xBook: Redesigning Privacy Control in Social Networking Platforms

Kapil Singh*
School of Computer Science
Georgia Institute of Technology
ksingh@cc.gatech.edu

Sumeer Bhola*
Google
sumeer@acm.org

Wenke Lee
School of Computer Science
Georgia Institute of Technology
wenke@cc.gatech.edu

Abstract

Social networking websites have recently evolved from being service providers to platforms for running third party applications. Users have typically trusted the social networking sites with personal data, and assume that their privacy preferences are correctly enforced. However, they are now being asked to trust each third-party application they use in a similar manner. This has left the users' private information vulnerable to accidental or malicious leaks by these applications.

In this work, we present a novel framework for building privacy-preserving social networking applications that retains the functionality offered by the current social networks. We use information flow models to control what untrusted applications can do with the information they receive. We show the viability of our design by means of a platform prototype. The usability of the platform is further evaluated by developing sample applications using the platform APIs. We also discuss both security and non-security challenges in designing and implementing such a framework.

1 Introduction

Social networking sites have transformed the way people express themselves on the Internet and have become a door to the social life of many individuals. Users are contributing more and more content to these sites in order to express themselves as part of their profiles and to contribute to their social circles online. While this builds up the online identity for the user, it also leaves the data vulnerable to be misused, as an example, for targeted advertising and sale.

More private data online has lead to growing privacy concerns for the users, and some have faced extreme repercussions for sharing their private information on these networking sites. For example, students have been fined for their online social behavior [29]; a mayor was forced to resign because of a controversial Myspace picture [32]. There are numerous such cases, and these incidents clearly underline the importance of privacy control in social networks.

With the advent of Web 2.0 technologies, web application development has become much more distributed with a growing number of users acting as developers and source of online content. This trend has also influenced social networks that now act as platforms allowing developers to run third-party content on top of their framework. Facebook opened up for third-party application development by releasing its development APIs in May 2007 [22]. Since the release of the Facebook platform, several other sites have joined the trend by supporting Google's OpenSocial [10], a cross-site social network development platform.

These third-party applications further escalate the privacy concerns as user data is shared with these applications. Typically, there is no or minimal control over what user information these applications are allowed to access. In most cases, these applications are hosted on third party servers that are difficult to monitor. As a result, it is not feasible to police the data being leaked from the application *after the data is shared with the application*. There have been several reported cases where users' private information was leaked by the applications, either due to intentional leaks [21] or due to vulnerabilities in the application [26].

Most social networking platforms, such as Facebook, currently provide the applications with full access to user profile information. This permission is granted in Facebook when the user adds the application, which requires the user to make a trust decision. Setting fine-grained access control policies for an application, even if they were supported, would be a complex task. Furthermore, access control policies are not sufficient in enforcing the privacy of an individual: once an application is permitted by a user's access control policy, it has possession of the user's data and can freely leak this information anytime for personal gains. For example, a popular Facebook application, Compare Friends, that promised users' privacy in exchange for opinions on their friends later started selling this information [35].

In this paper, we are concerned with protecting the users' private information from leaks by third-party ap-

*Part of the work was done when the first author was an intern and the second author was an employee at IBM Research T.J. Watson.

plications. We present a mechanism that controls not only what the third-party applications can access, but also what these applications can do with the data that they are allowed to access. We propose and implement a new framework called xBook that provides a hosting service to the applications and enforces information flow control within the framework. xBook provides three types of enforcement that encapsulate the privacy requirements in a typical social network setting: (1) user-user access control (e.g., access to only friends) for data flowing within one application, (2) information sharing outside xBook with external parties; and (3) protection of the application's proprietary data. While (1) and (2) protects the privacy of a user from information leaks, (3) prevents the application's proprietary data or algorithm from being leaked to the application users.

The third-party applications are redesigned in such a way that they have access to all the data they require (allowing them to perform their functionality) and at the same time, not allowing these applications to pass this data to an external entity unless it is approved by the user. *Our framework enforces that the applications make these communications explicit to the user* so that he is more informed before approving an application.

There are several challenges associated with the design of our xBook framework:
Confinement. The execution of application code needs to be confined. This problem needs to be dealt with independently on the client side within the browser and on the server side in the web server. We use "the web server" as a conceptual entity to represent one or more servers.
Mediation. All communication from and within an application needs to be mediated by the xBook platform for permissible information flow. To this end, we developed a labeling model that enforces user-defined security policies. High-level policies specified by the user are converted to low-level labels enforced by xBook.
Programmability. The programming abstraction to the application writers should be practical and easy to use. xBook provides a set of simple APIs in line with the existing social networking platforms.
Portability. The requirements imposed by xBook on the application design should not break the existing applications. In other words, it should be feasible to port most functionality of typical applications to xBook with little effort.

We show the viability of our framework design by implementing a working prototype of our xBook system and porting some of the popular applications from existing social networks, such as Facebook, on top of the framework. We also demonstrate a practical deployment strategy of our system by porting our framework itself as an application on Facebook. Our system is available online [33]. We evaluate the security of our platform by illustrating some possible application scenarios, and how xBook ensures privacy control in such cases. We also create some synthetic attacks that attempt to exploit the platform to leak information. Our results illustrate that xBook can successfully prevent all such attacks. Our performance results further demonstrate that xBook's privacy control mechanism incurs negligible overhead for typical social networking applications.

The rest of the paper is organized as follows. Section 2 motivates our work by analyzing some privacy issues with the current social networking platforms. We present an overview of our xBook framework in Section 3. Section 4 and 5 discuss the implementation details of xBook's client-side and server-side components, respectively. Our labeling model is described in Section 6. Section 7 presents the evaluation results. We discuss the limitations of our work in Section 8, followed by related work in Section 9. Finally, Section 10 concludes the paper.

2 Background

2.1 Social Networking Platforms

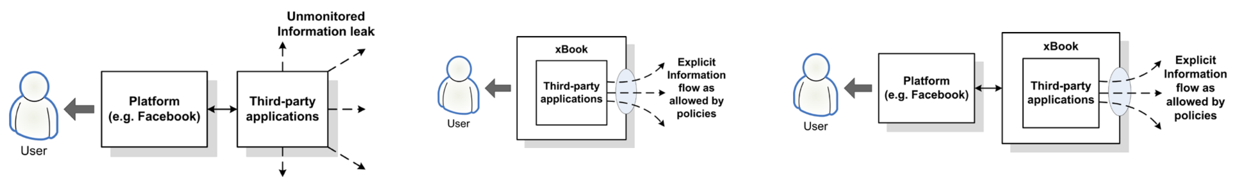
Social networks are the backbone of the online social life of many Internet users. These networks have expanded their development scope by allowing third-party developers to write their own applications, which in turn can be accessed and executed via the social network. An application is an entity that provides some value-added service to the user, and it requires user's profile data to perform its functionality. For example, a simple horoscope application generates daily horoscope based on user's birth information.

Facebook is one popular network that has pioneered the concept of the social network as a platform. The applications bring value both to the platform and its users in providing new features. Applications are deployed on their own servers and Facebook only acts as a proxy for integrating the applications' output to its own pages. The growing popularity of applications on Facebook has enticed other networks, such as Google's Orkut, to start supporting applications. The Orkut platform model is based on the OpenSocial framework [18]. OpenSocial provides a set of APIs for its partner sites (which it refers to as "containers") to implement. An application that is built for one container should be able to run with few modifications on other partner sites. The APIs allow third parties to have access to the social graph and personal user data.

For the rest of the paper, we use the Facebook case as an example; similar concepts apply to other social networking platforms.

2.2 Privacy Issues with Current Designs

Facebook supports customized policies for user-user access control, but currently provides no control on what



(a) (b) (c)

Figure 1: Application architecture for: (a) current platforms. (b) xBook platform. (c) xBook on Facebook.

user profile data can be accessed by third party applications. Applications run on their own servers that have no control administered by Facebook (Figure 1(a)). Applications need data to perform their functionality; they can request user data from the social platform and store it at their own servers. Facebook discourages storing user data on the application's own servers by barring it in their license agreement [5], but there is no way of enforcing it in Facebook's current architecture.

Application developers have access to a user's data even when they are not friends with the user. Unlike a regular friend relationship, this relationship is neither symmetric nor transparent: the application developer has access to the user's information, but the user does not necessarily know who the application developer is.

Before adding an application, the users are required to agree to a service agreement that allows the application to have access to their profile data. This general agreement is presented for every application, and no other specific information is provided about the application. Since a majority of the applications are known not to exploit users' personal data, the users tend to add any application, effectively defeating the purpose behind the service agreement. Additionally, second-degree permissions that allow applications to have access to the profiles of the users' friends add another layer of complexity.

There have been several reported incidents where users' information was leaked due to a vulnerability in the application [26]. The platform is trusting all third party developers, but the trust is misplaced since there is no restriction on who is allowed to develop an application. One of the most popular Facebook applications, TopFriends, had a vulnerability that allowed any user of TopFriends to see the profile of another user, even if they are not friends with each other [26]. Private information of some high profile users was leaked. Facebook's response to this controversy was that they *expect* third party applications to follow their policies, which is not acceptable considering that there is no effective way to police the application developers.

User data has a lot of commercial value to marketing companies, competing networking sites, and identity thieves. Therefore, it is not surprising that many applications have been observed to intentionally leak user data to external parties for profit [21]. Other surveys have also discovered similar violations based on an application's externally-visible behavior [19]. The situation could be even worse as it is not feasible to determine how many other applications violate the user's privacy with internal data collection.

Social networking sites have a responsibility to protect user data that has been entrusted to them. The current approach is to legally bind the third parties using a Terms of Service (TOS) agreement [4]. However, it is not possible to monitor the path of information once the information has been released to these parties. Therefore, social networks can not rely on untrusted third parties following their TOS agreements to protect user privacy. Instead, privacy policies should be enforced by the platform and applied to all data that has been entrusted to the social networking site. Our platform design, `xBook`, is one step forward in this direction.

Felt et al. [19] have proposed a solution to proxy the user information in the form of tags to the third-party applications. These applications do not have access to user data and instead use pre-defined tags to format their output being displayed to the user. Their solution limits the capability of some important and popular applications, such as the horoscope application, that perform processing on user data beyond just displaying it. Our work enforces no such restriction on the application behavior.

3 `xBook` Overview

xBook is an architectural framework for building social networks that prevents untrusted third-party applications from leaking users' private information. The applications are hosted on xBook's trusted platform (Figure 1(b)), and xBook provides complete mediation for all communication to and from these applications.

In a social network setting, an application might communicate with entities outside the xBook system, called *external entities*, to perform specific tasks. For example, the horoscope application may communicate with `www.tarot.com` to receive horoscopes for every sunsign. The application also encapsulates its own data or algorithm that needs to be protected from untrusted users.

In the xBook framework, applications are designed as a set of *components*; a component being the smallest granularity of application code monitored by xBook. A component is chosen based on what information the component has access to and what external entity it is allowed to communicate with. In the horoscope application, one component communicates with www.tarot.com and has no access to user data. Another component has access to user's birthday, but does not communicate with any external entity.

From an end user's perspective, the applications are monolithic as the user does not know about the components. At the time of adding a particular application, the user is presented with a manifest that states what user profile data is needed by the application and which external entity will it be sharing this data with. For example, horoscope's manifest would specify that it does not share any information with any external entity. Note that the horoscope application does not need to reveal that it communicates with www.tarot.com as no user information is being sent to www.tarot.com. The user can now make a more informed decision before adding the application. Admittedly, the user will need to make a trust decision with respect to the parties with which the application shares user data, but these external parties can be expected to be larger and better branded entities providing internet services, such as Google for ads, Yahoo for maps, etc.

Figure 2 shows a typical life cycle of an application. The developer of an application decides on the structure of the components for that application and during the application's deployment on xBook, he specifies the information required by each component and the external entity a particular component needs to communicate with. xBook uses this information to generate the manifest for the application. As shown in the figure, a manifest is basically a set that specifies all of the application's external communications (irrespective of the components) along with the user's profile data that is shared for each communication. Additionally, the xBook platform ensures that all of the application's components comply with the user's privacy policy and the manifest approved by the user. We discuss this further using the case study of an example application in Section 6.3.

The division of an application into multiple components allows the application writer to develop different functionality within an application that rely on different pieces of the user profile. For example, let us consider an application that requires a user's information to generate a customized profile for the user. It also requires his address information to be passed to Google to generate a map showing the address. In the application design of current social networks, the application would be able to pass all information about the user to Google. In the xBook framework, the application would be split into two components: the first component presents the customized profile of the user, has full access to the user's data and is not allowed to communicate with Google; the second component encapsulates the user's address (with no mapping to the user's profile) that is passed to Google to generate the map. We discuss some example applications in Section 7.1.

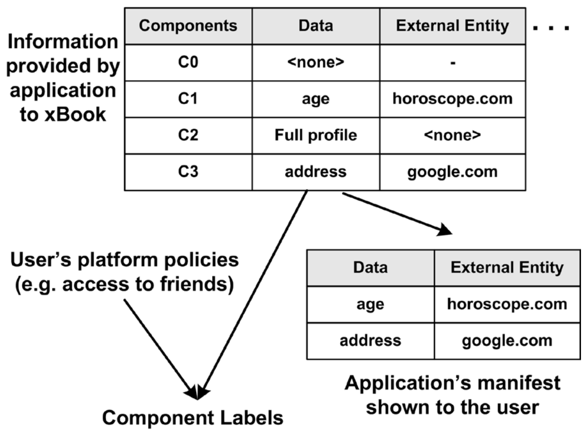

Information provided by application to xBook
Components | Data | External Entity
C0 | <none> | -
C1 | age | horoscope.com
C2 | Full profile | <none>
C3 | address | google.com
. . .
User's platform policies (e.g. access to friends)
Data | External Entity
age | horoscope.com
address | google.com
Application's manifest shown to the user
Component Labels

Figure 2: Typical life cycle of an application in xBook.

Figure 3 shows a high-level design of our xBook framework. There are two parts of the xBook platform, one that runs on the server-side and another that executes on the client-side in the user's browser (Figure 3). The application components, in turn, are also split into client-side and server-side components. The components are written in a safe subset of javascript, called ADsafe [1], which facilitates confinement of these components in our xBook implementation. Any communication to and from the components occurs by using xBook APIs, thereby allowing all such communication to be mediated by xBook. Each component is associated with a privilege level or label that is derived from the application's manifest. The platform mediates the information flow between the components based on these labels (Section 6).

Both client-side and server-side components communicate with server-side storage to retrieve data. There are two types of storage in xBook system: one for storing xBook data that includes user profiles, and second for the data stored by the application. While the structure of xBook data is known, the semantic of the application data is internal to the application and hence unknown to the platform. All data fields are labeled to control access by application components. These labels are assigned based on high-level user-defined policies, such as a policy allowing access to only the user's friends, and the manifest approved by the user (Figure 2).

To store application data with unknown structure and semantics, xBook contains a group of storage pools, where data is stored as a set of name-value pairs. An application can have multiple storage pools, which could be for each user or for user-independent data.

3.1 Leakage Prevention by xBook Design

In the current platform designs, a user's information can be leaked in three major ways: (1) applications can

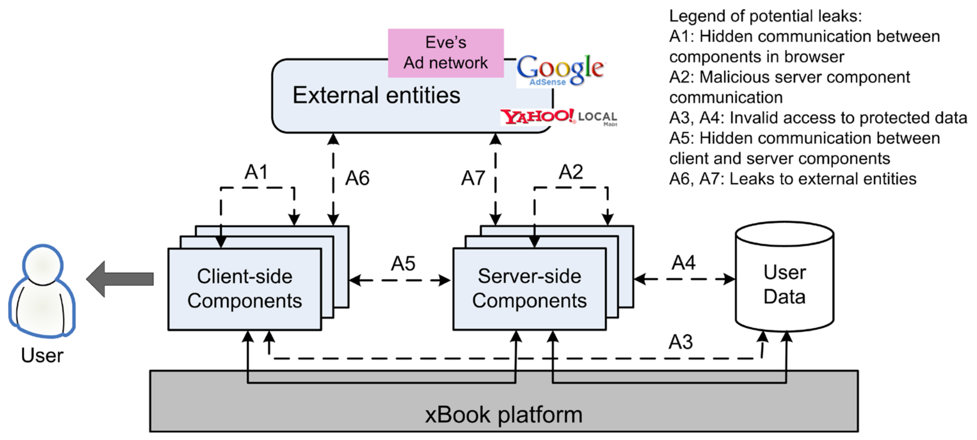

Eve's Ad network
Google AdSense
YAHOO! LOCAL Maps
External entities
Legend of potential leaks:
A1: Hidden communication between components in browser
A2: Malicious server component communication
A3, A4: Invalid access to protected data
A5: Hidden communication between client and server components
A6, A7: Leaks to external entities
A1
A6
A7
A2
A5
A4
A3
Client-side Components
Server-side Components
User Data
User
xBook platform

Figure 3: `xBook` architecture shown along with sources of potential leaks.

share user's information with any third party, including advertisers, or fraudulent parties [21], and as shown in Figure 1(a), there is no way such a leak can be monitored in the current designs; (2) an application can pass information of one application user to another user, breaking free from the platform restriction that only friends can view a user's profile. The reported vulnerability in TopFriends allowed such a leak [26]; (3) the application can recreate the social graph of all its users by connecting common friends as edges in the graph.

xBook's design enforces complete mediation of all communication with the external entities (Figure 1(b)), thus preventing these applications from leaking information, effectively preventing (1) by design. A separate application instance is created for every user, and that instance only has a view of the data accessible to that user. Data access is restricted to allowed user policies, such as access to friends. We mediate any direct or indirect communication between the components of two application instances, thereby deterring (2). (3) is prevented as no single component of an application can have direct access to the data of all its users: a component can only access an anonymized view of this data set (Section 5.2).

xBook, by design, solves most of the leakage problems of the current platforms. However, there are still some potential mechanisms to leak information in our system. We enumerate these possible threats in our formal model and address these threats one by one throughout the paper.

3.2 Formal Requirements

We present a formal model in this section that generalizes xBook's mediation of untrusted third party applications. We use this model to analyze possible attacks, in terms of potential data leaks, under an adversary that deploys an application for collecting users' private data. We also identify a list of requirements that our system should satisfy in order to defeat such attacks. These formal requirements drive the design and architecture of our system.

Consider an application A consisting of a set of client-side components and a set of server-side components. Let U be the set of all users of the platform and Y be the set of all external entities. Suppose the application is allowed to communicate to a set of external entities $X \subseteq Y$ and a set of users $F_u \subseteq U$ for a particular user $u \in U$ who is using the system. Now, we divide the set of all data items D into three categories. First, there is a set of proprietary data or code of the application represented as $d_A \subseteq D$. Second, the set of data items $d_{u \to x}$ belonging to the user $u \in U$ that the application can transfer to the external entity $x \in X$. This set could be in the form of user's age, interests, photos, etc. Third, for an application instance of user $u_i \in U$, the set of data items $d_{u_i \to u_j}$ is what the application can transfer to a user $u_j \in F_{u_i}$.

The platform wants to monitor the occurrence of a set of events E that can pass information outside an application component. Any event $e \in E$ is actively monitored by intercepting the information flow path between the point of the event occurring and the point where the event is handled. The platform monitors the content information I_e contained in the event. We express the response of the platform when the particular instance of the event has potential leaking information as $R(I_e)$, which may include filtering the content, blocking the communication, etc.

We can identify several sources of potential leaks in the xBook system (Figure 3). The first class of attacks (A1) bypasses the active monitoring by the xBook platform to leak private information from one client-side component to another, by creating a prohibited flow. Such attacks exploit some of the abstract features of the development language and the browser to leak information maliciously. In other words, A1 occurs if response $R(I_e)$ is not triggered even if the I_e contains private information content that is being leaked. Similar leaks (A2) are possible on the server-side where application components can break out of the sandbox to create a prohibited channel with other components. In addition, some attacks (A3 and A4) can occur during a component's access to data store, where the component gains access to restricted user or application data. Leaks (A5) can also occur in the communication between client-side and server-side components. Other attacks (A6 and A7) leak private information to entities outside the system. The leaks could be to an $x \in Y$ that is prohibited ($x \notin X$), or it could be leaking restricted piece of information $d \in D$ to an entity via communication that is allowed by the system, i.e., for $x \in X, d \notin d_{u \rightarrow x}$ for a user $u \in U$.

We completely forbid cross-application communication, effectively preventing leaks across applications. We also prevent direct communication between server-side components, only allowing them to communicate via storage, thereby preventing attacks of type A2. We mediate other communication paths based on the labels of the communicating parties (Section 6). We address all other identified classes of attacks in Section 7.3. The requirements of an ideal social networking platform that guides the xBook design are as follows:

- Response $R(I_e)$ is invoked if I_e contains prohibited private information. In other words, the platform should be able to monitor any event that might be potentially leaking information, and should take action to prevent such leaks.
- Applications can invoke an event e iff $e \in E$, i.e., applications are restricted to a limited set of events for passing information to external entities.
- Application component having access to user u's private data d can send information to an external entity $x \in Y$ iff $x \in X$ and $d \in d_{u \rightarrow x}$. In other words, the platform should enforce user policies by limiting the communication to only *allowed* external parties and passing only *allowed* information to these parties.
- Application component having access to user u_i's private data d can send information to another component acting for user u_j iff $u_j \in F_{u_i}$ and $d \in d_{u_i \rightarrow u_j}$. This means that the applications should inherit the user-user access control policies of the platform.
- Application component x can access d_A only if $x \in S$, i.e., only server-side component of the application should have access to application's proprietary data.

We do not cover attacks against the browser in this work and assume that the browser behaves non-maliciously. Although phishing attacks can entice the user in choosing policies that might leak user information, we do not consider such attacks here. This work enforces the policies specified by the user, and does not consider social engineering attacks against the user.

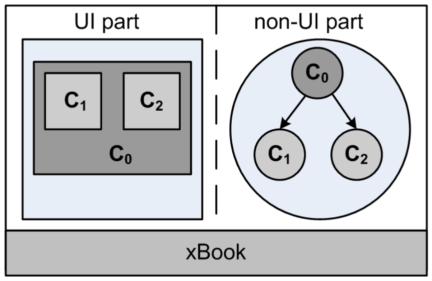


Figure 4: Client-side components in xBook design.

4 Client-side Components

The client-side of the xBook platform and the client components of the applications run within the web browser. The components are further divided into two parts: the user interface (UI) part that is visible as part of the page to the user, and the non-UI part that provides communication interfaces with the external parties and with the server side. There is a one-to-one mapping between the non-UI and the UI parts, i.e., for every non-UI part, there is a corresponding UI part visible to the user (Figure 4).

A component is allowed to create another component. Information can flow during the component creation and this opens up the possibility of an information leak. We prevent such leaks by allowing components to create other components that are at least as restricted as the creating component. This principle prevents the creating component from leaking information out of the system via a less restrictive component.

