An Architecture for Enforcing End-to-End Access Control Over Web Applications

Boniface Hicks, Sandra Rueda,
Dave King, Thomas Moyer, Joshua Schiffman, Yogesh Sreenivasan,
Patrick McDaniel, Trent Jaeger
Systems and Internet Infrastructure Security Laboratory
Department of Computer Science and Engineering
The Pennsylvania State University
boniface.hicks@email.stvincent.edu,
{ruedarod, dhking, tmmoyer, jschiffm, sreeniva, mcdaniel, tjaeger}@cse.psu.edu

ABSTRACT

The web is now being used as a general platform for hosting distributed applications like wikis, bulletin board messaging systems and collaborative editing environments. Data from multiple applications originating at multiple sources all intermix in a single web browser, making sensitive data stored in the browser subject to a broad milieu of attacks (cross-site scripting, cross-site request forgery and others). The fundamental problem is that existing web infrastructure provides no means for enforcing end-to-end security on data. To solve this we design an architecture using mandatory access control (MAC) enforcement. We overcome the limitations of traditional MAC systems, implemented solely at the operating system layer, by unifying MAC enforcement across virtual machine, operating system, networking and application layers. We implement our architecture using Xen virtual machine management, SELinux at the operating system layer, labeled IPsec for networking and our own label-enforcing web browser, called FlowwolF. We tested our implementation and find that it performs well, supporting data intermixing while still providing end-to-end security guarantees.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and protection; D.4.6 [**Operating Systems**]: Security and Protection—*access controls*

General Terms

Security, Virtual Machines

Keywords

Access Control, Xen Security Modules, Policy Compliance

1. INTRODUCTION

The web has evolved into a general purpose distributed computing platform. Wikis, bulletin board systems, collaborative editing environments, search engines, calendars, and many other web applications handle data that is obtained from and shared with large bodies of loosely associated hosts. In a wiki, for example, data originating from one user may be stored on a web server and then mixed with another user's data within that user's web browser. This rich intermixing of data from many sources allows users to create new, innovative models of data sharing, e.g., mashups.

A cost of this intermixing of data is client and server vulnerability to a broad milieu of attacks. For instance, XSS attacks that execute malicious scripts on an unsuspecting user's browser. Scripts that corrupt or leak sensitive data (passwords, credit cards, etc.) Such a vulnerability is a consequence of a fundamental limitation of the existing web infrastructure: existing environments provide no means for enforcing end-to-end security. As in the XSS attack, the lack of coordination between elements of the system allows the adversary to abuse the system—the fact that the malicious script originated from a potentially dangerous input is not known by the user, i.e., it is delivered in the same "security context" as the legitimate content.

Efforts to secure web applications have historically focused on client-side solutions. These can be divided into two categories: inter-browser separation and intra-browser separation. On the inter-browser side are systems, such as Tahoma [6] or NetTop [19], that use separate virtual machines to separate data with different security requirements. On the intra-browser side are various systems such as OP's process-based separation of plugins [8] and Chrome's process-based separation of tabs, as well as more fine-grained approaches like the same-origin policy. While these solutions provide additional security and protect against some attacks on the browser side, none are able to coordinate the management of data that is *necessarily intermixed* within the same browsing context. What is needed is a way to integrate the security enforcement at each layer of the system—thereby ensuring that mixing is consistent with the security policies of all elements of the system.

Our solution is to use mandatory access controls (MAC) to enforce an end-to-end security policy on web application data. It is insufficient to follow the traditional approach to MAC, however, which focuses on implementing MAC only at the operating system layer. This approach is too restrictive, preventing the rich intermixture of data from various sources that must take place in the web browser. Rather, we find that by unifying MAC enforcement across the virtual machine, operating system, net-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SACMAT'10, June 9–11, 2010, Pittsburgh, Pennsylvania, USA.
Copyright 2010 ACM 978-1-4503-0049-0/10/06 ...$10.00.

working, and application layers we can achieve more effective and more flexible enforcement than the operating system alone.

In this paper, we provide an architecture that implements a multi-layer, mandatory access control system for enforcing end-to-end security goals on web applications. Data originating on the server side is labeled by the web application administrator and data originating on the client side is labeled by the user. The system is configured such that it propagates labels with data across each layer of the system—from the server to the Virtual Machine (VM) to the Virtual Machine Manager (VMM) to the network layer to the client and back again. Each layer is configured such that it consistently enforces policy on labeled data with respect to the other layers.

We implement our architecture using a Xen VMM, SELinux VMs, labeled IPsec and our own instrumented label-enforcing browser called FlowwolF, along with some other custom-built components for supporting policy distribution and distributed label mapping. The major challenges we had to overcome in the implementation were augmenting a web browser to enforce mandatory policies, defining comprehensive policies for web applications covering each layer, automatically distributing policies across layers to appropriate enforcement points and caching policies appropriately to achieve performance levels that would not diminish the user experience expected of web applications.

This paper makes the following contributions:

- We develop the first architecture for a multi-layer, mandatory access control web application system;
- We provide a working implementation of the system using commodity as well as custom components;
- We implement a label-enforcing web browser that gives and receives labels from the system;
- We instrument an open source bulletin-board messaging system such that it enforces end-to-end security goals when used in our system.

The rest of the paper is organized as follows. In Section 2 we present the problems we must solve when developing an end-to-end secure web browsing system. In Section 3 we give some background on mandatory access control systems with the requirements a system must fulfill to get the security guarantees these systems provide. We follow that in Section 4 by proposing an architecture using a mandatory access control system that solves these problems. In Section 5 we describe our implementation of the architecture. In Section 6, we evaluate the architecture. To do so, we instrumented a commodity bulletin board application. Finally we conclude in Section 8.

2. WEB APPLICATION SECURITY

To deliver rich, dynamic content, today's Web 2.0 applications combine data from sources with varying security requirements and render them in a single browser tab. Many popular applications such as Mashups and social networking sites allow other applications to read and write data or act as the user. If the applications trusted to perform these actions are malicious, they could leak or destroy sensitive data.

