Flexible Security Configuration for Virtual Machines

Sandra Rueda, Yogesh Sreenivasan, Trent Jaeger
Systems and Internet Infrastructure Security Laboratory
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802
{ruedarod, sreeniva, tjaeger}@cse.psu.edu

ABSTRACT

Virtual machines are widely accepted as a promising basis for building secure systems. However, while virtual machines offer effective mechanisms to create isolated environments, mechanisms that offer controlled interaction among VMs are immature. Some VM systems include flexible policy models and some enable MLS enforcement, but the flexible use of policy to control VM interactions has not been developed. In this paper, we propose an architecture that enables administrators to configure virtual machines to satisfy prescribed security goals. We describe the design and implementation of such an architecture using SELinux, Xen and IPsec as the tools to express and enforce policies at the OS, VM and Network layers, respectively. We develop a web application using our architecture and show that we can configure application VMs in such a way that we can verify the enforcement of the security goals of those applications.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and protection —*Access Controls*

General Terms

Security

Keywords

Access Control, Policy, Compliance, Virtual Machines

1. INTRODUCTION

Virtual machine systems have been promoted recently as an effective basis for building secure systems. Virtual machine systems enable the execution of commodity operating systems and applications within an isolated environment, thus protecting them from other code running on the same platform (and vice versa). Virtual machines systems have provided this isolation mechanism for years [1], but the recent introduction of virtual machine systems for commodity processors [5] has brought such isolation to insecure commodity operating systems, Windows and Unix.

CSAW'08, October 31, 2008, Fairfax, Virginia, USA.

The question is how we are going to leverage virtual machine isolation to build secure systems. For distributed applications, a virtual machine system still needs to provide the ability for VMs to share data and communicate via untrusted networks. For example, commercial virtual machine systems do not provide control over access to network resources, so while a VM may be isolated from others on that platform, unauthorized communications may result in the leakage of information or integrity violations. Virtual machine systems are being extended with reference monitoring to control inter-VM communications [3, 20], but approaches to leverage such controls have yet to be developed.

Virtual machine systems that provide an architecture for enforcement are not very common. Karger et al developed the VAX VMM security kernel [9], which enforced an MLS policy over VM function. This work demonstrated that a commercial virtual machine system could be a basis for mandatory access control enforcement, although the system was not released publicly. NetTop [12, 6] and Solaris Containers [10] are products that use virtualization technologies to enforce MLS policies. These approaches benefit from the assumption of well-defined MLS policies across all platforms, so when a VM is launched, it is clear in a global sense what its permissions are. Some distributed applications are not so clear-cut. An alternative is the Tahoma approach. Tahoma is a virtual machine system that targets web applications [4]. While Tahoma enables the configuration of access control for a VM dynamically, the web server is assumed to be the source of such policies. Because the web server cannot be an authority over client data, policy is limited to the web sites that can be visited by the client. However, the client may still leak secret data or suffer integrity failures as a result of providing these VMs with access to any sensitive data, thus limiting the scope of web applications.

Our goal is to develop a virtual machine architecture whereby VMs may be used as flexible, controlled domains, unlike the MLS VM systems, and where the controls are defined to be consistent with system's security goals, unlike Tahoma. The insight is that it should be possible to configure VM policies necessary to ensure that the new VM's operations satisfy the system's security goals. In this paper, we identify the operations requiring control and how to configure VM and other policies to control such operations. We also use our prior work on *policy compliance* [7, 18] to ensure that the policies deployed within the VM are compliant with the system security goals that govern its function.

In this paper, we make the following contributions:

1. We build an architecture that enable administrators to configure flexible mandatory access control policies for VMs dynamically.

2. We leverage mandatory access control at operating system,

virtual machine monitors and network layers in order to enforce system policies that rule the communications between virtual machines.

3. The architecture also identifies the places where compliance must be evaluated in order to ensure the enforcement of common security goals across the involved layers (OS, VMM and network). We leverage our prior work on policy compliance to ensure compliance of VM security policy and system security goals [7, 18]. However, we do not focus on compliance on this paper.

The rest of the paper is organized as follows: Section 2 motivates the need for our architecture and its general requirements. Section 3 describes the components of our architecture and their configuration. Section 4 presents the way we ensure policy compliance. Section 5 describes the implementation of the architecture, and finally Section 6 concludes and elaborates on future work.

2. PROBLEM

Figure 1 shows a canonical distributed application. This is a client-server application where information may flow from the client to the server (e.g., web forms, document uploads), from the server to the client (e.g., information queries, media downloads), or both. Either information flow may be risky to the client (or server) because the client (or server) may leak information that she did not intend or she may receive information that may compromise the integrity of the client (or server) system. Example applications that fit this scenario include grid applications, where the results of the computation must be high integrity and may need to be secret, and web applications, which may be able to transfer secret data or may possibly include low integrity data.

