5-2009

CLAMP: Practical Prevention of Large-Scale Data Leaks

Bryan Parno
Carnegie Mellon University

Jonathan M. McCune
Carnegie Mellon University

Dan Wendlandt
Carnegie Mellon University

David G. Andersen
Carnegie Mellon University

Adrian Perrig
Carnegie Mellon University

CLAMP: Practical Prevention of Large-Scale Data Leaks

Bryan Parno, Jonathan M. McCune, Dan Wendlandt, David G. Andersen, Adrian Perrig
CyLab, Carnegie Mellon University

Abstract

Providing online access to sensitive data makes web servers lucrative targets for attackers. A compromise of any of the web server's scripts, applications, or operating system can leak the sensitive data of millions of customers. Unfortunately, many systems for stopping data leaks require considerable effort from application developers, hindering their adoption.

In this work, we investigate how such leaks can be prevented with minimal developer effort. We propose CLAMP, an architecture for preventing data leaks even in the presence of web server compromises or SQL injection attacks. CLAMP protects sensitive data by enforcing strong access control on user data and by isolating code running on behalf of different users. By focusing on minimizing developer effort, we arrive at an architecture that allows developers to use familiar operating systems, servers, and scripting languages, while making relatively few changes to application code – less than 50 lines in our applications.

1. Introduction

To decrease costs and increase consumer convenience, businesses and governments make increasingly large amounts of sensitive information available online [7,16]. While convenient, online services are attractive targets for attackers, since a single flaw in a service's implementation can leak the sensitive data of millions of users [9, 22, 27]. Indeed, a recent study of over 500 data breaches found that 73% were the result of external attacks [28].

The growth of web services is aided by the availability of commodity web application stacks that simplify development and deployment. For example, the Linux, Apache, MySQL, Perl/PHP (LAMP) stack provides a turnkey system that allows even inexperienced programmers to quickly and easily deploy a full-blown web service. As a result, this model, ubiquitous in online services, tends to promote features and ease of use at the cost of security.

While LAMP-style stacks simplify development, they significantly increase the size of an application's Trusted Computing Base (TCB), the collection of code that must be correct to prevent a data leak. The TCB of a typical web application includes not only the large operating system and webserver codebases, but also a vast collection of scripts and third-party libraries. These scripts parse input, perform access control, and generate dynamic content, and yet this code is often written by inexperienced programmers and seldom subject to peer review. Unfortunately, the monolithic LAMP-style approach means that *a single vulnerability anywhere in a web application's software stack will often expose all user data.*

In this work, we describe CLAMP, an architecture that adds data Confidentiality to the LAMP model while retaining the ease of use that has made it so popular. CLAMP prevents web server compromises from leaking sensitive user data by (1) ensuring that a user's sensitive data can only be accessed by code running on behalf of that user, and (2) isolating code running on behalf of different users.

While previous work has explored techniques to prevent data leaks (Section 8), these approaches typically require significant programmer effort to port existing code to new APIs. In this work, we explore the extent to which we can *reduce a developer's burden* for securing current web applications. We find that, by focussing on the specific environment of web applications, we can greatly simplify the adoption of data protection. In particular, instead of dynamically tracking which pieces of code and data a user *has touched*, we bundle everything the user *might touch* into an isolated environment, making enforcement (and adoption) much simpler. While this approach may not work in all application domains, for web applications, CLAMP allows web developers to continue using the operating systems, applications, and coding platforms to which they are accustomed, with minimal changes to application code; we changed less than 50 lines of code in our applications.

We developed a prototype using platform virtualization (based on the Xen hypervisor [1]) to isolate system components on the web server. A trusted User Authentication module verifies user identities and instantiates a new virtual web server instance for each user. The database queries issued by a particular virtual web server are constrained by a trusted Query Restrictor to access only the data for the user assigned to that web server.

Our experience adapting three real-world LAMP applications to use CLAMP demonstrates the benefits of an architecture designed for compatibility with web application stacks. It took us less than two hours to adapt osCommerce, a popular open-source e-commerce LAMP application used by over 14,000 stores [19], to work on CLAMP. MyPhpMoney [3], a personal finance manager, required comparable effort. HotCRP [10], a web application used to manage the paper review process for academic conferences (including IEEE S&P 2009), offered us a

'worst-case' test of porting complexity, due to its highly configurable policies that determine what data (e.g., author names) should be considered private. Despite having no previous exposure to the software, we fully ported HotCRP to CLAMP; the port changed six lines of code, and developing the access control policy took less than two days.

Finally, our unoptimized prototype suggests that the user-perceived slowdown due to CLAMP's use of virtualization is not prohibitive (typical request latency for osCommerce is 5-10 ms slower than native). While platform virtualization increases hardware resource requirements, ongoing research has already demonstrated significant efficiency gains [5, 13, 15]. As Moore's law continues to reduce the price of CPU cores and memory, CLAMP will be increasingly attractive in comparison to the costs of rewriting applications or, worse yet, dealing with a large-scale data leak. Just as e-commerce websites currently accept the increased overhead of SSL as a good security trade-off, the $5.4 million price-tag for a median-sized data leak [21,28] may justify CLAMP's increased hardware requirements.

2. Problem Definition

2.1. Goals

Our primary goal is to prevent a web server compromise from leaking sensitive user data. We achieve this goal by ensuring that sensitive data in the database is only released to code running on behalf of a user who has authenticated successfully and has legitimate access to that data. We then protect this code from code operating on behalf of other users. We aim to enable these strong data protection guarantees for commodity web applications while minimizing the porting effort required.

2.2. Assumptions

Vulnerable Web Server. The adversary can exploit vulnerabilities in the web server such that she can run arbitrary code with root privileges.

Sensitive Data Definition. A developer of the web service can accurately identify the sensitive data contained within the database and accurately map the data to the user to whom it belongs. CLAMP will only protect data that is explicitly identified as sensitive. As we show in Sections 4 and 6, developers already implicity identify such information in their database schemas and application code. Thus, making the labels explicit for CLAMP is a relatively simple process compared with previous approaches.

User Authentication. An uncompromised web server can accurately identify the users of the web service. CLAMP's design is orthogonal to the problem of user authentication, and hence does not address other attacks (such as phishing or cross-site scripting) that compromise user authentication without compromising the webserver.

No Insider Attacks. While code on the webserver may contain vulnerabilities, we assume it does not already contain malicious code. CLAMP is not designed to prevent attacks by insiders who have legitimate access to change the web server's code.

For ease of exposition, we also assume that the site protected by CLAMP employs one or more webservers that connect to a database executing on a dedicated machine. However, CLAMP can also be applied to more complex, multi-tiered architectures. At each tier, a trusted component isolates code running on behalf of different users. Trusted components coordinate to ensure that only processes working for the same user can communicate with each other.

2.3. The Problems with LAMP

LAMP typically refers to the combination of the Linux operating system, an Apache web server, a MySQL database, and a collection of PHP or Perl scripts. In this paper, we use LAMP to refer more generally to the dynamic manipulation and delivery of content generated from data stored in a database. Our model encompasses web applications using other languages (e.g., Python and Ruby), databases (e.g., PostgreSQL), web servers (e.g., Microsoft's IIS), and OSes (e.g., Windows and BSD).

In a LAMP application, the scripts on the web server typically make all access control decisions. These scripts are configured to access the database using a privileged account. As a result, the database is powerless to defend itself from a compromised web server. The web server's large TCB and the tendency to spread access control logic throughout a web application's scripts makes it particularly difficult to verify whether the intended access restrictions are actually enforced, even when the webserver has not been compromised.

3. Architecture

In this work, we aim to protect the database component of a LAMP system from compromises of the web server or applications running on it. To overcome the weaknesses of LAMP described in Section 2.3, we identify three interdependent principles necessary to provide information-flow control in web applications (see Figure 1). First, we must accurately *identify* the code running on behalf of each authenticated user. Such identification can only be meaningful if we enforce *isolation* between code acting for different users. Finally, code acting on behalf of a user should only be *authorized* to view data appropriate for that user. Below, we provide an overview of how the CLAMP architecture achieves these principles, followed by a more detailed examination of CLAMP's components.

