less restrictive form of traditional Bell-LaPadula Policy model [18]. The sHype system associates security labels to the entire Virtual Machine and the security policy is enforced based on the VM security label.

In our proposed work, we follow a more fine-grained approach wherein the inter-VM network communications are authorized also based on the security label of the process inside the VM.

3.2 Sandboxing Mechanism

We use the sandboxing approach in our proposed framework to mitigate the information leakage. Different mechanisms for confining the activities within sandboxes and their usage models have been explored previously [2] [4].

One of the earliest OS-based sandboxing techniques is TRON [7], which enforces capabilities by using system call wrappers compiled in to the system kernel. TRON system provides extra protection layered on top of existing operating system protection; an operation is allowed only if the process has both TRON and Unix permissions. Systems such as Janus [41] and BlueBox [8] implement user-level system call interception mechanism. These systems provide a secure environment for helper applications by restricting their use of system calls and require modifications to the existing kernels.

Isolation provided by these process-based sandboxing mechanisms are very limited; for example a non-sandboxed process in the operating system can access sandboxed processes and files, and is very much dependent on the guarantees provided by the operating system. Jaeger *et al.* [17] presented an access control model for a distributed system to flexibly control the downloaded content. It provides a sandboxed environment for the downloaded content by restricting the access rights to the subset of the downloading principle's access rights. In our design, we use virtual machines to isolate browser intanaces from each other.

GreenBorder [39] provides a sandboxed environment, called virtual sessions, for Internet Explorer and outlook applications within Windows. It insulates Internet Explorer from

the rest of the system by redirecting the modifications/ changes to the virtualized copies of the system resources. The virtual sandboxes are flushed at the end of each session. This approach prevents browser vulnerabilities from spreading to other system applications/resources but does not guarantee isolation between web applications accessed from same browser instance. Our approach, in addition to confining the information leakage to the virtual machine instance also provides the flexibility to completely isolate web applications.

The Collective project [35] encapsulates collections of applications like TiVos and NetApp filers within VMWare virtual machines and transfers them as Virtual Appliances over the network to be run on VMWare hosts. It focuses mainly on ease-of-use and maintenance of these applications and to avoid security breaches by allowing the makers to do the periodic updates. Our approach is similar in that browser instances are encapsulated within the VMs but focuses more on isolating web applications and enforcing information flow policies.

3.3 Secure Browser Architectures

In this thesis, we define a framework for secure web browsing. In addition to the projects we discussed in the previous sections, our work is more closely related to some previous works on browser-based security.

Tahoma [9] defines a new browser platform called the browser operating system (BOS), a trusted layer on top of which the web browsers execute. The BOS runs the client-side component of each Web Application in its own virtual machine. It also enables the web publishers to specify the list of URLs and resources the web browsers are allowed to access as part of the Web Application. Tahoma uses Xen VMM to implement virtual machine sandboxes, each containing a browser instance. We follow a very similar approach to Tahoma in using Xen virutal machines

to sandbox browser instances and limiting its access based on the scope of the web application. While Tahoma uses the sandboxing approach to confine the malicious activies, we mainly use it to provide additional guarantees on top of JIF, SELinux and labeled IPsec mechanisms. Our approach focuses more on leveraging VMs and other mechanisms to extend the information flow guarantees.

The OP web browser [12] introduces a new architecture for secure web browsing. In OP browser architecture, the browser is split into five main components: the web page subsystem, a user-interface component, a network component, a storage component, and a browser kernel. Each of these subsystems run within separate OS-level processes and the browser kernel manages interactions between these components. SELinux access control mechanism is used to limit subsystem interactions with the operating system. Plugin access control in OP browser architecture is done by the same security mechanisms and policies used for the browser. Each instance of a plugin is treated as a separate process and assigned a label by the browser kernel. Our architecture shares some of the design principles as the OP web browser but there are some key differences. Our approach uses VMM to isolate web applications while the OP browser depend OS level process isolation. Both approaches use SELinux to enforce access control decision at operating system level. Our approach is complementary to the OP browser architecture. One can imagine using both the approaches together to provide stronger security guarantees.

3.4 Distributed Access Control

Access control in a distributed environment such as web is quite a challenging issue and has been studied in the community for a while. Different security models have been proposed and studied for the problem of secure information access in a distributed web-based systems [34]. DCE [19] is an integrated set of services to support the development and deployment of distributed client/server applications in a heterogeneous networking environment. The DCE security service provides three elements of security in a distributed computing environment: authentication, data protection and authorization.

DCE authentication services are based on Kerberos authentication model. Kerberos [24] is a distributed authentication protocol that allows the entities communicating over a non-secure network to prove their identity to one another in a secure manner. Kerberos provides key-distribution and ticket- granting services. Schimpf [36] discusses the use of security services and infrastructure of the DCE model to secure Web accesses.

Sesame [21] (A Secure European System in A Multivendor Environment) is a role based access control system for distributed, heterogeneous computer environments. It provides single sign-on authentication mechanism along with cryptographic protection of data. In SESAME, to access the distributed system, the user first authenticates to the Authentication Server to get a protected taken used to prove his identity and this token also helps user to obtain a guaranteed set of access rights contained in what is called Privilege Attribute Certificate.

SAM [3] is a tool for enterprise-wide security management that implements many of the RBAC concepts. SAM consolidates the administration of information security systems distributed across the enterprise and it provides an administration layer through which the security

administrators can easily administer these systems. It is designed as a link between the high-level company security policy and low-level security systems and provides powerful RBAC features to administer these systems.

The methods discussed above are based on Discretionary Access Control or Role Based Access Control models for restricting/granting access to resources in a distributed environment. In this thesis, we propose a Mandatory Access Control model based on Information flow control policies for the distributed environment like Web. Both the models are complementary and can be used together to ensure stronger security guarantees.

Chapter 4

Architecture

In our proposed framework, distributed application components, such as browsers and web servers are hosted inside guest virtual machines running a mandatory access control operating system, such as SELinux. We assume that the applications are also written using mechanisms that enable them to enforce a mandatory access control policy, such as security-typed languages or user-level reference monitors.

The system design primarily consists of two distinct phases:

- **Configuration Phase**: This phase involves bootstrapping a new VM with appropriate security policies to support the web application and ensuring compliance between VM and application policies.

- **Policy Enforcement Phase**: This phase is responsible for authorizing requests from the VMs and extending the security guarantees by conveying the security label of the request to the operating system running in the web server.

The architecture consists of three components: (1) a privileged, *loader VM*, which performs configuration and enforces VMM level policies; (2) an *application authority*, which defines application-specific policies; and (3) a *web application VM (or Browser VM)* in which the sandboxed application runs. The idea is that the application authority defines the distributed application requirements, which the loader VM converts into policies which it enforces and configures into the web application VM, where necessary. The web application VM is sandboxed

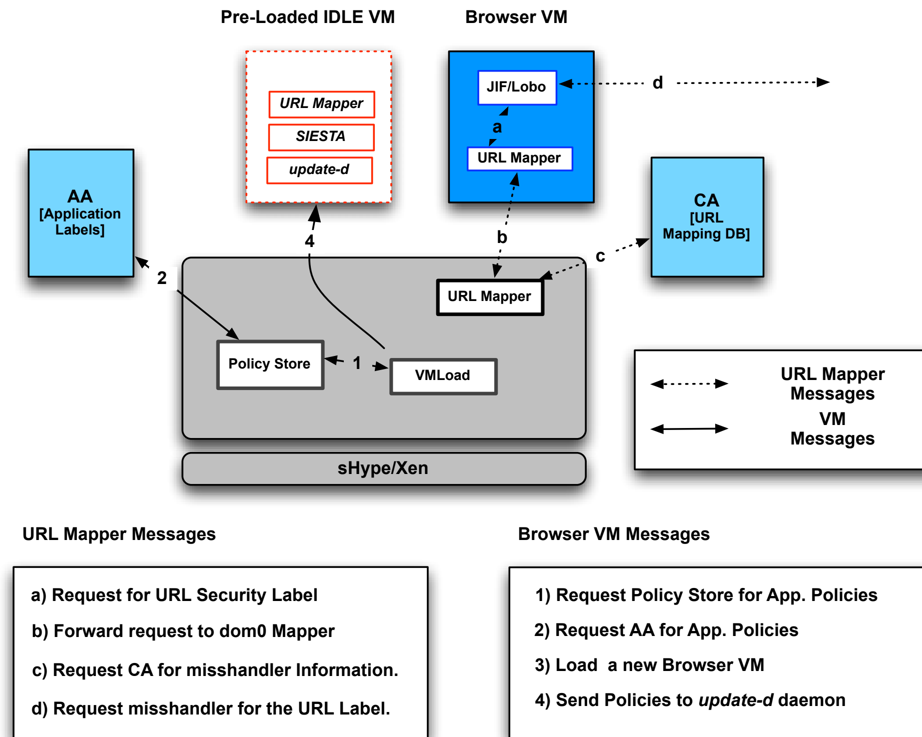

Fig. 4.1. Virtual Machine System Architecture applied to the client-side of a web application.

by the loader VM which also authorizes its inter-VM communications, but the web application VM may also be entrusted with some enforcement policies through its configuration.

