Fine-Grained Privilege Separation for Web Applications

Akshay Krishnamurthy Adrian Mettler David Wagner
Department of Electrical Engineering and Computer Science
University of California, Berkeley, USA
akshayk@berkeley.edu, {amettler, daw}@cs.berkeley.edu

ABSTRACT

We present a programming model for building web applications with security properties that can be confidently verified during a security review. In our model, applications are divided into isolated, privilege-separated components, enabling rich security policies to be enforced in a way that can be checked by reviewers. In our model, the web framework enforces privilege separation and isolation of web applications by requiring the use of an object-capability language and providing interfaces that expose limited, explicitly-specified privileges to application components. This approach restricts what each component of the application can do and quarantines buggy or compromised code. It also provides a way to more safely integrate third-party, less-trusted code into a web application. We have implemented a prototype of this model based upon the Java Servlet framework and used it to build a webmail application. Our experience with this example suggests that the approach is viable and helpful at establishing reviewable application-specific security properties.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Security, Design

Keywords

Web applications, privilege separation, object-capabilities

1. INTRODUCTION

Today, web applications are vulnerable to a variety of attacks that can compromise private data, bring down essential systems, or otherwise wreak havoc on our lives. One recent study reports that web-based attacks are now the "primary vector for malicious activity over the internet" and that 63% of reported attacks targeted web applications in 2008 [20]. With current tools, developing a secure web application is challenging, and enabling others to verify its security is even harder. In this paper, we address these two concerns with a novel web programming model that partitions an application into privilege-separated components and minimizes the privileges assigned to each one. This architecture enables a code reviewer to verify specific security properties by examining relevant application components, rather than auditing the application as a whole. At runtime, we isolate code on a per-user basis by default, avoiding accidental communication between users and making it easier to identify and review the correct operation of intended communication channels between users.

Web applications are increasingly being developed using frameworks that abstract and automate low-level implementation details. We believe that such frameworks should provide better support for security. In addition to providing automated protection against traditional web attacker threats such as cross-site scripting and SQL injection, they should support secure compartmentalization of applications. The framework is an ideal place to handle security concerns as it allows a single well-reviewed implementation to be used for a large number of applications. Also, handling security features in the framework reduces the opportunity for application programmers to introduce security vulnerabilities and reduces the need for them to be security experts.

We present a prototype framework, Capsules, that provides isolation between reduced-privilege application components via the strong security guarantees of an object-capability language. In our framework, each application component can be given a minimal set of privileges, bounding its behavior and limiting the damage from bugs in that component. Reducing trust in application code can simplify the security review process: when reviewing security properties, we can easily identify which components must be examined simply by understanding the privileges granted to them. For instance, to check that a security invariant is preserved, we need to review only those components that have the privileges that would be needed to violate that invariant. Isolation and limited privilege also provide a way to securely support third-party components and plugins, which are becoming a popular way to extend application functionality.

In addition, we leverage existing techniques to help the web application developer defend against attacks from malicious clients. Although our focus is not on lower-level vulnerabilities such as cross-site scripting, cross-site request forgery, and SQL injection, we incorporate known defenses for these vulnerabilities in our application framework. These problems are well-understood, and solutions for addressing them have been and continue to be proposed and deployed.

Capsules is built atop the Java Servlet framework and exposes a Servlet-like interface to the web application. A web application is composed of a number of servlets, and each (session, servlet) pair defines a protection domain that can only communicate with other protection domains in explicit ways. Each user session is associated with a session object, and only portions of this object are exposed to each servlet.

Copyright is held by the International World Wide Web Conference Committee (IW3C2). Distribution of these papers is limited to classroom use, and personal use by others.
WWW 2010, April 26–30, 2010, Raleigh, North Carolina, USA.
ACM 978-1-60558-799-8/10/04.

Capsules applications are largely written in Joe-E, an object-capability language in which all privileges are represented as object references, or capabilities [11]. In Joe-E, at each point in time, the program only has the privilege provided by the objects currently in scope. This allows us to follow the principle of least privilege by limiting the capabilities granted to each protection domain, thereby reducing the trust placed in it. In particular, in our model, one can ensure that the application code processing each request only receives the capability to access data associated with the currently logged-in user, so any security breach associated with that request can affect only a single user. In contrast, conventional web application architectures do not restrict a servlet's access to user data, which is more fragile because a single security compromise can affect every user of the web application. Joe-E also helps us to enforce the security boundary between the web application and the framework. Using Joe-E, we know that servlets will only be able to interact with the outside world through the objects exported by our servlet framework. These interfaces can be thoroughly reviewed once to ensure that the framework will maintain its integrity in the face of compromised or even malicious servlets.

In order to better evaluate our model and prototype framework, we built a simple web mail application. We found that our model makes it possible to verify important application-level security properties, including privacy and integrity of user data and isolation between users. We also perform experiments to assess the performance impact of our model for web application programming.

The rest of this paper is organized as follows. Section 2 identifies the security and usability goals of our system. In Section 3, we present a high-level overview of our approach. Section 4 describes the Capsules framework and its implementation. In Section 5, we evaluate how Capsules aids in the security review process, and we discuss its performance and practical deployability. We survey related work in Section 6 and conclude in Section 7.

2. GOALS

We want to enable programmers to build real-life web applications with high-level security properties that can be verified by a security review. To achieve this, our approach must make it feasible to conclude with confidence that application security goals are met while providing a model in which programmers can efficiently build real applications.

2.1 Security Goals

Our primary goal is to improve the security of web applications. Specifically, we wish to:

- **Support the principle of least privilege.** It should be easy to grant application components access to only the resources that they need, minimizing the potential damage they can do. Such reduction in privilege should meaningfully reduce trust in application components, so that irrelevant components can be ignored when reviewing security properties. It should be possible to restrict privileges to the extent that less-trusted components can be safely subjected to less thorough security review than more-trusted components.

- **Provide isolation.** We want to provide two kinds of isolation. First, our model should enforce *session isolation*: the code processing each request should have access only to data relating to that request and to the associated session, but not to data associated with other sessions or other requests. Developers can use session isolation to enforce proper isolation between users by ensuring that the session object only contains information associated with the currently logged-in user. Therefore, session isolation provides a way to restrict how a user's actions can affect other users. Second, our model should provide *component isolation*: application components should not be able to tamper with each other. Each component should be able to maintain the integrity of its own code and private state. One component should not be able to escalate its privilege by invoking functionality provided by another component. This form of isolation ensures that the capabilities provided to a component are not used inappropriately by others.

