Automated Detection of Client-State Manipulation Vulnerabilities

Anders Møller
Department of Computer Science
Aarhus University
amoeller@cs.au.dk

Mathias Schwarz
Department of Computer Science
Aarhus University
schwarz@cs.au.dk

***Abstract*—Web application programmers must be aware of a wide range of potential security risks. Although the most common pitfalls are well described and categorized in the literature, it remains a challenging task to ensure that all guidelines are followed. For this reason, it is desirable to construct automated tools that can assist the programmers in the application development process by detecting weaknesses. Many vulnerabilities are related to web application code that stores references to application state in the generated HTML documents to work around the statelessness of the HTTP protocol. In this paper, we show that such client-state manipulation vulnerabilities are amenable to tool supported detection.**

We present a static analysis for the widely used frameworks Java Servlets, JSP, and Struts. Given a web application archive as input, the analysis identifies occurrences of client state and infers the information flow between the client state and the shared application state on the server. This makes it possible to check how client-state manipulation performed by malicious users may affect the shared application state and cause leakage or modifications of sensitive information. The warnings produced by the tool help the application programmer identify vulnerabilities. Moreover, the inferred information can be applied to configure a security filter that automatically guards against attacks. Experiments on a collection of open source web applications indicate that the static analysis is able to effectively help the programmer prevent client-state manipulation vulnerabilities.

***Keywords*-Web application security; information flow analysis; static analysis**

I. Introduction

Errors in web applications are often critical. To protect web applications against malicious users, the programmers must be aware of numerous kinds of possible vulnerabilities and countermeasures. Among the most popular guidelines for programming safe web applications are those in the OWASP Top 10 report that covers "the 10 most critical web application security risks" [18]. Essential to many security properties is the flow of untrusted data in the programs. This flow is often not explicit in the program code, so it can be difficult to ensure that sensitive data is properly protected. Although tool support exists for detecting and preventing some risks, manual code review and testing remain crucial to ensure safety. However, code review and testing are tedious and error-prone means, so it is desirable to identify classes of vulnerabilities that are amenable to tool support.

As an example, consider the category *A4 - Insecure Direct Object References* from the 2010 OWASP Top 10 list. A direct object reference is a reference to an internal implementation object, such as a database record, that is exposed to the user as a form field or a URL parameter in an HTML document. Such references are examples of *client state*, which is used extensively in web applications to work around the statelessness of the HTTP protocol, for example to store session state in the HTML documents at the clients.

Figure 1 shows two snippets of source code from a web application named *JSPChat*. Part (a) shows a JSP page containing an HTML form for saving personal information in a chat service, and part (b) shows the servlet code that is executed when the form data is submitted by the user. The first thing to notice is that the `nickname` form field on line 20 in the JSP page functions as a direct object reference that refers to a `ChatRoom` object and a `Chatter` object stored on the server. As the application programmer cannot trust that the user does not modify such references in an attempt to access resources that belong to other users, it is important to ensure that object references are protected. This can be done for example using a layer of indirection (i.e. using a map stored on the server from client-state values to the actual object references), via cryptographic signatures or encryption of the client state, or by checking that the user is authorized to access the resources being referenced in the requests. This is, however, easy to forget when programming the web application. In the example, the servlet reads the `nickname` parameter, stores it in a field in the servlet object, and then uses it – without any security measures – to look up the corresponding `ChatRoom` and `Chatter` objects in the shared application state on lines 48 and 50. Obviously, a malicious user can easily forge the parameter value and thereby access another person's data. (The careful reader may have noticed another vulnerability in the program code; we return to that in Section VIII.)

As documented in security alerts and reports by, e.g., ISS [10], MSC [3], Advosys [1] and Sanctum [26], vulnerabilities of this kind have been known—and exploited—for more than a decade. However, they remain widespread on the web, as evident from the 2010 OWASP report. A recent study shows that application developers are still unaware of common classes of related vulnerabilities, despite awareness programs provided by, for example, OWASP, MITRE, and SANS Institute [19]. A notable recent example of client-state manipulation is the attack on the Citigroup website that allowed hackers to disclose account numbers and transaction history for 200,000 credit cards [17].

```
<% ChatRoomList roomList =
    (ChatRoomList)application.getAttribute("chatroomlist");
 ChatRoom chatRoom = roomList.getRoomOfChatter(nickname);
 Chatter chatter = chatRoom.getChatter(nickname); %>
<html><head>
<meta http-equiv="pragma" content="no-cache">
<title>
  Edit your (<%=chatter.getName()%>'s) Information
</title>
<link rel="stylesheet" type="text/css"
  href="<%=request.getContextPath()%>/chat.css">
</head>
<body bgcolor="#FFFFFF">
<form name="chatterinfo" method="post"
  action="<%=request.getContextPath()%>/servlet/saveInfo">
<table width="80%" border="0" cellspacing="0"
  cellpadding="2" align="center" bordercolor="#6633CC">
<tr><td valign="top"><h4>Nickname:</h4></td>
<td valign="top"><%=chatter.getName()%></td>
<input type="hidden" name="nickname"
  value="<%=chatter.getName()%>">
</tr>
<tr><td valign="top"><h4>Email:</h4></td>
<td valign="top">
<input type="text" name="email"
  value="<%=chatter.getEmail()%>">
</td></tr>
<tr><td valign="top">
<input type="submit" name="Submit" value="Save">
</td></tr></table></form></body></html>
```

(a) `editInfo.jsp`

```
public class SaveInfoServlet extends HttpServlet {
  String nickname = null;
  String email = null;
  HttpSession session = null;
  String contextPath = null;

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
       throws IOException, ServletException {
    nickname = request.getParameter("nickname");
    contextPath = request.getContextPath();
    email = request.getParameter("email");
    session = request.getSession(true);
    ChatRoomList roomList = (ChatRoomList)
      getServletContext()
      .getAttribute("chatroomlist");
    ChatRoom chatRoom =
      roomList.getRoomOfChatter(nickname);
    if (chatRoom != null) {
      Chatter chatter = chatRoom.getChatter(nickname);
      chatter.setEmail(email);
      ...
    }
  }
}
```

(b) `SaveInfo.java`

Figure 1. A simplified version of the *JSPChat*[1] web application.

