Enforcing Request Integrity in Web Applications

Karthick Jayaraman, Grzegorz Lewandowski,
Paul G. Talaga, and Steve J. Chapin

Department of EECS, Syracuse University
{kjayaram,grlewand,pgtalaga,chapin}@syr.edu

Abstract. A web application is constructed to process an intended sequence of requests. Failing to enforce the intended sequences can lead to request integrity (RI) attacks, wherein an attacker forces an application into processing an unintended request sequence. Cross-site-request forgeries (CSRF) and workflow violations are two classes of RI attacks. Enforcing the intended request sequences is essential for ensuring the integrity of the application. We describe a new approach for enforcing request integrity in a web application, and its implementation in a tool called Bayawak. Under our approach, the intended request sequences of an application are specified as a security policy, and a framework-level method enforces the security policy strictly and transparently without requiring changes in the application's source code. Our approach can be compared to operating system (OS) support for access control—access control is not built into the application, but based on OS level policy settings. We evaluated Bayawak using nine open source web applications. Our results indicate that our approach is effective against request integrity attacks and incurs negligible overhead.

1 Introduction

An upsurge of vulnerabilities that affect web applications has paralleled the trend toward their widespread deployment and use. Several web applications, particularly those comprising an office suite, have supplanted desktop applications. Furthermore, web applications such as web-mail systems, online retailing applications, and group-ware applications affect everyday life. However, existing methods for ensuring the integrity of these applications are inadequate and, as a result, web applications continue to be attractive targets of exploitation.

One aspect of web application integrity protection that remains inadequately addressed is enforcing request integrity. A web application is constructed to process certain intended request sequences. Violation of these sequences can lead to a compromise of application integrity or user privacy. Request integrity (RI) attacks are attacks wherein an external attacker or a malicious user tricks a web application into processing an unintended request sequence. There are two classes of attacks, namely cross-site-request forgeries (CSRF) and workflow attacks (WF), which until now have been treated as unrelated attacks, but which are actually subclasses of RI attacks.

S. Foresti and S. Jajodia (Eds.): Data and Applications Security XXIV, LNCS 6166, pp. 225–240, 2010.

Currently the task of enforcing request integrity is considered as belonging to the application developer. The proliferation of web application vulnerabilities rooted at developers' mistakes clearly shows that this approach did not succeed. There are several reasons for this failure. First, developers are not security experts. As a result, they may not be aware of the vulnerabilities and appropriate secure coding methods. Second, developers are prone to making mistakes. Consider the fact that all of the top 10 web application vulnerabilities can be traced to programming errors [1]. Finally, the weaknesses that make RI attacks possible are rooted at the very nature of web applications and web browsers. For example, the structure of a web application does not significantly change over time. As we explain later, this lack of diversity can be abused to gather knowledge about application's structure and then construct seemingly valid request sequences. Also, the way browsers and web applications manage user sessions is vulnerable to an attacker injecting request into existing request sequences.

Depending on developers to enforce request integrity will not only result in rapidly increasing application complexity, but also force developers to implement similar countermeasures time and again. Moreover, this approach will not work for application vulnerabilities discovered in legacy code, where modifying the source code is difficult or perhaps impossible. To avoid these problems, the task of enforcing request integrity should be performed by a security framework, not by the application developers. This is similar to the approach in operating systems, where access control is not built into individual applications, but controlled by OS level policy settings. Such an approach can also provide an assurance to the publisher that request integrity is strictly enforced.

In this paper, we describe a new approach for enforcing request integrity and its implementation in a tool called Bayawak that moves the enforcement mechanism into a security framework. In Bayawak, the valid request sequences for a web application are specified as a security policy, and a server-side method transparently enforces the valid request sequences, eliminating attacks that trick the application into processing an invalid request sequence. Bayawak does not require any changes in the web application. The valid request sequences can be abstractly specified using a request-flow graph (RFG)[1]. For example, Figure 1 contains the RFG of an online message board application. The RFG is enforced using three steps. First, Bayawak performs a behavior-preserving diversification of the RFG for each session. Second, Bayawak modifies the web pages produced by the application to be compatible with the varied per-session RFG. Third, Bayawak validates each incoming request against the per session RFG before forwarding to the application for further processing. We argue that these three steps eliminate the underlying root causes that facilitate RI attacks.