- **Facilitate security reviews.** It is not enough for a developer to be able to achieve desired security properties. It should also be possible for a reviewer to gain confidence that the claimed security properties are actually maintained. Our proposed model should make it easy to reason about security properties of an application and should facilitate such a security audit. In order to support the security properties that are important for a given application, our model must allow verification not just of common predefined security properties, but also of higher-level policies that vary between applications. Our model should be general enough to support the security policies of a diverse range of web applications.

2.2 Usability Goals

For our programming model to be adopted, it must be easy to learn and use. In particular, we have the following goals:

- **Familiarity and simplicity.** It is important for a new framework to be easy for existing web developers to learn. One way to achieve this is to base it on an existing framework in popular use. Changes to the existing framework should be the minimum required to achieve the desired security and reviewability benefits, and they should be straightforward and easy for programmers to understand. This allows existing developer expertise to be leveraged to build secure applications.

- **Maintain developer productivity.** A possible hazard of adding mechanisms to improve security during application development is that programmers can be overwhelmed by added complexity, dissuading adoption. To the extent possible, security enhancements should be transparent and automatic. When security concerns are abstracted away in this manner, developers no longer have to worry about them. Abstracting security features makes it less likely that developers will introduce security vulnerabilities by misuse or disuse of the feature. Where providing safe default behavior by abstraction is not possible, any security support should be explicit but naturally integrated into the language or library. "Bolted-on" security enforcement mechanisms can lead to unintuitive behavior such as unexpected errors or mysterious failures, as well as an increased risk of unsoundness due to mismatch between the framework and the security code.

- **Ease development of new applications.** Our primary focus is on new web applications; we want to ensure that developers do not have a hard time understanding the model and can easily develop applications using this model. Porting old applications is a lower priority.

3. APPROACH

We use a language-based approach to enforce isolation between web application components and grant each one a minimal set of privileges appropriate to its functionality. In our framework, application code is written in an *object-capability language* in order

to reliably achieve these properties. We prevent application components from interfering with each other and restrict the privileges granted to each one in accordance with the principle of least privilege [18]. These two properties serve to prevent vulnerabilities in a web application and limit the consequences of exploiting remaining ones.

3.1 Object-Capabilities

Object-capability languages are designed to enforce isolation and facilitate practicing the principle of least privilege at the granularity of individual objects. In such a language, capabilities (unforgeable tokens granting privileges) are implemented as object references in an object-oriented programming language.

Objects in these languages are used to store information and represent external resources. A capability is a reference to an object; code that has such a capability is thereby authorized to call the object's methods and access its public instance variables. It is not possible to perform any operations on an object or to read or write its state without a reference to the object.

In object-capability languages, all privileges to read sensitive information or to affect the state of the program or the outside world are carried by capabilities. Code that is not granted an explicit capability cannot communicate with the rest of the program (except to return a result when called) or the outside world. Libraries are designed to ensure that they do not expose any privileges to the program except when the caller presents an appropriate capability. Typically, each system resource is represented by an object provided by the library, and programs interact with that resource by invoking methods on the corresponding object.

Any real application will need to make use of application-specific resources such as databases or files. In a capability system where application components cannot instantiate these resources directly, there must be some component trusted to create the initial capabilities to these resources. In our approach, we propose that the framework should invoke some small body of trusted application-specific code in order to construct these capabilities, which can then be used by various application components as needed.

Object-capability languages are helpful from a security perspective for several reasons. They allow a developer to manage which privileges are made available to different parts of a system in such a way that a reviewer can determine this distribution while limiting the amount of code that needs to be examined. Thus, the object-capability model helps achieve least privilege in a easily-reviewable way. They also make it easier to reason about security invariants: the subset of the program that must be reviewed to verify an invariant can be minimized by limiting the distribution of capabilities that could be used to violate the invariant.

3.2 Design Overview

Object-capability languages can reliably isolate objects from each other, and we use this ability to separate web applications into components which can only interact in proscribed ways, achieving our component isolation property. Each application component forms a protection domain. The extent of a protection domain is specified by the set of capabilities assigned to that component, and by considering which capabilities are shared between domains, we can understand how components can communicate with each other. If two components have disjoint sets of capabilities, we immediately know that they are completely isolated.

In our approach, all application state, including capabilities to application-specific resources, is stored in a per-session data store. This architecture facilitates session isolation. The only way state can be shared between sessions is if shared capabilities are added to the session by an application's trusted session initialization code. This trusted code is a small fraction of the application that must be carefully reviewed. Each application component has a declarative policy governing its access to the per-session data store. This permits review of communication channels between components and limits how they can interfere with each other. Properly-defined access policies can achieve least privilege and thus facilitate a security review.

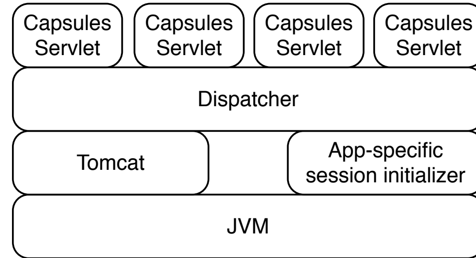


Figure 1: Overall architecture of our implementation. The dispatcher exposes a modified Servlet API to the application-level servlet instances. The dispatcher is part of the Capsules framework; the application developer writes the servlets and session initializer code. Typically, servlets are written in Joe-E, and the trusted session initializer is written in Java.

In many web applications, we expect that each session will be associated with a single logged-in user of the application (or, if the client has not logged in yet, with no user at all). For these applications, session isolation contributes to providing verifiable *user isolation*.

We make use of known techniques to automatically guard against client-side web attacks such as cross-site scripting and cross-site request forgery, as well as analogous attacks between application components. These defense mechanisms are implemented in the framework and are fully automatic, so that developers can focus on application functionality and higher-level security properties.

4. IMPLEMENTATION

We implemented a prototype web framework, called Capsules, atop the Java Servlet framework. Capsules introduces an additional layer, the *dispatcher*, between the servlet container and Capsules applications; see Figure 1. The dispatcher exposes a modified Servlet API to the application and controls communication between the user and the application, allowing it to provide additional security features not offered by traditional servlet containers.