According to OWASP, "automated tools typically do not look for such flaws because they cannot recognize what requires protection or what is safe or unsafe". Nevertheless, in this paper we show that it is possible to develop automated tools that can detect many of these flaws. Our approach is based on a simple observation: *Vulnerability involving client-state manipulation is strongly correlated to information flow from hidden fields or other kinds of client state to operations involving the shared application state on the server.* This approach is along the lines of previous work on static taint analysis [14], [22], however with crucial differences in how we characterize the sources and sinks of the information flow. We discuss related work in Section IX.

In summary, the main contributions of this paper are as follows:

- Our starting point is a characterization of *client-state manipulation vulnerabilities* (Section II) that has considerable overlap with category A4 from the OWASP 2010 list of the most critical risks. In particular, we describe safety conditions under which the use of client state is likely not to cause vulnerabilities.
- Based on this characterization, we present an automated approach to detect occurrences of client state in a given web application and check whether the safety conditions are satisfied (Sections III–VI).
- In addition to reporting the detected vulnerabilities to the application programmer, we describe how the information obtained by the analysis can also be used for automatically configuring a security filter (Section VII) that guards against client-state manipulation attacks at runtime.
- Through experiments performed on 7 open source web applications with a prototype implementation of our analysis, we show that the approach is effective for helping the programmer detect client-state manipulation vulnerabilities. On a total of 1536 servlets, JSP pages, and Struts actions, our tool identifies 3349 occurrences of hidden fields and other client-state parameters being read, and it reveals 183 exploitable vulnerabilities involving 28 different field names.

The static analysis that underlies our automated approach to detect client-state manipulation vulnerabilities consists of three components. The first component (Section IV) infers the dataflow between the individual servlets and pages that constitute the application, in order to identify the *client-state parameters*. This requires a static approximation of the dynamically constructed output of the servlets and pages and extraction of relevant URLs and parameter fields in forms and hyperlinks. The second component (Section V) analyzes the program code to find out which objects represent *shared application state* (i.e., state that is persistent or shared between multiple clients, as opposed to session state or transient state). The third component (Section VI) performs an information flow analysis to identify the possible flow of user controllable input from client-state parameters to

[1]`http://www.web-tech-india.com/software/jsp_chat.php`

shared application state objects. The vulnerability warnings being produced by the tool can either be used to guide the programmer to apply appropriate countermeasures by modifying the application source code, or to automatically configure a security filter that we provide.

Our goal is not to develop a technique that can fully guarantee absence of client-state manipulation vulnerabilities. Rather, we aim for a pragmatic approach that can detect many real vulnerabilities while producing as few spurious warnings as possible. Since authorization checks and other countermeasures come in many different forms, the information flow analysis component may require some customization, but the analysis is otherwise fully automatic.

II. Client-State Manipulation Vulnerabilities

In a web application, *client state* comprises information that is stored within a dynamically generated HTML document in order to be transmitted back to the server at a subsequent interaction, for example when a form is submitted. Since the HTTP protocol is stateless, client state is widely used for keeping track of users and session state that involves multiple interactions between the each client and the server. Client state appears as hidden fields in HTML forms – as in the example in Section I – and as URL query parameters in hyperlinks. Such state is not intended to be modified by the user, but nothing prevents malicious users from doing so, and this is easy to forget when programming web applications. A related situation occurs with HTML select menus, radio buttons, and checkboxes, which also contain a fixed set of values that the user is intended to choose from. We commonly refer to HTTP GET/POST request parameters that contain such state as *client-state parameters*. Cookies provide another mechanism for carrying client state; in this paper we focus on ordinary HTTP request parameters, but our approach in principle also works for cookies.

For the discussion we consider Java-based web applications, specifically ones based on Java Servlets, JSP or Struts. We use the general term *page* to refer to a servlet instance, a JSP page, or a Struts action instance; each produces an HTML *document* when executed.

Figure 2 illustrates the data flow of client state for the *JSPChat* example. The value of `nickname` is passed as client state from a JSP page, `editInfo.jsp`, to a servlet, `SaveInfo.java`, using a hidden field in the HTML document.

The following characterization is our key to automate detection of the vulnerabilities we consider: A web application is vulnerable to *client-state manipulation* if is is possible, by users modifying client state, to access or manipulate shared application state that is not otherwise possible.

This class of vulnerabilities is closely related to MITRE's weakness categories CWE-472 (External Control of Assumed-Immutable Web Parameter)[2] and CWE-639

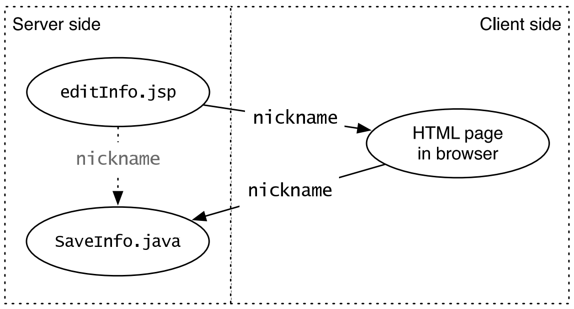

Figure 2. Data flow of client state from a JSP page to a servlet via an HTML document. The `nickname` parameter is sent from `editInfo.jsp` to `SaveInfo.java` via the HTML document.

(Authorization Bypass Through User-Controlled Key)[3] – and, as discussed in the previous section, to OWASP's risk category A4 (Insecure Direct Object References). Moreover, the description of CWE-472 mentions that it "is a primary weakness for many other weaknesses and functional consequences, including XSS, SQL injection, path disclosure, and file inclusion".

