

# Language-Based Security for Web Applications

## CS591 Final Paper

Lennon Day-Reynolds

June 10, 2008

### 1 Introduction

According to the MITRE corporation Common Vulnerabilities and Exposures (CVE) report on common vulnerability types [15], and the Open Web Application Security Project (OWASP) “Top 10” listing of web application vulnerabilities [5], the top two attack vectors used against web applications recently have been SQL injection and cross-site scripting (XSS). A related class of vulnerability, cross-site request forgery (XSRF), also appears in the OWASP listing, and provides a number of other interesting exploit opportunities for would-be attackers.

Each of these attacks depends on the poor containment and attribution of executable code and remote invocations on the World-Wide Web. However, because of the wide variety of both client and server software in use on the Web, traditional host or OS-level hardening techniques are not sufficient to protect against such attacks. Classic models such as the Bell-LaPadula security matrix [11] and DoD evaluation criteria [14] fail to accommodate the heavily-networked, dynamic security requirements of the modern Web.

For each type of vulnerability, I will show how they relate to (and where appropriate, differ from) historical models from the literature, explain the current industry countermeasures in common use, and offer areas of active research in the security research community which may offer improved resistance in future applications. Where appropriate, I will also offer sample code which illustrates both the vulnerability and an appropriate countermeasure.

Most of the current research I will examine is focused on programming language implementations, rather than libraries or standalone analysis tools. Based

on personal experience and bias, it is my belief that integrating security methodology into the actual languages used to implement applications is the best means to insure that such tools which actually be used by programmers in industry.

## 2 Challenges in Web Application Security

The underlying standards of the Web, including hypertext transfer protocol (HTTP) [23] and hypertext markup language (HTML), were designed first and foremost to allow transparent access to information [12]. To this end, web clients and servers are expected to support a wide variety of content types and encodings, as well as allowing aggregation of data from a wide variety of sources with varying levels of trust.

Since HTTP is a stateless protocol, other provisions such as “cookies” [24] must be made to associate application state with the multiple request/response invocations required to complete a transaction. This state may carry sensitive information, but only the most basic provisions are in place to prevent leakage of such information to untrusted hosts.

Finally, when we consider the increasingly widespread use of the JavaScript language (standardized as ECMAScript [4]), we see that untrusted remote code must be evaluated along with more static resources, presenting a wide variety of new attack vectors.

## 3 Example Vulnerabilities

### 3.1 SQL Injection

A SQL injection attack relies on two assumptions on the part of the attacker: first, that a web application depends on an relational database management system (RDBMS) system which supports the structured query language (SQL) language for its persistent data, and second, that some user inputs will not be properly filtered and validated on the server-side.

This class of vulnerability can be used to overwrite data in the application’s database, including security settings, financial records, or user contact information. More advanced attacks are also possible. Recent examples of wide-spread SQL injection have rewritten hundreds of thousands of web pages [6] to redirect clients towards overseas websites controller by the attacker. Those sites in turn exploited known browser security bugs to infect clients with malware.

```

1 public class ListingDetailController extends HttpServlet {
2     public void doPost(HttpServletRequest req, HttpServletResponse res)
3     throws ServletException, IOException, SQLException {
4
5         Connection conn = getPooledDBConn();
6
7         String ownerId = session.getAttribute("currentUserId");
8         String auctionId = req.getParameter("auctionId");
9
10        String sqlQuery =
11            "SELECT title, description, price " +
12            "FROM auctions " +
13            "WHERE auction_id = " + auctionId +
14            "AND owner_id = " + ownerId;
15
16        PreparedStatement stmt = conn.createStatement();
17
18        ResultSet rs = stmt.executeQuery(querySql);
19
20        // ... finish response handling
21    }
22 }

```

Figure 1: Unsafe use of request parameters in SQL generation

Superficially, SQL injection attacks resemble format string vulnerabilities, a class of bug in C programs long considered safe, but proven more recently [19] to be exploitable by attackers using precisely-crafted payloads.

In the general sense, they are an example of a “confused deputy” [25] vulnerability: the attacker substitutes a resource to which the server should in fact have access privileges, but which should not be accessible to the user in their current role.

However, because SQL injections can be written in a high-level language – namely, SQL itself – the potential community of attack authors is much wider than for attacks requiring intimate knowledge of a platform’s linker and assembler syntax. SQL injection vectors can be found almost mechanically by examination of program source code, and tested anonymously and remotely using any HTTP client.

Figure 1 provides an example of a simple SQL injection vulnerability. As we can see in lines 13-14, the SQL to be executed includes the value of the `auctionId` parameter from the current HTTP request without any escaping or validation. If the request were submitted with the string  `;DROP TABLE auctions;--` in place of a normal `auctionId` value, the attacker could cause the database server to entirely drop the `auctions` table.

```

1 public class ListingDetailController extends HttpServlet {
2     public void doPost(HttpServletRequest req, HttpServletResponse resp)
3         throws ServletException, IOException, SQLException {
4
5         Connection conn = getPooledDBConn();
6
7         String ownerId = session.getAttribute("currentUserId");
8         String auctionId = req.getParameter("auctionId");
9
10        String sqlQuery =
11            "SELECT title, description, price " +
12            "FROM auctions " +
13            "WHERE auction_id = ? " +
14            "AND owner_id = ? ";
15
16        PreparedStatement stmt = conn.prepareStatement(querySql);
17        stmt.setInt(1, Integer.parseInt(auctionId));
18        stmt.setInt(2, Integer.parseInt(ownerId));
19
20        ResultSet rs = stmt.executeQuery();
21
22        // ... finish response handling
23    }
24 }

```

Figure 2: Safe (type-checked) use of request parameters in SQL query

Correcting this issue requires that the type of input parameters be checked. As seen in Figure 2, this can in some cases be as simple as forcing a type check on each parameter to the SQL query, as is done in lines 17 and 18.