The loader VM defines three new services for configuring and running sandboxed VMs: (1) a *VMLoad service*; (2) a *policy store*; and (3) a *label mapper*. The VMLoad service authorizes requests to load new web application VMs, obtains the web application policies from the policy store, configures these VMs, initiates their execution, and ensures that the VMM enforces the inter-VM communications according to these policies. The policy store caches web application policies and retrieves web application policies from application authorities on a cache miss. The label mapper converts object identifiers for distributed objects (e.g., URLs) into security labels to ensure correct enforcement.

The web application VMs include two additional, new services: (1) an *update daemon*, which update application VM policies from the loader VM and (2) a *SIESTA service* [14], which escalates the privileges of applications only if they can verify compliance with the enforcement of application VM security goals. The update daemon accepts dynamic policy updates, from the loader VM only. We discuss the variety of policies below. The SIESTA service is used to load applications that enforce mandatory access control (MAC) policies. SIESTA verifies that the application's MAC policy only allows information flows permitted by the OS MAC policy (e.g., SELinux). Only applications that meet this requirement may be entrusted with permissions that would violate system security goals, if not enforced correctly by the application. Web application VMs also include label mappers that cache mappings specific to that application.

4.0.1 Configuration Phase

As mentioned above, this phase is responsible for loading a new VM to serve the web application. When a web applicaiton VM encounters a web page/object that it is not authorized to serve, it sends a request to the *loader VM*

1. The *VMLoad service*, with the help of the *label mapper*, determines the web application corresponding to the requested web page/object.
2. Downloads the necessary policies from the *policy store*, described in-detail in Section 4.4.
3. Installs the downloaded policies in the *loader VM* and also in the new web application VM.
4. Loads the new *web application VM* to serve the web page request.

The individual components are described in-detail in the below sections.

4.0.2 Enforcement Phase

This phase, like described above, is mainly responsible for conveying the security label of the browser communication to the other end of the system and enforcing the security policies. Each web application VM serves a single web-application, described in Section 4.1 and is confined to a pre-defined secrecy/integrity range.

During the run-time, when the user wants to visit a web page,

1. The Mandatory Access Control based browser retrieves the security label of the web page from the *label mapper* and assigns the label to the channel used for the browser communications.

2. Mandatory Access Control policy installed in the guest operating system authorizes the communication based the socket's security label and establishes a security channel, to convey the label, between the web application VM and the *loader VM*.

3. The *loader VM* retrieves the security label and authorizes the inter-VM communication based on the system policy.

4. It also extends the secure channel to convey the security label to the web server VM.

4.1 Web Application

The figure 4.2 shows a high-level view of different components of a web application. In the proposed architecture, each browser instance executes inside a virtual machine, and each browser VM serves a *single*, well-defined Web Application, e.g., a banking application, at a pre-defined secrecy/integriy range. For example, there can be a browser VM serving *TOPSECRET*

pages of the web application and another browser VM serving the rest of the pages. Web applications are defined based on the security label of the objects. For example, we can have a web application serving objects with label *app1_t* and another web application serving objects with labels *app2_t*, and *app3_t*. This approach gives us more flexibility in defining a web application. For instance, a single web server can serve objects belonging to multiple web applications and also a single web application can span across multiple web servers. The security labels and secrecy/integrity levels supported by the web servers must be interpreted unambiguously throughout the web application. For example, we can have a banking application span across multiple web server VMs and each of these web servers interpret the security labels in a unfirom manner. The Browser VMs are connected to these web server VMs, distributed across multiple physical machines using secure tunnels and controlled using security policies defined inside the *Loader VM*.

Each organization is assumed to have a central authority called the Application Authority that manages all these different web applications. Application Authority maintains information about different web servers serving the web application and the policies needed to support these web applications. And also application authority is responsible for mapping security labels to web applications.

The VM environment provided for the browser instances has several advantages. First, Web applications are isolated from one another and from other system components and thus confining the untrusted information flows. Second, isolating the web applications using VMs allows us to interpret the security labels unambiguously. For example, a URL labeled as 'secret' in a medical application need not necessarily be as important as a 'secret' page in a sensitive government application. Lastly, running browser instances inside a VM gives us more flexibility

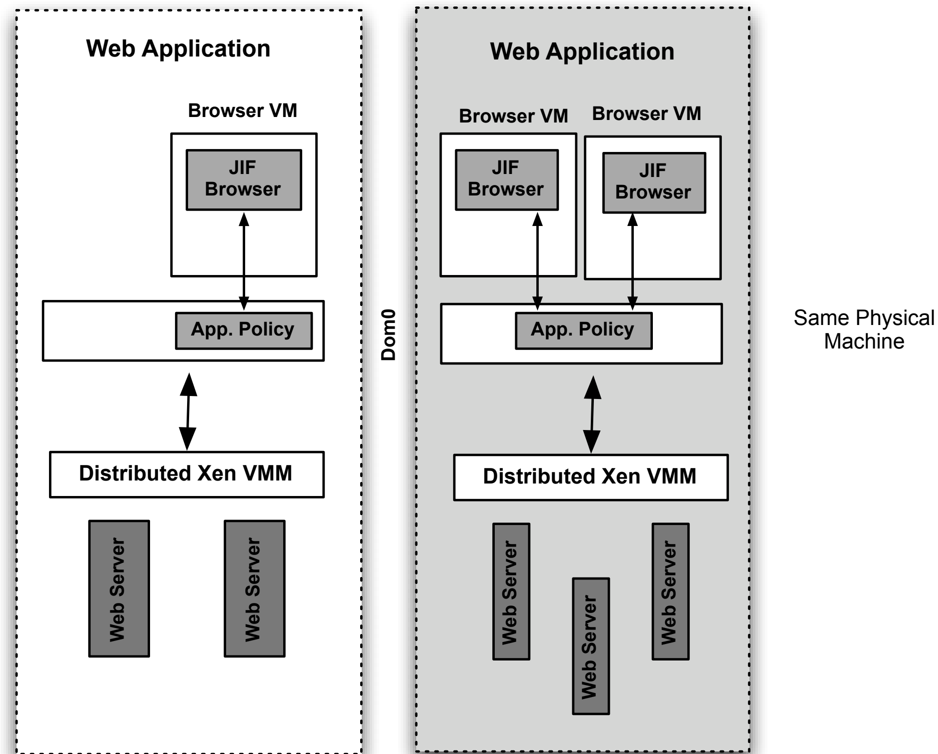

Fig. 4.2. Web Application

in terms of migrating the browser instances, to improve the native system performance and in customizing the browsers to fully utilize the VM features.

Each web application is identified by a unique identifier, which is encoded into the security labels of all the URLs served by the web application. The application id can be encoded into the security labels by using special characters (separators) such as underscore '_' or a dot '.'. Such an approach also has additional advantages. Firstly, policy modules for different web applications do not conflict with each other, as the types will be different for different applications. Secondly, also such an approach can have additional semantics as it represents a hierarchical relationship.

4.2 VMLoad Service

The VMLoad is a trusted network daemon running in the Loader VM and it is the central component of the architecture responsible for bootstrapping a browser VM to serve the web application and ensuring compliance (during VM bootup) between web application policy and the system policy. The protocol for the VMLoad daemon is given below.