All client-state parameters – most importantly, those that originate from hidden fields in HTML forms – are potential sources of client-state manipulation vulnerability. On the other hand, we observe that uses of client state are *safe*, that is, not vulnerable to client-state manipulation, if at least one of the following conditions is satisfied:

1) The client-state parameter value stored in the HTML document is encrypted using a key private to the server. The server then decrypts the value when it is returned. A variant is to leave the value unencrypted but add an extra hidden field or URL parameter containing a digital signature (or MAC, message authentication code) computed from the client-state value and the server's private key. The server then verifies that the client-state value is unaltered by checking the signature when the form data is returned. To prevent against replay attacks and impersonation attacks, a time stamp and a client ID can be included in the encryption or signature generation. A drawback of this approach is that extra work is needed when producing and receiving the client state.
2) The client state entirely consists of large random values that are practically impossible to predict by attackers. A typical example is the use of *session IDs*: in many web applications, all session state is stored on the server and the only client state being used consists of session IDs, i.e. references to the session state on the server. A drawback of this approach is that it requires extra space on the server to store the session state.
3) An indirection is used: the client state consists of, for example, only numbers between 1 and some small constant, and these numbers are then mapped to the

[2]`http://cwe.mitre.org/data/definitions/472.html`

[3]`http://cwe.mitre.org/data/definitions/639.html`

actual application state on the server. This approach is particularly useful for select menus, radio buttons, and checkboxes. In this way, client-state manipulation cannot provide access to data beyond what is accessible from this map. A drawback of this approach is the burden involved in maintaining the indirection map.

4) The client-state parameter is treated as untrusted input, no different from other kinds of parameters, and any access to application state involving the given client-state value is guarded by an authorization check.
5) Finally, a sufficient condition for safety according to the definition above is that all the shared application state that can be accessed through client-state manipulation is already available by other means – that is, the information should not be considered sensitive.

OWASP's ESAPI[4] library contains support for implementing the first four of these approaches. The use of encryption and signatures to prevent manipulation of hidden form fields was originally suggested by MSC [3] and Advosys [1]. Thus, the countermeasures are well-known; the goal of our analysis is to detect when they are applied inadequately.

III. Outline of the Analysis

We adapt the well-known approach to static information flow analysis for identifying the possible dataflow from *sources* to *sinks* that does not pass through *sanitizers* [8], [11], [14], [22], [24], [25]:

- The sources in our setting are the locations in the code where client-state parameters are read. With common web application frameworks, such as Java Servlets, JSP, and Struts, it is not explicit in the application source code which parameters contain client state, so we need a static analysis to infer this information.
- The sinks are the operations in the source code that affect shared application state. We conservatively assume that this application state is not accessed by other means. As is it not explicit which objects and methods involve shared application state (in contrast to session state or transient state), we need another static analysis component to extract this information.
- The sanitizers correspond to the various kinds of protection described in Section II. For example, decrypting an encrypted client-state value is one kind of sanitization.

Our analysis tool has built in a collection of application agnostic mechanisms for identifying sinks and sanitizers. The user can customize the analysis by providing additional application specific patterns.

We propose the following procedure for analyzing a given web application: (1) Run the analysis on the application, with only the default sink and sanitizer patterns. (2) Study the warnings being produced and add customization rules to those that are considered false positives to enable the analysis to reason more precisely about the relevant parts of the application. (3) Then run the analysis again, using the new customization. Provided that the analysis is sufficiently precise, most warnings now indicate actual exploitable vulnerabilities.

[4]https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API

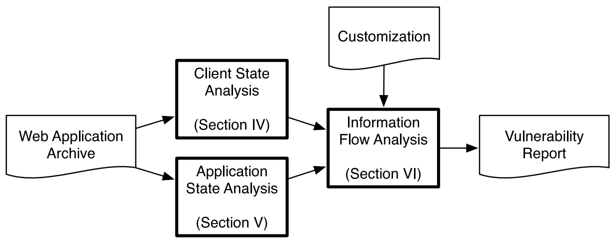
Customization
Client State Analysis (Section IV)
Web Application Archive
Application State Analysis (Section V)
Information Flow Analysis (Section VI)
Vulnerability Report

Figure 3. Structure of the analysis.

One approach to remedy the vulnerabilities detected by the analysis is that the programmer manually incorporates appropriate countermeasures into the web application source code as discussed in Section II. Another option is to feed the vulnerability report to our security filter that we describe in Section VII for automatic protection. Our experiments (see Section VIII) indicate that the burden of the customization step is manageable. However, we note that a fully automatic approach to protect against client-state manipulation can be obtained by omitting customization entirely and applying the security filter without having eliminated false positives. Compared to the more manual approach involving customization, the price is a modest runtime overhead incurred by the security filter since it may protect some client state unnecessarily.

The following sections explain how we identify client-state parameters and application state and perform the information flow analysis. The structure of the combined analysis is illustrated in Figure 3.

IV. Identifying Client State

When a page p reads an HTTP parameter, for example in lines 40 and 42 in Figure 1, the only way we can find out whether that is a client-state parameter is to analyze all the pages of the application that dynamically construct HTML documents with links or forms referring to p.

For each page q in the web application we first generate a context-free grammar G_q that conservatively approximates the set of HTML documents that may be generated by q. This can be done as in our previous work on analysis of dynamically generated HTML documents [16]. We then infer the references between the pages by identifying `a` and `form` elements in G_q. Combined with information extracted from the deployment descriptors (`web.xml` in Servlets and `struts.xml` in Struts) this results in a graph in which nodes correspond to pages and edges describe possible links and form actions. In the example in Figure 1, this step identifies the edge from `editInfo.jsp` to `SaveInfo.java`. To find the client-state parameters in the HTML documents generated by q, we now identify all elements that define hidden fields, select boxes, radio buttons, and links in G_q and collect the corresponding parameter names. Since these names may be generated dynamically in the program we approximate them conservatively by a regular language $C_{out}(q)$. In the example from Figure 1 this step identifies `nickname` as the only client-state parameter originating from `editInfo.jsp`, thus $C_{out}(\texttt{editInfo.jsp}) = \{\texttt{nickname}\}$.

