Toward Automated Detection of Logic Vulnerabilities in Web Applications

Viktoria Felmetsger Ludovico Cavedon Christopher Kruegel Giovanni Vigna
[rusvika,cavedon,chris,vigna]@cs.ucsb.edu
Computer Security Group
Department of Computer Science
University of California, Santa Barbara

Abstract

Web applications are the most common way to make services and data available on the Internet. Unfortunately, with the increase in the number and complexity of these applications, there has also been an increase in the number and complexity of vulnerabilities. Current techniques to identify security problems in web applications have mostly focused on input validation flaws, such as cross-site scripting and SQL injection, with much less attention devoted to application logic vulnerabilities.

Application logic vulnerabilities are an important class of defects that are the result of faulty application logic. These vulnerabilities are specific to the functionality of particular web applications, and, thus, they are extremely difficult to characterize and identify. In this paper, we propose a first step toward the automated detection of application logic vulnerabilities. To this end, we first use dynamic analysis and observe the normal operation of a web application to infer a simple set of behavioral specifications. Then, leveraging the knowledge about the typical execution paradigm of web applications, we filter the learned specifications to reduce false positives, and we use model checking over symbolic input to identify program paths that are likely to violate these specifications under specific conditions, indicating the presence of a certain type of web application logic flaws. We developed a tool, called Waler, based on our ideas, and we applied it to a number of web applications, finding previously-unknown logic vulnerabilities.

1 Introduction

Web applications have become the most common means to provide services on the Internet. They are used for mission-critical tasks and frequently handle sensitive user data. Unfortunately, web applications are often implemented by developers with limited security skills, who often have to deal with time-to-market pressure and financial constraints. As a result, the number of web application vulnerabilities has increased sharply. This is reflected in the Symantec Global Internet Security Threat Report, which was published in April 2009 [12]. The report states that, in 2008, web vulnerabilities accounted for 63% of the total number of vulnerabilities reported.

Most recent research on vulnerability analysis for web applications has focused on the identification and mitigation of input validation flaws. This class of vulnerabilities is characterized by the fact that a web application uses external input as part of a sensitive operation without first checking or sanitizing it properly. Prominent examples of input validation flaws are cross-site scripting (XSS) [20] and SQL injection vulnerabilities [3, 32]. With XSS, an application sends to a client output that is not sufficiently checked. This allows an attacker to inject malicious JavaScript code into the output, which is then executed on the client's browser. In the case of SQL injection, an attacker provides malicious input that alters the intended meaning of a database query.

One reason for the prior focus on input validation vulnerabilities is that it is possible to provide a concise and general specification that captures the essential characteristics of these vulnerabilities. That is, given a programming environment, it is possible to specify a set of functions that read inputs (called *sources*), a set of functions that represent security-sensitive operations (called *sinks*), and a set of functions that check data for malicious content. Then, various static and dynamic analysis techniques can be used to ensure that there are no unchecked data flows from sources to sinks. Since the specification of input validation flaws is independent of the application logic, once a detection system is available, it can be used to find bugs in many applications.

While it is important to identify and correct input validation flaws, they represent only a subset of the spectrum of (web application) vulnerabilities. In this paper, we explore another type of application flaws. In particular, we look at vulnerabilities that result from errors in the logic

of a web application. Such errors are typically specific to a particular web application, and might be domain-specific. For example, consider an online store web application that allows users to use coupons to obtain a discount on certain items. In principle, a coupon can be used only once, but an error in the implementation of the application allows an attacker to apply a coupon an arbitrary number of times, reducing the price to zero.

So far, web application logic flaws have received little attention, and their treatment is limited to informal discussions (a well-known example is the white paper by J. Grossman [14]). This is due to the fact that logic vulnerabilities are specific to the intended functionality of a web application. Therefore, it is difficult (if not impossible) to define a general specification that allows for the discovery of logic vulnerabilities in different applications.

One possible approach would be to leverage an application's requirement specification and design documents to identify parts of the implementation that do not respect the intended behavior of the application. Unfortunately, these documents are almost never available in the case of web applications. Therefore, other means to characterize the expected behavior of web application must be found for detection of application logic flaws.

In this paper, we take a first step toward the automated detection of application logic vulnerabilities. Our approach operates in two steps. In the first step, we infer specifications that (partially) capture a web application's logic. These specifications are in the form of likely invariants, which are derived by analyzing the dynamic execution traces of the web application during normal operation. The intuition is that the observed, normal behavior allows one to model properties that are likely intended by the programmer. This step is necessary to automatically obtain specifications that reflect the business logic of a particular web application. In the second step, we analyze the inferred specifications with respect to the web application's code and identify violations.

The current implementation of our approach is based on two well-known analysis techniques, namely, dynamic execution to extract (likely) program invariants and model checking to identify specification violations. However, to the best of our knowledge, the way in which we combine these two techniques is novel, has never been applied to web applications, and has not been leveraged to detect application logic flaws. Moreover, we had to significantly extend the existing techniques to capture specific characteristics of web applications and to scale them to real-world applications as outlined below.

In the first step of our analysis, we used a well-known dynamic analysis tool [9, 11] to infer program specifications in the form of likely invariants. We extended the existing general technique to be more targeted to the execution of web applications. In particular, we addressed two main shortcomings of the general approach: the fact that many invariants that relate to important concepts of web applications were not identified (e.g., invariants related to objects that are part of the user session) and the fact that many spurious invariants were generated as a result of the limited coverage of the dynamic analysis step or because of artifacts in the analyzed inputs.

To deal with spurious invariants, we developed two novel techniques to identify which derived invariants reflect real (or "true") program specifications. The first one uses the presence of explicit program checks, involving the variable(s) constrained by an invariant, as a clue that the invariant is indeed relevant to the behavior of the web application. The second one is based on the idea that certain types of invariants are intrinsically more likely to reflect the intent of the programmer. In particular, we focus on invariants that relate external inputs to the contents of user sessions and the back-end database. The use of these techniques to filter the derived invariants allows for a more effective extraction of specification of a web application's behavior, when compared to previously-proposed approaches that accept all generated likely invariants as correctly reflecting the behavior of a program.

In the second step of the analysis, we use model checking over symbolic input to analyze the inferred specifications with respect to the web application's code and to identify which real invariants can be violated. We had to extend existing model checking tools with new mechanisms to take into account the unique characteristics of web applications. These characteristics include the fact that web applications are composed of modules that can be invoked in any order and that the state of the web application must also take into account the contents of back-end databases and other session-related storage facilities.

By following the two steps outlined above, it is possible to automatically detect a certain subclass of application logic flaws, in which an application has inconsistent behavior with respect to security-sensitive functionality. Note that our approach is neither sound nor complete, and, therefore, it is prone to both false positives and false negatives. However, we implemented our approach in a prototype tool, called Waler, that is able to automatically identify logic flaws in web applications based on Java servlets. We applied our tool to several real-world web applications and to a number of student projects, and we were able to identify many previously-unknown web application logic flaws. Therefore, even though our technique cannot detect all possible logic flaws and our tool is currently limited to servlet-based web applications, we believe that this is a promising first step towards the automated identification of logic flaws in web applications.

In summary, this paper makes the following contributions:

- We extend existing dynamic analysis techniques to derive program invariants for a class of web applications, taking into account their particular execution paradigm.
- We identify novel techniques for the identification of invariants that are "real" with high probability and likely associated with the security-relevant behavior of a web application, pruning a large number of spurious invariants.
- We extend existing model checking techniques to take into account the characteristics of web applications. Using this approach, we are able to identify the occurrence of two classes of web application logic flaws.
- We implemented our ideas in a tool, called Waler, and we used it to analyze a number of servlet-based web applications, identifying previously-unknown application logic flaws.

2 Web Application Logic Vulnerabilities

Web application vulnerabilities can be divided into two main categories, depending on how a vulnerability can be detected: (1) vulnerabilities that have common characteristics across different applications and (2) vulnerabilities that are application-specific. Well-known vulnerabilities such as XSS and SQL injection belong to the first category. These two vulnerabilities are characterized by the fact that a web application uses external input as part of a sensitive operation without first checking or sanitizing it. Vulnerabilities of the second type (such as, for example, failures of the application to check for proper user authorization or for the correct prices of the items in a shopping cart) require some knowledge about the application logic in order to be characterized and identified. In this paper, we focus on this second type of vulnerabilities, and we call them *web application logic vulnerabilities*.

To detect web application logic vulnerabilities automatically, one needs to provide the detection tool with a specification of the application's intended behavior. Unfortunately, these specifications, whether formal or informal, are rarely available. Therefore, in this work, we propose an automated way to detect application logic vulnerabilities that do not require the specification of the web application behavior to be available. Our intuition is that often the application code contains "clues" about the behavior that the developer intended to enforce. These "clues" are expressed in the form of constraints on the values of variables and on the order of the operations performed by the application.