1. If the browser VM is unable to serve a web page request it sends a request to the VMLoad service running in the *Loader VM*. This situation could arise mainly because of two different reasons.

    - If the web page requested by the user does not belong to the web application currently served by the VM.
    - If the requested web object/page does not fall within the secrecy/ integrity range that the browser VM (and inturn the browser).

2. The VMLoad service authorizes the request based on the security policy enforced at the *Loader VM*. For example, we need to check whether the calling VM can serve the application, what transition should take place, whether that domain transition is authorized for the calling VM, and whether the calling VM can create a new Browser VM at all. The enforced policy should also have provisions to account for resource usages in future.

3. If the request is authorized, the service needs to locate the browser VM image. Currently the VM images are retrieved from a pre-defined location in the local filesystem but in future, this module could be extended to retrieve images from a central repository using a different naming convention.

4. Next, the VMLoad daemon queries the local *label Mapper* to retrieve the security label corresponding to the requested URL and requests the Policy Store to map the security label to the web application-id. The applicaiton-id and the secrecy/integrity range of the URL determines the Browser VM's secrecy/integrity range and web application.

5. The VMload daemon downloads the necessary policy modules from the Policy Store to support the web application. The VMLoad daemon requests,

   - MAC policy for Browser VM and *Loader VM* to support the web application.
   - Network policies for Browser VM and *Loader VM* to protect the browser communication and to convey the packet labels.
   - Certificates required to setup secure tunnels with the destination VM.

6. The VMLoad daemon updates the Browser VM image and *Loader VM* with respective policy modules. Downloaded MAC policy modules are compiled and inserted into the *Loader VM* and browser VM to support new application specific types and information flows. Network policies for setting up orthogonal secure tunnels are updated in the *Loader VM* and browser VM. In addition to these policies, it also updates SIESTA specific configurations to specify the secrecy/integrity range of the browser instance.

7. Finally, the VMload daemon requests the *Xend* with appropriate configuration options to load the new browser VM.

4.3 Label Mapper

Label Mapper is a trusted, distributed service that maps the URLs to security labels. In our proposed design, each of the VMs including the *Loader VM* has a local copy of the label mapper, serving requests for both local and remote components. Protocol used by the label mapper to retrieve security labels for the requested URLs,

1. The browser requests the local label mapper to get the security label for the web page it is trying to access.

2. The mapper queries its local cache for the security label and returns the label on a cache-hit.

3. On a cache-miss, it forwards the request to the mapper running in its Loader VM, which in-turn queries its local cache and returns the label if found.

4. If label information is not found, the mapper in the Loader VM queries the miss-handler DB [1] and returns the miss handler information.

   In the course of providing the miss-handler information, the Loader VM contacts the Coalition Authority [2] if the miss-handler information is not presents in its local database.

5. The mapper then forwards the request to the miss-handler to retrieve the security label.

6. By default, if mapper is not able to retrieve the label, it will return low secrecy/low integrity label corresponding to a default application.

[1] Miss Hander DB contains information about the label Mapper that will be able to service the mapping request. By default it will point to the copy of the label Mapper running in the web server VM

[2] Coalition Authority is a central authority that maintains information about the VMs and applications in the organization.

4.4 Policy Store

Policy Store acts as the policy engine responsible for generating application specific policies, it generates both MAC and network policy modules. We follow a more modular approach by separating the policy engine from the vmload service, which allows the vmload to do the bare minimum of servicing the load request. The Policy Store will be running in the Loader VM and services request for

1. Mapping the URL security label to the application-id.
2. Application policy modules (MAC policy) for the Browser VM.
3. Application policy modules for the Loader VM.
4. Network policies for the Browser VM to establish secure tunnels.
5. Network policies for the *Loader VM*.
6. Certificates (signed by the Certificate Authority) required to setup secure tunnels with the destination VM.

The vmload service requests only certain set of policies from the Policy Store depending upon the type of request. For example, vmload service requests only network policies in cases where it does not need to load a new browser VM since the application modules will already be present in the Loader VM.

In our design, Application Authority is a central authority that maintains information, such as MAC policy modules, and network certificates, for all the web applications in the organization. It also acts as a certificate authority for the organization and it is responsible for

creating and signing the network certificates. Policy Store can be considered as a local copy of the Application Authority servicing requests from the *vmload* daemon. The Policy Store in- turn queries the Application Authority if it does not find the required information in the local cache.

4.5 Compliance

The system must verify compliance of downloaded web application policy with system-level information flows. The compliance must be verified at two different levels, between the browser application policy and the guest operating system policy and between the browser VM policy and the overall system policy.

4.5.1 PIDSI Approach

Usually system and program policies cannot be directly compared and manually mapped from one policy to the other one. But, it is observed that usually trusted program objects are of higher integrity than the system objects, the integrity of the trusted programs must dominate the integrity of the system data. The PIDSI (Program Integrity Dominates System Integrity) [32] approach developed by Sandra *et al.* assigns trusted program objects a higher integrity level than the system data objects. This assignment results in a simple program policy that enables automated compliance verification of trusted programs.

4.5.2 SIESTA

SIESTA [14] is a service that checks compliance between policies of applications, written using security-typed languages like JIF, and the local system policy based on the allowed

information flows. At a high level, an application policy is compliant with a system policy if the set of information flows allowed by the application is also allowed by the system.

After downloading the application polices from the Policy Store, the vmload daemon requests the SIESTA service in the Loader VM to verify compliance between the new module and the current browser VM policy. The siesta service uses the PIDSI approach, described above, to ensure the compliance.

SIESTA is also started as one of services during the Browser VM bootup. While loading the browser VM, the vmload service assigns secrecy and integrity bounds to the VM and also appropriately updates the SIESTA configuration files. SIESTA ensures that the given MAC application, the browser, is executed on the system only if the system and application policies are consistent, i.e., if the JIF browser falls within the secrecy/ integrity range of the VM.

Chapter 5

Implementation

We implemented a prototype system according to the design presented in the previous chapter (Chapter 4). This chapter describes our implementation of the various system components in detail.

5.1 System Implementation

In our prototype implementation, we use the Mandatory Access Control based browser developed using JIF (Java Information Flow) language. However, in this thesis we will not be discussing about the browser implementation. We use SELinux as the guest operating system inside the virtual machine to enforce Mandatory Access Control policies and Labeled IPsec mechanism to secure browser communication and convey the security labels to the web server VM. Our implementation uses Xen virtual machine monitor (VMM) to sandbox browser instances and web servers and empower the management domain (dom0) of Xen with *Loader VM* functionalities.

We configured two Xen Virtual Machine Monitor systems, *Riise* and *Kewell* running Xen with sHype. *Riise* is responsible for hosting browser VMs and *Kewell* is dedicated to hosting web server VMs. The Management domains (or dom0s) in each system runs Ubuntu 7.04 (Fiesty Fawn) version with SELinux configured in *strict* mode. All network communications between the browser and web server VMs are mediated by these management domains and authorized

based on the configured IPsec and SElinux policies. Each of the management domains has a copy of the VMLoad daemon, Policy Store and label mapper running, started during the system boot-up.

We built our prototype on the recent unstable version of Xen: *xen-unstable* (change set 15011). The web server VMs (domUs) on *Kewell* run Ubuntu 7.04 with apache configured. The virtual machine images for Browser VMs are also based on Ubuntu 7.04 configured with java virtual machine (JVM) for running the JIF Browser. URL Mapper service is started up on all virtual machines during the boot-up. In addition to that, Browser VMs run the SIESTA service to enforce compliance.

5.2 Enforcement

The prototype implementation uses Labeled IPsec tunnels and SELinux policy modules in dom0 to enforce and extend the information flows enforced at the application and operating system layers.

When the user requests a web page, the browser retrieves the security label of the URL from the URL Mapper and assigns the label to the browser socket used for the connection. When the socket tries to send packets the kernel subsystem in the guest VM checks the IPsec policy database and triggers the SA (security Association) negotiation, using the *racoon* daemon, between the browser VM and source dom0. Racoon uses the pre-shared key between the browser VM and dom0 to authorize the negotiation request. The original labeled Ipsec mechanism has been extended recently to support full SELinux MLS enforcement and has been incorporated into the Linux 2.6.19 kernel. This extension allows fine-grained control of network communications by labeling the SAs with the security label of the socket instead of the application.

