Automatic Generation of XSS and SQL Injection Attacks with Goal-Directed Model Checking

Michael Martin
Computer Science Department
Stanford University
mcmartin@cs.stanford.edu

Monica S. Lam
Computer Science Department
Stanford University
lam@cs.stanford.edu

Abstract

Cross-site scripting (XSS) and SQL injection errors are two prominent examples of taint-based vulnerabilities that have been responsible for a large number of security breaches in recent years. This paper presents QED, a goal-directed model-checking system that automatically generates attacks exploiting taint-based vulnerabilities in large Java web applications. This is the first time where model checking has been used successfully on real-life Java programs to create attack sequences that consist of multiple HTTP requests.

QED accepts any Java web application that is written to the standard servlet specification. The analyst specifies the vulnerability of interest in a specification that looks like a Java code fragment, along with a range of values for form parameters. QED then generates a goal-directed analysis from the specification to perform session-aware tests, optimizes to eliminate inputs that are not of interest, and feeds the remainder to a model checker. The checker will systematically explore the remaining state space and report example attacks if the vulnerability specification is matched.

QED provides better results than traditional analyses because it does not generate any false positive warnings. It proves the existence of errors by providing an example attack and a program trace showing how the code is compromised. Past experience suggests this is important because it makes it easy for the application maintainer to recognize the errors and to make the necessary fixes. In addition, for a class of applications, QED can guarantee that it has found all the potential bugs in the program. We have run QED over 3 Java web applications totaling 130,000 lines of code. We found 10 SQL injections and 13 cross-site scripting errors.

1 Introduction

As more and more business applications migrate to the Web, the nature of the most dangerous threats facing users has changed. Web applications are typically written in languages that make classic exploits like buffer overruns impossible, but new infrastructures bring new vulnerabilities. Two of the most popular attacks in this domain are SQL injection and cross site scripting (XSS) [12]. This paper presents a practical, programmable technique that can automatically generate attacks for large web-based applications. The system also shows the statements executed over the course of the attack. This information can be used by application developers to close these security holes.

Many commercial systems, including Cenzic's Hailstorm [7] and Core Security's Core Impact [9], rely on *black-box testing*. In black-box testing of web applications, the tester only has the level of access available to any external attacker—that is, it may only make HTTP requests and examine the responses. This approach has the advantage that any such analysis is independent of the target application's implementation language, making it ideal for broad deployment. However, it cannot take advantage of the logic of the program; it may not be efficient, and it cannot provide any guarantee on coverage.

This paper presents a system called QED that automatically finds attack vectors for a large class of vulnerabilities in web applications written in the same application framework. This system is based on the approach of *concrete model checking*. This is a verification technique based on systematic exploration of a program's state space. It is an attractive approach to security problems because not only can it conclusively find vulnerabilities, if a systematic exploration proves exhaustive, it can prove that no vulnerabilities exist. However, this technique is generally not feasible for large, real-life programs. In addition, a web application continuously accepts inputs, so it seems impossible on the surface to exhaust all possible paths. To make QED a practical tool that works on real programs, we built the system based on the design principles listed below.

1. Many web application vulnerabilities, such as SQL injection and cross-site scripting, can be generalized as taint-based problems. By focusing on this class rather than one vulnerability at a time, the QED system is much more general. Users can specify taint-based vulnerabilities in a language called PQL [22]. In fact, PQL extends beyond even taint-based analysis as it includes execution patterns involving any sequence of methods on a set of objects that is describable via a context-free language.

   Users can use QED for finding different vulnerabilities, and even vulnerabilities that are specific to their own applications. It is very important that ordinary developers be able to generate these analyses on their own.

2. Today, application frameworks are heavily used in web application development as they greatly reduce software engineering time. We advocate extending the notion of frameworks beyond software development to include code auditing. Exploiting higher level semantic information about the framework makes it possible to generate more effective static analyses. Furthermore, by abstracting away the guts of a framework, we can concentrate our model checker's effort on the application code itself. This abstraction step needs only to be performed once for each framework, as the abstracted code is reusable. For this research, we have picked the following popular core frameworks for web applications:

   - Java servlets [27], which is a standard extension to the Java platform for writing web applications.
   - JSPs (Java Server Pages) [28], which allow page design to be commingled with database accesses.
   - Apache Struts [1], which is a web application framework that uses the model-view-controller paradigm. In this paradigm, a controller decouples the data model from the user view so they can easily be changed independently.

   Any Java web application intended for deployment in a standard application server conforms to the servlet specification. If a Java web application also uses JSP or Struts, our framework will take advantage of the additional semantics as well.

   To demonstrate the effectiveness of this approach, we report the result of applying our tool across three different Java web applications developed on this framework.

3. In model checking, we are simulating the program execution on candidate input sequences. QED uses JPF, the Java PathFinder model checking system [29], to do this. It is important that we concentrate the model checking time on sequences that are likely to identify vulnerabilities. Based on the query, QED automatically compiles user-supplied queries into static analyses for the web application that prune out input sequences that are guaranteed not to expose any vulnerability. The static analysis generates a set of input vectors. If it is small, this set can be tested exhaustively; if it is not, the static analysis's results—directed by the user's query—direct the checker to test more promising results first.

1.1 Contributions

This paper makes the following contributions.

- *A session-based model for user input in web applications.* Much work in testing web applications focuses on either analyzing individual pages [31] or simulating a browser user with a sophisticated spider [3]. We present a technique that bases its user model on data flow information across requests in a session. This helps restrict the search space while also exposing possible vulnerabilities that a spider or nonmalicious end user might never produce.

- *A programmable approach to checking event-driven applications.* QED is extremely flexible; its concept of vulnerability is merely "anything that matches a specification", and the permissible specifications include any context-free language of method calls on a consistent set of run-time objects. Though this paper focuses on taint vulnerabilities in web applications, the technique generalizes to other error patterns as well as other event-based systems such as GUI applications or file systems.