Principle	Description	Provided By
Code Identity	Binds executing server code to a user's identity	User Authenticator (UA)
Code Isolation	Isolates code running on behalf of different users	CLAMP Isolation Layer
Data Access Control	Prevents one user from retrieving another user's data from the database	Query Restrictor (QR)

Figure 1. *Principles for preventing data leaks. The rightmost column indicates which CLAMP component satisfies each principle.*

3.1. CLAMP Design Overview

The primary challenge in developing the CLAMP architecture is to provide the principles from Figure 1 while requiring a minimal amount of developer effort. We meet this challenge using two key insights.

First, existing web applications already contain the access-control logic necessary to authenticate users and authorize data access; however, a vulnerability anywhere in the web server's TCB can compromise this security-critical code. With CLAMP, we extract the existing user authentication logic and bundle it into an isolated User Authenticator (UA). We also extract the data access control logic and bundle it into an isolated Query Restrictor (QR) that mediates all web server access to the database. This approach allows us to recycle existing code and logic while simultaneously improving the security and auditability of these critical components.

Second, much of the complexity of applying general-purpose information-flow techniques (Section 8) to web applications arises from the fact that such systems track information flow at a fine granularity. However, for most web applications, a coarse-grained approach suffices. Indeed, rather than try to identify which particular pieces of memory, processes, or code segments a given user is using at any particular moment, CLAMP transparently assigns each user to an entire virtual web server that handles all of her interaction with the website. Thus, the virtual web server can be treated as a black box with a single label (the user's identity). All activity, including database requests, from that web server can be attributed to that user, and hence CLAMP ensures that it only operates on data belonging to that user. By isolating the virtual web servers from one another, CLAMP ensures that any damage a user does to her web server will only impact that one user; i.e., even if the user compromises her web server, she can retrieve only her own sensitive data from the database. Furthermore, a legitimate user cannot be affected by the actions of any other user, preventing a compromised virtual web server from extracting sensitive data from another user's virtual web server.

3.2. CLAMP Component Details

At a high level (Figure 2), clients contact the web application using SSL, and the Dispatcher assigns each client to a fresh virtual web server. When a user authenticates

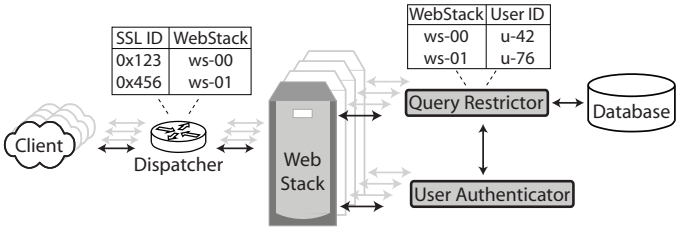

Figure 2. **CLAMP Architecture.** *Clients connect to the web application via the Dispatcher, which maps each SSL connection to a fresh virtual web server (a WebStack). The WebStack authenticates to the User Authenticator which updates the mapping of WebStack identity to user identity. The Query Restrictor limits a WebStack's view of the database to only include data belonging the WebStack's user.*

to her virtual web server, the web server presents the user's authentication credentials (e.g., the user name and password) to the User Authenticator (UA). The UA verifies the credentials and labels the web server with the user's identity, providing *code identity*. The Query Restrictor (QR) mediates the virtual web servers' access to the database based on the label assigned to each authenticated virtual web server, hence providing *data access control*. Finally, CLAMP provides strong *isolation* between the virtual web servers and strictly controls communication between CLAMP components. CLAMP allows each virtual web server to communicate with the UA and the QR but prevents the virtual web servers from communicating directly with the database or with each other. The database is completely isolated, except from the QR, and both the UA and QR are isolated from the Internet.

Below, we provide additional details on each component from Figure 2, starting with a discussion of how each component is isolated from the rest.

3.2.1. CLAMP Isolation Layer. The Isolation Layer isolates code acting on behalf of different users by instantiating a separate virtual web server (a WebStack) for each user and enforcing isolation between the WebStacks. It also restricts communication between the CLAMP components as shown in Figure 2 (and in more detail in Figure 4). Section 6.5 describes our implementation of these restrictions.

The code isolation required for CLAMP can be provided by mechanisms such as user-level processes, a `chroot` jail, or even threads within a Java virtual machine. For our prototype, we use platform virtualization (e.g., Xen [1]

or VMware ESX Server [29]); each WebStack operates inside its own virtual machine. Other isolation primitives would present different tradeoffs with respect to isolation, performance, and compatibility.

Compared with a higher-level isolation primitive, platform virtualization has several benefits. First, it makes our prototype instantly compatible with a wide range of existing LAMP applications. For example, the WebStack may run Apache on top of Linux, or it could equally run Microsoft's IIS on top of Windows. Developers can use existing tools, such as VMware's Converter[1] or Parallels' Transporter,[2] to create a WebStack directly from an existing physical web server. Second, leveraging the inherent isolation between code running in different VMs eliminates the need to rewrite server code not originally designed to strictly isolate user data. Finally, platform virtualization improves security, since an attacker seeking to break per-user isolation must not only compromise root-privileged code on the WebStack, but must also exploit the significantly more limited hypervisor interface to the virtualization layer. While using a VMM introduces additional memory and CPU load, our benchmarks (Section 7) indicate that an efficient hypervisor coupled with VM memory sharing can make these overheads tolerable.

3.2.2. Dispatcher. The Dispatcher allocates a fresh WebStack whenever a new client connects to the server and forwards all requests from that client to the same WebStack. For web applications that use HTTPS (our primary focus), the Dispatcher can use the SSL session ID to identify which TCP connections originate from the same client's browser. A non-HTTPS Dispatcher might instead rely on a cryptographic session cookie to achieve the same result. To reduce load, the Dispatcher forwards requests for the public portions of a site (e.g., non-HTTPS pages) to a single WebStack that handles such requests on behalf of all users.

3.2.3. Virtual Web Servers (WebStacks). Each WebStack is instantiated from a master image of the web-server software stack and is dedicated to a single client or user. If administrative changes are made to the web content or configuration, the master can be modified, and newly instantiated WebStacks will reflect these changes. Since data must not propagate from a user's WebStack back to the master image, work done on behalf of one user cannot be cached to improve another user's performance. However, caching can be done proactively by updating the master memory image to include, for example, the page displaying a new product promotion.

CLAMP maintains an identity for each WebStack, which must be a valid user's identity to enable access to sensitive information in the database. When first instantiated by the Dispatcher, WebStacks default to a restricted identity that can only access public information in the database. This allows a user to, for example, browse products prior to login. Section 4 describes these data access policies.

To acquire a user's identity, a WebStack must authenticate to the UA. This requires a small patch[3] to the web application's code to forward the user's authentication credentials (e.g., user name and password) to the UA. Once the UA verifies the credentials, the WebStack is labeled with the identity of the authenticated user.

When a WebStack determines that the user has ended her interaction with the web application (e.g., when the user clicks on a logout link or an inactivity timer expires), it sends a termination notice to the QR. The QR removes the mapping from WebStack to user identity, uses the Isolation Layer's management interface to destroy the WebStack, and forwards the termination notice to the Dispatcher, which closes its connection to the client and WebStack.

To preserve logging data, the web server in each WebStack is configured to write its logs to an append-only database. All major web servers provide this functionality.

3.2.4. User Authenticator (UA). Because the WebStacks are untrusted, CLAMP uses a special-purpose module, the UA, to authenticate users. When the WebStack provides the UA with the user's credentials, the UA performs the same verification check that the web server would normally do. For instance, the UA may check that the hash of the supplied password matches the user's entry in the database, which the UA accesses via the QR. Thus, creating the UA for a particular LAMP application is straightforward. We simply extract the existing user authentication logic from the application and replace it with calls to the UA.

3.2.5. Query Restrictor (QR) + Database. The QR (see Section 4 for details) ensures that a WebStack can only access database content for which its identity is authorized. Since only the UA can change a WebStack's identity, even a fully compromised WebStack (or a WebStack with a SQL-injection vulnerability) cannot access unauthorized data.