The kernel network subsystem in the dom0 extracts the label from the IPsec tunnel and authorizes the request. Once the communication is authorized the labeled IPsec tunnel is extended to the destination dom0 and then to the web server VM. The SA negotiation between the source and destination dom0 is authorized based on the IPsec certificates downloaded from the Certificate Authority during the configuration phase. The ipsec tunnel will be authorized only if the organizations Certificate Authority signs the certificate. The web server VM extracts the label from the IPsec tunnel and authorizes the communication based on its SELinux policy. The IPsec policies at the Source dom0 and the Browser VM we used to test our prototype implementation are given below.

IPsec policy at Source Dom0,

```
spdadd  <Browser VM> <Web Server> any -ctx 1 1 <security context> -P
        out ipsec esp/tunnel/ <Source Dom0> - <Dest. Dom0> /require;
spdadd <Browser VM> <Web Server> any -ctx 1 1 <security context> -P
        in ipsec esp/tunnel/ <Browser VM> - <Source Dom0> /require;
spdadd <Web Server> <Browser VM> any -ctx 1 1 <security context> -P
        in ipsec esp/tunnel/ <Dest. Dom0> - <Source Dom0> /require;
spdadd <Web Server> <Browser VM> any -ctx 1 1 <security context> -P
        out ipsec esp/tunnel/ <Source Dom0> - <Browser VM> /require;
```

IPsec policy in the Browser VM,

```
spdadd <browser VM> <Web Server> any -ctx 1 1 <security context> -P
        out ipsec esp/tunnel/ <Browser VM> - <Source Dom0> /require;
spdadd <Web Server> <Browser VM> any -ctx 1 1 <security context> -P
        in ipsec esp/tunnel/ <Source Dom0> - <Browser VM> /require;
```

The server-side (or destination) dom0 and the web server VM use a very generic IPsec policy like the one given below, and authorizes the request to create IPsec tunnels based on the *racoon* certificates.

```
spdadd <Browser VM> 0.0.0.0/0 any -ctx 1 1 <security context> -P in
        ipsec esp/tunnel/<Browser VM>-0.0.0.0/0/require;
spdadd 0.0.0.0/0 <Browser VM> any -ctx 1 1 <security context> -P out
        ipsec esp/tunnel/0.0.0.0/0-<Browser VM>/require;
```

In the proposed design, we use three orthogonal labeled IPsec tunnels to convey the security label of the browser socket; 1) between browser VM and source dom0, 2) between source and destination dom0s and 3) between destination dom0 and the web server VM. IPsec tunnels between the two dom0s, in addition to conveying the labels also provide strong security guarantees. The only purpose of the other two tunnels is to convey the security labels to their respective dom0s. The *ipsec-tools* package is used to configure and negotiate IPsec security policies and security associations. The *setkey* tool is used to update the SPD (Security Policy Database) and SAD (Security Associations Database) in the kernel and *racoon* is the IKE daemon used for negotiating security associations.

We also looked at other labeling mechanisms for conveying the security labels between VM and their respective dom0s. Labeled VIF, a recent extension to the Xen, allows us to label the virtual interfaces. But it is coarse grained and the implementation is not fully mature. Netlabel, as discussed in Section 2.3.1, currently supports only MLS levels and does not support SELinux types.

5.3 Supporting Services

This section will give the implementation details of the services used for bootstrapping the browser VMs and ensuring the policy compliance.

5.3.1 VMLoad Daemon

As explained in Section 4.2 VMLoad daemon is network daemon running in the management domain and is dedicated to bootstrapping the browser VMs and ensuring compliance. The implementation consists of more than 1100 lines of python code. The VMLoad service is started during the system boot-up and implements the TCPThreadingServer class provided by the SocketServer library. The VMLoad is run on all the dom0s in the organization.

The Browser VM sends a request to the VMLoad daemon to serve the URL request. In our prototype, a simple python client emulates the browser VM and sends a request to the VMload with the following parameters,

1. *VM Type*: This parameter specifies the kind of VM to be loaded. Although currently the implementation has provision to load only Browser VMs, it can be extended in future to load other kinds of VM, like email clients.

2. *Command Line*: This parameter will be passed as a command line argument to the application we intend to start inside the VM. In our prototype, it should contain the URL of the browser instance.

3. *Args*: In the current implementation this parameter is not used but can be used for troubleshooting and for conveying additional information to the VM in future.

The request is authorized at the application-level based on the ip address. The client ip address is checked against a pre-defined list to check whether the VM requesting the load is authorized or not. From the *type* parameter, the Browser VM image is located and a pointer to the virtual machine image is created. It then queries the local URL mapper to retrieve the security label and parses the returned SELinux security context. Like mentioned in Section 4.1 the application-id of the URL is retrieved from the Policy Store based on the security label and the MLS level, say *s4*, of the web page is mapped to an mls range, say *s0-s5*, corresponding to the virtual machine. The VMload daemon then queries the PolicyStore using the application-id to retrieve the IPsec and SELinux policies. The Labeled IPsec policy is cached and the SELinux policy modules are stored on the local disk. The VMLoad daemon also downloads the *racoon* certificates, required for setting up the IPsec tunnel with the destination dom0, from the Policy Store, updates the *racoon* configuration file and restarts the *racoon* daemon to use the downloaded certificates.

The VMload mounts the virtual machine image and updates the system/network and siesta configurations. It also updates the IPsec policy files, which is then loaded into the kernel during the system startup. It also updates the VM SElinux policies in a similar way to provide support for application specific labels. Similarly, the vmload daemon also updates the source dom0 policies. The IPsec policies are loaded using the *setkey* tool. The *xm* tool is used to start the Browser Virtual Machine.

5.3.2 Policy Store

In our prototype, we separated the policy engine from the main VMLoad service to make the system more modular and flexible. The Policy Store is a network daemon running in the

dom0 and with 500 lines of python code. The Policy Store receives request from the *vmload* service to download policies mentioned in the 4.4.

In the current implementation, the Policy Store maintains a template for the IPsec policies, separate template for domU and dom0 policies, and appropriately updates the template with the ip addresses to create the necessary IPsec policies. It also maintains a table containing a list of web servers serving each of the web applications.

```
table = ('PUBLIC', '1' ,'webserver-vm3'),
        (<appid>, '2' ,'webserver-vm1', 'webserver-vm2'),
        ('DEFAULT', '0')
```

On receiving the request for downloading the IPsec policies, the policystore queries the table to retrieve the web servers corresponding the web applications and updates the policy templates. It retrieves the browser VM and dom0 IP addresses from the *vmload*. The SELinux policy modules corresponding to the web application is stored inside the directory with the same name, application ID. The Policy Store retrieves the policy modules from within the directory and sends them to the VMLoad service.

In our prototype implementation, the Certificate Authority is installed in the same machine as the Policy Store. The Policy Store uses the application-id as the 'domain' name to create the private key for the IPsec certificate and requests the CA to sign the generated private key. The *vmload* downloads both the key file and certificate file from the Policy Store. We are yet to implement the Application Authority and currently the Policy Store is responsible for handling all the requests.

5.4 Network Address Translation

In our proposed design, we need to setup Labeled IPsec tunnels between source and destination dom0s to convey the security label, for which we need the IP address of the destination dom0. Currently there is no standard method to find out the dom0's ip address given the ip address of the VM running on it. We can have a central authority maintaining this information and the VMLoad daemon can query the authority for dom0 information. But this approach has some disadvantages, as it is not fully scalable, like most centralized approaches, and also less flexible. Another method is to have a well-define naming convention for assigning ip address to VM. This approach again is not very flexible as it will constrain the VMs and might cause extra overhead during activities such as VM migration.

To overcome these shortcomings, we propose a more flexible and dynamic mechanism based on Network Address Translation, to identify the destination dom0 and setup IPsec tunnels between the source and destination dom0s given the ip address of the web server VM. In this approach, we assign the global ip addresses of all the virtual machines to its dom0 and virtual machines themselves will have private, internal ip addresses. All the virtual machines running on a single system will be part of the internal network with dom0 functioning as the NAT box forwarding the packets to the appropriate virtual machines.