The names of the incoming client-state parameters to a page p can now be expressed as $C_{in}(p) = \cup_{q_i} C_{out}(q_i)$ for each page q_i that has an edge to p. (In principle, p may have multiple incoming edges with different C_{out} sets, in which case the analysis issues a warning, though that never happens in our experiments.) For the example, this gives $C_{in}(\texttt{SaveInfo.java}) = \{\texttt{nickname}\}$. Parameter values in the Servlet framework are read using the `getParameter` method of the `HttpServletRequest` object. As the request parameter name that is given as argument to this method may not be a constant in the source code, we approximate for each call to `getParameter` the possible values as a regular language. If the language overlaps with $C_{in}(p)$, we mark the method call as a *client-state value source*. This step will mark the method call in line 40 in Figure 1 as such a source. The call in line 42 will not be marked since `email` is not in $C_{in}(\texttt{SaveInfo.java})$. We identify parameter read operations in a similar way for JSP and Struts.

The result of these steps is a set of method calls in the application code that will serve as sources of client-state values in the information flow analysis in Section VI. All of the steps can be done soundly in the sense that every call to `getParameter` and related operations that may return client state is always included in the statically inferred set of client-state value sources. In our experiments, we never observe any imprecision of this phase.

V. Identifying Shared Application State

To find the operations in the code that affect shared application state, i.e. state that is shared between all requests, we first identify the application state that is stored in memory, which we call the *internal* application state. This includes:

- all `HttpServlet` objects (and hence the value of `this` inside servlet classes) and `ServletContext` objects, and all values of static fields,
- all values of fields of objects that have been classified as internal application state, and conversely, all objects that have non-static fields containing internal application state (since session state and transient state sometimes points to shared application state, this rule may conservatively classify such state as application state), and
- all values returned from static methods or from methods on internal application state objects.

Finding all expressions in the code that may yield internal application state according to these rules can be done with a simple iterative fixpoint algorithm combined with an off-the-shelf alias analysis, such as the one provided by Soot [23].

We also find the *external* application state stored in files and databases. Such state is read and written using special API functions. Our analysis currently recognizes signatures of methods from the Java IO library and the Java persistence API. Other methods can be added using the customization mechanism described in Section VI.

Web applications often rely on libraries, such as Hibernate or Apache Commons, which are typically provided in separate jar files. We allow libraries to be omitted from the analysis for analysis performance reasons. This will simply cause the analysis to treat all method calls to those libraries conservatively as operations on external application state.

The result of this analysis component is consequently an over-approximation of the set of expressions in the code that yield internal application state and of the set of method calls that involve external application state. We use this information in the following section.

VI. Information Flow from Client State to Shared Application State

As outlined in Section III, we use an information flow analysis to identify flow of the client-state values in the program to the shared application state. In general, information flow analysis considers of two kinds of flow: *explicit* and *implicit* flow [4]. Explicit flow is caused by assignments and parameter passing. Other forms of explicit flow may be described using customized derivation rules, as described below. Implicit flow arises when the value of a variable depends on a branch condition involving another variable. We believe explicit flow is the most important indication of client-state manipulation vulnerability, so we choose to disregard implicit flow. A similar choice was made in other work about information flow in web applications [15], [22].

Information flow analysis requires a characterization of sources, sinks, and sanitizers. The sources in our analysis are the client-state value sources that were identified in Section IV. The sinks are program points where the application writes to fields of internal application state objects or calls methods that involve external application state.

Sanitizers can be methods that determine whether a given client-state value is safe, for example by performing access control or MAC checking cf. Section 2. Other sanitizers are methods that convert unsafe values to safe ones, for example by decrypting the values. As sanitizers are highly application specific, they are provided through customization, and none are built into the analysis.

The information flow analysis is flow sensitive, meaning that different information is obtained at different program points. Our current implementation is also context sensitive, so methods are analyzed separately for each call site.

The information flow analysis can be customized to improve precision by eliminating different kinds of flow. The customization rules fall in three categories:

C1: The first category consists of derivation rules that describe explicit flow that may occur as a result of method invocations. Each rule consists of a method signature and a description of the relevant dataflow between arguments and return values. We provide a collection of predefined derivation rules for string manipulation methods and wrapper class methods in the `java.lang` package of the Java standard library, but additional application specific rules can be added by the user of the analysis. Sanitizers of the kind that convert unsafe values to safe ones can be described in this way.

C2: The second category is for sanitizers that return a boolean indicating whether the given value is safe or not. When this boolean is used as a branch condition, the analysis will consider the sanitized value as safe in the true branch, so the analysis is path sensitive in this special case.

C3: The third category allows fine-tuning of the conditions under which methods are treated as sinks. As mentioned in Section V, library methods are by default considered external application state sinks. Since this in some cases leads to false positives, it is useful to be able to adjust the behavior.

The customization rules can be given either as annotations in the code or in a separate file. We give concrete examples of customizations in Section VIII.

VII. Automatic Configuration of a Security Filter

The approach of using MACs to protect against client-state manipulation attacks that we discussed in Section II can be implemented with a generic servlet filter that intercepts all HTML documents generated by the application and all HTTP request that are sent by the clients, without modifying the web application code [20]. For every use of client state in the HTML documents, an additional hidden field or query parameter containing the MAC is automatically inserted. Whenever an HTTP request is received from a client, the MAC check is performed on the appropriate request parameters. For this to work, the filter needs to be configured with information about which fields and parameters contain client state that should not be manipulated, and this information is precisely what our static analysis can provide. It is of course important that the client-state analysis is precise enough to correctly distinguish between parameters that carry client state and ones that do not. It is less critical that the information flow analysis is able to correctly distinguish between safe and unsafe uses of client state. However, to avoid the overhead of generating and checking MACs for parameters that are already safe by other means, it is nevertheless useful that also this analysis component is as precise as possible.