- *A model-checking framework to systematically explore standard Java web applications.* We have implemented a simulated environment for the Java PathFinder model checker that will systematically explore programs based on the Java Servlet Specification. We have refined it further to work more effectively with the popular Apache Struts framework.

- *Experimental validation of our approach.* We supplied specifications for two major security vulnerabilities (cross-site scripting and SQL injections) and applied the QED system to three large Web applications. These applications totaled roughly 130,000 lines of non-library code. QED detected 10 SQL injection vulnerabilities and 13 XSS vulnerabilities.

1.2 Paper Organization

Section 2 describes the class of vulnerabilities of interest. Section 3 describes how we apply model checking to web applications to generate the attack vectors and get the execution trace. Section 4 describes how we use static analysis to reduce the search space of model checking. Section 5 demonstrates the QED algorithm step by step on an example application. Section 6 details experimental results. Section 7 discusses related work, and Section 8 concludes.

2 Problem Statement

Our algorithm accepts a web application and a vulnerability specification, then generates a set of attack path components with corresponding execution traces. This section describes the class of applications and vulnerabilities our system addresses.

2.1 Taint Vulnerabilities

SQL injection and cross-site scripting are both instances of *taint vulnerabilities*. All such vulnerabilities are detected in a similar manner: untrusted data from the user is tracked as it flows through the system, and if it flows unsafely into a security-critical operation, a vulnerability is flagged. In SQL injection, the user can add additional conditions or commands to a database query, thus allowing the user to bypass authentication or alter data. With XSS, an attacker can inject his own HTML (including JavaScript or other executable code) into a web page; this is exploitable in many ways, up to complete compromise of the browser. In the so-called “reflection attack” [12] XSS is used by a phisher to inject credential-stealing code into official sites without having to redirect the user to a copy of the site. This means that any security credentials will be valid on the attack site, and even whitelisting will not prevent the attack.

Given the gravity of the vulnerabilities, we would like to eliminate their existence before deploying our applications. Some of these vulnerabilities can be subtle, however. It is not sufficient to just consider URLs in isolation because an attack may consist of a sequence of URLs. Consider a scenario with the example web application in Figures 1 and 2. An attack on this application can go as follows: the attacker sends the victim an email containing the URL `http://example.com/search_begin.jsp?s=<script...` where the `s` parameter carries a JavaScript payload crafted to log users’ keyboard entries. The victim clicks on the link. Since this is the user’s first interaction with `example.com`, a new session is created by the web server, and when the JSP checks the value of `login`, it finds nothing. It thus stores the search string in the session and generates a redirect page to `search_login.jsp`. That page then generates an error and requests login information. However, at this point it echoes the value from the session blindly, thus injecting the script and allowing the attacker to log the user’s password. This example illustrates that we need to analyze more than just individual requests to be sure we have found all vulnerabilities in a web application.

```
<html>
<head>
<% HttpSession s = getSession();
   if (s.getAttribute("login") == null) {
     s.setAttribute("text",
                    getParameter("s"); %>
  <meta http-equiv="refresh"
      content="10;URL=search_login.jsp">
</head>
<body></body>
</html>
<% } else { %>
<!-- rest of page... -->
```

Figure 1: Snippet from `search_begin.jsp`.

```
<html>
<body>
<h1>Login required</h1>
<p>To search for
   <%=getSession().getAttribute("text")%>,
   you must first log in.</p>
<form>
<!-- rest of page... -->
```

Figure 2: Snippet from `search_login.jsp`.

We model the behavior of a web application as a series of request-response events; each URL corresponds to an HTTP request, and this request is processed to produce a response. We may characterize an attack vector by a sequence of URL requests in a session where untrusted input data propagates into security-critical operations.

2.2 Domain of Web Applications

We model a web application as a reactive system that operates on a *session* at a time. A session consists of a series of *events*, with each event being an HTTP request submitted by the same user. Note that while the request originates from the same user, its contents may actually be manipulated by an attacker. We do not place any restriction on the ordering of events. In particular, it is not necessary that requests be constrained by the links available on the last page viewed. This is necessary because an attacker can construct and send malicious requests directly. This also argues against using web-spider techniques to collect potential attack vectors.

In response to an event, a web application may modify the *session data*. This is information that is user-specific but maintained temporarily on the webserver over the course of a user's interaction with the machine. In a webserver, a separate data structure is normally maintained for each user, and cookies or special arguments would be set to match each users to their sessions.

Sessions are assumed to be independent of each other. An attack may consist of a sequence of events within a session, but cannot span multiple sessions. Our reasoning here is that any attack usable against another user should also be usable against oneself, and so the attack will still manifest.

2.3 Vulnerability Specifications

The set of taint-based vulnerabilities addressed by our technique consists of all attacks that match the following pattern:

1. Untrusted data is read in from some *taint source*, such as a user-controlled file, URL request, cookie value, or network source. It may subsequently be stored in arbitrary objects and passed in and out as parameters or returned results.

2. Some methods may derive new objects from old. Some of these, if passed an untrusted object, will produce an untrusted object. Examples include methods that parse a request and create subobjects from the untrusted data, or methods that create larger strings by appending characters to the untrusted data. We call these methods *propagators*.

3. No untrusted data, whether from the original taint source or derived via propagators, may be used in any *taint sink*, such as a database access routine.

4. The previous rule does not apply if the object has been passed through one of several *sanitizers*, that quote or escape the contents of the object.

This is an abstraction of the general problem of *information flow control*. Information is tracked from the source, through propagators, until it either hits a sanitizer and becomes safe, or hits a taint sink and possibly does damage. Once the tracker can confirm that all dangerous data only reaches sanitizers, a proof of the correctness of these sanitizers will suffice to prove the correctness of the entire program.