2.1 Web Applications

In this paper, our focus is on web applications. Originally, web applications were simple client-server applications in which the client downloads content from a server, but web applications are now complex, multi-party computations where content may be composed based on client inputs from multiple sources. Web mashups compose web pages from content, including executable code, that originate from multiple sources. As a result, a web application may involve downloading data from multiple servers. Instead of the traditional client-server application shown in Figure 1, such web mashups have a client interacting with multiple servers, each in a different administrative domain potentially.

Further, clients may use such web applications to upload data to any one of those servers. For example, U.S. spy agencies envision using web applications as the basis for sharing information among clients [23]. This presents both secrecy and integrity issues. Clearly, secret information may be passed among users, but

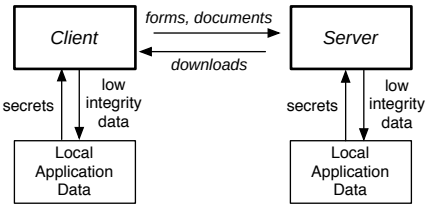

Figure 1: Canonical Distributed Application. Information may flow in both directions.

a variety of information also comes from public, so-called "open sources" [1], so inadvertent and malicious leakage must be prevented. Also, integrity protection is an issue, as web mashups that interact with some untrusted servers may compromise the integrity of computations with trusted servers.

Previous security architectures for web applications derived security requirements from the server's perspective, but the web applications above indicate that the client also has security requirements that must be enforced. The Tahoma security architecture isolates web applications into individual VMs that can only access URLs specified by the server [4]. As multiple servers may be involved in a web application, some determined by third parties (e.g., to provide advertisements), it is neither practical nor effective for a server to define the security policy over a client's interaction with all servers. The Swift architecture protects server data from unauthorized access by clients while automatically decomposing applications into client and server components to improve performance [2]. If the client has secret information that it wants to protect from the server or believes that the server is providing low integrity content, it cannot express such requirements in the Swift system.

2.2 Layered Architectures

Virtual machine monitors and applications are now capable of enforcing mandatory access controls, in addition to operating systems. We argue that each layer will play an important role in enforcing system security goals. Virtual machine monitor security architectures, such as Xen sHype [20] and XSM [3], enable flexible, controlled interaction among VMs. Using such approaches, we can configure web applications into their own VMs and can control which servers each VM can communicate with. This is called the *separation kernel* approach [19], and is the basis for the Tahoma approach. The problem is that not all interactions within a web application are the same from a security perspective. Some interactions may involve secret communications, where others do not. Also, some servers in a web application may provide high integrity content, where others do not. As a result, we believe that using VMs is beneficial, but access control within the VM will also be necessary.

Within the VM, operating systems that implement mandatory access controls, such as SELinux [14], and policy enforcing applications, such as those based on security-typed languages [13, 21] or the ones that use application-level reference monitors [22, 11, 15], can enforce mandatory access control policies. For web applications, we believe that enforcement both at the OS and application level will be necessary. We find that some security functions are more effectively handled by applications, for instance integrity protection [17].

The problem is that having multiple layers of enforcement means that the MAC enforcement may be inconsistently enforced in some levels. The administrator must ensure that MAC enforcement requirements are supported by the VMM, VM's operating system, and applications, where necessary. Since we do not want to put the burden of examining multiple policies on administrators our approach must provide automatic mechanisms to convey MAC policy consistently among these layers.

As a basis for defining consistent MAC policies, all MAC policies must comply with the host system's security goals. Unlike the Tahoma system, where each VM is an application component that does not interact with the host system, we envision that our distributed applications may leverage some host system resources.

[1]Freely available information, which is analogous to open source software.

Thus, we cannot use the Tahoma approach where policies are defined by the web server, but rather the host system's administrative domain must govern the accesses available to each VM. In this work, we develop a virtual machine architecture that ensures that the host's MAC requirements are enforced across all layers (VMM, OS, and application) in a VM system.

Based on the analysis above, we claim that a virtual machine architecture must support the functions described below to deploy VMs in a manner that ensures enforcement of a system's mandatory access control (MAC) requirements.

1. *Policy Compliance*: The virtual machine architecture must ensure that each component enforces the system's MAC policy. If a layer's MAC policy enforces the system's MAC policy, we say that its MAC policy *complies* with the system's MAC policy. In our architecture, we only deploy layers that have compliant MAC policies.

2. *Policy Configuration*: The compliant MAC policies must be configured at each layer of the system. This process should be automated as much as possible to ensure that the result is compliant.

3. *Policy Enforcement*: Since we want to exchange information between the involved layers we need to be able to convey the labeled data between layers and effectively authorize its communication.

3. ARCHITECTURE

This section presents a framework that addresses the issues presented in Section 2. It enforces security policies at multiple layers, application, VM, VMM and network, and includes mechanisms to ensure that the policies are compliant with security goals, defined by the host domain over the application. In this 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. However, we focus primarily on the client (browser) side of web applications in this paper. 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 [13, 21] or user-level reference monitors [22, 11, 15].