Note that using this security filter is optional; as discussed in Section III it can be viewed as an alternative or supplement to manually eliminating the vulnerabilities by appropriately patching the application source code.

VIII. Evaluation

Our prototype implementation, *WARlord*[5], reads in a Java web archive (.war) file containing a web application built with Java Servlets, JSP or Struts, together with an analysis customization file, and performs the analysis described in the preceding sections. The implementation is based on the Soot analysis infrastructure [23], the JSP compiler from Tomcat[6], and our HTML grammar analysis [16]. With this tool, we aim to answer the following research questions:

Q1: Is the analysis precise enough to detect client state vulnerabilities with a low number of false positives? Specifically, can it identify the common uses of client state, and is it capable of distinguishing between safe and unsafe uses of client state in the sense described in Section II?

Q2: Are the warning messages produced by the tool useful to the programmer for deciding whether they are false positives or indicate exploitable vulnerabilities?

Q3: In situations where the programmer decides that a vulnerability warning is a false positive, is it practically possible to exploit the customization mechanism to eliminate the false positive?

Q4: Is the analysis fast enough to be practically useful during web application development?

To answer these questions, we experiment with a collection of web applications. For each application, we go through the process suggested in Section III: We first run the WARlord tool on the application with no customization. After a manual study of the warnings being produced, appropriate customization is added, if possible, to address the false positives. If any exploitable vulnerabilities are found after running the analysis again, this time with the new customization, we fix them manually using one of the techniques mentioned in Section II.

Our experiments are based on 7 open source web applications found on the web: *JSPChat*[1] (the small chat application mentioned in Section I), *Hipergate*[7] (a customer resource management application written entirely in JSP), *Takatu*[8] (a large tax administration system), *JWMA*[9] (a web mail application), *Pebble*[10] (a widely used blogging application),

[5]`http://www.brics.dk/WARlord/`

[6]`http://tomcat.apache.org/tomcat-7.0-doc/jasper-howto.html`

[7]`http://hipergate.sourceforge.net/`

[8]`http://takatu.sourceforge.net/`

[9]`http://jwma.sourceforge.net/`

[10]`http://pebble.sourceforge.net/`

	Frameworks	Pages	Client-state params (Unique names)	
JSPChat	Servlets, JSP	16	19	(3)
Hipergate	JSP	760	1333	(282)
Takatu	JSP, Struts	558	1840	(31)
Pebble	Servlets, JSP	122	22	(11)
Roller	JSP, Struts	53	86	(27)
JWMA	Servlets, JSP	26	48	(18)
WebGoat	Servlets	1	1	(1)

Figure 4. List of benchmarks. The 'frameworks' column shows which web frameworks that are used in each benchmark; 'pages' is the total number of JSP pages, servlet source files, and Struts action source files; 'client-state params' is the number of client-state parameters inferred by the analysis, and 'unique names' is the number of distinct names of such parameters.

Roller[11] (another blogging application), and *WebGoat*[12] (a web application written by OWASP to demonstrate typical security problems in web applications). Our prototype supports Struts 2 but not version 1, so we do not include the full list of benchmarks from Stanford SecuriBench [13]. The benchmarks on our list cover a variety of application kinds of different size, they are written by different programmers, and they use different web frameworks (a mix of Java Servlets, JSP, and Struts). The Takatu project does not appear to be active, but it represents an interesting snapshot of an incomplete web application. Some characteristics of the benchmarks are listed in Figure 4. 'Client-state params' shows the total number of client-state parameters computed as $\sum_p |C_{in}(p)|$ for all pages p. Although $C_{in}(p)$ may in principle be infinite, each of the sets is a singleton in our experiments. Note that client-state values appear in all the benchmarks. The number of distinct names of the parameters, i.e. $|\bigcup_p C_{in}(p)|$, gives an indication of how many different kinds of client state that occur.

A. Experiments

JSPChat: The analysis identifies uses of 19 client-state parameters, and only 1 warning is produced about potential client-state manipulation vulnerability. The single warning is shown in Figure 5: as hinted in Section I, the application is prone to a timing attack since the request variables are stored in fields on the Servlet, which the analysis reveals. Notice that the output includes a trace from the source to the sink, which can make it easier to confirm or dismiss the error by manual inspection. If we manually correct this error by changing the field into a local variable, the analysis finds another error: the application is also prone to a classical client-state manipulation attack, since a malicious user may change the `nickname` request parameter and consequently change the information for another user. This error can be corrected by fetching the nickname from the session instead of a client-state parameter. After also correcting this error,

[11]http://roller.apache.org/

[12]https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

```
Write of client-state value (nickname) to
application state in line 23 of
 sukhwinder.chat.servlet.SaveInfoServlet.
Trace:
 sukhwinder.chat.servlet.SaveInfoServlet:
  void doGet(HttpServletRequest,HttpServletResponse)
```

Figure 5. Output from the WARlord tool for the *JSPChat* benchmark.

WARlord gives no more warnings. A manual inspection confirms that the remaining occurrences of client-state parameters are indeed safe. No customization is necessary for this application.

Hipergate: Hipergate uses an extreme number of client-state parameters to pass data between pages. All client-specific values are passed around using hidden fields. Running the analysis yields 119 warnings. With 6 customizations this number is brought down to 80 warnings, almost all of which are caused by client-state parameter values that flow into parameterized database queries without any checks. We have inspected all of the warnings, and many of them correspond to code that is vulnerable to attacks.

The main source of false positives originates from a use of randomly generated ID strings for database rows. Such strings are hard to guess and we do not consider this as vulnerable. If we exclude warnings given on uses of these random strings, 41 warnings remain.

