Open access • Proceedings Article • DOI:10.1145/1180337.1180344

Bridging the gap between web application firewalls and web applications
— **Source link** ↗

Lieven Desmet, Frank Piessens, Wouter Joosen, Pierre Verbaeten

Institutions: Katholieke Universiteit Leuven

Topics: Web application security, Web modeling, Web service, Data Web and Web development

Related papers:

- Bridging the gap between web application firewalls and web applications: extended abstract

- Securing web application code by static analysis and runtime protection

- A Review on 0-day Vulnerability Testing in Web Application

- Position paper: why are there so many vulnerabilities in web applications?

- Web Application Security Tools Analysis

Bridging the Gap Between Web Application Firewalls and Web Applications

Lieven Desmet, Frank Piessens, Wouter Joosen, and Pierre Verbaeten
DistriNet Research Group, Department of Computer Science
Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001 Leuven, Belgium
Lieven.Desmet@cs.kuleuven.be

ABSTRACT

Web applications are the Achilles heel of our current ICT infrastructure. NIST's national vulnerability database clearly shows that the percentage of vulnerabilities located in the application layer increases steadily. Web Application Firewalls (WAFs) play an important role in preventing exploitation of vulnerabilities in web applications. However, WAFs are very pragmatic and ad hoc, and it is very hard to state precisely what security guarantees they offer.

The main contribution of this paper is that it shows how, through a combination of static and dynamic verification, WAFs can formally guarantee the absence of certain kinds of erroneous behaviour in web applications. We have done a prototype implementation of our approach building on an existing static verification tool for Java, and we have applied our approach to a medium-sized J2EE based web application.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Security, verification.

Keywords

Web Application Firewall, shared data repository, static verification, run-time enforcement.

1. INTRODUCTION

Nowadays web applications are wide-spread and more and more companies incorporate e-commerce in their business model to increase their revenues. But web applications tend to be error-prone, and these bugs are a welcome target for attackers due to their high accessibility and possible profit

gain. Therefore, the number of security incidents with web applications is rapidly increasing [7, 20].

A wide range of countermeasures exists and more recently Web Application Firewalls (WAFs) are added to the network infrastructure to counter the shortcomings of traditional network firewalls. WAFs may among others prevent broken access control vulnerabilities such as vulnerabilities leading to forceful browsing by enforcing a strict request flow. One of the problems with using WAFs for the strict request flow enforcement is the fact that they tend to have a loose coupling between their configuration and the application implementation. In that way, they can protect applications against quite general attacks, but there is no direct relationship to the bugs that actually reside in the application they want to protect. Thus, there is no guarantee that an enforced WAF policy on incoming requests protects application-specific implementation bugs.

The main contribution of this paper is that it shows how, through a combination of static and dynamic verification, WAFs can formally guarantee the absence of certain kinds of erroneous behaviour in web applications. We have done a prototype implementation of our approach building on an existing static verification tool for Java, and we have applied our approach to a medium-sized J2EE based web application. In particular, we guarantee that if the combination of a web application and a WAF policy passes our verification process, no client/server interaction will break the data dependencies on the shared session state between server-side components.

The research presented in this paper is based upon previous work [8], in which formal contracts specify interactions with a shared data repository and static verification is used to guarantee that no data dependencies are broken within a given software composition. In the previous work, we investigated applications with a deterministic, sequential program execution, whereas in this paper we model reactive, indeterministic program execution, combine static and dynamic verification, and use semantically richer component contracts.

The rest of this paper is structured as follows. Section 2 provides some background information on web applications, web vulnerabilities and Web Application Firewalls. Next, the problem statement is elaborated in section 3 and our solution to guarantee that no client/server interaction leads to unintended repository interactions is proposed in section 4. Section 5 applies the proposed solution to a small e-commerce site and section 6 discusses some identified problems with ESC/Java2, the verification tool that we build

on. In section 7, the presented work is related to existing research and, finally, section 8 summarises the contributions of this paper.

2. BACKGROUND

2.1 Web applications

Web applications are server-side applications that are invoked by thin web clients (browsers), typically using the HyperText Transport Protocol (HTTP). A user can navigate through a web application by clicking links or URLs in his browser, and he is also able to supply input parameters by completing web forms. A URL maps to a server-resident program that is executed with the user's supplied input parameters. The result of the program execution (often expressed in the HyperText Markup Language (HTML)) is then sent back to the browser where it is rendered for further user interaction.

HTTP is a stateless, application-level request/response protocol and has been in use on the World Wide Web since 1990 [9]. Since the protocol is stateless, each request is processed independently, without any knowledge of previous requests. To enable the typical user's session concept in a web application, the web application needs to add session management on top of the stateless HTTP layer. Different techniques exist to embed web requests within a user session such as the use of cookies, URL rewriting or hidden form fields [28].

Nowadays, most web applications use an underlying framework or web technology to facilitate the development and the deployment of the web application. Widespread technologies such as PHP, ASP.NET, JSP/Servlets incorporate among others the management of user sessions. Next to tracking to which user session a web request belongs, these technologies also provide server-side state for each user session. While processing a web request, server-side web components can store non-persistent, user-specific data (e.g. a shopping cart in an e-commerce-site) in a data container bound to the user session. Other web components can then retrieve this data while processing future requests in the same user session.