The effectiveness of our approach depends on the correctness of the RFG. There are several methods for obtaining the RFG for a web application. The RFG could be derived from the specification of the application. In the case of legacy web applications, the RFG could be derived from the source code using reverse engineering. The reverse engineering methods vary in their sophistication

[1] We will define the term in section 2.

ranging from simple web spiders to advanced program analysis methods such as WAMse [2] and Tansuo [3]. In our evaluation, we used a web spider for simple web applications and WAMse for relatively more complex web applications.

We evaluated Bayawak using nine open source web applications. We identified several RI attacks in each of the applications. After configuring Bayawak instances for each of the application, all the attacks were eliminated. Furthermore, the Bayawak instances incurred negligible overhead.

The key contributions of the paper can be summarized as follows:

1. An approach for enforcing request integrity in web applications that moves the enforcement from the application into a security framework.
2. Our approach eliminates both classes of RI attacks, namely CSRF and workflow violations, which were previously considered unrelated attacks.
3. Implementation of our approach for the Apache web server in a tool called Bayawak, and evaluation using nine open-source web applications.

Organization. The remainder of the paper is structured as follows. In Section 2, we provide background information on web applications. In Section 3, we describe RI attacks. In Section 4, we describe our approach and its implementation. In Section 5, we describe our experimental evaluation and results. We compare our work with related work in Section 6 and conclude in Section 7.

2 Anatomy of Web Applications

This section will provide background information on web applications and define the terminology we will use in the remainder of the paper. A web application comprises components such as server-side scripts, databases, and resources such as images and JavaScript programs, and is accessed over the Internet using the HTTP protocol. Typically, a user accesses a web application using a web browser. The web browser constructs HTTP requests in response to the user interacting with hyperlinks and forms in a web page, forwards them to the application, and displays the web page received in the response. Web applications receive and process incoming requests using their *interfaces* [2]. An interface receives an HTTP request and returns a web page in the response. Each HTTP request contains a target URL and several arguments in the form of name-value pairs that are either part of the URL (known as a query string) or the message body. The target URL and the name-value pairs in the request identify the target interface and we will refer to them as *interface names*.

Usually, web applications need to group incoming requests into sessions. For example, in an online shopping application, a user may add products to his shopping cart in one request, and then initiate a purchase transaction in another request. The shopping cart application should be able to group these requests into a single session and also associate the contents of the shopping cart with the correct user's session. However, the HTTP protocol was designed to be stateless so that hosts do not have to retain information about users between requests [4].

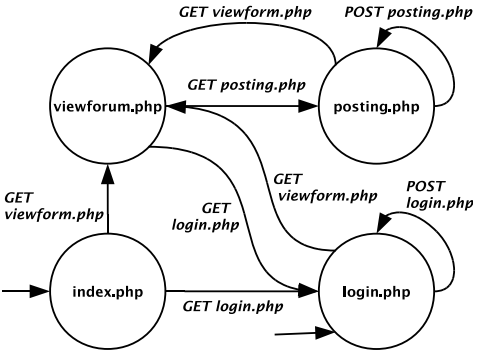

Fig. 1. RFG for an online message board

Therefore, web applications use cookies to group requests into sessions. Whenever a new session is created, web applications create a cookie in the web browser using the *set-cookie* HTTP header in the response. Web browsers attach all the cookies created by the application to all subsequent requests, helping the application associate each incoming request with its session.

Enforcing Intended Request Sequences: Each web application is designed to process certain intended sequences of requests. For example, in an online-shopping application, a request to initiate a purchase transaction is expected only after a user is signed in, and a request to finalize the purchase is expected only after a user provides valid payment information. Similarly, some web applications display the URL for the administrative interface in a web page only if the user is logged in as an administrator. These rules reflecting the intended request sequence can be abstractly represented using a graph; each node corresponds to an interface and edges correspond to HTTP requests. Two nodes are connected by a directed edge, if the web page created by an interface contains a hyperlink or form targeted to the other node. We refer to this graph as the *request flow graph (RFG)* of a web application.

In a typical intended access model, users access the web application starting from a *session-initializing* interface (SII), which creates a new user session that will be shared by all subsequent requests from the user until the session terminates. In most applications, all requests targeted to the domain name of the application are redirected to a SII. Also, in the absence of a session, all requests to non-SII are redirected to a SII. After the session is initialized, the browser issues all subsequent requests based on the user interaction with the hyperlinks and forms in the web page.

Example. The online message board application in Figure 1 has four interfaces and 10 interlinks between the interfaces. For the sake of illustration, we explain one node and its edges in the RFG. The message board application contains two SII, namely *index.php* and *login.php*. The node *login.php* has two outgoing edges. The edge to itself corresponds to a form in the web page that constructs an

HTTP POST request for *login.php* using the username and password supplied by the user. The other edge corresponds to a hyperlink in the web page for *viewforum.php*.