Linux Networking subsystem provides support for assigning multiple ip addresses to the network card using the *ifconfig* tool.

```
ifconfig eth0:0 <VM1 address> netmask 255.255.255.0 up
ifconfig eth0:1 <VM2 address> netmask 255.255.255.0 up
```

We use the *iptables* tool to setup dom0 to forward packets to the respective VMs. The first set of iptable rules performs the source network address translation. The packets coming

from the virtual machines are captured and their ip header is modified according these rules. The second set of iptable rules perform destination network address translation for packets coming into dom0, destined for guest VMs.

```
iptables -t nat -A POSTROUTING -o eth0 -j SNAT
                --to <VM1 address> -s 192.168.0.1
iptables -t nat -A POSTROUTING -o eth0 -j SNAT
                --to <VM2 address> -s 192.168.0.2
```

The above NETFILTER rules are invoked for packets coming into the dom0 from the guest VMs (Browser VMs) and after the routing decisions are made at the dom0. The *iptable* rules replaces the VM's internal IP address with the VM's global IP address in all the IP packets from the guest VMs forwards it.

```
iptables -t nat -A PREROUTING -p tcp -i eth0
                -d <VM1 address> -j DNAT --to 192.168.0.1
iptables -t nat -A PREROUTING -p tcp -i eth0
                -d <VM2 address> -j DNAT --to 192.168.0.2
```

The above rules are invoked for packets coming into the dom0 and destined for one of the guest VMs. These rules replace the IP packet's global IP address with the VM's internal IP address and forwards it to the corresponding VM. These transformations takes place only for TCP packets as the UDP packets, used for racoon negotiations, need to be captured at dom0.

5.4.1 Interaction with Labeled IPsec

The current Labeled IPsec extension to the Linux kernel does not work correctly along with NAT kernel modules. Once the iptable/nat kernel modules are inserted into the Linux

kernel, the ip packet takes a more elaborate route through the *iptables* related functions upon which the packet loses its security identifier, the *secid* field, causing the kernel to re-negotiate the security associations. The patch for 2.6.19 kernel version, provided by Venkat, which resolves the issue is provided in the Appendix A and has not been incorporated into the mainline kernel version yet.

This approach does have some disadvantages. The current iptable rules, mentioned above, in the dom0 forwards the TCP traffic to the corresponding VMs but the UDP traffic is retained in the dom0. The *racoon* daemon uses UDP packets via port 500 to negotiate Security Associations. In our scenario, the racoon negotiations, the IPsec tunnels, should happen between the dom0s while the web traffic needs to be forwarded to the VMs but the ip packets for both the traffic uses the same ip address, i.e., the global ip address of the web server VM.

This approach allows us more flexibility in-terms of moving the VMs without affecting the system configurations. This method is particularly helpful in scenarios where virtual machines have well-defined set of functions like web browsing, email clients etc., and may not work well with general-purpose machines.

5.5 Pre-Loaded VMs

In the proposed mechanism, every-time the VMLoad daemon receives a request it loads a new browser VM to serve the web page. As analyzed in Section 6.2 loading a new browser VM from scratch takes up considerable amount of time, which may not be acceptable especially in a web-based environment. And also this approach will result in redundant browser VMs, i.e., more than one browser VM serving the same web application as there is no mechanism to

forward requests to existing virtual machines. To minimize the URL access time and also to avoid redundant browser VMs we propose the concept of Pre-Loaded VMs.

Three kinds of Browser VMs can be running in the system at any point of time.

1. IDLE browser VMs that are not currently serving any web application.

2. Virtual Machines that are currently serving a web application within a particular secrecy/integrity range and ready to accept requests to serve more web pages.

3. Virtual Machines that are currently serving a web application within a particular secrecy/integrity range and not ready to accept further requests.

The VMLoad daemon described in Chapter 4.2 is modified to support this scenario. The VMLoad daemon running in dom0 maintains a table of currently running VMs in the system with other details such as application-id, secrecy range etc.

```
table = (1, <VM1 address>, 'MED-001', 's0-s5', 'SERVING', 'ACCEPT'),
        (2, <VM2 address>, 'MED-001', 's6-s7', 'SERVING', 'BUSY'),
        (3, <VM3 address>, '', '', 'IDLE', 'ACCEPT')
```

When the VMLoad daemon receives the request to serve a web page it downloads the necessary policies as described in Chapter 4.2 and queries the table to get VM details.

1. If a Browser VM is already serving the web application at appropriate secrecy/integrity range and is ready to accept more web page requests, the VMLoad daemon sends a request to the SIESTA service to start a new browser instance.

2. If there is no such VM, the daemon chooses an IDLE browser VM and sends a request to the *update-d* daemon inside the Browser VM update the IPsec and SELinux policy modules. It also sends a request to the SIESTA service to start a new JIF browser.

3. Finally, if no IDLE VM is found, the VMLoad daemon follows the conventional path to load a new browser VM.

This table is updated accordingly each time the VMLoad daemon forwards the URL request. In the current implementation, the decision as to whether the browser VM serving the web application can accept further requests or not is taken randomly, but in future it could be extended to take into account the resource usage and other such properties of the VMs.

5.5.1 Supporting Services

To incorporate the above-mentioned design into the system prototype we added the *update-d* service to each of the running browser VMs and modified the SIESTA to act as a network daemon to serve requests from its dom0.

The *update-d* is a network daemon implemented using python and started using the *initscripts* during the VM boot-up. This service receives requests from dom0 to update the IPsec and SELinux policies in the browser VMs. *iptable* rules were used in the Browser VMs to restrict *update-d* and *siesta* to receive requests only from their respective dom0s.

Chapter 6

System Evaluation

This chapter analyses two aspects of the system performance: (1) Cost of booting a new virtual machine to execute a browser instance, and (2) latency and throughput measurements for fetching a web page through a series of IPsec tunnels. The prototype implementation is not yet optimized and hence the performance results should be considered as an upper bound on the overhead.

6.1 System Setup

The setup used for taking measurements is described in Section 5.1. Our measurements were made on an Intel Pentium Dual Core CPU with 2.80GHz clock, with 1GB of RAM, and 250GB serial ATA hard drive. For Network related measurements two machines of similar configurations are connected to an DYNEX 10/100 Ethernet Switch using an on-board NetXtreme Gigabit Ethernet NIC.

The Xen dom0 kernel is installed in each of these machines with support for 420MB of dom0 Memory and 8GB of swap space. All web server VMs are configured to use 256MB of RAM and 2GB of swap space. Similarly, the browser VMs are configured to use 128MB of RAM and 2GB of swap space. Our initial prototype uses *vif-route* for Xen network configuration where dom0 acts as a router forwarding traffic to/from the guests. For the approach described in

Section 5.4, we use the *vif-nat* configuration where dom0 acts as a Network Address Translation box.

For measuring the Network Performances, we use *netperf* [1] version 2.4.2-1. Netperf is a benchmark that can be used to measure various aspects of networking performance including TCP/UDP throughput and Request/Response times. The Netperf toolkit consists of two components: *netperf* and *netserver*. The *netperf* component is the performance benchmark client and it sends out network packets to *netserver* daemon.

6.2 Browser Startup Latency

In our prototype implementation, we create a new virtual machine to execute the browser instance. The VM creation happens whenever the user wants a view a page belonging to a new web application or a URL that the VM is not authorized to access, not within the secrecy/integrity range of the browser VM. We wanted to measure the overhead of loading a new Xen virtual machine on a new web page request.

Operation		**Average Latency**
Proposed design	Configure/Load Policy	4.90 seconds
	Loading Browser	18.50 seconds
	Loading Webpage	13.80 seconds
Pre-Loaded VMs	Configure/Load Policy	0.90 seconds
	Loading Browser	2.80 seconds
	Loading Webpage	13.50 seconds
Native Browser	Cold-start	7.45 seconds
	Warm-start	1.50 seconds

Table 6.1. Browser Startup Latency

Table 6.1 shows the cost (in seconds) of starting a new JIF browser inside a Xen VM compared to the cost of starting a new *firefox* browser in the dom0. The top row of the table shows the cost incurred by different phases while loading the web page inside a new Virtual Machine. We loaded the VMs three times to ensure repeatability, and took an average of the three. The first line in the top row corresponds to configuring and loading application specific policies. The latency of 4.90 seconds is low considering the amount of socket-based operations required to download policies from the Policy Store. In the initial prototype, the Policy Store and Certificate Authority are installed locally, hence the latency is less but we expect the latency to increase quite a bit once we move these components to a centralized server. And also we expect some increase in the latency once we implement the URL Mapper fully based on the specification.