2.2 Servlet-based web applications

The Java Servlet technology is part of the J2EE specification [14] and provides mechanisms for extending the functionality of a web server and for accessing existing business systems [32]. A J2EE web application is typically a collection of Java Servlets, deployed in a servlet-based web container such as Tomcat, JBoss or WebSphere. Java Servlets are functional units of the web tier and extra-functional properties such as load-balancing and security are added to the webcontainer rather than to the servlets themselves.

The core functionality of the container is to handle incoming web requests and to use servlets for processing the requests. A container casts incoming HTTP requests into an object-oriented form (i.e. a *HTTPServletRequest* object) and checks to see if there is a servlet registered for processing that request. If there is a match, the request is processed by the corresponding servlet.

In addition, the J2EE specification also defines filters [31]. A filter operates as a wrapper around the processing servlet and dynamically intercepts the request and response object before or after the servlet processes the request. Filters are used to transform or process information contained in the requests or responses. Popular examples are filters for access control, for output transformations (such as XSLT), for logging and auditing.

Servlets and filters are typically stateless components and operate on a per-request basis. In order to save non-persistent, session-relevant state, servlets can store and retrieve data from a shared data repository (*HttpSession*) that is uniquely bound to a user's session.

2.3 Web vulnerabilities and Web Application Firewalls (WAFs)

Existing network security fails to effectively protect web applications against attackers [16]. Network firewalls such as stateful packet filters typically operate on the network or transport layer (e.g. granting access to a complete web application by allowing TCP port 80 traffic), whereas web applications are typically attacked on the application layer. For example, attackers exploit among others design flaws in the application logic and known weaknesses in the HTTP protocol, the browser or the web server technology. Hence, a network firewall only addresses network access control in order to control whether or not a web server can be reached, irrespective of the kind of web requests and associated data that is sent to the server.

The Open Web Application Security Project (OWASP) documented the ten most critical web applications vulnerabilities in their OWASP Top Ten [23]. In this paper, we mainly focus on broken access control vulnerabilities, in particular on vulnerabilities leading to forceful browsing [25]. Forceful browsing is the act of directly accessing web pages (URLs) without consideration for their context within an application session. Bypassing intended application flow can lead to unauthorised access to resources or unexpected application behaviour [35].

To counter web application vulnerabilities, Web Application Firewalls (WAFs) operate on the application layer (OSI layer-7) and analyse web requests between a browser and the web server [29, 2]. Often, WAFs are placed inline between the browser and server (as displayed in figure 1), and enforce real-time access control, based on application-level information such as the requested URL, the supplied credentials and input parameters and the user session's history.

A WAF can either use a positive or negative security model as basis for access decisions. In case of a positive security model, access control is based upon known positive behaviour; in case of a negative security model, access is denied to requests that reflect known dangerous traffic. A positive security model can be configured manually by the administrator or can be built automatically by observing legitimate network traffic.

In the remainder of this paper, we will focus on WAFs with a positive security model that implement criterion 4.6 of the "Web Application Firewall Evaluation Criteria" [34], i.e. the strict request flow enforcement. This criterion refers to the technique where a WAF monitors individual user sessions and keeps track of the links already followed and of the links that can be followed at any given time [34].

3. PROBLEM STATEMENT

Without strict request flow enforcement, forceful browsing attacks can compromise the correct functioning of a web application in various ways. Depending on the application,
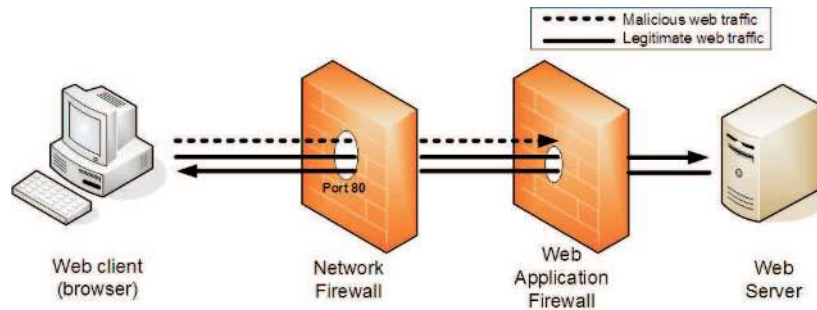
Figure 1: Web Application Firewall infrastructure

these attacks can among others circumvent access control or input validation, can corrupt server-side state or can bring the server in an inconsistent state and thus result in unexpected behaviour. More generally, the outcome of tampering with the client/server protocol can break the application logic or trigger server-side bugs.

WAFs with strict request flow enforcement are accepted as an effective countermeasure for forceful browsing attacks. However, since WAFs are mainly configured in a heuristic way, either manually or by observing legitimate network traffic, they tend to have a loose coupling between their configuration and the application implementation. Therefore, no formal guarantees can be given about the effectiveness of applying a WAF policy to protect against certain types of implementation bugs.

In section 4, we propose our solution to formally bridge the gap between the WAF enforcement policy and the web application. By combining static verification and dynamic enforcement, we are able to guarantee that the enforcement engine used protects the application against certain types of application-specific implementation bugs. Although we restrict our focus in this paper to errors that can occur on the non-persistent, server-side session state, we strongly believe that our approach is also applicable to other types of implementation bugs or infringements of the application logic.