The system design primarily consists of two distinct phases:

- **Configuration Phase**: This phase involves bootstrapping a new VM for a web application with appropriate security policies to ensure compliance with the host domain's application security goals at each layer.

- **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 end-to-end for that application.

3.1 System Design

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; it maintains information about all applications hosted within an organization and (3) a *web application VM (or Browser VM)* in which the sandboxed application runs. The idea is that the application authority defines an application policy that the Loader VM installs at the VMM level and at the web application VM. The web application VM is sandboxed 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 Figure 2 shows different services in our system design and how these services map to each of the components mentioned above.

The Loader VM defines three services for configuring and running sandboxed VMs: (1) a *VMLoad service*; (2) a *policy store*; and (3) a *label mapper*. The *VMLoad* daemon is responsible for bootstrapping new Browser VMs to serve the web application and ensuring compliance between application and system policies. The VMLoad service authorizes requests to load a new web application VM, it identifies the web application corresponding to the URL, obtains the web application policies from the policy store, configures and starts the new VM, and ensures that the VMM enforces the inter-VM communications according to these policies.

The policy store is the policy engine that generates and maintains the host domain's application policies. It services requests for (1) mapping URLs to web applications; (2) generating policies for specific instances of web applications; and (3) generating certificates for authenticating IPsec tunnels. The policy store can be considered as a local copy of the host domain's application authority, which defines the policies for all clients in that domain. The policy store caches such web application policies and retrieves the policies from the application authority on a cache miss. On a similar note, the policy store also acts as a local copy of the certificate authority and is responsible for generating and signing network certificates for authorizing connection requests, if trusted (e.g., based on hardware attestation), or passes such requests onto the application authority.

The label mapper is a trusted service that maps object identifiers for distributed objects (e.g., URLs) into security labels to ensure correct enforcement. This solves the problem that the host may not know the labels of objects defined externally to the system. Each of the VMs, including the Loader VM, has a local copy of the label mapper serving mapping requests for such remote objects. The label mapper caches two kinds of policy. First, there is a labeling policy that maps URL objects to security labels. The labeling policy maps regular expressions for the object names to labels. Second, there is a miss-handler policy that identifies where labeling requests should be forwarded given a miss in the URL mapper. The miss-handling policy maps regular expressions to the server to send the response and also defines limits on the label responses (e.g., secrecy and integrity ranges) for that server.

The web application VMs include two additional services: (1) an *update daemon* (called `update-d`) that receives requests from the VMLoad service to update application policies on the local VM and (2) a *SIESTA service* [7] that evaluates compliance of application and VM security policies. The update daemon accepts dynamic policy update requests from the VMLoad service 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.

3.2 Configuration Phase

As mentioned, this phase is responsible for loading a new VM to support a particular web application. A web application VM may encounter a web page/object that it is not authorized to serve. This

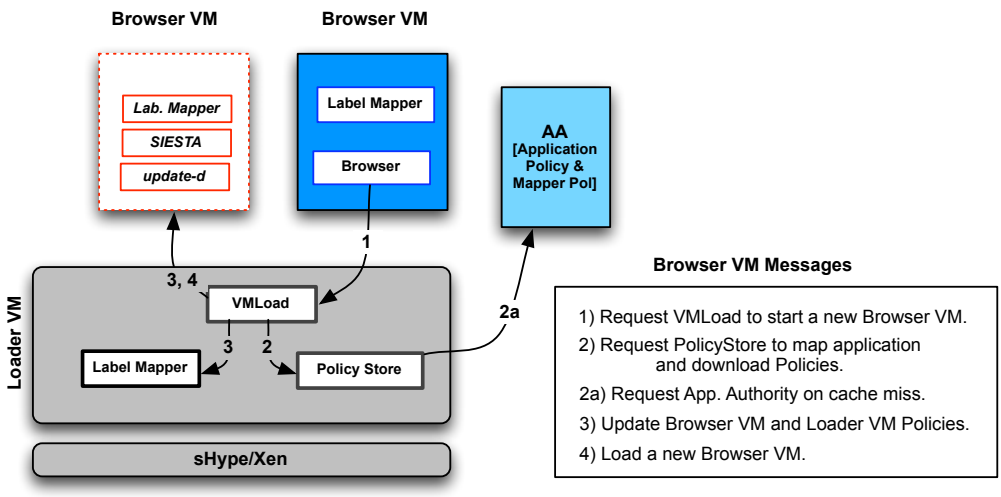

Figure 2: Virtual Machine System Architecture showing the interaction of components during the configuration phase.

situation could arise because:

- the web page/object requested by the user does not belong to the web application currently served by the VM.
- the web page/object requested does not fall within the secrecy/integrity range of the web application VM.