There are many ways in which constraints can be implemented in an application. In this work, we focus on two concrete types of constraints. The first (and most intuitive) way to encode application-specific constraints is in the form of program checks (i.e., *if*-statements). The presence of such a check in the program before certain data or functionality is accessed often represents a "clue" that either the range of the allowed input should be limited or that an access to an item is limited. The absence of a similar check on an alternate program path to the same program point might represent a vulnerability. For example, vulnerabilities like authentication bypass, where an attacker is able to invoke a privileged operation without having to provide the necessary credentials, could be detected using this approach.

The second type of constraints, which often exist in web applications, is the implicit correlation between the data stored in back-end databases and the data stored in user sessions. More specifically, in web applications, databases are often used to store persistent data, and user sessions are used to store the most accessed parts of this data (such as user credentials). Thus, there often exist implicit constraints on what is currently stored in the user session when a database query is issued. A "clue," in this case, is an explicit relation between session data and database data. Certain application logic vulnerabilities, like unauthorized editing of a post belonging to another user, can be detected if a path where these relations are violated is found. More detailed examples of this type of vulnerabilities will be provided in Section 4.3.2.

3 Detection Approach

Based on the discussions in the previous section, it is clear that an analysis tool that aims to detect web application logic vulnerabilities requires a specification of expected behavior of the program that should be checked. If such specifications are available (e.g., in the form of formal specifications or unit testing procedures), they can be leveraged to validate the behavior of the application's implementation. However, in many cases there is no specification of the expected behavior of a web application. In these cases, we need a way to derive it in an automated fashion.

A number of techniques has been proposed by various researchers to derive program specification automatically. However, regardless of the approach used, none of them can derive a complete specification without human feedback. To overcome this problem, we propose to use one of the existing dynamic techniques to derive partial program specifications and use an additional analysis step to refine the results and find vulnerabilities.

In particular, we observe that web applications are typically exercised by users in a way that is consistent with

the intentions of the developers. More specifically, users usually browse the application by following the provided links and filling out forms with expected input. These program paths are usually well-tested for normal input. As a result, when monitoring a web application whose "regular" functionality is exercised, it is possible to infer interesting relationships between variables, constraints on inputs and outputs, and the order in which the application's components are invoked. This information can be used to extract specifications that partially characterize the intended behavior of the web application.

As a result, in our approach, we use an initial dynamic step where we monitor the execution of a web application when it operates on a number of normal inputs. In this step, it is important to exercise the application functionality in a way that is consistent with the intentions of the developer, i.e., by following the provided links and submitting reasonable input. Note that the information about a web application's "normal" behavior cannot be gathered using automatic-crawling tools, as these tools usually do not interact with an application following the workflow intended by the developer or using inputs that reflect normal operational patterns.

In this work, as the result of the dynamic analysis step, we infer partial program specifications in the form of likely invariants. These invariants capture constraints on the values of variables at different program points, as well as relationships between variables. For example, we might infer that the Boolean variable `isAdmin` must be true whenever a certain (privileged) function is invoked. As another example, the analysis might determine that the variable `freeShipping` is true only when the number of items in the shopping cart is greater than 5. We believe that these invariants provide a good base for the detection of logic flaws because they often capture application-specific constraints that the programmer had in mind when developing the web application. Of course, it is unlikely that the set of inferred invariants represents a complete (or precise) specification of a web application's functionality. Nevertheless, it provides a good, initial step to obtain a model of the intended behavior of a program and can be used to guide further, more elaborate program analysis.

As the second step of the analysis, we use model checking with symbolic inputs to check the inferred specifications. The goal is to find additional evidence in the code about which invariants are likely to be part of the real program specification and then to identify paths where these invariants are violated.

A naïve approach would assume that all the generated invariants represent real invariants (specifications) for an application. Unfortunately, this straightforward solution leads to an unacceptably large number of false positives. The reason is the incompleteness of the dynamic analysis step. In particular, the limited variety of the input data frequently leads to the discovery of spurious invariants that do not reflect the intended program specification. To address this problem, we propose two novel techniques to distinguish between spurious and real program invariants.

The first technique aims to distinguish between a spurious and a true invariant by determining whether a program contains a *check* that involves the variables contained in the invariant on a path leading to the program point for which this likely invariant was generated. A check on a variable is a control flow operation that constrains this variable on a path. For example, the *if*-statement `if (isAdmin == true) {...}` represents a check on the variable `isAdmin`. Intuitively, we assume that a certain invariant was intended by a programmer if there is at least one program path that contains checks that enforce the correctness of this invariant (i.e., the checks imply that the invariant holds). We call such invariants *supported invariants*. When we find a supported invariant that can be violated on an alternative program path leading to the same program point, we report this as a potential application logic vulnerability. When a likely invariant can be violated, but there are no checks in the program that are related to this invariant, then we consider it to be spurious.

The second technique identifies a certain type of invariant that we always consider to reflect actual program specifications. These invariants represent equality relations between web application state variables (in particular, variables storing the content of user sessions and database contents). Relationships of that kind often reflect important internal consistency constraints in a web application and are rarely coincidental. A vulnerability is reported when the analysis determines that the equality relation is not enforced on all paths.

The vulnerability detection process and our techniques to distinguish between spurious and real invariants are discussed in more detail in Section 4.3.

4 Implementation

We chose to implement the proposed approach for servlet-based web applications written in Java. Servlets are frequently used for implementing web applications. In addition, there are a number of existing tools available for Java that can be used for program analysis. In this section, we describe the tools that we used, the extensions that we developed, and the challenges that we had to overcome to make them work together.

We first briefly introduce servlets [24]. A typical servlet-based web application consists of servlets, static documents, client-side code, and descriptive meta-information. A *servlet* is a Java-based web component

```
package myapp;
public class User {
  private String username;
  private String role;
}
public class Order {
  private int tax;
  private int total;
  private Cart cart;
}
public class Cart {
  private List products;
  private int total;
}
```

Class Definitions

```
_jspService(javax.servlet.http.HttpServletRequest req,
            javax.servlet.http.HttpServletResponse res)
            :::EXIT106

   // invariants for the field "role" belonging to an
   // object stored in the session under the key "user"
   req.session.user.role != null
   req.session.user.role.toString == ``admin''

   // invariants for the fields "cart" and "total"
   // stored in the session under the key "order"
   req.session.order.cart.total
                   == req.session.order.total
   req.session.order.total > req.session.order.tax
```

Generated Invariants

Figure 1: Example of invariants generated for an exit point on line 106 of the *_jspService* method of a servlet.

whose methods are executed on the server in response to certain web requests. Servlets are managed by a *servlet container*, which is an extension of a web server that loads/manages servlets and provides services via a well-defined API. These services include receiving and mapping requests to servlets, sending responses, caching, enforcing security restrictions, etc. Servlets can be developed as Java classes or as JavaServer Pages (JSPs). JSPs are a mix of code and static HTML content, and they are translated into Java classes that implement servlets.

4.1 Deriving Specifications

As mentioned previously, in this work, we consider program specifications that can be expressed as invariants over program variables. To derive these invariants, we leverage Daikon [9, 11], a well-known tool for dynamic detection of likely program invariants.

Daikon. Daikon generates program invariants using application execution traces, which contain values of variables at concrete program points. It is capable of generating a wide variety of invariants that cover both single variables (e.g., $total \geq 50.0$) and relationships between multiple variables (e.g., $total = price * num + tax$). Daikon-generated invariants are called *likely invariants* because they are based on dynamic execution traces and might not hold on all program paths.

Daikon comes with a set of front-ends. Each front-end is specific to a certain programming language (such as C or Java). The task of a front-end is to instrument a given program, execute it, and create data trace files. These trace files are then fed to Daikon for invariant generation. For our analysis, we leveraged the existing front end for Java, called Chicory, and plugged it into a JVM on top of which the Tomcat servlet engine [13] is executed. This allowed us to intercept and instrument all servlets executed by the Tomcat server.

The current implementation of Chicory produces traces only for procedure entry and exit points and non-local variables. Therefore, Daikon generates invariants for method parameters, function return values, static and instance fields of Java objects, and global variables.

Our changes. In addition to altering Chicory's invocation model to work with Tomcat, we extended Chicory with a way to include the content of user sessions into the generated execution traces. Invariants over this data are important for the vulnerability analysis of web applications because user sessions are an integral part of an application's state and directly affect its logic.

The content of user sessions is stored by a servlet container in the form of dynamically-generated mappings from a key to a value, i.e., as elements in a hash map container. We found that, given the current design of Daikon and Chicory, it is not possible to generate useful invariants for the contents of such containers. The reason is that Daikon requires the type and the name of all variables that can appear at a particular program point to be declared before the first trace for a particular program point is generated. This information is not available beforehand for containers like hash maps because they are dynamically-sized and can contain elements of different types.