Our vulnerability specification consists of four patterns, one for each of the previously enumerated components. These patterns are expressed as PQL queries. PQL is a powerful specification language that permits one to specify patterns of events on objects in a manner similar to program snippets. It permits subqueries to be defined and then matched against as well. We can exploit this by defining the components of our specification as subqueries and then linking them together with a generic main query that works for any taint problem.

```
query source(object * x)
matches
    HttpServletRequest.getParameter(x)
 |  x = Cookie.getValue();

query prop(object * x, object *y)
matches
    (StringBuilder) y.append(x)
 |  y = (StringBuilder) x.toString();


query sink(object * x)
matches
    JspWriter.print*(x)
 |  JspWriter.write(x, ...);
```

Figure 3: XSS vulnerability specification.

A simple example for XSS in JSPs is shown in Figure 3. All three of its defined subqueries are a logical OR between individual method calls. Its taint sources, `HttpRequest.getParameter` and `Cookie.getValue`, are defined for all Java web applications [27]. Likewise, the `JspWriter` class in the taint sink is defined in the JSP specification [28]. PQL permits method names to be regular expressions, and so we collect all `print` and `println` method calls within a single clause.

The propagation rules in the `prop` query handle string concatenation in Java 1.5. In the full specification, other versions of Java and other modes of string propagation are also handled. These are simply added as additional OR clauses; we omit them here for clarity.

Care must be taken when developing the specification—missing a propagator may lead to false negatives in the final result, while missing sanitizers is likely to lead to many false positives. A suitably crafted general specification, however, can apply to many applications directly or with only minor modifications to specify details and application-specific sanitizers. Furthermore, the operation of the model checker will suggest which modifications need to be made to refine the query.

Due to the design of the Java libraries, web application queries will rarely need to explicitly specify sanitizers. Java's `String` class is immutable, and it is also the class that represents the beginning and end points of any web transaction. Since the sanitization process will generally create an entirely new `String`, this freshly created object would thus be considered safe. This is another reason we must be particularly careful not to miss any propagators: any propagator we fail to specify will be treated as a san-

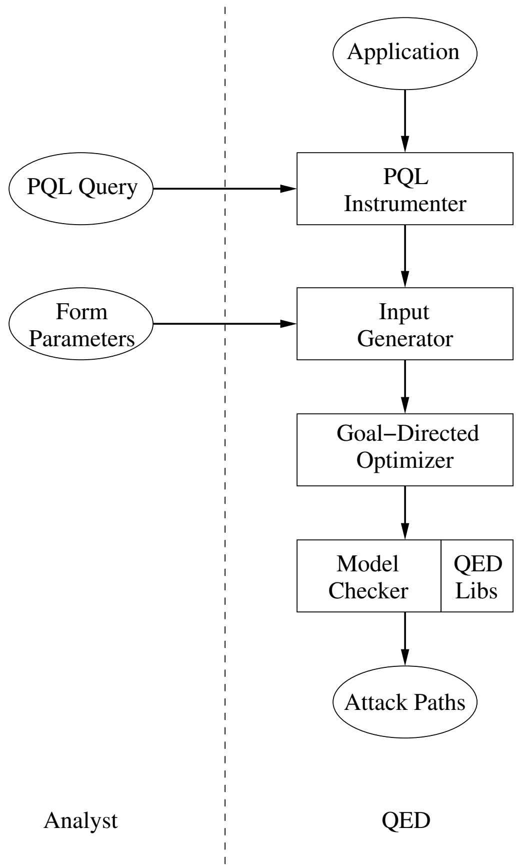

Figure 4: QED architecture. User-supplied information is on the left.

itizer.

It is also possible that a sanitizer might perform its transforms using propagator methods. This would require explicitly marking the result as sanitized. However, this situation never occurred in our experiments. We never found it necessary to explicitly specify sanitizers, and our XSS query worked unmodified with all applications.

2.4 PQL Instrumentation and Matching

The vulnerability specification is translated by the PQL compiler into a set of instrumentation directives. When applied to the target application, they weave in monitoring code to detect matches to the query, and to report on the objects involved [22]. When a match is found by the monitor, it signals the model checker to report that a failure condition has been found. If no match occurs, the model checker's backtracking mechanism will also roll back the matching machinery to the appropriate state.

3 Input Generation

In this section, we describe how QED enumerates attack vectors for a target web application. An analyst must provide two components: the PQL query specifying the vulnerability, and a set of input values for any form parameters. Given these, QED will do the rest. A diagram of the process is shown in Figure 4.

The input application is first instrumented according to the provided PQL query, as described in Section 2.4. The instrumented application is then combined with a custom, automatically generated harness. This is a program that will systematically explore the space of URL requests. Each URL consists of a page request (the *path*, covered in Section 3.1), and an optional set of input parameters (the *query*, discussed in Section 3.2). The harnessed application is then fed to the model checker, along with stub implementations of the application server's environment. The results of that model checker correspond directly to sequences of URLs that demonstrate the attack paths.

We may also optionally improve our search by optimizing the harness before the model checking step; we discuss these refinements in Section 4.

3.1 Generating Page Requests

An attack path is a sequence of URLs, each of which consists of a page request (the *path*) and a set of input parameters (the *query)* [4]. The web application translates a URL into a method invocation with a set of parameters.

Thus, a URL corresponding to our sample JSP earlier:

$$\texttt{http}://\texttt{www.example.com/search.jsp?s} = \texttt{foo}$$

would translate into the method invocation

$$\texttt{org.apache.jsp.search\_jsp.doGet(req, resp)}$$

where the call `req.getParameter(x)` yields the value "`foo`" if `x` is "`s`", and yields `null` otherwise. The `resp` parameter represents the response to be returned.

There is a simple correspondence between a URL and a method invocation. We refer to the method invocation as an *event*. An *event* consists of:

1. a reference to an event handler. The event handler corresponds to the path of the URL. It identifies the name of the Java method to be invoked when a matching URL is received.