In such cases, we would like to open a new web application VM that can service this request. Configuring this new application VM correctly is the focus of the configuration phase.

The Figure 2 shows how different system components interact with each other to load a new web application VM. Below we describe the steps to configure and load a new web application VM,

1. The VMLoad service authorizes the request based on the security policy enforced at the Loader VM, i.e. whether calling VM is allowed to serve the application, if there is a transition involved then whether that transition is allowed for the calling VM, etc. If the request is authorized, the service locates the Browser VM image from a pre-defined location in the local file system.
2. The VMLoad service queries the policy store to identify the web application corresponding to the URL object and to define the application identity and the secrecy/integrity range for that identity. The secrecy/integrity range of the identity determine the Browser VM's secrecy/integrity range and web application to run.
3. The VMLoad daemon requests the policy store to download necessary policies to support the new web application. It requests,
   - Mandatory Access Control policies for Loader VM and Browser VM, to authorize system access to application objects.
   - Network Policies to setup secure tunnels, to convey the security label.
   - Mapping Policy for the label mapper, to support URL mapping for the web application.
   - Certificates to authorize the setup of secure tunnels with the destination.
4. The policy store generates application specific policy modules from appropriate templates and sends them to the VMLoad daemon.
5. The VMLoad daemon updates the Browser VM image directly or sends a request to the update daemon running inside the web application VM. It also updates the Loader VM with the downloaded policy modules in order to support new application specific types and secure tunnels for inter-VM communication control (e.g., SELinux policies in Xen's dom0). In addition to these policies, the VMLoad daemon also updates SIESTA specific configurations to set the secrecy/integrity range of the browser instance.
6. Finally, the VMLoad daemon loads the new Browser VM and starts up the browser instance to serve the web page/object request.

3.3 Enforcement Phase

This phase is mainly responsible for enforcing end-to-end mandatory access control on the web application. Each web application VM serves a single web-application and is confined to a pre-defined secrecy/integrity range.

1. The browser receives the request to load a web page/object, retrieves the security label of the web page from the label mapper, and assigns the label to the socket that will be used to retrieve information from the web server. The Figure 3 shows how label mapper interacts with different components.
   - the label mapper in the VM checks its local cache for the mapping. On a cache-miss, it forwards the request to the label mapper specified in the miss-handling policy.
   - if no label mapper can resolve the label of the web page/object, the label mapper returns a default security label, which in our case is a low secrecy/low integrity label.
2. The Mandatory Access Control policy installed in the VM operating system authorizes the communication based on the

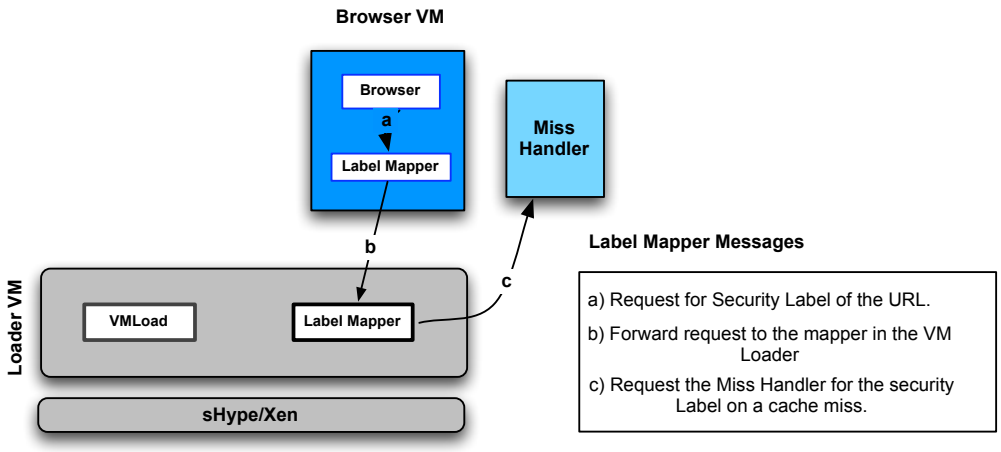

Figure 3: Virtual Machine System Architecture showing the *label mapper* messages during run-time.

socket's security label and establishes a secure communication channel, to convey the label, between the web application VM and the VMM (which is also the Loader VM currently).

3. The VMM determines the security label of the inter-VM communication and authorizes the communication based on the system policy installed during the configuration phase

4. If authorized, the communication is forwarded on the secure communication channel, to convey the security label and protect the secrecy/integrity of the communication, to the web server. Communication is authenticated using certificates generated by the policy store (or application authority, depending on the trust model).

4. POLICY COMPLIANCE

We want to enforce end-to-end information flow policies by integrating enforcing mechanisms at several layers. Since we have independent tools at every layer we need to ensure that each layer enforces the system's security goals, which we call the *policy compliance* problem. In our architecture, we need to verify policy compliance at two layer transitions: (1) between the application policy and the operating system policy in the web application (Browser) VM and (2) between the web application VM policy and the overall system policy stored in the VMM.