4.1 Joe-E

Capsules applications are written primarily in Joe-E, an object-capability language designed as a subset of the Java language [11]. The Joe-E verifier is used to statically check that Joe-E code conforms to the Joe-E language subset; then, the Joe-E code is compiled with a standard Java compiler. Joe-E code is executed on an unmodified JVM, like ordinary Java programs, requiring only the addition of a Joe-E support library. Joe-E code can be combined with unrestricted Java code, but since Java code is fully trusted and has the ability to violate the security properties enforced by Joe-E, any code written in Java should be reviewed and verified not to interfere with the Joe-E code's security properties.

Joe-E removes a number of features from Java that preclude secure isolation, such as unsafe reflection and mutable static fields. It

additionally defines a subset of the Java library, excluding methods and fields that convey privilege without an appropriate capability. This subsetting is necessary to satisfy the principle of least privilege, as otherwise all code in the system would be granted inappropriate privilege in excess of its need. In some cases where the Java APIs cannot be made safe by subsetting alone, Joe-E provides additional library methods that expose the underlying functionality in a capability-safe way. One aspect of Capsules is the definition of a safe subset of the Servlet API and a capability-safe implementation of this subset.

4.2 The Servlet API

The Java Servlet framework is a standard platform for writing Java web applications. An application in this framework is hosted by a servlet container and consists of a set of servlet classes that process incoming requests from users. When the container receives a request, it forwards the request to the appropriate servlet, typically by calling its `doGet` or `doPost` method. The servlet generates a response and the container sends this back to the user. An interaction with an application consists of one or more requests to its servlets and the associated responses.

When the servlet container is started, it installs an application by constructing an instance of each servlet defined in a configuration file. Throughout the lifetime of the application, these singleton servlets are responsible for handling user requests. When invoking a servlet, the web server gives the servlet two objects, representing the HTTP request and response. The servlet is allowed to read any information from the request and to populate the response with data to be sent back to the user. Bundled with the HTTP request is an `HTTPSession` object, associated with the current user session, as identified by a session cookie. The session object provides a session-specific mapping from strings to arbitrary objects. Every servlet can access the session object and the entire mapping associated with it.

4.3 Capsules API

The Capsules framework presents a modified version of the Servlet API to its applications. The Capsules API is a subset of the standard Servlet API along with replacement functionality mediated by a dispatcher component. The dispatcher component handles all communication between Capsules applications and the rest of the Capsules framework.

Our implementation builds on top of an existing servlet container. The dispatcher interposes between the application-level servlets and the underlying servlet container. In particular, the dispatcher is a servlet registered with the underlying servlet container; the underlying servlet container is unaware of the application-level servlets and delivers all incoming requests to the dispatcher. We maintain our own URI-to-servlet mapping in the dispatcher, so that we can appropriately forward requests to application-level servlets. This mapping is specified declaratively in an application configuration file in the same format used by the servlet container and loaded into the dispatcher upon initialization.

Capsules applications, while written mostly in Joe-E, include a small Java component to initialize capabilities to various resources such as the filesystem or a database. In our implementation, these resources are placed in every session upon initialization.

4.3.1 Isolation

In the Servlet framework, a single instance of each servlet is created and shared across all sessions. The instance variables of servlets can be used to store servlet-local state that is shared across sessions (though this is not a recommended practice), and application developers may not realize that these shared variables can cause concurrency bugs or leak sensitive information between users. This violates our session isolation goal. To eliminate this communication channel, we require that servlets contain no mutable state, which we enforce by declaring the `JoeEServlet` class to implement the `Immutable` marker interface. This interface is defined by the Joe-E library and causes the Joe-E verifier to check that all of the class's fields are final and are declared with a (transitively) immutable type.

This restriction ensures that all application state is maintained in `HTTPSession` objects. The dispatcher makes application state available to servlets when it calls the `doGet` and `doPost` methods, by passing a reference to the session object as a parameter. Since servlets cannot maintain state on their own, any objects associated with users must be reachable from that user's `HTTPSession` object. Thus, we can achieve user isolation by ensuring that every object in the `HTTPSession` contains only data that should be accessible to the current user.

4.3.2 Restricted Views

The standard servlet model makes the entire session object and all cookies received available to every servlet. In this model, it is difficult to pinpoint where session members and cookies are used, and it is challenging to fully understand the consequences of modifying or misusing these objects. This lack of documentation complicates security reviews. Capsules provides restricted access to the HTTP session object and cookies in order to reduce privileges granted to components and facilitate review.

Our dispatcher exposes a servlet-specific view of the HTTP session. Much like a database view, this provides only a portion of the session state to the servlet. We augment the application configuration file to list which session members—i.e., which entries in the mapping from strings to objects maintained by the session object—can be read and written by each servlet. From this specification, we automatically generate code to define a wrapper class, called a `SessionView`, that provides a restricted interface to the underlying `HTTPSession` object. There is one `SessionView` class per servlet, and it restricts that servlet's access to a subset of the session state. A `SessionView` object consists of a reference to the `HTTPSession` and getter and setter methods for accessing specific session members. `SessionView`s are constructed on each request and passed to Capsules servlets as a parameter to their `doGet` and `doPost` methods.

Each servlet also defines a `CookieView` class specifying which cookies it can read and write. Like the `SessionView` objects, `CookieView` objects are constructed on each request and passed to the servlet as a method argument. In addition to the set of cookies included in the HTTP request, `CookieView`s include a set of cookie updates to be sent as part of the HTTP response.

Defining session and cookie views as wrappers around the underlying session and cookie objects lets programmers define fine-grained access policies. For example, we can enforce read-only or write-only access to the underlying objects simply by removing a getter or setter method. From the programmer's perspective, access policies are specified declaratively, and a code generation tool automatically generates code implementing the session and cookie views. While session and cookie view classes may contain many getter and setter methods, these methods do not need to be verified for correctness because the view classes are automatically generated. Instead, it suffices to review the code generation tool. If finer-grained access to a session member is desired, an application developer can define a wrapper class that mediates access to that object and enforces a custom access policy on all accesses, and

then expose only the wrapper (not the object it wraps) to servlets by placing the wrapper object in the session instead of the wrapped object.

We make use of Joe-E's taming functionality, which exposes only a subset of the Java libraries to Joe-E code, to ensure that application servlets can only access session data and cookies through the view objects. In particular, we prevent Joe-E code from calling the `getHTTPSession` and `getCookies` methods on the HTTP request object and the `addCookie` method on the HTTP response object.