In the next paragraph, a simple servlet-based web application illustrates the kind of errors that can occur on the server-side session state due to forceful browsing. We will then retake this application in sections 4 and 5 to clarify parts of our solution. Although we mainly use servlets in this paper as an illustration, the problem is as well valid in other web technologies such as ASP.NET or PHP, and our solution could be applied there as well.

The Duke's BookStore web application.

The Duke's BookStore web application is an exemplary Java Servlet application that is bundled together with the J2EE 1.4 Tutorial [1]. This small e-commerce application consists of about 4000 lines of code, and implements the basic functionality of a web shop by using Java Servlets. The core application logic is supplied by 6 servlets and 1 filter:

BookStoreServlet The BookStore servlet returns the main web page for the Duke's Bookstore. From this start page, links are provided to browse the book catalog, or jump to the bookdetails of a particular book (e.g. a book in promotion).

BookDetailsServlet The BookDetail servlet returns information about any book that is available from the

bookstore. A user can either add the book to the shopping cart, or look further into the book catalog.

CashierServlet The Cashier servlet asks for the user's name and credit card number so that the user can buy the books in his shopping cart. Payment information is sent to the Receipt servlet.

CatalogServlet The Catalog servlet displays the book catalog, and provides the possibility to add books to the user's shopping cart or to buy the books in the shopping cart by redirecting to the cashier servlet.

ReceiptServlet The Receipt servlet processes the order by updating the book database inventory. Afterwards the servlet invalidates the user session.

ShowCartServlet The ShowCart servlet returns information about the books in the user's shopping cart.

OrderFilter The Order filter provides server-side logging of shopping orders, whenever the ReceiptServlet is called.

These components interact with the shared session repository as listed in table 1. The interactions are specified by a type (e.g. ResourceBundle), a string identifier (e.g. messages) and the type of interaction. The interaction types used in table 1 are more fine-grained that just simple read and write operations. The type *def. read/write* stands for a defensive read/write operation as shown in listing 1, i.e. the application can handle a null pointer as result of the read operation, and in that case the servlet stores a not null object of the expected type to the shared session repository. The label *cond.* means that the operation possibly occurs, depending on an unspecified condition such as run-time state of the book database inventory.

Listing 1: Example of a defensive read/write operation in BookDetailsServlet

```
Currency c = (Currency) session.getAttribute("currency");
if (c == null) {
    c = new Currency();
    session.setAttribute("currency", c);
}
```

Session repository interactions are typically not specified in a Servlet-based application, and neither they are in this J2EE tutorial application. Thus, the implicit assumptions of the developer on how a servlet or filter should be used with respect to its interactions with the shared session repository are not shipped together with the source code. This makes correct deployment or software evolution very hard without reanalysing the complete source code.

BookDetailsServlet:
ResourceBundle messages (read)
Currency currency (cond. def. read/write)

BookStoreServlet :
ResourceBundle messages (def. read/write)

ReceiptServlet:
ResourceBundle messages (read)
ShoppingCart cart (def. read/write)

OrderFilter:
ShoppingCart cart (read)
Currency currency (read)

CashierServlet:
ResourceBundle messages (read)
ShoppingCart cart (def. read/write)
Currency currency (def. read/write)

CatalogServlet:
ResourceBundle messages (read)
ShoppingCart cart (def. read/write)
Currency currency (def. read/write)

ShowCartServlet:
ResourceBundle messages (read)
ShoppingCart cart (def. read/write)
Currency currency (cond. def. read/write)

Table 1: Interactions with the shared session repository in the BookStore application

Even in this small e-commerce application, the interactions with the shared session repository impose restrictions on the allowed client/server interaction protocol. If for example a user session starts with any URL path other than the `/bookstore` starting point of the application (which is a typical forceful browsing attack), the execution of any servlet ends up with a *NullPointerException*: every servlet retrieves the messages data item from the shared repository and assumes in its execution that the retrieved *Resource-Bundle* is not null. Another *NullPointerException* occurs in this small application if the *OrderFilter* (applied on the *ReceiptServlet*) is called in a user's session before the cart and currency data items are stored to the shared repository. The problem does however not occur if the *OrderFilter* is not applied to the *ReceiptServlet*, which may indicate that this error was introduced to the application due to evolution.

The impact of a NullPointerException during execution depends on the particular application. Possible consequences include the execution of unexpected application logic, information leakage due to bad error handling, broken data integrity by storing null strings to the database back-end, skipping of clean-up code (such as the code that closes database connections) which in turn may lead to a Denial-of-Service, and many more. In the remainder of this paper we assume that the occurrence of a *NullPointerException* due to data repository interactions in a web application negatively affects the security of the application and thus should be prevented from happening. More precisely, we define the desired application property as follows:

No broken data dependencies in the user's session shared data repository:

No client request causes a data item to be read from the server-side, shared session repository before it is actually written. For each shared data read interaction, the shared data item that is already written to the shared session repository is of the type expected by the read operation.