At the front end, the creating component needs to delegate some screen space to the created component. One challenge is to isolate the third-party application components within the Document Object Model (DOM) of the webpage. A DOM is a platform- and language-independent standard model for representing HTML or XML documents in a browser. We present our confinement approach in the next section.

4.1 Confinement Mechanism

The components of an application encapsulate different levels of private information for the users. Therefore, these components need to be isolated from each other in order to prevent information leaks. On the client side, the components form a part of the DOM of the web page. The web page's DOM may include multiple components from one or multiple applications, apart from the platform's DOM objects.

In the current browser specifications, any script in a page has intimate access to all of the information and relationships of the page. As a result, the components are free to access information about the DOM objects of other components. In order to confine the components within their own control domain, we limit the application code to be written in an object capability language called ADsafe [1]. In an object capability language, references are represented by capabilities and objects are accessed using these references. Other alternatives to ADsafe, such as Caja [25], are also available; we decided in favor of ADsafe due to its simpler design and easier feature addition and customization to meet our system needs.

ADsafe. ADsafe defines a subset of javascript that makes it safe to put guest code (such as third-party scripted advertising or widgets) on any web page. ADsafe removes features from javascript that are unsafe or grant uncontrolled access to browser elements. Some of the features that are removed from javascript are global variables and functions such as `this`, `eval` and `prototype`. It is powerful enough to allow guest code to perform valuable interactions, while at the same time preventing malicious or accidental damage or intrusion. The ADsafe subset can be verified mechanically by static tools like JSLint [8].

ADsafe was initially developed to host untrusted advertising content safely on a webpage. xBook's isolation mechanism is designed with the code base taken from an earlier version of ADsafe. We customized ADsafe by adding code for our component confinement model and mediation based on our labeling model, to prevent information leaks from the "sandboxed" application components. A recent version of ADsafe have since implemented some of our features, but still would require changes to be useful for our system.

One such example is that ADsafe runtime supports only a single level of confinement: all subtrees of the untrusted guest applications exist as children of the trusted web page code. One guest application does not have another guest application as a child to its subtree. In contrast, xBook design requires *nested* DOM subtrees that need to be isolated from each other. Figure 4 shows an example of a nested subtree, where component C_3 is a child of component C_1, which in turn is a child of C_0.

Our requirement is to restrict an application component to within a set of connected DOM elements that form the component. In the current DOM specification, any DOM element can parse through the tree of the page via its parent, children or siblings. We enforce confinement by providing the component elements only with a partial view of the page's DOM and only indirect access to the DOM objects.

Confinement Rule 1. One DOM element belonging to an application component should only access another DOM element of the page (that includes accessing its properties, adding a new element to it, etc.) iff they both belong to the same component.

As part of the implementation, xBook associates each component with a unique *DOM wrapper* object at the time of creation. Figure 5 shows the partial code of our DOM wrapper implementation. Before deploying an application, xBook verifies that each component code is ADsafe compliant. The code must be wrapped in a `<div>` element having an identifier, which forms the root of the component. xBook ensures that this identifier is unique to the application page. The `ADSAFE.go` method gives the component code access to the `API` object that maps to our DOM wrapper object. The `ADSAFE` code ensures that the second parameter passed to the `createDOMWrapper` function is equal to the identifier of the encapsulating `<div>` element, effectively preventing the developer from faking the identity of the components. It also ensures that the DOM wrapper instance gets the right identity of the component's root node.

The wrapper allows an untrusted component to view DOM nodes simply as integer handles; the component has no direct access to the real DOM. To read or modify the DOM, the component code passes the appropriate handles to the wrapper DOM object using the xBook APIs, which in turn interacts with the real DOM. Additionally, element creation and modification are administered using this component-specific wrapper object. For example, `createTextNode` method in Figure 5 would return an integer handle. Since a wrapper instance is identified by its root element `<div>` that is unique, the DOM wrapper object restricts the untrusted component code to interacting only with the portion of the document tree that belongs to that component. All direct accesses to any real DOM elements are forbidden: the wrapper is the only interface for accessing the elements and it is mediated by the xBook platform.

4.1.1 Event Handling

Another possibility of an application breaking the confinement mechanism originates from the way event handling is designed in the current DOM specification.

Every event has a target, i.e., the XML or HTML element most closely associated with the event. An event handler is a piece of executable code or markup that responds to a particular event. Any element of the DOM can register an event handler to receive a particular event type. Since an event generated from within a component can be received outside the component, the flow of events within a DOM needs to be controlled by the xBook platform for any potential leaks.

In the current DOM implementation, it is possible to assign multiple handlers for a given event. It allows a DOM element to capture events during either of the two phases in the event flow. The event flows down from the root of the document tree to the target element in the first phase called *capture*, then it bubbles back up to the root in the *bubbling* phase. An element can receive the event only if it lies in the path between the document root and

```
function createDOMWrapper(compID, root_node) {
   ...
   /* node2Handle returns the integer mapping of the node */
   /* handle2Node returns the node of the integer handle */
    API.createTextNode = function(str) {
      /* check if str is a string type */
      var node = document.createTextNode(str);
      return node2Handle(node);
   };

   API.appendChild = function(node_handle, child_handle) {
      /* check if node_handle and child_handle are valid */
      var child = handle2node(node_handle);
      handle2node(node_handle).appendChild(child));
   };

   API.addEventListener = function(node_handle,
   eventtype, listenfunction, useCapture) {
      /* check if node_handle is an valid handle */
      handle2Node(node_Handle).addEventListener(
      eventtype,
      function(e) {
         /* copy e to new_e while passing only the integer
         handle of the target */
         listenfunction(new_e);
      },
      useCapture);
   };

   API.sendMessage = function(destCompID, message) {
      /* check if destCompID and message are string types */
      /* sendMessage checks validity of information flow
      before passing the message */
      sendMessage(currentUser, compID, destCompID, message);
   };
   return API;
}
```

```
ADSAFE = function() {
/* provides the core ADsafe runtime */
   ...
   return {
      go:function(id, f) {
         /* check if 'id' refers to the <div>
         element (root of the component) */
         var dom = document.getElementById(id);
         if(dom.tagName != 'DIV')
            error();
         /* create the DOM wrapper and pass its
            reference to the component */
         var API = createDOMWrapper(id, dom);
         f(API);
      }
      ...
   }
}
```

Skeleton ADsafe code added to encapsulate a component code

```
<div id="a0C0">
   <script>
      ADSAFE.go("a0C0", function(API) {
         /* create a button with 'Horoscope' label */
         var elem = API.createElement("button");
         API.appendChild(elem,
            API.createTextNode("Horoscope"));
         ...
         /* send a message to component C1 */
         API.sendMessage("C1", "C0 to C1");
         ...
      });
   </script>
</div>
```

Component Code made ADsafe compliant and verified by JSLint

Figure 5: DOM wrapper implementation with sample functions.

the event target.

One of the goals of our event handling model is to keep the functionality of the current DOM model (including preserving the concept of the two stages). Therefore, we specify our event flow model as follows: for any application component, an element can receive an event iff it lies in the path between the *root of the component* and the target element for the event. We still need to restrict this access to a single component so that no outside component can receive the event; we provide such a restriction by the following confinement rule:

Confinement Rule 2. A DOM element belonging to an application component can receive an event iff the event target belongs to the same component.

We implemented our event handling model using the DOM wrapper object introduced in the previous section. As shown in Figure 5, the object makes a wrapper to the event handling interface available to applications. The wrapper receives the event from the browser's DOM implementation and filters the information presented in the received event object before passing the event to the applications. Any information about the real DOM elements, such as the handler to the target element, is filtered; this prevents application's component code from breaking the confinement. The `addEventListener` method copies the received event `e` into `new_e` while transforming the real DOM element references to wrapped integer values. The xBook platform mediates the event delivery and as a result, ensures that an event can only be received by elements that belong to the same component that contains the target, thereby enforcing the second confinement rule.

4.2 Communication with External Entities

It is common for the applications to communicate with external parties to perform specific tasks. One typical example is the use of Google map APIs to generate maps of some address known to the application [9]. In other cases, a user's date of birth is used by applications to contact external providers to generate horoscopes [3]. What we achieve in our architecture as compared to the existing social networking platforms is that *we enforce the applications to make these communications explicit* so that more informed decisions can be made. The user or the platform can decide on the policies regarding which external entities are allowed to receive what piece of the user's private information. These policies could be coarse-grained for all applications of a user or fine-grained specific to each application. xBook ensures that the information flows from a specific application component to an external entity according to the defined policies.

There are two kinds of communication flows that can happen in our system:

Symmetric communication in which the response is received by the requesting component. This is a typical case for most client-server communication in which there is a two-way exchange of information between the two parties.

Asymmetric communication in which the response is not received by the component that made the request, but is handled by another component of the application. Our motivation for supporting this type of communication is to enable some specific application scenarios. One motivating example is the advertising scenario where advertisements are generated by external parties based on the information passed to them: Google generating advertisements based on the address passed to it. These external party advertisements are typically in the form of links that users click to access the related site. If we design this scenario using symmetric communication, these advertising links would not work, since the receiving component has been restricted to communicate only with Google and not any other party. In order to solve this problem, we can create another application component that is considered part of Google's trust domain; since Google servers are unconfined or public from xBook's point of view, the created component is also unconfined. We do not allow any other application component to peek into this new component or disrupt its integrity. Since we are only showing Google's view in this component and the application is not allowed to change this component, this component maintains the trust level of Google. The new component is placed in an `iframe` with its own DOM and hence cannot communicate with any other component. However, since the component is unconstrained, it is allowed to communicate with any external entity and as a result, the advertising links would work.

4.3 Communication between Components: Message Passing Interface

xBook exposes a one-way message passing API that the components use to pass messages to other components. We implement this interface using the DOM wrapper object as shown in Figure 5. The platform mediates this communication and ensures that the information flow model is enforced. Since each component is associated with a unique wrapper object that is used to send the message (Section 4.1), the sending component of the message can not fake its identify to fraudulently pass the information flow checks: as seen in Figure 5, the value of `currentUser` and sender's `compID` are implicitly provided by the wrapper object to xBook's `sendMessage` function. A component can register a message listener with the platform through the xBook API. Any message intended for a particular component is delivered to its message listener. Since the platform knows the identity of each component, it makes sure that the message is delivered to the right component.

The purpose of our message passing interface is to allow xBook-mediated communication among untrusted components of an application, while still preventing creation of any hidden channels. To this end, we needed to evaluate some of the features of javascript that gives application writers alternatives to pass hidden information in the messages.

Javascript is a weakly typed language and allows any property to be added to any object. For example, an object `message` can take a property `foo` using `message.foo = value;` where `value` could be a number, string or any other object type. Since all application components run in the same scope, a component can pass information to another component if it has access to an object of that component. Let us assume that a component C_1 is allowed to talk to another component C_2 as per the information flow policies, but C_2 can not communicate to C_1. Effectively, we have a one-way communication channel from C_1 to C_2. If C_1 passes the object `message` to C_2, the platform can observe `message`, but cannot identify the object handler `foo` being passed. C_2 can pass information to C_1 by writing to this handler.

We counter such leaks by limiting the message passing to being a JSON container [7], that is pure data. A javascript JSON container is a collection of key/value pairs or an array of values. These key/values are limited to pure data types such as string or numbers. We make a copy of the JSON object and pass the copy to guarantee that there are no additional properties in the passed object. This solution is also effective against attacks by a message sender that use getters and setters.

The simplest way of designing the message passing interface is to pass messages from a source to a destination in a single thread of execution. This option opens up the possibility of a covert communication channel from a more restricted to a less restricted component. For example, let us consider that a less secret component C_0 is passing multiple messages to a more secret component C_1. Because of the single-threaded non-preemptive nature of javascript, C_1 will complete processing the first message before the control goes back to C_0. This creates a covert timing channel from C_1 to C_0. The amount of time taken by C_1 can be observed by C_0 and C_1 can change this time to pass the desired information bits to C_0.

We reduce the effect of this timing channel by making the message passing interface asynchronous. We achieve asynchronous behavior by implementing a global queue for message passing that is shared among all the components of an application. The receiving components register listeners with the platform in order to receive messages. A timer event dequeues an available message and delivers it to the message listener of the target component of the message. Note that addressing all covert channels in our system is beyond the scope of this paper; we discuss this further in Section 8.

5 Server-side Components

The server-side of the application contains the main functionality for typical applications. It follows a familiar web server model where a server-side component is instantiated for every client request.

Besides the regular user-specific components on the server side, there are certain components that are user independent and works on non-user data or user public data. These components perform two tasks: First, they communicate with external parties to provide functionality independent of the user data. Second, they handle statistical aggregation on user data sets. We discuss declassification based on data anonymization in Section 5.2.

The server components also protect application proprietary data that needs to be declassified before sending it to the client. The threat model is reversed in this case: the applications do not trust the user for their data, so they protect their internal data from being leaked to the users. For example, an application might be giving horoscope predictions to users based on their birth date, but it wants to protect the data or algorithm used for such predictions.

There is no direct communication between the server-side components: all such communication happens via application-specific storage. The platform ensures that the information flow is enforced while accessing the database. The platform also administers the communication with external parties and client-side as allowed by the labeling system.

5.1 Component Confinement

The server-side components need to be isolated from each other. The server-side of xBook mediates all communication flowing in and out from these components. There are several options available for server-side isolation. Operating system isolation mechanisms [12, 30] can be used to sandbox the application components. Another option is a language level confinement similar to the client-side isolation with options like Caja (Javascript) [25], ADsafe (javascript) [1] and JoeE (Java) [20] available. We use ADsafe on the server-side in order to have the same language for developing application components for both client and server.

To the best of our knowledge, we are the first ones to port ADsafe to the server side. We had to make some modification to the ADsafe object to implement our server-side xBook APIs and to perform checking of the information flow labels. Each server-side component holds a unique handle to the modified ADsafe object, and access is restricted to the set of APIs provided by the modified ADsafe object. The modified ADsafe object is conceptually similar to the DOM wrapper object on the client side, but is customized to work in the server-side environment. The platform verifies the validity of the information flow before any access is granted. The javascript execution environment is provided by Helma [6], a popular open source web application framework.

5.2 Anonymized Statistics

xBook ensures that no user data is leaked against the user's policies. A particular instance of an application can only have access to profile data that belongs to the user and only his friends. Different instances of the applications cannot share data due to the restrictions posed by xBook's labeling system.

It is desirable for some applications to have a view of all its users so that some statistical results can be published for the whole application. In other words, a component of the application needs to receive data of all the application users and still should be able to share these statistics as output to all users, crossing the boundary of friends.

In order to facilitate this case, we are exploring a three-step anonymization algorithm that provides conservative access to data for the applications. Currently, case 1 and 3 have been implemented, case 2 will be explored as part of our future work.

Case 1. If an application component requests a single field of user information for all application users, it is given access to the requested set in an unmodified form, but in a random order of sequence.

Case 2. If an application component requests multiple fields of user information for all application users, it is given access to the requested set in a form generated by anonymizing the original dataset and then randomizing the resulting tuples' order of sequence. We plan to leverage some of the existing work [15, 24, 31] to generate the anonymized statistics. We acknowledge that providing security in anonymity and statistical queries is a challenging problem and has its own limitations [13, 24]. Addressing these limitations is orthogonal to our work and is not the focus of this paper.

Case 3. Applications can also request the xBook platform for statistics on unanonymized data. This gives the applications more accurate statistics as compared to case 2, where some fields might be filtered or altered to preserve anonymity. xBook provides a limited list of such operations, including aggregation, maximum and minimum value over one or multiple fields.

Discussion. Anonymizing the data might limit some applications that rely on the original data for their functionality. One such example is an application that plots the location of a user's friends on Google maps, and would need to pass names and addresses of the user's friends to Google. The application also makes subsequent queries to Google (for example, to build a Google calendar of friends' birthdays). If the data is anonymized, the application might not produce completely accurate results.

On the other hand, if Google is provided with unanonymized data, it can use the data to cross-reference and identify the friends. This is a conflict between privacy and functionality. If functionality is preferred and unanonymized information is passed to external entities, user's personal information can be leaked. In such a case, our xBook design, at the minimum, enforces the applications to explicitly declare all external communication (including the data that will be transferred). Based on such information, the user can make a much more informed decision about adding the application.

6 Labeling Model

The xBook platform tracks and enforces information flow using a labeling system defined based on existing models [17,23,27,36]. All system abstractions are layered on top of two types of entities – active and passive. Application components represent active entities that actively participate in label compatibility checks; database entries are passive entities. Every active entity corresponds to a principal and a label; passive entities only have a label.

We do not enforce information flow at the language level [27], but instead at the level of application components and database entries. There are multiple reasons for this choice: (1) it is simpler for the application programmers as they do not need to learn a new language or perform fine-grained code annotations, (2) information flow on a language like javascript with dynamically created source code may not be feasible, and (3) run-time information flow at fine-grained language level would probably be expensive as compared to a much coarser level of components.

The label specifies the secrecy level of an entity. It represents what information is contained in a passive entity and what information the active entity currently has or will read. The entity's principal defines whether the entity has declassification privileges over the label. xBook labels originated along the lines of the language based labels in Jif [27]. Labels represent the confidentiality or secrecy level of an entity in the system. Integrity labeling is not the focus of this work since we are focusing on privacy.

A label L is represented as a set of tags, with each tag having one principal as owner o and another set of principals called readers $R(L, o)$. The owner is the principal whose data was observed in order to construct the data value. The readers represent principals to whom the owner is willing to release the information. An example of a typical label is $L = \{o_1 : r_1, r_2;\ o_2 : r_2, r_3\}$, where $O(L) = \{o_1, o_2\}$ denote the owner set for the label and readers sets are $R(L, o_1) = \{r_1, r_2\}$ and $R(L, o_2) = \{r_2, r_3\}$.

In the xBook system, principals represent the identities of various entities in the labeling model. There are five types of principals in our system:

- $C(a_i, u_j)$ and $S(a_i, u_j)$ represents the client-side and server-side components for an application a_i specific to a user u_j.
- $C(a_i)$ and $S(a_i)$ represents user-independent client-side and server-side components for an application a_i.
- u_j represents the entities that the user u_j is in complete control of. Once the user u_j is logged into the xBook system, the user's browser is assigned the principal u_j.
- $\top$**,** $\bot$ where $\top$ is highest priority principal in the system and is allotted to the xBook platform. For the sake of completeness, $\bot$ is the least privileged principal.
- External entities also have principal names that contain the hostname and optionally the scheme and port (like in URLs). For example, `https://www.example.com:8888` represents one such principal.

Our model assumes static labels for the entities and information flows from one entity to another if allowed by the label comparison of the end points. Information can flow from one label L_1 to another label L_2 only if L_2 is more *restricted* than L_1 denoted as $L_1 \preceq L_2$.

Restriction. $L_1 \preceq L_2 \Longleftrightarrow O(L_1) \subseteq O(L_2)$ and $\forall o \in O(L_1), R(L_1, o) \supseteq R(L_2, o)$

6.1 acts-for Hierarchy

To facilitate easier conversion of user policies to low-level labels, system entities are statically labeled. We decided on immutable labels since it improves usability of the application programming model from the perspective of the application programmer. Unexpected runtime failures can occur when labels of components change at runtime [23]. With immutable labels one can statically verify that all the communication dependencies with respect to other components, external entities, storage will be satisfied.

Some principals have the right to act for other principals and assume their power. The acts-for relation is transitive, defining a hierarchy or partial order of principals [17]. The right of one principal to act for another is predefined by the platform. Figure 6 presents the acts-for relationship within the xBook system. This hierarchy defines the priority of different principles in the system. The reasoning behind the defined hierarchy is as follows:

- $\top$ defines the xbook platform and has the highest security label. As a result, it can declassify any label.
- Any data sink or source that is not explicitly defined by xBook is modeled as an unprivileged entity with label $\bot$.
- The client-side components are given lower priority than server-side components, because intuitively server-side components residing on xBook servers

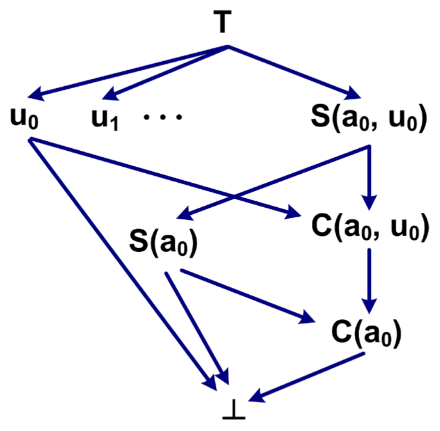


Figure 6: Label hierarchy in `xbook`.

Algorithm 1 Label Compatibility Check Algorithm.

```
$eL_1$ = ($entity_1$ is a database) ? $L_1$ : maxDeclassify($L_1$, $P_1$)
$eL_2$ = ($entity_2$ is a database) ? $L_2$ : maxRestrict($L_2$, $P_2$)
if $eL_1 \preceq eL_2$ then
    ALLOW flow from $entity_1$ to $entity_2$
else
    DENY flow
end if
```

Figure 7: Algorithm to check if the information flow from $entity_1$ to $entity_2$ is allowed.

are more trustworthy than client-side components. For example, $S(a_0, u_0)$ has higher priority over $C(a_0, u_0)$ for application a_0 and user u_0. The server-side components can declassify an application's proprietary data, which has been labeled in a manner such that it cannot be directly read by client-side components.

- User-independent principals are at a lower priority than any user-specific principal. This allows user-specific components to read user-independent data generated by an application, also effectively allowing users to read statistical data generated for the whole application.
- Principals representing the end user are higher than the corresponding client-side principals since the user controls the client.

6.2 Flow Enforcement

Information flows within the xBook system if the label of source is less restricted than that of destination. Such flow restrictions have been proposed earlier in classical information flow control models [14]. We introduce the concept of endpoints similar to the Flume model [23]. Instead of changing the labels of the entities, for every communication the source and the destination create an endpoint each to facilitate the flow. The entity, based on its principal, can restrict or declassify its label and allocate it to an endpoint for communication. While restricting a label means adding more owners and removing readers, declassification either adds some readers for an owner o or removes the owner o. This relabeling can be done only if the principal of the entity is higher than an owner o in the hierarchy.

Figure 7 shows our flow enforcement algorithm, where maxRestrict and maxDeclassify are defined as:

- ***maxRestrict(L, P).*** $O(L) = O(L) \cup descendent(P)$; $\forall o \in descendent(P)$: $R(L, o) = \{\}$
- ***maxDeclassify(L, P).*** $\forall o \in O(L)$: if $(o \in descendent(P))$ then $O(L) = O(L) - \{o\}$

where $descendent(P)$ represents all descendents of a principal P in the acts-for hierarchy, $O(L)$ is the set of owners for label L and $R(L, o)$ represents a set of readers in label L for owner o. Intuitively, the communicating end points support the communication with the sender declassifying its label to the maximum possible using $maxDeclassify$ and the receiver restricting its label using $maxRestrict$. Since the information can only flow from a less restricted to a more restricted component, these functions facilitate the flow of information.