Restricted views support the principle of least privilege, as they permit programmers to grant each servlet with only the capabilities it needs. These views also facilitate security audits, as reviewers can quickly understand what capabilities are granted to each servlet. Additionally, they help enforce servlet isolation, because servlets can only communicate using channels explicitly defined as session members and two servlets with disjoint views have almost no way of interacting with each other[1].

Nothing in our framework currently prevents one servlet from directly invoking a method on another servlet, but it will be unable to escalate its privilege by doing so. To invoke another servlet with its normal privileges requires constructing the same `SessionView` as it would receive from the dispatcher, but this is not possible because doing so would require direct access to the `HTTPSession` and servlets are not given direct access to the underlying `HTTPSession`.

A malicious servlet could invoke another servlet with a `SessionView` constructed from an `HTTPSession` it generates itself, but such a `HTTPSession` can only contain capabilities the malicious servlet already had access to. In most cases, this means that the attacking servlet could have replicated the results of any such attack by duplicating and calling a local copy of the code of the victim servlet. The exception is if the victim servlet is able to construct an object of a type that the attacking servlet cannot, such as a private inner class. If the type of this class is used elsewhere as a trust marker, and the victim class assumes that it has a trusted path to the user, then this attack could violate the trust assumptions placed on the type. We expect this pattern to be rare, but we acknowledge that it is unfortunate to require developers using this type of type-based reasoning to be aware of this hazard in our current implementation.

4.3.3 Automated Defenses

In addition to isolation and restricted views, Capsules provides defenses against certain common kinds of attacks. Since the dispatcher mediates all communication between the user and the application in both directions, it can transparently transform requests and responses to defend the application against certain attacks. In our implementation, we prevent cross-site scripting attacks, make it possible to review that application JavaScript does not violate servlet isolation, and guard against cross-site and cross-servlet request forgery (CSRF).

We implement these mechanisms by restricting how application servlets can write to the HTTP response. In our current implementation, servlets can only construct a response page via a restricted DOM API, rather than specifying arbitrary strings to a `PrintWriter` as in the standard Servlet API. We check that the constructed DOM contains no dynamically-constructed active content; our API prohibits adding `script` and `embed` tags to the document. As these restrictions prevent any dynamically-constructed JavaScript, they suffice to protect the page from cross-site scripting. We also prohibit inline CSS, as some browsers allow script to be defined in CSS. While it is much easier to make a DOM API safe from active content injection, we acknowledge that applications written using a DOM API are more verbose than those written using unsafe string concatenation or a templating system, and thus our system may be less convenient for developers to use. Additionally, DOM construction prevents streaming of the document as it is generated, substantially increasing perceived latency for large pages. A useful addition to our system would be a safe templating language which permits streaming, similar to Genshi [3].

While these restrictions prevent dynamically constructed JavaScript, we allow developers to specify (in the application's configuration file) a static JavaScript header file and a CSS file for each servlet. Once a servlet returns control to the dispatcher with its DOM response, the dispatcher augments the DOM to link in the appropriate files. The included script can then modify the DOM tree to add any desired action handlers to the document, guided by `id` tags or CSS styles. These JavaScript and CSS files must be manually audited to ensure that the JavaScript does not access pages belonging to other servlets in violation of the application's desired security properties. A better approach would be to create an appropriate JavaScript validator or rewriter similar to AdSafe [2] or Caja [13] to automate this process. This tool would verify that the active content on the page adheres to a "same-servlet policy" and does not access other servlets in a manner that would be prevented by the browser's same-origin policy if they resided at a different domain. However, we have not implemented this extension.

In our framework, we must guard against not only standard cross-site request forgery, but also cross-servlet request forgery, as we must protect each application component individually and provide servlet isolation. The risk is that one servlet could cause requests to be performed to another servlet, effectively escalating its privilege. We extend the well-known CSRF defense of one-time keys, inserted into forms as a hidden field, by associating a distinct securely-random key with each servlet at session construction. We insert these keys into forms by having our DOM API implementation automatically add a hidden input whenever the servlet creates a form object. When the dispatcher receives a `POST` request, it compares the value of the associated HTTP parameter with the one-time key stored in the `HTTPSession` before forwarding the request to the application servlet. At present, it is up to the developer to ensure that `GET` requests do not have undesired side effects, but this task is simplified in comparison with other frameworks as most servlets' `SessionView` policies will restrict what portion of the session state each servlet can modify.

5. EVALUATION

To evaluate the effectiveness of our programming model for building secure web applications, we built a webmail application as a case study. While our application is simple, we are able to demonstrate high-level security properties. Our application is implemented as a collection of servlets, where each servlet corresponds to a specific application feature. We wrote 8 servlets for creating user accounts, authenticating and logging out of the application, and for reading, writing, and deleting emails. Each of these servlets defines a `SessionView` and a `CookieView` that we use to restrict the capabilities provided to that component. Users' mail is stored in the file system, in the form of a mailbox directory for each user following the Maildir specification. A top-level `users` directory contains all the user directories. We use Postfix to accept incoming email on port 25 and deliver it to the user's mailbox directory.

[1] We do not prevent communication over covert channels. However, covert channels can only be used to transfer information, i.e., bits, not capabilities.

5.1 Security Analysis

We identify two critical application-specific security properties that we want our webmail service to achieve:

1. **Integrity.** If Alice is a user of the webmail service, an attacker who does not know Alice's password is unable to use the webmail service to affect (directly or indirectly) the contents of Alice's mailbox, except by using the webmail service's defined interface to send Alice an email.

2. **Privacy.** If Alice is a user of the webmail service, an attacker who does not know Alice's password is unable to use the webmail service to gain any new information on the content of messages in Alice's mailbox through any overt channel.

We assume that the attacker controls a malicious web client. We allow for the possibility that the attacker might be colluding with the programmers who wrote the webmail servlets (so we do not exclude malicious code from our threat model). However, we assume that the system administrator and platform code (e.g., Tomcat, Postfix) are trusted and not malicious. The privacy property makes no promises about information that might be leaked through covert channels.

To evaluate how effectively Capsules facilitates security reviews, we conducted a security review of our webmail application to verify the two critical security properties listed above. Unlike many security reviews, which often consist of a best-effort search for bugs starting with the most likely places, we instead aimed to convince ourselves that our two critical security properties hold. We constructed an explicit argument that each property holds and then checked that the code satisfies each of the assumptions made by that argument. Due to the isolation and privilege separation properties of the framework, it sufficed to manually review only a portion of the code in order to check each property: we were able to identify a subset of the code that was critical to enforcing each property and then informally reason about these subset.