4. SOLUTION

In this section we propose our solution: we specify a component's interactions with the shared session repository and use static and dynamic verification to guarantee that no client/server interactions leads to violation of the the desired application property. Figure 2 depicts an overview of our solution. At the left side of the figure the different artifacts of our application are listed. Next to the implementation and the deployment information, also the WAF strict flow enforcement policy and the run-time web traffic are used as input for our verification process.

The verification process consists of three steps. Firstly, the interactions with the shared session repository are explicitly specified in component contracts, and static verification is used to verify that the component implementations obey to the contract specification. Secondly, static verification ensures that any client/server interaction protocol that complies to the WAF enforcement policy actually satisfies the component's preconditions on the shared session repository. Finally, run-time policy enforcement is used to guarantee that only web requests that obey the WAF enforcement policy are allowed to be processed by the web application. By combining these three verification steps, our solution ensures the desired application property. Our solution is sound (if the assumptions underlying our pragmatic framing hold for the application being protected, see subsection 6.2). Our solution is however not complete: to avoid undecidability issues, the static verification is necessarily conservative.

Server-side specification and verification

In order to specify a component's interactions with the shared session repository, each web component is extended with an appropriate component contract. The contract is expressed in a problem-specific contract language, which is easy to understand for application developers. Listing 2 for example, shows such a problem-specific contract of the *ShowCartServlet*, which is a straightforward mapping of the interactions specified in table 1. Next, static verification is used to verify that a component's implementation obeys its contract, i.e. that only the read and write interactions happen that are specified in the contract.

Application-specific protocol verification

As the first input artifact for the application-specific protocol verification, the strict enforcement policy of the WAF is required. This representation of the intended client/server interaction protocol can be expressed in various ways such as a regular expression, an EBNF notation or a labelled state transition system. For example, listing 3 and figure 3
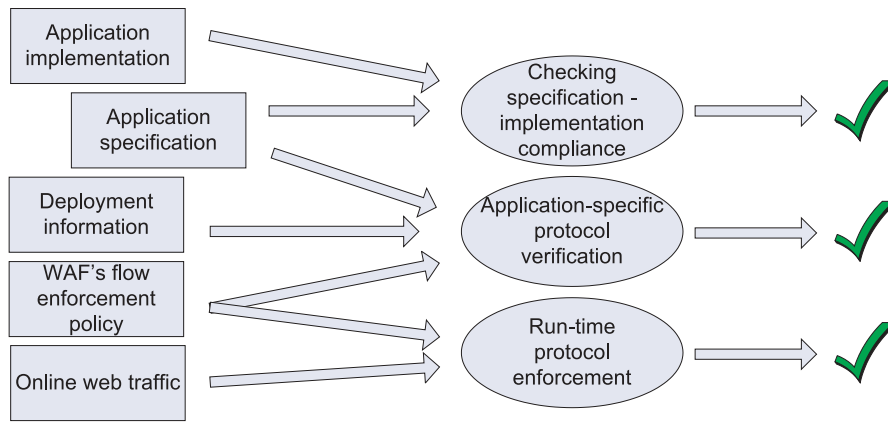
Figure 2: Solution overview

Listing 2: Problem-specific specification of ShowCartServlet

```
//spec: reads {ResourceBundle messages, Nullable<ShoppingCart> cart, Nullable<Currency> currency} from session;
//spec: writes {cart == null => ShoppingCart cart} on session;
//spec: possible writes {currency == null => Currency currency} on session;
```

are two different representations of the intended protocol for the Duke's BookStore application.

In order to statically verify that any client/server interaction (that conforms to the intended protocol) does not violate the desired application property, the intended protocol is verified in combination with the component's contracts in a given deployment. In a J2EE web application for example, the web deployment descriptor contains among others the mapping between URLs and servlets, as well the servlets on which filters are applied.

Run-time protocol enforcement

Finally, the verified client/server protocol needs to be enforced at run-time. This is done by loading the protocol specification into a supporting WAF.

5. PROTOTYPE IMPLEMENTATION

In this section, we describe our prototype implementation an discuss how it can be used to secure the Duke's BookStore application.

5.1 Server-side specification and verification

In order to use existing verifiers to check if the implementation of a component adheres to its contract, the problem-specific contracts are translated into the Java Modeling Language (JML) [18] which is a popular formal contract specification language for components written in Java.

The JML contract in listing 4 expresses interactions between actions and the shared data repository in terms of pre- and post-state of the repository. For read interactions, the component's contract indicates that the component requires that a non-null data item of the specified type can be read from the shared repository. For write interactions, the ensures pragma states which data items on the shared repository will be non-null and of the specified type after method execution. In Listing 4 for example, the JML con-

tract of the *doGet* method of the *ShowCartServlet* states that among others the shared data item *messages* will be a non-null *ResourceBundle* object before execution and after execution that the data item *cart* is ensured to be a non-null *ShoppingCart*. In addition, the *modifies* clause expresses the frame condition, i.e. what part of the session state a method is allowed to modify.

Finally, notice the use of the *also* keyword. The *ShowCartServlet* extends the *HttpServlet*, and by doing so it inherits the public method specification of the *doGet* method. To refine the specification of an overridden method (e.g. by weakining preconditions or by strengthening postconditions), the specification in JML starts with the *also* keyword, which combines the specifications of the supertype and the subtype. Similarly, the *also* keyword can also be used in regular specification to combine different specification blocks into a nested specification. More information about the desugaring of *also combinations* in JML can be found in [27, 6].