To illustrate, consider our instrumented Bulletin Board application running in a multi-level secrecy (MLS) environment [1]. In this case, the posted messages have security requirements that originate at the client side (in the browser). A user in the MLS environment is gicen a set of security labels when she logs into her OS (her clearance). These labels must be checked by the browser before allowing her to open a secret browser tab to read and write secret data. Before she can post, the browser must ensure she is posting from a secret tab and the content of her message was entered only from secret input fields (not replying to a top-secret message, for example). Furthermore, the system must ensure that the remote server she is posting to is secret, such that she can open a secret socket that is connected to a secret web server application running on a remote OS that protects its applications' secrecy. This opens up various attack vectors. We divide these attack vectors into three categories—network-layer attacks, OS-layer attacks and browser-layer attacks.

Network-layer attacks focus on the interception of secret data and hijacking connections to impersonate trusted authorities. One example, a man-in-the-middle attack (MITM), allows an attacker to act like a proxy for the browser, intercepting web requests and providing the browser with the content returned from the server. The attacker can then modify the message content or steal secret messages. More simply, if a network channel is not encrypted, it is subject to modification by injection or leakage by eavesdropping.

At the OS layer, secret posts stored on the Bulletin Board server or stored persistently in the web browser's system are vulnerable to attacks by any malware running on the system. Malware could modify the web browser and server application by modifying runtime libraries, leaking secure message data or causing other mischief that undermines the message security requirements enforced by web browser and server.

At the browser layer, there are myriad attacks that involve compromising the browser or confusing it (or the user) to behave incorrectly. If public and secret messages are loaded into the same browser tab, malicious scripts in public messages might leak secret messages to an attacker's site (i.e., *cross-site scripting*, or XSS). Other attacks like *cross-site request forgery* (CSRF), can trigger a request on behalf of the user without their knowledge. This script then uses the user's authenticated credentials to access secrets for the attacker. OWASP listed these as the most common browser vulnerabilities [25]. Another prevalent danger is *phishing*: innocent-looking public messages can solicit viewers to click on links that would send secret data as parameters to malicious sites or request that a user fill out a form that will post secret information to malicious sites. Next we describe the various models developed that employ isolation techniques to prevent data leakage and compromise.

VM-layer or inter-browser separation Approaches such as NetTop [19] and Tahoma [6] separate data sources of different security levels into individual virtual machines (VMs). It has the advantage of providing strong separation and preventing many of the attacks we have described. Browser level attacks like XSS and CSRF attacks are prevented because access to untrusted domains are prohibited. However, this approach reduces efficiency and violates our usability requirements. In our bulletin board example, a user with secret clearance would have to view different messages in different VMs rather than viewing multi-level messages in the same browser or even the same tab. Furthermore, a flexible use model would allow a single browser to access multiple trusted web applications without total separation such that they could modify each others' data in controlled, policy- driven ways. Finally, these approaches alone fail to protect against attacks at the network layer (like MITM) or the OS-layer attacks, like those from malware.

Intra-browser separation In order to maintain the usability advantage of displaying data from various origins in a single page, recent research attempts to separate data of different secu-

[1]Our approach is not specific to MLS, but it will require a mandatory access control policy, see Section 3.

Layer	*Protection State*	*Labeling State*	*Transition State*	*Enforcement*
Application	DLM restrictions	Data	Low to high secrecy	Control data leakage
VM	SELinux policy	Processes	Within VM label range	Prohibit loading illegal security level browsers
VMM	XSM policy	VMs	Spawning VMs	VM ranges limited to policy
Network	IPsec policy	Sockets	Single level	Prevents connection with insecure remote machine

Table 1: Elements of the mandatory protection system needed at multiple layers for end-to-end secure web systems.

rity levels using isolation policies within the browser. For example, the same origin policy prevents Javascript from one origin from modifying Javascript data originating elsewhere. Other innovative approaches refine this policy to prevent some attacks by requiring some mutual authentication [24], by marking up DOM content with accents [5], instrumenting the Javascript interpreter mediation points [17, 28], or performing client or server filtering to remove malicious code [27]. These approaches have the advantage of minimal modifications to the system, modifications to the client-side alone, and incremental deployment. A more robust approach to intra- browser separation introduces process separation to protect the browser from malicious plugins [8]. None of these approaches, however, provides protection against phishing, network-layer or VM-layer attacks and only provides partial protection against browser attacks.

Most of these intra-browser approaches suffer from not knowing precisely or confidently the security properties of web data. They impose a heuristic policy, presuming for example, that data from the same origin should have the same security properties. This assumption would be false like in our bulletin board example, which serves up both public and secret data on the same site.

None of the above approaches alone is adequate for systems that seek to enforce strong, end-to-end security goals while still combining data of mixed security in a single browser window. What is needed is a web application system that combines the previous two approaches, using inter-browser security techniques for protection against OS-layer and network-layer attacks as well as intra-browser protections for improved usability and protection against XSS, CSRF, and drive-by download attacks. To maintain security policies on data between server and browser, protection is needed at each layer: the application (intra-browser) layer, VM layer, VMM layer and network layer. Intra-browser separation can be improved by having authenticated security policies on secret, trustworthy data, and network-layer and OS-layer attacks can be prevented by enforcing mandatory policies on web application data and programs.

Designing such a web system introduces a host of challenges. One challenge is properly configuring the system to statically and dynamically label web content while propagating those labels faithfully. Another is carefully dividing up the label enforcement between layers while ensure security requirements are maintained end to end. Furthermore, this must be done efficiently.

3. END-TO-END ENFORCEMENT

We claim that a system provides secure end-to-end enforcement for an application if a mandatory access control (MAC) policy is enforced consistently across all software layers. Anderson defined the *reference monitor concept* [1], which states the guarantees that must be satisfied to enforce a MAC policy correctly. We propose the construction of a multi-layer reference monitor for end-to-end enforcement. Table 1 shows the system layers (in rows), the MAC policy concepts (in columns), and the requirement assignment of MAC policy to layers (in each cell). Our task is to define a multi-layer reference monitor that enforces a coherent system-wide MAC policy and demonstrate what is necessary to build it correctly (Section 3.1). We also define a mandatory protection system, which motivates why a MAC policy is necessary for our multi-layer reference monitor and identifies the MAC policy concepts that must be enforced (Section 3.2).