To generate valid traces for Daikon, Chicory generates all declarations for program points at the application loading time. At this time, it needs to know the exact type of each variable/object in declaration to be able to traverse the object structure and generate precise (or interesting) invariants. For example, in order to generate a definition for the field *role* of the object of type *User* (defined in Figure 1), which might be stored in the user session of a servlet application under the key "user," Chicory needs to know that the object of the type *User* is expected in the session.

To overcome these problems, we provide our front-end with possible mappings from a key to an object type that can be observed in a session during execution. For example, for the code shown in Figure 1, we would need to provide the following mappings:

```
user:myapp.User
cart:myapp.Cart
order:myapp.Order
```

We modified Chicory to use this information to generate more precise traces for session data. This information allows for the generation of more interesting invariants, such as the ones shown in the Figure 1. We extended the front-end to generate traces for the content of user sessions for every method in an application. As future work, we plan to generate these mapping automatically for arbitrary containers by generating new declarations as new elements are found in a container, and then merging the resulting traces before feeding them to Daikon.

To generate program execution traces, we wrote scripts to automatically operate web applications. For each application, these scripts simulate typical user activities, such as creating user accounts, logging into the application, choosing and buying items from a store, accessing administrative functionality, etc. The main idea of this step is to exercise the application's common execution paths by following the links and filling out the forms presented to the user during a typical interaction with the application. The final outcome of the dynamic analysis step is a file containing a serialized version of likely invariants for the given web application. These invariants serve as a (partial, simplified) specification of the web application, and they are provided as input to the next step of the analysis.

4.2 Model Checking Applications

Once the approximate specifications (i.e., the likely invariants) for a web application have been derived, the next step is to analyze the application for supporting "clues" and identify invariants that are part of a true program specification. Any violation of such an invariant represents a vulnerability.

We chose to use model checking for this step of the analysis and implemented it in a tool called Waler (Web Application Logic Errors AnalyzeR). Given a servlet-based application and a set of likely invariants, Waler systematically instantiates and executes symbolically the servlets of the application imitating the functionality of a servlet container. As the application is executed, Waler checks the truth value of provided likely invariants, analyzes the application's code for "clues," and reports possible logic errors. In this section, we describe the architecture and execution model of Waler. Then, in Section 4.3 we explain how Waler identifies interesting invariants and application logic vulnerabilities.

4.2.1 System Top-level Design

Waler is implemented on top of the Java PathFinder (JPF) framework [19, 35], and its general architecture is shown

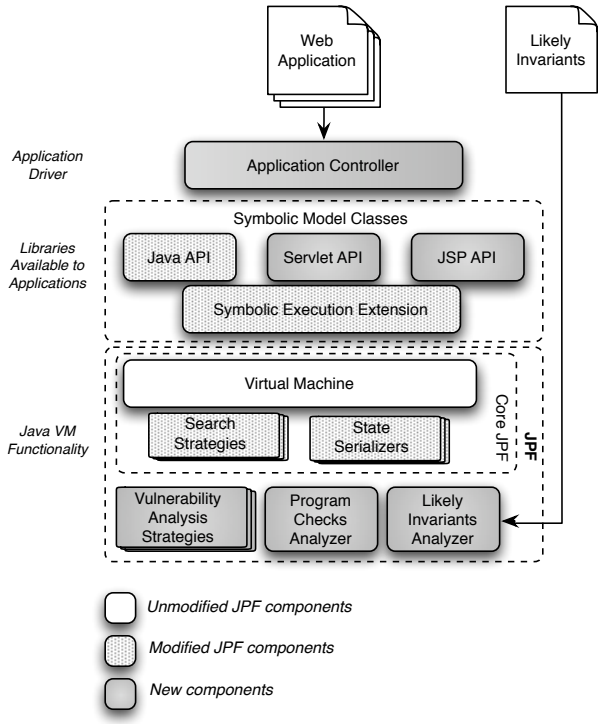

Figure 2: Waler's architecture.

Figure 2. In this figure, dark gray boxes represent new modules that we implemented, while dotted (light gray) boxes represent parts of JPF that we had to extend.

JPF overview. JPF is an open-source, explicit-state model checker that implements a JVM. It systematically explores an application's state space by executing its bytecode. JPF consists of a number of configurable components. For example, the specific way in which an application's state space is explored depends on a chosen *Search Strategy* – JPF core distribution includes a number of basic strategies. The *State Serializer* component defines how an application state is stored, matched against others, and restored. JPF also comes with a number of interfaces that allow for its functionality to be extended and modified in arbitrary ways.

In general, JPF is capable of executing any Java class-file that does not depend on platform-specific native code, and many of the Java standard library classes can run on top of JPF unmodified. However, in JPF, some of the Java library classes are replaced with their *model* versions to reduce the complexity of their real implementations and/or to enable additional features. For example, Java classes that have native method calls (such as file I/O) have to be replaced by their models, which either emulate the required functionality or delegate the native calls to the actual JVM on top of which JPF is executed.

Also, JPF comes with a number of extensions that provide additional functionality on top of JPF. Below, we discuss the JPF-SE extension for JPF, which we leveraged in Waler to enable symbolic execution.

The JPF-SE Extension. The JPF-SE extension for JPF enables symbolic execution of programs over unbounded input when using explicit-state model checking [2]. With this extension, the Java bytecode of an application needs to be transformed so that all concrete basic types, such as integers, floats, and strings, are replaced with the corresponding symbolic types. Similarly, concrete operations need to be replaced with the equivalent operations on symbolic values. For example, all objects of type *int* are replaced with objects of type *Expression*. An addition of two integers is replaced with a call to the *_plus* method of the *Expression* class. Following the standard symbolic execution approach, all newly-generated constraints are added to the *path condition* (PC) over the current execution path. The generation of constraints is done in the methods of symbolic classes, and it is transparent to the application. Whenever the PC is updated, it is checked for satisfiability with a constraint solver, and infeasible paths are pruned from execution.

Unfortunately, we found that JPF-SE was missing a considerable amount of functionality that needed to be added to make the system suitable for real-world applications. For example, the classes implementing symbolic string objects were missing a significant number of symbolic methods with respect to the *java.lang.String API*, which is used extensively in web applications. Also, in order to execute an arbitrary application using JPF-SE, symbolic versions of many standard Java libraries are required. These libraries were not provided with the extension. Finally, a tool to perform the necessary transformations of Java bytecode was not publicly available, and, therefore, we implemented our own transformer by leveraging ASM [25], a Java bytecode engineering library.

Waler overview. In order to execute servlet-based web applications and analyze them for logic errors, we had to extend JPF in a number of ways. As shown in Figure 2, we implemented from scratch four main components: the *Application Controller* (AC), the *Vulnerability Analysis Strategies* (VAS), the *Program Checks Analyzer* (PCA), and the *Likely Invariants Analyzer* (LIA). The AC component is responsible for loading, mapping, and systematically initiating execution of servlets in a servlet-based application. As the analyzed application itself, it runs on top of the JVM implemented by core-JPF and uses symbolic versions of Java libraries.

The other three components are internal to JPF, i.e., they are not visible to web applications and do not rely on model classes. The LIA component is responsible for parsing Daikon-generated invariants and checking their truth value as a program executes. The PCA component keeps track of all the program checks performed by an application on an execution path. Finally, the VAS component provides various strategies for vulnerability detection based on the information provided by LIA and PCA. We provide more details on how these modules work in the following sections.

In addition, we had to extend a number of existing JPF components to address the needs of our analysis. In particular, we modified existing search strategies, state information tracking, and implemented some missing parts of JPF-SE. Due to space limitations, we will not explain all of the changes unless they are significant for understanding our approach.

Finally, we extended JPF with a set of 40 model classes that provide the servlet API and related interfaces (such as the JSP API). These classes implement the standard functionality of a servlet container, but instead of reading and writing actual data from/to the network, they operate on symbolic values. Our implementation is based on the real implementation of the servlet container for Tomcat.

4.2.2 Execution Model

To systematically analyze a web application for logic errors, Waler needs to be able to model all possible user interactions with the application. To achieve that, it needs to find all possible entry points to the application and execute all the possible sequences of invocations using symbolic input.

In general, a user can interact with a web application in different ways: one can either follow the links (leading to URLs) presented by the application (as part of a web page) or can directly point the browser to a certain URL. On the server side, after (and if) a request URL is mapped to a servlet-based application, the path part of the URL is used to locate a particular servlet that will handle the request. We call the set of all such URL paths that lead to the invocation of a servlet the "application entry points."

Thus, before a program can be analyzed, we need to identify all possible application entry points. In the general case, there can be an infinite number of URLs that lead to an invocation of a servlet; however, for each particular application, there is a finite and well-defined number of possible mappings from a request URL pattern to a servlet. Thus, for the analysis, it is sufficient to find all such mappings. For example, if an application has the URL */login* mapped to the *AuthManager* servlet and the URLs */cart* and */checkout* mapped to the *CartManager* servlet, it can be said that the application has three entry points. In servlet-based applications, it is also possible to have wildcard mappings, such as *account/**, mapped to a servlet. In this case, all URL paths starting with */account/* are mapped to the same servlet. We consider

such mappings to represent single entry points and simply treat the part of the URL that matches the "*" as a symbolic input. This is consistent with our handling of other request parameters accessed by servlets, which are also represented by symbolic values.