Since the *doGet* method of the *HttpServlet* does not provide common behavior for all the inheriting servlets, the supertype method is annotated with the strongest possible precondition, i.e. the *requires false* pragma. In this way, all inheriting servlets are able to weaken this precondition conform the Liskov principle.

One of the main advantages of JML is the large amount of tool support that is available [5]. Tools are available for run-time contract checking, test generation, static verification and inference of specifications. A variety of static verification tools is available that make different trade-offs in verification power and need for user interaction.

In our prototype, we chose to use the ESC/Java2 verifier [17]. The main advantage of this verifier is that it requires no user interaction. On the downside, the verifier is far from complete, for instance reasoning about loops is fairly weak. In addition, some known sources of unsoundness [19, 6] regarding to framing and reentrancy should be avoided.
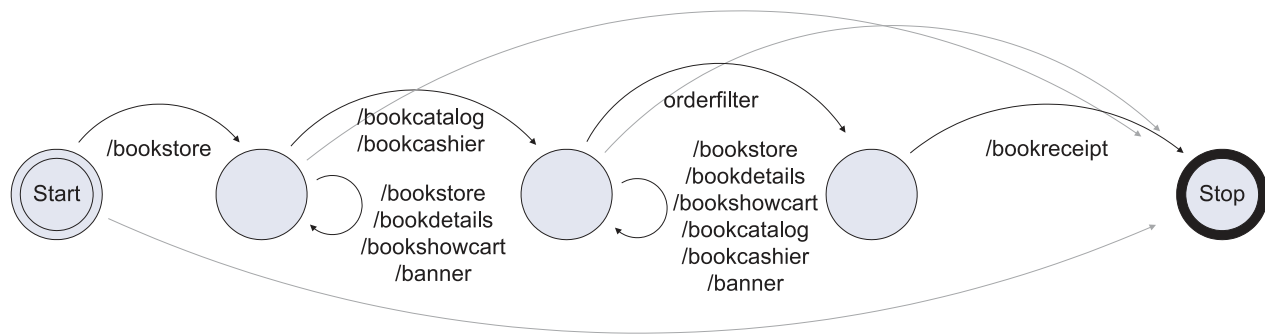
Figure 3: Client/server interaction protocol

Listing 3: EBNF notation of the client-server protocol

PROTOCOL := /bookstore + **SERVLET_A** ∗ + **RECEIPT**
RECEIPT := (**SERVLET_B** + **SERVLET** ∗ + orderfilter + /bookreceipt) | nil
SERVLET := **SERVLET_A** | **SERVLET_B**
SERVLET_A := /bookstore | /bookdetails | /bookshowcart | /banner | nil
SERVLET_B := /bookcatalog | /bookcashier

Listing 4: Contract for shared session repository interactions (ShowCartServlet.spec)

```
package servlets;

public class ShowCartServlet extends HttpServlet {
    //@ also
    //@ requires request != null;
    //@ requires response != null;
    //@ requires request.session.messages instanceof ResourceBundle;
    //@ requires request.session.cart instanceof ShoppingCart || request.session.cart == null;
    //@ requires request.session.currency instanceof Currency || request.session.currency == null;
    //@ ensures request.session.cart instanceof ShoppingCart;
    //@ ensures request.session.currency instanceof Currency || request.session.currency == null;
    //@ ensures \old(request.session.currency) instanceof Currency ==> \old(request.session.currency) ==
                                                                      request.session.currency;
    //@ modifies request.session.cart;
    //@ modifies request.session.currency;
    public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException;
}
```

To check the compliance of the component implementation with ESC/Java2, the specification of the shared repository is generated (listing 5). Hereby, *explicit JML pragmas* provide a mapping between a *ghost field* and the state of a specific data item in the hashtable since the current version of the ESC/Java2 tool does not support reasoning about hashtable indirections. This mapping allows us to express the state of the data repository in a component's contract in terms of the object fields rather than hashtable indirections, and allows us to still reason about this state without losing the verification power of ESC/Java2.

Listing 5: JML contract of the session repository (HttpSession.spec)

```
package javax.servlet.http;

public interface HttpSession {
    //@ public ghost Object cart;
    //@ public ghost Object currency;
    //@ public ghost Object messages;

    //@ requires name == "cart";
    //@ ensures this.cart  == value;
    //@ modifies this.cart;
    //@ also
    //@ requires name == "currency";
    //@ ensures this.currency == value;
    //@ modifies this.currency;
    //@ also
    //@ requires name == "messages";
    //@ ensures this.messages == value;
    //@ modifies this.messages;
    public void setAttribute(String name, Object value);

    //@ requires name == "messages";
    //@ ensures \result == this.messages;
    //@ also
    //@ requires name == "currency";
    //@ ensures \result == this.currency;
    //@ also
    //@ requires name == "cart";
    //@ ensures \result == this.cart;
    public /*@ pure @*/ Object getAttribute(String name);

    //@ requires false;
    public void removeAttribute(String name);
}
```

In verifying the compliance, we used a pragmatic framing approach instead of verifying the JML modifies clauses with ESC/Java2, as will be explained in subsection 6.2. Hereby, we verify the specified frame conditions only with respect to state changes in the shared repository, and ignore state changes in other parts of the application. This pragmatic framing is sufficient for our approach since we are only interested in the component's interactions with the shared repository.