3.1 Multi-Layer Reference Monitor

Table 1 shows four layers needed to enforce a MAC policy: (1) the *application layer* controls access to application objects (e.g., browser tabs and URLs); (2) the *VM layer* consists of the operating system that controls process (including the application) access to VM system objects (e.g., files and sockets); (3) the *VMM layer* (e.g., for Xen, its hypervisor and privileged host VM) controls inter-VM interactions (e.g., shared memory and communications); and (4) the *network layer* that authorizes communication and dictates how secure communication is performed (e.g., chooses cryptographic protocols). We state that there may be multiple *components* at each layer that are required to enforce a MAC policy, such as multiple application processes or multiple VMs. Each of the components that we depend upon to enforce a MAC policy must be part of the multi-layered reference monitor for that application.

A true reference monitor must satisfy [1]: (1) *complete mediation*—all security-sensitive operations must be mediated by the reference monitor; (2) *tamperproofing*—the reference monitor must be protected from illicit modification; and (3) *simple enough to verify*—the reference monitor mechanisms and policies must be verified to enforce site security goals correctly. A multi-layer reference monitor must ensure that the composition of layers satisfies these requirements. Below, we examine these requirements and the tasks that must be performed to satisfy them in a layered environment.

We leverage existing mediation in OSs (e.g., SELinux [23]) and VMMs (e.g., Xen sHype [7]), but we also require mediation for other layers and an approach for inter-layer communication of security information. For the application layer, we extend the browser application with mediation mechanisms that leverage OS labels. We use labeled IPsec [16] to authorize access at the network layer using system labels that specifies secure communication requirements. Additionally, we use the same set of system labels for all layers for consistence across layers.

Tamperproofing each layer is generally done by the layer below. For example, an SELinux policy must protect the browser process from tampering by other processes if it is to enforce its policy correctly. In prior work, we evaluated SELinux policies to ensure that reference monitoring processes cannot be tampered with by untrusted processes [**?**]. As the browser application is just another instance of a reference monitoring application, the same technique can be used, so we do not discuss this further. Ensuring that a VMM policy protects a VM from tamperproofing can be performed similarly.

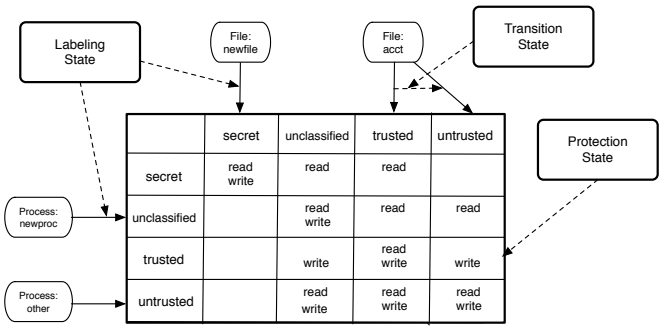

Figure 1: A Mandatory Protection System: The protection and transition states are defined in terms of labels and are immutable. The labeling state associates individual objects with labels and is mapped to new objects dynamically.

For the verifiability requirement, it is necessary to show that the multi-layer reference monitor implementation enforces the intended MAC policy. We claim that a monolithic MAC policy is enforced by a multi-layer reference monitor if: (1) each component in the multi-layer reference monitor meets the guarantees for complete mediation and tamperproofing, accounting for trust in its environment; (2) each policy decision by a component in the multi-layer reference monitor is the same decision as that would have been made by a monolithic reference monitor using that policy. First, we note that a component in a multi-layered reference monitor may only be entrusted with a subset of the MAC policy, depending on its environment and its level of assurance. For example, a conventional OS may not be trusted to protect itself from untrusted processes, so it must not be given access to high integrity data. Second, each component in the multi-layer reference monitor must be given a policy specification that enables it to make the same decision as the monolithic reference monitor on decisions it is trusted to make. We aim to achieve this claim by construction. To do so, we need to precisely define our policy model and how policy specifications can be distributed while enforcing the same semantics as the monolithic case.

3.2 Mandatory Protection Systems

The four columns of Table 1 indicate the three policy concepts that must be supported by each layer (A, B, and C) and types of enforcement decisions that that MAC policy enables (D). We state that an access control policy defines whether a particular subject (e.g., process) can perform an operation on a particular object (e.g., file and URL) based on a fixed set of security labels. In traditional discretionary access control models, access control is specified in terms of dynamically-created entities, such as files, and managed by processes. The dynamic nature of this model makes it intractable to determine whether a process may obtain an unauthorized permission (see the undecidability of the *safety problem* [9]). As a result, security-critical systems use MAC, where the system (e.g., an administrator and/or trusted system service) defines a fixed policy using an immutable set of security labels. While the security policy can be fixed, new processes and files are still created dynamically, so MAC policy includes concepts to maintain consistency between the dynamically-evolving system and the static access policy. Ultimately, a service that configures a MAC policy in a multi-layer reference monitor must be able to interpret these MAC policy concepts to deploy MAC policy across all layers to enforce end-to-end security correctly.

We state that our MAC policy is defined using a model, called for historical reasons [2], a *mandatory protection system* (MPS). An MPS is derived from the classical protection system model of Lampson [18]. As shown in Figure 1, an MPS consists of: (1) a *protection state* (e.g., an access matrix) that defines the operations that subjects can use to access objects (i.e., the access control policy); (2) a *labeling state* that defines the mapping between system objects (including subjects) and their MAC labels; and (3) a *transition state* that defines how the assignment of a MAC label to an object may be changed. A classical protection system also defines a set of *protection state operations* that may be used to change from one state to another, but as a MAC system only allows a trusted entity to modify policy no such specification for protection state operations is necessary.

An MPS protection state is the same as a traditional protection state, except that subjects and objects are defined in terms of a set of labels which are immutable. The protection state then defines the operations that subject labels can perform on object labels, rather than subjects and objects directly.

Since new objects may be created and must be assigned a label, we need a *labeling state* to define the rules for mapping a new object (e.g., process or file) to its label. When `newfile` is created, it must be assigned one of the object labels in the protection state. In Figure 1, it is assigned the `secret` label. MAC models must include labeling state. For Bell-LaPadula models [4], the labeling state is implicit, as the labels of new objects inherit the label of the creating process (although writeup is possible). For SELinux [23], policy rules define exceptional labeling requirements, such as labeling a file based on the directory in which it is created.