4. The Query Restrictor

The Query Restrictor (QR) is a trusted module that restricts virtual web servers' (WebStacks') access to sensitive database content. In our implementation, the QR is a specialized SQL proxy that interposes on all database traffic without requiring changes to the WebStacks. When a WebStack acting on behalf of a user attempts to connect to the database, the QR instead connects the client to a separate *restricted database* tailored specifically to that user. This restricted database has a schema identical to that of the full database, but contains no sensitive data except data available to that WebStack's user. The QR builds a restricted database by interpreting a web application's *data*

1. http://www.vmware.com/products/converter/
2. http://www.parallels.com/en/products/desktop/features/transporter/
3. For our applications (Section 6), the patch added 5-10 lines of code.

Table Name	Schema
Users	cust_id, email, pw_hash
Orders	cust_id, order_id, cc_num
Shipments	order_id, address

(a) **Example Database Tables**

Table	Access Predicate
Users	cust_id = UID
Orders	cust_id = UID
Shipments	Shipments.order_id = Orders.order_id and Orders.cust_id = UID

(b) **Policy for the schema in Figure 3(a).**

Figure 3. *Example database with its access policy.*

access policy, which enumerates sensitive tables in the database and indicates, for a given user, what rows the user can legitimately access. Using existing database functionality, CLAMP can provide these restricted databases efficiently, without the need to populate temporary databases or perform expensive copies.

We now describe how to associate database entries with specific users and how data access policies enable a generic QR to protect sensitive data.

4.1. Identifying a User's Sensitive Data

To construct a restricted database for a particular user, the QR must be able to identify the data that belongs to that user. Fortunately, any multi-user web application must already explicitly or implicitly tag each user's data so that it can be retrieved when the user logs into the website. For instance, database tables containing user information typically include a user identifier (UID) field that enables the application to differentiate one user's data from another's. For example, the simplified online shopping cart schema shown in Figure 3(a) uses the cust_id value as a UID. Even tables that do not explicitly contain a UID are implicitly linked back to a single user. For instance, each row in the 'Shipments' table links back to a single customer via order_id field shared between the 'Shipments' table and the 'Orders' table.

While this example is simple, in our sampling of real-world schemas, we found the task of attributing sensitive data to a system-wide UID value to be surprisingly straightforward, even in large systems. Developers often intentionally design database schemas to minimize the complexity of the code to retrieve user data.

CLAMP, using data access policies as described below, relies on this same UID value to identify what data a WebStack acting on behalf of a particular user is permitted to access. The QR learns of a new binding between a WebStack and a UID from the UA, which reports each successful authentication request made by a WebStack.

With this approach, CLAMP effectively implements access control policies identical to those intended by the application developer, but with two key improvements. First, CLAMP provides a straightforward way to express and audit access control policies using a single policy file instead of using checks scattered throughout the code. Second, CLAMP's QR enforces access control in a small isolated module that is robust to web-server compromises.

4.2. Data Access Policies

The QR uses a data access policy and the UID of a WebStack to construct the user's restricted database. The data access policy encapsulates application-specific knowledge of the database's schema, and hence allows the QR's implementation to be independent of the application's schema. The policy is parameterized based on the user's UID and describes which tables and potentially which rows within those tables the user can access. More specifically, the data access policy for a database consists of an *access predicate* P for each table in the database. In other words, if access predicate P_T is applied to table T in the full database, then the restricted database table T_R will contain only the rows that match the predicate.

Read Restrictions. Access to an entire table can be permitted or denied using the predicates 'True' or 'False'. The more interesting case is when a WebStack is permitted to access some, but not all, rows in a table. For example, applying the predicate 'cust_id = UID' to the 'Orders' table indicates that the restricted database for a WebStack associated with a particular UID contains only the rows of the table where the value of column 'cust_id' is equal to that UID. Such a policy would prevent a malicious WebStack from accessing sensitive credit card numbers ('cc_num' values) stored with orders for other users. Figure 3(b) shows a complete data access policy for the example schema.

In more complex schemas, a table may contain sensitive data without having a column specifying a UID. Data specific to a user is accessed with a SQL inner join.[4] A single "join" predicate could refer to many tables in the worst case. Fortunately, the joins required by a CLAMP access predicate directly map to joins the developer must already implement in order to enforce those access controls at the application layer. Since database schemas are often created in tandem with the software, the developer has a strong incentive to keep access control simple. The data access policies for osCommerce, MyPhpMoney, and HotCRP (see Section 6 and Appendices A and B) provide further evidence that crafting access policies is straightforward for anyone familiar with the database schema.

4. An SQL inner join combines two tables A and B into a new table that has a row for each pair of rows in the original tables (i.e., $A \times B$). Inner joins are often subject to an SQL "where" clause requiring that the value of a column X in A matches the value of a column Y in B.

Write Restrictions. CLAMP's access policies also control whether a user may modify (update, insert, or delete) rows in her restricted database, since modifying database contents can have important confidentiality implications. For example, a malicious WebStack that could change the passwords of other users could then authenticate as those users and access their data. Thus, for each restricted database, the CLAMP policy specifies whether the database is read-only (the default), fully modifiable, or modifiable only in conformance with the restricted database's access predicate. The last option ensures that any user modifications match the predicate that defines the restricted database. For example, if the access predicate restricts the database to only contain rows with the user's UID, then updates and inserts will only succeed if they include the appropriate UID.

One additional complication arises if the schema contains "link tables". Link tables allow the database to express many-to-many relationships. For example, a conference-management application may need to associate multiple author accounts with a single paper entry. To do so, a database designer might create a table (e.g., `Author2Paper`) with two columns, one for author ID and one for paper ID. Each author on a paper would have a single row in the table associating their user ID with the paper's ID. The access restrictions on the paper table will then depend on the presence of an appropriate "link" entry in the `Author2Paper` table associating the paper with its authors.

Unfortunately, if we apply the straightforward predicate-based approach to link tables, an adversary can gain access to other users' sensitive data. For example, the `Author2Paper` table would normally have an access predicate that forbids the insertion of rows unless those rows have the current user's UID. However, an adversary could exploit this by inserting a new row into the `Author2Paper` table with her own UID but with the ID of someone else's paper. The access predicate on the paper table, which relies on the `Author2Paper` table, would then allow the adversary to access that paper.

Fortunately, we can extend our basic predicates to protect link tables by applying two simple rules. New links (i.e., entries in the link table) may only be added if 1) the object of the link has no existing links (e.g., a new paper entry was just inserted), or 2) the current user already has an existing link to the object of the new link. The second condition allows a user to associate other users with an entry she created. For example, the author who first creates a paper entry may associate other users with that paper. We can automatically identify link tables that require these extended restrictions by searching for writable tables that appear in the WHERE clause of another table's read restriction policy.

In our experiments (Section 6), we found that these modification restrictions sufficed to protect user data.

4.3. Authentication Classes

The above description assumes that database requests are made on behalf of a user of the web application. This assumption, however, is sometimes violated, e.g., by new users registering for the first time or by privileged administrative consoles that offer broad access to sensitive data to special users identified as administrators.

CLAMP provides the flexibility to handle these scenarios by using *authentication classes*. WebStacks are tagged with an authentication class identifier (e.g., `new`, `user`, `admin`) as well as a UID, and the QR enforces a different data access policy depending on the class of the requesting WebStack. For example, at a university, students should only be permitted to access their own schedules, but teaching assistants may be allowed to access the list of all of the students in the particular class they are assisting with. Meanwhile, a professor should be able to view and update the grades for the students in her class, but not the grades for other classes or other students. Each of these roles can be encapsulated by a customized data access policy.

Authentication classes also permit WebStacks to access the database prior to authentication, for example, to retrieve generic data like product descriptions or promotions. For this reason, the QR also has a special `nobody` policy that denies access to all tables containing sensitive data. Users registering with a web application for the first time may also be members of the `nobody` class until they have registered as a user of the system, at which point their WebStack can be upgraded to a `user` class and labeled appropriately with the newly created UID.

4.4. Enforcing Data Access Control

WebStacks have neither network-level access to the database nor database login credentials, meaning that WebStacks must communicate with the database through the QR proxy or not at all.

For each valid authentication request seen by the UA, the UA passes a ($WebStack_{ID}$, UID, class) triple to the QR, which stores the mapping in its *WebStack-associations table* (see Figure 2). When the QR receives a database connection request from $WebStack_i$, the QR finds the corresponding $class_i$ and UID_i values in its associations table. The QR then uses the data access policy defined for $class_i$ to instantiate a temporary database containing only data accessible by user UID_i.