Developers typically enforce the intended request sequence using a combination of *interface hiding* and *validation*. Interface hiding aims to prevent users from performing an illegal action by not providing GUI that would be used to initiate the action. Web pages created by the application typically display only the necessary hyperlinks and forms in web pages that are required in the next interaction step. For example, the hyperlink or form for the next step in a transaction is displayed only if a prior step completed successfully. Similarly, the hyperlink for the administrative interface is only displayed if a user is logged in as an administrator. Validation refers to the process of embedding checks in the application in order to verify that the request in the previous step completed successfully by checking the application's state before processing the current request.

3 Request Integrity (RI) Attacks

Request integrity (RI) attacks violate the intended RFG of a web application by tricking interfaces into accepting and processing unintended requests[2]. RI attacks take advantage of the very nature of web applications and browsers and attack the underlying assumptions or weaknesses of prevailing methods used to enforce the intended request sequences. The root causes of RI attacks can be traced to three weaknesses. We will explain the three weaknesses and then present two classes of existing RI attacks.

First, the web pages created by a web application do not significantly vary between sessions because the interface names do not change. An attacker who understands the application and interface names can forge requests for the application. There are several opportunities for understanding an application. Because web applications are easily accessible to both users and attackers, attackers can understand the application by using it. Furthermore, the source code of some widely used web applications such as phpBB are publicly available.

Second, methods such as interface hiding and validation used by web applications do not strictly enforce intended request sequences. Interface hiding enforces request sequences only if the application is accessed using its web pages. For example, many web applications send web pages containing login forms to users, and expect an HTTP POST request in response. However, those applications will often process a similar HTTP POST request containing login information even if the user has not retrieved a web page containing a login form. In this case, the applications naively assume that they are accessed only via their web pages and do not strictly enforce their implicit access restrictions. In the case of validation, the checks embedded inside the application have to be complete and there should be no way to bypass the checks in order to strictly enforce the request sequence.

[2] Unintended by the application designer; clearly these requests are intentional on the part of the attacker.

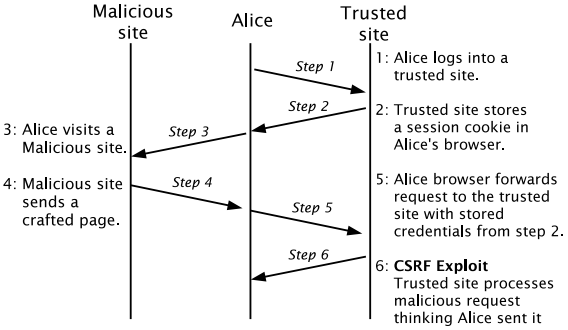

Fig. 2. Cross-site-request forgeries

Attackers attack these underlying assumptions to identify a vulnerability in the application. Furthermore, both these methods are implemented by a developer. Therefore, the efficacy of the enforcement is dependent on the security knowledge of the developer.

Third, the prevailing access policy used by web browsers for managing cookies can be abused by malicious web sites to inject session requests—web browsers attach all the cookies associated with a web application to all requests targeting the application irrespective of the origin of a request. Therefore, if a web site A embeds a HTML form or hyperlink that invokes an interface of web site B, the browser automatically attaches all the cookies (which may include a session cookie) of web site B (if any) to the requests created by web site A. As a result, web site A can inject requests into a session that the user has with web site B without the user's knowledge or consent.

Cross-site-request forgeries (CSRF): In a CSRF attack [5], an attacker uses a malicious web site to forge a request for a trusted site as though it is coming from the victim user. In a typical scenario, a user unknowingly visits a malicious site while having an active session with a trusted site. Figure 2 contains an example. Alice visits a trusted site and creates a new session (steps 1 and 2). Simultaneously, Alice also visits a malicious site (step 3), which sends a crafted page to Alice (step 4). Browsers do not have any restrictions on the URL that can be used in HTML tags such as *img*, *form*, *iframe*, etc. Using a crafted page, a malicious site can trick either the user or the browser into making a malicious request to the trusted site. When the web browser renders the crafted page, it forwards a request to the trusted site and also attaches all the cookies of the trusted site to the request (step 5). The trusted site processes the malicious request thinking it was created by Alice.