The MPS also defines when an object's label is changed via a *transition state*. A transition state changes label of a process, thereby altering the its protection domain which defines the permissions it can access. For example, SELinux defines rules that change the label of a process on `exec`. This mechanism enables the appropriate permissions to be assigned when a program is run (e.g., to control `setuid`). The transition state also describes label transitions on objects.

Our challenge is to develop an architecture where we can distribute MAC policy in terms of these concepts among the components in each layer, such that the original, monolithic policy is enforced correctly. Table 1 shows that the VM Layer is assigned a protection state that is an SELinux policy specific to running that application securely. If the goal were only to implement a single application, a manual policy configuration would suffice, but for a general purpose system, automated configuration is necessary. Therefore, we propose that an architecture (i.e., set of services) is necessary to take a MAC policy for the application, identify and configure components in the multi-layer reference monitor, and provide supporting function to ensure that the enforcement is consistent with a monolithic MAC policy.

4. END-TO-END SECURITY ARCHITECTURE

Here we provide an architecture for a multi-layered MAC system for web applications that solves the problems presented at the end of Section 3.

[2]In this context, the "system" in a *mandatory protection system* is the code that manages the permissions assigned to subjects (processes) and objects (files). It does not refer to a reference monitor or system at-large. In fact, the reference monitor enforces the policy created using a mandatory protection system.

4.1 Components

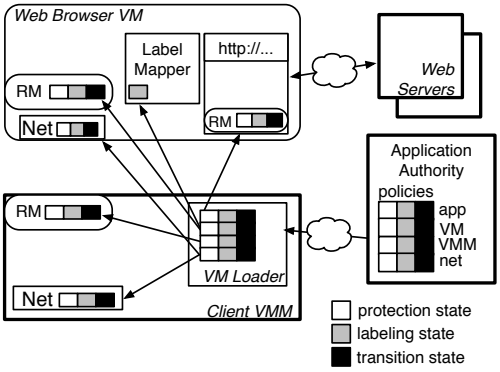

Figure 2: A Multi-layer MAC Architecture enforces a single MAC application policy across multiple reference monitor components. Upon starting a new Browser VM, (1) the Application Authority sends the application's policies to the VM Loader, which then (2) installs the policy into the VM and VMM, networking and BrowserLE reference monitors. (3) The BrowserLE retrieves URL labels from the Label Mapper as needed and communicates securely with the application's server.

Our system, illustrated in Figure 2, consists of an Application Authority, which stores monolithic mandatory policies for one or more web applications, and a Client VMM, which loads application VMs that it runs under the MAC policy. The Client VMM has a VM Loader that distributes a monolithic policy to the appropriate layer's reference monitors, including its own. Figure 2 shows a Browser VM that hosts a label-enforcing browser, called a BrowserLE. The Browser VM is instantiated by the VM Loader with MAC policy for its reference monitor and the browser's reference monitor. The Browser VM also has a Label Mapper that knows where to find the authoritative labeling state for the application's web resources (URLs), described below.

The monolithic policy is distributed from the Application Authority to the layers and components that will enforce it. The protection state, labeling state and transition state for each layer is installed into that layer's reference monitor, as depicted in Figure 2. The VM Loader has to instantiate network policy itself. For example, we only know the IP address for IPsec policy when the server is contacted. The network system uses IPsec certificates to authenticate that the web server at that IP address is valid.

The mapping of object identifiers (e.g., file names and URLs) to labels is not stored in the reference monitor. Each application VM is installed with a *label mapper*, which determines the authority for labeling an object identifier and retrieves the label from that authority. For example, the label mapper can retrieve the label of a file from the local file system should the user request file access (e.g., via `file://`).

Since not all web application data will be available a priori, the web server may be an authority for labeling URL objects. For example, when a dynamic page is constructed from data retrieved from a database, the web server can use the data's labels from the database to suggest a label for the data. The label mapper enforces policy from the Application Authority to limit the scope of labels that an authority can suggest. For example, a web server may be limited to serving only confidential and public data. Note that if a web server is only authorized to serve only one label, then the web server need not be queried for that label.

4.2 Protocols

Here, we define the main protocols executed by our architecture. These protocols aim to implement the required tasks listed at the end of Section 3. First, the architecture must disseminate the MAC policy across each relevant component, constructing policy enforcement that is consistent with the overall MAC policy. Policy enforcement occurs when a Browser VM makes a data request. In some cases, the policy may stipulate a browser transition when such a request is made. We give protocols here for initial policy configuration, URL processing and browser transitions.

To make the protocol descriptions more concrete, we draw on the bulletin board application we instrumented to work with our system. The instrumented application enables users to create message threads with security labels. Our BrowserLE displays those message threads in a single browser tab only so long as the tab's label dominates the thread's label (i.e. no read up). In this way, the browser can display messages with various security requirements in the same tab and at the same time ensure secret data is not leaked. Figure 4 presents a view of the instrumented bulletin board (IBB).

4.2.1 Initial Policy Configuration

A policy configuration is triggered when a browser loads a new web application. If there is not already a Browser VM for this application, the VM Loader must load one and install the application MAC policy in the VM and VMM as follows:

1. To start a new web application, a URL must be sent to the VM Loader. The VM Loader looks up the application's Authority and retrieves the application's complete policy. An example policy for the IBB application is shown in Table 2.
2. The VMM and networking policy are installed in the VMM's own reference monitor and in its networking subsystem.
3. If a VM is already running, it can be updated with the application's VM and network policies. Otherwise, the VM is configured with the policies and then started up by the VM Loader. If the VM is newly started, it is assigned a label based on the VMM labeling state.
4. The BrowserLE protection state, labeling state and transition state are configured according to the application's policy. The BrowserLE is started by the VM and assigned a label determined by the VM labeling state.
5. The BrowserLE is sent the URL that was initially sent to the VM Loader and it processes that URL within the newly configured system (see Section 4.2.2).

4.2.2 Processing a URL

The BrowserLE does not have a priori knowledge of the label state for a given web application. Thus, each time a URL is requested, it must load that URL's label from the URL Label Mapper. Here, we illustrate how our architecture carries out URL processing, starting with looking up the URL label. In the process we show how policy is enforced with the reference monitor distributed over multiple layers. For correctness, the system must authorize communication requests from the browser VMs and propagate security requirements to support end-to-end information-flow enforcement.

