SECURING WEB APPLICATIONS AND FINDING SECURITY VULNERABILITIES IN JAVA

Tejinder Singh*

ABSTRACT

SQL injection and cross-site scripting are two of the most com-mon security vulnerabilities that plague web applications today. These and many others result from having unchecked data input reach security-sensitive operations. This paper describes a language called PQL (Program Query Language) that allows users to declare to specify information flow patterns succinctly and declaratively. We have developed a static context-sensitive, but flow-insensitive information flow tracking analysis that can be used to find all the vulnerabilities in a program. In the event that the analysis gener-ates too many warnings, the result can be used to drive a model-checking system to analyze more precisely. Model checking is also used to automatically generate the input vectors that expose the vulnerability. Any remaining behavior these static analyses have not isolated may be checked dynamically. The results of the static analyses may be used to optimize these dynamic checks.

Our experimental results indicate the language is expressive enough for describing a large number of vulnerabilities succinctly. We have analyzed over nine applications, detecting 30 serious security vulnerabilities. We were also able to automatically recover from attacks as they occurred using the dynamic checker.

Keywords: *pattern matching, web applications, SQL injection, cross-site scripting, model check-ing.*

*Research Scholar, JJTU, Jhunjhunu, Rajasthan.

1. INTRODUCTION

The security of Web applications has become increasingly important in the last decade. With more and more Web-based applications deal with sensitive financial and medical data, it is crucial to protect these applications from hacker attacks. A security assessment by the Application Defense Center, which included more than 250 Web applications from e-commerce, online banking, enterprise col-laboration, and supply chain management sites concluded that at least 92% of Web applications are vulnerable to some form of attack . Another survey found that about 75% of all attacks against Web servers target Web-based applications

1.1 Information Tracking

Many vulnerabilities in web applications are caused by permitting unchecked input to take control of the application, which an at-tacker will turn to unexpected purposes. *SQL injection* is one of the top five external threats to corporate IT systems Via SQL injection, an attacker can introduce additional conditions or com-mands to a database query, thus allowing the attacker to bypass authentication or even alter or destroy data. In *cross-site scripting* (XSS), one of the top vulnerabilities in the past two years, an attacker can trick a victim into clicking on a URL that takes over the browser. In the so-called "reflection attack" XSS is used by a phisher to inject credential-stealing code into official sites without having to actually mimic the site he hopes to penetrate. SQL injection and XSS are but two *taint-based vulnerabilities* which can be detected by tracking the flow of untrusted data entered by the user and seeing if it flows unsafely into security-critical operations.

1.2 Causes of Vulnerabilities

Of all vulnerabilities identified in Web applications, problems caused by *unchecked input* are recognized as being the most common. To exploit unchecked input, an attacker needs to achieve two goals: **Inject malicious data intoWeb applications.** Common methods used include:

- ✓ **Parameter tampering**: pass specially crafted malicious values in fields of HTML forms.
- ✓ **URL manipulation**: use specially crafted parameters to be submitted to the Web application as part of the URL.
- ✓ **Hidden field manipulation**: set hidden fields of HTML forms in Web pages to malicious values.
- ✓ **HTTP header tampering**: manipulate parts of HTTP requests sent to the application.

- ✓ **Cookie poisoning**: place malicious data in cookies, small files sent to Web-based applications.

Manipulate applications using malicious data. Common methods used include:

- ✓ **SQL injection**: pass input containing SQL commands to a database server for execution.
- ✓ **Cross-site scripting**: exploit applications that output unchecked input verbatim to trick the user into executing malicious scripts.
- ✓ **HTTP response splitting**: exploit applications that output input verbatim to perform Web page defacements or Web cache poisoning attacks.

• **Path traversal**: exploit unchecked user input to control which files are accessed on the server.

• **Command injection**: exploit user input to
execute shell commands.