To find all entry points, our system inspects the application deployment descriptor (typically, the *web.xml* file), which defines how URLs requested by a user are mapped to servlets. When analyzing the URL-to-servlet mapping, we take into account that not all servlets are directly accessible to users (those servlets that are not directly accessible are typically invoked internally by other servlets). Following the standard servlet invocation model, all URLs that point to accessible (public) servlets are assumed to be possible entry points.

Once the application's entry points are determined, the *Application Controller* systematically explores the state space of the application. To this end, it initiates execution of servlets by simulating all possible user choices of URLs. For example, if the application has three servlets mapped to the URLs */login*, */cart*, and */checkout*, the application controller attempts to execute all possible combinations (sequences) of these servlets. The actual order in which servlets are explored depends on the chosen *search strategy*. JPF offers a limited depth-first search (DFS) and a heuristics-based breadth-first search (BFS) strategy. We found that DFS works better for our system because it requires significantly less memory during model checking. With DFS, a path is explored until the system reaches a specific (configurable) limit on the number of entry points that are executed.

4.2.3 State Space Management

Similar to other model checkers, Waler faces the state explosion problem. Thus, to make Waler scale to real-world web applications, we had to take a number of steps to manage (limit) the exponential growth of the application's state space. In particular, after careful analysis of several servlet-based applications, we found that JPF often fails to identify equivalent states. The two main reasons for that are: (1) the constraints added to the symbolic PC are never removed from it due to the design of JPF-SE[1], and (2), without domain-specific knowledge, JPF is not able to identify "logically equivalent" states. Here we present three techniques that we implemented to overcome these problems.

States in JPF. JPF comes with some mechanisms to identify equivalent states. A state in JPF is a snapshot of the current execution status of a thread, and it consists of the content of the stack, heap, and static variables storage. This snapshot is created when a sequence of executed instructions reaches a *choice* point, i.e., a point where there is more than one way to proceed from the current instruction. Choice points are thread-scheduling instructions, branching instructions that operate on symbolic values, or instructions where a new application entry point needs to be chosen. Whenever JPF finds a choice point, a snapshot of the current state is created. Then, the serialized version of the state is compared to hashes of previously-seen states. The execution path is terminated when the same state has been seen before.

We found that the basic version of JPF performs garbage collection and canonicalization of objects on the heap before hashing a state. However, it does not perform any additional analysis of memory content when comparing states for equality, as JPF has no knowledge of the domain-specific semantics of the objects in memory. As a result, JPF fails to recognize certain states as logically equivalent. This leads to a large number of states that are created unnecessarily. We discuss examples of some cases in which the standard JPF mechanism fails to identify equivalent states below.

States in Waler. In Waler, we extend the concept of JPF state to a "logical state" using the domain-specific knowledge that Waler has about web applications. In particular, we observe that the only information that is preserved between two user requests in a servlet-based application are the content of user sessions, application-level contexts, the symbolic PC (which stores constraints on symbolic variables stored in sessions), and data on persistent storage. Since we do not model persistent storage in Waler and always return a new symbolic value when it is accessed, we ignore this information in our analysis. Thus, the logical state of servlet-based application is defined as the content of user sessions and application contexts, and the PC. This is the only information that should be considered when comparing states after execution of a user request is finished.

State space reduction. Given the design of JPF and using our concept of logical state, we implemented three solutions to reduce the state space of a web application.

First of all, we implemented an additional analysis step to remove a constraint from the PC when it includes at least one variable that is no longer live[2]. This is especially important when the execution of a user request is finished, because, in a web application, input received by one servlet is independent from input received by another servlet, and, unless parts of it are stored in a persistent storage, any constraints on previous input are unrelated to the new one. The implemented solution is safe (it does not affect the soundness of the analysis) and allows our system to identify many states that are equivalent.

The second solution to reduce an application state space is to prune many "irrelevant" paths from state exploration. Consider, for example, an */error* servlet, which simply displays an error message, or a */products* servlet, which displays a list of available products. Exe-

```
1 public void _jspService(HttpServletRequest req,
2                         HttpServletResponse res) {
3
4   User user = (User) session.getAttribute("User");
5   if(user==null) {
6     User.adminLogin(request,response);
7     return;
8   }
9   ...
10  if(request.getMethod().equalsIgnoreCase("post")) {
11    result = website.variables.
12            insert(new Variable(req));
13  }
14 }
```

/admin/variables/Add.jsp

```
1  public void _jspService(HttpServletRequest req,
2                          HttpServletResponse res) {
3
4    User user = (User) session.getAttribute("User");
5    if(user==null || (!user.isAdmin())) {
6      User.adminLogin(request,response);
7      return;
8    }
9    ...
10   out.println("<a href=\"admin/variables/\
11           Add.jsp\">Add New</a>");
12 }
```

/admin/variables/index.jsp

Figure 3: Simplified version of an unauthorized access vulnerability in the JspCart application.

cuting such servlets often results in changes to the state of the memory, for example, due to different Java classes that must be loaded. However, once such a servlet is executed, the application is still in the same logical state. Also, the state after executing, for example, the servlet */login* will be logically equivalent to the state resulting from the execution of the sequence of servlets *[/error, /login]*. From this observation, it is clear that it would be beneficial to identify servlets whose executions do not modify the logical state of the application. The reason is that there is no need to consider them for vulnerability analysis. Therefore, after a servlet is executed, we analyze the content of the application's memory to determine whether the application logical state has been changed (for example, because of changes to the content of the user session). When no changes are detected, the exploration of the current execution path is terminated. This modification also does not compromise the soundness of the analysis, assuming that the memory analysis takes into the account all the component of the application logical state.

A third technique to limit the state space explosion problem is to identify irrelevant entry points, so that the servlets mapped to these URLs do not need to be executed. More precisely, during model checking, when our analysis determines that a servlet does neither read from nor write to the application's logical state at all, the execution of this page can be ignored for all other execution paths. The pruning of irrelevant servlets is especially helpful in large applications, where the execution of a servlet over symbolic inputs can take several minutes (and thus, can result in days of model checking time if the servlet is executed on multiple paths).

To summarize, the state explosion problem that can rise in the model checking of web applications can be significantly improved in many cases. In particular, we developed the following three techniques to limit the growth of an application's state space: we improved the existing JPF state hashing algorithm to disregard a path condition when its variables are out of scope, we found a way to prune the exploration of irrelevant paths, and we identify irrelevant servlets and discard them from our vulnerability analysis. We found that these techniques often allow for a significant reduction in the number of states explored by Waler. For example, running Waler on the *Jebbo-2* application (described in Section 5) without using any of our state reduction techniques resulted in the execution of 322,637 states, and it took around 223 minutes to terminate. When the same application was executed using our three heuristics, Waler terminated in about a minute and needed to explore only 529 states to obtain the same result.

4.3 Vulnerability Detection

As described in the previous section, Waler uses model checking to systematically explore the state space of an application. During the model checking process, the system checks whether the likely invariants generated by Daikon for a program point hold whenever that point is reached. In our current implementation, we only consider likely invariants that are generated for exit points of methods (note that we differentiate between different exit points). The reason is that methods often check their parameters inside the function body (rather than in the caller). As a result, entry invariants are typically less significant.

To see an example of invariants that can be produced by our system, consider the code in Figure 3, which shows a vulnerability that Waler found in the JspCart applications (see Section 5). The left listing shows the code of the */admin/variables/Add.jsp* servlet, which is a privileged servlet that should only be invoked by an administrator. This is reflected by the set of likely invariants that are generated for the exit point on Line 14 for *Add.jsp*[3]:

```
(1) session.User != null
(2) session.User.isAdmin == true
(3) session.User.txtUsername == "admin@jspcart.com"
```

It can be seen that the first two invariants are part of the "true" program specification, while the third invariant

is spurious (an artifact of the limited test coverage). As a side note, the invariant for the exit point at *Add.jsp: Line 7* would be `session.User == null`.

To help us to determine whether a likely invariant holds or fails on a path, we implemented the *Program Checks Analyzer* module that keeps information about all the checks performed on an execution path. When a comparison instruction is executed, the PCA records the names of the variables involved and the result of the comparison. Also, the PCA keeps track of all variable assignments in the program. As a result, whenever the PCA encounters a check that operates on local variables, it can determine how this check constrains (affects) non-local variables. Recall that Daikon does not generate invariants for local variables, and, therefore, we are not interested in comparisons over local variables unless they store session data or method parameters.