The QR leverages standard database features—database views and user permissions—to efficiently limit WebStack access to sensitive data based on data access policies. Relational databases already implement views using efficient mechanisms that automatically translate view queries into queries on the full database. Thus, no temporary databases or expensive copies are required for CLAMP to implement data access policies. We discuss additional details, including support for data modifications, in Section 6.2.

From \ To	Internet	Dispatcher	WebVMs	QR	UA	DB
Internet		✓				
Dispatcher	✓		✓			
WebStacks		✓		✓	✓	
QR		✓	✓			✓
UA				✓		
DB				✓		

Figure 4. **Permitted Communication.** *Each row indicates whether the component in the left column may communicate with the component in the top row. Note that communication is not necessarily symmetric.*

5. Security Analysis

CLAMP relies on trusted system components to enforce its security properties. These CLAMP components have three primary sources of attack robustness: a reduced trusted computing base (TCB), a minimized interface for each component of the TCB, and defense-in-depth. CLAMP reduces its TCB by selecting only the code/policy it must trust for data protection and extracting it from the web server stack into small modules with minimal interfaces. These smaller chunks are more amenable to static analysis or audit and are more easily (re)written using programming languages that facilitate secure coding. By reducing the interfaces to trusted components, CLAMP minimizes their attack surfaces and simplifies their implementation. Finally, CLAMP incorporates defense-in-depth using a tiered communication structure designed to enhance isolation (Figure 4). The result is an architecture that requires the compromise of at least two distinct component types (a WebStack and a more trusted component) to gain access to the database.

Below, we consider the security impact of an attacker compromising each component (Compromise Result) in CLAMP and each component's vulnerability to an attack (Attack Surface). We also consider additional attacks, such as Denial-of-Service and covert channels.

5.1. Dispatcher

Compromise Result: Because the Dispatcher holds the server's SSL private key and terminates all SSL connections, its compromise gives an attacker complete control over all active client sessions. An attacker could sniff sensitive data including login credentials, and extract any sensitive data the web application exposes during normal operation. However, users who do not connect while the Dispatcher is compromised are not at risk. Thus, the Dispatcher is in the TCB for active users. A Dispatcher compromise does not help the attacker launch a subsequent attack on the UA or the QR, since it is not directly connected to either component (Figures 2 and 4).

Attack Surface: The Dispatcher, which listens for incoming SSL connections on a TCP port, is the only CLAMP component that is directly accessible from the Internet. Its attack surface is limited to the VM OS's TCP stack and the SSL implementation running on top of it (in our case, OpenSSL) since no other ports are externally accessible. The Dispatcher application itself reads and writes socket data without inspecting the contents.

5.2. Web server Virtual Machine (WebStack)

Compromise Result: The compromise of a WebStack exposes any sensitive data that the WebStack has retrieved from the database on behalf of the user. However, since the WebStack is vulnerable to attacks only from the client for whom it is retrieving data, no invalid data access occurs. Thus, the WebStack is not in CLAMP's TCB. A compromise of a WebStack may be a stepping stone to attack the UA or QR.

Attack Surface: Vulnerabilities either in the HTTP server itself or in web application code can lead to the compromise of a WebStack. The size of its code base and the complexity of the interface it exposes make the web server a prime target for compromise. (This is the major class of attacks that CLAMP is designed to thwart.) Because each user receives a fresh WebStack image independent of other clients and because the Dispatcher sends traffic from each SSL connection to a unique WebStack that is isolated from all other WebStacks, no party except the user herself can even reach a particular WebStack. As a result, assuming that the pristine WebStack does not contain malware, legitimate users need not worry about other parties compromising their WebStack.

5.3. User Authenticator (UA)

Compromise Result: A malicious UA can assign arbitrary UID and class identifier credentials to any WebStack. As a result, an attacker that compromises both a WebStack and the UA could let the WebStack iteratively masquerade as each legitimate user, eventually extracting all sensitive user information from the database. Note that without compromising a WebStack, the UA cannot communicate with external systems. For web applications with an "admin" class, a compromised UA could also escalate the privilege of the malicious WebStack to increase information access. As a result, the UA is included in CLAMP's TCB.

Attack Surface: The UA is only reachable by an attacker that has successfully compromised a WebStack. The UA's interface is extremely narrow: it accepts UDP data from WebStacks consisting solely of user credentials. The UA then passes this data to the application-specific user au-

thentication code. Thus, the interface exposed to WebStacks includes the IP/UDP stack of the OS and an easily verifiable (fewer than 200 lines of code) authentication server.

5.4. Query Restrictor (QR)

Compromise Result: The QR has full access to the database, meaning that its compromise exposes all of a web application's sensitive data. The QR is part of CLAMP's TCB.
Attack Surface: Like the UA, network level access to the QR interface is only possible via a compromised WebStack. The QR receives simple network messages from the UA to add WebStack-associations and from WebStacks to remove associations. As with the UA, such simple server code permits manual verification. The QR interface, however, also includes the code to proxy WebStack database connections. Our analysis of the full-fledged open source MySQL proxy we use in our implementation suggests that a bare-bones proxy capable of acting as a QR would require under 5,000 lines of code. Manually auditing and/or rewriting the proxy in a robust programming language would be drastically easier than a similar procedure for a WebStack and its significant server software stack.

The QR uses database views to enforce database access policies. Unfortunately, specific database implementations may have vulnerabilities that allow an attacker to gain full access to a database from a view. While patches for the database software will presumably be issued eventually, it may actually be easier and faster to patch the QR until the database patch can be tested and deployed.

5.5. Database

Compromise Result: A database compromise yields all sensitive data to the attacker, so the database is also in CLAMP's TCB.
Attack Surface: The QR is the only module permitted to communicate with the database. This provides defense-in-depth, as a compromise of web server code neither exposes database credentials nor provides a channel for direct network communication with or attacks upon the database server. An attacker must compromise at least two components to gain direct access to the database.

5.6. Isolation Layer

Compromise Result: CLAMP must trust a security kernel, in our prototype a VMM, both to isolate WebStacks from each other and to protect trusted components like the UA and QR. A compromised Isolation Layer could take over these trusted components, exposing all sensitive data. The Isolation Layer is in CLAMP's TCB.
Attack Surface: When instantiated with a VMM, the Isolation Layer is difficult to compromise, since a malicious guest OS sees only a well-defined virtual hardware interface. The code size of popular VMMs such as Xen [1] is typically several orders of magnitude smaller than a commodity OS, or security architectures that apply to the OS (e.g., Flume [11] and SELinux [25,26]). Recent counts estimate Xen at 83K lines of code [14] versus nearly 5 million for the Linux kernel [32]. While these numbers can be reduced by trimming unnecessary functionality, the Linux kernel is likely to remain significantly larger than the Xen VMM. Using a specialized microkernel, such as L4 [6], would reduce the TCB by another order of magnitude.

Another potential route to Isolation Layer compromise is its management interface. Only the QR can access this interface (in order to recycle WebStacks that have finished a client session). Since a QR compromise already exposes all sensitive data, the internal management interface does not provide a useful attack vector.

5.7. Other Potential Attacks

Covert Channels and Side Channels. To prevent data leaks, we must consider CLAMP's vulnerability both to covert channels (surreptitious communication between two malicious entities) and to side channels (data leaked by an honest party). Since users have no interest in using a covert channel to leak their own data, and since a WebStack can only be compromised by the user on whose behalf it is acting, covert channels are not a primary concern.

However, we do not wish a compromised WebStack to be able to extract sensitive data about other users via side channels. To prevent such leaks, we rely on the Isolation Layer. In our prototype, a VMM strictly isolates the memory assigned to each WebStack and can enforce strong performance isolation between VMs [1]. By manipulating each VM's perception of time, the VMM can further limit the size of any side or covert channels that may be present [8]. Thus, side channels will provide an extremely limited amount of bandwidth.

Denial of Service (DoS) Attacks. Assigning each user to her own WebStack potentially leaves CLAMP vulnerable to resource exhaustion or DoS attacks. For example, an attacker can keep a WebStack in memory by sending periodic HTTP requests that simulate user activity. To consume resources, the attacker must keep WebStacks in memory. However, such attacks require the adversary to control a considerable number of user accounts, as a WebStack is only assigned after a user logs into the website. In security-sensitive applications, a user's account is often tied to a real-world object or identity. For example, many banks require a customer to be physically present to open an account. This binding between an account and a physical identity thwarts an attacker who tries to create hundreds of accounts in order to waste resources on the server. DoS attacks can also be mitigated by the Isolation Layer using constraints on resources assigned to any one WebStack.