2. CODE AUDITING FOR SECURITY

Many attacks described in the previous section can be detected with code auditing. Code reviews pinpoint potential vulnerabilities before an application is run. In fact, most Web application development methodologies recommend a security assessment or review step as a separate development phase after testing and *before* application deployment. Code reviews, while recognized as one of the most effective defense strategies, are time-consuming, costly, and are therefore performed infrequently. Security auditing requires security expertise that most developers do not possess, so security reviews are often carried out by external security consultants, thus adding to the cost. In addition to this, because new security errors are often introduced as old ones are corrected, *doubleaudits* (auditing the code twice) is highly recommended. The current situation calls for better tools that help developers avoid introducing vulnerabilities during the development cycle.

3. PQL LANGUAGE OVERVIEW

The focus of PQL is to track method invocations and accesses of fields and array elements in related objects. To keep the language simple, PQL(Program Query Language) currently does not allow references to variables of primitive data types such as integers, floats and characters, nor primitive operations such as additions and multiplications. This is acceptable for object-oriented languages like Java because small methods are used to encapsulate most meaningful groups of primitive operations. The ability to match against primitive objects may be added to PQL as an extension in the future. Conceptually, we model the dynamic program

execution as a sequence of primitive events, in which the checkers find all subsequences that match the specified pattern. We first describe the abstract execution trace, then define the patterns describing subsequences of the trace.

4. Overview of Vulnerabilities'

In this section we focus on a variety of security vulnerabilities in Web applications that are caused by unchecked input. According to an influential survey performed by the Open Web Application Security Project, unvalidated input is the number one security problem in Web applications. Many such security vulnerabilities have recently been appearing on specialized vulnerability tracking sites such as SecurityFocus and were widely publicized in the technical press, Recent reports include SQL injections in Oracle products and cross-site scripting vulnerabilities in Mozilla Firefox

4.1 SQL Injection

Let us start with a discussion of SQL injections, one of the most well-known kinds of security vulnerabilities found in Web applications. SQL injections are caused by unchecked user input being passed to a back-end database for execution The hacker may embed SQL commands into the data he sends to the application, leading to unintended actions performed on the back-end database. When exploited, a SQL injection may cause unauthorized access to sensitive data, updates or deletions from the database, and even shell command execution.

Example 1. A simple example of a SQL injection is shown below:

```
HttpServletRequest request = ...;
String userName = request.getParameter("name");
Connection con = ...
String query = "SELECT * FROM Users " +
" WHERE name = '" + userName + "'";
con.execute(query);
```

This code snippet obtains a user name (userName) by invoking request.getParameter("name") and uses it to

construct a query to be passed to a database for execution (con.execute(query)). This seemingly innocent piece of code may allow an attacker to gain access to unauthorized

information: if an attacker has full control of string userName obtained from an HTTP request, he can for example set it to 'OR 1 = 1;—. Two dashes are used to indicate comments in the Oracle dialect of SQL, so the WHERE clause of the query effectively becomes the

tautology name = '' OR 1 = 1. This allows the attacker to circumvent the name check and get access to all user

records in the database.

4.2 Injecting Malicious Data

vulnerabilities is difficult because applications can obtain information from the user in a variety of different ways. One must check all sources of user-controlled data such as form parameters, HTTP headers, and cookie values systematically. While commonly used, client-side filtering of malicious values is not an effective defense strategy. For example, a banking application may present the user with a form containing a choice of only two account numbers; however, this restriction can be easily circumvented by saving the HTML page, editing the values in the list, and resubmitting the form. Therefore, inputs must be filtered by the Web application on the server. Note that many attacks are relatively easy to mount: an attacker needs little more than a standard Web browser to attack Web applications in most cases.

4.3 Parameter Tampering

The most common way for aWeb application to accept parameters is through HTML forms. When a form is submitted, parameters are sent as part of an HTTP request. An attacker can easily tamper with parameters passed to aWeb application by entering maliciously crafted values into text fields of HTML forms.