We say that one policy (e.g., an application's) is *compliant* with another policy (e.g., a OS's) if all the information flows authorized by the first policy are also authorized by the second policy. If so, then the reference monitor enforcing the first policy cannot permit an illegal flow of information relative to the second policy. In previous work, we developed SIESTA service [7] to check compliance of a program before it can be executed to ensure that every program executed complies with an OS policy.

Compliance testing is necessary for a class of programs, called *trusted programs*, that may be entrusted with responsibilities in enforcing the system's policy. A program is said to be *trusted* if it is given permissions that enable it to create an information flow that would violate the system policy, but it is trusted to prevent such a flow. While trusted programs are often trusted blindly in current systems, the emergence of application-level reference monitors [22, 11, 15] and security-typed languages [13, 21] now provide a basis for a program to control its information flows.

4.1 Application Compliance

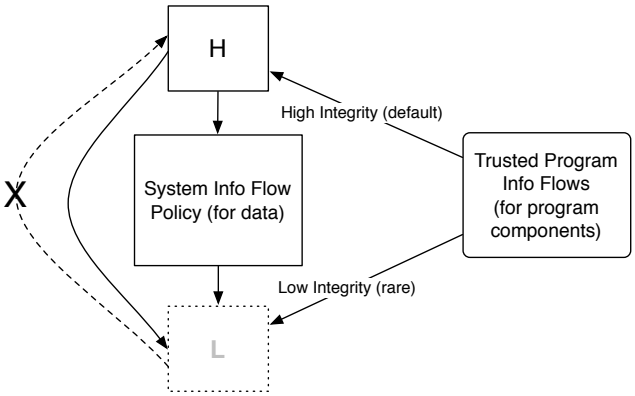

Figure 4: System data must not flow into Trusted Program components (e.g., configuration files and executables), so they have a natural integrity relationship *Program Integrity Dominates System Integrity* (PIDSI).

However, operating system and trusted program policies are developed independently, meaning that they cannot be directly compared because of differences in access control models, semantics and/or name spaces. Manual mapping is an option, but it is not scalable. To overcome this problem we observed that trusted programs and their components (e.g., configuration files and executable) usually are higher integrity than the data upon which the programs are applied. As shown in Figure 4, the system security policy defines the legal information flows for the data in the system, whereas information should normally flow from trusted program components to the data upon which they are applied. Intuitively, a program's code and configuration should not depend on the data in which it processes, but the output data necessarily depends on the code used to generate it. As a result, we propose The PIDSI (Program Integrity Dominates System Integrity) [18] approach where trusted program objects are assigned a higher integrity level than the system data, which the trusted program processes. This assignment results in a simple integration of program policy and system policy that enables automated compliance verification of trusted programs.

Using PIDSI, verifying policy compliance of a trusted program

policy (e.g., browser) with the application VM policy works as follows. The SIESTA service is started 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. SIESTA ensures that the given trusted program (e.g., the browser) is executed on the system only if the system and application policies comply: (1) the browser falls within the secrecy/integrity range of the VM (i.e., no flows to illegal labels in system) and (2) does not permit illegal indirect flows between system policy labels (i.e., does not permit a flow from data to a trusted program component directly, which is prevented by construction, or indirectly through any uncommon low integrity program objects). By checking this we guarantee that the browser and the Browser VM system policies comply.

4.2 VM Compliance

In this work, we introduce the policy compliance between a VMM policy and its application VMs. As the PIDSI approach helped us configure program policies to enable simplifications in compliance testing, we also want to configure application VMs to comply with VMM policies. The advantage here is that the application VM starts as a blank slate, so it can be configured to comply. The challenge then is to provide an architecture for configuring VMs based on the functional and security goals of the application.

In our architecture, the VMLoad service obtains dynamically-generated application policies for access control, network security, labeling, etc. from the policy store. The policy store uses policy templates as a basis for generating the access control and network policies, as these must include the network identity (i.e., IP address) of the server. The architecture uses IPsec certificates to verify that the servers are authorized to participate in the application. Since the VMM and VM policies are downloaded together, they are presumed to be compliant by default. This compliance can be verified offline, as the information flows are independent of the servers implementing them.

The label mapper policies for labeling and miss handling do not need to be specialized at runtime at present. We use the DNS names for servers, and use IPsec certificates to justify their participation in the application, as described above. An important issue is that objects in the VM may already be labeled prior to this execution. If a malicious system mis-labels some secret data as public, then the VM could leak it despite using a compliant policy. We do not assess whether a prior labeling of VM objects is correct at present. This problem is similar to verifying the labels of imported objects in system assurance.