Even with these techniques to limit resource consumption, CLAMP still requires more resources per user than an insecure site. Still, we envision CLAMP's use in scenarios where security is a priority, and the extra computing resources required to guarantee availability may be an acceptable cost. For example, this is true of SSL-enabled web servers today. Many web hosts have decided to devote the extra resources needed for SSL, even to the point of investing in special-purpose hardware (e.g., network cards that offload SSL computation). In the future, we plan to investigate additional techniques to further mitigate the effects of DoS attacks.

Security with Shared Database Content. Some content in web-application databases may be legitimately writable by any client, and may later be returned to other clients as HTML (e.g., user reviews). An attacker who injects malicious HTML or javascript might be able to modify the behavior of a web page so that it submits sensitive data to an untrusted server. While "scrubbing" user-submitted content could detect many such attacks, a more reliable defense is to isolate user-generated content from potentially sensitive content using iframe HTML elements – an approach that is widely used in so-called "mash-up" sites today [31]. Many sites with highly sensitive user data already avoid shared user content, obviating this concern.

6. Implementation

We have developed a proof-of-concept implementation of the CLAMP architecture and applied it to osCommerce [19], MyPhpMoney [3], and HotCRP [10]. osCommerce is an e-commerce web application currently in use by over 14,000 online merchants [19], making it (to the best of our knowledge) the most widely used open-source e-commerce web application. MyPhpMoney is a personal finance management application. HotCRP is conference management software that has been employed for numerous conferences (including this one) and allows conference organizers to configure a wide range of options to control information access. Below (and in our performance evaluation in Section 7), we focus on the more widely used osCommerce. We defer detailed discussion of MyPhpMoney to Appendix A and HotCRP to Appendix B.

Below, we describe the implementation of each component. We evaluate CLAMP's performance in Section 7.

6.1. User Authenticator (UA)

Our experience creating User Authenticator modules for osCommerce, MyPhpMoney, and HotCRP demonstrates that "porting" an existing LAMP application to CLAMP is simple. We implemented the UA in its own VM based on a minimal Debian Linux installation. The UA is divided into two components: (1) a generic UDP server that accepts requests and communicates with the QR; and (2) application-specific code that implements user authentication. The generic server is under 50 lines of PHP code and is used for all web applications.

To implement authentication specific to osCommerce, we readily identified three PHP files with login functionality (`login.php`, `create_account.php`, and `admin/login.php`) and moved the password checking code to the UA. This involved less than 150 lines of code, all of which were taken directly from osCommerce. In each of the three PHP files, we replaced the login code with three lines of PHP to make simple UDP calls to the UA with authentication credentials.

MyPhpMoney and HotCRP proved similarly straightforward: the user authentication code was readily identified, and each application required changes to fewer than 10 lines of code (see Appendices A and B).

6.2. Query Restrictor (QR)

Our QR is implemented as an extension to mysql-proxy,[5] one of MySQL's Enterprise Tools. The mysql-proxy is designed to monitor and optionally transform the database connections it proxies. Each event of significance (e.g., initial handshake, authorization exchange, or SQL query) prompts a call to a user-supplied Lua script. We modified the proxy to improve its scalability. We also added functionality to accept calls from the UA to add and remove mappings between WebStack identities and user identities. After trimming unused functionality, the proxy consists of less than 5,000 lines.

We implemented the QR functionality as a Lua script consisting of 294 lines of code. The Lua script tracks the WebStack identities. When a WebStack first connects, the QR assigns it to a temporary database (preallocated before the QR starts). The temporary database is populated with *views* of the main database's tables. These views are represented internally by MySQL as select statements; thus, they require little state, and they do not duplicate data.

Once a user logs in, the QR customizes the temporary database's views based on the WebStack's identity and the data access policy supplied by an administrator. For example, if the user's UID is 124, then the view of the Orders table would be customized (using the SQL command `ALTER VIEW`) such that it contains only rows where `user_id` = 124. Subsequent connections from the same WebStack are routed to the same temporary database (avoiding customization) until the WebStack is recycled and given a new identity.

Whenever the WebStack attempts to connect to the database machine, the QR rewrites the WebStack's authorization SQL commands to use a temporary MySQL user who only has access to the temporary database assigned to the WebStack. Thus, each WebStack sees only its own

5. http://forge.mysql.com/wiki/MySQL_Proxy

temporary database, and cannot access any of the other temporary databases or the main database itself. Hence, the QR does not need to modify the WebStack's database queries, since they will be processed against the temporary database that only contains sensitive information for the WebStack's user. In addition, because the views in the temporary database have the same names as the tables in the original database, all of the queries generated by the web application work as they did before. Thus, the QR's functionality is transparent to the code in the WebStacks. The QR's transparency enables QR reuse across web applications. We use the same QR for osCommerce, MyPhpMoney, and HotCRP by simply loading the appropriate policy files.

We also use existing database functionality to control data modifications (update, insert, delete). By default, we use existing database-level access restrictions to limit the temporary MySQL user to `SELECT` statements, effectively making the temporary database read-only. To allow modifications, we can grant the MySQL user insert, update, and/or delete rights on a per-table basis, a feature supported by all major databases.

However, databases differ in how they handle writable views. MySQL, Microsoft SQL Server, and Oracle Database all permit writable views and support the standard SQL clause (`WITH CHECK OPTION`) during view creation. This clause causes the database to automatically check that all inserts and updates conform with the predicate used to define the view. These databases do prohibit modifications to views that use functions (e.g., `SUM`) that destroy the one-to-one mapping between rows in the view and rows in the underlying database. Fortunately, since CLAMP uses views to present a subset of the existing rows in the database, its view definitions do allow modification. In the case of link tables (discussed in Section 4.2), MySQL's view implementation will not allow the types of restrictions we require. To work around these restrictions, we currently duplicate the data in the link tables, using triggers to maintain data consistency. Fortunately, in the applications we examined, link tables are rare and contain relatively little data. PostgreSQL is one major database that does not directly support updateable views. However, it allows the creation of rules that rewrite modifications of a view's content into appropriate actions on other tables, and hence could be made to support CLAMP's access policies.

6.3. Data Access Policies

Of the 47 tables in the osCommerce database, we identified 7 that contain sensitive data (either related to customers or their orders). Thus, each policy file contains 7 lines, one for each table. We crafted policies for three access classes: `user`, `admin`, and `nobody`. The `admin` class (used by the store's owner) was given full access to the tables with sensitive data, while the `nobody` class was given no access. The `user` policy (Figure 5) restricts the data in each sensitive table based on a customer_id value used as an index in all 7 tables. Even as newcomers to osCommerce, we found it straightforward to identify the tables with sensitive information and to craft the policy files. Altogether, this effort required less than an hour.

Table	Restriction
address_book	customers_id = UID
customers	customer_id = UID
customers_info	customer_info_id = UID
customers_basket	customer_id = UID
customers_basket_attributes	customer_id = UID
products_notifications	customer_id = UID
orders	customer_id = UID

Figure 5. **osCommerce User Data Access Policy**

The data access policy for MyPhpMoney proved similarly effortless (Appendix A). HotCRP's extremely flexible and configurable access model makes it a worst case for data policy development, and indeed, it took considerably more work. Nonetheless, a few days of effort proved sufficient (Appendix B).

6.4. Dispatcher

The Dispatcher VM has two virtual network interfaces: one connecting to the Internet and another connecting to the virtual LAN segment containing the WebStacks. The Dispatcher is approximately 750 lines of C++ code built on top of the OpenSSL library. To simplify our prototype implementation, the Dispatcher is co-located with a *VM pool manager*, which notifies the Dispatcher when a WebStack is finished and when a clean replacement is ready (our full design places this functionality within the QR to provide defense-in-depth). Additionally, the Dispatcher forwards non-SSL (port 80) traffic to a special, unprivileged WebStack that serves public, non-sensitive data.