Some typical flows in the xBook system are depicted in Figure 8. To demonstrate the validity of our algorithm, let us consider the example of the flow between the client-side component C_1 and the server-side component S_1. For the flow from S_1 to C_1,

$$\begin{aligned} eL_1 &= maxDeclassify(\{S(a_0) :; \top : C(a_0, u_0)\}, \\ &\quad S(a_0, u_0)) = \{\top : C(a_0, u_0)\} \\ eL_2 &= maxRestrict(\{\top : C(a_0, u_0)\}, C(a_0, u_0)) \\ &= \{C(a_0, u_0) :; C(a_0) :; \top : C(a_0, u_0)\} \end{aligned}$$

Recollecting the definition of restriction, we can see that $eL_1 \preceq eL_2$, therefore S_1 can send data to C_1. Considering the reverse flow from C_1 to S_1,

$$\begin{aligned} eL_1 &= maxDeclassify(\{\top : C(a_0, u_0)\}, C(a_0, u_0)) \\ &= \{\top : C(a_0, u_0)\} \\ eL_2 &= maxRestrict(\{S(a_0) :; \top : C(a_0, u_0)\}, S(a_0, u_0)) \\ &= \{S(a_0, u_0) :; S(a_0) :; C(a_0, u_0) :; (a_0) :; \\ &\quad \top : C(a_0, u_0)\} \end{aligned}$$

We can see that $eL_1 \preceq eL_2$, i.e., C_1 can send data to S_1. Effectively, there is a two-way communication between C_1 and S_1.

6.3 Case Study: Horoscope Application Lifecycle

An application's lifecycle consists of three steps: the application being hosted by xBook, a user adding the ap-

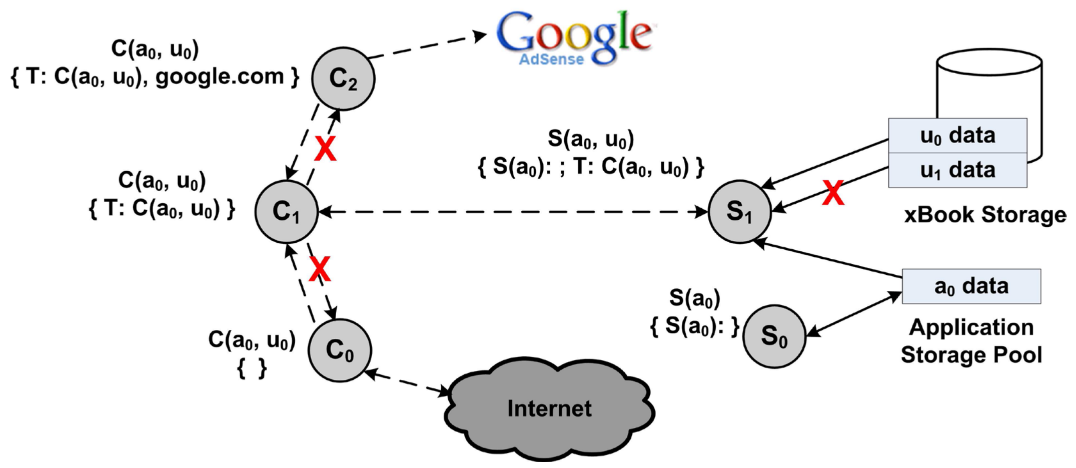


Figure 8: Typical Flows in `xBook` system with the corresponding labels. For every component, the first parameter is the principal and the second is the label associated with the component.

plication and then the user accessing it.

Hosting. Before xBook accepts a new application, the developer needs to provide the following information:

- The application provides the components to be deployed, in each case specifying if the component is client-side or server-side and if it is user-dependent or not, what user data would the component require and which external entities and other components will it communicate with. In our horoscope example, there are three components: S_0 communicates with `www.tarot.com` and requires no user data; S_1 requires user's birthday; C_1 is on the client-side and also requires user's birthday.
- The application also states that there are user-independent or user-dependent storage pools and each is named declaratively by the application. This ensures that the storage pool names do not leak any user information, as the application has no user information at this time. For example, horoscope application declares a storage pool for storing its application data generated by S_0.

Based on the label of the user data, xBook derives the labels and the principals of the components. The birthday field has a label $\{\top : C(a_i, u_j)\}$, therefore the following labels are allocated to the horoscope components:

- S_0 *Principal:* $S(a_i)$, *Label:* $\{S(a_i) : \}$
- S_1 *Principal:* $S(a_i, u_j)$, *Label:* $\{S(a_i) : ; \top : C(a_i, u_j)\}$
- C_1 *Principal:* $C(a_i, u_j)$, *Label:* $\{\top : C(a_i, u_j)\}$

The principals define if the component is server-side or client-side, and if it is user-dependent or not. The labels allow S_1 and C_1 to read the birthday field. S_0's label allows it to declassify itself to be public to communicate with `www.tarot.com`, and write to the storage pool that is given S_0's label. The storage pool label prevents any of the client-side components (C_1) from viewing this data, thereby protecting application data from untrusted users. S_1 is allowed to read from the storage pool. The labels of S_1 and C_1 correspond to the labels of S_1 and C_1 respectively in Figure 8, where $i = 0$ and $j = 0$. As we have observed in the last section, the labels of S_1 and C_1 effectively allow a two-way communication channel. Thus, S_1 can pass the results to C_1 that, in turn, can present a formatted form of the horoscope to the user's browser.

Application Addition. When the user is adding the application, he is provided with a manifest that declares what information is passed to which external entity. xBook derives the manifest from the component information provided by the application developer. For example, since none of the components of the horoscope application share any user information with any external entity, horoscope's manifest would specify that it does not pass any information to any external entity. Since the user's birthday is not shared with any external entity, the application does not need to declare its need to access the birthday information.

Application Access. When the user is accessing an application, all user-specific components are instantiated for that user, replacing the user wildcard in the template of labels and principals with the user identifier. This enforces access control across multiple users: access is only granted if it is aligned with the user's privacy policy, for example, access is granted to only user's friends.

7 Evaluation

7.1 Prototype System and Example Applications

We developed a working prototype of the xBook system, which includes platform code and APIs for developing third-party applications. We also implemented the labeling model that enforces information flow control for

Attack Step	Attack Type	Prevented by xBook?
One client component accessing another component's DOM object	A1	$\surd$
Leaks via the message passing interface	A1	$\surd$
A component creates or destroys a less restricted component leaking information	A1	$\surd$
Retrieve information of another user not in the friend list	A3/A4	$\surd$
Client component retrieves more restricted information from the server	A5	$\surd$
Leaks to an unknown external entity	A6/A7	$\surd$
Leaking restricted information to an allowed external entity	A6/A7	$\surd$

Table 1: Prevention of information leaks against various kinds of synthetic attacks.

the data flowing through the system and prevents any information leaks. Our xBook platform consists of about 4300 lines of javascript code.

We developed two sample applications using the xBook APIs to show the ease and viability of application development in xBook. These applications are similar in functionality to two popular Facebook applications: Horoscope [3] and TopFriends [11].The horoscope application produces a user's daily horoscope based on his birthday information. The utility application based on TopFriends produces a customized profile for the user based on his complete profile information. It also generates a Google map showing the user's home location on the map. The applications are written in javascript using xBook APIs, with the horoscope application having about 180 lines and the application based on TopFriends having around 480 lines of code. We tested these applications against a series of synthetic scenarios, where these applications tried to leak the user's private information. Our tests showed that the xBook system was successful in detecting and preventing all such leaks.

7.2 Porting xBook on Facebook

In order to show the practical viability of the system and to demonstrate that our system can be incrementally deployed, we ported the xBook platform as an application on Facebook. Since Facebook allows any application to have access to user data, including their friends' data, of any user adding the application, xBook as an "application" is able to receive the data of the users agreeing to use the xBook platform. Applications developed using xBook APIs can execute on top of xBook, while still running on xBook servers. Since xBook act as an application for Facebook, xBook's response would be rendered as part of Facebook's web page. Since the third party applications are encapsulated in the page forming xBook's response, the output of these applications would also be effectively rendered on Facebook (Figure 1(c)). Facebook provides the data feed to xBook, which then enables access to this data for xBook applications in a controlled manner through xBook APIs. Facebook's user identity is maintained within xBook. Our running system is available online on Facebook [33].

We envision xBook to be assimilated into the Facebook platform with Facebook providing two levels of application service. First, the current applications based on current Facebook design would be supported. Second, applications that are developed using xBook APIs are supported, with added privacy protection advantage. Users can be given the discretion to choose between the two options, and the users' choice can drive new application development on xBook.

7.3 Security Analysis

Our analysis shows that xBook prevents the applications from leaking any user information. All of the documented leaks in the current social networks are prevented in the xBook system. For example, the TopFriends leak [26] cannot happen in our system because a separate application instance is created for every user. Each instance only has view of the data accessible to that user and xBook mediates all cross user data accesses.

We evaluated the privacy protection ability of our system in three steps. First, we analyzed the security of the xBook design in view of the potential leaks specified in the formal model (Section 3.2). Second, we developed a set of synthetic attacks targeting the xBook framework and performed experiments to show that our prototype successfully prevents these attacks. Finally, we prove that xBook's information flow model ensures that information leaks cannot happen in the xBook design.

We first analyze the security of our prototype and show that all the attacks discussed in Section 3.2 will not succeed against our design. Attack type A1 is prevented due to the various mechanisms developed in our system for client-side confinement (Section 4.1), such as component isolation, event handling, etc. A2 is prevented by server-side confinement of application components, only allowing them to communicate via storage. Leaks via A3 and A4 are inherently prevented by mediating the information flow from the database to application components with label enforcement based on user-defined policies, and also by anonymizing data for statistical purposes (Section 5.2). A5 is also prevented by label enforcement before the client-side request is passed to the server-side component and before response is returned. Enforcing the confinement model to mediate the external communication, both in synchronous and asynchronous communica-

Application	User latency	Server processing time	Time for label checks (Number of checks)	Overhead
Horoscope	183.1ms	128.8ms	7.7ms (6)	4.2%
Map utility	111.4ms	51.2ms	3.5ms (2)	3.1%

Table 2: Performance results of various operations in typical xBook applications.

tion scenarios, prevents A6 leaks (Section 4.2). Following the same lines, A7 is prevented on the server-side.

Second, we tested the ability of our prototype by creating synthetic exploits that try to break out of xBook's information flow control model to leak user information. We developed a sample application to launch these attacks against our prototype; if successful, these attacks allow the application to leak information to entities outside the system. Table 1 contains the results of testing our prototype against a wide range of these synthetic attacks. In all our experimental tests, xBook successfully prevented the leaks before the information could be passed outside the system.

We can also prove that if xBook's confinement mechanism is correctly enforced, the information model ensures that no user information is leaked to external entities (Theorem 1) and to any other user (Theorem 2) outside the user-defined policies.

Theorem 1. *Given a set of policies $P = D \times X$, where the application can pass user's information field $d \in D$ to external entity $x \in X$, and assuming that the intended confinement is enforced, the information flow model ensures that there is no possible leak outside the xBook system. In other words, if $(d, x) \notin P$ then $\forall C_i : C_i \not\twoheadrightarrow^d x$, where C_i are application components and $C_i \not\twoheadrightarrow^d x$ shows that C_i can not pass data item d to x.*

Proof. Let $C^0, C^1, \cdots C^k$ represents the information flow path of a data element d from the xBook database to external entity x.

We present the proof by contradiction. Let us assume that C^i can pass any information (represented by $*$) to x, illustrated as $C^i \xrightarrow{*} x$. This communication is monitored by our xBook platform, but the platform does not know the semantics of the information being passed.

Also, $\forall i \in [0, k] : C^{i-1} \xrightarrow{*} C^i \Longrightarrow L^{i-1} \preceq L^i$ (flow is a restriction)

$C^i \xrightarrow{*} x \Longrightarrow L^i \preceq L^x$

Therefore, $L^{i-1} \preceq L^x \Longrightarrow C^{i-1} \xrightarrow{*} x$

Continuing this by induction, $C^0 \xrightarrow{*} x$

In our labeling model, the computational granularity is at the component level. Therefore, we consider that $\forall C_i$: $Output(C_i) = \digamma(Input(C_i))$ for any computation $\digamma$.

For component C^0, $Input(C^0) = d$, $Output(C^0) = *$ $\Longrightarrow * = \digamma(d)$

Since the input to C^0 is supplied by the xBook platform, and since $(d, x) \notin \mathbb{P}$, $C^0 \not\twoheadrightarrow^* x$.

This is a contradiction. Therefore, $C^i \not\twoheadrightarrow^* x$.

By definition, $*$ represents any information (including d). Therefore, $C^i \not\twoheadrightarrow^d x$.

Theorem 2. *Given a set of user policies $P(x) = D \times U$, where the application can pass user $x \in U$'s information field $d \in D$ to another user $y \in U$, and assuming that the intended confinement is enforced, the information flow model ensures that user-user access control is enforced in the xBook system. In other words, if $(d, y) \notin P(x)$ then $\forall C_i(x), C_j(y) : C_i(x) \not\twoheadrightarrow^d C_j(y)$, where $C_i(x)$ and $C_j(y)$ are components of application instance for user x and y, respectively.*

Proof. Similar to Theorem 1.

7.4 Performance Estimates

xBook does not impose a substantial burden on the performance of the third party applications. With an architectural framework of developing applications, it is difficult to accurately predict the impact of our design on the performance of these applications as perceived by the user. To get a rough estimate of the cost of supporting the xBook design and the overhead involved in our system, we conducted some experiments with our sample applications, measuring latency at the user end and overhead imposed by the mediating design of xBook.

The xBook server side is hosted on a 2.4GHz Pentium 4 machine with 512MB of RAM. The requests are made from Firefox 3.0 browser on a 2.33GHz, 2GB RAM, Pentium Core Duo laptop. Each test was run 10 times and values were averaged. We define user latency as the difference in the time when the request is made at the browser and the time at which the response is received by the browser. Table 2 shows the time required by xBook's information flow control in comparison to the user's overall latency. Server processing includes the application's logic, database access to retrieve required user data, and xBook flow checks, and is independent of the network latency experienced by the application. We instrumented our code to derive the time for performing label checks in the system, and measured overhead as a function of the label checking time over the total latency experienced by the user. Our results show that the overhead introduced by xBook's label checks is considerably small: about 4% for the horoscope application and 3% for the map utility marking user's hometown location on Google maps.

On a cluster of commercial servers with much better computational capacity, these values will be even smaller. Although it is not possible to precisely determine the cost of our approach without a large scale experiment, both the details of our design and the results from these experiments, support the conclusion that xBook design would not substantially increase the latency experienced by users.

8 Discussion

In this section, we discuss the limitations of the application design in xBook and address some of the challenges arising from the new requirements imposed by our design.

Our xBook design imposes no limitations on applications that follow a "pull model", i.e., xBook would preserve the functionality of applications that only receive data from external entities without passing any private information to these entities. Our horoscope application is an example of such as application: one public component of horoscope pulls horoscope data from `www.tarot.com` and does not pass any of the user's profile information. Note that the xBook platform does not need to sanitize the request parameters (in both GET and POST requests), as the component making such requests has no user information that can be leaked. Another component, which has access to the user's birthday information, uses the data to calculate the daily horoscope corresponding to the particular user. This component has no communication with any external entity.

On the other hand, our design might limit some of the applications that require data to be sent to external entities for receiving user-specific information. One typical example is the use of Google APIs to generate maps: it requires a location to be passed to Google before the map is generated. In many cases, we expect these external entities to be larger and well branded entities, such as Google, Yahoo, etc. Such cases could be whitelisted after explicit approval from the user. Note that xBook makes no recommendation about which websites can be trusted, including Google and Yahoo; such trust decisions are made by an individual user from his own knowledge and experiences. Our xBook system can keep track of these approvals across applications for every user, so the users need to approve an interaction only once.

Any social networking application would follow either the pull model or the push model to get data from external entities. In both cases, our platform enforces the applications to make all such interactions explicit and allows the user to make a more informed decision based on the information available. We argue that an application using the pull model would be more acceptable to the users as it requires minimal trust decisions from a user's perspective. It is possible to transform many of the current social networking applications that use the push model to start using the pull model. We acknowledge that such a transformation would require some changes to the application design, and in some cases, such transformations might not be practical due to large download size of the required data. However, if enough users decide not to use the application in view of privacy concerns, it would motivate the developers to consider such a transition.

Our system also suffers from classical covert channels, e.g. timing, memory, process, etc. However, in general these channels have limited bandwidth and viable approaches such as randomizing the time (for example, the delivery time of our message queue discussed in Section 4.3) can further limit their utilities. We plan to study some of these channels as part of our future work.

Scalability of the applications is not a concern in our system: applications hosted on clusters outside xBook would now be hosted on clusters inside the xBook platform. The application developers are already paying for hosting their applications, in most cases to third-parties or cloud owners like Amazon EC2 [2]. Thus, instead of the developers paying to these parties, they would be paying to xBook for the hosting service. xBook, in turn, can outsource the hosting to third-parties, still assuming control of the hosted applications.

We also propose a hybrid model where only the application components that require access to xBook's private data needs to be hosted at the xBook servers. Other public components can be controlled by the application developers on their own servers. Such an approach is useful for many applications as research has shown that a large number of applications do not use any private data to perform their functionality [19].

9 Related Work

Information flow control at the language level has been well studied [16,27]. Jif is a Java-based programming language that enforces decentralized information flow control within a program, providing finer grained control than xBook [27]. In comparison to these language level techniques that require the applications to be rewritten, the xBook platform provides a simpler interface to the application programmers: they do not need to learn a new language or perform any fine-grained code annotations. Additionally, information flow on a language like javascript with dynamically created source code may not be feasible. Cong et al. [16] presented a technique of writing secure web applications, which generates javascript code on the client side and java code on the server side. However, the applications are still written in the Jif language.

There are other systems [23, 36] that have utilized the information flow concept to control data flow at the operating systems (OS) level. Information flows are tracked at low-level OS object types such as threads, processes, etc. xBook works at a much coarser level at the applications, with smallest unit of information being an application component. As a result, run-time information flow in xBook would probably be less expensive as compared to a much finer granularity level used in these systems. In order to make these systems useful for a typical social networking environment, it would require the systems to be installed at a user's computer because leaks can also happen at the browser, which might not be feasible. In comparison, xBook runs on a typical web server without any changes to the OS environment.

Similar to the ADsafe environment, other safe subsets of programming languages, such as JoeE [20] (for java) and Caja [25] (for javascript), allow third-party applications to provide active content safely and flexibility within the existing web standards. While we used ADsafe for its simplicity and suitability to meet our system needs, we expect that it would be similarly possible to develop xBook using these alternatives.

10 Conclusions

We presented a novel architecture for a social networking framework, called xBook, that substantially improves privacy control in the presence of untrusted third-party application. Our design allows the applications to have access to user data to preserve their functionality, but at the same time preventing them from leaking users' private information.

We developed a working prototype of the system that is available as an application on Facebook [33]. We showed the viability of our system by developing sample applications using the xBook APIs: these applications are similar in functionality to the applications on existing social networks.

Our system shows promise in designing potentially valuable future applications, that would require user data to provide more customized service to the user. The growing popularity of social networks would attract increasing attention from attackers because of the value of user information available in these networks. This user information not only has commercial value, but when combined with some anonymized public data such as medical records, might leak more sensitive information [28, 34]. The current design of social networking applications poses a serious threat to the privacy of individuals that needs to be mitigated; the xBook platform is a major step in protecting user privacy in social networking applications.

Acknowledgement

This material is based upon work supported in part by the NSF under grants no. 0716570 and 0831300 and the Department of Homeland Security under contract no. FA8750-08-2-0141. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or the Department of Homeland Security. We would also like to thank Monirul Sharif, Roberto Perdisci and the anonymous reviewers for their helpful comments and our shepherd George Danezis for his valuable suggestions.

References

[1] ADsafe. `http://adsafe.org`. Last accessed Feb. 1, 2009.

[2] Amazon elastic computing cloud. `http://aws.amazon.com/ec2/`. Last accessed Feb. 1, 2009.

[3] Daily horoscopes. `http://apps.facebook.com/daily-horoscope`. Last accessed Feb. 1, 2009.

[4] Facebook developers: Developer terms of service. `http://developers.facebook.com/terms.php`. Last accessed Feb. 1, 2009.

[5] Facebook's privacy policy. `http://www.facebook.com/policy.php`. Last accessed Feb. 1, 2009.

[6] Helma javascript web application framework. `http://www.helma.org`.

[7] Javascript object notation (JSON). `http://www.json.org`. Last accessed Feb. 1, 2009.

[8] JSLint: The javascript verifier. `http://www.jslint.com`. Last accessed Feb. 1, 2009.

[9] Map your friends. `http://apps.facebook.com/mapyourfriends`. Last accessed Feb. 1, 2009.

[10] Opensocial. `http://www.opensocial.org/`. Last accessed Feb. 1, 2009.

[11] Topfriends. `http://apps.facebook.com/topfriends`. Last accessed Feb. 1, 2009.

[12] A. Acharya and M. Raje. MAPbox: using parameterized behavior classes to confine untrusted applications. In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, Aug. 2000.

[13] L. Backstrom, C. Dwork, and J. Kleinberg. Wherefore art thou r3579x?: Anonymized social networks, hidden patterns, and structural steganography. In *Proceedings of the 16th International Conference on World Wide Web (WWW)*, Banff, Canada, May 2007.

[14] D. E. Bell and L. J. Lapadula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, MITRE Corp., Bedford, MA, Mar. 1976.

[15] A. Blum, C. Dwork, F. McSherry, and K. Nissim. Practical privacy: the SuLQ framework. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, Baltimore, MD, 2005.

[16] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, Oct. 2007.

[17] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[18] D. Farber. Google to open orkut opensocial developer sandbox tonight, Nov. 2007. `http://blogs.zdnet.com/BTL/?p=6856`. Last accessed Feb. 1, 2009.

[19] A. Felt and D. Evans. Privacy protection for social networking platforms. In *Web 2.0 Security and Privacy Workshop*, Oakland, CA, May 2008.

[20] M. Finifter, A. Mettler, N. Sastry, and D. Wagner. Verifiable functional purity in java. In *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*, Alexandria, VA, Oct. 2008.

[21] S. Hacking. More advertising issues on facebook (updated), 2008. http://theharmonyguy.com/2008/06/20/more-advertising-issues-on-facebook/. Last accessed Feb. 1, 2009.

[22] R. Konrad. Facebook opens to third-party developers, May 2007. http://www.msnbc.msn.com/id/18899269/. Last accessed Feb. 1, 2009.

[23] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, Oct. 2007.

[24] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. L-diversity: Privacy beyond k-anonymity. *ACM Transactions of Knowledge Discovery from Data*, 1(1):3, 2007.

[25] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: safe active content in sanitized javascript, Oct. 2007. http://google-caja.googlecode.com/files/caja-spec-2007-10-11.pdf.