4.5 Hidden Field Manipulation

Because HTTP is stateless, many Web applications use hidden fields to emulate persistence. Hidden fields are just form fields made invisible to the end-user. For example, consider an order form that includes a hidden field to store the price of items in the shopping cart: **<input type="hidden" name="total_price" value="25.00">** A typical Web site using multiple forms, such as an online store will likely rely on hidden fields to transfer state

information between pages. Unlike regular fields, hidden fields cannot be modified directly by typing values into an HTML form. However, since the hidden field is

part of the page source, saving the HTML page, editing the hidden field value, and reloading the p age will causethe Web application to receive the newly updated value of the hidden field.

4.6 Cross-site Tracing Attacks

Analysis of webgoat and several other applications revealed a previously unknown vulnerability in core J2EE libraries, which are used by thousands of Java applications.

This vulnerability pertains to the TRACE method specified in the HTTP protocol. TRACE is used to echo the contents of an HTTP request back to the client for debugging purposes. However, the contents of userprovided headers are sent back verbatim, thus enabling cross-site scripting attacks. In fact, this variation of cross-site scripting caused by a vulnerability in HTTP protocol specification was discovered before, although the fact that it was present in J2EE was not previously announced. This type of attack has been dubbed *cross-site tracing* and it is responsible for CERT vulnerabilities 244729, 711843, and 728563. Because this behavior is specified by the HTTP protocol, there is no easy way to fix this problem at the source level. General recommendations for avoiding cross-site tracing include disabling TRACE functionality on the server or disabling client-side scripting.

5. CONCLUSIONS

In this paper we showed how a general class of security errors in Java applications can be formulated as instances of the general *tainted object propagation* problem, which involves finding all *sink objects* derivable from *source objects* via a set of given *derivation rules*. We developed a precise and scalable analysis for this problem based on a precise context-sensitive pointer alias analysis and introduced extensions to the handling of strings and containers to further improve the precision. Our approach finds all vulnerabilities matching the specification within the statically analyzed code. Note, however, that errors may be missed if the user-provided specification is incomplete.

Information flow is one of the basic analyses used in compiler optimizations; as such it is usually applied only to local variables within a procedure. Whole-program information flow of dynamically allocated objects is necessary for higher level software engineering tools. We formulated a variety of widespread vulnerabilities including SQL injections, cross-site scripting, HTTP splitting attacks, and other types of vulnerabilities as tainted object propagation problems.

REFERENCES

[1] C. Anley. Advanced SQL injection in SQL Server applications. http://www.nextgenss.com/papers/advanced sql injection.pdf, 2002.

[2] C. Anley. (more) advanced SQL injection. http://www.nextgenss. com/papers/more advanced sql injection.pdf, 2002.

[3] B. Arkin, S. Stender, and G. McGraw. Software penetration testing. *IEEE Security and Privacy*, 3(1):84–87, 2005.

[4] K. Beaver. Achieving Sarbanes-Oxley compliance for Web applications through security testing. http://www.spidynamics.com/ support/whitepapers/WI OXwhitepaper.pdf, 2003.

[5] B. Buege, R. Layman, and A. Taylor. *Hacking Exposed: J2EE and Java: Developing Secure Applications with Java Technology*. McGraw-Hill/Osborne, 2002.

[6] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software - Practice and Experience (SPE)*, 30:775–802, 2000.

[7] CGI Security. The cross-site scripting FAQ. http://www.cgisecurity.net/articles/xss-faq.shtml.

[8] B. Chess and G. McGraw. Static analysis for security. *IEEE Security and Privacy*, 2(6):76–79, 2004.

[9] Chinotec Technologies. Paros—a tool for Web application security assessment. http://www.parosproxy.org, 2004.

[10] Computer Security Institute. Computer crime and security survey. http://www.gocsi.com/press/20020407.jhtml?requestid=195148, 2002.

[11] S. Cook. A Web developers guide to cross-site scripting. http://www. giac.org/practical/GSEC/Steve Cook GSEC.pdf, 2003

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