Consider now what happens when Waler analyzes the *Add.jsp* servlet. After Waler executes the *if*-statement on Line 5, information about a new check is added to the set of current constraints accumulated by the PCA. If the user is authenticated, the value stored in the `session` object under the key `User` is not null. In this case, the PCA adds `session.User != null` to the set of checks along the current execution path, and the execution proceeds at Line 9[4]. Otherwise, the PCA records the fact `session.User == null`, and execution proceeds at Line 6.

Once the Line 14 of *Add.jsp* is reached, Waler checks whether all likely invariants generated for this point hold. A likely invariant holds on the current path if we can determine that the relationship among the involved variables is true. An invariant fails otherwise. To determine whether a likely invariant holds, we check whether the truth of this invariant can be determined directly given the current application state (i.e., the invariant involves concrete values). If not, we check whether the set of constraints accumulated on the current path implies the relationship defined by the invariant using the constraint solver employed by the JPF-SE.

Following the example, it can be seen that the first invariant for Line 14 always holds (because of the check on Line 5), while the other two might fail on some paths. In principle, we could immediately report the violations of the last two invariants as a potential program flaw. However, this would raise too many false positives, due to spurious invariants. In the following sections, we introduce two techniques to identify those invariants that are relevant to the detection of web application logic flaws.

4.3.1 Supported Invariants

The first technique to identify real invariants is based on the insight that many vulnerabilities are due to developer oversights. That is, a developer introduces checks that enforce the correct behavior on most program paths, but misses an unexpected case where the correct behavior can be violated.

To capture this intuition, we defined a technique that keeps track of which paths contain checks that support an invariant and which paths are lacking such checks. More precisely, an execution path on which a likely invariant holds **and** it is supported by a set of checks on that path is added to the set of *supporting paths* for this invariant. That is, along a supporting path, the program contains checks that ensure that an invariant is true. A path on which a likely invariant can fail is added to the set of *violating paths*. When a likely invariant holds on all program paths to a given program point, then we know that it holds for all executions and there is no bug. When all paths can possibly violate a likely invariant, then we assume that the programmer did not intend this invariant to be part of the actual program specification, and it is likely an artifact of the limited test coverage. An application logic error is only reported by Waler if at least one *supporting path* and at least one *violating path* are found for an invariant at a program point.

Let us revisit the example of Figure 3. Waler determines that the first invariant on Line 14 of `Add.jsp` always holds. The third one is never supported, and, thus, it is correctly discarded as spurious. Moreover, Waler finds a violating path for the second invariant (`session.User.isAdmin == true`) by calling the *Add.jsp* servlet with a user in non-administrative role. However, the system also inspects the path where *index.jsp* is called first, which reflects the normal, intended flow of the application. This servlet, shown on the right of Figure 3, contains a check on Line 5 that adds the fact `session.User.isAdmin == true` to the PC (assuming that the user is authenticated as an administrator). In this case, when *Add.jsp* is invoked after *index.jsp*, the system determines that the invariant `session.User.isAdmin == true` holds and is supported. Thus, Waler finds a supporting path for this invariant. As a result, the fact that one can execute the main method of *Add.jsp* directly, violating its exit invariant `session.User.isAdmin == true`, is correctly recognized as an unauthorized access vulnerability.

We found that checking for supported invariants works well in practice. However, it can produce false positives and is not capable of capturing all possible logic flaws. The main source of false positives stems from the problem that the violation of an invariant, even when it is supported by a program check on some paths, does not necessarily result in a security vulnerability. For example, access to a normally protected page does not always result in a vulnerability because either (1) a sensitive operation performed by the page fails if a set of pre-

```
1 public void _jspService(HttpServletRequest req,
2                         HttpServletResponse res) {
3
4  if(req.getMethod() == "GET") {
5   ...
6   out.println("<form method=post"
7     + " action=\"edituser.jsp\">");
8   out.println("<input type=hidden"
9     + " name=\"username\" value="
10    + session.getAttribute("username") + ">");
11  ...
12  out.println("</form>");
13 }
14 if(req.getMethod() == "POST") {
15  ...
16  stmt = conn.prepareStatement("UPDATE users SET"
17    + " password = ?, name = ? WHERE username = ?");
18  stmt.setString(1, req.getParameter("password"));
19  stmt.setString(2, req.getParameter("name"));
20  stmt.setString(3, req.getParameter("username"));
21  stmt.executeUpdate();
22 }
23 }
```

edituser.jsp

Figure 4: Simplified user profile editing vulnerability (Jebbo-6).

```
1 public void doPost(HttpServletRequest req,
2                    HttpServletResponse res) {
3  ...
4  sess = request.getSession(true);
5  if(action.equals("/editpost")){
6    s = conn.prepareStatement("UPDATE posts SET"
7      + " author= ?, title = ?, entry = ?"
8      + " WHERE id = ?");
9    s.setString(1, (String)sess.getAttribute("auth"));
10   s.setString(2, req.getParameter("title"));
11   s.setString(3, req.getParameter("entry"));
12   s.setString(4, req.getParameter("id"));
13   s.executeUpdate();
14 }
15 }
```

PostController.java

Figure 5: Simplified post editing vulnerability (Jebbo-5).

conditions, uncontrolled by an attacker, is not satisfied, or (2) there is no sensitive operation on the path executed during the access. Reasoning about these cases is extremely hard for any automated tool. However, we found that such false positives often indicate non-security bugs in the code, and, thus, they are still useful for a developer. This technique also fails to identify logic vulnerabilities when the programmer does not introduce any checks for a security-relevant invariant at all. In such cases, Waler incorrectly concludes that an invariant is spurious because it cannot find any support in the code. To improve this limitation, we introduce an additional technique in the following section.

4.3.2 Internal Consistency

As mentioned previously, Waler will discard invariants as spurious when they are not supported by at least one check along a program path. This can lead to missed vulnerabilities when the invariant is actually security-relevant. To address this problem, we leverage general domain knowledge about web applications and identify a class of invariants that we always consider significant, regardless of the presence of checks in the program.

We consider a likely invariant to be *significant* when it relates data stored in the user session with data that is used to query a database. Capturing this type of relationships is important because both the user session object and the database are the primary mechanism to store (persistent) information related to the logical state of the application. Moreover, we do not allow any arbitrary relationships: instead, we require that the invariant be an equality relationship. Such relationships are rarely coincidental because, by design, session objects and the database often replicate the same data.

Whenever Waler finds a path through the application that violates a significant invariant, it reports a logic vulnerability. To implement this technique, the system needed to be extended in two ways. First, we instrumented database queries so that the variables used in creating SQL queries are captured by Daikon and included into the invariant generation process. To this end, for each SQL query in the web application, we introduced a "dummy" function. The parameters of each function represent the variables used in the corresponding database query, and the function body is empty. The purpose of introducing this function is to force Daikon to consider the parameters for invariant generation at the function's exit point. Second, we require a mechanism to identify significant invariants. This was done in a straightforward fashion by inspecting equality invariants for the presence of variables that are related to the session object and database queries.

To see how the internal consistency technique can be used to identify a vulnerability, consider the code shown in Figure 4. This figure shows a snippet of code taken from the *edituser.jsp* servlet in one of the Jebbo applications (see Section 5)[5]. The purpose of this servlet is to allow users to edit and update their profiles. When the user invokes the servlet with a GET request, the application outputs a form, pre-filled with the user's current information. As part of this form, the application includes the user's name in the hidden field `username`, which is retrieved from the session object (shown in the upper half of Figure 4). When the user has finished updating her information, the form is submitted to the same servlet via a POST request. When this request is received, the application extracts the name of the user from the `username` parameter and performs a database query (lower half of Figure 4).

For this servlet, the dynamic analysis step (Daikon) generates the invariant `session.username == db_query.parameter3`, which expresses the fact

that a user can only update her own profile. Unfortunately, it is possible that a malicious client tampers with the hidden field `username` before submitting the form. In this case, the profile of an arbitrary user can be modified. Waler detects this vulnerability because it determines that there exists a path in the program where the aforementioned invariant is violated (as the parameter `username` is not checked by the code that handles the POST request). Since this invariant is considered significant, a logic flaw is reported.

The idea of checking the consistency of parameters to database queries can be further extended to also take into account the fields of the database that are affected by a query, but that do not appear explicitly in the query's parameters. Consider, for example, a message board application that allows users to update their own entries. It is possible that the corresponding database query uses only the identifier of the message entry to perform the update. However, when looking at the rows that are affected by legitimate updates, one can see that the name of the owner of a posting is always identical to the user who performs the update. To capture such consistency invariants, we extended the parameters of the "dummy" function to not only consider the inputs to the database query but to also include the values of all database fields that the query affects (before the query is executed). When multiple database rows are affected, the "dummy" function is invoked for each row, allowing Daikon to capture aggregated values of fields.

By extending the "dummy" function as outlined previously, Daikon can directly generate invariants that include fields stored in the database, even when these fields are not directly specified in the query parameters. Again, we consider invariants as significant if they introduce an equality relationship between database contents and session variables. The intuition is that these invariants imply a constraint on the database contents that can be accessed/modified by the query. If it was possible to violate such invariants, an attacker could modify records of the database that should not be affected by the query.