All in all, 20 of the warnings reveal exploitable client-state manipulation vulnerabilities. One of the warnings reveals that a file can be read from the disk using a file name originating from a client-state parameter in `wb_style_persist.jsp`, which can be exploited to change the file on the disk. Although the programmer has carefully inserted authorization checks to ensure that the user should be granted access to the page in question, no checks are made for any of the client-state parameters, and they can therefore be manipulated by the client. The tool also gives a warning on the page `docrename_store.jsp`, which can be exploited to rename files. The programmer has inserted a check to ensure that the user has rights to rename the files, but this is performed on another parameter than the one holding the file name, and an attacker can therefore create an exploit that changes only the file name. Furthermore, the tool emits 5 warnings for the page `reference.jsp` where parameters can be injected into an SQL string. 1 warning on the page `catusrs_store.jsp` reveals that a client-state parameter can give access to update permissions for any user, and 2 warnings reveal a similar problem for `catgrps_store.jsp`. Similarly, 10 warnings in 6 other pages reveal places where client-state values give direct access to the database. In all 10 cases, data is queried and changed using a client-state parameter.

For the remaining 21 warnings, we find that attacks could give access to shared application state but only in ways that are harmless. The tool is able to classify 1253 out of 1333 uses of client-state parameters as safe.

Method	Behavior
`FileUtils.underneathRoot(File,File)`	C2 Sanitizer for arg 2
`FileManager.isUnderneathRootDirectory(File)`	C2 Sanitizer for arg 1
`Element.get(Serializable)`	C1 No value flow
`StaticPageIndex.getStaticPage(String)`	C1 No value flow
`FileUtils.getContentType(String)`	C1 No value flow

Figure 6. Customization rules for the *Pebble* benchmark.

Takatu: The analysis identifies 1840 client-state parameters. 184 warnings are issued, all but 9 caused by reading from the database using an ID that comes from a hidden field. These IDs are used for querying objects from the database. After manually inspecting the warnings we can see that 162 of them can be exploited to change data on the server. Other 13 warnings indicate places where objects are read from the database and calculations are performed based on these objects, which violates safe use of client state. The remaining 9 warnings indicate places where a client-state parameter holds the value of a flag that is used to query the database but none of them can be exploited. No customization is required for this application.

Interestingly, this web application at multiple places asks the user to confirm the deletion of an object. The ID of the object is stored in a hidden field that is not protected, so the client can delete any object of the same type if he changes the ID used as object reference. The errors are easily corrected e.g. by signing the vulnerable parameters and checking the signature when the parameter is sent back to the server.

Pebble: WARlord identifies 22 uses of client-state parameters and initially produces 4 warnings. It uses a dispatcher, so all requests except those to JSP pages go through a single servlet. The number of client-state parameters seems small because of this structure, but the classes being dispatched to make heavy use of the client-state parameters.

The web application stores files on the disk such that each blog has its own directory, and it uses the value of a parameter from a hidden field to determine the name of the file to save to, which is the cause of 2 warnings. However, each value used this way is verified to be a child of the blog folder, so the folder structure ensures that users cannot overwrite each other's files. The two first customization rules shown in Figure 6 handle this check of the parent folder.

Only 1 warning is produced after the customization. It is caused by the page where a new blog is added. This page uses an `id` parameter originating from a hidden field to set the database ID of the newly created blog and to create a directory for the files belonging to the blog. The `id` parameter is verified to only contain letters, and another check ensures that the ID is not already in use. Together, these two checks mean that there are no exploitable vulnerabilities related to the 4 warnings. The safety depends on a subtle invariant about the directory structure where files are stored on the disk. While this invariant is beyond what we can express with the customization mechanism, extracting the relevant code into a separate method would make the code easier to read, less prone to become vulnerable as a result of future changes, and it would become expressible as a sanitizer using the customization mechanism.

Roller: This web application has been systematically reviewed for the class of vulnerabilities we are trying to detect. All client-state parameters are protected with authorization checks that are well-documented in the code. Running WARlord initially results in 53 warnings on the 53 pages. We added 14 customization rules, which mainly describe information flow for a few string manipulation functions and information about queries of public information such as blog comments. Those functions are part of the Apache Commons API, so these rules are generally useful in all applications that use this API.

Only 1 warning remains after adding these rules. That warning refers to a page that allows blog comments to be deleted using a client-state parameter to identify the blog comments. All comments belong to a blog, and user rights are defined for each blog. The page checks whether each comment belongs to the blog and refuses any attempt to delete comments on other blogs in a way that cannot be modeled with our customization mechanism. However, if the code was rewritten slightly to use a separate method to check the ownership directly, this method could be marked as a sanitizer. That would also make it possible to check that future changes to this code does not create a vulnerability, and it would make the code more readable.

JWMA: This web application acts as a front-end for an email server using the Java Mail API, and it stores almost all data in the session state belonging to the user. It has little shared application state, but it uses client-state parameters extensively, in particular hidden fields.

With no customization, WARlord produces 10 warnings. Almost all of them involve the Java Mail API, which is part of the J2EE platform. We added the 7 customizations shown in Figure 7 to the the analysis (5 of them about the Java Mail API). The remaining warnings are caused by the client being able to change the recipient and contents of a message by changing client state. Through manual inspection it can be seen that JWMA allows clients to send emails to any recipient using other pages in the application, so we do not classify this as exploitable, though the reasoning cannot be captured with the customization mechanism.

WebGoat: We also tested a single servlet in the WebGoat application. The purpose of this servlet, which uses a single hidden field, is to demonstrate vulnerabilities of exactly the kind we want to detect. Unlike the other benchmarks, this application generates output using a custom DOM-like framework and we decided to manually create the set of parameters that may hold client-state values.