The *login CSRF* attack [6] is an interesting variation of CSRF that does not affect a user's active session. Rather, login CSRF creates a new session using the attacker's username and password. The attacker hosts a crafted page in his site that, when visited by the user, sends a login request for a trusted site using the attacker's credentials. This results in a session cookie associated

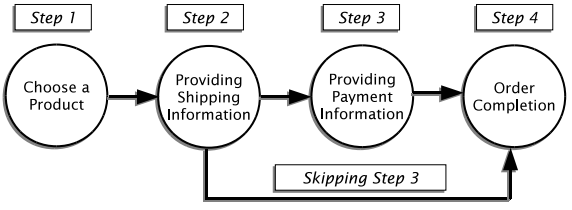

Fig. 3. A workflow violation in a purchase transaction: Using a workflow attack, an attacker skips the third step and completes the order without paying

with the attacker's credentials being stored in the user's browser. The attacker hopes that the user will later visit the trusted site; in such an event, all user activity will be attached to the attacker's session. An attacker could use this to monitor the activity of the user on a trusted site or for other malicious purposes. For instance, an attacker may be able to track all the videos that a user views on `http://www.youtube.com`.

Workflow Attacks: A workflow is a specific sequence of interactions that a web application expects a user to perform to complete a transaction. Workflows range from simple two-step workflows to highly complicated workflows. An example of a simple workflow is a web application expecting an admin user to be signed in before accessing an administrative interface. An example of a slightly more complex workflow is a purchase transaction consisting of choosing a product, providing shipping information, providing payment information, and reviewing the order before final submission (Figure 3). Recall that interface hiding and validation are typically used to enforce the workflow. Workflow attackers exploit errors in these checks, or the lack of such checks, to bypass certain steps. In the simple workflow example, an attacker could directly visit the administrative interface using its URL while being logged in as a normal user. Similarly, in the a purchase workflow, an attacker may directly visit the page associated with the final step after submitting the shipping information, thereby submitting an order without payment.

4 Bayawak

In this section, we describe the architecture of Bayawak, explain how it avoids the RI attacks, and describe Bayawak's implementation.

4.1 Architecture

The input to Bayawak is a configuration file that specifies the list of interface names in the web application, the sequences of workflows, and the name of the session cookie. Methods for obtaining the interface names vary with respect to the complexity of the application. In the case of simple applications, where each

interface corresponds to a single server-side script, the list of interfaces is essentially the list of server-side scripts in the deployment directory. Web spiders could also be used for the purpose. In more complex applications, each server-side script may implement several interfaces, each of which are distinguished by the parameters in the URL. For such applications, we need more sophisticated program analysis methods such as WAMse [2] or Tansuo [3]. WAMse uses an analysis technique based on symbolic execution for precisely identifying the interfaces in web applications.

Bayawak instantiates a run-time monitor for the web application based on the configuration file. The run-time monitor protects the application using the following three steps:

1. Diversify the interface names in the application for each session.
2. Modify the web pages created by the application to reference the correct interface names for the session.
3. Verify whether each incoming request references the correct interfaces names and only forward conforming requests to the application.

Step 1: Behavior-preserving Diversification: Bayawak creates a session-specific RFG whenever the web application creates a new session. Bayawak tracks the *set-cookie* header in the responses to detect the creation of a new session. A web application creates a new session in two steps. First, the application initializes a new session and assigns an identifier. Second, the application instructs the web browser to create a session cookie using the *set-cookie* header in the response. Bayawak intercepts each response created by the application and detects for the presence of the *set-cookie* header. On detecting a new session, Bayawak labels all the interfaces using a set of random numbers. We will refer to the labels identifying each node as interface identifiers (IID). In the session RFG, the interface names additionally include the IID.

The IID for the interfaces is a server-side secret associated with each session. Bayawak stores the mapping between the interfaces and IID for each session

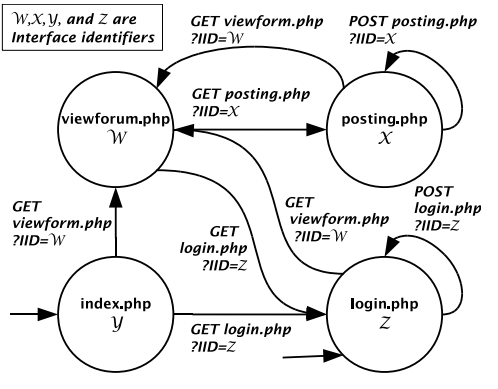

Fig. 4. Diversified session RFG

in an in-memory map. The IID should be sufficiently long, so that it is nearly impossible to guess them. By default, the IID are 256 bit numbers, but can be configured to be larger numbers. Figure 4 contains a session RFG for the online message board we described in section 2. There are no changes to the edges, which determine the request-response behavior of the application. Essentially, the RFG of each session vary only in the IID, so the RFG of various sessions can be considered isomorphic to each other. Therefore, the behavior of the application does not vary with the sessions.