5.2   Application-specific protocol verification

To statically verify that any client/server interaction does not violate the desired application property, a server-side protocol check is automatically generated from the protocol specification. This protocol check simulates the intended
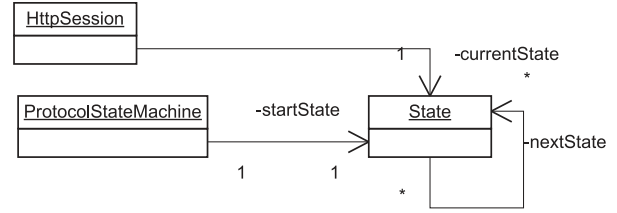

Figure 4: Class diagram of the run-time enforcement engine

protocol in a server-side method body, in which every web interaction is translated into a method call to the appropriate request processing component (if needed preceded by one or more filters). In addition, reactive or indeterministic behaviour is translated by applying the *java.util.Random* class, if-then-else branches, switch-cases and while-loops. The protocol-simulating check method for the Duke's BookStore application is listed in listing 6.

The application-specific protocol verification is then reduced to statically verifying the implementation of the check method with ESC/Java2. Compliance to a component's assumption on the shared session state is verified implicitly since ESC/Java2 checks that the preconditions are fulfilled for each method that is called.

5.3   Run-time protocol enforcement

As a proof of concept, we embedded a lightweight WAF in our web application container by installing a J2EE Filter. Before a servlet is invoked by means of the *service(ServletRequest request, ServletResponse response)* method in a J2EE web application, a chain of deployed filters is always applied to the request.

At deployment time, our enforcement engine is loaded with an object-oriented instantiation of the labelled state transition system (figure 4). For each user session the current state is stored, and for each incoming web request, the enforcement engine verifies that the transition is allowed and the current state is updated before the request is dispatched to the servlet. In case of a protocol violation, a pluggable strategy is consulted, defining the action that should be taken ranging from blocking access to the originator's IP or invalidating the user's session to just logging the access violation.

6.   DISCUSSION

6.1   Results of the BookStore experiment

Annotation overhead

As a quantification of annotation overhead, a specification line count is performed on the annotated components. At most 4 lines of specification are used to express the interactions with the shared session repository. In addition, thanks to the pragmatic framing (see subsection 6.2) almost no library calls need to be instrumented to verify the different components.

Static verification performance

To evaluate the performance of the static verification process, the verification time is measured. The performance

Listing 6: Protocol-simulating check method to be verified by ESC/Java2

```java
//@ requires request != null;
//@ requires request.session.messages == null && request.session.cart == null && request.session.currency == null;
public void protocolCheck(HttpServletRequest request, HttpServletResponse response){
    try {
        Random random = new Random();
        bookstore.doGet(request,response);
        while(random.nextBoolean()){
            int randomInt = random.nextInt();
            switch(randomInt){
                case 0: showcart.doGet(request,response); break;
                case 1: banner.doGet(request,response); break;
                case 2: bookstore.doGet(request,response); break;
                case 3: bookdetail.doGet(request,response); break;
                default: break;
            }
        }
        if(random.nextBoolean()){
            switch(random.nextInt()){
                case 0: cashier.doGet(request,response); break;
                default: catalog.doGet(request,response); break;
            }
            while(random.nextBoolean()){
                switch(random.nextInt()){
                    case 0: showcart.doGet(request,response); break;
                    case 1: catalog.doGet(request,response); break;
                    case 2: cashier.doGet(request,response); break;
                    case 3: bookstore.doGet(request,response); break;
                    case 4: bookdetail.doGet(request,response); break;
                    case 5: banner.doGet(request,response); break;
                    default: break;
                }
            }
            orderFilter.doFilter(request,response,null);
            receipt.doPost(request,response);
        }
    }
    catch(Exception e) { e.printStackTrace(); }
}
```

tests were run on a Pentium Mobile (1.4GHz) with 512MB RAM, running Debian Linux, while using Java 1.4.2_09, ESC/Java2 2.0a9 and Simplify 1.5.4. Table 2 shows the performance results of verifying the implementation compliance. The verification of the protocol-simulating method succeeded smoothly in about 11 seconds.

Run-time enforcement overhead

To estimate the overhead of the run-time flow enforcement, we ran the following experiment on the BookStore application with and without our enforcement filter. We sequentially simulated 1000 different visitors, in which each user's protocol consisted of 6 web requests and 2 % of the visitors applied forceful browsing. In this experiment, we measured a run-time overhead of 1.3 %. The BookStore application was deployed on the Sun Java System Application Server Platform Edition 8.2.

6.2 Limitations

Sequential processing

In our solution, we assume that the web requests in a user's session are processed sequentially on the web server, in the same order as they pass the WAF. In practice however web servers tend to process incoming requests in a multithreaded way (e.g. by using thread pools). At this moment our solution is only valid if our assumption on sequential execution holds, and therefore, supporting concurrent request processing is an important topic for future research.