The policy store retrieves the policies from an application authority. The question is whose authority is represented in the application authority: the client's, the server's, or both. In our applications, we are concerned about inadvertent leaks or misuse of low integrity data by our clients, so our application authorities represent the client administrative domain's view of the policy. Where the server only provides data of one label, it only needs to use the client's labels in Labeled IPsec communication [8]. Where the server may choose among multiple labels, the server is trusted to choose the appropriate label. We imagine that an offline process is necessary for client domain administrators to agree with application servers on such labeling. Further investigation on this topic is required.

5. IMPLEMENTATION AND EVALUATION

We implemented a prototype according to the design presented in Section 3. In this prototype we use a Java-based web browser, Lobo [16], whose core components are re-written using Jif (Java Information Flow) [13], to enforce mandatory access control within the browser. Jif ensures application-level enforcement of mandatory access control policies. We also use Xen virtual machine monitor to sandbox browser instances and web servers. Finally, we use SELinux as the guest operating system inside the virtual machines to ensure enforcement of mandatory access control policies at the operating system layer and empower the management domain (dom0) of Xen with functionalities of the *Loader VM*.

In this section, we demonstrate the prototype functionality with an information flow-aware web application that may be used to share sensitive information. The web application is based on the Open Source Intelligence [24] model for processing information. In that model an *analyst* gathers information about a particular *area of interest* from different sources, both trusted and untrusted.

In our prototype the web pages that the *analyst* uses for finding and analyzing information are secret. Additional sources the analyst visits for gathering information may be secret also, but some may be public, "open" sources, which are therefore, public. The secret and open sources are hosted in separate web server VMs in our prototype. The main security goal of this web application prototype is to enforce information flows, so that secret information is not leaked despite the analysts being able to view secret and public information together (e.g., via a mashup).

5.1 Configuration Phase

The privileged Loader VM is the key component responsible for configuring and loading new Browser VMs to support the web application. We implemented Loader VM as a collection of Linux-based services running in dom0 (the Xen management domain).

On system startup, the analyst requests the VMLoad daemon in dom0 to load a new Browser VM to access the homepage of his trusted web server. The VMLoad is a network daemon running in dom0. It is dedicated to bootstrapping new Browser VMs and ensuring compliance. Currently, VMLoad authorizes the request based on the IP address. In our current implementation, only Browser VMs, with internal IP addresses are allowed to send the request. VMLoad queries the policy store to retrieve web application details corresponding to the URL object. The VMLoad also requests the policy store to download necessary IPsec and SELinux policy modules.

The policy store is a network daemon running in dom0 and acts as a local cache for the global application authority. The policy store is responsible for mapping URLs to web applications, generating and maintaining IPsec policies, IPsec certificates, SELinux policy modules, and label mapper policies. In the current implementation, the policy store maintains different templates for IPsec policies and appropriately updates them based on the type of policy requested. This example shows an IPsec policy template for the Browser VM.

```
spdadd <src> <dest> any -ctx 1 1 < context> -P
        out ipsec esp/tunnel/ <src> - <dest> /req;
spdadd <dest> <src> any -ctx 1 1 < context> -P
        in ipsec esp/tunnel/ <dest> - <src> /req;
```

The policy store also maintains a database containing details about the web application such as the list of web servers (hostnames), the MLS-range, etc. In the table given below, *ws3* serves URL objects belonging to *analyst* application within the mentioned MLS range from sensitivity level 0 (s0 in SELinux) to 5. However, *ws1* and *ws2* serve data in the MLS range from sensitivity level 6 to 15.

```
table = ('analyst', s0-s5,'1' ,'ws3'),
        ('analyst', s6-s15,'2' ,'ws1', 'ws2'),
        ('DEFAULT', '0')
```

The policy store gets the list of web servers from the table and updates the template to generate the required policy. In the case of the analyst application, the Labeled IPsec policy should allow the Browser VM to access the web server containing the analyst's secret profiles (s6-s15) and search other "open" web servers (s0-s5).

The SELinux policy modules for the new Browser VM and the dom0 corresponding to the web application are maintained separately by the policy store. In our implementation, the policy store also acts as the certificate authority and is responsible for creating and signing IPsec certificates to support the web application. Our implementation uses *openssl* to generate and sign certificates. The policy store creates a new key for each of the web servers and signs it. The label mapper policies for the Browser VM and the dom0 are maintained in the folder corresponding to the web application. On request, policy store retrieves the necessary mapper policies and sends them to the VMLoad service.

The VMLoad daemon checks compliance between the downloaded SELinux policy modules and the overall system policy using SIESTA as described in Section 4. Finally, it updates the virtual machine image with these policies and loads a new Browser VM using the *Xend* daemon. After the basic VM services are loaded, the SIESTA daemon verifies compliance and then loads the browser instance at the specified secrecy/integrity range.

5.2 Enforcement Phase

Our implementation uses Labeled IPsec tunnels [8] and SELinux policy modules in the Browser VM and Loader VM to enforce and extend the mandatory access control policies implemented at the application and operating system layers.