Bayawak refreshes the IID for workflow interfaces on a per-transaction basis. Therefore, the set of IID that identify the interfaces involved in a workflow are unique to each transaction. All other interfaces are issued a per-session IID, which expires only at the end of the session.

Step 2: Modifying the web pages: Because the interfaces names vary with the session, the web pages created by the application do not have the correct IID and are no longer compatible with the session RFG. Therefore, the web pages are also varied for each session to incorporate the correct IID in all the URL, each time a web page is created by the application. The IID can be incorporated in the URL as a parameter; Figure 4 describes how the IID can be added to the URL as a parameter. The following HTML tags can specify a URL as an attribute and instruct the browser to create an HTTP request for the application:

1. *href* attribute of *a*, *style*, and *link* tags.
2. *action* attribute of *form* tags.
3. *src* attribute of *frame*, *iframe*.
4. *onclick* attributes of *button* tags.
5. *refresh* attribute of *button* and *meta* tags.
6. *url* attribute of *refresh* meta tags.

Bayawak modifies the URL in all these tags to incorporate the IID. HTTP redirects are a special case of responses that should be handled separately. Sometimes web applications may redirect users to URL2 in response to requests for URL1. The target for redirection, URL2, is specified using the *Location* header in an HTTP 302 response. The browser issues a request for the target URL2 when it receives the redirect response. Bayawak intercepts redirect responses and adds the IID to the URL specified in the *Location* header.

Because the IID are specific for each session and are only contained in the web pages, users can access the application only using the web pages. Essentially, the web pages become a capability required to access the application. Without the capability, users cannot access the application.

Step 3: Validating requests: Bayawak validates each incoming request before allowing the web application to process the request. There are two type of requests, namely session and non-session requests. Session requests are part of an on-going session and carry a session identifier. Non-session requests are not part of a session and typically target a SII and the application creates a new session when processing the requests. Non-session requests are directly forwarded

to the application. All session requests are expected to carry the correct IID required to invoke the interface. If a session request does not contain the correct IID, the request is treated as a non-session request and redirected to an SII, after invalidating the session. Whenever a request carries the correct IID, it is evidence that the request was created as a result of the user interacting with the web page created by the application.

4.2 Avoiding RI Attacks

Bayawak addresses the root causes of RI attacks as follows:

1. The web pages and the RFG are varied per session, so any information obtained from using one application instance or reading the source code is not adequate for forging requests to the application. This is because the IID required for making a request vary with session and are sufficiently long to thwart brute-force attacks.
2. Users are forced to access the application using the web pages because only the web pages carry the correct IID required to invoke the interfaces. The web pages force users to access the application in the intended way and all the intended request sequences are strictly enforced irrespective of the completeness of the validation checks or the integrity of the session variables used in the interfaces.
3. Malicious web sites cannot access the IID necessary to invoke an interface. The IID are only embedded inside the web pages of the application. The *same-origin policy* prohibits web applications belonging to one domain from accessing the contents of the web page belonging to other domains [7].

We now describe how our approach avoids the two RI attacks we described in section 3.

Cross-site-request Forgeries (CSRF): A malicious site cannot access the IID required to invoke an interface. Therefore, the malicious site can only create a request without the IID. Such requests are treated as non-session requests and are redirected to an SII, thwarting the attacks. A Login CSRF attack forges a login request for the application. Depending on how an application creates a new session, a login request may or may not be a session request, but our approach avoids the attack in either case. In general, applications use one of two methods for creating new sessions. First, applications may create a new session in response to a non-session request. In this case, the user is not authenticated when the session is created and is expected to login only after the session is created. As a result, all login requests are session requests and are expected to have the IID that the attacker cannot access. Hence, the attack is thwarted. Second, applications may create a new session only after user authentication. In this case, the login request is a non-session request, which is forwarded to the application. Therefore, an attacker may be able to forge a login request for the application using the attacker's credentials. However, the session initializing processes creates a new session and a session RFG and the victim user does not

have access to the IID compatible with session RFG. Therefore, when the user initiates a new request to the application independently using the browser, it will be associated with the session but will not have the appropriate IID. Recall that such requests invalidate the session, and are redirected to a SII, forcing the user to authenticate.