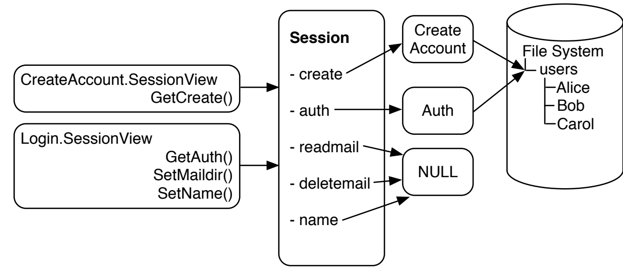
5.1.1 Verifying Integrity

To convince ourselves that our webmail application achieves the integrity property above, we first identified all application components that have the ability to directly modify the contents of Alice's mailbox by finding all components that can obtain a capability to any file in Alice's mail directory. We found that only the session initialization module and `DeleteServlet` can acquire a capability to modify any such file, the latter only when it executes within Alice's session. We also verify that each session is associated with at most one user of the webmail service and that the authentication logic prevents logging into a session as Alice without knowledge of Alice's password. Therefore, no one else can gain control of a session belonging to Alice and directly violate her integrity.

Next, we check that the attacker cannot indirectly affect the contents of Alice's mailbox. The only way this could happen is if some other session were able to influence the execution of code running in Alice's session, causing that code to modify Alice's mailbox in a way she did not request. Here our basic strategy is to show that the set of heap objects reachable from the attacker's session is disjoint from the set of heap objects reachable from Alice's sessions and then argue that this prevents an attacker from influencing the behavior of code running on Alice's behalf.

Session Initialization. We reviewed the application's session initialization code and confirmed that it doesn't use unsafe Java features to violate the isolation properties that Joe-E guarantees for the application code. This enables us to soundly reason about the propagation of capabilities.

(a): an unauthenticated session:



(b): an authenticated session:

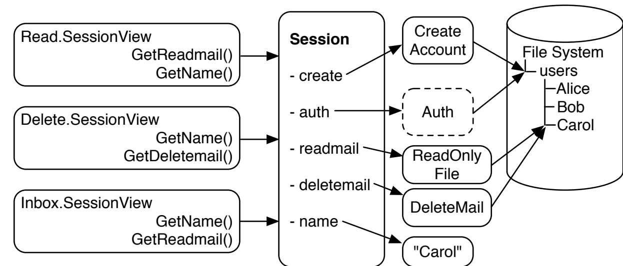


Figure 2: Our authentication scheme. A fresh session is preloaded with capabilities that can be used to create a new user account or authenticate a user. However, once a user is authenticated, the `Auth` capability is invalidated.

The session initialization module is the only application code that can create file capabilities from scratch. We verified that it only uses this power to construct an `Auth` capability, giving it a reference to the `users` directory. The `Auth` capability is then stored in the session object.

Authentication. When a user successfully authenticates and logs into the webmail application, the application populates her session object (that of the session in which the successful authentication occurred) with capabilities to her mailbox files; before then, the session object does not contain any mailbox file capabilities. See Figure 2, which shows the application state before and after successfully authenticating a user.

In our framework, the only way capabilities can become available to application code processing a request is if they are stored in the session. Since application code cannot construct new capabilities to the outside world, all external capabilities must derive from the ones initially placed in the session by the session initializer. Therefore, the only way for code to access Alice's mailbox is if its session's initial capabilities gave access to her directory. Due to the immutability of session objects and Joe-E's prohibition on mutable static fields, there is no way to "smuggle" Alice's data or a capability to her mailbox from a previous session.

In our application, there are several capabilities initially stored in the session, but only two can be used to access the `users` directory and thus potentially read or modify Alice's mail: namely, a reference to the `Auth` class, which authenticates a user and retrieves her mailbox, and a reference to the `CreateAccount` class, which can be used to add new users to the system.

In reviewing the `CreateAccount` class, we confirmed it can only create and initialize an account that does not already exist at creation time. It appropriately restricts the user name via a whitelist of valid characters, chosen to avoid creation of multiple names that

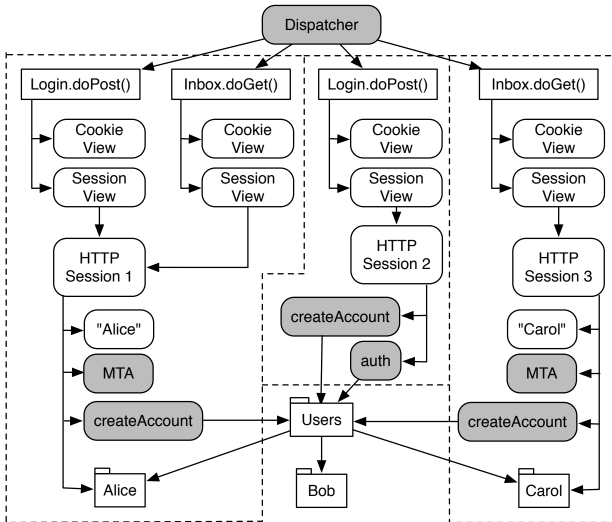


Figure 3: A graph of the heap of our running application. Isolation can be verified by examining a limited number of trusted components (shaded) that bridge isolation domains.

the Postfix mail transfer agent will map to the same user mailbox directory. It then invokes Postfix to create and initialize the new mailbox directory and updates the Postfix configuration to recognize this username. As a `CreateAccount` object cannot be used to gain access to an existing mailbox, we do not need to review it further to establish the integrity property.

The `Auth` class has a single method which accepts a username and password and returns the user's mailbox directory. We reviewed the implementation of this method and verified that it requires a correct password before returning a capability to the user's mailbox files. Before returning, the `Auth` capability invalidates itself by clearing a boolean flag. This operation is atomic because our framework serializes all requests per session by default. This means that each `Auth` object can only be used for at most one successful authentication. As the session initialization code adds only a single `Auth` object to the session, there is no way for that session to gain a reference to any other `Auth` object. The `Auth` object encapsulates a capability for the user's mailbox files and releases this capability only upon successful authentication. There is no other way to gain a capability to a user's mailbox. Thus, during the lifetime of any one session, the session can contain capabilities for at most a single user's mailbox files.