2. event handler parameters. These typically correspond to the query part of the URL. They provide extra parameters used by the handler, and generally carry the more free-form data. They may include

cookies or information supplied by a user when filling out forms. They may also contain the payload that an attacker wishes to inject into the system. Thus, it is very important to model these inputs carefully.

Most Java web applications are developed using a framework that makes explicit the set of URLs it accepts and its corresponding event handlers. Our system currently handles three popular frameworks, as discussed below.

- *Servlets* are the most basic form of server-side Java, and are the lowest level of abstraction available. Any Java web application intended to be run in a standard application server must ultimately use this specification. Individual servlets are Java classes that implement a well-specified API [27]. The URL-to-servlet mapping is specified by an XML file as part of the application's metadata. QED simply interprets the XML file to determine the list of event handlers in the application.

- *Java Server Pages*, or *JSPs*, provide a PHP-like interface to Java [28]. They are compiled by a JSP compiler such as Jasper into servlets. The URL-to-servlet mapping in this case is specified by a transformation of the JSP's path in the file system, which generates the class name.

- *Apache Struts* is a popular application platform built on top of JSPs and the core servlet specification [1, 13]. It implements its own `Action` API similar to the servlets API, but which forwards to JSP files for actual HTML output. A URL in a Struts application thus maps to two calls in sequence; a call to an `Action`'s entry point, and a call to the associated JSP's entry point. These mappings from URLs to `Actions` and JSPs are specified in an XML file in a manner similar to the specifications for servlets.

For each of these, QED can produce a comprehensive list of paths understood by the application. To test each sequence, it does, by default, a breadth-first search through them - first checking all sequences of length 1, then all of length 2, and so forth. This has no obvious termination condition, however; our optimizations and heuristics in Section 4 provide limits.

3.2 Parameters to Event Handlers

In Java web applications, data from the user is represented by a set of key-value pairs mapping strings to strings. Applications conforming to the Java Servlet Specification use a method called `getParameter` to retrieve a value for a given key. QED rewrites methods corresponding to taint sources to call out to the model-checker, indicating to the model checker that there is non-determinism associated with the returned value of the method. The model checker will cycle through the possible values, including the option that no such key was provided by the user.

We rely on the analyst to provide a sufficient pool of values to test the application. It would be infeasible to test every possible string that could be supplied to the event handlers, but it is also not necessary. Our goal is merely to show that it is possible for data from a taint source to reach a taint sink. If a controlled string is displayed, this is a vulnerability.

In cases where the contents of an input string do matter, the data are often expected to be in a certain form: if they do not conform to the expected type, some paths may not be executable. For our experiments, we supplied one of the common default types used by web applications in general: integers, booleans ("yes", "true", etc.) and generic strings. We also included the null object to represent the lack of an argument.

Applications may also require application-specific "magic" values that influence control flow. The most common case for this is an `action` variable or similar, which holds one of several values depending on the value of a list box or similar. In such cases, QED can usually extract the information we need via a constant propagation analysis; this will tell us if an argument from the query string is compared against constant strings. By enumerating these strings and ensuring they are possible values for our keys, we search the input space more exhaustively.

It would be possible to combine this work with an analysis similar to EXE [6] to determine a set of inputs that would exercise all predicates in the web application. For our experiments so far, however, we have found that even our simple constant-propagation analysis is overkill. Almost all data read from the user is processed and dumped directly into a data sink. In these circumstances the control flow cannot change based on input.

4 Goal-Directed Optimization

In this section, we present several optimizations to reduce the search space of model checking. The key insight is that the we should not treat all URL sequences as equally likely to yield a new vulnerability, since we may have already checked a shorter, equivalent sequence. Since we check in increasing order of length, any match it finds will have already been discovered. There are four principles we apply to focus the search:

- *The final request in the sequence must finish the demonstration of a vulnerability* (Section 4.1).

- *Every request must, directly or indirectly, influence the final result* (Section 4.2).
- *No sequence ever repeats a request* (Section 4.3).
- *A match can only occur in a sequence if there are objects that would satisfy that match participating in that sequence* (Section 4.4).

4.1 Filtering Final Events

QED's model checker searches through candidate sequences in length order. This means that for any given vulnerability in the code, the shortest demonstration of it will appear first. If it does not, any possible vulnerability would have already been shown before the final request was processed, so a prefix of the sequence would suffice, and will in fact have already been checked. This condition is thus stronger than a simple breadth-first search, which can only confidently eliminate sequences with a prefix corresponding to a known vulnerability.

To perform the final event filter, we need two pieces of information. First, we need to know which method calls in the application can in fact complete a match. For a taint problem, this is straightforward, as it is any method listed as a taint sink. For PQL in general it may be necessary to perform a simple control-flow analysis on the query to determine the set of events that can occur last.

We then need to determine which URL requests can lead to match completion. We do this by writing a simple harness program that calls each entry point in the application in turn. We then compute a call graph of this harness and determine which entry points can eventually call a match-completing method.

Any sequence which does not end in a call to one of these entry points is guaranteed to not affect the final result, and thus may be discarded.

4.2 Eliminating Redundant URL Sequences

HTTP is a stateless protocol. Web applications maintain state across requests either client-side with cookies or server-side with session data. We treat cookies as a source of user input, as cookie information may be forged, deleted mid-session, or otherwise tampered with. Session information remains under the control of the server and can thus be tracked more precisely.

The motivation behind this optimization is that this mechanism is the sole form of data-flow through the session. If there is no data-flow contributed by a part of a candidate sequence, we need not include that part. Furthermore, since we are checking in increasing order of length, removing this redundant part of the sequence produces a sequence that we have already either checked or proven irrelevant.