Workflow Attacks: Workflow attacks are eliminated in two ways. First, the web pages would only carry the IID for the interfaces they reference. Therefore, users cannot access interfaces that are not referenced by the web pages, thwarting arbitrary URL accesses. Second, the IID for the workflow interfaces are unique for each transaction. Typically, the web pages display the hyperlinks for the workflow steps in the intended sequence. The hyperlink for a step is displayed only on successful completion of the previous step. Therefore, the user is forced to step through the workflow only in the intended way. Moreover, because the IID for workflow interfaces expire at the end of each transaction, IID collected from completing a prior transaction in the session cannot be used to directly invoke the interface associated with the final step in a subsequent transaction.

4.3 Implementation

Bayawak is available in two forms—an Apache module written using mod_perl and a Java class implementing a Servlet API filter. The Apache module extends the request-response processing pipeline to implement Bayawak. The Java filter is essentially a hook into the Servlet interpreter for manipulating the requests and responses processed by the application. Both of our implementations are functionally equivalent.

5 Experimental Evaluation

We evaluated Bayawak using open source web applications to ascertain the following:

1. Resistance to RI Attacks.
2. Run-time overhead of using Bayawak instances for protecting applications.

Experimental Setup. We installed and configured nine web applications, namely phpBB [8], punBB [9], Scarf [10], osCommerce [11], WebCalendar [12], Bookstore [13], Classifieds [14], Employees [15], and Events [16] on a web server configured with Intel Pentium-4 933MHz processor, 1GB RAM, Ubuntu Linux 8.04, MySQL 5.0, and the Apache 2 web server. phpBB and punBB are discussion board applications, Scarf is a conference management system, WebCalendar is a multi-user calendar application, osCommerce is an e-retailer application complete with a shopping cart, Bookstore is an online bookstore application, Events is a multi-user group-ware application, Classifieds is an online classifieds management application, and Employees is an online employee directory. Each application was installed as specified for use with a MySQL database. Web clients

accessed the applications over a 100Mb Ethernet connection to measure their performance. phpBB, punBB, Scarf, osCommerce, and WebCalendar are built using PHP, so we used Bayawak available in the form of a mod_perl module. Bookstore, Classifieds, Employees, and Events are built using JSP, so we used Bayawak available in the form of a Servlet API filter.

Collecting Interface Names. We collected interface names for the various web application using two methods. For phpBB, punBB, Scarf, osCommerce, and WebCalendar, we used a simple web spider. For Bookstore, Classifieds, Employees, and Events, we used the WAMse tool to extract the list of interface names.

5.1 Resistance to RI Attacks

We identified several RI attacks for all the nine web applications. Table 1 provides a summary of attacks.

We found several CSRF vulnerabilities in all the applications. In the discussion board applications, phpBB and punBB, the vulnerabilities allow an attacker to forge new messages or delete existing ones. In osCommerce, we identified attacks that can add, modify, or delete products in the shopping cart and submit forged product reviews. In Scarf, the identified attacks can add or delete papers to sessions in a conference. In WebCalendar, the attacker can add or delete entries in the calendar and add or delete users from the calendar. In Classifieds, an attacker may add add, update, or delete the classified category headings or advertisements. In Events, an attacker may add, update, or delete events, user records, or category headings for events. In Employees, an attacker may add, update, or delete employee records or department names. In Bookstore, an attacker may add items to the shopping cart or add artificially high or low ratings for a book.

Table 1. RI attacks on example applications

Web Application	Attack Type	Attacks	Attacks Eliminated
osCommerce	CSRF	7	7
phpBB	CSRF	5	5
	Workflow attacks	1	1
punBB	CSRF	6	6
Scarf	CSRF	5	5
	Workflow attacks	1	1
WebCalendar	CSRF	5	5
Bookstore	CSRF	4	4
Employees	CSRF	3	3
	Workflow attacks	1	1
Classifieds	CSRF	6	6
	Workflow attacks	1	1
Events	CSRF	3	3

Scarf and Classified applications contained illegal URL access attacks. In Scarf, a server-side script that processes the site-wide configuration settings does not check whether the user has administrator privileges before making changes. The URL for the configuration page is only displayed in the web page if an administrator logs in. However, users can directly visit the URL associated with the configuration page and make changes. Similarly, in Classifieds, a server-side script that updates or deletes the category headings does not check whether the user has administrative privileges before making changes. Therefore, normal users can directly visit the URL associated with the server-side script and make changes. We created a illegal URL access vulnerability in phpBB. By default, the application displays the URL for the administrative interface only when the administrator logs in and the administrative script additionally checks the permission of the user. We disabled the permission checks to create an illegal URL access vulnerability.