[26] E. Mills. Facebook suspends app that permitted peephole, 2008. http://news.cnet.com/8301-10784_3-9977762-7.html. Last accessed Feb. 1, 2009.

[27] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP)*, Saint-Malo, France, Oct. 1997.

[28] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

[29] T. Panja. Oxford using Facebook to snoop. http://www.msnbc.msn.com/id/19813092/. Last accessed Feb. 1, 2009.

[30] D. S. Peterson, M. Bishop, and R. Pandey. A flexible containment mechanism for executing untrusted code. In *Proceedings of the 11th USENIX Security Symposium*, San Franscisco, CA, Aug. 2002.

[31] P. Samarati. Protecting respondents' identities in microdata release. *IEEE Transactions on Knowledge and Data Engineering*, 13(6):1010–1027, 2001.

[32] D. Sciba. Mayor in myspace photo flap asked to resign. http://www.katu.com/news/13670287.html. Last accessed Feb. 1, 2009.

[33] K. Singh, S. Bhola, and W. Lee. xBook on Facebook. http://apps.facebook.com/myxbook. Last accessed Feb. 1, 2009.

[34] L. Sweeney. Weaving technology and policy together to maintain confidentiality. *Journal of Law, Medicine and Ethics*, 25:98–110, 1997.

[35] C. Williams. Facebook application hawks your personal opinions for cash, Sept. 2007. http://www.theregister.co.uk/2007/09/12/facebook_compare_people/. Last accessed Feb. 1, 2009.

[36] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006.

Nemesis: Preventing Authentication & Access Control Vulnerabilities in Web Applications

Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich
Computer Systems Laboratory *CSAIL*
Stanford University *MIT*
{*mwdalton, kozyraki*}*@stanford.edu* *nickolai@csail.mit.edu*

Abstract

This paper presents *Nemesis*, a novel methodology for mitigating authentication bypass and access control vulnerabilities in existing web applications. *Authentication attacks* occur when a web application authenticates users unsafely, granting access to web clients that lack the appropriate credentials. *Access control attacks* occur when an access control check in the web application is incorrect or missing, allowing users unauthorized access to privileged resources such as databases and files. Such attacks are becoming increasingly common, and have occurred in many high-profile applications, such as IIS [10] and WordPress [31], as well as 14% of surveyed web sites [30]. Nevertheless, none of the currently available tools can fully mitigate these attacks.

Nemesis *automatically determines* when an application safely and correctly authenticates users, by using Dynamic Information Flow Tracking (DIFT) techniques to track the flow of user credentials through the application's language runtime. Nemesis combines authentication information with programmer-supplied access control rules on files and database entries to *automatically ensure* that only properly authenticated users are granted access to any privileged resources or data. A study of seven popular web applications demonstrates that a prototype of Nemesis is effective at mitigating attacks, requires little programmer effort, and imposes minimal runtime overhead. Finally, we show that Nemesis can also improve the precision of existing security tools, such as DIFT analyses for SQL injection prevention, by providing runtime information about user authentication.

1 Introduction

Web applications are becoming increasingly prevalent because they allow users to access their data from any computer and to interact and collaborate with each other. However, exposing these rich interfaces to anyone on the internet makes web applications an appealing target for attackers who want to gain access to other users' data or resources. Web applications typically address this problem through *access control*, which involves *authenticating* users that want to gain access to the system, and ensuring that a user is properly *authorized* to perform any operation the server executes on her behalf. In theory, this approach should ensure that unauthorized attackers cannot subvert the application.

Unfortunately, experience has shown that many web applications fail to follow these seemingly simple steps, with disastrous results. Each web application typically deploys its own authentication and access control framework. If any flaw exists in the authentication system, an authentication bypass attack may occur, allowing attackers to become authenticated as a valid user without having to present that user's credentials, such as a password. Similarly, a single missing or incomplete access control check can allow unauthorized users to access privileged resources. These attacks can result in the complete compromise of a web application.

Designing a secure authentication and access control system in a web application is difficult. Part of the reason is that the underlying file system and database layers perform operations with the privileges of the web application, rather than with privileges of a specific web application user. As a result, the web application must have the superset of privileges of all of its users. However, much like a Unix `setuid` application, it must explicitly check if the requesting user is authorized to perform each operation that the application performs on her behalf; otherwise, an attacker could exploit the web application's privileges to access unauthorized resources. This approach is ad-hoc and brittle, since these checks must be sprinkled throughout the application code whenever a resource is accessed, spanning code in multiple modules written by different developers over a long period of time. It is hard for developers to keep track of all the security policies that have to be checked. Worse yet, code written for other applications or third-party libraries with different security assumptions is often reused without considering the security implications. In each case, the result is that it's difficult to ensure the correct checks are always performed.

It is not surprising, then, that authentication and access control vulnerabilities are listed among the top ten vulnerabilities in 2007 [17], and have been discovered in high-profile applications such as IIS [10] and WordPress [31]. In 2008 alone, 168 authentication and access control vulnerabilities were reported [28]. A recent survey of real-world web sites found that over 14% of surveyed sites were vulnerable to an authentication or access control bypass attack [30].

Despite the severity of authentication or authorization bypass attacks, no defensive tools currently exist to automatically detect or prevent them. The difficulty in addressing these attacks stems from the fact that most web applications implement their own user authentication and authorization systems. Hence, it is hard for an automatic tool to ensure that the application properly authenticates all users and only performs operations for which users have the appropriate authorization.

This paper presents *Nemesis*,[1] a security methodology that addresses these problems by *automatically tracking* when user authentication is performed in web applications without relying on the safety or correctness of the existing code. Nemesis can then use this information to *automatically enforce* access control rules and ensure that only authorized web application users can access resources such as files or databases. We can also use the authentication information to improve the precision of other security analyses, such as DIFT-based SQL injection protection, to reduce their false positive rate.

To determine how a web application authenticates users, Nemesis uses Dynamic Information Flow Tracking (DIFT) to track the flow of user credentials, such as a username and password, through the application code. The key insight is that most applications share a similar high-level design, such as storing usernames and passwords in a database table. While the details of the authentication system, such as function names, password hashing algorithms, and session management vary widely, we can nonetheless determine when an application authenticates a user by keeping track of what happens to user credentials at runtime. Once Nemesis detects that a user has provided appropriate credentials, it creates an additional HTTP cookie to track subsequent requests issued by the authenticated user's browser. Our approach does not require the behavior of the application to be modified, and does not require any modifications to the application's existing authentication and access control system. Instead, Nemesis is designed to secure legacy applications without requiring them to be rewritten.

To prevent unauthorized access in web applications, Nemesis combines user authentication information with authorization policies provided by the application developer or administrator in the form of access control rules for various resources in the application, such as files, directories, and database entries. Nemesis then automatically ensures that these access control rules are enforced at runtime whenever the resource is accessed by an (authenticated) user. Our approach requires only a small amount of work from the programmer to specify these rules—in most applications, less than 100 lines of code. We expect that explicitly specifying access control rules per-resource is less error-prone than having to invoke the access control check each time the resource is accessed, and having to enumerate all possible avenues of attack. Furthermore, in applications that support third-party plugins, these access control rules need only be specified once, and they will automatically apply to code written by all plugin developers.

By allowing programmers to explicitly specify access control policies in their applications, and by tying the authentication information to runtime authorization checks, Nemesis prevents a wide range of authentication and access control vulnerabilities seen in today's applications. The specific contributions of this paper are as follows:

- We present Nemesis, a methodology for inferring authentication and enforcing access control in existing web applications, while requiring minimal annotations from the application developers.
- We demonstrate that Nemesis can be used to prevent authentication and access control vulnerabilities in modern web applications. Furthermore, we show that Nemesis can be used to prevent false positives and improve precision in real-world security tools, such as SQL injection prevention using DIFT.
- We implement a prototype of Nemesis by modifying the PHP interpreter. The prototype is used to collect performance measurements and to evaluate our security claims by preventing authentication and access control attacks on real-world PHP applications.

The remainder of the paper is organized as follows. Section 2 reviews the security architecture of modern web applications, and how it relates to common vulnerabilities and defense mechanisms. We describe our authentication inference algorithm in Section 3, and discuss our access control methodology in Section 4. Our PHP-based prototype is discussed in Section 5. Section 6 presents our experimental results, and Section 7 discusses future work. Finally, Section 8 discusses related work and Section 9 concludes the paper.

[1]Nemesis is the Greek goddess of divine indignation and retribution, who punishes excessive pride, evil deeds, undeserved happiness, and the absence of moderation.

2 Web Application Security Architecture

A key problem underlying many security vulnerabilities is that web application code executes with full privileges while handling requests on behalf of users that only have limited privileges, violating the principle of least privilege [11]. Figure 1 provides a simplified view of the security architecture of typical web applications today. As can be seen from the figure, the web application is performing file and database operations on behalf of users using its own credentials, and if attackers can trick the application into performing the wrong operation, they can subvert the application's security. Web application security can thus be viewed as an instance of the confused deputy problem [9]. The rest of this section discusses this architecture and its security ramifications in more detail.

2.1 Authentication Overview

When clients first connect to a typical web application, they supply an application-specific username and password. The web application then performs an authentication check, ensuring that the username and password are valid. Once a user's credentials have been validated, the web application creates a login session for the user. This allows the user to access the web application without having to log in each time a new page is accessed. Login sessions are created either by placing authentication information directly in a cookie that is returned to the user, or by storing authentication information in a session file stored on the server and returning a cookie to the user containing a random, unique session identifier. Thus, a user request is deemed to be authenticated if the request includes a cookie with valid authentication information or session identifier, or if it directly includes a valid username and password.

Once the application establishes a login session for a user, it allows the user to issue requests, such as posting comments on a blog, which might insert a row into a database table, or uploading a picture, which might require a file to be written on the server. However, there is a *semantic gap* between the user authentication mechanism implemented by the web application, and the access control or authorization mechanism implemented by the lower layers, such as a SQL database or the file system. The lower layers in the system usually have no notion of application-level users; instead, database and file operations are usually performed with the privileges and credentials of the web application itself.

Consider the example shown in Figure 1, where the web application writes the file uploaded by user Bob to the local file system and inserts a row into the database to keep track of the file. The file system is not aware of any authentication performed by the web application or web server, and treats all operations as coming from the web application itself (e.g. running as the Apache user in Unix). Since the web application has access to every user's file, it must perform internal checks to ensure that Bob hasn't tricked it into overwriting some other user's file, or otherwise performing an unauthorized operation. Likewise, database operations are performed using a per-web application database username and password provided by the system administrator, which authenticates the web application as user `webdb` to MySQL. Much like the filesystem layer, MySQL has no knowledge of any authentication performed by the web application, interpreting all actions sent by the web application as coming from the highly-privileged `webdb` user.

2.2 Authentication & Access Control Attacks

The fragile security architecture in today's web applications leads to two common problems, authentication bypass and access control check vulnerabilities.

Authentication bypass attacks occur when an attacker can fool the application into treating his or her requests as coming from an authenticated user, without having to present that user's credentials, such as a password. A typical example of an authentication bypass vulnerability involves storing authentication state in an HTTP cookie without performing any server-side validation to ensure that the client-supplied cookie is valid. For example, many vulnerable web applications store only the username in the client's cookie when creating a new login session. A malicious user can then edit this cookie to change the username to the administrator, obtaining full administrator access. Even this seemingly simple problem affects many applications, including PHP iCalendar [20] and phpFastNews [19], both of which are discussed in more detail in the evaluation section.

Access control check vulnerabilities occur when an access check is missing or incorrectly performed in the application code, allowing an attacker to execute server-side operations that she might not be otherwise authorized to perform. For example, a web application may be compromised by an invalid access control check if an administrative control panel script does not verify that the web client is authenticated as the admin user. A malicious user can then use this script to reset other passwords, or even perform arbitrary SQL queries, depending on the contents of the script. These problems have been found in numerous applications, such as PhpStat [21].

Authentication and access control attacks often result in the same unfettered file and database access as traditional input validation vulnerabilities such as SQL injection and directory traversal. However, authentication and access control bugs are more difficult to detect, because their

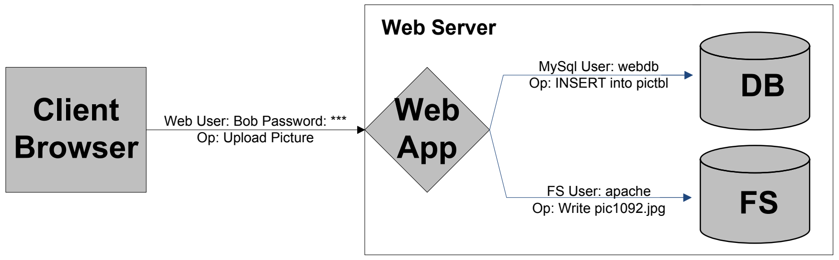


Figure 1: The security architecture of typical web applications. Here, user Bob uploads a picture to a web application, which in turn inserts data into a database and creates a file. The user annotation above each arrow indicates the credentials or privileges used to issue each operation or request.

logic is application-specific, and they do not follow simple patterns that can be detected by simple analysis tools.

2.3 Other Web Application Attacks

Authentication and access control also play an important, but less direct role, in SQL injection [27], command injection, and directory traversal attacks. For example, the PHP code in Figure 2 places user-supplied search parameters into a SQL query without performing any sanitization checks. This can result in a SQL injection vulnerability; a malicious user could exploit it to execute arbitrary SQL statements on the database. The general approach to addressing these attacks is to validate all user input before it is used in any filesystem or database operations, and to disallow users from directly supplying SQL statements. These checks occur throughout the application, and any missing check can lead to a SQL injection or directory traversal vulnerability.

However, these kinds of attacks are effective only because the filesystem and database layers perform all operations with the privilege level of the web application rather than the current authenticated webapp user. If the filesystem and database access of a webapp user were restricted only to the resources that the user should legitimately access, input validation attacks would not be effective as malicious users would not be able not leverage these attacks to access unauthorized resources.

Furthermore, privileged users such as site administrators are often allowed to perform operations that could be interpreted as SQL injection, command injection, or directory traversal attacks. For example, popular PHP web applications such as DeluxeBB and phpMyAdmin allow administrators to execute arbitrary SQL commands. Alternatively, code in Figure 2 could be safe, as long as only administrative users are allowed to issue such search queries. This is the very definition of a SQL injection attack. However, these SQL injection vulnerabilities can only be exploited if the application fails to check that the user is authenticated as the administrator before issuing the SQL query. Thus, to properly judge whether a SQL injection attack is occurring, the security system must know which user is currently authenticated.

3 Authentication Inference

Web applications often have buggy implementations of authentication and access control, and no two applications have the exact same authentication framework. Rather than try to mandate the use of any particular authentication system, Nemesis prevents authentication and access control vulnerabilities by automatically inferring when a user has been safely authenticated, and then using this authentication information to automatically enforce access control rules on web application users. An overview of Nemesis and how it integrates into a web application software stack is presented in Figure 3. In this section, we describe how Nemesis performs *authentication inference*.

3.1 Shadow Authentication Overview

To prevent authentication bypass attacks, Nemesis must infer when authentication has occurred without depending on the correctness of the application authentication system., which are often buggy or vulnerable. To this end, Nemesis constructs a *shadow authentication system* that works alongside the application's existing authentication framework. In order to infer when user authentication has safely and correctly occurred, Nemesis requires the application developer to provide one annotation—namely, where the application stores user names and their known-good passwords (e.g. in a database table), or what external function it invokes to authenticate users (e.g. using LDAP or OpenID). Aside from this annotation, Nemesis is agnostic to the specific hash function or algorithm used to validate user-supplied credentials.

To determine when a user successfully authenticates, Nemesis uses Dynamic Information Flow Tracking (DIFT). In particular, Nemesis keeps track of two bits

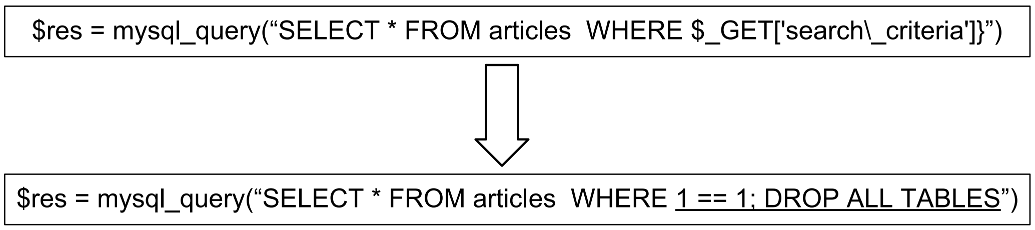

$res = mysql_query("SELECT * FROM articles WHERE $_GET['search_criteria']}")
$res = mysql_query("SELECT * FROM articles WHERE 1 == 1; DROP ALL TABLES")

Figure 2: Sample PHP code vulnerable to SQL injection, and the resulting query when a user supplies the underlined, malicious input.

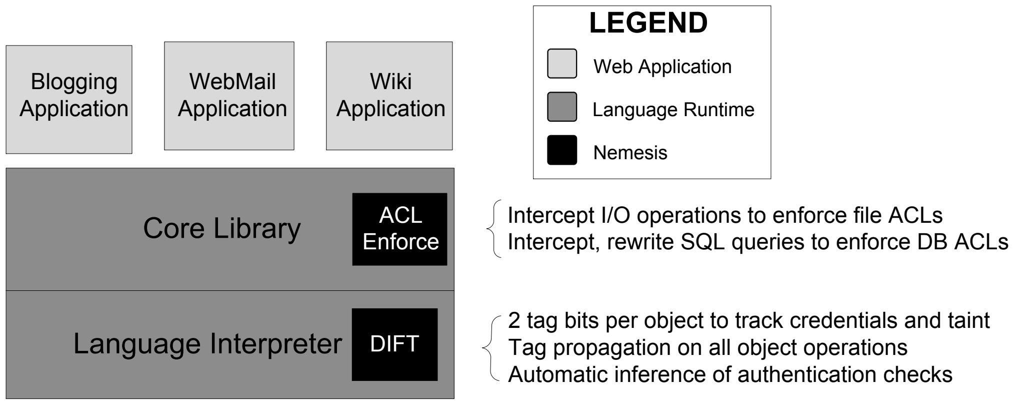

LEGEND
Web Application
Language Runtime
Nemesis
Blogging Application
WebMail Application
Wiki Application
Core Library
ACL Enforce
Intercept I/O operations to enforce file ACLs
Intercept, rewrite SQL queries to enforce DB ACLs
Language Interpreter
DIFT
2 tag bits per object to track credentials and taint
Tag propagation on all object operations
Automatic inference of authentication checks

Figure 3: Overview of Nemesis system architecture

of *taint* for each data item in the application—a "credential" taint bit, indicating whether the data item represents a known-good password or other credential, and a "user input" taint bit, indicating whether the data item was supplied by the user as part of the HTTP request. User input includes all values supplied by the untrusted client, such as HTTP request headers, cookies, POST bodies, and URL parameters. Taint bits can be stored either per object (e.g., string), or per byte (e.g., string characters), depending on the needed level of precision and performance.

Nemesis must also track the flow of authentication credentials and user input during runtime code execution. Much like other DIFT systems [8, 15, 16], this is done by performing taint propagation in the language interpreter. Nemesis propagates both taint bits at runtime for all data operations, such as variable assignment, load, store, arithmetic, and string concatenation. The propagation rule we enforce is *union*: a destination operand's taint bit is set if it was set in any of the source operands. Since Nemesis is concerned with inferring authentication rather than addressing covert channels, implicit taint propagation across control flow is not considered. The rest of this section describes how Nemesis uses these two taint bits to infer when successful authentication has taken place.

3.2 Creating a New Login Session

Web applications commonly authenticate a new user session by retrieving a username and password from a storage location (typically a database) and comparing these credentials to user input. Other applications may use a dedicated login server such as LDAP or Kerberos, and instead defer all authentication to the login server by invoking a special third-party library authentication function. We must infer and detect both of these authentication types.

As mentioned, Nemesis requires the programmer to specify where the application stores user credentials for authentication. Typical applications store password hashes in a database table, in which case the programmer should specify the name of this table and the column names containing the user names and passwords. For applications that defer authentication to an external login server, the programmer must provide Nemesis with the name of the authentication function (such as ldap_login), as well as which function arguments represent the username and password, and what value the function returns upon authentication success. In either case, the shadow authentication system uses this information to determine when the web application has safely authenticated a user.

3.2.1 Direct Password Authentication

When an application performs authentication via direct password comparisons, the application must read the username and password from an authentication storage location, and compare them to the user-supplied authentication credentials. Whenever the authentication storage location is read, our shadow authentication system records the username read as the current user under authentication, and sets the "credential" taint bit for the password string. In most web applications, a client can only authenticate as a single user at any given time. If an application allows clients to authenticate as multiple users at the same time, Nemesis would have to be extended to keep track of multiple candidate usernames, as well as multiple "credential" taint bits on all data items. However, we are not aware of a situation in which this occurs in practice.

When data tagged as "user input" is compared to data tagged as "credentials" using string equality or inequality operators, we assume that the application is checking whether a user-supplied password matches the one stored in the local password database. If the two strings are found to be equal, Nemesis records the web client as authenticated for the candidate username. We believe this is an accurate heuristic, because known-good credentials are the only objects in the system with the "credential" taint bit set, and only user input has the "user input" taint bit set. This technique even works when usernames and passwords are supplied via URL parameters (such as "magic URLs" which perform automatic logins in HotCRP) because all values supplied by clients, including URL parameters, are tagged as user input.

Tag bits are propagated across all common operations, allowing Nemesis to support standard password techniques such as cryptographic hashes and salting. Hashing is supported because cryptographic hash functions consist of operations such as array access and arithmetic computations, all of which propagate tag bits from inputs to outputs. Similarly, salting is supported because prepending a salt to a user-supplied password is done via string concatenation, an operation that propagates tag bits from source operands to the destination operand.

This approach allows us to infer user authentication by detecting when a user input string is compared and found to be equal to a password. This avoids any internal knowledge of the application, requiring only that the system administrator correctly specify the storage location of usernames and passwords. A web client will only be authenticated by our shadow authentication system if they know the password, because authentication occurs only when a user-supplied value is equal to a known password. Thus, our approach does not suffer from authentication vulnerabilities, such as allowing a user to log in if a magic URL parameter or cookie value is set.

3.2.2 Deferred Authentication to a Login Server

We use similar logic to detect authentication when using a login server. The web client is assumed to be authenticated if the third-party authentication function is called with a username and password marked as "user input", and the function returns success. In this case, Nemesis sets the authenticated user to the username passed to this function. Nemesis checks to see if the username and password passed to this function are tainted in order to distinguish between credentials supplied by the web client and credentials supplied internally by the application. For example, phpMyAdmin uses MySQL's built-in authentication code to both authenticate web clients, and to authenticate itself to the database for internal database queries [23]. Credentials used internally by the application should not be treated as the client's credentials, and Nemesis ensures this by only accepting credentials that came from the web client. Applications that use single sign-on systems such as OpenID must use deferred authentication, as the third-party authentication server (e.g., OpenID Provider) performs the actual user authentication.