1. A running instance of the browser receives a request to load a new URL. In the case of the IBB, this could be a new message thread. The browser queries the Label Mapper to find the labeling state for the particular URL. For instance: `GETLABEL http://website/bulletinb/showthread.php?tid=5`.
2. The Label Mapper has been configured to know where to find the label for the given URL. It queries the appropriate authority (e.g., the web server holding that resource) and returns the label associated with the URL, e.g. `Label: TopSecret`. The label mapper stores a mapping between the server identifier and the set of labels that that server can suggest for the application.
3. The BrowserLE checks to determine whether the current tab can read in data with the requested label (`TopSecret`). If not, a transition may be triggered (see Section 4.2.3).
4. Otherwise, the instrumented browser creates a socket connection with the proper label to retrieve the message thread data. As the network connection must pass through both the VM's and the VMM's networking subsystems, the requested label (`TopSecret`) must be allowed by their policies' protection state, as well.
5. If authorized at each layer, the communication is forwarded on a secure communication channel (labeled IPsec) to the web server. The secure channel conveys the requested security label and protects the secrecy/integrity of the communication, with the web server. Communication is authenticated using certificates generated by the Application Authority.

4.2.3 Browser Transitions

Browser transitions may be triggered at the application, VM or VMM layers, causing a new tab to be opened, a new process to be started or a new VM to be loaded, respectively. Here, we describe a protocol for a VM transition.

1. When a BrowserLE receives back a URL label, e.g., `TopSecret`, that cannot be supported by the current VM, the browser sends a message to the VM Loader asking for a new VM.
2. The VM Loader service checks the VMM transition state and authorizes the request if the calling VM is allowed to transition to a new VM, labeled such that it can serve the application. If the request is authorized, the service looks for an appropriate VM image that can handle the URL's label (`TopSecret`).
3. The VM Loader dynamically creates a new VM with the appropriate label for handling the `TopSecret` URL. It then loads the application's policies and follows the initial configuration protocol as described in Section 4.2.1.

4.3 Enforcement

In this section, we illustrate how attacks are prevented at each layer, as labeled data is communicated across all layers.

VMM Layer The VMM's reference monitor controls whether a VM can be loaded at a particular lable range for a particular web application. In Figure 3, the protection state for the client VMMs' reference monitors had to be consulted before the client could be opened with label VM_2.

VM Layer The Browser VM's reference monitor controls access to all operating system resources, including web data stored in files, network sockets, and all process behaviors. The VM's reference monitor prevents any malware from modifying data, unauthorized users from corrupting browser code, malicious browser plugins from leaking data to unprotected files or from opening web sites that are outside the VM's range (too secret or too public). In short, the VM sandboxes applications and monitors system-level activity to prevent any process from violating its policy.

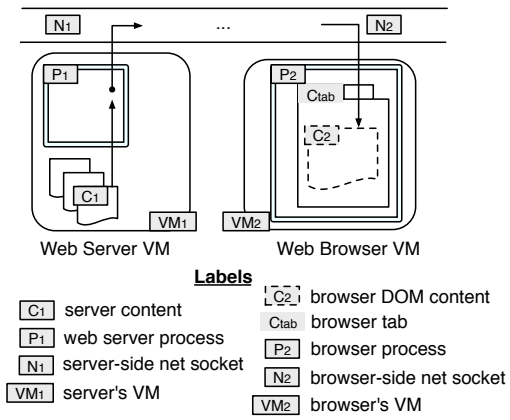

Figure 3: An example of the labeling state of our distributed web system when a browser makes an HTTP request to a web server. Label meanings are defined by the protection state in the security policies of the application, VM, VMM and network. Note: some labels may be the same, e.g. N_1 and N_2.

Network Layer The network layer is responsible for negotiating connections between VMs according to network policy. In Figure 3 this would require checking whether labels N_1 and N_2 were compatible.

Application Layer The application layer controls access to labeled objects in the browser application. From Figure 3 this would control whether data labeled N_2 can be read into a tab labeled C_{tab} or whether inputs entered into the tab (receiving the label C_2) could be written out to N_2. This can prevent a variety of common web attacks like script injection (cross-site scripting), cross-site request forgery (CSRF), drive-by downloads and can implement the same origin policy, the same origin mutual authentication (SOMA) policy or be flexible enough to implement other, relaxed versions of these policies. In short, the BrowserLE reference monitor ensures that secret inputs (username, password, or other data entered by the user in a secret tab) are not leaked to less secret sites through HTML posts or Javascript `XMLHttpRequest` operations. The application reference monitor also prevents integrity violations that might result in a cross-site request forgery (CSRF) attack.

The labeling state determines how each subject and object is labeled, but the protection state defines the meaning of the labels. Consider the example in Figure 3. In this figure, checks must be made for each object access. For example, when the web browser is started with label P_2, the VM policy must be checked to determine whether the VM labeled VM_1 may start up a program labeled P_2. Then the VM policy must be checked to see whether the process labeled P_2 may read and write to a socket object labeled N_2. Within the application, when a new tab is opened labeled C_{tab}, the VM policy is queried by the application to determine whether an application labeled P_2 may open a tab labeled C_{tab}. Furthermore, before content data labeled C_2 may be read into a tab labeled C_{tab}, the VM policy must be queried by the browser to determine if that is legal.

Figure 4: The Instrumented Bulletin Board Application enables users to assign security label to threads(s0,s1,s2,s3). The tab at the top shows the tab's current label. The tab security label must dominate the security label of all data displayed in that tab.

5. IMPLEMENTATION

We implemented a prototype based on requirements and design presented in Section 4, which simplifies some components as a proof of concept. We used some existing components and systems to achieve the end-to-end security goal. And we implemented the mechanisms to coordinate those components. For the browser, extensive instrumentation was required.

5.1 Components

For distribution and installation of policy on Client VMs, we implement the Application Authority and VM Loader components described in the architecture (Section 4.1). The IBB's Application Authority contains a complete policy (protection state, label state and transition state) for the VMM, VM, network and application layers, as shown in Figure 2. The initial configuration protocol, given in Section 4.2.1, directs the dissemination of these policies to the VM Loader and the subsequent installation of the policies in the appropriate layers of the Client VM.