To perform this optimization we need a way to characterize the cross-request data-flow. We do this via a *dependence relation*: an event handler m_1 *depends* on another event handler m_2 if m_1 can potentially read the data written by m_2. To compute the dependence relation, we must determine the flow of data within a session.

The Java Servlet Specification provides an explicit API to capture this. Data are passed between handlers via a special object of type `javax.servlet.HttpSession`. This session object functions as a string-to-object map. For each request, we determine what string values can be used as keys to the map for reads and writes. This information is available via a call graph analysis as in Section 4.1, supplemented with pointer and constant-propagation information to determine which string values may be used as keys. If a nonconstant string is used as a key, we assume that handler may access anything in the session.

With this information we can compute the dependence relation by treating each key as a storage location and determining def-use information. We then take the transitive closure of the dependence relation, and eliminate any sequence in which there are requests that do not influence the final request.

4.3 Removing Repetitive Cycles

If the dependency relation is cyclic, there will be a countably infinite number of possible candidates to test. To keep the test sequence finite, we restrict our sequences to only call any given entry point once.

This heuristic would need to be refined for web applications where one physical page serves as multiple logical pages (controlled, say, by some `action` parameter); however, this situation did not arise in any of our experiments.

4.4 Statically Eliminating Sequences

We further reduce the search space by using a static analysis to prune off sequences that cannot possibly match our query. This is especially important for sequences that use a large number of widely variable parameters, as eliminating a single sequence can translate into thousands or even millions of candidates that need not be checked. The algorithm is described below.

1. QED constructs a new harness for the application that iterates through all sequences that pass the preceding three criteria. The harness defines a method for each input sequence, and the method calls the

entry point for each of the URL request in the sequence.

2. QED translates the PQL query specifying the defect of interest into a sound context-sensitive interprocedural analysis that determines if the query can be satisfied. QED applies the analysis to the harness to find the methods (input sequences) that can potentially generate a match. The algorithm used has been been described in a previous paper [22]. This analysis tracks pointers in a context-sensitive but flow-insensitive manner. The analysis is sound—no approximation done by the pointer analysis will produce false negatives. All sequences found by the analysis to be incapable of generating a match may be ignored without compromising the soundness of the model checker.

The success of this step hinges on both the precision and the conservativeness of the pointer analysis used. An overly imprecise analysis will not be able to eliminate any candidates, while a non-conservative analysis will prune away candidates that might be valid. The QED system applies the context-sensitive, conservative, interprocedural, and inclusion-based analysis of Whaley and Lam [32], along with improvements by Livshits et al. to handle reflection [21]. The results of this analysis are stored in a deductive database which QED consults throughout the optimization process [19].

5 Example

We will now show the operation of this algorithm by detecting an XSS vulnerability in a simple three-page web application. The pages in this application are as follows:

- `search.jsp`, which presents a search form to the user and sends the results on to `searching.jsp`.

- `searching.jsp`, which reads a search parameter `s` and stores it in the session. The display is a simple timed redirect to `result.jsp`.

- `result.jsp`, which prints the results of the search. It also echoes the initial input, retrieved from the session. This represents a cross-site scripting vulnerability.

For our example, we use the stock XSS vulnerability query from Figure 3. The PQL instrumenter will transform the application, tracking all calls to sources, sinks, and propagators.

For our model environment, we will only concern ourselves with whether or not an argument is present, so we will set `null` and "`SampleString`" as our input pool. QED will generate a test harness for the application, providing these values as plausible results for the sources, and calling all possible sequences of events. Since we only concider non-repeating sequences, there are ten: three of length 1, six of length 2, and one of length 3. The entry points for these events will simply be the `doGet` methods on the classes corresponding to each JSP.

In the optimization step, the final events filter has no effect for this query. The sink for the XSS query is `JspWriter.print()`, which all three pages call as part of their output generation.

The dependency criterion is much more fruitful. Our session-based def-use analysis concludes that `searching.jsp` writes the session, while `result.jsp` reads it with the same key. This yields a dependency relation with one fact, and the dependency criterion eliminates all but four sequences—each page alone, and the [`searching.jsp`, `result.jsp`] sequence. Factoring in the choice of `s` in `searching.jsp`, this yields a grand total of five test runs.

The pointer analysis phase shows that `searching.jsp` is the only request handler with a source in it, thus eliminating two of the length-one sequences immediately. It can then show that, as `searching.jsp`'s parameter is only fed into a session and the handler itself only emits constant strings, the lone `searching.jsp` request also cannot complete a match. Thus, for our example application, we are able to pre-prune every sequence of events but one. The only task remaining for the model checker is to demonstrate which values for `s`, if any, will actually produce a vulnerability.

The model checker will return the following sequence as a demonstration of an XSS attack path:

- `searching.jsp?s = SampleString`

- `result.jsp`

Despite the fact that a typical use case would derive its input from `search.jsp`, the page does not actually contribute anything to the vulnerability itself.

In general, the amount of search space that can be removed by our optimizations will depend on several factors. The number and prevalence of taint sinks is one; if there are more places where the path can end, there will clearly be more paths. However, the dominant factor will be the fan-out from the session data-flow. With a low fan-out, even a large number of sinks will not multiply unduly.

6 Experimental Results

We applied QED to three Struts-based web applications from the open-source repository Sourceforge. Basic information about these is shown in Figure 5. They are

Benchmark	Description	Lines of Code	Classes	Event Handlers	Dependency Pairs
PersonalBlog	Blogging software	17,149	132	15	0
JOrganizer	Address book	31,897	263	46	49
JGossip	Forum system	79,685	556	80	267

Figure 5: Applications used in the experiments. (The lines of code do not include library classes)

Benchmark	Non-redundant URL Sequences	Ends in SQL Sink	SQL Sessions	SQL Errors	XSS Sessions	XSS Errors
PersonalBlog	15	2	2	2	1	1
JOrganizer	356,358	260	153	8	86	3
JGossip	1,062,539	16,031	9,436	0	30	9

Figure 6: Analysis results.

listed in order of their size. For each application, we list the number of classes defined in the program, the size of the application itself (not counting library classes), and the total number of event handlers specified by the application's deployment metadata. The last column of Figure 5 shows the number of dependency pairs found by our dependence analysis described in Section 4.2.