3.3 Resuming a Previous Login Session

As described in Section 2.1, web applications create login sessions by recording pertinent authentication information in cookies. This allows users to authenticate once, and then access the web application without having to authenticate each time a new page is loaded. Applications often write their own custom session management frameworks, and session management code is responsible for many authentication bypass vulnerabilities.

Fortunately, Nemesis does not require any per-application customization for session management. Instead, we use an entirely separate session management framework. When Nemesis infers that user authentication has occurred (as described earlier in this section), a new cookie is created to record the shadow authentication credentials of the current web client. We do not interpret or attempt to validate any other cookies stored and used by the web application for session management. For all intents and purposes, session management in the web application and Nemesis are orthogonal. We refer to the cookie used for Nemesis session management as the *shadow cookie*. When Nemesis is presented with a valid shadow cookie, the current shadow authenticated user is set to the username specified in the cookie.

Shadow authentication cookies contain the shadow authenticated username of the current web user and an HMAC of the username computed using a private key kept on the server. The user cannot edit or change their shadow authentication cookie because the username HMAC will no longer match the username itself, and the user does not have the key used to compute the HMAC. This cookie is returned to the user, and stored along with any other authentication cookies created by the web application.

Our shadow authentication system detects a user safely resuming a prior login session if a valid shadow cookie is presented. The shadow authentication cookie is verified by recomputing the cookie HMAC based on the username from the cookie. If the recomputed HMAC and the HMAC from the cookie are identical, the user is successfully authenticated by our shadow authentication system. Nemesis distinguishes between shadow cookies from multiple applications running on the same server by using a different HMAC key for each application, and including a hash derived from the application's HMAC key in the name of the cookie.

In practice, when a user resumes a login session, the web application will validate the user's cookies and session file, and then authorize the user to access a privileged resource. When the privileged resource access is attempted, Nemesis will examine the user's shadow authentication credentials and search for valid shadow cookies. If a valid shadow cookie is found and verified to be safe, the user's shadow authentication credentials are updated. Nemesis then performs an access control check on the shadow authentication credentials using the web application ACL.

3.4 Registering a New User

The last way a user may authenticate is to register as a new user. Nemesis infers that new user registration has occurred when a user is inserted into the authentication credential storage location. In practice, this is usually a SQL INSERT statement modifying the user authentication database table. The inserted username must be tainted as "user input", to ensure that this new user addition is occurring on behalf of the web client, and not because the web application needed to add a user for internal usage.

Once the username has been extracted and verified as tainted, the web client is then treated as authenticated for that username, and the appropriate session files and shadow authentication cookies are created. For the common case of a database table, this requires us to parse the SQL query, and determine if the query is an INSERT into the user table or not. If so, we extract the username field from the SQL statement.

3.5 Authentication Bypass Attacks

Shadow authentication information is only updated when the web client supplies valid user credentials, such as a password for a web application user, or when a valid shadow cookie is presented. During authentication bypass attacks, malicious users are authenticated by the web application without supplying valid credentials. Thus, when one of these attacks occurs, the web application will incorrectly authenticate the malicious web client, but shadow authentication information will not be updated.

While we could detect authentication bypass attacks by trying to discern when shadow authentication information differs from the authenticated state in the web application, this would depend on internal knowledge of each web application's code base. Authentication frameworks are often complex, and each web application typically creates its own framework, possibly spreading the current authentication information among multiple variables and complex data structures.

Instead, we note that the goal of any authentication bypass attack is to use the ill-gotten authentication to obtain unauthorized access to resources. These are exactly the resources that the current shadow authenticated user is not permitted to access. As explained in the next section, we can prevent authentication bypass attacks by detecting when the current shadow authenticated user tries to obtain unauthorized access to a system resource such as a file, directory, or database table.

4 Authorization Enforcement

Both authentication and access control bypass vulnerabilities allow an attacker to perform operations that she would not be otherwise authorized to perform. The previous section described how Nemesis constructs a shadow authentication system to keep track of user authentication information despite application-level bugs. However, the shadow authentication system alone is not enough to prevent these attacks. This section describes how Nemesis mitigates the attacks by connecting its shadow authentication system with an access control system protecting the web application's database and file system.

To control what operations any given web user is allowed to perform, Nemesis allows the application developer to supply access control rules (ACL) for files, directories, and database objects. Nemesis extends the core system library so that each database or file operation performs an ACL check. The ACL check ensures that the current shadow authenticated user is permitted by the web application ACL to execute the operation. This enforcement prevents access control bypass attacks, because an attacker exploiting a missing or invalid access control check to perform a privileged operation will be foiled when Nemesis enforces the supplied ACL. This also mitigates authentication bypass attacks—even if an attacker can bypass the application's authentication system (e.g., due to a missing check in the application code), Nemesis will automatically perform ACL checks against the username provided by the shadow authentication system, which is not subject to authentication bypass attacks.

4.1 Access Control

In any web application, the authentication framework plays a critical role in access control decisions. There are often numerous, complex rules determining which resources (such as files, directories, or database tables, rows, or fields) can be accessed by a particular user. However, existing web applications do not have explicit, codified access control rules. Rather, each application has its own authentication system, and access control checks are interspersed throughout the application.

For example, many web applications have a privileged script used to manage the users of the web application. This script must only be accessed by the web application administrator, as it will likely contain logic to change the password of an arbitrary user and perform other privileged operations. To restrict access appropriately, the beginning of the script will contain an access control check to ensure that unauthorized users cannot access script functionality. This is actually an example of the policy, "only the administrator may access the admin.php script", or to rephrase such a policy in terms of the resources it affects, "only the administrator may modify the user table in the database". This policy is often never explicitly stated within the web application, and must instead be inferred from the authorization checks in the web application. Nemesis requires the developer or system administrator to explicitly provide an access control list based on knowledge of the application. Our prototype system and evaluation suggests that, in practice, this requires little programmer effort while providing significant security benefits. Note that a single developer or administrator needs to specify access control rules. Based on these rules, Nemesis will provide security checks for all application users.

4.1.1 File Access

Nemesis allows developers to restrict file or directory access to a particular shadow authenticated user. For example, a news application may only allow the administrator to update the news spool file. We can also restrict the set of valid operations that can be performed: read, write, or append. For directories, read permission is equivalent to listing the contents of the directory, while write permission allows files and subdirectories to be created. File access checks happen before any attempt to open a file or directory. These ACLs could be expressed by listing the files and access modes permitted for each user.

4.1.2 SQL Database Access

Nemesis allows web applications to restrict access to SQL tables. Access control rules specify the user, name of the SQL database table, and the type of access (INSERT, SELECT, DELETE, or UPDATE). For each SQL query, Nemesis must determine what tables will be accessed by the query, and whether the ACLs permit the user to perform the desired operation on those tables.

In addition to table-level access control, Nemesis also allows restricting access to individual rows in a SQL table, since applications often store data belonging to different users in the same table.

An ACL for a SQL row works by restricting a given SQL table to just those rows that should be accessible to the current user, much like a *view* in SQL terminology. Specifically, the ACL maps SQL table names and access types to an SQL predicate expression involving column names and values that constrain the kinds of rows the current user can access, where the values can be either fixed constants, or the current username from the shadow authentication system, evaluated at runtime. For example, a programmer can ensure that a user can only access their own profile by confining SQL queries on the profile table to those whose *user* column matches the current shadow username.

SELECT ACLs restrict the values returned by a SELECT SQL statement. DELETE and UPDATE query ACLs restrict the values modified by an UPDATE or DELETE statement, respectively. To enforce ACLs for these statements, Nemesis must rewrite the database query to append the field names and values from the ACL to the WHERE condition clause of the query. For example, a query to retrieve a user's private messages might be "SELECT * FROM messages WHERE recipient=$current_user", where $current_user is supplied by the application's authentication system. If attackers could fool the application's authentication system into setting $current_user to the name of a different user, they might be able to retrieve that user's messages.

Using Nemesis, the programmer can specify an ACL that only allows SELECTing rows whose sender or recipient column matches the current shadow user. As a result, if user Bob issues the query, Nemesis will transform it into the query "SELECT * FROM messages WHERE recipient=$current_user AND (sender=Bob or recipient=Bob)", which mitigates any possible authentication bypass attack.

Finally, INSERT statements do not read or modify existing rows in the database. Thus, access control for INSERT statements is governed solely by the table access control rules described earlier. However, sometimes developers may want to set a particular field to the current shadow authenticated user when a row is inserted into a table. Nemesis accomplishes this by rewriting the INSERT query to replace the value of the designated field with the current shadow authenticated user (or to add an additional field assignment if the designated field was not initialized by the INSERT statement).

Modifying INSERT queries has a number of real-world uses. Many database tables include a field that stores the username of the user who inserted the field. The administrator can choose to replace the value of this field with the shadow authenticated username, so that authentication flaws do not allow users to spoof the owner of a particular row in the database. For example, in the PHP forum application DeluxeBB, we can override the author name field in the table of database posts with the shadow authenticated username. This prevents malicious clients from spoofing the author when posting messages, which can occur if an authentication flaw allows attackers to authenticate as arbitrary users.

4.2 Enhancing Access Control with DIFT

Web applications often perform actions which are not authorized for the currently authenticated user. For example, in the PHP image gallery Linpha, users may inform the web application that they have lost their password. At this point, the web client is unauthenticated (as they have no valid password), but the web application changes the user's password to a random value, and e-mails the new password to the user's e-mail account. While one user should not generally be allowed to change the password of a different user, doing so is safe in this case because the application generates a fresh password not known to the requesting user, and only sends it via email to the owner's address.

One heuristic that helps us distinguish these two cases in practice is the taint status of the newly-supplied password. Clearly it would be a bad idea to allow an unauthenticated user to supply the new password for a different user's account, and such password values would have the "user input" taint under Nemesis. At the same time, our experience suggests that internally-generated passwords, which do not have the "user input" taint, correspond to password reset operations, and would be safe to allow.

To support this heuristic, we add one final parameter to all of the above access control rules: taint status. An ACL entry may specify, in addition to its other parameters, taint restrictions for the file contents or database query. For example, an ACL for Linpha allows the application to update the password field of the user table regardless of the authentication status, as long as the query is untainted. If the query is tainted, however, the ACL only allows updates to the row corresponding to the currently authenticated user.

4.3 Protecting Authentication Credentials

Additionally, there is one security rule that does not easily fit into our access control model, yet can be protected via DIFT. When a web client is authenticating to the web application, the application must read user credentials such as a password and use those credentials to authenticate the client. However, unauthenticated clients do not have permission to see passwords. A safe web application will ensure that these values are never leaked to the client. To prevent an information leak bug in the web application from resulting in password disclosure, Nemesis forbids any object that has the authentication credential DIFT tag bit set from being returned in any HTTP response. In our prototype, this rule has resulted in no false positives in practice. Nevertheless, we can easily modify this rule to allow passwords for a particular user to be returned in a HTTP response once the client is authenticated for that user. For example, this situation could arise if a secure e-mail service used the user's password to decrypt e-mails, causing any displayed emails to be tagged with the password bit.

5 Prototype Implementation

We have implemented a proof-of-concept prototype of Nemesis by modifying the PHP interpreter. PHP is one of the most popular languages for web application development. However, the overall approach is not tied to PHP by design, and could be implemented for any other popular web application programming language. Our prototype is based on an existing DIFT PHP tainting project [29]. We extend this work to support authentication inference and authorization enforcement.

5.1 Tag Management

PHP is a dynamically typed language. Internally, all values in PHP are instances of a single type of structure known as a *zval*, which is stored as a tagged union. Integers, booleans, and strings are all instances of the *zval* struct. Aggregate data types such as arrays serve as hash tables mapping index values to *zval*s. Symbol tables are hash tables mapping variable names to *zval*s.

Our prototype stores taint information at the granularity of a *zval* object, which can be implemented without storage overhead in the PHP interpreter. Due to alignment restrictions enforced by GCC, the *zval* structure has a few unused bits, which is sufficient for us to store the two taint bits required by Nemesis.

By keeping track of taint at the object level, Nemesis assumes that the application will not combine different kinds of tagged credentials in the same object (e.g. by concatenating passwords from two different users together, or combining untrusted and authentication-based input into a single string). While we have found this assumption to hold in all encountered applications, a byte granularity tainting approach could be used to avoid this limitation if needed, and prior work has shown it practical to implement byte-level tainting in PHP [16]. When multiple objects are combined in our prototype, the result's taint bits are the union of the taint bits on all inputs. This works well for combining tainted and untainted data, such as concatenating an untainted salt with a tainted password (with the result being tainted), but can produce imprecise results when concatenating objects with two different classes of taint.

User input and password taint is propagated across all standard PHP operations, such as variable assignment, arithmetic, and string concatenation. Any value with password taint is forbidden from being returned to the user via echo, printf, or other output statements.

5.2 Tag Initialization

Any input from URL parameters (GET, PUT, etc), as well as from any cookies, is automatically tainted with the 'user input' taint bit. Currently, password taint initialization is done by manually inserting the taint initialization function call as soon as the password enters the system (e.g., from a database) as we have not yet implemented a full policy language for automated credential tainting. For a few of our experiments in Section 6 (phpFastNews, PHP iCalendar, Bilboblog), the admin password was stored in a configuration php script that was included by the application scripts at runtime. In this case, we instrumented the configuration script to set the password bit of the admin password variable in the script.

At the same time as we initialize the password taint, we also set a global variable to store the candidate username associated with the password, to keep track of the current username being authenticated. If authentication succeeds, the shadow authentication system uses this candidate username to set the global variable that stores the shadow authenticated user, as well as to initialize the shadow cookie. If a client starts authenticating a second time as a different user, the candidate username is reset to the new value, but the authenticated username is not affected until authentication succeeds.

Additionally, due to an implementation artifact in the PHP $setcookie()$ function, we also record shadow authentication in the PHP built-in session when appropriate. This is because PHP forbids new cookies to be added to the HTTP response once the application has placed part of the HTML body in the response output buffer. In an application that uses PHP sessions, the cookie only stores the session ID and all authentication information is stored in session files on the server. These applications may output part of the HTML body before authentication is complete. We correctly handle this case by storing shadow authentication credentials in the server session file if the application has begun a PHP session. When validating and recovering shadow cookies for authentication purposes, we also check the session file associated with the current user for shadow authentication credentials. This approach relies on PHP safely storing session files, but as session files are stored on the server in a temporary directory, this is a reasonable assumption.

5.3 Authentication Checks

When checking the authentication status of a user, we first check the global variable that indicates the current shadow authenticated user. This variable is set if the user has just begun a new session and been directly authenticated via password comparison or deferred authentication to a login server. If this variable is not set, we check to see if shadow authentication information is stored in the current session file (if any). Finally, we check to see if the user has presented a shadow authentication cookie, and if so we validate the cookie and extract the authentication credentials. If none of these checks succeeds, the user is treated as unauthenticated.

5.4 Password Comparison Authentication Inference

Authentication inference for password comparisons is performed by modifying the PHP interpreter's string comparison equality and inequality operators. When one of these string comparisons executes, we perform a check to see if the two string operands were determined to be equal. If the strings were equal, we then check their tags, and if one string has only the authentication credential tag bit set, and the other string has only the user input tag bit set, then we determine that a successful shadow authentication has occurred. In all of our experiments, only PhpMyAdmin used a form of authentication that did not rely on password string comparison, and our handling of this case is discussed in Section 6.

5.5 Access Control Checks

We perform access control checks for files by checking the current authenticated user against a list of accessible files (and file modes) on each file access. Similarly, we restrict SQL queries by checking if the currently authenticated user is authorized to access the table, and by appending additional WHERE clause predicates to scope the effect of the query to rows allowed for the current user.

Due to time constraints, we manually inserted these checks into applications based on the ACL needed by the application. ACLs that placed constraints on field values of a database row required simple query modifications to test if the field value met the constraints in the ACL.

In a full-fledged design, the SQL queries should be parsed, analyzed for the appropriate information, and rewritten if needed to enforce additional security guarantees (e.g., restrict rows modified to be only those created by the current authenticated user). Depending on the database used, query rewriting may also be partially or totally implemented using database views and triggers [18, 26].

5.6 SQL Injection

Shadow authentication is necessary to prevent authentication bypass attacks and enforce our ACL rules. However, it can also be used to prevent false positives in DIFT SQL injection protection analyses. The most robust form of SQL injection protection [27] forbids tainted keywords or operators, and enforces the rule that tainted data may never change the parse tree of a query.

Our current approach does not support byte granularity taint, and thus we must approximate this analysis. We introduce a third taint bit in the *zval* which we use to denote user input that may be interpreted as a SQL keyword or operator. We scan all user input at startup (GET, POST, COOKIE superglobal arrays, etc) and set this bit only for those user input values that contain a SQL keyword or operator. SQL quoting functions, such as `mysql_real_escape_string()`, clear this tag bit. Any attempt to execute a SQL query with the unsafe SQL tag bit set is reported as a SQL injection attack.

We use this SQL injection policy to confirm that DIFT SQL Injection has false positives and real-world web applications. This is because DIFT treats all user input as untrusted, but some web applications allow privileged users such as the admin to submit full SQL queries. As discussed in Section 6, we eliminate all encountered false positives using authentication policies which restrict SQL injection protection to users that are not shadow authenticated as the admin user. We have confirmed that all of these false positives are due to a lack of authentication information, and not due to any approximations made in our SQL injection protection implementation.

6 Experimental Results

To validate Nemesis, we used our prototype to protect a wide range of vulnerable real-world PHP applications from authentication and access control bypass attacks. A summary of the applications and their vulnerabilities is given in Table 1, along with the lines of code that were added or modified in order to protect them.

For each application, we had to specify where the application stores its username and password database, or what function it invokes to authenticate users. This step is quite simple for all applications, and the "authentication inference" column indicates the amount of code we had to add to each application to specify the table used to store known-good passwords, and to taint the passwords with the "credential" taint bit.

We also specified ACLs on files and database tables to protect them from unauthorized accesses; the number of access control rules for each application is shown in the table. As explained in Section 5, we currently enforce ACLs via explicitly inserted checks, which slightly increases the lines of code needed to implement the check (shown in the table as well). As we develop a full MySQL parser and query rewriter, we expect the lines of code needed for these checks to drop further.

We validated our rules by using each web application extensively to ensure there are no false positives, and then verifying that our rules prevented real-world attacks that have been found in these applications in the past. We also verified that our shadow authentication information is able to prevent false positives in DIFT SQL injection analyses for both the DeluxeBB and phpMyAdmin applications.

6.1 PHP iCalendar

PHP iCalendar is a PHP web application for presenting calendar information to users. The webapp administrator is authenticated using a configuration file that stores the admin username and password. Our ACL for PHP iCalendar allows users read access to various template files, language files, and all of the calendars. In addition, caches containing parsed calendars can be read or written by any user. The admin user is able to write, create, and delete calendar files, as well as read any uploaded calendars from the uploads directory. We added 8 authorization checks to enforce our ACL for PHP iCalendar.

An authentication bypass vulnerability occurs in PHP iCalendar because a script in the admin subdirectory incorrectly validates a login cookie when resuming a session [20]. This vulnerability allows a malicious user to forge a cookie that will cause her to be authenticated as the admin user.

Using Nemesis, when an attacker attempts to exploit the authentication bypass attack, she will find that her shadow authentication username is not affected by the attack. This is because shadow authentication uses its own secure form of cookie authentication, and stores its credentials separately from the rest of the web application. When the attacker attempts to use the admin scripts to perform any actions that require admin access, such as deleting a calendar, a security violation is reported because the shadow authentication username will not be 'admin', and the ACL will prevent that username from performing administrative operations.

6.2 Phpstat

Phpstat is an application for presenting a database of IM statistics to users, such as summaries and logs of their IM

Program	LoC in program	LoC for auth inference	Number of ACL checks	LoC for ACL checks	Vulnerability prevented
PHP iCalendar	13500	3	8	22	Authentication bypass
phpStat (IM Stats)	12700	3	10	17	Missing access check
Bilboblog	2000	3	4	11	Invalid access check
phpFastNews	500	5	2	17	Authentication bypass
Linpha Image Gallery	50000	15	17	49	Authentication bypass
DeluxeBB Web Forum	22000	6	82	143	Missing access check

Table 1: Applications used to evaluate Nemesis.

conversations. Phpstat stores its authentication credentials in a database table.

The access control list for PhpStat allows users to read and write various cache files, as well as read the statistics database tables. Users may also read profile information about any other user, but the value of the password field may never be sent back to the Web client. The administrative user is also allowed to create users by inserting into or updating the users table, as well as all of the various statistics tables. We added 10 authorization checks to enforce our ACL for PhpStat.

A security vulnerability exists in PhpStat because an installation script will reset the administrator password if a particular URL parameter is given. This behavior occurs without any access control checks, allowing any user to reset the admin password to a user-specified value [21]. Successful exploitation grants the attacker full administrative privileges to the Phpstat. Using Nemesis, when this attack occurs, the attacker will not be shadow authenticated as the admin, and any attempts to execute a SQL query that changes the password of the administrator are denied by our ACL rules. Only users shadow authenticated as the admin may change passwords.

6.3 Bilboblog

Bilboblog is a simple PHP blogging application that authenticates its administrator using a username and password from a configuration file.

Our ACL for bilboblog permits all users to read and write blog caching directories, and read any posted articles from the article database table. Only the administrator is allowed to modify or insert new entries into the articles database table. Bilboblog has an invalid access control check vulnerability because one of its scripts, if directly accessed, uses uninitialized variables to authenticate the admin user [3]. We added 4 access control checks to enforce our ACL for bilboblog.

In PHP, if the register_globals option is set, uninitialized variables may be initialized at startup by user-supplied URL parameters [22]. This allows a malicious user to supply the administrator username and password that the login will be authenticated against. The attacker may simply choose a username and password, access the login script with these credentials encoded as URL parameters, and then input the same username and password when prompted by the script. This attack grants the attacker full administrative access to Bilboblog.

This kind of attack does not affect shadow authentication. A user is shadow authenticated only if their input is compared against a valid password. This attack instead compares user input against a URL parameter. URL parameters do not have the password bit set – only passwords read from the configuration do. Thus, no shadow authentication occurs when this attack succeeds. If an attacker exploits this vulnerability on a system protected by our prototype, she will find herself unable to perform any privileged actions as the admin user. Any attempt to update, delete, or modify an article will be prevented by our prototype, as the current user will not be shadow authenticated as the administrator.

6.4 phpFastNews

PhpFastNews is a PHP application for displaying news stories. It performs authentication via a configuration file with username and password information. This web application displays a news spool to users. Our ACL for phpFastNews allows users to read the news spool, and restricts write access to the administrator. We added 2 access control checks to enforce our ACL for phpFastNews.

An authentication bypass vulnerability occurs in phpFastNews due to insecure cookie validation [19], much like in PHP iCalendar. If a particular cookie value is set, the user is automatically authenticated as the administrator without supplying the administrator's password. All the attacker must do is forge the appropriate cookie, and full admin access is granted.