To handle MAC at the VMM layer, we use the Xen [2] VMM to sandbox browser instances and web servers. To ensure MAC policy enforcement at the VM layer, we run SELinux [23]. At the network layer we enforce security requirements with labeled IPsec [16], which works with SELinux access control and applies SELinux labels to network sockets. At the application layer, we created FlowwolF, an instrument a Java-based web browser (Lobo [26]) to label input and output data and enforce MAC security requirements on them as they flow through the browser.

5.1.1 FlowwolF: Browser Implementation

A major goal of our system is to retrieve and render data with varying security requirements in a single tab without violating those security requirements. To achieve this, we implement a mandatory protection system *within* the browser. We begin with an existing browser and add labels to tabs and to network sockets (retrieved from the URL Label Mapper service). The labels are then propagated onto the Document Object Model (DOM) elements as they are created from data read in on the socket. The label on the tab can be set by the user (within the range granted to the browser by the VM policy) or a new tab can be created from a URL by inheriting the URL's label.

Labels are enforced within the browser through a reference monitor which is composed of mediation points. Mediation points ensure that secret data is not included in parameters to GETs or POSTs made to public sites. Also, network sockets to sites more secret than the current tab are prevented.

For our instrumentation, we use the Lobo browser because it is an all-Java browser. As future work, we plan to transform our implementation into Jif [21], a security-typed language based on Java, that would automatically ensure we are attaining complete mediation of all information flows. Jif uses static and dynamic labels on types. Through a label-checking algorithm, the compiler can determine whether an application enforces complete mediation on dynamic labels and enforces sound information flows on static labels. Then the compiler transforms the Jif code into Java code for which the main Jif residue consists of dynamic labels and their label checks (mediation points).

Though we have not annotated the code with static labels for a full Jif label analysis, we are using Jif's dynamic labels and dynamic label checks. We inherit labels from the system and enforce them according to system policy [11]. We also use dynamic labeling data structures [10] for all web requests to prepare for the eventual Jif conversion. These structures utilize special APIs for getting labels on URLs from the URL Label Mapper. They then check those labels to ensure they are legitimate for the desired GET or POST request for the given Tab and request parameters. Additionally, they use other custom APIs to set the SELinux label on the network socket object appropriately before attempting the connection to the URL's domain. Finally, they use APIs for requesting a new VM to the VM Loader when the URL Label Mapper returns a label for the network socket that is stopped by the local SELinux policy. Figure 4 shows FlowwolF displaying the our instrumented Bulletin Board application (IBB). IBB enables users to assign security labels (s0-s3) to threads. Each tab displays its current security label. For an object to be displayed within a tab, the security label of the tab must dominate the security label of the data.

5.2 Implementing an Application Policy

When an administrator installs a new application in the system, he must also define or add a link to the Application Authority for the new application. The Application Authority is expected to define a complete security policy (protection state, labeling state, and transition state) for the new application. Table 2 shows excerpts of the policy for the IBB application. The VMM and VM policies enable the application to access its own resources.

A dimension of the policy is missing in Table 2: the transition policy at the VMM layer. This policy determines how to label new Browser VMs when a transition is called for (see the transition protocol in Section 4.2.3). We expect administrators to define such policy as a predefined table in the VMM. This decision recognizes the VMM administrator's stake in controlling how new VMs may be opened and labeled. The VMM administrator may be in a better position than the application developer for making this decision. The following table illustrates a possible transition policy. In this table, ws3 serves bb objects with MLS ranges s0 and s1, ws1 and ws2 serve serve bb objects with MLS range s2 and s3.

```
table = ('bb', s0-s1,'1','ws3'),
```

VMM Policy:

```
# socket management
allow fwolf_t fwolf_t:tcp_socket {create .. read};
allow fwolf_t bb_t:tcp_socket {create .. read};
allow fwolf_t bb_t:association {recvfrom sendto};
allow fwolf_t ipsec_spd_t:association polmatch;
...
```

VM Policy:

```
# socket management (same as VMM policy)
allow fwolf_t bb_t:tcp_socket {create .. read};
...
# file management
allow fwolf_t bb_t:file {read .. setattr};
...
```

Network Policy: (IPsec labeled)

```
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;
```

Static URL Policy:

```
http://abc.mil/bb/index.php system_u:system_r:bb_t:s0
http://abc.mil/bb/js/main.js?ver=1400 ...
   ... system_u:system_r:bb_t:s0
...
```

Dynamic URL Policy:

```
http://abc.mil/bb/showthread.php?tid=2
http://abc.mil/bb/showthread.php?tid=3
```

Table 2: Excerpts of the mandatory policies for the IBB application. The VMM and VM policies enable FlowwolF (`fwolf_t`) to handle IBB objects (`bb_t`). The Network Policy is a template to be instantiated with the actual IP addresses of the involved nodes. The URL policy specifies the labels assigned to every URL object in the IBB application. In some cases, this label only can be generated on request (dynamic labeling policy). Also, notice that the labels assigned to elements that belong to an application can be different.

```
('bb', s2-s3,'2' ,'ws1', 'ws2'),
('DEFAULT', '0')
```

In our implementation we largely derived the application layer policy from the hosting VM's policy. Since labeled resources originate outside the browser, it is sensible for the browser to inherit the policy for those resources from the OS. *By default, FlowwolF allows MLS-style transitions of data (i.e. data may become more secret or lower integrity, but not vice versa).* It would be possible to define the application layer policy in the Application Authority and install it from the VM Loader. In any event, policy validation services [11] should be used to ensure that application policy is compliant with OS policy (this is important for ensuring the distributed reference monitor enforces complete mediation). Ensuring that OS policy is compliant with application policy is important for tamperproofing [**?**].

5.3 Improving Performance

We find that caching is possible in various parts of the system: for loading new VMs, for retrieving labels for URLs and for retrieving application policies.

Loading a new VM when the transition policy requires it, for instance, is a time-consuming operation. To reduce response time, we keep a pool of VMs pre-loaded for certain expected security ranges. In our example above, it would be sensible to keep a pre-loaded VM for reading Top Secret messages. To adjust pre-loaded VMs, to the actual requirements, we developed an *update daemon*. It loads, for instance, the VM and network policies for for a particular application. Having pre-loaded VMs makes a significant performance improvement as described in Section 6.2.