6.5. Isolation Layer

Our prototype implementation of the CLAMP architecture uses the Xen 3.1.0 VMM [1] to isolate server components, though as we note in Section 3.2.1, other isolation techniques offer viable alternatives. The prototype uses a master WebStack to create a read-only file system from which each ramdisk-based WebStack is instantiated. This maximizes performance by removing the hard disk from the performance-critical path, and minimizes the time required to refresh a WebStack between clients.

Ideally, refreshing a WebStack (after a client's session has terminated) should be implemented using the *delta virtualization* technique developed by Vrable et al. to enable a single machine to serve as a honeypot for thousands of IP addresses [30]. Delta virtualization refers to the ability to *fork* (similar to a process-level fork) a running reference VM many times, using copy-on-write memory sharing to

minimize the memory footprint of additional VMs. For systems such as honeypots and our WebStacks, the memory savings can be substantial, since all WebStacks are identical until client activity influences their execution. In addition, WebStacks can be instantiated so rapidly that we can fork additional WebStacks on demand, i.e., in response to incoming client connections.

Unfortunately, due to bugs and instabilities in the delta-virtualization version of Xen, we were unable to test the throughput of our implementation using delta virtualization. Thus, to simulate CLAMP's throughput with a stable version of delta virtualization, we create a pool of 50 static WebStacks and assign each one 64 MB of RAM. When a client terminates a connection to a WebStack, the Dispatcher waits an amount of time equal to the time needed to destroy and then fork a new VM using delta-virtualization, and then reuses the existing WebStack. Without delta virtualization, CLAMP would have to start new WebStacks from scratch.

The VMM also enforces the communication restrictions shown in Figure 4. With Xen, the Domain 0 VM provides the backend driver for the network cards in the guest VMs. Hence, all communication between CLAMP components travels through Domain 0, and Domain 0 can always authoritatively identify a packet's source. Thus, we assign each VM a unique IP address and then use `iptables` in Domain 0 to prevent VMs from spoofing their IP addresses and to control which VMs can communicate.

7. Evaluation

While our CLAMP prototype provides strong security benefits via VMM isolation and QR database access control, it comes at the expense of additional processing overhead. As x86 virtualization becomes increasingly vital to IT infrastructures, we expect this overhead to diminish. Experience also suggests that companies are willing to invest additional hardware resources in exchange for tangible security benefits (e.g., some e-commerce sites use dedicated hardware to offload SSL processing). Alternatively, CLAMP can utilize other isolation techniques with different performance-security tradeoffs (Section 3.2.1).

We use our proof-of-concept prototype to estimate the impact CLAMP may have on web server performance, both in terms of web request latency (Section 7.1) and the overall throughput of the system (Section 7.2). As explained in Section 6.5, the current version of delta virtualization is unreliable [30], and hence the throughput experiments use static VMs to simulate the effects of delta virtualization.

A practical deployment of CLAMP would obviously require improving the efficiency and robustness of delta virtualization, developing better documentation, constructing an installer, and creating a better management interface. We believe these are all tractable tasks.

Experimental Setup. We run all of our experiments against the same database installed on a dedicated machine. We use MySQL 5.1.31 running on Debian Linux on a 2.00 GHz Pentium IV with 512 MB RAM. We run Xen 3.3 with a para-virtualized Linux kernel on a four-core 1.80 GHz AMD Opteron with 6 GB RAM. Our "native" web server used as a baseline runs on the same AMD machine, but with the Linux kernel running directly on the hardware. Both the Xen VMs and the baseline installation use version 2.6.18 of the Linux kernel. Our test client is equipped with a 3.00-GHz Core 2 Duo and 2 GB of RAM.

Results Overview. Our results indicate that while our current prototype imposes substantial request processing overhead, the overall performance of the system remains reasonable. The most significant overhead that our prototype faces comes from spawning new virtual machines. Thus, efficient implementation of CLAMP using virtualization will benefit from additional improvements in rapid VM spawning, an area of active research [5, 13, 30]. We focus on the results for osCommerce.

7.1. Latency

We use a series of macrobenchmarks to measure our prototype's impact on the latency of several classes of web requests.

7.1.1. Macrobenchmarks. For these benchmarks, clients retrieve osCommerce pages from either a "native" server running directly on hardware or a CLAMP server as described in Section 6. Both servers run on the same hardware, use the same version of osCommerce, and access the same database server. The servers' caches are warmed prior to measurements, and we report the average and standard deviation of 50 trials for each request type.

These experiments measure the time it takes to complete a single client's first request to an unloaded server. The time includes SSL establishment time, and, with CLAMP, the time required for the Dispatcher to select and connect to a WebStack. Since we assume the server is lightly loaded, this WebStack can be pre-forked, and hence we do not include the time needed to fork a WebStack. We discuss forking overhead below in Section 7.2.2.

Figure 6 compares request latency with and without CLAMP. The first two requests show the time to fetch (with SSL, since we expect CLAMP applications to use SSL) a small (8 KB) or a large (3 MB) static file that does not require database access. The static file retrieval with SSL reveals that the cost of SSL session establishment—a cost companies accept today—dominates, and the CLAMP prototype adds less than 2% overhead for small files. With large files, Xen's virtualized networking overhead reduces performance, but overhead remains under 14% for 3MB files. If an SSL connection has already been established, then our CLAMP prototype adds 0.12 ms (16%) to small

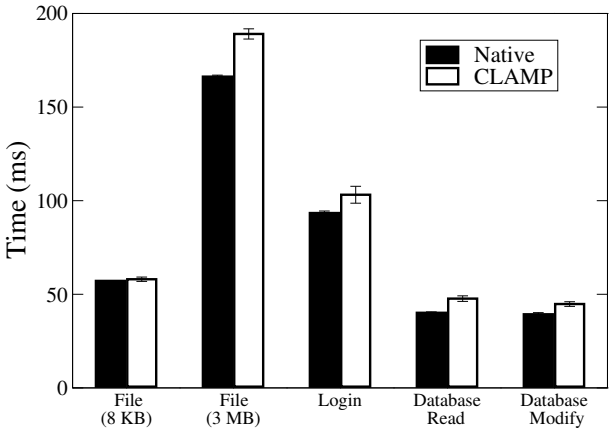

Figure 6. **Macrobenchmark Latency.** *Comparison of the average time to complete different web requests within osCommerce on native hardware versus our CLAMP prototype. Smaller is better.*

files and 22 ms (20%) to large files. Improvements to virtualized networking performance is an active area of research [12, 15, 24].

The "Login" measurement quantifies the overhead from the additional work that CLAMP performs when a user logs in. The login page is SSL protected, makes several database queries, and requires inter-VM communication between WebStacks, the UA, and the QR. Importantly, login times are only slightly longer (10 ms, or 10% longer) using our CLAMP prototype. These results indicate that the QR's step of creating a restricted database for an individual user does not increase login completion time excessively.

Finally, the "Database Read" test measures the time required to load an SSL-protected PHP page that makes 20 unique database `SELECT` queries after the user has logged in (and hence established an SSL connection), while the "Database Modify" test measures the time required to load an SSL-protected PHP page that makes 10 unique database `INSERT` queries and 10 unique `UPDATE` queries after the user has logged in. These tests represent the most common use scenarios for a CLAMP application; the CLAMP prototype adds only 7 ms (19% overhead) to pages based on database reads and 5 ms (14% overhead) to pages that make database modifications, amounts well below the threshold at which users will notice a delay.

In microbenchmarks, we found that the use of MySQL's views added 50% overhead to read requests. Other databases (e.g., PostgreSQL) offer better view performance, with overheads of less than 7% on the same workload.

7.2. Throughput

As shown above, the CLAMP prototype only slightly increases the latency of individual requests. The other important metric is our prototype's effect on the server's throughput (i.e., the number of users that can be handled simultaneously), which is affected by both memory and CPU resources.

7.2.1. WebStack Memory Usage. Unlike the native web server, the number of simultaneous users that CLAMP can support is limited by the number of WebStacks that fit in memory. As discussed in Section 6.5, delta virtualization creates a copy of a master WebStack using copy-on-write memory sharing.[6] Thus, the memory consumed by a WebStack is limited to the number of unique memory pages to which it writes.