Using Nemesis, when the authentication bypass attack occurs, our prototype will prevent any attempt to perform administrator-restricted actions such as updating the news spool because the current user is not shadow authenticated as the admin.

6.5 Linpha

Linpha is a PHP web gallery application, used to display directories of images to web users. It authenticates its users via a database table.

Our ACL for Linpha allows users to read files from the images directory, read and write files in the temporary and cache directories, and insert entries into the thumbnails table. Users may also read from the various settings, group, and configuration tables. The administrator may update or insert into the users table, as well as the settings, groups, and categories tables. Dealing with access by non-admin users to the user table is the most complex part of the Linpha ACL, and is our first example of a database row ACL. Any user may read from the user table, with the usual restriction that passwords may never be output to the Web client via echo, print, or related commands.

Users may also update entries in the user table. Updating the password field must be restricted so that a malicious user cannot update the other passwords. This safety restriction can be enforced by ensuring that only user table rows that have a username field equal to the current shadow authenticated user can be modified. The exception to this rule is when the new password is untainted. This can occur only when the web application has internally generated the new user password without using user input. We allow these queries even when they affect the password of a user that is not the current shadow authenticated user because they are used for lost password recovery.

In Linpha, users may lose their password, in which case Linpha resets their password to an internally generated value, and e-mails this password to the user. This causes an arbitrary user's password to be changed on the behalf of a user who isn't even authenticated. However, we can distinguish this safe and reasonable behavior from an attack by a user attempting to change another user's password by examining the taint of the new password value in the SQL query. Thus, we allow non-admin users to update the password field of the user table if the password query is untainted, or if the username of the modified row is equal to the current shadow authenticated user. Overall, we added 17 authorization checks to enforce all of our ACLs for Linpha.

Linpha also has an authentication bypass vulnerability because one of its scripts has a SQL injection vulnerability in the SQL query used to validate login information from user cookies [13]. Successful exploitation of this vulnerability grants the attacker full administrative access to Linpha. For this experiment, we disabled SQL injection protection provided by the tainting framework we used to implement the Nemesis prototype [29], to allow the user to submit a malicious SQL query in order to bypass authentication entirely.

Using Nemesis, once a user has exploited this authentication bypass vulnerability, she may access the various administration scripts. However, any attempt to actually use these scripts to perform activities that are reserved for the admin user will fail, because the current shadow authenticated user will not be set to admin, and our ACLs will correspondingly deny any admin-restricted actions.

6.6 DeluxeBB

DeluxeBB is a PHP web forum application that supports a wide range of features, such as an administration console, multiple forums, and private message communication between forum users. Authentication is performed using a table from a MySQL database.

DeluxeBB has the most intricate ACL of any application in our experiments. All users in DeluxeBB may read and write files in the attachment directory, and the admin user may also write to system log files. Non-admin users in DeluxeBB may read the various configuration and settings tables. Admin users can also write these tables, as well as perform unrestricted modifications to the user table. DeluxeBB treats user table updates and lost password modifications in the same manner as Linpha, and we use the equivalent ACL to protect the user table from non-admin modifications and updates.

DeluxeBB allows unauthenticated users to register via a form, and thus unauthenticated users are allowed to perform inserts into the user table. As described in Section 3.4, inserting a user into the user table results in shadow authentication with the credentials of the inserted user.

The novel and interesting part of the ACLs for DeluxeBB are the treatment of posts, thread creation, and private messages. All inserts into the post, thread creation, or private message tables are rewritten to use the shadow authenticated user as the value for the author field (or the sender field, in the case of a private message). The only exception is when a query is totally untainted. For example, when a new user registers, a welcome message is sent from a fake system mailer user. As this query is totally untainted, we allow it to be inserted into the private message table, despite the fact that the identity of the sender is clearly forged. We added fields to the post and thread tables to store the username of the current shadow authenticated user, as these tables did not directly store the author's username. We then explicitly instrumented all SQL INSERT statements into these tables to append this information accordingly.

Any user may read from the thread or post databases. However, our ACL rules further constrain reads from the private message database. A row may only be read from the private message database if the 'from' or 'to' fields of the row are equal to the current shadow authenticated user.

We manually instrumented all SELECT queries from the private message table to add this condition to the WHERE clause of the query. In total, we modified 16 SQL queries to enforce both our private message protection and our INSERT rules to prevent spoofing messages, threads, and posts. We also inserted 82 authorization checks to enforce the rest of the ACL.

A vulnerability exists in the private message script of this application [6]. This script incorrectly validates cookies, missing a vital authentication check. This allows an attacker to forge a cookie and be treated as an arbitrary web application user by the script. Successful exploitation of this vulnerability gives an attacker the ability to access any user's private messages.

Using Nemesis, when this attack is exploited, the attacker can fool the private message script into thinking he is an arbitrary user due to a missing access control check. The shadow authentication for the attack still has the last safe, correct authenticated username, and is not affected by the attack. Thus, the attacker is unable to access any unauthorized messages, because our ACL rules only allow a user to retrieve messages from the private message table when the sender or recipient field of the message is equal to the current shadow authenticated user. Similarly, the attacker cannot abuse the private message script to forge messages, as our ACLs constrain any messages inserted into the private message table to have the sender field set to the current shadow authenticated username.

DeluxeBB allows admin users to execute arbitrary SQL queries. We verified that this results in false positives in existing DIFT SQL injection protection analyses. After adding an ACL allowing SQL injection for web clients shadow authenticated as the admin user, all SQL injection false positives were eliminated.

6.7 PhpMyAdmin

PhpMyAdmin is a popular web application used to remotely administer and manage MySQL databases. This application does not build its own authentication system; instead, it checks usernames and passwords against MySQL's own user database. A web client is validated only if the underlying MySQL database accepts their username and password.

We treat the MySQL database connection function as a third-party authentication function as detailed in Section 3.2.2. We instrumented the call to the MySQL database connection function to perform shadow authentication, authenticating a user if the username and password supplied to the database are both tainted, and if the login was successful.

The ACL for phpMyAdmin is very different from other web applications, as phpMyAdmin is intended to provide an authenticated user with unrestricted access to the underlying database. The only ACL we include is a rule allowing authenticated users to submit full SQL database queries. We implemented this by modifying our SQL injection protection policy defined in Section 5.6 to treat tainted SQL operators in user input as unsafe only when the current user was unauthenticated. Without this policy, any attempt to submit a query as an authenticated user results in a false positive in the DIFT SQL injection protection policy. We confirmed that adding this ACL removes all observed SQL injection false positives, while still preventing unauthenticated users from submitting SQL queries.

6.8 Performance

We also performed performance tests on our prototype implementation, measuring overhead against an unmodified version of PHP. We used the bench.php microbenchmark distributed with PHP, where our overhead was 2.9% compared to unmodified PHP. This is on par with prior results reported by object-level PHP tainting projects [29]. However, bench.php is a microbenchmark which performs CPU-intensive operations. Web applications often are network or I/O-bound, reducing the real-world performance impact of our information flow tracking and access checks.

To measure performance overhead of our prototype for web applications, we used the Rice University Bidding System (RUBiS) [24]. RUBiS is a web application benchmark suite, and has a PHP implementation that is approximately 2,100 lines of code. Our ACL for RUBiS prevents users from impersonating another user when placing bids, purchasing an item, or posting a bid comment. Three access checks were added to enforce this ACL. We compared latency and throughput for our prototype and an unmodified PHP, and found no discernible performance overhead.

7 Future Work

There is much opportunity for further research in preventing authentication bypass attacks. A fully developed, robust policy language for expressing access control in web applications must be developed. This will require support for SQL query rewriting, which can be implemented by developing a full SQL query parser or by using database table views and triggers similar to the approach described in [18].

Additionally, all of our ACL rules are derived from real-world behavior observed when using the web application. While we currently generate such ACLs manually, it should be possible to create candidate ACLs automatically. A log of all database and file operations, as well as the current shadow authenticated user at the time of the operation, would be recorded. Using statistical techniques such as machine learning [14], this log could be analyzed and access control files generated based on application behavior. This would allow system administrators to automatically generate candidate ACL lists for their web applications without a precise understanding of the access control rules used internally by the webapp.

8 Related Work

Preventing web authentication and authorization vulnerabilities is a relatively new area of research. The only other work in this area of which we are aware is the CLAMP research project [18]. CLAMP prevents authorization vulnerabilities in web applications by migrating the user authentication module of a web application into a separate, trusted virtual machine (VM). Each new login session forks a new, untrusted session VM and forwards any authentication credentials to the authentication VM. Authorization vulnerabilities are prevented by a trusted query restricter VM which interposes on all session VM database accesses, examining queries to enforce the appropriate ACLs using the username supplied by the authentication VM.

In contrast to Nemesis, CLAMP does not prevent authentication vulnerabilities as the application authentication code is part of the trusted user authentication VM. CLAMP also cannot support access control policies that require taint information as it does not use DIFT. Furthermore, CLAMP requires a new VM to be forked when a user session is created, and experimental results show that one CPU core could only fork two CLAMP VMs per second. This causes significant performance degradation for throughput-driven, multi-user web applications that have many simultaneous user sessions.

Researchers have extensively explored the use of DIFT to prevent security vulnerabilities. Robust defenses against SQL injection [15, 27], cross-site scripting [25], buffer overflows [5] and other attacks have all been proposed. DIFT has been used to prevent security vulnerabilities in most popular web programming languages, including Java [8], PHP [16], and even C [15]. This paper shows how DIFT techniques can be used to address authentication and access control vulnerabilities. Furthermore, DIFT-based solutions to attacks such as SQL injection have false positives in the real-world due to a lack of authentication information. Nemesis avoids such false positives by incorporating authentication and authorization information at runtime.

Much work has also been done in the field of secure web application design. Information-flow aware programming language extensions such as Sif [4] have been developed to prevent information leaks and security vulnerabilities in web application programs using the language and compiler. Unfortunately these approaches typically require legacy applications to be rewritten in the new, secure programming language.

Some web application frameworks provide common authentication or authorization code libraries [1, 2, 7]. The use of these authentication libraries is optional, and many application developers choose to partially or entirely implement their own authentication and authorization systems. Many design decisions, such as how users are registered, how lost passwords are recovered, or what rules govern access control to particular database tables are application-specific. Moreover, these frameworks do not connect the user authentication mechanisms with the access control checks in the underlying database or file system. As a result, the application programmer must still apply access checks at every relevant filesystem and database operation, and even a single mistake can compromise the security of the application.

Operating systems such as HiStar [32] and Flume [12] provide process-granularity information flow control support. One of the key advantages of these systems is that they give applications the flexibility to define their own security policies (in terms of information flow categories), which are then enforced by the underlying kernel. A web application written for HiStar or Flume can implement its user authentication logic in terms of the kernel's information flow categories. This allows the OS kernel to then ensure that one web user cannot access data belonging to a different web user, even though the OS kernel doesn't know how to authenticate a web user on its own. Our system provides two distinct advantages over HiStar and Flume. First, we can mitigate vulnerabilities in existing web applications, without requiring the application to be re-designed from scratch for security. Second, by providing sub-process-level information flow tracking and expressive access control checks instead of labels, we allow programmers to specify precise security policies in a small amount of code.

9 Conclusion

This paper presented Nemesis, a novel methodology for preventing authentication and access control bypass attacks in web applications. Nemesis uses Dynamic Information Flow Tracking to automatically detect when application-specific users are authenticated, and constructs a *shadow authentication system* to track user authentication state through an additional HTTP cookie.

Programmers can specify access control lists for resources such as files or database entries in terms of application-specific users, and Nemesis automatically enforces these ACLs at runtime. By providing a shadow authentication system and tightly coupling the authentication system to authorization checks, Nemesis prevents a wide range of authentication and access control bypass attacks found in today's web applications. Nemesis can also be used to improve precision in other security tools, such as those that find SQL injection bugs, by avoiding false positives for properly authenticated requests.

We implemented a prototype of Nemesis in the PHP interpreter, and evaluated its security by protecting seven real-world applications. Our prototype stopped all known authentication and access control bypass attacks in these applications, while requiring only a small amount of work from the application developer, and introducing no false positives. For most applications we evaluated, the programmer had to write less than 100 lines of code to avoid authentication and access control vulnerabilities. We also measured performance overheads using PHP benchmarks, and found that our impact on web application performance was negligible.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback. Michael Dalton was supported by a Sequoia Capital Stanford Graduate Fellowship. This work was supported NSF Awards number 0546060 and 0701607.

References

[1] Turbogears documentation: Identity management. `http://docs.turbogears.org/1.0/Identity`.

[2] Zope security. `http://www.zope.org/Documentation/Books/ZDG/current/Security.stx`.

[3] BilboBlog admin/index.php Authentication Bypass Vulnerability. `http://www.securityfocus.com/bid/30225`, 2008.

[4] S. Chong, K. Vikram, and A. C. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *Proceedings of the 16th Annual USENIX Security Symposium*, 2007.

[5] M. Dalton, H. Kannan, and C. Kozyrakis. Real-World Buffer Overflow Protection for Userspace and Kernelspace. In *Proceedings of the 17th Annual USENIX Security Symposium*, 2008.

[6] DeluxeBB PM.PHP Unauthorized Access Vulnerability. `http://www.securityfocus.com/bid/19418`, 2006.

[7] Django Software Foundation. User authentication in Django. `http://docs.djangoproject.com/en/dev/topics/auth/`.

[8] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for java. *Annual Computer Security Applications Conference*, 2005.

[9] N. Hardy. The Confused Deputy: (or why capabilities might have been invented). *SIGOPS Operating System Review*, 1988.

[10] Microsoft Internet Information Server Hit Highlighting Authentication Bypass Vulnerability. `http://www.securityfocus.com/bid/24105`, 2007.

[11] M. Krohn, P. Efstathopoulos, C. Frey, F. Kaashoek, E. Kohler, D. Mazières, R. Morris, M. Osborne, S. VanDeBogart, and D. Ziegler. Make Least Privilege a Right (Not a Privilege). In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, 2005.

[12] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[13] Linpha User Authentication Bypass Vulnerability. `http://secunia.com/advisories/12189`, 2004.

[14] E. Martin and T. Xie. Inferring access-control policy properties via machine learning. In *Proc. 7th IEEE Workshop on Policies for Distributed Systems and Networks*, 2006.

[15] S. Nanda, L.-C. Lam, and T. Chiueh. Dynamic multi-process information flow tracking for web application security. In *Proceedings of the 8th International Conference on Middleware*, 2007.

[16] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications using Precise Tainting. In *Proceedings of the 20th IFIP Intl. Information Security Conference*, 2005.

[17] Top 10 2007 - Broken Authentication and Session Management. `http://www.owasp.org/index.php/Top_10_2007-A7`, 2007.

[18] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig. CLAMP: Practical Prevention of Large-Scale Data Leaks. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, May 2009.

[19] phpFastNews Cookie Authentication Bypass Vulnerability. `http://www.securityfocus.com/bid/31811`, 2008.

[20] PHP iCalendar Cookie Authentication Bypass Vulnerability. `http://www.securityfocus.com/bid/31320`, 2008.

[21] Php Stat Vulnerability Discovery. `http://www.soulblack.com.ar/repo/papers/advisory/PhpStat_advisory.txt`, 2005.

[22] PHP: Using Register Globals. `http://us2.php.net/register_globals`.

[23] PhpMyAdmin control user. `http://wiki.cihar.com/pma/controluser`.

[24] Rice University Bidding System. `http://rubis.objectweb.org`, 2009.

[25] P. Saxena, D. Song, and Y. Nadji. Document structure integrity: A robust basis for cross-site scripting defense. *Network and Distributed Systems Security Symposium*, 2009.

[26] S. R. Shariq, A. Mendelzon, S. Sudarshan, and P. Roy. Extending Query Rewriting Techniques for Fine-Grained Access Control. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2004.

[27] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Proceedings of the 33rd Symposium on Principles of Programming Languages*, 2006.

[28] The MITRE Corporation. Common vulnerabilities and exposures (CVE) database. `http://cve.mitre.org/data/downloads/`.

[29] W. Venema. Taint support for php. `http://wiki.php.net/rfc/taint`, 2008.

[30] Web Application Security Consortium. 2007 web application security statistics. `http://www.webappsec.org/projects/statistics/wasc_wass_2007.pdf`.

[31] WordPress Cookie Integrity Protection Unauthorized Access Vulnerability. `http://www.securityfocus.com/bid/28935`, 2008.

[32] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

Static Enforcement of Web Application Integrity Through Strong Typing

William Robertson
wkr@cs.ucsb.edu
Computer Security Group
UC Santa Barbara

Giovanni Vigna
vigna@cs.ucsb.edu
Computer Security Group
UC Santa Barbara

Abstract

Security vulnerabilities continue to plague web applications, allowing attackers to access sensitive data and co-opt legitimate web sites as a hosting ground for malware. Accordingly, researchers have focused on various approaches to detecting and preventing common classes of security vulnerabilities in web applications, including anomaly-based detection mechanisms, static and dynamic analyses of server-side web application code, and client-side security policy enforcement.

This paper presents a different approach to web application security. In this work, we present a web application framework that leverages existing work on strong type systems to statically enforce a separation between the *structure* and *content* of both web documents and database queries generated by a web application, and show how this approach can automatically prevent the introduction of both server-side cross-site scripting and SQL injection vulnerabilities. We present an evaluation of the framework, and demonstrate both the coverage and correctness of our sanitization functions. Finally, experimental results suggest that web applications developed using this framework perform competitively with applications developed using traditional frameworks.

Keywords: Web applications, strongly typed languages, functional languages, cross-site scripting, SQL injection.

1 Introduction

In the last decade, web applications have become an extremely popular means of providing services to large numbers of users. Web applications are relatively easy to develop, the potential audience of a web application is a significant proportion of the planet's population [36, 10], and development frameworks have evolved to the point that web applications are approaching traditional thick-client applications in functionality and usability.

Unfortunately, web applications have also been found to contain many security vulnerabilities [40]. Web applications are also widely accessible and often serve as an interface to large amounts of sensitive data stored in back-end databases. Due to these factors, web applications have attracted much attention from cyber-criminals. Attackers commonly exploit web application vulnerabilities to steal confidential information [41] or to host malware in order to build botnets, both of which can be sold to the highest bidder in the underground economy [46].

By far, the most prevalent security vulnerabilities present in web applications are cross-site scripting (XSS) and SQL injection vulnerabilities [42]. Cross-site scripting vulnerabilities are introduced when an attacker is able to inject malicious scripts into web content to be served to other clients. These scripts then execute with the privileges of the web application delivering the content, and can be used to steal authentication credentials or to install malware, among other nefarious objectives. SQL injections occur when malicious input to a web application is allowed to modify the structure of queries issued to a back-end database. If successful, an attacker can typically bypass authentication procedures, elevate privileges, or steal confidential information.

Accordingly, much research has focused on detecting and preventing security vulnerabilities in web applications. One approach is to deploy web application firewalls (WAFs), usually incorporating some combination of misuse and anomaly detection techniques, in order to protect web applications from attack [6, 14, 8, 29, 45]. Anomaly detection approaches are attractive due to their black-box approach; they typically require no *a priori* knowledge of the structure or implementation of a web application in order to provide effective detection.

Another significant focus of research has been on applying various static and dynamic analyses to the source code of web applications in order to identify and mitigate security vulnerabilities [21, 33, 25, 2, 7, 50]. These approaches have the advantage that developers can continue to create web applications using traditional languages and frameworks, and periodically apply a vulnerability analysis tool to provide a level of assurance that no security-relevant flaws are present. Analyzing web applications is a complex task, however, as is the interpretation of the results of such security tools. Additionally, several approaches require developers to specify security policies to be enforced in a specialized language.

A more recent line of research has focused on providing client-side protection by enforcing security policies within the web browser [43, 22, 13]. These approaches show promise in detecting and preventing client-side attacks against newer web applications that aggregate content from multiple third parties, but the specification of policies to enforce is generally left to the developer.

In this paper, we propose a different approach to web application security. We observe that cross-site scripting and SQL injection vulnerabilities can be viewed as a failure on the part of the web application to enforce a separation of the *structure* and the *content* of documents and database queries, respectively, and that this is a result of treating documents and queries as untyped sequences of bytes. Therefore, instead of protecting or analyzing existing web applications, we describe a framework that strongly types both documents and database queries. The framework is then responsible for automatically enforcing a separation between structure and content, as opposed to the *ad hoc* sanitization checks that developers currently must implement. Consequently, the integrity of documents and queries generated by web applications developed using our framework are automatically protected, and thus, *by construction*, such web applications are not vulnerable to server-side cross-site scripting and SQL injection attacks.

To illustrate the problem at hand, consider that HTML or XHTML documents to be presented to a client are typically constructed by concatenating strings. Without additional type information, a web application framework has no means of determining that the following operations could lead to the introduction of a cross-site scripting vulnerability:

```
String result = "<div>" + userInput + "</div>";
```

The key intuition behind our work is that because both documents and database queries are strongly typed in our framework, the framework can distinguish between the structure (`<div>` and `</div>`) and the content (`userInput`) of these critical objects, and enforce their integrity automatically.

In this work, we leverage the advanced type system of Haskell, since it offers a natural means of expressing the typing rules we wish to impose. In principle, however, a similar framework could be implemented in any language with a strong type system that allows for some form of multiple inheritance (e.g., Java or C#).

In summary, the main contributions of this paper are the following:

- We identify the lack of typing of web documents and database queries as the underlying cause of cross-site scripting and SQL injection vulnerabilities.

- We present the design of a web application development framework that automatically prevents the introduction of cross-site scripting and SQL injection vulnerabilities by strongly typing both web documents and database queries.

- We evaluate our prototype web application framework, demonstrate the coverage and correctness of its sanitization functions, and show that applications under our framework perform competitively with those using existing frameworks.

The remainder of this paper is structured as follows. Section 2 presents the design of a strongly typed web application framework. The specification of documents under the framework and how their integrity is enforced is discussed in Section 3, and similarly for SQL queries in Section 4. Section 5 evaluates the design of the framework, and demonstrates that web applications developed under this framework are free from certain classes of vulnerabilities. Related work is discussed in Section 6. Finally, Section 7 concludes and presents avenues for further research.

2 Framework design

At a high level, the web application framework is composed of several familiar components. A web server component processes HTTP requests from web clients and forwards these requests in an intermediate form to the application server based on one of several configuration parameters (e.g., URL path prefix). These requests are directed to one of the web applications hosted by the application server. The web application examines any parameters to the request, performs some processing during which queries to a back-end database may be executed, and generates a document. Note that in the following, the terms "document" or "web document" shall generically refer to any text formatted according to the HTML or XHTML standards. This document is then returned down the component stack to the web server, which sends the document as part of an HTTP response to the web client that originated the request. A graphical depiction of this architecture is given in Figure 1.