The second line corresponds to the Xen VM boot-up time. We believe it is possible considerably reduce VM boot-up time from the measured 18.50 seconds if we optimize the virtual machine image to load only necessary services and also it depends heavily on the system's resource usage model. The third line corresponds to the time it takes to load the new web page inside the JIF browser. The latency of 13.80 seconds is mainly due to IPsec SA negotiation, and happens only when the browser accesses the web server for the first time. There is not much literature in the field analyzing IKE negotiation delay. Okhee *et al*. [23] analyze the performance of IPsec/IKE in large scale VPN systems. With their setup, it took 0.9 seconds to establish a pair of SAs (Initial SA delay). The reasons, we believe, for high latency in our prototype are

- Setup of three orthogonal IPsec tunnels. At each point, we need to buffer the actual packet, initiate the IPsec SA negotiation, and forward the buffered packet.

- Overhead on the system due to virtual machine configurations.

By predefining SAs between the web server VMs and destination dom0, we were able to reduce the load latency to 7-8 seconds. The 13.80 seconds is the upper bound and we strongly believe that it is possible to considerably reduce the latency with some optimizations to the linux system and tweaking the IPsec/racoon configurations.

From the values we can conclude that majority of the time is spent on loading a new Xen VM. To avoid this overhead, our system maintains a pool of IDLE Browser VM, Section 5.5. The middle row shows the cost of loading a new web page in the pre-loaded VM scenario. Surprisingly, it took less than a second to configure and load policies compared to the 5 seconds latency in the previous case. We believe the difference is mainly due to costly disk writes. In the initial setup, the browser VM is mounted onto to a local directory and the application policies are installed into the mounted directory whereas in the pre-loaded VM scenario the application policies are sent directly to the *update-d* daemon inside the VM through a network connection. With pre-installed SAs it took less than 4 seconds to load a new web page in this scenario.

For comparison, the bottom row of the table shows the latency of opening a *firefox* browser window on the dom0. We measured two cases: (1) the ”cold-start” latency of launching the browser freshly, (2) the ”warm-start” latency of launching the browser, assuming it has been previously launched. We believe that with further optimizations it is very much possible to reduce the system latency on par with the *firefox* ”cold-start” latency.

6.3 Network Performance

To evaluate the network performance of our prototype, we measure the achieved TCP throughput. Our main aim is to measure the performance overhead due to IPsec processing at dom0, web server VMs and browser VMs. In our prototype implementation, we setup three orthogonal IPsec tunnels: (1) between Browser VM and dom0, (2) between source and destination dom0s, and (3) between web server VM and dom0.

For all measurements, IPsec was used in tunnel mode with *hmac-sha1* algorithm for authentication and *3des* algorithm for encryption. The *null* encryption mode was used for IPsec tunnels with domUs as the only purpose of those tunnels is to convey the security label of the Browser Socket.

TCP_STREAM tests were used to take throughput measurements. TCP_STREAM is the default test that netperf runs, and it measures bulk data transfer performance in megabits/sec. We ran each test three times from inside the VM to ensure repeatability, and took an average of the three test runs. We set the test time for each run to be one minute. For 2/3 clients scenarios, we loaded 2/3 browser VMs and issued the *netperf* command from inside each of the VMs. We measured the throughput values from the initial browser VM and the other VMs are used to create additional processing load on the machines.

The command used for taking measurements is given below,

```
netperf -H <webserver VM> -t TCP_STREAM -C -c -l $time
```

Figure 6.1 shows the results. In the case of the native Xen system, without any IPsec tunnels, the measured TCP throughput was 93.95 Mbps. TCP throughput measurements are

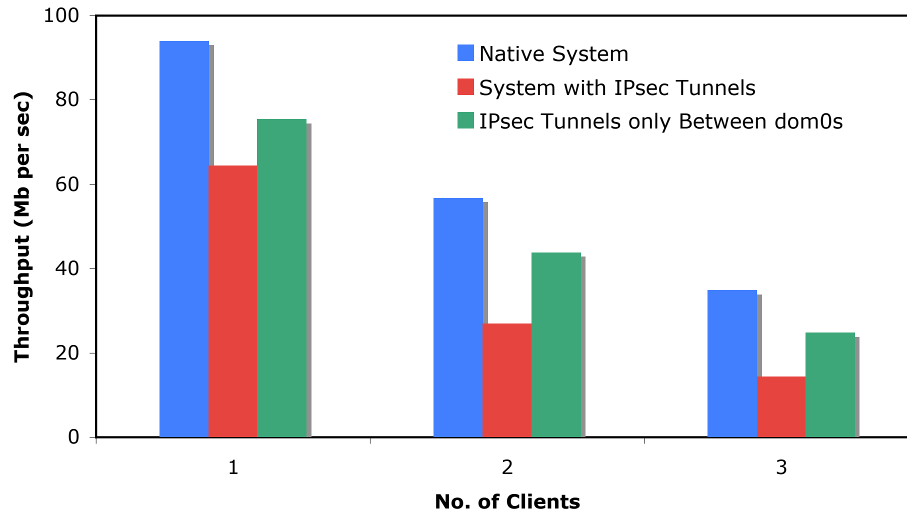

Fig. 6.1. TCP Throughput values for measuring IPsec Processing overhead

sensitive to small parameter changes (such as buffer sizes, segment sizes etc.,) and system configurations. Hence, it is difficult to specify an accurate comparison, though the above metrics can be considered as a base for further evaluation. The TCP throughput measurement for the same system configured with all three IPsec tunnel segments, described above, was around 64.5 Mbps. We can see significant, one third, drop in the system throughput, though we believe it is possible to improve the throughput by allocating more resources for dom0 IPsec processing or use asynchronous processing mechanisms such as Acrypto [31].

We were able to get significant performance improvement by neglecting the IPsec tunnels with domUs (VMs) and having only the source dom0 to destination dom0 IPsec tunnel. The IPsec tunnel between source and destination dom0s, in addition to conveying the security label, also authenticates and protects the network packets whereas the only purpose of the other two IPsec tunnels it to convey the security label of the browser socket. Other labeling mechanisms,

such as *netlabel* or *Labeled VIFs*, can be used to convey labels to/from the dom0. The values for the three scenarios scaled linearly for 2 and 3 clients scenario.

All browser network communications flow through the dom0, where they are translated according to the *iptable* rules. We also measure the throughput difference incurred because of this additional overhead of indirection and filtering.

System Setup		**TCP Throughput**
Native System	with NAT	93.90 Mbps
	w/o NAT	94.10 Mbps
System with IPsec Tunnels	with NAT	64.50 Mbps
	w/o NAT	64.20 Mbps

Table 6.2. Overhead due to Network Address Translation

From Table 6.2, we can presume that the additional cost of the *iptable* filtering is almost negligible. The difference in the throughput values can be attributed to different system-dependent parameters, but we can safely conclude from the above values that the NAT approach defined in Section 5.4 does not incur additional overhead on the system's overall performance.

6.3.1 Summary

Summarizing the performance benefits. The benchmarks demonstrate that despite the overhead of virtualization, address translation and tunneling the proposed approach can achieve the latency, throughput characteristics required by the most of the common web applications.

6.4 Web Application Mock-up

We demonstrate the working of our system prototype by building an analyst- driven web application, which we believe, is used commonly for sharing sensitive information. In our model, the analyst is responsible for gathering information about a particular *area of interest* from different sources, both trusted and untrusted. In addition, analyst also maintains a list of people's profiles mapped to their *area of interest* in his trusted web server. In the prototype, the analyst can browse/search through the list from inside a browser VM.