It follows that a web client who does not know Alice's password cannot obtain a session with access to her mailbox.

User Isolation. We verified a user isolation property: the execution of a request on behalf of one user (say, the attacker) cannot influence the execution of code running on Alice's behalf in a way that would affect the contents of Alice's mailbox, except for authorized transmission of e-mail messages to Alice. We verified this in two phases: first, we ruled out influence via overt channels by reasoning about all possible overt channels between two sessions; then, we ruled out unwanted influence via covert channels by examining the code running on Alice's behalf.

We eliminated the possibility of unwanted influence via overt channels by reasoning about the possible runtime heap graphs of our application. In particular, the heap graph separates into mostly-disjoint regions. Each region corresponds to a distinct user: the region contains all of the objects transitively reachable from any session associated with that user, except that we prune the transitive traversal at certain *bridge objects*. In addition, there is a set of shared resources that are not associated with any particular region. See Figure 3 for an example. Bridge objects are objects that have a capability to a shared resource, an external resource, or an external communication channel. Bridge objects enable a potential communication channel between regions and as such must be reviewed to verify that they do not violate user isolation.

In our application, the bridge objects are the `CreateAccount`, `Auth`, and `MTA` classes. We reviewed their code to confirm that they do not permit unintended communication. `CreateAccount` and `Auth` could in principle be used by servlets to communicate across sessions, but we verified that the servlets do not in fact use this potential channel to communicate in a way that would violate integrity. No servlet listens on this communication channel and uses the message received to influence its modifications to mailbox contents. `DeleteServlet` does not query the existence or non-existence of other accounts or allow their existence to affect the modifications it makes to Alice's mailbox. `LoginServlet` does not use its capability to the mailbox files to modify the contents of any mailbox. No other servlet ever receives a capability to modify the contents of any mailbox file.

Mail transport can also be used to communicate across sessions, but it cannnot violate the recipient's integrity because the `DeleteServlet` does not look at the contents of any mail message before deciding which one to delete.

Since servlets are immutable, they cannot be used as a communication channel between users. Therefore, they are not bridge objects and need not be reviewed when verifying user isolation. As part of the framework, the dispatcher is assumed not to violate servlet isolation.

5.1.2 Verifying Privacy

To ensure that the privacy of Alice's messages is preserved, we reviewed all application code that can potentially read the contents of Alice's messages. For each such class, we verified that it does not send information about the messages to anyone other than Alice. We check that (a) it does not send this information to other connected web users or out via outgoing email, and (b) it does not communicate any of this sensitive information to other classes, e.g., by storing it in the session. If condition (b) did not hold, we would also have to review any additional code that could read that data.

Immediately after session initialization, before a user is logged in, only the `CreateAccount` and `Auth` capabilities grant any access to the user mailboxes in the filesystem. `CreateAccount` does not directly access the filesystem; it instead instructs the MTA to add a user for mail delivery and then sends a welcome mail to that user. It reads data from the filesystem only when it checks that a user does not already exist, so it does not reveal any information about the content of messages in Alice's mailbox. Use of this capability is guarded by a lock to eliminate possible race conditions from concurrent creations of the same user name. The `Auth` capability itself never looks at the contents of a user's directory, so it does not reveal any information about the content of the user's messages. Additionally, our prior review of the `Auth` capability showed that it returns a capability to a user mailbox only when supplied with the appropriate password, so an attacker who does not know Alice's password cannot use it to gain a capability to Alice's mailbox.

The only remaining way that Alice's privacy might be compromised is if servlets running on behalf of Alice read her email and

leak information about it to the attacker. After authentication, the objects in the session that provide access to her mailbox include the `CreateAccount` capability and the `ReadOnlyFile` and `DeleteMail` wrappers pointing to her mailbox directory. Of these, only the `ReadOnlyFile` is actually able to read the contents of Alice's mail. We therefore need to review only the servlets that are allowed to get this capabiltity, as specified by the policy file.

The policy file specifies that only the `Read` and `Inbox` servlets have access to the `ReadOnlyFile` capability used to read Alice's mail. The `Inbox` reads in the file names of Alice's messages and reads the content of these files in order to get their subject lines. The file names are used for URLs that point to the `Read` servlet; as web browsers do not reliably protect the URLs of pages, these are potentially readable by a malicious website acting in concert with the attacker. While these file names contain timing information used to generate a unique identifier by the local mail transfer agent, they do not reveal any information about the content of the message. The subject line, on the other hand, is included only in the response to Alice as part of a text block in the HTML page sent to Alice. Unlike the URL of a page, the browser's same-origin policy is assumed to reliably protect the content of a page from being read by script originating from other domains. Our webmail application is hosted on its own server, so we do not need to verify any script belonging to other applications. In our current implementation, our application does not have any script content, so we do not need to worry about other servlets being able to read the content output by the `Inbox` servlet. If our application had used JavaScript, we would have to review the scripts to ensure that they do not allow other servlets to read sensitive information from the `Inbox` servlet.

The `Read` servlet reads the contents of a file from the user's mailbox directory, with the filename specified by a GET parameter, and simply outputs the file's contents as a single DOM text node in the response to be sent back to Alice. We assume that the browser does not leak this text to any other domain; that should be prevented by the same-origin policy. If we had JavaScript in our application, we would have to verify that it does not read and exfiltrate the private information output by this servlet, but at present our webmail application does not use any scripts. The `Read` servlet does not make any other overt use of the contents of the email message, and thus does not violate Alice's privacy.

5.2 Usability

For the most part, our APIs are consistent with the servlet API, differing only to meet our security goals. Legacy servlet applications are not directly compatible with this model, and we have not evaluated the difficulty in porting them to it. From our experience using the model, we did not find it appreciably more difficult to use than the standard servlet API. We have not built large, complex applications atop our framework, and thus there may be issues that arise only at scale that we have not encountered. While we believe that the properties of our framework will make large programs easier to review for security than in existing frameworks, it remains unknown whether it reduces the difficulty enough to make security verification practical for large, complex applications.

Our application uses files for persistent storage, but a database is a more practical alternative for many web applications. Capsules does not currently support databases because we do not have a good least-privilege-compatible means for doing so; remedying this is an important open problem.