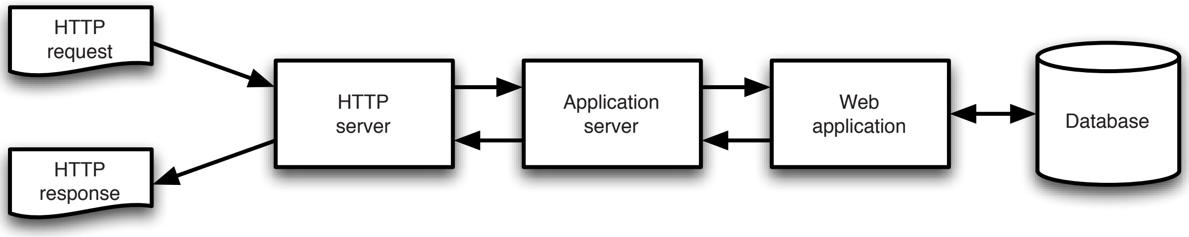


Figure 1: Architectural overview of the web application framework.

Web applications developed for our framework are structured as a set of functions with access to a combination of configuration data and application state. More precisely, web applications execute inside the App monad. Monads are a category theoretic construction that have found wide application in Haskell to sequence actions or isolate code that can produce side effects.[1] For the purposes of our framework, we use the App monad to thread implicit state through the functions comprising a web application, and to provide a controlled interface to potentially dangerous functions. In particular, the App monad itself is structured as a stack of monad transformers that provide a functional interface to a read-only configuration type AppConfig, a read-write application state type AppState, and filtered access to the IO monad. The definitions for AppConfig and AppState are given in Figures 2 and 3.

```
data AppConfig = AppConfig {
     appCfgPort :: Int,
     appCfgPrefix :: String,
     appCfgRoutes :: RouteMap,
     appCfgFileRoot :: FilePath,
     appCfgDBConn :: Connection,
     appCfgDBStmts :: StmtMap
}
```

Figure 2: Definition for the AppConfig type.

The AppConfig type holds static information relating to the configuration of the application, including the port on which to listen for HTTP requests and the root directory of static files to serve from the filesystem. Of particular interest, however, are the RouteMap and StmtMap fields. The RouteMap type describes how URL paths are mapped to values of type DocumentGen, which are simply functions that generate documents within the App monad. In addition, the RouteMap type contains a default DocumentGen type that specifies an error page. Given an incoming HTTP request destined for a particular web application, the application server uses that application's RouteMap type to determine the proper function to call in order to generate the document to be returned to the client.[2] Finally, the StmtMap type associates unique database query identifiers to prepared statements that can be executed by a document generator.

[1]For further information on monads, please refer to [37, 48].

```
data AppState = AppState {
     appStClient :: Maybe SockAddr,
     appStUrl :: Maybe Url
}
```

Figure 3: Definition for the AppState type.

The AppState type contains mutable state that is specific to each request for a document. In particular, one field records information indicating the source of the request. Additionally, another field records the URL that was requested, including any parameters that were specified by the client. More complex state types that hold additional information (e.g., cached database queries or documents) are possible, however.

3 Document structure

In this section, we introduce the means by which documents are specified under the framework. Then, we discuss how these specifications allow the framework to automatically contain the potentially harmful effects of dynamic data.

3.1 Document specification

Once an appropriate route from the RouteMap structure has been selected by the application server, the associated document generator function is executed within the context of the App monad (i.e., with access to the

[2]This construction is similar to the "routes" packages present in popular web development frameworks such as Rails [18] and Pylons [4].

```
data Document Document {
     docType :: DocumentType,
     docHead :: DocumentHead,
     docBody :: DocumentBody
}

data DocumentType = DOC_TYPE_HTML_4_01_STRICT
                  | DOC_TYPE_HTML_4_01_TRANS
                  | ...
                  | DOC_TYPE_XHTML_1_1

data DocumentHead = DocumentHead {
     docTitle :: String,
     docLinks :: [Node],
     docScripts :: [Node],
     docBaseUrl :: Maybe Url,
     docBaseTarget :: Maybe Target,
     docProfile :: [Url]
}

data DocumentBody = DocumentBody {
     docBodyNode :: Node
}
```

Figure 4: Definition for the `Document` type.

```
data Node = TextNode {
     nodeText :: String
}        | AnchorNode {
     anchorAttrs :: NodeAttrs,
     anchorHref :: Maybe Url,
     anchorRel :: Maybe Relationship,
     anchorRev :: Maybe Relationship,
     anchorTarget :: Maybe Target,
     anchorType :: Maybe MimeType,
     anchorCharset :: Maybe CharSet,
     anchorLang :: Maybe Language,
     anchorName :: Maybe AttrValue,
     anchorShape :: Maybe Shape,
     anchorCoords :: Maybe Coordinates,
     anchorNodes :: [Node]
}        | DivNode {
     divAttrs :: NodeAttrs,
     divNodes :: [Node]
} ...
```

Figure 5: Sample `Node` definitions.

configuration and current state of the application). The document generator function processes the request from the application server and returns a variable of type `Document`. The definition of the `Document` type and its constituent types are shown in Figure 4.

As is evident, documents in our framework are not represented as an unstructured stream of bytes. Rather, the structure of the `Document` type closely mirrors that of parsed HTML or XHTML documents. The `DocumentType` field indicates the document's type, such as "HTML 4.01 Transitional" or "XHTML 1.1". The `DocumentHead` type contains information such as the title and client-side code to execute. The `DocumentBody` type contains a single field that represents the root of a tree of nodes that represent the body of the document.

Each node in this tree is an instantiation of the `Node` type. Each `Node` instantiation maps to a distinct (X)HTML element, and records the set of possible properties of that element. For instance, the `TextNode` data constructor creates a `Node` that holds a text string to be displayed as part of a document. The `AnchorNode` data constructor, on the other hand, creates a `Node` that holds information such as the `href` attribute, `rel` attribute, and a list of child nodes corresponding to the text or other elements that comprise the "body" of the link. A partial definition of the `Node` type is presented in Figure 5.

With this construction, the entire document produced by a web application in our framework is strongly typed. Instead of generating a document as a byte stream, document structure is explicitly encoded as a tree of nodes. Furthermore, each element and element attribute has an associated type that constrains, to one degree or another, the range of possible values that can be represented. For instance, the `MimeType`, `CharSet`, and `Language` types are examples of enumerations that strictly limit the set of possible values the attribute can take to legal values. Standard (X)HTML element attributes (e.g., `id`, `class`, `style`) are represented with the `NodeAttr` type. Optional attributes are represented using either the `Maybe` type,[3] or as an empty list if multiple elements are allowed.

Note that it is possible for a `Document` to represent an (X)HTML document that is not necessarily consistent with the respective W3C grammars that specify the set of well-formed documents. One example is that any `Node` instantiation may appear as the child of any other `Node` that can hold children, which violates the official grammars in several instances. Strict conformance with the W3C standards is not, however, our goal.[4]

Instead, the typing scheme presented here allows our framework to specify a separation between the *structure* and the *content* of the documents a web application generates. More precisely, the dynamic data that enters a web application as part of an HTTP request (e.g., as a GET or POST parameter) can indirectly influence the structure of a document. For instance, a search request to a web application may result in a variable number of

[3]`Maybe` allows for the absence of a value, as Haskell does not possess nullable types. For example, the type `Maybe a` can be either `Just "..."` or `Nothing`.

[4]Indeed, standards-conforming documents have been shown to be difficult to represent in a functional language [12].

```
class Render a where
      render :: a -> String

instance Render AttrValue where
         render = quoteAttr

quoteAttr :: AttrValue -> String
quoteAttr a = foldl' step [] (attrValue a)

step acc c | c == '<' = acc ++ "&lt;"
           | c == '>' = acc ++ "&gt;"
           | c == '&' = acc ++ "&amp;"
           | c == '"' = acc ++ "&quot;"
           | otherwise = acc ++ [c]
```

Figure 6: `Render` typeclass definition and (simplified) instance example. Here, `quoteAttr` performs a left fold over attribute values using `foldl'`, which applies the `step` function to each character of the string and accumulates the result. The definition of `step` specifies a number of *guards*, where `| c == '<'` is a condition that must be satisfied for the statement `acc ++ "&lt;"` to execute. This statement simply appends the string `"&lt;"` to `acc`, the accumulator, in order to build a new, sanitized string. If no guard condition is satisfied, the character is appended without conversion.

table rows in the generated document depending on the number of results returned from a database query. Due to our framework, however, client-supplied data cannot *directly* modify the structure of the document in such a way that a code injection can occur.

3.2 Enforcing document integrity

Once a `Document` has been constructed by the web application in response to a client request, it is returned to the application server. The application server is responsible for converting this data structure into a format the client can understand – that is, it must *render* the document into a stream of bytes representing an (X)HTML document. Consequently, the set of types that can comprise a `Document` are instances of the `Render` typeclass, shown in Figure 6.[5]

The `Render` type class specifies that any instance of the class must implement the `render` function. From the type signature, the semantics of the function are clear: `render` converts an instance type into a string representation suitable for presentation to a client. Crucially, for our purposes, the `Render` type class is also responsible for enforcing the integrity of a document's structure.

[5]Haskell typeclasses are roughly similar to Java interfaces, in that they specify a function interface that all instances (in Java, implementors) must provide.

As an example, Figure 6 presents a simplified `render` definition for the `AttrValue` type that is used to indicate element attribute strings that may assume (almost) arbitrary values. In order to preserve the integrity of the document, an attribute value must not contain certain characters that would allow an attacker to inject malicious code into the document. Consider, for instance, the following element:

```
<input type="hidden" name="h1" value="..."/>
```

Now, suppose an attacker submitted the following string as part of a request such that it was reflected to another client as the value of the hidden input field:

```
"/>
<script src="http://example.com/malware.js">
</script>
<span id="
```

The result would be the following:

```
<input type="hidden" name="h1" value=""/>
<script src="http://example.com/malware.js">
</script>
<span id=""/>
```

To prevent such an injection from occurring, the `render` function for the `AttrValue` class applies a sanitization function on the string wrapped by `AttrValue`. Any occurrence of an unsafe character is replaced by an equivalent HTML entity encoding that can safely appear as part of an attribute value.[6] Similar `render` functions are defined for the set of types that can comprise a `Document`.

Therefore, to prepare a `Document` as part of an HTTP response to a client, the application server applies the `render` function to the document, which recursively converts the data structure into an (X)HTML document. As part of this process, the content of the document is sanitized by type-specific `render` functions, ensuring that client-supplied input to the web application cannot modify the document structure in such a way as to result in a client-side code injection.

4 SQL query structure

Similarly to the case of documents, SQL queries are given structure in our framework through the application of strong typing rules that control how the structure of the query can be combined with dynamic data. In this section, we examine the structure of SQL queries and discuss two mechanisms by which SQL query integrity is enforced under the framework.

[6]In the real implementation, the sanitization function is somewhat more complex, as there are multiple encodings by which an unsafe character can be injected. The example function given here is simplified for the purposes of presentation.

```
INSERT INTO users(login, passwd)
VALUES(?, ?)

SELECT * FROM users
WHERE login='admin' AND passwd='test'

UPDATE users SET passwd='$passwd'
WHERE login='$login'
```

Figure 7: Examples of SQL queries.

4.1 Query specification

SQL queries, as shown in Figure 7, are composed of clauses, predicates, and expressions. For instance, a clause might be `SELECT *` or `UPDATE users`. An example of a predicate is `login='admin'`, where `'admin'` is an expression. Clauses, predicates, and expressions are themselves composed of static tokens, such as keywords (`SELECT`) and operators (`=`), and dynamic tokens, such as table identifiers (`users`) or data values (`'admin'`).

Typically, the structure of a SQL query is fixed.[7] Specifically, a query will have a static keyword denoting the operation to perform, will reference a static set of tables and fields, and specify a fixed set of predicates. Generally, the only components of a query that change from one execution to the next are data values, and, even then, their number and placement remain fixed.

SQL injection attacks rely upon the ability of the attacker to modify the structure of a query in order to perform a malicious action. When SQL queries are constructed using string operations without sufficient sanitization applied to user input, such attacks become trivial. For instance, consider the `UPDATE` query shown in Figure 7. If an attacker were to supply the value "`quux' OR login='admin`" for the `$login` variable, the following query would result:

```
UPDATE users SET passwd='foo'
WHERE login='quux' OR login='admin'
```

Because the attacker was able to inject single quotes, which serve as delimiters for data values, the structure of the query was changed, resulting in a privilege escalation attack.

4.2 Integrity enforcement with static query structure

In contrast to the case of document integrity enforcement, a well-known solution exists for specifying SQL query structure: prepared statements. Prepared statements are a form of database query consisting of a parameterized query template containing placeholders where dynamic data should be substituted. An example is shown in Figure 8, where the placeholders are signified by the "?" character.

```
SELECT * FROM users WHERE login=? AND passwd=?
   UPDATE users SET passwd=? WHERE login=?
```

Figure 8: Examples of prepared statements, where "?" characters serve as placeholders for data substitution.

A prepared statement is typically parsed and constructed prior to execution, and stored until needed. When an actual query is to be issued, variables that may contain client-supplied data are *bound* to the statement. Since the query has already been parsed and the placeholders specified, the structure of the query cannot be modified by the traditional means of providing malicious input designed be interpreted as part of the query. In the case of the injection attack described previously, the result would be as the following (note that the injected single quotes have been escaped):

```
UPDATE users SET passwd='foo'
WHERE login='quux'' OR login=''admin'
```

From our perspective, the query has been typed as a composition of static and dynamic elements; it is exactly this distinction between structure and content that we wish to enforce. Haskell's database library (HDBC), as do most other languages, supports the use of prepared statements. Therefore, the framework exports functions that allow a web application to associate prepared statements with a unique identifier in the `AppConfig` type. During request processing, a document generator can then retrieve a prepared statement using the identifier, bind values to it, and execute queries that are not vulnerable to injection attacks.

One detail remains, however. The HDBC library also provides functions allowing traditional *ad hoc* queries that are assembled as concatenations of strings to be executed. Without any other modification to the framework, a web application developer would be free to directly call these functions and bypass the protections afforded by the framework. Therefore, an additional component is required to encapsulate the HDBC interface and prevent execution of these unsafe functions. This component takes the form of a monad transformer `AppIO`, which simply wraps the `IO` monad and exposes only those functions that are considered safe to execute. The structure of this stack is shown in Figure 9. In this environment, within which all web applications using the framework operate, unsafe database execution functions are inaccessible, since they will fail to type-check. Thus, assuming

[7] This is not always the case, but the case of dynamic query structure will be considered later in this section.

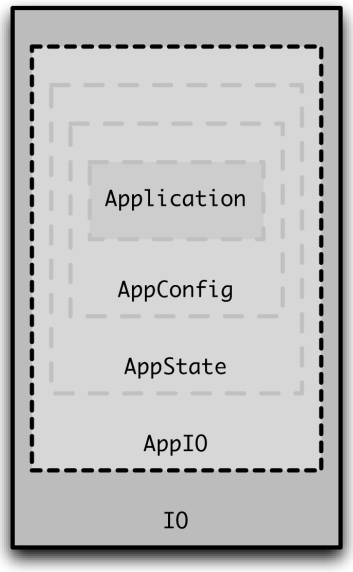


Figure 9: Graphical representation of the monad stack within which framework applications execute. The `AppIO` monad encapsulates applications, preventing them from calling unsafe functions within the `IO` monad.

the correctness of the HDBC prepared statement interface, web applications developed using the framework are not vulnerable to SQL injection.

4.3 Integrity enforcement with dynamic query structure

Though most SQL queries possess a fixed structure, there does exist a small class of SQL queries that exhibit dynamic structure. For instance, many SQL database implementations provide a set membership operator, where queries of the form

```
SELECT * FROM users WHERE
login IN ('admin', 'developer', 'tester')
```

can be expressed. In this case, the size of the set of data values can often change at runtime. Another example is the case where the structure of queries is determined by the user, for instance through a custom search form where many different combinations of predicates can be dynamically expressed. Unfortunately, since these queries cannot be represented using prepared statements, they cannot be protected using the monadic encapsulation technique described previously.

Therefore, a second database interface is exposed by the framework to the application developers. Instead of relying upon prepared statements, this interface allows developers to dynamically construct queries as a tree of algebraic data types as in the case of web documents.

```
data Select = Select {
     sFields :: [Expr],
     sTables :: [Expr],
     sCons :: Maybe Expr,
     sGrpFields :: [Expr],
     sGrpCons :: Maybe Expr,
     sOrdFields :: [Expr],
     sLimit :: Maybe Int,
     sOffset :: Maybe Int,
     sDistinct :: Bool
}

data Expr = EXPR_TABLE Table
          | EXPR_FIELD Field
          | EXPR_DATA String
          | EXPR_NOT Expr
          | EXPR_OR Expr Expr
          | EXPR_AND Expr Expr
          | ...

data Table = Table {
     tName :: String,
     tAlias :: Maybe String
}

data Field = Field {
     fName :: String,
     fAlias :: Maybe String
}
```

Figure 10: Definition for the `Select` type.

Figure 10 shows an example of the type representing a `SELECT` query.

To populate instances of these types, the interface provides a set of combinators, or higher-order functions, that can be chained together. These combinators, which assume names similar to SQL keywords, implement an embedded domain-specific language (DSL) that allows application developers to naturally specify dynamic queries within the framework. For instance, a query could be constructed using the following sequence of function invocations:

```
qSelect [qField "*"] >>=
qFrom [qTable "users"] >>=
qWhere (((qField "login") == (qData "admin")) &&
((qField "passwd") == (qData "test")))
```

Similar to the case of the `Document` type, queries constructed in this manner are transformed into raw SQL statements solely by the framework.[8] Therefore, the types that represent queries also implement the `Render`

[8]Note that, as in the case of web documents, we do not attempt to enforce the generation of correct SQL, but rather focus on preventing attacks by preserving query structures specified by the developer.

Context	Semantics
Document nodes	Conversion to static string
Document node attributes	Encoding of HTML entities
Document text	Encoding of HTML entities
URL components	Encoding of HTML entities, percent encoding
SQL static value	Removal of spaces, comments, quotes
SQL data value	Escaping of quotes

Table 1: Example contexts for which specific sanitization functions are applied, and the semantics of those sanitization functions under various encodings.

typeclass. Consequently, sanitization functions must be applied to each of the fields comprising the query types, such that the intended structure of the query cannot be modified. This can be accomplished by enforcing the conditions that no data value may contain an unescaped single quote, and that all remaining query components may not contain spaces, single quotes, or character signifying the beginning of a comment. Assuming that these sanitization functions are correct, this construction renders applications developed under the framework invulnerable to SQL injection attacks while allowing for more powerful query specifications.

5 Evaluation

To demonstrate that web applications developed using our framework are secure by construction from server-side XSS and SQL injection vulnerabilities, we conducted an evaluation of the system. First, we demonstrate that all dynamic content contained in a `Document` must be sanitized by an application of the `render` function, and that a similar condition holds for dynamically-generated SQL queries. Then, we provide evidence that the sanitization functions themselves are correct – that is, they successfully strip or encode unsafe characters. We also verify that the prepared statement library prevents injections, as expected. Finally, to demonstrate the viability of the framework, an experiment to evaluate the performance of a web application developed using the framework is conducted.

5.1 Sanitization function coverage

The goal of the first experiment was to justify the claim that all dynamic content contained in a `Document` or query type must be sanitized prior to presentation to the client that originated the request. To accomplish this, a static control flow analysis of the framework was performed. Figure 11 presents a control flow graph of the application server in a simplified form, where function calls are sequenced from left to right. Of particular interest is the `renderDoc` function, which retrieves the appropriate document generator given a URL path, executes it in the call to `route`, sanitizes it by applying `render`, and creates an HTTP response by calling `make200`. The sanitized document is then returned to `procRequest`, which writes it to the client. Therefore, the entire process of converting the document to a byte stream for presentation to the client is solely due to the recursive `render` application. Similarly, because the only interface exposed to applications to execute SQL queries are `execStmt` and `execPrepStmt` from within the `App` monad, queries issued by applications under the framework must be sanitized either by the framework or the HDBC prepared statement functions.

Figure 12 displays a subset of the full control flow graph depicting an instance of the `render` function for the `AnchorNode Node` instantiation. For clarity of presentation, multiple calls to `render` and `maybeRenderAttr` have been collapsed into single nodes. Recall from Figure 5 that the definition of `AnchorNode` does not contain any bare strings; instead, each field of the type is either itself a composite type, or an enumeration for which a custom `render` function is defined. Since no other string conversion function is applied in this subgraph, we conclude that all data contained in an `AnchorNode` variable must be filtered through a sanitization function.

The analysis of this single case generalizes to the set of all types that can comprise a `Document` or query type. In total, 163 distinct sanitization function definitions were checked to sanitize the contexts shown in Table 1. For each function, our analysis found that no irreducible type was concatenated to the document byte stream without first being sanitized.

5.2 Sanitization function correctness

The goal of this experiment is to determine whether the sanitization functions employed by the framework are correct (i.e., whether all known sequences of dangerous characters are stripped or encoded). To establish this, we applied a dynamic test-driven approach using the

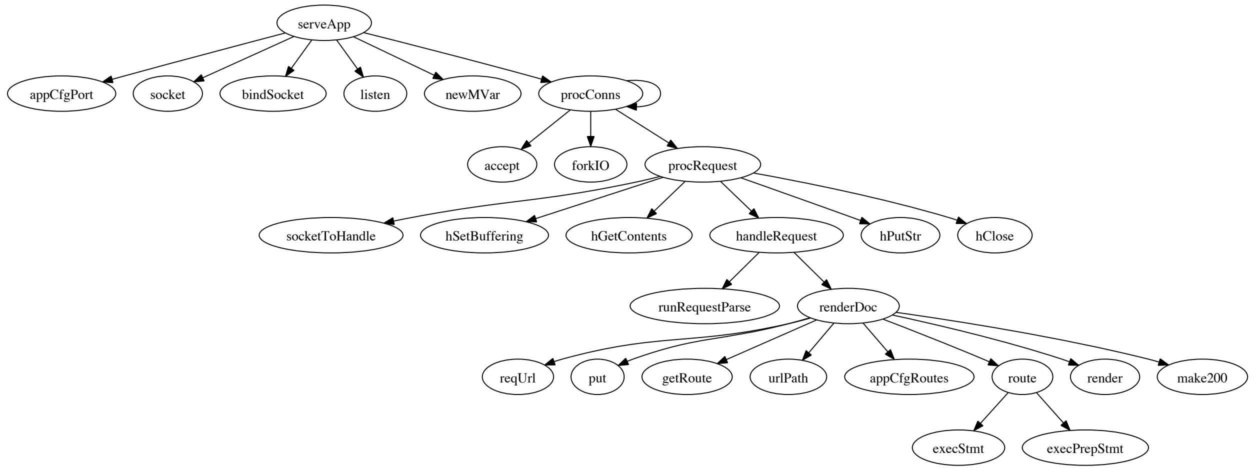


Figure 11: Simplified control flow graph for application server.