For example, this allows us to detect vulnerabilities where an attacker can modify the messages of other users in the Jebbo application. Consider the `doPost` function shown in Figure 5. The problem is that an authenticated user is able to edit the message of any other user by simply providing the application with a valid message `id`. During the dynamic analysis, the invariant `db.posts_author == session.auth` is generated, even though the `posts_author` field is not used as part of the update query. During model checking, we determine that this invariant can be violated (and report an alert) because there is no check on the `id` parameter that would enforce that only the messages written by the current user can be modified.

4.3.3 Vulnerability Reporting

For each detected bug, Waler generates a vulnerability report. This report contains the likely invariant that was violated, the program point where this invariant belongs to, and the path on which the invariant was violated (given as a sequence of servlets and corresponding methods that were invoked). This information makes it quite easy for a developer or analyst to verify vulnerabilities. Currently, vulnerabilities are simply grouped by program points. Given the low number of false positives, this allows for an effective analysis of all reports. However, not every alert generated by Waler currently maps directly to a vulnerability or a false positive. We found several situations where several invariant violations referred to the same vulnerabilities (or a false positives) in application code. For example, Waler generated several alerts in situations when (conceptually) the same invariant is violated at different program points or when two distinct invariants refer to the same application's concept. Finding better techniques to aggregate and triage reports in such situations is an interesting topic of research, which we plan to investigate in the future.

4.3.4 Limitations

Our approach aims at detecting logic vulnerabilities in a general, application-independent way. However, the current prototype version of Waler has a number of limitations, many of which, we believe, can be solved with more engineering. First, the types of vulnerabilities that can be identified by Waler are limited by the set of currently-implemented heuristics. For example, if an application allows the user to include a negative number of items in the shopping cart, we would be able to identify this issue only if the developer checked for that number to be non-negative on at least one program path leading to that program point. In addition, this check needs to be in a direct *if*-comparison[6] between variables. Conditions deriving from *switch* instructions or resulting from complex operations (such as regular expression matching) are not currently implemented.

Another limitation stems from the fact that we need a tool to derive approximations of program specifications. As a result, the detection rate of Waler is bounded by the capabilities of such a tool. In the current implementation, we chose to use Daikon. While Daikon is able to derive a wide variety of complex relationships between program variables, it has a limited support for some complex data structures. For example, if the `isAdmin` flag value is stored in a hash table, and it is not passed as an argument to any application function, Daikon will not be able to generate invariants based on that value. This limitation could be improved by implementing a smarter exploration technique for complex objects and/or by tracing

local and temporary variables for the purpose of likely invariant generation. However, care needs to be exercised in this case to avoid an explosion in the number of invariants generated.

Another issue that we faced when working with Daikon was scalability: in its current implementation, Daikon creates a huge data structure in main memory when processing an execution trace. As a result, using Daikon on a larger application requires a large amount of RAM. We worked around this limitation by partitioning the application into subsets of classes and by performing the likely invariant generation on each subset separately.

A more import limitation of Daikon is that invariants generated by the tool cannot capture all possible relations. For example, the currently supported by Daikon invariants do not directly capture such temporal relations, as "operation A has to precede operation B." To address these limitations, different "intended behavior" capturing tools (such as [1]) could be employed by Waler in the first step of the analysis, although we leave this research direction for future work.

Another, more general, limitation of the first step of our analysis is the fact that we need to exercise the application in a "normal" way (i.e., not deviating from the developer's intended behavior). This part cannot be fully automated and needs human assistance. Nevertheless, many tools exist to ease the task of recording and scripting browsing user activity, such as Selenium [31].

Finally, the state explosion problem is one of the main limitations of the chosen model checking approach. We have already described several heuristics that help Waler limiting the state space of an application, and currently, we are working on implementing a combination of concrete and symbolic execution techniques to further improve scalability.

5 Evaluation

We evaluated the effectiveness of our system in detecting logic vulnerabilities on twelve applications: four real-world applications, (namely, *Easy JSP Forum*, *JspCart*[7], *GIMS* and *JaCoB*), which we download from the SourceForge repository [28], and eight servlet-based applications written by senior-level undergraduate students as part of a class project, named *Jebbo*. When choosing the applications, we were looking for the ones that could potentially contain interesting logic vulnerabilities, were small-enough to scale with the current prototype of Waler, and did not use any additional frameworks (such as Struts or Faces). While we show that it is possible to scale Waler to real-world applications, its scalability is still a work in progress as it is based on two tools, JPF and Daikon, that were not designed to work on large applications.

All chosen applications were analyzed following the techniques introduced in Section 4. During the model checking phase, we explored paths until a depth of 6 (that is, the limit for the depth-first search of JPF was set to 6). Note that all vulnerabilities reported below were found at depth of three or less; we then doubled the search depth to let Waler check for deeper bugs. All tests were performed on a PC with a Pentium 4 CPU (3.6 GHz) and 2 Gigabytes of RAM.

The results of our analysis are shown in Table 1. Waler found 29 previously-unknown vulnerabilities in four real-world applications and 18 previously-unknown vulnerabilities in eight Jebbo applications. It also produced a low number of false positives. In Table 1, the columns *Lines of Code* and *Bytecode Instructions* show the size of the applications in terms of the number of lines of Java code (JSP pages were first compiled into their servlet representations) and of the number of bytecode instructions, respectively. The column *Entry Points* shows how many entry points were found and analyzed by Waler and the column *States Explored* shows how many states were covered. The columns *Likely Invariants* and *Invariants Violated* respectively show how many invariants were generated by Daikon and how many of them were reported as violated by Waler. The numbers in the column *Alerts* represent the (manual) aggregation of the reported invariants violations (as it is discussed in Section 4.3.3). The columns *Vulnerabilities*, *Bugs*, and *False Positives* show the aggregated number of vulnerabilities, security-unrelated bugs, and false alarms that were produced by Waler. Note that the numbers on these columns are based on the analysis of the aggregated alerts. Finally, the column *Running Time* shows the time required for the analysis.

5.1 Vulnerabilities

Easy JSP Forum: The first application that we analyzed is the Easy JSP Forum application, a community forum written in JSP. Using Waler, we found that any authenticated user can edit or delete any post in a forum. To enforce access control, the Forum application does not show a "delete" or "edit link" for a post if the current user does not have moderator's privileges for the current forum but fails to check these privileges when a delete or an edit request is received. Thus, if a user forges a delete/edit request to the application using a valid post id (all ids can be obtained from the source code of web pages accessible by all users), a post will be deleted/modified.

GIMS: The second application that we analyzed is the Global Internship Management System (*GIMS*) web application, a human resource management software. Using Waler, we found that many of the pages in the ap-

Application	Lines of Code	Bytecode Instructions	Entry Points	States Explored	Likely Invariants	Invariants Violated	Alerts	Vulnerabilities	Bugs	False Positives	Runtime (min)
Easy JSP Forum	2,416	7,348	2	251,657	5,824	6	3	2	0	1	319
GIMS	6,153	11,269	40	36,228	6,993	55	27	23	2	2	88
JaCoB	8,924	15,129	38	26,809	81,832	0	0	0	0	0	79
JspCart	21,294	45,765	86	1,152,661	34,286	5	5	5	0	0	4,576
Jebbo											
1	1,027	2,304	16	1,725	8,777	2	2	2	0	0	1.5
2	1,882	4,227	20	529	7,767	3	2	0	0	2	1
3	1,438	2,993	17	195	7,388	2	2	2	0	0	1
4	1,182	2,709	8	73	4,474	3	3	0	2	1	0.5
5	804	2,025	8	59	2,792	3	3	1	0	2	0.5
6	1,524	3,709	19	268	5,159	9	9	6	3	0	0.5
7	1,499	2,826	15	398	3,342	10	5	4	1	0	0.5
8	1,463	2,782	15	1,031	8,468	15	6	3	3	0	1.2

Table 1: Experimental results.

plication do not have sufficient protection from unauthorized access. In particular, our tool correctly identified 14 servlets that can be accessed by an unauthenticated user (a user that is not logged in at all). Most of these pages do contain a check that ensures that there is some user data in a session (which is only true for authenticated users). When a check fails, the application generates output that redirects the client's browser to a login page. Unfortunately, at this point, the application does *not* stop to process the request due to a missing return statement. Moreover, we found that certain pages in the *GIMS* application that should only be accessible to users with administrative privileges do not have checks to confirm the role of the current user. As a result, nine administrative pages were correctly reported as vulnerable.