Perhaps surprisingly, our tool reports 0 warnings for this application. The reason is that WebGoat does not use the input variable for anything else than selecting a message

Method	Behavior
`dtw.webmail.pluging.RandomAppendPlugin.supportsAppendType(String,Locale)`	C1 Flow from arg 1 to return
`javax.mail.internet.InternetAddress.parse(String)`	C1 Flow from arg 1 to return
`javax.mail.Store.getFolder(String)`	C1 Flow from base and arg 1 to return
`dtw.webmail.JWMASession.authenticate(String,String,boolean)`	C1 No value flow
`javax.mail.Folder.create(int)`	C3 Writes base object to app state
`javax.mail.Folder.getMessages(int[])`	C3 Reads app state if base is app state
`javax.mail.Folder.getMessage(int)`	C3 Reads app state if base is app state

Figure 7. Customization rules used for the *JWMA* benchmark. 5 out of the 7 rules relate directly to the Java Mail API.

to send back to the client. This usage does not violate any of the safe usages presented in Section II and we therefore conclude that while the illustrative servlet of course mimics the behavior of a vulnerable piece of server code, it is actually not vulnerable to any attack. A manual inspection of the code confirms that the client is indeed not able to change the shared application state in any way by changing the value of the hidden field.

B. Summary of Results

Figure 8 summarizes the benchmark results from the previous section. The first column, 'Client-state params', is the same as in Figure 4. The following columns show the number of warnings before customization, the number of customization rules, and the number of warnings after customization. The tool produces at most one warning for each of the client-state parameters from the first column (however each warning may contain multiple traces from sources to sinks). The next column, 'Exploitable', shows how many of the warnings we could manually verify to be exploitable by malicious clients performing client-state manipulation attacks. The column 'Safe client-state params' shows the number of client-state parameters that the analysis after customization determines *not* to be vulnerable. For each category, the numbers in parentheses show the results after grouping together data that involve parameters of the same name, which, as in Figure 4 gives an indication of the variety. The final column shows the time spent for the full analysis.

The tests have been performed on a 2.4 GHz Core i5 laptop running OS X. The JVM was given 1 GB of heap space for each benchmark. The time and memory was primarily used by the Soot framework for loading classes and performing the pointer analysis.

With this, we are able to answer the research questions:

Q1: A manual inspection of the application code confirms that the client-state analysis succeeds in finding all client-state value sources without any imprecision. This amounts to a total of 3344 client-state parameters. The analysis determines that 98% of those parameters are safe, that is, they are not involved in any warnings. Moreover, 27 of the 57 warnings that are produced in total reveal exploitable vulnerabilities. The false positives are not evenly distributed among the benchmarks, and they are concentrated on a small number of different parameter names.

Q2: Based on the warnings given by the tool, especially the trace information, it was in each case possible for us to quickly determine whether it indicated a vulnerability or not. The entire process of classifying the warnings and adding customization rules for all 7 benchmarks took one person less than a day, despite having no prior knowledge of the benchmark code.

Q3: Adding customization rules in many cases reduced the number of spurious warnings considerably. As discussed for the individual benchmarks, the remaining cases typically involve subtle, undocumented invariants. Moreover, if allowing simple refactorings, such as extracting a safety check to a separate method, most of these cases could be captured within the existing customization framework. In the case of *Hipergate*, however, some uses of client state are safe for reasons that go beyond the current capabilities of customization.

Q4: The tool analyzes between 10 and 200 pages per minute. Pages can be analyzed individually, so when a programmer is modifying the application, he can decide to run the tool only on pages that have changed.

IX. Related Work

Client-state manipulation vulnerabilities, in particular the kind involving hidden fields, have been known for many years, as described in Sections I and II. Likewise, automated techniques for protecting against security vulnerabilities in web applications have a long history. We here explain the connections between our approach and the most closely related alternatives that have been proposed.

One direction of work is using runtime enforcement of security policies, as exemplified by the security gateway proposed by Scott and Sharp [20]. Given a security policy, their gateway can, for example, automatically attach MACs to hidden fields. The approach requires that the programmer specifies which input fields need this kind of protection, which, as discussed in Section I, is too easy to forget. In contrast, the idea in our approach is to inform the programmer – using static analysis of the application source code – that protection may be inadequate. We adopt Scott and Sharp's security gateway as presented in Section VII, however the configuration of our security filter is provided by static analysis, not by the programmer.

An essential constituent of our approach is the observation that client-state manipulation vulnerabilities are correlated to

	Client-state params		Warnings before customization		Customization rules	Warnings after customization		Exploitable		Safe client-state params		Time
JSPChat	19	(3)	1	(1)	0	1	(1)	1	(1)	18	(2)	30 s
Hipergate	1333	(282)	119	(61)	6	80	(50)	20	(18)	1253	(232)	60 m
Takatu	1840	(31)	184	(10)	0	184	(10)	162	(9)	1656	(22)	3 m
Pebble	22	(11)	4	(4)	5	1	(1)	0	0	21	(10)	1 m
Roller	86	(27)	53	(1)	14	1	(1)	0	0	85	(26)	4 m
JWMA	48	(18)	10	(10)	7	5	(5)	0	0	43	(13)	3 m
WebGoat	1	(1)	0	(0)	0	0	(0)	0	0	1	(1)	30 s

Figure 8. Summary of experimental results.

information flow from client state to application state. Together with automatic inference of client state (Section IV) and shared application state (Section V), this allows us to detect likely errors largely without requiring the programmers to provide any specifications. Some application specific customization is required though, as seen in Section VIII. For future work, it may be interesting to apply probabilistic specification inference [15] to automate this phase.

The WebSSARI tool by Huang et al. [8] pioneered the use of static information flow analysis to enforce web application security, and numerous researchers have since followed that path (see for example [11], [14], [22], [24], [25]). Our proof-of-concept implementation uses a simple information flow analysis, as described in Section VI. More advanced alternatives include the algorithms by Livshits and Lam [14] and Tripp et al. [22].

The first phase of our analysis that identifies the client-state parameters (Section IV) applies techniques from our earlier work on static analysis of HTML output of Java-based web applications [12], [16]. The WAM-SE and WAIVE analysis tools by Halfond et al. [6], [7] also infer interface specifications for web applications, however without identifying which parameters contain client state, for example originating from hidden fields.