Another practical concern for our framework implementation is the current incompleteness of Joe-E's taming database, which specifies which Java methods can be called from Joe-E code. Java has a very large number of useful libraries and only a handful have

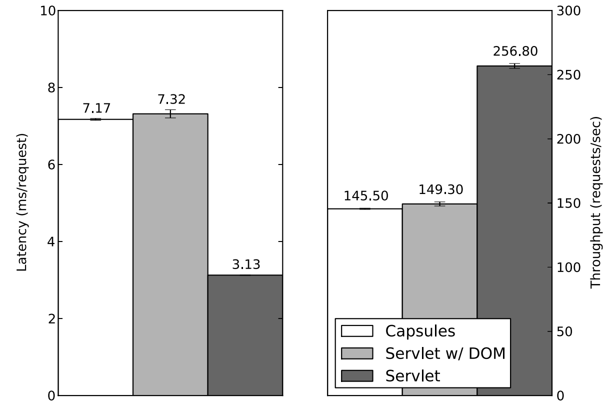


Figure 4: DOM Microbenchmark. Error bars in all figures indicate a 95% confidence interval for the mean.

been carefully scrutinized for capability discipline and enabled for use with Joe-E applications. To let applications use these libraries, it will be necessary to carefully tame these libraries to determine which methods are safe to allow, and in some cases, add supplemental APIs to expose missing functionality in a capability-safe way. This is a substantial amount of work, though the taming effort for a library can be leveraged by all Joe-E applications.

5.3 Performance Analysis

We evaluated the performance impact of Capsules by comparing against the unmodified servlet framework. To facilitate this comparison, we implemented an identical webmail application as a set of regular Java servlets. We deployed both applications on the same web server (2.26 GHz Pentium 4 with 512 MB RAM) and ran experiments that compared the two in terms of latency, throughput, and memory usage. For latency measurements, we measured the round-trip time for requests issued by a single client. For throughput, we saturated the server by simulating 20 simultaneous clients issuing sequential requests. In both cases, we eliminated network overhead by running the clients on the same machine as the server. In all our measurements, we noticed short, relatively infrequent pauses due to periodic garbage collection; despite this, our measurements were consistent when averaged over longer time periods.

DOM Microbenchmark. We evaluated the performance impact of `org.w3c` DOM API. We built three applications: one using Capsules and the DOM API; another using regular servlets and the DOM API; and a third using regular servlets and the default `PrintWriter` API. Each application writes over 200 HTML elements to the HTTP response. This enables us to separately assess the overhead of the DOM API alone and the additional overhead added by the Capsules framework for a DOM-heavy workload.

We found that the DOM API has a substantial performance penalty relative to simple string concatenation; in our experiments it nearly halved throughput and more than doubled latency (see Figure 4). Perceived latency may be even worse due to the inability to stream the response with a DOM API. We anticipate that substantial performance gains could be made by using a streamable safe templating system or SAX API instead of the DOM API.

Session Construction Microbenchmark. The Capsules dispatcher adds overhead to session construction and initialization because it populates each fresh session with capabilities needed by

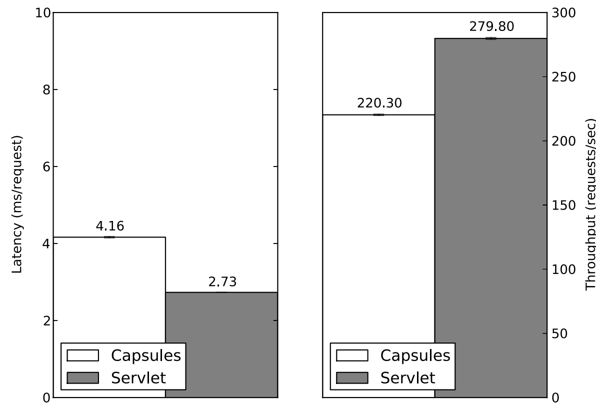


Figure 5: Session Construction Microbenchmark.

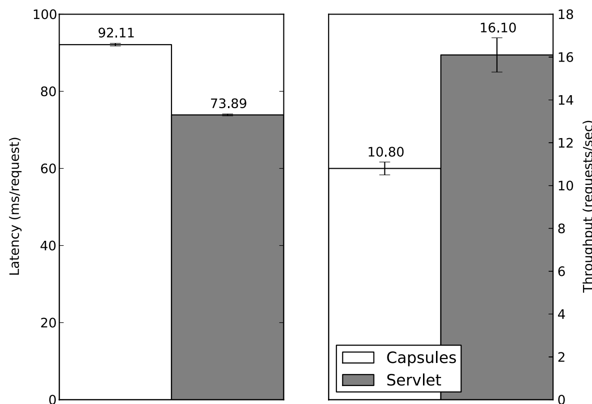


Figure 6: Application Functionality Macrobenchmark.

the application. This overhead is application-specific, depending largely on the complexity of constructing these capabilities. For this experiment, we used our webmail application's session initializer, which is nontrivial but likely to be less complex than that of a larger application. We implemented two servlets, one for Capsules and one for the Servlet Framework, that simply invalidate the associated session, without producing any output, so that on each request a fresh session is constructed. Our experiments show that Capsules adds approximately 1.5 ms of latency on session construction, which is unlikely to be noticeable when amortized over the lifetime of a typical session. See Figure 5.

Application Functionality Macrobenchmark. Our microbenchmarks do not reflect realistic web application workloads; it is likely that real applications will spend a larger proportion of their time in application-specific logic. Therefore, we measured performance on a synthetic workload: a servlet that lists the subject of 1000 messages from the user's mailbox. See Figure 6. Our measurements show approximately a 20% throughput penalty, which is substantial but may be acceptable in some cases given the security benefits.

Memory Experiments. We evaluated the memory overhead of our approach by seeing how many active sessions could be supported by each application. We found that with a 256MB heap, the Capsules implementation could handle 71,500 sessions while the control implementation allowed 153,590 sessions. This works out to an overhead of approximately 2.0KB to each session, which we believe is not substantial.

6. RELATED WORK

Our work on building secure web applications with Joe-E has parallels with SIF [1], which extends and applies the information-flow-typed language Jif [15] to web application development. Information-flow languages like Jif allow the programmer to declare security labels on program variables. These labels specify the confidentiality and integrity policies to be enforced on the variables' contents. This allows one to verify statically that integrity-critical data is influenced only by trusted sources and that private data is not leaked to unauthorized sinks. The SIF platform builds atop the Java Servlet framework to allow developers to build web applications in Jif. Like our work, it exposes a slightly modified servlet API to web applications in order to enforce its security properties. Both Capsules and SIF restrict the output of application servlets to prevent them from triggering client-side actions that would violate the intended security properties.