JaCoB: The third application is *JaCoB*, a community message board application that supports posting and viewing of messages by registered users. For this program, our tool neither found any vulnerabilities nor did it generate any false alert. However, closer analysis of the application revealed two security flaws, which could not be identified with the techniques used by Waler. For example, when a user registers with the message board or logs in, she is expected to provide a username and a password. Unfortunately, when this information is processed by the application, the password is simply ignored. Also, in this application, a list of all its users and their private information is publicly available. These two problems represent serious security issues; however, they cannot be detected by Waler because the program specification that can be inferred from the application's behavior does not contain any discrepancies with respect to the application's code.

JspCart: The fourth test application is *JspCart*, which is a typical online store. Waler identified a number of pages in its administrative interface that can be accessed by unauthorized users. In *JspCart*, all pages available to an admin user are protected by checks that examine the *User* object in the session. More precisely, the application verifies that a user is authenticated and that the user has administrative privileges. However, Waler found that four out of 45 pages are missing the second check. Therefore, any user that has a regular account with the store can access administrative pages and add, modify, or delete settings (e.g., the processing charge for purchases). A simplified version of one of these vulnerabilities is shown in Figure 3. Waler also found a logic vulnerability that allows an authenticated user to edit the personal information of another user by submitting a valid email address of an existing user. This vulnerability is similar to the one shown in Figure 4.

Jebbo: We analyzed a set of eight Jebbo applications that were written by senior-level undergraduate students as a class project. Jebbo is a message board application that allows its users to open accounts, post public messages, and update their own messages and personal information. Some of the applications also implement a message rating functionality. For this project, all students were provided with a description of the application to implement along with a set of rules (including security constraints) that were expected to be enforced by the application.

After running Waler on this set of applications, we found that six out of eight applications contained one or more logic flaws. Examples of the vulnerabilities found by Waler include the fact that unauthenticated users can post a message to the board, and the lack of authorization checks when users rate an existing message (e.g., in order to avoid for a user to rate its own messages). Ironically, most of the student followed the provided specification carefully and were checking that access to certain pages is limited to authenticated users only; however, due to various mistakes, the enforcing checks were not always sufficient. For example, common problems that we found are missing return statements on an error path and a failure to foresee all possible paths available to a user to access a certain functionality.

Waler identified a number of application logic flaws that are associated with unauthorized data modification,

such as the possibility to edit personal information or posts belonging to another user. Some of the examples of these vulnerabilities are shown in Figure 4 and Figure 5. These vulnerabilities are classic examples of inconsistent usage of data by the application. It is interesting to observe that even though the students were aware of possible parameter tampering vulnerabilities, and, in many cases, they were very careful about checking user input for validity, they often failed to apply this knowledge to cases where there were multiple paths to the same program point.

The results for the Jebbo application demonstrate that logic flaws are hard to avoid, even in simple web applications. Almost all applications in this set were found to be vulnerable despite the fact that the students were given a clear program specification and knew basic web security practices. Given the class level of the students who were enrolled in the class, it is reasonable to assume that their programming skills are not far off from those of entry-level programmers. This, together with the fact that the complexity of real-world applications is much higher than the complexity of the Jebbo application, can be seen as an indication of how wide-spread web application logic flaws are. Moreover, it can be argued that many real-world application are, at least partially, written by students who are widely employed year-round as interns.

5.2 Discussion

As it is shown in Table 1, Waler generated a low number of false positives. Careful analysis of the alerts which did not represent a vulnerability revealed that the majority of them represent true weaknesses in code. These alerts were classified as bugs. We found that these bugs were either potential vulnerabilities that turned out to be unexploitable in particular situations or were not interesting for exploitation. For example, an unauthenticated user might be able to access a certain page, but this access does not contain any sensitive information. We classified the rest of the alerts as false positives.

We also carefully analyzed the applications for false negatives. We found that Waler missed some security problems, like the ones in JaCoB, but we consider these vulnerabilities to be out of scope as they cannot be detected using our approach. We also identified several cases where Waler missed vulnerabilities that should be detectable using the described approach. The main reason for such false negatives is the incomplete modeling of all application features in the current version of Waler. For example, Waler only identifies program checks in the form of *if*-statements, but in real applications, checks can be implemented using, for instance, database queries and regular expressions. Precise modeling of such constructs is left for future work.

The other way to evaluate the false negatives rate of Waler would be to run it on an application that has some known logic vulnerabilities. Unfortunately, we found a very limited number of such applications to be available, and none of them met all of our current selection criteria for test applications.

6 Related Work

Our work is related to several areas of active research, such as deriving application specifications, using specifications for bug finding, and vulnerability analysis of web applications. However, due to the limited space available, in this section we will only highlight the research that, in our opinion, is most related.

First, our approach is related to a number of approaches that combine dynamically-generated invariants with static analysis. For example, Nimmer and Ernst explore how to integrate dynamic detection of program invariants and their static verification on a set of simple stand-alone applications using Daikon and the ESC/Java static checker [27]. The invariants that are verified by the static checker on all paths are determined to be the real invariants for an application, and the invariants that could not be statically verified are shown as warnings to the user. The main goal of this research is to show the feasibility of the proposed approach rather than to find bugs. Another work that explores benefits of combining Daikon-generated invariants with static analysis is the DSD-Crasher tool by Csallner and Smaragdakis [8]. The main goal of this system is to decrease the false positives rate of a static bug-finding tool for stand-alone Java applications. Dynamically-generated invariants are used by the CnC tool (also based on ESC/Java) as assumptions on methods arguments and return values to narrow the domain searched by the static analyzer. In Waler, in contrast to both approaches, we do not assume that the invariants generated by Daikon are correct, and we only consider them to be clues for vulnerability analysis. Introducing our two additional techniques to differentiate between real and spurious invariants allows us to avoid many of the false positives due to limitations of the dynamic analysis step.

Our work is also related to the research on using an application's code to infer application-specific properties that can be used for guided bug finding. To the best of our knowledge, one of the first techniques that uses inferred specifications to search for application-specific errors is the work by Engler et al. [10]. Their goal is similar to ours in the sense that both works are trying to identify violations of likely invariants in applications. The way it is achieved, though, is very different in the two approaches.

While we infer specifications from dynamic analysis and check for possible violations in the code via symbolic execution, Engler's work carries out all the steps via static analysis: a set of given *templates* is used to extract a set of "beliefs" from the code. Afterward, patterns contradicting these "beliefs" are identified in the code. While some of the templates may be useful for web applications, most of the bugs they try to identify are relative to kernel and memory-unsafe programming languages operations. Moreover, we believe that having an additional source of information (i.e., dynamic traces) for application invariants makes our system more robust.

There is also recent work that uses statistical analysis and program code to learn certain properties of the application, with the goal of searching for application-specific bugs. For example, Kremenek et al. propose a statistical approach, based on factor graphs, to automatically infer which program functions return or claim ownership of a resource [21]. The AutoISES tool applies the idea of using statically-inferred specifications to the detection of vulnerabilities in the implementations of access control mechanisms for OS-level code [34]. The differences between these approaches and ours are similar to the ones with the Engler's work. Both approaches use statistical analysis to find violations of properties that must hold for all program points, and they do not require reasoning about the values of variables.

Learning invariants through dynamic analysis has already found application security purposes, mostly in order to train an Intrusion Detection System. Baliga et al. [4] employ Daikon to extract invariants on kernel structures from periodic memory snapshot of a non-compromised running system. After the training phase, these learned invariants are used to detect the presence of kernel rootkits that may have altered vital kernel structures. A conceptually similar approach has also been applied by Bond et al. [6] to Java code through instrumentation of the Java Virtual Machine. An initial learning phase is employed to record the calling context and call history for security-sensitive functions. Afterwards, the collected information is used to identify function invocations with an anomalous context. An anomalous context or history is considered an indicator of an attempt to divert the intended flow of the application, possibly by the exploitation of a logic error in the code. In that case, an alert is triggered or the execution is aborted.

Although both the techniques proposed by Baliga and Bong share with ours an initial dynamic learning phase, how the information is leveraged differs. For example, unlike the two approaches above, we do not assume that the likely invariants generated by the first phase are real invariants, rather we simply use them as hints for further analysis. In addition, while in our second phase we try to identify logic errors in the code by means of static analysis, they instead try to detect attacks being performed on a live system. Such run-time detection imposes an overhead, which results in the requirement for dedicated hardware for [4] and a 2%-9% penalty in performance for [6]. The authors of the latter work, in particular, traded some coverage of the code (limiting to security-related functions) in order to retain acceptable performance. Even though they focused on logic errors, a direct comparison with their evaluation environment was not possible, because of the different targets of the analysis. More precisely, they looked for bugs in the Java libraries triggered by Java applets, rather than bugs in Java-based web applications.

Another direction of research deals with protection of web service components against malicious and/or compromised clients. Guha et al. [15] employ static analysis on JavaScript *client* code in order to extract an expected client behavior as seen by the server. The server is then protected by a proxy that filters possibly malicious clients which do not conform to the extracted behavior.

Finally, our work is related to a large corpus of work, such as [16, 5, 7, 17, 18, 22, 26, 30, 33, 36, 23, 29], in the area of vulnerability analysis of web applications. However, most of these research works focus on the detection of or the protection against input-validation attacks, which do not require any knowledge of application-specific rules.