We used QED to locate both cross-site scripting and SQL injection vulnerabilities in each of these applications. Each of these applications depends on a database backend. The JGossip application used JDBC directly; the other two used object persistence libraries that we modeled as stubs. All three applications, since they are Struts-based, rely on JSPs for their output, and so the XSS analysis dealt primarily with those.

Figure 6 presents some measurements of our experiment. The first column (Non-redundant URL Sequences) lists the number of sessions whose URLs are not repeated and not redundant according to their data dependencies. Personalblog does not have cycles in its dependence graph, so it is possible to exhaustively model check the program by testing the specified number of input sequences. The next column (Ends in SQL Sink) shows the result of applying the full redundancy elimination analysis algorithm presented in Section 4.2. The next column (SQL Sessions) shows the number of sessions that needs to be checked after the feasibility analysis from Section 4.4 is also taken into account. The next column gives the number of SQL injections QED discovered.

The final two columns provide similar information for XSS. We do not provide an equivalent to the "Ends in SQL Sink" column because the XSS sink is HTML output, and so every HTTP response by definition includes a sink. Between SQL injection and cross-site scripting, we thus cover both rare and common sinks in our applications.

For comparison, even if we restrict ourselves to non-repeating URL sequences, the naïve approach of Section 3 would test a number of sessions proportional to the factorial of the number of event handlers. In JGossip, this is approximately 10^{120} sequences.

6.1 PersonalBlog

The PersonalBlog system is a web application based on Struts and the Hibernate 2 object persistence system [2]. It makes no interesting use of session objects, so there are no dependencies between handlers. Thus, the dependence analysis shows that we can consider each event handler in isolation without compromising any guarantee on security. Since there are only 15 event handlers in the program, and each request has few parameters, the model checker can run through all the cases quickly.

QED found one XSS attack vector and two SQL attack vectors. Note that a single vector can have multiple vulnerabilities. In this case, one of the SQL vectors has two SQL injection possibilities. Thus, there are actually three SQL vulnerabilities that we have found. The static analysis in this case was accurate in identifying all the vulnerabilities, without generating any false positives. The model checker generates the input vectors and a program execution trace showing the details of their existence.

The results of running PQL itself, as a dynamic checker, on PersonalBlog has also been reported previously [22]. Not only did QED find all the vulnerabilities previously identified, it found an additional one. This discrepancy is due to QED having a more inclusive specification than in the previous work, tracking information from HTTP headers and not just from the URL proper.

6.2 JOrganizer

JOrganizer is a personal contact and appointment manager of moderate size. Access to the backing database is managed within the application by an "Object Query Language" that reduces directly to SQL, much like Hibernate 2.

The application has 46 event handlers in total. The dependence analysis shows that there are 49 pairs of dependent event handlers. The dependence relations are cyclic, which means that we will have to restrict our attention to acyclic sequences to keep the test space finite.

QED then further focuses the model checking effort by using information specific to each vulnerability. We found that 15 of the event handlers cannot touch the database at all, and thus cannot be final events for SQL injection. Furthermore, none of the single-event sequences exhibits a SQL vulnerability. The reason is that no event is allowed to touch the database unless it is preceded by a "log-in" event. Our analysis shows a dependence between these events and the "log-in" event. QED ignores the independent pairs, and keeps testing sequences with first a log-in event and then a database access event.

QED is able to iterate through all the filtered, non-redundant, and non-repeating sequences in this case, finding three XSS vulnerabilities and eight SQL vulnerabilities.

6.3 JGossip

JGossip is a large application with nearly 80,000 lines of code in 80 actions. There are many cyclic dependencies among event handlers in JGossip. Even if we restrict the sequences under consideration to non-redundant and non-repeating events, over a million sequences still remain. Furthermore, within these sequences, many requests used enormous numbers of input parameters. One event had 15 parameters, which, with a pool of 5 possible inputs per parameter, would generate over 30 billion test cases simply for that one URL. For event handlers such as those we restricted our model checker's input pool to two possibilities per parameter, lowering the number of test cases per handler to a more manageable 32,000.

Next QED tries to reduce the number of candidate vectors based on the vulnerability specification. For SQL injection, the taint sink method is database queries. A majority of the 80 actions touch the database. However, our static feasibility analysis shows that only seven of these database accesses may touch tainted objects. Thus we have only 7 final events to consider. Of the seven, five have no dependency chains longer than length two. They are responsible for a total of 37 potential attack vectors, and they are all of length 2. The remaining two have many dependencies, and the static analyzer can only narrow them to 9,436 candidate attack vectors. Once parameters are factored in, this still yields hundreds of millions of candidates to check, so there are still too many to consider. At the end, we managed to check only all the seven sequences with a single taint-sink event, and the all 37 sequences of length 2. The model checker found no SQL vulnerability.

As it happens, this lack of SQL injections is unsurprising, because JGossip is constructed to be independent of its database backend. As such, all of its database requests are ultimately constant strings; it uses a hash table to look up which strings are appropriate for the appropriate action, based on the SQL dialect used by the backend. Since all SQL queries end up being constant strings, this suggests that no injections are possible; however, its use of hash tables forced the program analyzer to make conservative approximations on seven of the actions, thus leading to the need for a model checking step.

The tests for XSS were much more straightforward; all of the actions corresponding to possibly dangerous output JSPs had few inputs and few dependencies, leading to a grand total of only 30 sessions to check. The XSS vulnerabilities so found were also located immediately, since session data did not affect their outputs.