For the purpose of evaluation, we created a workflow vulnerability in the osCommerce application. The checkout workflow comprises adding items to the shopping cart, entering shipping information, payment information, and final submission of the order. We created a vulnerability so that users could skip the payment step and directly proceed to the final submission step by visiting the URL directly.

All the attacks failed when we configured Bayawak instances for the web applications.

5.2 Performance Overhead

We measured the performance overhead of Bayawak by comparing the average response times for typical use cases of the applications with and without Bayawak instances. For each application multiple use cases were repeated with different users and content, providing at least 100 request-response pairs per application. Table 2 summarizes the average overhead of using Bayawak for all nine applications. The performance overhead significantly varied between the two forms of Bayawak. While the Apache mod_perl module incurred an

Table 2. Performance overhead from using Bayawak instances

Web Application	**Application Response (msec)**	**Avg. Bayawak Overhead (msec)**	**Percent Overhead (%)**
phpBB	278	55	19%
punBB	106	29	27%
Scarf	65	62	94%
WebCalendar	295	31	10%
osCommerce	325	96	29%
Bookstore	136	7	5%
Employees	121	4	3.5%
Classifieds	165	15	9%
Events	119	6	5%

overhead of 55ms, the Java-based Servlet API filter incurred an overhead of 8ms. The overhead of the Apache module could be reduced by implementing using C instead of Perl.

Bayawak's absolute overhead is related to the HTML document length, not application complexity. Bayawak detects the *set-cookie* header, creates a new RFG if necessary, but then must parse the HTML and rewrite URL. Therefore, the relative slowdown incurred by Bayawak will be the smallest for applications with non-trivial logic and relatively simple output. Conversely, simple applications with verbose output will have a higher relative overhead. Scarf is an example of a simple application with minimal server-side processing. Hence, its relative slowdown was the highest of all tested applications and is misleading as it represents the worst-case scenario for Bayawak deployment. All the other tested applications feature non-trivial logic and had significantly smaller relative overhead. In all cases Bayawak's overhead was imperceptible to end users. Moreover, our relative overhead estimates are conservative because the network latency in our test environment is likely to be much smaller compared to real deployments.

6 Related Work

In this section, we provide a comparison of our approach to current work.

Intrusion detection. Bayawak is related to intrusion detection approaches that enforce a security policy derived from a program's implementation. For example, there is work on constraining the execution of a program based on the model of system-call invocation derived from the program [17,18]. In these approaches, a run-time monitor tracks the current state/context of the program and detects malicious system-call sequences. Guha et al. [19] proposes a similar method for detecting malicious client-side behavior in AJAX-based applications. These approaches are broadly not applicable to web applications for two reasons. First, in web applications, unlike desktop applications, a single application instance is shared by multiple users. Second, a user may be simultaneously viewing several web pages. Both these aspects of web applications can confuse these state-tracking methods, leading to a lot of false positives.

Bayawak is related to control flow integrity (CFI) enforcement, in which a control-flow graph derived from a program is enforced by inserting run-time checks into the program binary [20]. Our approach can be considered a variant of this technique for web applications for the purpose of avoiding RI attacks. The enforcement mechanism under our setting is more complex compared to CFI, because we vary the RFG for each session.

Bayawak is related to work on using probabilistic models for detecting web-based attacks [21,22,23,24]. In particular, Swaddler [21] uses probabilistic models to characterize the internal session variables and associate invariants with blocks of code for the purpose of detecting workflow attacks. Bayawak avoids all RI attacks as opposed to just workflow attacks. Moreover, Bayawak does not require any training.

Mitigation Techniques. Current work has proposed several mitigation methods for preventing CSRF attacks such as purely client-side methods [25, 26], HTTP referrer header validation [27], proposals for new headers [6], and secret-token validation techniques [5]. A key difference of our approach from this body of work is that our approach more generally avoids RI attacks. In particular, Bayawak avoids workflow violations, which are outside the scope of these techniques.

Ripley [28] uses redundant execution of client-side code at the server-side to detect malicious JavaScript clients that subvert the client-side computation in AJAX-based applications. This is different from our problem setting, which is RI attacks.