To evaluate the effectiveness of this memory sharing, we measure the amount of private memory (i.e., memory that must be allocated to a WebStack after it writes to a memory page) used by a WebStack. We warm the master WebStack prior to forking the WebStacks that handle benchmark requests. We perform experiments to benchmark WebStack memory overhead involving small file requests (between 1 and 4 KB) and full osCommerce PHP page requests. The osCommerce page requests involve retrieving embedded images and issuing multiple database queries to generate the resulting web page.

Figure 7 summarizes our results. The first data point (Unique URL "0") shows the memory usage of the forked WebStack before it has served any requests. The subsequent points show the memory usage of the forked WebStack 10 seconds after an additional unique URL is retrieved. The line labeled *Single Object* shows that requesting individual files increases only slightly (less than 1 MB) the amount of memory consumed by a WebStack. The line labeled *Complete Page* indicates that retrieval of complete osCommerce pages increases the memory consumed by a WebStack by approximately 1 MB.

These results indicate that even a client who browses many image-rich and database intensive pages will only incur a virtualization memory overhead of a few tens of megabytes. Thus, if memory were the only bottleneck, our server with 6 GB of RAM could support at most 500 simultaneous WebStacks (and hence authenticated users), though each WebStack can handle multiple requests from its user. However, in practice, we find that CLAMP hits CPU resource limits before it reaches memory limits.

7.2.2. CPU Usage. CPU resources limit the rate at which CLAMP can process client logins (due to the need to fork new WebStacks), and the rate at which it can handle connections from established users (due to context switching between WebStacks).

Logins. When a new user logs in, CLAMP must allocate a new WebStack. We implement this by forking the master WebStack image using the version of Xen developed

6. This does not create a security risk, since the master WebStack's memory image does not contain any sensitive data. Any modifications made by a WebStack will be seen only by that particular WebStack.

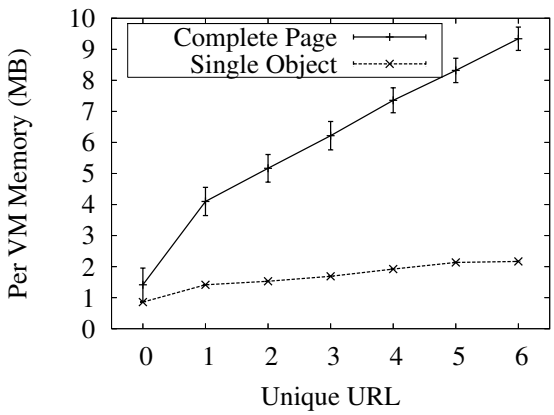

Figure 7. **WebStack Memory Usage.** *With delta virtualization, a WebStack's memory usage grows as it handles additional requests. Here, we measure that growth by fetching individual images (the "Single Object" line) and complete osCommerce pages.*

for the Potemkin Honeyfarm [30]. The Potemkin authors report that VM forking requires approximately 500 ms, and in our tests, we found that one CPU could fork two WebStacks/second, while the native server can handle up to 85 logins/second. While low, this approach is still faster than launching a new WebStack, which takes an average of 36 seconds. We also believe CLAMP's results can be improved using multi-core platforms and optimized forking techniques [5, 30]. In addition, the Dispatcher can buffer incoming client logins, allowing CLAMP to tolerate larger bursts of users at the cost increasing the latency of the HTTP response.

Connections. To measure the number of simultaneous connections our prototype can support, we spawn 500 clients at a fixed rate, i.e., X clients per second. Each client requests an SSL-protected PHP page that makes 10 unique database queries. We measure the amount of time taken for each client's request, and we judge a request successful if it completes in under two seconds. We define the system's overall throughput as the highest value of X for which all 500 clients' requests are successfully serviced. This approach represents a worst-case scenario for CLAMP, since each request must be directed to a different WebStack. As explained in Section 6.5, we use 50 static WebStacks to simulate the effects of delta virtualization.

On the native server (running directly on the hardware), we measured a throughput of 83 connections/second, while with our prototype, we measured a throughput of 35 connections/second (i.e., 42% of native). The main sources of overhead are the virtualized networking, and the expense of context switching between so many WebStacks. Nonetheless, given the unoptimized state of our prototype, and the security benefits CLAMP provides, we feel that this performance is reasonable, given that it would allow the server to process over three million requests per day.

8. Related Work

CLAMP focuses on *tolerating* the compromise of a web server. It operates by restricting the flow of sensitive data among code modules using virtualization. We omit an extensive discussion of the significant prior work that focuses on detecting and preventing exploits in network servers; instead we focus on more closely related work in information flow control.

Mandatory Access Control (MAC) partitions software systems to protect data confidentiality and integrity by limiting how components access one another (e.g., MULTICS [23], SELinux [25, 26] and AppArmor [20]). Recent research has yielded more flexible MAC called distributed information flow control (DIFC) [17]. DIFC is data-centric, enabling security policy enforcement even if systems do not provide strong protection via isolation. While DIFC systems provide considerable expressive power, creating applications within this model (or retrofitting legacy code to use it) requires specialized knowledge of DIFC-specific application platforms. For example, Asbestos [4] and HiStar [33] propose a new OS and require applications to be ported to use new DIFC-specific abstractions. Other systems (e.g., Flume [11]) implement DIFC using system call inter-positioning specific to a particular OS. Similarly, specialized programming language constructs (e.g., JIF [17] and SIF [2]) provide fine-grained DIFC, but only for applications tailored to those constructs.

We observe that the high cost of adoption has hampered the deployment of DIFC techniques in production systems, and design CLAMP to be readily applicable to real-world web applications. Although CLAMP does not provide all of the properties of a full DIFC system (e.g., it does not explicitly label data), our focus on the specific domain of web applications allows CLAMP to protect user data, while providing developers with the flexibility to select the web application components (OS, web server, programming language) that best fit their needs.

Commercial products are available that provide row-level database access control [18], similar in spirit to our Query Restrictor. Since these solutions may allow a compromised web server to access sensitive data for all active users, a CLAMP-like approach of extracting user authentication into an isolated module and isolating code running on behalf of different users is still desirable.

9. Conclusion

In this work, we have investigated techniques to secure LAMP applications against a large number of threats while requiring minimal changes to existing applications. We developed the CLAMP architecture to isolate the large and complex web server and scripting environments from a small trusted computing base that provides user authentication and data access control. CLAMP occupies a point

in the web security design space notable for its simplicity for the web developer. Our proof-of-concept implementation indicates that porting existing LAMP applications to CLAMP requires a minimal number of changes, and the prototype can handle millions of SSL sessions per day.

Acknowledgments

The authors would like to thank Diana Parno, Amar Phanishayee, Arvind Seshadri, and Matthew Wachs for their insightful comments and suggestions. The anonymous reviewers also provided valuable feedback.

This research was supported in part by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office, grants CNS-0509004, CNS-0716287, CCF-0424422, and CNS-0831440 from the National Science Foundation, and support from the iCAST project, National Science Council, Taiwan under the Grant NSC96-3114-P-001-002-Y. Bryan Parno is supported in part by an NSF Fellowship. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of ARO, CMU, iCast, NSF, or the U.S. Government or any of its agencies.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebar, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, Oct. 2003.

[2] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *Proceedings of the USENIX Security Symposium*, Aug. 2007.

[3] Courou. MyPhpMoney v2.0. http://sourceforge.net/projects/myphpmoney/, Apr. 2008.

[4] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *SOSP*, Oct. 2005.

[5] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, A. Vahdat, G. Varghese, and G. M. Voelker. Difference engine: Harnessing memory redundancy in virtual machines. In *OSDI*, Dec. 2008.

[6] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg. The performance of μ-kernel-based systems. In *SOSP*, 1997.

[7] H. Havenstein. Google unveils plans for online personal health records. *Computerworld*, Oct. 2007.

[8] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering*, 17(11):1147–1165, Nov. 1991.

[9] R. Kerber. Court filing in TJX breach doubles toll. *The Boston Globe*, Oct. 2007.

[10] E. Kohler. Hot crap! In *Proceedings of WOWCS*, Apr. 2008.

[11] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *SOSP*, Oct. 2007.

[12] S. Kumar and K. Schwan. Netchannel: a VMM-level mechanism for continuous, transparent device access during VM migration. In *Proceedings of VEE*, Mar. 2008.

[13] H. A. Lagar-Cavilla, J. A. Whitney, A. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *Proceedings of Eurosys*, Apr. 2009.