6.4 Experimental Summary

The three web applications in our experimental study illustrate a spectrum of effects we can get with QED. PersonalBlog shows an example where QED is able to prove that there are no vulnerabilities other than the ones found. By proving that the events have no dependencies, QED can simply check the URLs one at a time. JOrganizer shows that in the presence of dependencies, our analyses can greatly improve the effectiveness of model checking and provides good coverage. QED was able to check all the sequences without repeated URLs. Lastly, JGossip shows that model checking for really large programs remains a challenge. The static analyzer is useful as a way of directing the model checking to focus on sequences with higher payoffs.

7 Related Work

Systematic automated testing is not entirely novel, but it is also not commonplace. Our work was informed by both the FiSC system [34] and WebSSARI [17]. WebSSARI's approach is much different from QED's, in that it focuses on abstract interpretation of PHP code looking for violations of data flow control. QED, on the other hand, owes more of its design philosophy to FiSC. FiSC operated in an entirely different problem domain (filesystem correctness) and simply searched for evidence of errors rather than the cause. Its implementation was based on the CMC model checker [23] which is also much closer to our JPF-based system than WebSSARI's runtime solution.

The techniques described in this paper touch upon a wide variety of disciplines. Model checking is the most directly obvious of these. Our system uses the Java PathFinder system [29]. JPF was suitable for our system primarily due to the ability to directly run sizable Java applications as bytecode; this permitted us to treat our dynamic analysis as just another part of the application being checked. Classical model checkers such as SPIN [14] require a special specification language which abstracts the application greatly. Other model checking systems such as Bandera [8] also directly abstract the Java source, which complicates its utility for our purposes.

The more general field of bugfinding comprises an enormous amount of work. In recent years, web applications have received a good deal of attention due to their unique vulnerabilities and flaws. SABER is a static tool that detects flaws based on pattern templates [25]. Livshits and Lam made progress in creating a sound analysis on web applications that produced a usably low false positive rate [20]. The WebSSARI system, in its pre-model-checking work, allows the specification of taint-style data-flow problems on PHP-based applications, and systematically searches for dangerous information flows [16]. Nguyen-Tuong et al. use similar approaches, also for PHP [24]. In a more general context FindBugs attempts to locate a broad class of bugs in Java applications of all kinds [15], and the Metal system let the user specify state machines to represent error conditions [11]. The SQLCHECK system uses a much more precise technique to detect grammatical changes in commands as a result of user input [26]. The QED system provides a general analysis that the user specializes, while SQLCHECK is SQL-injection specific and FindBugs is a battery of unrelated analyses. Taint flow within an application is tracked incidentally, and only if the PQL specification demands it.

Our characterization of inputs, when combined with model checking, can be seen as a form of testing, and all testing techniques perform better with a better set of inputs. Some work has been done on systematically deducing inputs that will explore the state space of an application. Systems such as Korat [5] attempt to systematically produce only consistent inputs; this is rarely relevant to web applications, whose arguments can be nearly-arbitrary strings. Korat's general principle of deducing input sets from execution constraints, however, may still be applicable. Symbolic execution techniques, such as DART [10] and EXE [33], suitably adapted to deal with string and URL data, are more likely to be a fruitful adjunct to the techniques in this paper. Some work has been done already to provide these techniques for JPF but the results given seem to indicate that at present it scales only to smaller applications [30].

For the specific problem of cross-site scripting, recent work has focused on extending the DOM to permit browser extensions to block out any unauthorized scripts [18]. While, if fully implemented, this system will block out any possible attacks, it requires cooperation between both site authors and clients. Client-side protection is also of limited use against taint problems such as SQL injection that attack the server.

8 Summary and Conclusions

Security concerns regarding web applications are here to stay, and likely only to grow in importance. Cross-site scripting and SQL injection are two of the most popular kinds of vulnerabilities. This paper presented a technique called goal-directed model checking that can find attack vectors for these vulnerabilities automatically and efficiently. Armed with actual attack vectors and their corresponding execution trace, it is easier to convince the developers that it is necessary to change the code, and also to pinpoint how the problem can be fixed.

Our technique is implemented in a system called QED. Users can use the system for any taint-based vulnerability on Java applications developed using servlets, JSPs, or Struts. We applied QED to three programs and found errors in every one of them, yielding a total of 10 SQL injection and 13 XSS vulnerabilities. This result is worrisome, suggesting that there are plenty of security risks in using web applications.

This work also shows for the first time how we can combine techniques from three approaches to generate a useful and powerful system:

Sound, sophisticated program analysis. Sophisticated analysis based on context-sensitive pointer alias analysis is precise enough to use on production software, despite being conservative to retain soundness. Nonetheless, false positives are still bound to occur with a conservative analysis.

Dynamic monitoring. Dynamic analysis does not have false positives, but it can only spot problems that its input happens to trigger.

Model checking. Model checking has many advantages: it executes all the paths in a program; it has no false positives; it has no false negatives with respect to the set of possible inputs tried; it identifies actual attack vectors; and it can generates an execution trace for any input. However, it is too slow.

QED combines the advantages of all the three approaches. It uses sound analysis to optimize both dynamic monitoring and model checking, dynamic monitoring to follow the flow of taint, and finally model checking to generate the actual attack vectors.

Cross-site scripting and SQL injection are examples of errors that exist at the application layer and that are not due to simple language deficiencies like buffer overruns. We can expect to see many more varieties of errors that operate at this higher semantic level. This suggests that programmable systems like bddbddb, PQL, and QED are important so that developers can utilize the technology, without being analysis experts, for their own programs.

The widespread adoption of application frameworks in software development opens up a new opportunity for managing software complexity. These software frameworks should come with testing, model checking, static analysis, and dynamic monitoring submodules; they should be programmable and specialized for that framework. Perfecting them as part of the framework will put these advanced technologies in the hands of many more developers.