Secure construction frameworks. SIF [29] uses language-based information flow control to enforce confidentiality and integrity policies on data. Robertson and Vigna [30] propose the use of strong typing to statically enforce strict separation of structure and content in web pages for avoiding cross-site scripting attacks. In contrast to both these approaches, our approach is focused on RI attacks and does not require any changes in the web application. At the same time, our approach is complimentary and can be implemented in these frameworks for avoiding RI attacks.

7 Conclusion

We described an approach for enforcing request integrity in web applications, and its implementation in a tool called Bayawak. Bayawak moves the request integrity enforcement mechanism from the application code into a security framework. Under our approach, the application's intended request sequences, or the request-flow graph (RFG), are specified as security policy and a server-side component transparently enforces the intended request sequences, without requiring any changes in the application's source code. Our approach is based on applying a form of behavior-preserving diversification on the RFG. We evaluated Bayawak using nine open source web applications. We identified several RI attacks in these applications. All the attacks were eliminated, when we configured Bayawak instances for the application. Moreover, Bayawak instances incurred negligible performance overhead.

References

1. Williams, J., Wichers, D.: OWASP Top 10 2010 rc1, `http://www.owasp.org/images/0/0f/OWASP_T10_-_2010_rc1.pdf`
2. Halfond, W.G., Anand, S., Orso, A.: Precise interface identification to improve testing and analysis of web applications. In: ISSTA (2009)
3. Wang, W., Lei, Y., Sampath, S., Kacker, R., Kuhn, R., Lawrence, J.: A combinatorial approach to building navigation graphs for dynamic web applications. In: ICSM (2009)
4. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, Draft Standard (1999)

5. Jovanovic, N., Kirda, E., Kruegel, C.: Preventing Cross Site Request Forgery Attacks. In: IEEE Secure Comm. (2006)
6. Barth, A., Jackson, C., Mitchell, J.C.: Robust Defenses for Cross-Site Request Forgery. In: ACM CCS (2008)
7. Ruderman, J.: The Same origin policy, https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript
8. phpBB Group: phpbb, http://www.phpbb.com/
9. PunBB: Punbb, http://punbb.informer.com/
10. SCARF, http://scarf.sourceforge.net/
11. osCommerce, http://www.oscommerce.com/
12. WebCalendar, http://sourceforge.net/projects/webcalendar/
13. Bookstore, http://www.gotocode.com/apps.asp?app_id=3&/
14. Classifieds, http://www.gotocode.com/apps.asp?app_id=5&/
15. Employee, http://www.gotocode.com/apps.asp?app_id=6&/
16. Events, http://www.gotocode.com/apps.asp?app_id=7&/
17. Wagner, D., Dean, D.: Intrusion Detection via Static Analysis. In: IEEE SP (2001)
18. Xu, H., Du, W., Chapin, S.J.: Context Sensitive Anomaly Monitoring of Process Control Flow To Detect Mimicry Attacks and Impossible Paths. In: RAID (2004)
19. Guha, A., Krishnamurthu, S., Jim, T.: Using Static Analysis for Ajax Intrusion Detection. In: WWW (2009)
20. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: ACM CCS (2005)
21. Cova, M., Balzarotti, D., Felmetsger, V., Vigna, G.: Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications. In: RAID (2007)
22. Ingham, K.L., Somayaji, A., Burge, J., Forrest, S.: Learning DFA representations of HTTP for protecting web applications. Computer Networks 51(5) (2007)
23. Kruegel, C., Vigna, G.: Anomaly detection of web-based attacks. In: ACM CCS (2003)
24. Valeur, F., Vigna, G., Kruegel, C., Kirda, E.: An anomaly-driven reverse proxy for web applications. In: ACM SAC (2006)
25. Johns, M., Winter, J.: RequestRodeo: Client-side Protection Against Session Riding. In: OWASP Europe (2006)
26. Mao, Z., Li, N., Molloy, I.: Defeating Cross-Site Request Forgery Attacks with Browser-Enforced Authenticity Protection. In: Financial Cryptography and Data Security (2009)
27. Kerschbaum, F.: Simple cross-site attack prevention. In: Secure Comm. (2007)
28. Vikram, K., Prateek, A., Livshits, B.: Ripley: Automatically securing web 2.0 applications through replicated execution. In: ACM CCS (2009)
29. Chong, S., Vikram, K., Myers, A.C.: SIF: Enforcing confidentiality and integrity in web applications. In: USENIX-SS (2007)
30. Robertson, W., Vigna, G.: Static Enforcement of Web Application Integrity Through Strong Typing. In: USENIX-SS (2009)