In the above-discussed model, some of the profiles are linked to trusted web servers while others are linked to untrusted ones. In our design, the analyst should be able to access the trusted web servers from inside the same browser VM, either using different JIF browsers or using different tabs inside the same JIF browser. When the analyst tries to access a web page from an untrusted web server, the JIF browser sends a request to the *vmload* daemon, which authorizes the request and either loads a new VM to serve the web page or allocates one from the pool of IDLE VMs.

6.4.1 System Design

On system startup, the analyst requests VMLoad service in dom0 to load a new browser VM to access the homepage of his trusted web server. In the current implementation, the VMLoad service authorizes this request as it is initiated from inside the dom0. The VMLoad service contacts the local URL mapper to retrieve the security label of the web page.

Currently, the URL Mapper retrieves the label from its local cache of mappings like the one given below,

```
http://kewell-vm1.cse.psu.edu/public.html --
```

```
                  root:sysadm_r:analyst_t:s4
http://kewell-vm1.cse.psu.edu/synth/index.html  --
                  root:sysadm_r:analyst_t:s7
```

But eventually it uses the protocol described in Section 4.3 to download the mappings.

The VMLoad service requests the Policy Store to map the security label to a distinct application ID and also maps the security level encoded in the label to a generic security range. VMLoad daemon requests the policy store to download the necessary IPsec and SELinux policies. Policy Store maintains a DB containing the list of web servers hosting the web application and retrieves the web servers corresponding to the analyst application from the DB. And it requests the VMLoad for the browser VM ip addresses and generates the IPsec policy for dom0 and browser VM. Currently, we do not generate SELinux policies but we define a mechanism to download existing policies.

Sample dom0 IPsec policy, IPsec policy at Source Dom0,

```
spdadd  <Browser VM> <Web Server> any -ctx 1 1 <security context> -P
        out ipsec esp/tunnel/ <Source Dom0> - <Dest. Dom0> /require;
spdadd <Browser VM> <Web Server> any -ctx 1 1 <security context> -P
        in ipsec esp/tunnel/ <Browser VM> - <Source Dom0> /require;
spdadd <Web Server> <Browser VM> any -ctx 1 1 <security context> -P
        in ipsec esp/tunnel/ <Dest. Dom0> - <Source Dom0> /require;
spdadd <Web Server> <Browser VM> any -ctx 1 1 <security context> -P
        out ipsec esp/tunnel/ <Source Dom0> - <Browser VM> /require;
```

Policy Store also acts as a local Certificate Authority and also generates and signs the racoon certificates. The VMLoad service also installs configuration options for the SIESTA

daemon and specifies the secrecy/integrity range within which the Jif browser should be started. The VMLoad service after installing these policies appropriately requests the *Xend* to load the browser VM.

Once the browser VM is loaded, SIESTA loads the Jif browser. When the browser tries to load the homepage, *racoon* triggers the SA negotiation and all the three orthogonal tunnels are established before the web page is loaded. The browser VM allows the analyst to browse through the trusted web servers. All objects served by these web servers fall within the secrecy range of the browser.

When the Jif browser tries to access an object from the untrusted webserver, the security level does not fall within the secrecy range and hence SELinux policy should prevent. When it tries to access a different web application, the system will throw an error due to lack of labeled IPsec policy. In either case, it will send a request to the VMLoad service to load a new Browser VM.

6.4.2 Security Guarantees

One of the main security goals of this analyst-based application is to clearly isolate trusted and untrusted web sites. We differentiate untrusted and trusted web objects based on their security level and the web server address.

- SIESTA and the enforced SELinux policy ensures that the Jif browser does not handle objects beyond the secrecy/integrity range of the virtual machine.
- Labeled IPsec policy and correspoding SELinux policy modules ensure that Jif browser cannot access objects not belonging to the web application.

SELinux policy mediates all accesses to system objects and Labeled IPsec policy controls all the network accesses. In addition, dom0 kernel mediates all network communications from the virtual machines.

6.4.3 Summary

In this model, we assume that the level of assurance provided by the JIF browser and SELinux operating system is enough to allow the same browser VM to access different trusted web servers, at different secrecy ranges but it is not safe to allow both trusted and untrusted web pages to be co-browsed. In the latter case, we browse the untrusted web pages using a different browser VM. Further, we assume that all the web servers discussed above, hosting people's websites, belong to the same *web application*.

Chapter 7

Conclusion and Future Work

In this thesis, we describe a framework for extending and enforcing the information flow policies, that is currently being enforced at application and operating system layers, at the virtual machine and network layers and also ensuring compliance, thereby guaranteeing system-wide enforcement of information flows. Major insights are that the use of Labeled IPsec for conveying the security labels, SELinux policy modules for authorizing the IPsec tunnels and Xen virtual machine monitor for isolating web applications from one another.

In addition, we also introduced the concept of pre-loaded VMs, where we maintain a pool of IDLE VMs, to reduce the latency and propose a mechanism based on Network Address Translation to identify destination dom0 address without having to query some central authority.

Our performance evaluation demonstrates that despite the overhead imposed, the proposed system can achieve performance characteristics required by common web applications, although there is always some tradeoff between security and performance.

Future work includes reducing the browser load latency and exploring other mechanisms, such as *netlabel*, *labeled vifs* and *secmark*, for conveying the label of the network packets to the dom0. Instead of running a full Linux kernel with other supporting services in the browser VM, we can run a stripped down version of the kernel with very minimal code needed to support a Java Browser. This will significantly improve the VM startup time and also reduce the load on the system allowing more virtual machines to run simultaneously.

We also envision moving all the services in the dom0 to a separate Xen Domain and allow the dom0 to only enforce the application security policy. In our current prototype, the decision to authorize the *vmload* request is made randomly, but in future we envision defining an authorization mechanism incorporating system's resource usage model and the requesting VM's security label. In future, we also plan to define a mechanism for automatically generating application specific SELinux policy modules and extend the framework to support other distributed applications as well.

Web applications, because of its ease-of-use and ubiquitous nature, are being used more and more regularly for sharing information in secure environments, which requires strong security guarantees. In the recent past, there has not been much work on integrating the information flow guarantees that are being provided at different layers. We believe, this thesis will help in someway in achieving the overall goal.

Appendix A

Interaction Between Labeled IPsec and NAT in Linux Kernel

The current Labeled IPsec extension to the linux kernel does not work correctly along with NAT kernel modules. Once the iptables NAT related rules are inserted, the labeled IPsec connection breaks and the kernel trigger a re-negotiation creating duplicate SAs. These re-negotiations happen continuously and the packets never go through. This is a bug in the current labeled IPsec implementation in the kernel.

A.1 Details

When the *NAT* rules are inserted into the kernel, the NETFILTER NAT hooks are enabled. When the ip packet traverses through the netfilter hooks, the *ip_route_me_harder* function is called. In this the *ip_route_me_harder* function, the *flowi* structure is newly constructed from the *skb* (*skbuff* structure) and calls the *xfrm_decode_session* function to assign the *flowi->secid* field. The *secid* field should correspond to the security label of the Security Association (SA). The *xfrm_decode_session* function assigns the secid field based on the *skb->secpath* which is NULL at this stage of the processing. Since the *secid* field is '0' the *xfrm_lookup* function triggers a re-negotiation. This whole processing will be skipped if the nat related modules are not inserted into the kernel and thus the system will behave correctly.

A.2 Patch

Venkat Yekkirala provided us a patch to solve the issue and it has not been incorporated into the mainline linux kernel. Venkat's patch uses the *skb->sk* (socket) structure to fill in the *flowi->secid* field. It uses another function, *security_sk_classify_flow*, which also fills in the *flowi->secid* field, but from the socket strucutre, *sock->sk_security*, instead of the *skb->secpath*.

```
--- netfilter_orig.c    2008-05-15 10:45:44.000000000 -0400
+++ netfilter.c 2008-05-15 10:45:32.000000000 -0400
@@ -58,9 +58,12 @@

 #ifdef CONFIG_XFRM
        if (!(IPCB(*pskb)->flags & IPSKB_XFRM_TRANSFORMED) &&
-           xfrm_decode_session(*pskb, &fl, AF_INET) == 0)
+           xfrm_decode_session(*pskb, &fl, AF_INET) == 0) {
+               if (!fl.secid && (*pskb)->sk)
+                       security_sk_classify_flow((*pskb)->sk, &fl);
                if (xfrm_lookup(&(*pskb)->dst, &fl, (*pskb)->sk, 0))
                        return -1;
+       }
 #endif

        /* Change in oif may mean change in hh_len. */
@@ -93,6 +96,9 @@
        if (xfrm_decode_session(*pskb, &fl, AF_INET) < 0)
                return -1;
```

```
+       if (!fl.secid && (*pskb)->sk)
+               security_sk_classify_flow((*pskb)->sk, &fl);
+
        dst = (*pskb)->dst;
        if (dst->xfrm)
                dst = ((struct xfrm_dst *)dst)->route;
```

References

[1] Netperf, A Network Performance Benchmark. `http://www.netperf.org/netperf/NetperfPage.html`.