Among the approaches cited above, Swaddler [7] and MiMoSA [5] are tools developed by our group that look for workflow violation attacks in PHP-based web applications, using a number of different techniques (including Daikon-generated invariants). However, Waler's approach is more general and is able to identify any kind of a policy violation that is either reflected by a check in the application or that violates a consistency constraint.

Our work is also related to the QED tool presented in [23]. QED uses concrete model checking (with a set of predefined concrete inputs) to identify taint-based vulnerabilities in servlet-based applications. The main similarity between the two tools is that they both use a set of heuristics to limit an application's state space during model checking. Heuristics used by QED, however, are more specific to the taint-propagation problem and require an additional analysis step.

7 Conclusions

In this paper, we have presented a novel approach to the identification of a class of application logic vulnerabilities, in the context of web applications. Our approach uses a composition of dynamic analysis and symbolic model checking to identify invariants that are a part of the "intended" program specification, but are not enforced on all paths in the code of a web application.

We implemented the proposed approaches in a tool, called Waler, that analyzes servlet-based web applications. We used Waler to identify a number of previously-unknown application logic vulnerabilities in several real-world applications and in a number of senior undergraduate projects.

To the best of our knowledge, Waler is the first tool that is able to automatically detect complex web application logic flaws without the need for a substantial human (annotation) effort or the use of *ad hoc*, manually-specified heuristics.

Future work will focus on extending the class of application logic vulnerabilities that we can identify. In addition, we plan to extend Waler to deal with a number of frameworks, such as Struts and Faces. This will require creating "symbolic" versions of the libraries included in these frameworks. This initial development effort will allow us to apply our tool to a much larger set of web applications, since most large-scale, servlet-based web applications rely on one of these popular frameworks, and the lack of their support in Waler was a serious limiting factor when choosing real-world applications for the evaluation described in this paper.

8 Acknowledgments

We want to thank David Evans, Vinod Ganapathy, Somesh Jha, and a number of anonymous reviewers who gave us very useful feedback on a previous version of this paper.

References

[1] AMMONS, G., BODÍK, R., AND LARUS, J. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2002), ACM, pp. 4–16.

[2] ANAND, S., PASAREANU, C., AND VISSER, W. JPF-SE: A Symbolic Execution Extension to Java PathFinder. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2007), Springer.

[3] ANLEY, C. Advanced SQL Injection in SQL Server Applications. Tech. rep., Next Generation Security Software, Ltd, 2002.

[4] BALIGA, A., GANAPATHY, V., AND IFTODE, L. Automatic Inference and Enforcement of Kernel Data Structure Invariants. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual* (2008), pp. 77–86.

[5] BALZAROTTI, D., COVA, M., FELMETSGER, V., AND VIGNA, G. Multi-module Vulnerability Analysis of Web-based Applications. In *Proceedings of the ACM conference on Computer and Communications Security (CCS)* (2007), pp. 25–35.

[6] BOND, M., SRIVASTAVA, V., MCKINLEY, K., AND SHMATIKOV, V. Efficient, Context-Sensitive Detection of Semantic Attacks. Tech. Rep. TR-09-14, UT Austin Computer Sciences, 2009.

[7] COVA, M., BALZAROTTI, D., FELMETSGER, V., AND VIGNA, G. Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)* (2007), pp. 63–86.

[8] CSALLNER, C., SMARAGDAKIS, Y., AND XIE, T. Article 8 (37 pages)-DSD-Crasher: A Hybrid Analysis Tool for Bug Finding. In *ACM Transactions on Software Engineering and Methodology (TOSEM)* (April 2008).

[9] The Daikon invariant detector. `http://groups.csail.mit.edu/pag/daikon/`.

[10] ENGLER, D., CHEN, D., HALLEM, S., CHOU, A., AND CHELF, B. Bugs as deviant behavior: a general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review 35*, 5 (2001), 57–72.

[11] ERNST, M., PERKINS, J., GUO, P., MCCAMANT, S., PACHECO, C., TSCHANTZ, M., AND XIAO, C. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming 69*, 1–3 (Dec. 2007), 35–45.

[12] FOSSI, M. Symantec Global Internet Security Threat Report. Tech. rep., Symantec, April 2009. Volume XIV.

[13] FOUNDATION, T. A. S. Apache Tomcat. `http://tomcat.apache.org/`.

[14] GROSSMAN, J. Seven Business Logic Flaws That Put Your Website at Risk. `http://www.whitehatsec.com/home/assets/WP_bizlogic092407.pdf`, September 2007.

[15] GUHA, A., KRISHNAMURTHI, S., AND JIM, T. Using static analysis for Ajax intrusion detection. In *Proceedings of the 18th international conference on World wide web* (2009), ACM New York, NY, USA, pp. 561–570.

[16] HALFOND, W., AND ORSO, A. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the International Conference on Automated Software Engineering (ASE)* (November 2005), pp. 174–183.

[17] HUANG, Y.-W., YU, F., HANG, C., TSAI, C.-H., LEE, D., AND KUO, S.-Y. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the International World Wide Web Conference (WWW)* (May 2004), pp. 40–52.

[18] JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2006).

[19] Java pathfinder. `http://javapathfinder.sourceforge.net/`.

[20] KLEIN, A. Cross Site Scripting Explained. Tech. rep., Sanctum Inc., June 2002.

[21] KREMENEK, T., TWOHEY, P., BACK, G., NG, A., AND ENGLER, D. From Uncertainty to Belief: Inferring the Specification Within. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)* (November 2006), pp. 161–176.

[22] LIVSHITS, V., AND LAM, M. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the USENIX Security Symposium* (August 2005), pp. 271–286.

[23] MARTIN, M., AND LAM, M. Automatic Generation of XSS and SQL Injection Attacks with Goal-Directed Model Checking. In *Proceedings of the USENIX Security Symposium* (July 2008), pp. 31–43.

[24] MICROSYSTEMS, S. Java Servlet Specification Version 2.4. `http://java.sun.com/products/servlet/reference/api/index.html`, 2003.

[25] MIDDLEWARE, O. W. O. S. ASM. http://asm.objectweb.org/.

[26] NGUYEN-TUONG, A., GUARNIERI, S., GREENE, D., AND EVANS, D. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the International Information Security Conference (SEC)* (May 2005), pp. 372–382.

[27] NIMMER, J., AND ERNST, M. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proceedings of RV'01, First Workshop on Runtime Verification* (2001).

[28] OPEN SOURCE SOFTWARE. SourceForge. http://sourceforge.net.

[29] PALEARI, R., MARRONE, D., BRUSCHI, D., AND MONGA, M. On race vulnerabilities in web applications. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (July 2008).

[30] PIETRASZEK, T., AND BERGHE, C. V. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detections (RAID)* (2005), pp. 372–382.

[31] SELENIUM DEVELOPMENT TEAM. Selenium: Web Application Testing System. http://seleniumhq.org.

[32] SPETT, K. Blind SQL Injection. Tech. rep., SPI Dynamics, 2003.

[33] SU, Z., AND WASSERMANN, G. The Essence of Command Injection Attacks in Web Applications. In *Proceedings of the Annual Symposium on Principles of Programming Languages (POPL)* (2006), pp. 372–382.

[34] TAN, L., ZHANG, X., MA, X., XIONG, W., AND ZHOU, Y. AutoISES: Automatically Inferring Security Specifications and Detecting Violations. In *Proceedings of the USENIX Security Symposium* (July 2008), pp. 379–394.

[35] VISSER, W., HAVELUND, K., BRAT, G., PARK, S., AND LERDA, F. Model Checking Programs. *Automated Software Engineering Journal 10*, 2 (Apr. 2003).

[36] XIE, Y., AND AIKEN, A. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the USENIX Security Symposium* (August 2006).

Notes

[1]As a consequence of that, JPF includes constraints that are no longer relevant to the current execution into the application's state, preventing it from detecting otherwise equivalent states.

[2]Note that by using the simple strategy of removing all constraints that reference no longer live variables, we might potentially lose some of the implied constraints in the PC. This can reduce the effectiveness of the reduction of the state space, but it does not interfere with the soundness of the analysis.

[3]The names of the variables are generated as explained in Section 4.1.

[4]When session data is accessed on a path, the PCA records that fact, along with the key that was used. This is done by storing the item *session.<key>* in an attribute of the memory location that holds the reference to the object. The information is then propagated by JPF with each bytecode instruction that accesses this memory location.

[5]A similar vulnerability was found by Waler in the JspCart application. We use Jebbo as a simpler example.

[6]Note that our tool works on Java bytecode rather than source code. Therefore, loop exit conditions are implicitly included, as they are implemented in terms of *IF* opcodes.

[7]The code for the *JspCart* application is located in the SourceForge repository under the name *B2B eCommerce Project*.