Pragmatic framing

In order to reduce the verification complexity and the overhead of instrumenting all library calls, we use a pragmatic framing approach to verify if a component's implementation obeys to its contract. Instead of letting ESC/Java2 verify the *modifies* clauses, we use a component-specific specification of the session repository, in which we constrain the allowed write operations to the actual write interactions that the component claims to have in its *modifies* clauses.

Listing 7 is an example of such a component-specific annotation to use with the *ShowCartServlet*: the precondition of the *setAttribute* method states that only write operations are allowed for the *cart* and *currency* data item. In contrast the the complete specification of the session repository (listing 5), the *messages* data item may not be modified by the *ShowCartServlet*.

Listing 7: Component-specific specification of the repository (HttpSession.spec for ShowCartServlet)

```
package javax.servlet.http;

public interface HttpSession {
  //@ public ghost Object cart ;
  //@ public ghost Object currency ;
  //@ public ghost Object messages ;

  //@ requires name == "cart";
  //@ ensures this.cart == value;
  //@ modifies this.cart ;
  //@ also
  //@ requires name == "currency";
  //@ ensures this.currency == value;
```

```
  //@ modifies this.currency;
  public void setAttribute(String name, Object value);

  //@ requires name == "messages";
  //@ ensures \result == this.messages;
  //@ also
  //@ requires name == "currency";
  //@ ensures \result == this.currency;
  //@ also
  //@ requires name == "cart";
  //@ ensures \result == this.cart;
  public /*@ pure @*/ Object getAttribute(String name);

  //@ requires false ;
  public void removeAttribute(String name);
}
```

In case the component's implementation triggers an unspecified state change in the shared data repository, the verification of the component with ESC/Java2 will detect this contract violation (even without checking the component's *modifies* clauses), since the state change will also violate the precondition of the component-specific *setAttribute* annotation of the shared repository.

Of course, our pragmatic framing approach only guarantees correct framing regarding to state changes on the shared session repository, since other state changes in the application are neglected on purpose. Although such a pragmatic framing is not applicable for general verification purposes, this framing approach is sufficient for our verification process since we are only interested in the component's interactions with the shared repository.

In fact, the pragmatic framing approach can also be seen as an advantage rather than a limitation of our approach. Thanks to the pragmatic framing, we are able to verify partially specified components, i.e. we only specify the parts of the contract that we are actually interested in (namely the interactions with the shared repository). In case we apply traditional framing, we also have to specify every state change that occurs in the application by executing a component's method, as well as to annotate every called library method with its appropriate frame condition.

For the application-specific protocol verification, we let ESC/Java2 verify the original *modifies* clauses of the different components, in combination with the full specification of the shared session repository from listing 5.

7. RELATED WORK

Several implementation-centric security countermeasures for web applications have already been proposed [26, 11, 4, 21, 12], but most of them focus on injection attacks (SQL injection, command injection, XSS, ...) and use tainting or data flow analysis. Our solution targets another set of implementation bugs, namely bugs due to broken data dependencies on the shared, server-side state and to do so we rely on the verification of component contracts.

We combine in our solution static and dynamic verification to reduce the run-time enforcement overhead. This idea however is not new, and is for instance already adopted by Yao-Wen Huang et al. in securing web application against injection attacks [13].

In [22], Jeff Offutt et al. generate bypass tests which check if an online web application is vulnerable to forceful brows-

Component	Verif. time	Code lines	Component	Verif. time	Code lines	Component	Verif. time	Code lines
BookDetailsServlet	67.285 s	74	OrderFilter	31.632 s	61	CatalogServlet	225.866 s	123
BookStoreServlet	16.943 s	61	CashierServlet	62.742 s	60	ShowCartServlet	216.212 s	157
ReceiptServlet	5.433 s	65						

Table 2: Verification performance

ing or parameter tampering attacks. They define three levels of bypass testing: value level, parameter level and control flow level bypass testing. At this moment, our approach only counters the latter one, but in future work we want to investigate how well our approach is suited to counter the other two levels as well.

Firewall configuration analysis is proposed to manage complex network infrastructures (such as networks with multiple network firewalls and network intrusion detection systems) [33, 10]. Their approaches aim to achieve efficiency and consistency between the different network-layer security devices, whereas our approach focusses on the application-layer consistency between the WAF and the web application.

The use of JML or related languages such as Spec# [3] for verifying component properties is a very active research domain. For example, Smans et al. [30] specify and verify code access security properties, Jacobs et al. [15] verify absence of data races and Pavlova et al. [24] focus on security properties of applets. Other applications of JML are surveyed in [5].

8. CONCLUSION

This paper has focussed on bridging the gap between WAFs which enforce strict request flow, and some of the implementation-specific bugs that these kind of firewalls try to protect. We showed that through a combination of static and dynamic verification, WAFs can formally guarantee the absence of certain kinds of erroneous behaviour in web applications. In particular, we did guarantee that if the combination of a web application and a WAF policy passes our verification process, no client/server interaction will break the data dependencies on the shared session state between server-side components.

Although there are still some limitations with our proposed solution (as discussed in subsection 6.2), the conducted experiment shows that using existing verification tools to improve web application security looks promising.