[14] D. Magenheimer. Xen/IA64 code size stats. Xen developer's mailing list: http://lists.xensource.com/, Sept. 2005.

[15] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing network virtualization in Xen. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, June 2006.

[16] Microsoft Corp. Microsoft announces HealthVault. http://www.microsoft.com/industry/government/solutions/healthvault.mspx%, Oct. 2007.

[17] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, Oct. 2000.

[18] A. Nanda. Keeping information private with VPD. In *Oracle Magazine*, Mar. 2004.

[19] Open Source E-Commerce Solutions. osCommerce. http://www.oscommerce.com/, Apr. 2008.

[20] openSUSE. AppArmor. http://en.opensuse.org/AppArmor.

[21] PGP Corporation. 2006 annual study: Cost of a data breach. http://www.computerworld.com/pdfs/PGP_Annual_Study_PDF.pdf.

[22] M. Rhor. Alum charged with hacking into Texas A&M. http://www.guardian.co.uk/worldlatest/story/0,,-6902530,00.html, Sept. 2007.

[23] J. H. Saltzer and M. D. Schroeder. Protection of information in computer systems. *Proceedings of IEEE*, 63(9), 1975.

[24] J. R. Santos, G. J. Janakiraman, Y. Turner, and I. Pratt. Netchannel 2: Optimizing network performance. In *Proceedings of the XenSource/Citrix Xen Summit*, Nov. 2007.

[25] S. Smalley and P. Loscocco. Integrating flexible support for security policies into the Linux operating system. In *Proc. of the USENIX Annual Technical Conference*, 2001.

[26] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *Proceedings of the USENIX Security Symposium*, 1999.

[27] TD AMERITRADE releases results of client SPAM investigation. http://www.amtd.com/newsroom/releasedetail.cfm?ReleaseID=264044, Sept. 2007.

[28] Verizon Business. 2008 data breach investigations report. http://www.verizonbusiness.com/resources/security/databreachreport.pdf.

[29] VMware Corporation. VMware ESX, bare-metal hypervisor for virtual machines. http://www.vmware.com/products/vi/esx/, Nov. 2008.

[30] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity and containment in the Potemkin virtual honeyfarm. In *SOSP*, 2005.

[31] H. Wang, X. Fan, and J. H. C. Jackson. Protection and communication abstractions for web browsers in MashupOS. In *SOSP*, Oct. 2007.

[32] D. A. Wheeler. Linux kernel 2.6: It's worth more! Available at: http://www.dwheeler.com/essays/linux-kernel-cost.html, Oct. 2004.

[33] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.

Appendix A.
Applying CLAMP to MyPhpMoney

This appendix describes the process by which we ported MyPhpMoney [3], a personal finance manager, to CLAMP.

User Authenticator (UA). Porting MyPhpMoney was straightforward. We identified password checking code (less than 150 lines of code) in the original source and copied it to the UA. We added calls to the UA to one file (`login.php`) which handles user creation, login, and logoff, adding less than 10 lines altogether. Finally, we replaced two deprecated PHP database-access functions with their modern equivalents. In total, identifying the relevant code and making the necessary modifications required about two hours.

Query Restrictor (QR). We use the same QR implementation for all CLAMP applications. The QR operations that are unique to MyPhpMoney are specified by the appropriate data access policies.

Data Access Policies. Developing access policies for MyPhpMoney was also simple. We identified 7 tables that contain sensitive data. Thus, each policy file contains 7 lines, one for each table. Since MyPhpMoney does not include an administrative interface, we crafted policies for two access classes: `user` and `nobody`. Altogether, this effort required less than an hour.

Appendix B.
Applying CLAMP to HotCRP

We also ported the HotCRP conference management software [10] to CLAMP. HotCRP allows authors to submit papers and PC members to review, comment on, and rank the papers. Porting HotCRP to CLAMP required considerably more effort than our other examples.

User Authenticator (UA). Extracting the user authentication functions for HotCRP was straightforward, supporting our hypothesis that the authentication functionality for most websites is largely self-contained. We copied the login functionality (approximately 40 lines of code) to the UA, and added calls to the UA to one file (`index.php`) that handles user creation, login, and logoff, adding less than 6 total lines of code. In total, creating the UA for HotCRP required less than an hour of effort.

Query Restrictor (QR). As with our previous ports, HotCRP required no changes to the QR. All of the HotCRP-specific knowledge was captured in the data-access policies.

Data Access Policies. HotCRP defines many potential user roles, and it is specifically designed for flexibility, allowing PC Chairs to choose from a variety of security policies. This flexibility adds to the complexity of the software, raising the possibility of information leaks. Indeed, the author of HotCRP expresses a desire for a "flexible information flow control layer" in order to prevent inadvertent information exposure [10].

With HotCRP, a user can be an author, an external reviewer, a PC member, a PC Chair, a Chair's Assistant, or any combination of these. For example, a PC member can be an author as well. To create access policies for all of these potential roles, we first developed policies for users that only fall into one category, for example, users who are only authors. This gave us five access classes. We then developed policies for "hybrid" users who act in multiple roles. Not all permutations were needed. For instance, the PC Chair has full access rights to all of the data in the database. If the PC Chair is also an author, she still retains all of her access rights. On the other hand, PC members typically should not see reviews for papers they have conflicts with, but authors can (after decisions have been made) see the reviews for their own papers. Fortunately, even for these cases, the hybrid policy proved to be relatively straightforward to create, with most of the tables simply using the same restrictions as the more permissive role. In the end, we only added two hybrid access classes.

The real challenge for porting HotCRP to CLAMP came from the extreme flexibility that HotCRP gives to the PC Chair. For example, the PC chair can decide that submissions are anonymous, not anonymous, or optionally anonymous. Similar options are available for reviews. Thus, the definition of the sensitive data CLAMP should protect can change radically based on the PC Chair's choices. As developers unfamiliar with HotCRP, we found it challenging to extract all of this logic from the code and encode it in SQL view restrictions for CLAMP's data access policies. Nonetheless, with only a few days of effort, we created a full set of reasonable policies for HotCRP. Figure 8 illustrates one of the policy statements we developed.

To validate our policies, we asked HotCRP's creator, Eddie Kohler, to review their accuracy. He agreed that the policies seemed reasonable and noted a few mistakes in our initial version. This review highlights several key points.

First, it is quite possible to develop reasonable data access policies even for complex applications with dynamic data access controls. Indeed, in many ways, HotCRP represents an extreme in this regard. Many other applications that handle sensitive data require far less flexibility. For example, a bank will always want users' financial data protected, and is unlikely to purposefully include application options that allow customers to see each other's data.

Second, the errors we did make in our initial policies illustrate that CLAMP can provide significant benefits even if its policies are not completely accurate. A policy may incorrectly limit access to data, in which case security is not harmed, and the missing data will likely be easy to notice and debug. Even when a policy permits access to

```
select
  Paper.paperId, title, ...,
  /* Blank out the outcome field, if authors aren't allowed to see it */
  (if( (select count(*) from Settings where name ='au_seedec') > 0, outcome, 0)) as outcome,
  null as leadContactId, ...  /* Authors can never see the lead PC contact ID */
from Paper
  join PaperConflict as Conf on
    (Conf.paperId=Paper.paperId and Conf.conflictType>=@author and Conf.contactId=UID);
```

Figure 8. **Example HotCRP Access Control.** *This abbreviated statement restricts an author's view of the* `Paper` *table. Individual fields are hidden based on the conference's settings. The rows returned are restricted to papers that were authored (*`Conf.conflictType>=@author` *indicates an author) by the authenticated author (*`Conf.contactId=UID`*).*

data that should be kept private, the policy still protects other data. For example, when writing the `author` policy, we incorrectly believed that the field `leadContactId` in the `Paper` table referred to the lead author, rather than the lead PC member. While our policy would not have protected the user ID of this PC member from a determined attacker, the policy still prevents authors from seeing each other's papers, hides reviews appropriately, etc.

Finally, CLAMP's design consolidates all access control decisions in one place (the QR) in the form of policy files. These files can be independently reviewed for accuracy. This is much simpler than asking someone to learn an entire code base and decide whether the access control decisions sprinkled throughout the code will effectively preserve the secrecy of user data. As a result, independent auditing of a site's security policy becomes more feasible.