Locally caching the URL labels also has a significant impact on the usability of the FlowwolF browser, because every figure, page and other web objects must have a label. In some cases we use regular expressions (following the practice of SELinux file system labeling) to represent groups of labeled objects.

Lastly, caching application policies in the policy store (part of the VM Loader) accelerates the time for installing the policies on new VMs or pre-loaded VMs. These policies include label state for an application's URLs, maps from applications to their hosting application authorities and certificates for authenticating IPsec tunnels. On a local miss, the policy store queries the application's hosting Application Authority.

6. EVALUATION

To evaluate our approach, we instrumented a bulletin board messaging application [20] as described in Sections 2 and 5. IBB enables users to create message threads with security labels, and displays the content of such threads in a single browser tab only if the tab's label dominates the thread's label. We deployed the IBB application on top of our architecture and evaluated results. We presented parts of the IBB complete policy in Table 2.

6.1 Security Evaluation

Recall our claim of security from Section 3: a monolithic MAC policy is enforced by a multi-layer reference monitor if: (1) each component in the multi-layer reference monitor meets the guarantees for complete mediation and tamperproofing, accounting for trust in its environment; (2) each policy decision by a component in the multi-layer reference monitor is the same decision as that would have been made by a monolithic reference monitor using that policy. In addition to the mediation in SELinux [23] and Xen [2], we also added mediation to the FlowwolF browser (see Section 5.1.1) and defined labeled IPsec [16] policies for networking and inter-VM communication. For tamperproofing, we performed a tamperproof analysis [**?**] of the deployment of our browser package [3] on SELinux, and found that only trusted processes could modify the browser files. For policy decisions, our VMLoader protocol distributes MAC policy for protection state and transition states as is, so the same decisions are made as in the monolithic policy. A component cannot make an unauthorized decision as the VMLoader assigns each a security label. For labeling state, the monolithic MAC policy defines the labeling authorities (components) for different parts of the name space.

We run the following attacks. (1) We injected content into a web page to load a top secret message, regardless of the label of the browser tab. When the content is loaded, the browser requests the label of the message (top secret) and compares it against the label of the tab. If the tab's label is lower, for instance secret, the browser does not request the content. (2) We also injected content into a web page to load a URL for a malicious site. The URL points to an untrusted web site, i.e.

[3] A Linux package is a self-contained distribution of files necessary for an application, including its libraries, configuration files, and executable.

we did not configure an application authority for the site. As a consequence, the mapper will label the URL with the default label. The browser will detect this label and will not allow the browser to leak any data.

Our architecture can prevent the following attacks. (1) In a stored XSS attack, a public user of the IBB might try to poison a secret user by planting a URL to a malicious script that will be automatically loaded when the secret user views the message. Because the malicious script's URL will be labeled public by default, however, the script will have no access to secret data in the secret user's browser. (2) In an CSRF attack, a public user attempts to trick a secret user into posting a secret message using a command like `XMLHTTPRequest` that will not be noticed by the secret user. In our system, the data being posted by the `XMLHTTPRequest` would be labeled public by default, and so could not be posted in a secret message.

At the Network layer, attacks like eavesdropping and MITM are prevented by the authentication and encryption provided by Labeled IPsec. Finally, MAC at the VM layer contains malware and prevents it from modifying data stored in the system and owned by other applications.

6.2 Performance

In this section, we evaluate FlowwolF by measuring its overall runtime overhead and overhead for its major components. All of the experiments were run on two machines that represent the client and server sides of our applications. These machines run Linux 2.6.19 with Xen support. They both have Intel Pentium processors with 2.8 GHz, and 1 GB of RAM. We repeated all the experiments 10 times and present the average of each operation (Table 3).

Operation		Time(s)
New VM	Configure/Load Policy	4.90
	Load VM/Browser/Page	18.50
Pre-Loaded	Configure/Load Policy	2.52
	Load Browser/Page	3.44
Firefox	Cold-start Load Page	7.45
	Warm-start Load Page	1.50

Table 3: Browser configuration and (first) page loading times for a new VM, for a pre-loaded VM, and Firefox.

In Table 3, we compare three cases: (1) install of a new browser VM to load a web page into FlowwolF; (2) configuring a pre-loaded VM to load a web page into FlowwolF; and (3) a native browser. In the first case, we configure the policy files in the VM image, and then load the VM, FlowwolF browser, and finally the web page. The overall load time is over 20 seconds. Comparing the new VM and pre-loaded page loading times, we can see that the time to load the VM dominates the 18.50 seconds. The time to load VM services takes a major portion of this time, particularly the VNC server we use to load the browser automatically (nearly 8 seconds to load). The Tahoma prototype takes over 9 seconds to load a web page [6]. Removing most services would result in a similar performance. We also note that the Tahoma prototype was unoptimized, so further significant improvements are likely.

We also note that the pre-loaded VM takes half as long to configure as the new VM – 2.52 seconds for a pre-loaded VM versus 4.90 seconds for a new VM. For a pre-loaded VM, it is not necessary to write to the VM image, so several disk operations were removed. As a result, we recommend using an update daemon to load policy, even for new VM loading.

In comparison, the bottom line shows the page load times for a Firefox browser on the VMM (Xen domain 0). Loading a page in a pre-loaded VM with FlowwolF is comparable, due to FlowwolF's low cold-start time. In general, we would expect to add the `configure/load policy` time of 2.5 seconds to any security-aware browser on the first page load.

The IPsec overhead required to negotiate secure channels is not shown in Table 3. At present, we are using three IPsec channels to convey security labels: (1) from browser VM to client host VMM; (2) from client host VMM to server host VMM; and (3) from server host VMM to server VM. Each channel takes approximately 3 seconds to negotiate, but once it is established it can be reused. In general, we only need one channel to protect the communication between the server and the host, so we are looking into mechanisms with faster setup to convey labels locally, such as netlabel [22].

7. RELATED WORK

The primary focus of our work is in developing a multi-layer mandatory protection system. While others have developed MAC systems at various layers, including the components we use (SELinux, labeled IPsec, Xen, etc.), there are no other efforts seeking to harmonize these layers for implementing a single end-to-end policy. We apply our developments directly to web applications. There are some other web systems that use separation technologies for improving security.