SIF's architecture focuses on security properties that can be expressed as information-flow properties, whereas Capsules was designed more around the notions of privilege separation, least privilege, and control of side effects. We speculate that some application security properties may be more naturally expressed in SIF, and that others may be more naturally expressed in Capsules, but we have not made any attempt to evaluate this hypothesis.

Capabilities have a long history as an approach for securing systems [9]. Early multi-user capability systems were based upon hardware support for capabilities, where each capability indicated a resource and a set of access rights. Due to the need for specialized hardware, these systems declined in popularity, but they provided the inspiration for capability-based operating systems [4, 19] and later object-capability languages. The concept of programming environments supporting security and isolation dates back as far as work by Morris in 1973 [14]. W7 implemented these features in a Scheme environment and provided an early example of language support for capabilities [17]. Joe-E, the object-capability language we use for our prototype framework, was heavily influenced and inspired by E, a seminal object-capability language [12].

Multiple projects have taken a capability-based approach to securing client-side JavaScript for web applications. The Caja project is designed to securely support coexistence of active web content from multiple sources in a single domain. They provide a tool to rewrite a webpage containing JavaScript into a self-contained module that can be used as a web gadget. This transformation sandboxes the code so it can only interact with other modules using capabilities [13]. ADsafe [2] is a more restrictive object-capability subset of JavaScript, designed to support advertisements whose security can be checked without requiring code rewriting. Our work is analogous, but applied to the server side of web development; the two can complement each other to provide client- and server-side isolation between components of a multi-tiered web application.

The term "privilege separation" traditionally refers to dividing an application into distinct components that can run in separate processes with different OS-level privileges. A typical example involves a trusted, high-privileged process that delegates most of the real work to less-privileged slave processes [16]. The OK Web Server follows this technique to provide limited privilege to web server and web application components [8]. Each component of the web application is chrooted to a jail directory and only allowed to communicate via RPC to the dispatcher and database components.

Hawblitzel et al. provide an alternate model of Java module componentization to the one we use for application servlets [5]. Their J-Kernel system establishes protection domains for Java code so that communication between domains is only possible using special `Capability` objects: normal objects are never shared across protection domains. Separate classloaders are used to enforce isolation between domains except for the special `Capability` objects, which can be retrieved from the J-Kernel's call-gate-like public interfaces of modules. The J-Kernel isolation model has the advantage of focusing attention on the precise interface between components, at the cost of reduced flexibility. Using Joe-E, our framework can provide isolation for components without the infrastructure needed to establish and permit sharing between protection domains. J-Kernel's protection domains have been used to host servlets, providing isolation between distinct servlet-based applications. However, unlike our work, they do not provide isolation between users or components of the same servlet application.

The Open Web Application Security Project (OWASP) provides a guide for developing secure web applications [21]. We feel that this type of effort in assembing best practices is valuable as an aid to web application developers. Ideally, web development frameworks should automate many of the best practice defenses described in this guide, and Capsules aims to move in this direction.

In addition to frameworks aimed at developing secure web applications, there has been a substantial body of work on static analysis for web applications to find security vulnerabilities [6, 7, 10]. We feel that these tools have a place, especially for legacy applications, but moving forward, we believe that applications should be written in higher-level frameworks that prevent many of the vulnerabilities that static analysis tools currently detect.

7. CONCLUSION

In this paper, we presented a programming model for building web applications that enables verification of high-level security properties by compartmentalizing an application into isolated components and limiting the privileges allotted to each one. We realized this model with Capsules, a prototype framework that exposes a modified Servlet API to applications and harnesses Joe-E, an object-capability language. Using this framework, a security review was able to verify important security properties of a webmail application.

8. ACKNOWLEDGEMENTS

Part of this work was inspired by a helpful suggestion from Ben Laurie, whom we gratefully acknowledge. We thank Devdatta Akhawe, Matt Finifter, Jayant Krishnamurthy, and our anonymous reviewers for helpful feedback on earlier drafts of this paper. This material is based upon work supported by the National Science Foundation under grants CNS-0716715, CCF-0424422, and CCF-0430585. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

9. REFERENCES

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

[2] D. Crockford. ADsafe. `http://www.adsafe.org`.

[3] Edgewall Software. Genshi. `http://genshi.edgewall.org`.

[4] N. Hardy. KeyKOS architecture. *SIGOPS Oper. Syst. Rev.*, 19(4):8–25, 1985.

[5] C. Hawblitzel, C.-c. Chang, G. Czajkowski, D. Hu, and T. Von Eicken. Implementing multiple protection domains in Java. In *USENIX Annual Technical Conference*, 1998.

[6] Y. W. Huang, F. Yu, C. Hang, C. H. Tsai, D. T. Lee, and S. Y. Kuo. Securing web application code by static analysis and runtime protection. In *13th International World Wide Web Conference*, 2004.

[7] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security & Privacy*, pages 258–263, 2006.

[8] M. Krohn. Building secure high-performance web services with OKWS. In *Proceedings of the USENIX Annual Technical Conference*, pages 15–28, 2004.

[9] H. M. Levy. *Capability-based computer systems*. Digital Press, Maynard, MA, USA, 1984.

[10] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *14th USENIX Security Symposium*, 2005.

[11] A. Mettler, D. Wagner, and T. Close. Joe-E: A security-oriented subset of Java. In *17th Network & Distributed System Security Symposium*, 2010.

[12] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[13] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript (draft), 2008. `http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf`.

[14] J. H. Morris, Jr. Protection in programming languages. *Commun. ACM*, 16(1):15–21, 1973.

[15] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Symposium on Operating Systems Principles*, pages 129–142, 1997.

[16] N. Provos. Preventing privilege escalation. In *12th USENIX Security Symposium*, pages 231–242, 2003.

[17] J. A. Rees. A security kernel based on the lambda-calculus. *A. I. Memo 1564, MIT*, 1564, 1996.

[18] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. In *Communications of the ACM*, 1974.

[19] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *17th ACM symposium on Operating Systems Principles (SOSP'99)*, pages 170–185, 1999.

[20] Symantec Corporation. Symantec Global Internet Security Threat Report: Trends for 2008, April 2009.

[21] A. Wiesmann, A. van der Stock, M. Curphey, and R. Stirbei, editors. *A Guide to Building Secure Web Applications*. The Open Web Application Security Project, September 2005.