9 Acknowledgements

We are grateful to Benjamin Livshits and Christopher Unkel for their reviews of previous versions of this draft, and to Willem Visser, Peter Mehlitz, the rest of the JPF4 team, the TRUST project, and the anonymous reviewers for their encouragement and suggestions. This work was supported in part by the National Science Foundation under Grant No. UCB-0424422 and Grant No. 0326227.

References

[1] Apache Software Foundation. Apache Struts. `http://struts.apache.org`, 2002.

[2] Bauer, C., and King, G. *Hibernate in Action*. Manning Publications, New York, NY, 2004.

[3] Benedikt, M., Friere, J., and Godefroid, P. VeriWeb: Automatically Testing Dynamic Web Sites. In *Proceedings of the Alternate Track of the 11th International World Wide Web Conference (WWW'02)* (2002).

[4] Berners-Lee, T., Fielding, R., and Masinter, L. RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax. `http://www.ietf.org/rfc/rfc2396.txt`, 1998.

[5] Boyapati, C., Khurshid, S., and Marinov, D. Korat: Automated Testing Based on Java Predicates. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis* (2002), pp. 123–133.

[6] Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., and Engler, D. R. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)* (2006).

[7] Cenzic. Hailstorm. `http://www.cenzic.com/`.

[8] Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Păsăreanu, C. S., Robby, and Zheng, H. Bandera: Extracting Finite-State Models from Java Source Code. In *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering* (2000), pp. 439–448.

[9] Core Security. Core impact overview. `http://www.coresecurity.com/?module=ContentMod&action=item&id=32`.

[10] Godefroid, P., Klarlund, N., and Sen, K. DART: Directed Automated Random Testing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (2005), pp. 213–223.

[11] Hallem, S., Chelf, B., Xie, Y., and Engler, D. A System and Language for Building System-Specific, Static Analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)* (2002), pp. 69–82.

[12] Hoglund, G., and McGraw, G. *Exploiting Software: How to Break Code*. Addison-Wesley Publishing, 2004.

[13] Holmes, J. *Struts: The Complete Reference*. McGraw-Hill/Osborne, Emeryville, CA, 2004.

[14] Holzmann, G. J. The Model Checker SPIN. *Software Engineering 23*, 5 (1997), 279–295.

[15] Hovemeyer, D., and Pugh, W. Finding Bugs is Easy. In *Proceedings of the Onward! Track of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2004), pp. 132–136.

[16] Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D.-T., and Kuo, S.-Y. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the 13th Conference on the World Wide Web* (2004), pp. 40–52.

[17] Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D.-T., and Kuo, S.-Y. Verifying Web Applications Using Bounded Model Checking. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN2004)* (2004), pp. 199–208.

[18] Jim, T., Swamy, N., and Hicks, M. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *Proceedings of the 16th International World Wide Web Conference (WWW'07)* (2007), pp. 601–610.

[19] Lam, M. S., Whaley, J., Livshits, V. B., Martin, M. C., Avots, D., Carbin, M., and Unkel, C. Context-Sensitive Program Analysis as Database Queries. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems* (New York, NY, USA, 2005), ACM Press, pp. 1–12.

[20] Livshits, V. B., and Lam, M. S. Finding Security Errors in Java Programs with Static Analysis. In *Proceedings of the 14th Usenix Security Symposium* (Aug. 2005), pp. 271–286.

[21] Livshits, V. B., Whaley, J., and Lam, M. S. Reflection Analysis for Java. In *APLAS'05 - the Third Asian Symposium on Programming Languages and Systems* (2005), pp. 139–160.

[22] Martin, M. C., Livshits, B., and Lam, M. S. Finding Application Errors and Security Flaws using PQL: a Program Query Language. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2005), pp. 365–383.

[23] Musuvathi, M., Park, D. Y. W., Chou, A., Engler, D. R., and Dill, D. L. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the Conference on Operating System Design and Implementation (OSDI)* (2002), pp. 75–88.

[24] Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., and Evans, D. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the 20th IFIP International Information Security Conference (SEC)* (2005), pp. 295–308.

[25] Reimer, D., Schonberg, E., Srinivas, K., Srinivasan, H., Alpern, B., Johnson, R. D., Kershenbaum, A., and Koved, L. SABER: Smart Analysis Based Error Reduction. In *Proceedings of International Symposium on Software Testing and Analysis* (2004), pp. 243–251.

[26] SU, Z., AND WASSERMANN, G. The Essence of Command Injection Attacks in Web Applications. In *POPL '06: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2006), pp. 372–382.

[27] SUN MICROSYSTEMS. JSR-000154 Java Servlet 2.5 Specification. `http://jcp.org/aboutJava/communityprocess/mrel/jsr154/index.html`, 2004.

[28] SUN MICROSYSTEMS. JSR-000245 JavaServer Pages 2.1. `http://jcp.org/aboutJava/communityprocess/final/jsr245/index.html`, 2006.

[29] VISSER, W., HAVELUND, K., BRAT, G., PARK, S.-J., AND LERDA, F. Model Checking Programs. *Automated Software Engineering 10*, 2 (2003), 203–232.

[30] VISSER, W., PĂSĂREANU, C. S., AND KHURSID, S. Test Input Generation with Java PathFinder. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis* (2004), pp. 97–107.

[31] WASSERMAN, G., AND SU, Z. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)* (2007), pp. 32–41.

[32] WHALEY, J., AND LAM, M. S. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)* (2004).

[33] YANG, J., SAR, C., TWOHEY, P., CADAR, C., AND ENGLER, D. Automatically Generating Malicious Disks using Symbolic Execution. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)* (2006), pp. 234–248.

[34] YANG, J., TWOHEY, P., ENGLER, D., AND MUSUVATHI, M. Using Model Checking to Find Serious File System Errors. In *Proceedings of the Conference on Operating System Design and Implementation (OSDI)* (2004), pp. 273–288.