9. ACKNOWLEDGEMENTS

10. REFERENCES

[1] E. Armstrong, J. Ball, S. Bodoff, D. B. Carson, I. Evans, D. Green, K. Haase, and E. Jendrock. *The J2EE 1.4 Tutorial*. Sun Microsystems, Inc., December 2005.

[2] I. Bar-Gad. Web application firewalls protect data. `http://www.networkworld.com/news/tech/2002/0603tech.html`, March 2002.

[3] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. *Lecture Notes in Computer Science*, 3362, 2004.

[4] S. W. Boyd and A. D. Keromytis. Sqlrand: Preventing sql injection attacks. In *ACNS*, pages 292–302, 2004.

[5] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

[6] D. R. Cok. ESC/Java2 Implementation Notes. `http://secure.ucd.ie/products/opensource/ESCJava2/ESCTools/docs/Escjava2-ImplementationNotes/Escjava2-ImplementationNotes.pdf`.

[7] W. A. S. Consortium. The Web Hacking Incidents Database. `http://www.webappsec.org/projects/whid/`.

[8] L. Desmet, F. Piessens, W. Joosen, and P. Verbaeten. Static Verification of Indirect Data Sharing in Loosely-coupled Component Systems. In *Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 34–49. Springer Berlin / Heidelberg, 2006.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. `http://www.ietf.org/rfc/rfc2616.txt`, 1999. Request For Comments: 2616 (Category: Standards Track).

[10] K. Golnabi, R. K. Min, L. Khan, and E. Al-Shaer. Analysis of Firewall Policy Rules Using Data Mining Techniques. In *10th IEEE/IFIP Network Operations and Management Symposium (NOMS 2006)*, April 2006.

[11] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for java. *acsac*, 0:303–311, 2005.

[12] W. G. J. Halfond and A. Orso. Amnesia: analysis and monitoring for neutralizing sql-injection attacks. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 174–183, New York, NY, USA, 2005. ACM Press.

[13] 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 *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 40–52, New York, NY, USA, 2004. ACM Press.

[14] J2EE platform specification. `http://java.sun.com/j2ee/`.

[15] B. Jacobs, K. R. M. Leino, F. Piessens, and W. Schulte. Safe concurrency for aggregate objects with invariants. In *Proceedings of the Third IEEE*

International Conference on Software Engineering and Formal Methods, pages 137–146. IEEE Computer Society, 2005.

[16] Karl Forster, Lockstep Systems, Inc. Why Firewalls Fail to Protect Web Sites. `http://www.lockstep.com/products/webagain/why-firewalls-fail.pdf`.

[17] KindSoftware. The Extended Static Checker for Java version 2 (ESC/Java2). `http://secure.ucd.ie/products/opensource/ESCJava2/`.

[18] G. T. Leavens. The Java Modeling Language (JML). `http://www.jmlspecs.org/`.

[19] K. R. M. Leino, G. Nelson, and J. B. Saxe. ESC/Java User's Manual.

[20] National Institute of Standards and Technology (NIST). National vulnerability database. `http://nvd.nist.gov/statistics.cfm`.

[21] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *SEC*, pages 295–308, 2005.

[22] J. Offutt, Y. Wu, X. Du, and H. Huang. Bypass testing of web applications. In *ISSRE*, pages 187–197, 2004.

[23] Open Web Application Security Project (OWASP). Top ten most critical web application vulnerabilities. `http://www.owasp.org/documentation/topten.html`, 2005.

[24] M. Pavlova, G. Barthe, L. Burdy, M. Huisman, and J.-L. Lanet. Enforcing high-level security properties for applets. In *CARDIS*, pages 1–16, 2004.

[25] S. Pettit. Anatomy of a web application: Security considerations. Technical report, Sanctum, Inc., July 2001.

[26] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID2005)*, pages 124–145, 2005.

[27] A. D. Raghavan and G. T. Leavens. Desugaring JML method specifications. Technical Report 00-03e, Iowa State University, Department of Computer Science, May 2005.

[28] V. Raghvendra. Session tracking on the web. *Internetworking*, 3(1), March 2000.

[29] I. Ristic. Web application firewalls primer. *(IN)SECURE*, 1(5):6–10, January 2006.

[30] J. Smans, B. Jacobs, and F. Piessens. Static verification of code access security policy compliance of .NET applications. *Journal of Object Technology*, 5(3), April 2006.

[31] Sun Microsystems, Inc. The essentials of filters. `http://java.sun.com/products/servlet/Filters.html`.

[32] Sun Microsystems, Inc. Java Servlet Technology. `http://java.sun.com/products/servlet/`.

[33] T. E. Uribe and S. Cheung. Automatic analysis of firewall and network intrusion detection system configurations. In *FMSE '04: Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, pages 66–74, New York, NY, USA, 2004. ACM Press.

[34] Web Application Security Consortium. Web Application Firewall Evaluation Criteria, version 1.0. `http://www.webappsec.org/projects/wafec/`, January 2006.

[35] webScurity, Inc. The Weakest Link: Mitigating Web Application Vulnerabilities. `http://www.webscurity.com/pdfs/webapp_vuln_wp.pdf`.