Tahoma [6] provides support for running isolated web applications. Tahoma enables web servers to define policy for the clients, but we are also interested in enabling clients to express their own policy. *OP* [8] redesigns the browser's architecture. The new architecture has subsystems for separate plug-ins and makes explicit all communications between them. The browser kernel, the heart of the OP design, can mediate all communications between subsystems and enforce policies on those communications. While OP can enforce policies on the communications between subsystems, it is not designed to enforce policies on the communication channels between clients and servers, nor to dynamically obtain and configure client and server policies. Like *OP*, MashupOS [12] suggests the creation of abstractions to address security issues on browser implementations. In particular, abstractions to represent resources within the browser (i.e. disk, network) and access controls to rule access to those resources. The Same Origin Mutual Approval policy, *SOMA* [24], aims to block malicious scripts from downloading data from arbitrary web sites as part of pages served by known sites. To do so SOMA policy requires browsers to verify, with web servers, whether data inclusion is allowed or not.

None of the previous approaches addresses security requirements at layers other than the application layer. *NetTop* [19] provides support for running multiple single security level systems. These systems are virtually isolated and may communicate with others only if they belong to the same security level. NetTop provides support at additional layers (not only application layer) but it only supports a fix predefined MLS policy while we want to support a wider range of policies. DStar [30] aims to provide support for distributed applications. It propagates labels defined at the OS layer across systems, throughout label enforcement at the network layer. However, DStar does not integrate enforcement at the application layer, although that is mentioned as future work. OMOS [29] provides communication mechanism for allowing different domains to exchange data

among them authorized by a security policy. As DStar, it does not enforce any restrictions on data of different security levels within an application itself. Also it does not provide the system wide enforcement guarantees of our architecture.

Other approaches that aim to address browser flaws include instrumenting the Javascript interpreter [13], instrumenting and filtering code [28, 27], extending involved protocols [3], and limiting communication among frames; communication may be limited by classifying web servers into trusted and untrusted and blocking communications between frames that belong to different categories [15], and by implementing the same origin policy [14]. Our approach, on the other hand allows data mixing, while enforcing isolation based on the labels assigned to the data.

8. FUTURE WORK AND CONCLUSIONS

In the current implementation, Javascript objects are labeled, just like all DOM elements, but the labels are not yet pushed into the Javascript interpreter. Consequently, Javascript can be used to launder object labels. Developing a Javascript interpreter to enforce labels remains as future work.

We provide the first architecture for a multi-layer, mandatory access control system for enforcing system-wide security goals on web applications. We implement our architecture using commodity components as well as our own instrumented label-enforcing browser called FlowwolF, and some other custom-built components for supporting distributed label mapping and enforcement. We find that our implementation provides a flexible, efficient solution for enforcing end-to-end security requirements on web applications.

9. REFERENCES

[1] J. P. Anderson. Computer security technology planning study, volume II. Technical Report ESD-TR-73-51, Deputy for Command and Management Systems, HQ Electronics Systems Division (AFSC), October 1972.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03*, pages 164–177. ACM, 2003.

[3] A. Barth, C. Jackson, and J. Mitchell. Robust defenses for cross-site request forgery. In *CCS '08*. ACM, 228.

[4] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), March 1976.

[5] S. Chen, D. Ross, and Y.-M. Wang. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In *CCS '07*. ACM, 2007.

[6] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen. A safety-oriented platform for web applications. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 350–364. IEEE Computer Society, 2006.

[7] R. S. et al. Building a mac-based security architecture for the xen open-source hypervisor. In *ACSAC '05*, pages 276–285. IEEE Computer Society, 2005.

[8] C. Grier, S. Tang, and S. T. King. Secure Web Browsing with the OP Web Browser. In *IEEE Symposium on Security and Privacy*, pages 402–416, 2008.

[9] M. Harrison, W. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, Aug. 1976.

[10] B. Hicks, T. Misiak, and P. McDaniel. Channels: Runtime system infrastructure for security-typed languages. In *ACSAC*, December 2007.

[11] 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.

[12] J. Howell, C. Jackson, H. Wang, and X. Fan. Mashupos: Operating system abstractions for client mashups. In *HotOS.*, 2007.

[13] O. Ismail, M. Etoh, Y. Kadobayashi, and S. Yamaguchi. A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability. In *AINA '04*. IEEE.

[14] C. Jackson, A. Bortz, D. Boneh, and J. Mitchell. Protecting browser state from web privacy attacks. In *WWW '06*.

[15] C. Jackson and H. Wang. Subspace: Secure cross-domain communication for web mashups. In *WWW '07*.

[16] T. Jaeger, D. King, K. Butler, S. Hallyn, J. Latten, and X. Zhang. Leveraging IPsec for mandatory access control across systems. In *SecureComm*, 2006.

[17] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW*, New York, NY, USA. ACM.

[18] B. W. Lampson. Protection. In *5th Princeton Conference on Information Sciences and Systems*, 1971.

[19] R. Meushaw and D. Simard. NetTop: Commercial Technology in High Assurance Applications, 2000.

[20] MyBB Group. MyBB. `http://www.mybboard.net/`.

[21] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. `http://www.cs.cornell.edu/jif,July2001-2003`.

[22] Netlabel - explicit labeled networking for linux. `http://netlabel.sourceforge.net/`, 2007.

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

[24] T. Oda, G. Wurster, P. van Oorschot, and A. Somayaji. SOMA: Mutual Approval for Included Content in Web Pages. In *CCS '08*.

[25] OWASP Foundation. Open web application security project. `http://www.owasp.org/index.php/Top_10_2007`.

[26] T. L. Project. Lobo: Java Web Browser. `http://lobobrowser.org/`.

[27] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: vulnerability-driven filtering of dynamic html. In *OSDI*. USENIX Association.

[28] D. Yu, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser security. In *POPL '07*, pages 237–249, New York, NY, USA. ACM.

[29] S. Zarandioon, D. Yao, and V. Ganapathy. OMOS: A Framework for Secure Communication in Mashup Applications. In *ACSAC'08*. IEEE.

[30] N. Zeldovich, S. Boyd-Wickizer, and D. Mazieres. Securing distributed systems with information flow control. In *NSDI '08*.