When the Jif browser tries to load the homepage of the analyst, it retrieves the security label of the URL of the homepage from the local label mapper and assigns that label to the socket used for the communication.

The label mapper is a Linux-based daemon running in dom0 and in each one of the Browser VMs. It caches mappings of URLs to security labels and the VMLoad daemon updates the necessary mapping policy while loading the web application VM. (the policy store provides the mapping). The following example shows one of the entries that map a given URL to a defined security label (recall that the object name can be a regular expression).

```
http://ws-vm1.cse.psu.edu/public.html --
                    root:sysadm_r:analyst_t:s4
```

In our prototype, the label of the URL identifies the web application to which the object belongs (unless it is a default label, *high* or *low*). If the communication is authorized, the kernel subsystem checks the IPsec policy and creates a Labeled IPsec tunnel between the Browser VM and the Loader VM; in our case it is the dom0. The kernel subsystem in dom0 extracts the security label of the communication from the labeled IPsec tunnel, authorizes the communication based on the label, and extends the tunnel segment to the destination dom0 to convey the security label. Furthermore, the IPsec tunnel segment with the destination dom0 is authorized based on the IPsec certificates installed during the VM load.

To summarize, we establish three orthogonal labeled IPsec tunnels to convey the security label of the browser communication to the web server at the other end: 1) between Browser VM and its dom0, 2) between source and destination dom0s, and 3) between the web server VM and its dom0. The Labeled IPsec tunnels between the VMs and its dom0 is used only for conveying the security label of the browser socket.

Finally, the SELinux policy at the web server has to authorize the browser access before sending the requested web page/object. The *ipsec-tools* package is used to configure and setup IPsec security policies. The *setkey* tool is used to maintain the policies and *racoon* is the IKE daemon used for negotiating security associations.

The IPsec and SELinux policies should allow the analyst to access the secret web server data. Every time the analyst tries to access a URL with different security label, a new series of Labeled IPsec tunnels is created to convey the new security label.

When the browser tries to access an object from an untrusted web server, the security level does not fall within the secrecy range of the local VM and hence SELinux policy prevents that action. When it tries to access a different web application, the system will throw an error due to lack of Labeled IPsec policy. At present, both errors will result in the browser sending a request to the VMLoad service to load a new Browser VM for that URL.

5.2.1 Pre-Loaded VM

In the initial implementation, the VMLoad daemon loads a new Browser VM every time it receives a request. The latency to load a new VM from scratch is high and may not be acceptable in some web-based environments. Also this approach will result in redundant Browser VMs as there is no mechanism to forward requests to existing virtual machines. To mitigate these problems we propose that the implementation use pre-loaded VMs.

Three kinds of VMs can be running in the system at any point of time: (1) Idle VMs that are not currently serving any application, (2) VMs that are already serving a particular application but ready to accept further requests and (3) VMs that are already serving an application and not ready to service further requests.

The VMLoad daemon maintains a table with details about the VMs in the system, their status, the web application identity, and other details such as secrecy/integrity range.

```
table = (1, <VM InternalIP>', 'analyst', 's0-s5',
            'SERVING', 'ACCEPT', <VM Global IP>),
        (3, <VM Internal IP>, '', '',
            'IDLE', 'ACCEPT', <VM GLobal IP>)
```

In this setup, the VMLoad daemon first queries the table for VMs that are already serving the web application and ready to accept new requests. If it does not find one, it looks for idle VMs and forwards the request, instead of loading a new VM.

The update daemon running in the VMs uploads any policy updates for an idle VM. It is also a Linux-based network daemon that retrieves IPsec, SELinux policy modules, and system configurations from the VMLoad daemon and updates the VM to control the web application.

5.3 Evaluation

This section mainly evaluates two aspects of our implementation: (1) effectiveness of our prototype in providing the security guarantees required by the application and (2) system performance when loading a new URL.

5.3.1 Effectiveness

One of the main security goals of this analyst-based application is to clearly isolate secret and public web sites. We differentiate secret and public 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 corresponding 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.

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 web servers, at different secrecy ranges but it is not safe to allow content from low integrity servers to be accessed where secret data (i.e., above s5) is being accessed. A new Browser VM would be initiated for the low integrity pages at these secrecy levels.

5.3.2 Browser Startup Latency

In our prototype implementation, we sandbox browser instances inside a virtual machine. The VM creation happens whenever the user wants a access a URL 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		Latency
From Scratch	Configure/Load Policy	4.90 seconds
	Loading Browser and Page	18.50 seconds
Pre-Loaded VMs	Configure/Load Policy	0.90 seconds
	Loading Page	2.80 seconds
Native Browser	Cold-start Loading Page	7.45 seconds
	Warm-start Loading Page	1.50 seconds

Table 1: Browser Startup Latency

Table 1 shows the cost (in seconds) of: (1) loading a web page in a new Jif Browser VM from scratch; (2) loading a web page inside an idle Browser VM; and (3) loading a web page in a native Firefox browser in dom0. For the Browser VM measurements, we break the measurement into the two phases, configuration and enforcement (i.e., loading the browser and page). The *From Scratch* entry of the table shows the cost incurred by these phases while loading the web page inside a new Virtual Machine. We loaded the VMs several times to ensure repeatability. The prototype implementation is not yet optimized and hence the performance results should be considered as an upper bound on the overhead. Nonetheless, 20 plus seconds is considered too expensive. The Tahoma prototype consumes approximately 10 seconds, but has fewer configuration tasks [4]. We believe it is possible to 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.