Providing comprehensive support for diverse web application frameworks, such as Java Servlets, JSP, and Struts, is a challenging endeavour. A general framework, F4F, has recently been proposed by Sridharan et al. [21], however we have found that it is not sufficiently flexible for our setting, in particular for the client state identification phase. Still, the ideas in F4F may be adapted in future work to enable support for additional web application frameworks.

Finally, we note that several commercial tools are capable of detecting security vulnerabilities in web applications. According to a 2007 IBM white paper [9], the AppScan tool is capable of detecting vulnerabilities involving hidden field manipulation and parameter tampering. The latest version uses techniques from TAJ [22], however we have been unable to perform a proper comparison and obtain further information about the techniques applied by AppScan. Microsoft's CAT.NET[13] also uses static information flow analysis, but it cannot detect client-state manipulation vulnerabilities without detailed specifications provided by the user. Other commercial tools include NTOSpider[14] from NT OBJECTives, WebInspect[15] from Fortify/HP, and CodeSecure[16] and HackAlert[17] from Armorize. To our knowledge, most of these tools (with the exception of CodeSecure, which is developed from WebSSARI) employ crawling [2], [5], not static analysis. We believe static analysis can be a promising supplement to dynamic approaches as it may provide better coverage of the web application source code.

X. Conclusion

We have demonstrated that it is possible to provide tool support that can effectively help programmers prevent client-state manipulation vulnerabilities in web application code. The static analysis we have presented is capable of precisely identifying client state, in particular state stored in hidden fields, and help distinguishing between safe and unsafe use of such state. With WARlord, our prototype implementation of the analysis, we quickly discovered 183 exploitable weaknesses in 7 web applications. The analysis has high precision: for a total of 3166 non-exploitable client-state parameters, 97% were classified as safe.

Moreover, we have argued that the information inferred by the analysis can also be used for automatic configuration of a security filter that at runtime protects against client-state manipulation attacks.

Our experiments also indicate potential for improvements. Specifically, although analyzing the *Hipergate* benchmark revealed 20 weaknesses, it also resulted in a number of false positives originating from a small group of client-state parameters. It appears that many of these false positives can be avoided if the analysis is extended to also infer the provenance of the client-state values, which can be a subject for future work. It may also be worthwhile to extend the technique to reason about client state stored in cookies.

Acknowledgements

This work was supported by Google, IBM, and The Danish Research Council for Technology and Production.

[13]`http://blogs.msdn.com/b/securitytools/archive/2010/02/04/cat-net-2-0-beta.aspx`

[14]`http://www.ntobjectives.com/ntospider`

[15]`https://www.fortify.com/products/web_inspect.html`

[16]`http://armorize.com/index.php?link_id=codesecure`

[17]`http://armorize.com/index.php?link_id=hackalert`

References

[1] Advosys Consulting. Preventing HTML form tampering, 2000. http://advosys.ca/tips/form-tampering.html.

[2] J. Bau, E. Bursztein, D. Gupta, and J. C. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Proc. 31st IEEE Symposium on Security and Privacy*, 2010.

[3] D. I. Brussin. A white paper analyzing the MSC hidden form field web site vulnerability, 1998. Miora Systems Consulting.

[4] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20, July 1977.

[5] A. Doupé, M. Cova, and G. Vigna. Why Johnny can't pentest: An analysis of black-box web vulnerability scanners. In *Proc. 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, July 2010.

[6] W. G. J. Halfond, S. Anand, and A. Orso. Precise interface identification to improve testing and analysis of web applications. In *Proc. International Symposium on Software Testing and Analysis*. ACM, July 2009.

[7] W. G. J. Halfond and A. Orso. Automated identification of parameter mismatches in web applications. In *Proc. 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, November 2008.

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

[9] IBM. The dirty dozen: preventing common application-level hack attacks, 2007. ftp://ftp.software.ibm.com/software/rational/web/whitepapers/r_wp_dirtydozen.pdf.

[10] Internet Security Systems. Form tampering vulnerabilities in several web-based shopping cart applications, 2000. ISS E-Security Alert, http://www.iss.net/threats/advise42.html.

[11] N. Jovanovic, C. Kruegel, and E. Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. *Journal of Computer Security*, 18(5):861–907, 2010.

[12] C. Kirkegaard and A. Møller. Static analysis for Java Servlets and JSP. In *Proc. 13th International Static Analysis Symposium*, volume 4134 of *LNCS*. Springer-Verlag, August 2006.

[13] B. Livshits. Defining a set of common benchmarks for web application security. In *Workshop on Defining the State of the Art in Software Security Tools*, August 2005.

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

[15] V. B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: specification inference for explicit information flow problems. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2009.

[16] A. Møller and M. Schwarz. HTML validation of context-free languages. In *Proc. 14th International Conference on Foundations of Software Science and Computation Structures*, 2011.

[17] L. Moran. How Citigroup hackers broke in 'through the front door' using bank's website, 2011. http://www.dailymail.co.uk/news/article-2003393/How-Citigroup-hackers-broke-door-using-banks-website.html.

[18] Open Web Application Security Project. OWASP top 10, 2010. https://www.owasp.org/.

[19] T. Scholte, D. Balzarotti, and E. Kirda. Quo vadis? a study of the evolution of input validation vulnerabilities in web applications. In *Proc. 15th International Conference on Financial Crypto*, February 2011.

[20] D. Scott and R. Sharp. Abstracting application-level web security. In *Proc. 11th International World Wide Web Conference*, May 2002.

[21] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg. F4F: Taint analysis of framework-based web applications. In *Proc. 26th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2011.

[22] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: effective taint analysis of web applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2009.

[23] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot – a Java optimization framework. In *Proc. IBM Centre for Advanced Studies Conference*. IBM, November 1999.

[24] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *Proc. 30th International Conference on Software Engineering*. ACM, May 2008.

[25] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proc. 15th USENIX Security Symposium*, July/August 2006.

[26] ZDNet. New e-rip-off maneuver: Swapping price tags, 2001. http://www.zdnetasia.com/new-e-rip-off-maneuver-swapping-price-tags-21187583.htm.