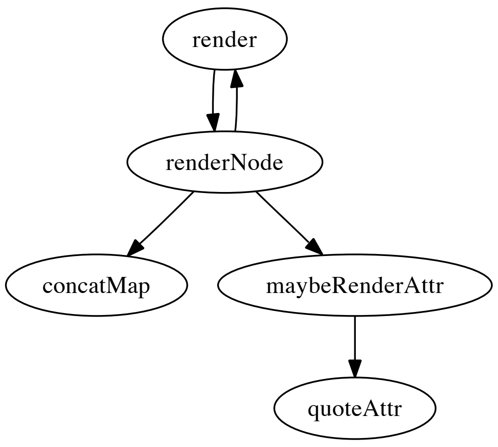


Figure 12: Example control flow graph for `Render Node` instance.

QuickCheck property testing library [9]. QuickCheck allows a developer to specify invariants in an embedded language that should hold for a given set of functions. The library then automatically generates a set of random test cases, and checks that the invariants hold for each test. In our case, we selected invariants based upon known examples of XSS [44] and SQL injection attacks [15]. In addition, we introduced modifications of the invariants that account for different popular document encodings, since these encodings directly affect how browser parsers interpret the sequences of bytes that comprise a document.

Since the coverage of the sanitization functions has been established by the control flow analysis, we focused our invariant testing on the low-level functions responsible for processing string data. In particular, we specified invariants for 7 functions that are responsible for sanitizing (X)HTML content, element attributes, and various URL components.[9] An example invariant specification is shown in Figure 13.

```
propAttrValueSafe :: AttrValue -> Bool
propAttrValueSafe input =
    (not $ elem '<' output) &&
    (not $ elem '>' output) &&
    (not $ elem '&' $ stripEntities output) &&
    (not $ elem '"' output) where
    output = render input
```

Figure 13: Simplified example sanitization function invariant specification. Here, `propAttrValueSafe` is a conjunction of predicates, where `not $ elem c output` specifies that the character `c` should not be an element of the output of `render` in this context. Since "&" is used to indicate the beginning of an HTML entity (e.g., &amp;), the `stripEntities` function ensures that ampersands may only appear in this form.

For each of the sanitization functions, we first tested the correctness of the invariants by checking that they were violated over a set of 100 strings corresponding to real-world cross-site scripting, command injection, and other code injection attacks. Then, for each sanitization function, we generated 1,000,000 test cases of random strings using the QuickCheck library. In all cases, the invariants were satisfied.

In addition to performing invariant testing on the set of document sanitization functions, we also applied a similar testing process to the sanitization of query types described in Section 4.3. Finally, we applied manual invari-

[9]The 163 functions noted above eventually apply one of these 7 context-specific sanitization functions for web documents.

ant testing on the HDBC prepared statement interface. In all cases, the invariants on the integrity of the queries and the database itself held.

5.3 Framework performance

In this experiment, we compared the performance of a web application developed using our framework to similar applications implemented using other frameworks. In particular, we developed a small e-commerce site with a product display page, cart display page, and checkout page under our framework, using the Pylons framework 0.9.7 [4], and as a Java servlet using Tomcat 6.0.18.[10] Each application was backed by a SQLite database containing product information. The application servers were hosted on a server running Ubuntu Server 8.10 with dual Intel Core 2 Duo CPUs, 2 GB of RAM, and 1000BaseT Ethernet network interfaces. The `httperf` [20] web server benchmarking tool was deployed on a similar server to generate load for each application.

Figure 14 presents averaged latency and throughput plots for 8 benchmarking runs for each framework tested. In each run, the number of concurrent clients issuing requests was varied, and the average response latency in milliseconds and the aggregate throughput in kilobytes was recorded. In this experiment, our framework performed competitively compared to Pylons and Tomcat, performing somewhat better than Pylons in both latency and throughput scaling, and vice versa for Tomcat. In particular, the latency plot shows that our framework scales significantly better with the number of clients than the Pylons framework. Unfortunately, our framework exhibited approximately a factor of two increase in latency compared to the Tomcat application. Cost-center profiling revealed that this is mainly due to the overhead of list-based `String` operations in Haskell,[11] though this could be ameliorated by rewriting the framework to prefer the lower-overhead `ByteString` type. Therefore, it is not unreasonable to assume that web applications developed using our framework would exhibit acceptable performance behavior in the real world.

5.4 Discussion

The security properties enforced by this framework are effective at guaranteeing that applications are not vulnerable to server-side XSS and SQL injection. There are limitations to this protection that need to be highlighted, however, and we discuss these here.

[10]Pylons is a Python-based framework that is similar in design to Ruby on Rails, and is used to implement a variety of well-known web applications (e.g., Reddit (`http://reddit.com/`)).

[11]Strings are represented as lists of characters in Haskell – that is, `type String = [Char]`.

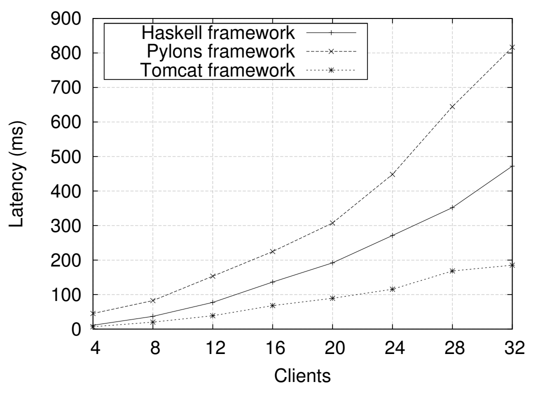
First, web applications can, in some cases, be vulnerable to *client-side* XSS injections, or DOM-based XSS, where the web application can potentially not receive any portion of such an attack [28]. This can occur when a client-side script dynamically updates the DOM after the document has been rendered by the browser with data controlled by an attacker. In general, XSS attacks stemming from the misbehavior of client-side code within the browser are not addressed by the framework in its current form.

Recently, a new type of XSS attack against the content-sniffing algorithms employed by web browsers has been demonstrated [5]. In this attack, malicious non-HTML files that nevertheless contain HTML fragments and client-side code are uploaded to a vulnerable web application. When such a file is downloaded by a victim, the content-sniffing algorithm employed by the victim's browser can potentially interpret the file as HTML, executing the client-side code contained therein, resulting in an XSS attack. Consequently, our framework implements the set of file upload filters recommended by the authors of [5] to prevent content-sniffing XSS. Since, however, the documents are supplied by users and not generated by the framework itself, the framework cannot guarantee that it is immune to such attacks.

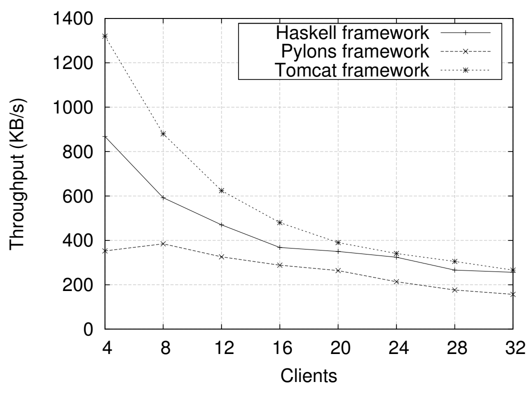
Finally, CSS stylesheets and JSON documents can also serve as vectors for XSS attacks. In principle, these documents could be specified within the framework using the same techniques applied to (X)HTML documents, along with context-specific sanitization functions. In the case of CSS stylesheets that are uploaded to a web application by users, additional sanitization functions could be applied to strip client-side code fragments. However, the framework in its current form does not address these vectors.

6 Related work

An extensive literature exists on the detection of web application vulnerabilities. One of the first tools to analyze server-side code for vulnerabilities was WebSSARI [21], which performs a taint propagation analysis of PHP in order to identify potential vulnerabilities, for which runtime guards are inserted. Nguyen-Tuong *et al*. proposed a precise taint-based approach to automatically hardening PHP scripts against security vulnerabilities in [39]. Livshits and Lam [33] applied a points-to static analysis to Java-based web applications to identify a number of security vulnerabilities in both open-source programs and the Java library itself. Jovanovic *et al*. presented Pixy, a tool that performs flow-sensitive, interprocedural, and context-sensitive data flow analysis to detect security vulnerabilities in PHP-based web applications [25]; Pixy was later enhanced with precise alias analysis to improve



(a)



(b)

Figure 14: Latency and throughput performance for the Haskell, Pylons, and Tomcat-based web applications.

the accuracy of the technique [26]. A precise, sound, and fully automated technique for detecting modifications to the structure of SQL queries was described by Wassermann and Su in [49]. Balzarotti *et al.* observed that more complex vulnerabilities in web applications can manifest themselves as interactions between distinct modules comprising the application, and proposed MiMoSA to perform multi-module vulnerability analysis of PHP applications [2]. In [7], Chong *et al.* presented SIF, a framework for developing Java servlets that enforce legal information flows specified by a policy language. A syntactic technique of string masking is proposed by Johns *et al.* in [23] in order to prevent code injection attacks in web applications. Lam *et al.* described another information flow enforcement system using PQL, and additionally propose the use of a model checker to generate test cases for identified vulnerabilities [30]. In [1], Balzarotti *et al.* applied a combination of static and dynamic analysis to check the correctness of web application sanitization functions. Wassermann and Su applied a combination of taint-based information flow and string analysis to enforce effective sanitization policies against cross-site scripting in [50]. Nadji *et al.* propose a similar notion of document structure integrity in [38], using a combination of web application code randomization and runtime tracking of untrusted data on both the server and the browser. Finally, Google's ctemplate [17], a templating language for C++, and Django [11], a Python-based web application framework, include an Auto-Escape feature that allows for context-specific sanitization of web documents, while Microsoft's LINQ [35] is an approach for performing language-integrated data set queries in the .NET framework.

The approach described in this work differs from the above server-side techniques in several respects. First, an advantage of several of the above techniques is that they provide greater generality in their enforcement of security policies; in particular, SIF allows for the enforcement of complex information flows and uses some of the techniques presented in this work. Our framework, on the other hand, requires neither information flow policy specifications or additional static or dynamic analyses to protect against cross-site scripting or SQL injection vulnerabilities. String masking embodies a similar notion of a separation of code and data for web applications, but is implemented as a preprocessor for existing web applications and allows the possibility for both false positives and false negatives. Django and ctemplate are similar in spirit to this work in that they apply a similar context-sensitive sanitization of documents generated from template specifications. In both cases, however, this sanitization is optional and relies upon a separate document parser, whereas documents in our framework are specified in the language itself. ctemplate in particular has an advantage in that it supports limited sanitization of CSS and JSON documents, though this analysis is not currently based upon a robust parser. Finally, LINQ provides a language-based mechanism for dynamically constructing parameterized queries on arbitrary data sets, including SQL databases, and is therefore similar to the system proposed in this framework. Use of this interface is, however, optional and can be bypassed.

In addition to server-side vulnerability analyses, much work has focused on client-side protection against malicious code injection. The first system to implement client-side protection was due to Kirda *et al.* In [27], the authors presented Noxes, a client-side proxy that uses manual and automatically-generated rules to prevent cross-side scripting attacks. Vogt *et al.* proposed a combination of dynamic data tainting and static analysis to prevent cross-site scripting attacks from successfully executing within a web browser [47]. BrowserShield, due to Reis *et al.*, is a system to download signatures for known cross-site scripting exploits; JavaScript wrappers that implement signature detection for these attacks are then installed into the browser [43]. Livshits and Erlingsson described an approach to cross-site scripting and RSS attacks by modifying JavaScript frameworks such as Dojo, Prototype, and AJAX.NET in [32]. BEEP, presented by Jim *et al*. in [22], implements a coarse-grained approach to client-side policy enforcement by specifying both black- and white-lists of scripts. Erlingsson *et al*. proposed Mutation-Event Transforms, a technique for enforcing finer-grained client-side security policies by intercepting JavaScript calls that would result in potentially malicious modifications to the DOM [13].

In contrast to the client-side approaches discussed here, our framework does not require a separate analysis to determine whether cross-site scripting vulnerabilities exist in a web application. In the case of web applications that include client-side scripts from untrusted third parties (e.g., mashups), a client-side system such as BEEP or Mutation-Event Transforms can be considered a complementary layer of protection to that provided by our framework.

Several works have studied how the safety of functional languages can be improved. Xu proposed the use of pre/post-annotations to implement extended static checking for Haskell in [51]; this work has been extended in the form of contracts in [52]. Li and Zdancewic demonstrated how general information flow policies could be integrated as an embedded security sublanguage in Haskell in [31]. A technique for performing data flow analysis of lazy higher-order functional programs using regular sets of trees to approximate program state is proposed by Jones and Andersen in [24]. Madhavapeddy *et al*. presented a domain-specific language for securely specifying various Internet packet protocols in [34]. In [16], Finifter *et al*. describe Joe-E, a capability-based subset of Java that allows programmers to write pure Java functions that, due to their referential transparency, admit strong analyses of desirable security properties.

The work presented in this paper differs from those above in that our framework is designed to mitigate specific vulnerabilities that are widely prevalent on the Internet. The generality of our approach could be enhanced, however, by integrating general information flow policies, at the cost of an additional burden on developers.

Several application servers for Haskell have already been developed, most notably HAppS [19]. To the best of our knowledge, however, none of these frameworks implement specific protection against cross-site scripting or SQL injection vulnerabilities. Finally, Elsman and Larsen studied how XHTML documents can be typed in ML [12]. Their focus, however, is on generating standards-conforming documents; they do not directly address security concerns.

7 Conclusions

In this paper, we have presented a framework for developing web applications that, by construction, are invulnerable to server-side cross-site scripting and SQL injection attacks. The framework accomplishes this by strongly typing both documents and database queries that are generated by a web application, thereby automatically enforcing a separation between structure and content that preserves the integrity of these objects.

We conducted an evaluation of the framework, and demonstrated that all dynamic data that is contained in a document generated by a web application must be subjected to sanitization. Similarly, we show that all SQL queries must be executed in a safe manner. We also demonstrate the correctness of the sanitization functions themselves. Finally, we give performance numbers for representative web applications developed using this framework that compare favorably to those developed in other popular environments.

In future work, we plan to investigate how the framework can be modified to allow developers to specify "safe" transformations of document structure to occur in a controlled manner. Also, we plan to investigate static techniques for verifying the correctness of the sanitization functions in terms of their agreement with invariants extracted from web browser document parsers and database query parsers, for instance using a combination of static and dynamic analyses [3, 5]. Finally, future work will consider how language-based techniques for ensuring document integrity could be applied on the client.

Acknowledgments

The authors wish to thank the anonymous reviewers for their insightful comments. We would also like to thank Adam Barth for providing feedback on an earlier version of this paper. This work has been supported by the National Science Foundation, under grants CCR-0238492, CCR-0524853, and CCR-0716095.

References

[1] D. Balzarotti, M. Cova, V. V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (S&P 2008)*, Oakland, CA, USA, May 2008.

[2] D. Balzarotti, M. Cova, V. V. Felmetsger, and G. Vigna. Multi-module Vulnerability Analysis of Web-

based Applications. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security (CCS 2007)*, Alexandria, VA, USA, October 2007.

[3] D. Balzarotti, W. Robertson, C. Kruegel, and G. Vigna. Improving Signature Testing Through Dynamic Data Flow Analysis. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC 2007)*, Miami Beach, FL, USA, December 2007.

[4] B. Bangert and J. Gardner. PylonsHQ. `http://pylonshq.com/`, February 2009.

[5] A. Barth, J. Caballero, and D. Song. Secure Content Sniffing for Web Browsers, or How to Stop Papers from Reviewing Themselves. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2009. IEEE Computer Society.

[6] Breach Security, Inc. Breach WebDefend. `http://www.breach.com/products/webdefend.html`, January 2009.

[7] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *Proceedings of the 2007 USENIX Security Symposium*, Boston, MA, USA, 2007. USENIX Association.

[8] Citrix Systems, Inc. Citrix Application Firewall. `http://www.citrix.com/English/PS2/products/product.asp?contentID=25636`, January 2009.

[9] K. Claessen and J. Hughes. Testing Monadic Code with QuickCheck. *ACM SIGPLAN Notices*, 37(12):47–59, 2002.

[10] M. de Kunder. The Size of the World Wide Web. http://www.worldwidewebsize.com/, May 2008.

[11] Django Software Foundation. Django Web Application Framework. `http://www.djangoproject.com/`, June 2009.

[12] M. Elsman and K. F. Larsen. Typing XHTML Web Applications in ML. In *Proceedings of the* 6th *International Symposium on Practical Aspects of Declarative Languages*, pages 224–238. Springer-Verlag, 2004.

[13] U. Erlingsson, B. Livshits, and Y. Xie. End-to-end Web Application Security. In *Proceedings of the* 11th *USENIX Workshop on Hot Topics in Operating Systems*, San Diego, CA, USA, 2007. USENIX Association.

[14] F5 Networks, Inc. BIG-IP Application Security Manager. `http://www.f5.com/products/big-ip/product-modules/application-security-manager.html`, January 2009.

[15] Ferruh Mavituna. SQL Injection Cheat Sheet. `http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/`, June 2009.

[16] M. Finifter, A. Mettler, N. Sastry, and D. Wagner. Verifiable Functional Purity in Java. In *Proceedings of the* 15th *ACM Conference on Computer and Communications Security*, pages 161–174, Alexandria, VA, USA, October 2008. ACM.

[17] Google, Inc. ctemplate. `http://code.google.com/p/google-ctemplate/`, June 2009.

[18] D. H. Hansson. Ruby on Rails. `http://rubyonrails.org/`, February 2009.

[19] HAppS LLC. HAppS – The Haskell Application Server. `http://happs.org/`, February 2009.

[20] Hewlett Packard Development Company, L.P. httperf. `http://www.hpl.hp.com/research/linux/httperf/`, February 2009.

[21] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. Lee, and S.-Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the* 13th *International Conference on the World Wide Web*, pages 40–52, New York, NY, USA, 2004. ACM.

[22] T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-Enforced Emdedded Policies. In *Proceedings of the* 16th *International Conference on the World Wide Web*, Banff, Alberta, Canada, May 2007. ACM.

[23] M. Johns and C. Beyerlein. SMask: Preventing Injection Attacks in Web Applications by Approximating Automatic Data/Code Separation. In *Proceedings of ACM Symposium on Applied Computing*, Seoul, Korea, March 2007. ACM.

[24] N. D. Jones and N. Andersen. Flow analysis of lazy higher-order functional programs. *Theoretical Computer Science*, 375(1–3):120–136, 2007.

[25] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P 2006)*, pages 258–263, Oakland, CA, USA, May 2006. IEEE Computer Society.

[26] N. Jovanovic, C. Kruegel, and E. Kirda. Precise Alias Analysis for Static Detection of Web Application Vulnerabilities. In *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, pages 27–36, Ottawa, Ontario, Canada, 2006. ACM.

[27] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A Client-side Solution for Mitigating Cross-Site Scripting Attacks. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC 2006)*, Dijon, France, April 2006. ACM.

[28] A. Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. `http://www.webappsec.org/projects/articles/071105.shtml`, July 2005.

[29] C. Kruegel, W. Robertson, and G. Vigna. A Multi-model Approach to the Detection of Web-based Attacks. *Journal of Computer Networks*, 48(5):717–738, July 2005.

[30] M. S. Lam, M. Martin, B. Livshits, and J. Whaley. Securing Web Applications with Static and Dynamic Information Flow Tracking. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 3–12, San Francisco, CA, USA, 2008. ACM.

[31] P. Li and S. Zdancewic. Encoding Information Flow in Haskell. In *Proceedings of the* 19th *IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 2006.

[32] B. Livshits and U. Erlingsson. Using Web Application Construction Frameworks to Protect Against Code Injection Attacks. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, pages 95–104, San Diego, CA, USA, 2007. ACM.

[33] B. Livshits and M. Lam. Finding Security Errors in Java Programs with Static Analysis. In *Proceedings of the* 14th *USENIX Security Symposium (USENIX Security 2005)*, pages 271–286. USENIX Association, August 2005.

[34] A. Madhavapeddy, A. Ho, T. Deegan, D. Scott, and R. Sohan. Melange: Creating a "Functional Internet". In *Proceedings of the* 2nd *ACM European Conference on Computer Systems*, pages 101–114, Lisbon, Portugal, April 2007. ACM.

[35] Microsoft, Inc. LINQ. `http://msdn.microsoft.com/en-us/netframework/aa904594.aspx`, June 2009.

[36] Miniwatts Marketing Group. World Internet Usage Statistics. http://www.internetworldstats.com/stats.htm, May 2008.

[37] E. Moggi. Notions of Computation and Monads. *Information and Computation*, 93(1):55–92, 1991.

[38] Y. Nadji, P. Saxena, and D. Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Proceedings of the Network and Distributed System Security Symposium*, February 2009.

[39] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shifley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the 2005 International Information Security Conference*, pages 372–382, 2005.

[40] Ofer Shezaf and Jeremiah Grossman and Robert Auger. Web Hacking Incidents Database. `http://www.xiom.com/whid-about`, January 2009.

[41] Open Security Foundation. DLDOS: Data Loss Database – Open Source. `http://datalossdb.org/`, January 2009.

[42] Open Web Application Security Project (OWASP). OWASP Top 10 2007. `http://www.owasp.org/index.php/Top_10_2007`, February 2009.

[43] C. Reis, J. Dunagan, H. J. Wang, and O. Dubrovsky. BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. *ACM Transactions on the Web*, 1(3):11, 2007.

[44] Robert Hansen (RSnake). XSS (Cross Site Scripting) Cheat Sheet. `http://ha.ckers.org/xss.html`, June 2009.

[45] W. Robertson, G. Vigna, C. Kruegel, and R. A. Kemmerer. Using Generalization and Characterization Techniques in the Anomaly-based Detection of Web Attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2006)*, San Diego, CA, USA, February 2006.

[46] Symantec, Inc. Symantec Report on the Underground Economy – July 07 – June 08. `http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_underground_economy_report_11-2008-14525717.en-us.pdf`, November 2008.

[47] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross Site Scripting

Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2007)*, February 2007.

[48] P. Wadler. The Essence of Functional Programming. In *Proceedings of the* 19th *Annual Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, NM, USA, 1992. ACM.

[49] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. *ACM SIGPLAN Notices*, 42(6):32–41, April 2007.

[50] G. Wassermann and Z. Su. Static Detection of Cross-Site Scripting Vulnerabilities. In *Proceedings of the 2008 International Conference on Software Engineering (ICSE 2008)*, pages 171–180, Leipzig, Germany, 2008. ACM.

[51] D. N. Xu. Extended Static Checking for Haskell. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, pages 48–59, Portland, OR, USA, 2006. ACM.

[52] D. N. Xu, S. P. Jones, and K. Claessen. Static Contract Checking for Haskell. In *Proceedings of the* 36th *Annual ACM Symposium on the Principles of Programming Languages*, pages 41–52, Savannah, GA, USA, 2009. ACM.