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 VMs, see Section 5.2.1. The *Pre-Loaded VMs* entry in the table 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 which are eliminated when we send the policies to the update daemon in this case. With pre-loaded VMs it took less than 3 seconds to load a new web page after configuration.

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 and (2) the "warm-start" latency of launching the browser, assuming it has been previously launched. From the performance values provided in the table, we can gather that with further optimizations it is very much possible to reduce the browser load latency of our prototype on par with the "warm-start" latency of native Firefox browsers.

6. CONCLUSIONS AND FUTURE WORK

Currently, virtual machine environments do not provide adequate mechanisms to configure flexible security policies for VMs and the other layers that are involve in security enforcement. In this paper, we presented an architecture to address this issue. Our architecture enables administrators to set up secure communications between targeted virtual machines based on application templates that can be configured automatically at runtime. We implemented such architecture based on tools that allow us to express and enforce mandatory access controls at operating system, virtual machine monitor and network layers and to convey security information between them.

We evaluated our implementation using a simple web application. It establishes secure channels between targeted virtual machines across different layers (OS, VMM and network) according to a given configuration. We have also evaluated the response time, in the context of our test application, to load a new VM. We have introduced the concept of pre-loaded VMs to reduce the latency to load a new VM. Our evaluation shows that despite the overhead imposed by different components, the proposed system incurs only nominal performance overhead.

As future work, we will examine other applications in order to test the robustness of our approach. Further, we plan to explore ways to reduce the time to load a new virtual machine and using other mechanisms to label network communications and convey security requirements between virtual machines.

7. REFERENCES

[1] R. J. Adair, R. U. Bayles, L. W. Comeau, and R. J. Creasy. A Virtual Machine System for the 360/40. Technical Report 320-2007, IBM Corporation, Cambridge Scientific Center, Cambridge MA, USA, May 1966.

[2] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web application via automatic partitioning. In *SOSP*, pages 31–44, 2007.

[3] G. Coker. Xen Security Modules (XSM). `http://www.xen.org/files/xensummit_4/xsm-summit-041707_Coker.pdf`, April 2007. These are presentation slides from the 2007 Xen Summit.

[4] 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.

[5] S.W. Devine, E. Bugnion, and M. Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. VMWare, Inc., October 1998. US Patent No. 6397242.

[6] HP NetTop: A Technical Overview. Available at: http://h71028.www7.hp.com/enterprise/downloads/HP_NetTop_Whitepaper2.pdf, 2004.

[7] B. Hicks, S. Rueda, T. Jaeger, and P. McDaniel. From trusted to secure: Building and executing applications that enforce system security. In *Proceedings of the USENIX Annual Technical Conference*, 2007.

[8] 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.

[9] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A retrospective on the vax vmm security kernel. *IEEE Trans. Softw. Eng.*, 17(11):1147–1165, 1991.

[10] M. Lageman. Solaris Containers – What They Are and How to Use Them. http://www.sun.com/blueprints/0505/819-2679.pdf, May 2005.

[11] G. McGraw and E. Felten. *Java Security*. Wiley Computer, 1997.

[12] R. Meushaw and D. Simard. Nettop - commercial technology in high assurance applications, 2000. http://www.vmware.com/pdf/TechTrendNotes.pdf.

[13] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java + information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

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

[15] Oracle database. http://www.oracle.com/database.

[16] The Lobo Project. Lobo: Java Web Browser. http://lobobrowser.org/.

[17] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium*, August 2003.

[18] Sandra Rueda, Dave King, and Trent Jaeger. Verifying the compliance of trusted programs. In *Proceedings of the 17th Annual USENIX Security Symposium (USENIX '08)*, Aug 2008.

[19] John Rushby. The design and verification of secure systems. In *Eight ACM Symposium on Operating System Principles*, 1981.

[20] 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.

[21] V. Simonet. The Flow Caml System: Documentation and User's Manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), July 2003. ©INRIA.

[22] E. Walsh. Application of the Flask architecture to the X Window System server. In *Proceedings of the 2007 SELinux Symposium*, March 2007. Available at http://selinux-symposium.org/2007/agenda.php.

[23] 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.

[24] Wikipedia. Open Source Intelligence. http://en.wikipedia.org/wiki/Open_source_intelligence.