[2] A. Acharya and M. Raje. Mapbox: Using parameterized behavior classes to confine applications. Technical report, University of California at Santa Barbara, Santa Barbara, CA, USA, 1999.

[3] Roland Awischus. Role based access control with the security administration manager (sam). In *RBAC '97: Proceedings of the second ACM workshop on Role-based access control*, pages 61–68, New York, NY, USA, 1997. ACM.

[4] Dirk Balfanz and Daniel R. Simon. Windowbox: a simple security model for the connected desktop. In *WSS'00: Proceedings of the 4th conference on USENIX Windows Systems Symposium*, pages 4–4. USENIX Association, 2000.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[6] D. E. Bell and L.J. LaPadula. Mathematical foundations and model. Technical report, MITRE, 1973.

[7] Andrew Berman, Virgil Bourassa, and Erik Selberg. Tron: process-specific file protection for the unix operating system. In *TCON'95: Proceedings of the USENIX 1995 Technical Conference Proceedings on USENIX 1995 Technical Conference Proceedings*, pages 14–14, Berkeley, CA, USA, 1995. USENIX Association.

[8] Suresh N. Chari and Pau-Chen Cheng. Bluebox: A policy-driven, host-based intrusion detection system. *ACM Trans. Inf. Syst. Secur.*, 6(2):173–200, 2003.

[9] Richard S. Cox, Steven D. Gribble, Henry M. Levy, and Jacob Gorm Hansen. A safety-oriented platform for web applications. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 350–364, Washington, DC, USA, 2006. IEEE Computer Society.

[10] Paloma Díaz, Ignacio Aedo, and Fivos Panetsos. Modelling security policies in hypermedia and web-based applications. In *Web Engineering*, pages 90–104, 2001.

[11] FreeBSD Foundation. SEBSD: Port of SELinux FLASK and type enforcement to TrustedBSD. `http://www.trustedbsd.org/sebsd.html`.

[12] Chris Grier, Shuo Tang, and Samuel T. King. Secure web browsing with the op web browser. In *IEEE Symposium on Security and Privacy*, pages 402–416, 2008.

[13] IETF CIPSO Working Group. COMMERCIAL IP SECURITY OPTION (CIPSO 2.2). `http://netlabel.sourceforge.net/files/draft-ietf-cipso-ipsecurity-01.txt`, July 1992.

[14] Boniface Hicks, Sandra Rueda, Trent Jaeger, and Patrick McDaniel. From Trusted to Secure: Building and executing applications that enforce systems security. In *Proceedings of the 2007 USENIX Annual Technical Conference*, pages 205–218, May 2007.

[15] Boniface Hicks, Sandra Rueda, Trent Jaeger, and Patrick McDaniel. Integration of SELinux and security-typed languages. In *Proceedings of the 2007 Security-Enhanced Linux Workshop*, pages 85–92, March 2007.

[16] Trent Jaeger, Kevin Butler, David H. King, Serge Hallyn, Joy Latten, and Xiaolan Zhang. Leveraging IPsec for mandatory access control across systems. In *Proceedings of the Second International Conference on Security and Privacy in Communication Networks*, August 2006.

[17] Trent Jaeger, Aviel D. Rubin, and Atul Prakash. Building systems that flexibly control downloaded executable context. In *SSYM'96: Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography*, pages 14–14, Berkeley, CA, USA, 1996. USENIX Association.

[18] Trent Jaeger, Reiner Sailer, and Yogesh Sreenivasan. Managing the risk of covert information flows in virtual machine systems. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*, pages 81–90, June 2007.

[19] Leon Jololian. Distributed computing environment (dce).

[20] James B. D. Joshi, Walid G. Aref, Arif Ghafoor, and Eugene H. Spafford. Security models for web-based applications. *Commun. ACM*, 44(2):38–44, 2001.

[21] Per Kaijser, Tom Parker, and Denis Pinkas. Sesame: The solution to security for open distributed systems. *Computer Communications*, 17(7):501–518, 1994.

[22] Paul A. Karger. Multi-level security requirements for hypervisors. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 267–275, Washington, DC, USA, 2005. IEEE Computer Society.

[23] Okhee Kim and Doug Montgomery. Behavioral and performance characteristics of ipsec/ike in large performance scale vpns. Technical report, Advances Network Technologies Division, National Institute of Standards and Technology.

[24] J. Kohl and C. Neuman. The kerberos network authentication service (v5), 1993.

[25] Robert Meushaw and Donald Simard. Nettop: Commercial technology in high assurance applications, 2000.

[26] SUN Microsystems. Trusted solaris operating environment - a technical overview. `http://www.sun.com`.

[27] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow. `http://www.cs.cornell.edu/jif, July2001-2003`.

[28] R. E. Niemeyer. Using web technologies in two mls environments: a security analysis. In *ACSAC '97: Proceedings of the 13th Annual Computer Security Applications Conference*, page 205, Washington, DC, USA, 1997. IEEE Computer Society.

[29] NSA. Security-enhanced Linux. `http://www.nsa.gov/selinux`.

[30] Elliott I. Organick. *The multics system: an examination of its structure*. MIT Press, Cambridge, MA, USA, 1972.

[31] Evgeniy Polyakov. Acrypto - asynchronous crypto layer for Linux Kernel. `http://tservice.net.ru/~s0mbre/old/?section=projects&item=acrypto`.

[32] Sandra Rueda, Dave King, and Trent Jaeger. Verifying the compliance of trusted programs, Aug 2008.

[33] Reiner Sailer, Trent Jaeger, Enriquillo Valdez, Ramon Caceres, Ronald Perez, Stefan Berger, John Linwood Griffin, and Leendert van Doorn. Building a mac-based security architecture for the xen open-source hypervisor. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 276–285, Washington, DC, USA, 2005. IEEE Computer Society.

[34] Pierangela Samarati, Elisa Bertino, and Sushil Jajodia. An authorization model for a distributed hypertext system. *IEEE Trans. on Knowl. and Data Eng.*, 8(4):555–562, 1996.

[35] Constantine Sapuntzakis and Monica S. Lam. Virtual Appliances in the Collective: A Road to Hassle-Free Computing. In *Proceedings of the Ninth Workshop on Hot Topics in Operating System*, May 2003.

[36] Brian C. Schimpf. Securing web access with dce. In *SNDSS '97: Proceedings of the 1997 Symposium on Network and Distributed System Security*, page 102, Washington, DC, USA, 1997. IEEE Computer Society.

[37] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing selinux as a linux security module. Technical report, NAI Labs, 2001.

[38] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. The flask security architecture: system support for diverse security policies. In *SSYM'99: Proceedings of the 8th conference on USENIX Security Symposium*, pages 11–11, Berkeley, CA, USA, 1999. USENIX Association.

[39] Green Border Technologies. GreenBorder desktop DMZ solutions. `http://www.greenborder.com`, November 2005.

[40] Christopher Vance, Todd Miller, and Rob Dekelbaum. Security-enhanced darwin: Porting selinux to mac osx. In *Proceedings of the Third Annual Security Enhanced Linux Symposium*, Baltimore, MD, USA, march 2007. IEEE Computer Society.

[41] David A. Wagner. Janus: an approach for confinement of untrusted applications. Technical report, Berkeley, CA, USA, 1999.

[42] Information Week. U.S. Spy Agencies Go Web 2.0 In Effort To Better Share Information. `http://www.informationweek.com/news/internet/showArticle.jhtml?articleID=201801990`.