Type safety offers a number of potential advantages in insuring application security given untrusted input. Some modern programming languages, including Java and C# [22] include basic type-safety as core language features.

The Perl [9] programming language provides a "taint" mode runtime pragma which provides a simple sort of binary label, preventing untrusted input from being passed to potentially-destructive subroutines. Examples of protected include paths passed to filesystem operations and parameters to SQL queries using the DBI [13] library.

More advanced techniques include representing information flow as defined by Denning and Denning [20] using programming language types [32]. The FlowCaml [35] and Jif [37] languages implement such analysis via extensions of the type systems of the Caml and Java programming languages.

Such information flow analysis attaches security labels to all program values as part of their type information, and forces type checking on all values to prevent potential leaks or misuse of user-provided input.

In our sample application, the SQL injection vulnerability could be detected at compilation time in a language supporting information flow analysis if the `HttpServletRequest` class's `getParameter` method returned a value with an “untrusted” security label, while the `Connection` class's `prepareStatement` method did not list that label amongst those of its SQL string parameter.

### 3.2 Cross-Site Scripting

XSS attacks rely on the “active” nature of content on the Web. One particularly entertaining demonstration of the potential of such attacks was the MySpace Worm, also known as the “Samy Worm,” [7], which embedded self-replicating JavaScript in a user's profile on the popular MySpace social networking site. The actual effect of the worm was rather innocuous: it simply added the author to the “friend” list of any user who visited an infected page on the site (starting with the author's own profile) and then added the worm to their own profile page.

However mild the effects, the rate of infection was staggering: in less than 20 hours, more than one million users had been affected, and in the end, the entire MySpace domain had to be taken down for several hours in order to purge the malicious code from affected areas of the site.

XSS attacks are based on the fundamental lack of confinement [27] separating untrusted mobile code from trusted resources within the browser itself. Two primary mechanisms, enforced by the web browser security policy [1] seek to prevent untrusted JavaScript from performing malicious actions: first, scripts are not allowed to access resources on the client machine itself, and second, they may only access remote resources loaded from the same server as the JavaScript code itself.

However, since JavaScript code may be loaded and evaluated from within the content of HTML pages themselves, myriad opportunities exist for attackers to inject code that they have crafted into the context of a trusted site. The above-mentioned MySpace Worm attached fragments of code (suitably escaped using alternate character encodings supported by the browser) throughout the attacker's profile page, and relied on the browser to correctly assemble and evaluate them.

Figure 3 shows a simple example of an XSS vulnerability introduced by the direct inclusion of user input in the output of a web application. To exploit this vulnerability, an attacker could provide a URL which linked to the above search

```

1 public class SearchController extends HttpServlet {
2     public void doGet(HttpServletRequest req, HttpServletResponse resp)
3     throws ServletException, IOException, SQLException {
4
5         String searchTerms = req.getParameter("q");
6         SearchIndex index = getSearchIndex();
7
8         ArrayList results = index.doQuery(searchTerms);
9         PrintWriter out = resp.getWriter();
10
11         out.print("<h2>Search results for your query  '");
12         out.print(searchTerms);
13         out.print("':</h2>");
14
15         // ... finish response handling
16     }
17 }

```

Figure 3: Unsafe inclusion of user input into generated HTML content

service, and include a value of the `q` HTTP query argument which included an HTML `<SCRIPT>` tag. The code inside that tag would then be evaluated by the user’s browser in the context of the website hosting the search application, and could perform arbitrary actions with that user’s credentials.

(Note: This vulnerability is in fact an example of both XSS and XSRF, since the malicious code can be injected from a site elsewhere on the Web. XSRF attacks are discussed in more depth in the Section 3.3 of this paper.)

The modification made in Figure 4 is the addition of the regular-expression based filtering on lines 11-13, and the substitution of that filtered value for the original request parameter. Unfortunately, while this particular fix will block unsafe HTML scripting from being passed to the search query, it will also block many potentially-acceptable strings, including characters in languages other than the assumed English of the application writer.

As shown in early work on virus and trojan detection [16], correctly detecting malicious code is undecidable. Because of this, the only truly reliable mechanism to prevent XSS attacks while maintaining the completeness of JavaScript as a language is to block all untrusted code from being included in web application output, even if we wished to allow innocuous code from some users.

### 3.3 Cross-Site Request Forgery

An XSRF attack is accomplished by connecting to an authenticated remote resource using a simple URL, or the dynamic request API present in the JavaScript

```

1 public class SearchController extends HttpServlet {
2     public void doGet(HttpServletRequest req, HttpServletResponse resp)
3     throws ServletException, IOException, SQLException {
4
5         String searchTerms = req.getParameter("q");
6         SearchIndex index = getSearchIndex();
7
8         ArrayList results = index.doQuery(searchTerms);
9         PrintWriter out = resp.getWriter();
10
11        Pattern safeChars = Pattern.compile("[^a-zA-Z0-9 ]");
12        Matches match = safeChars.matcher(searchTerms);
13        String cleanTerms = match.replaceAll("");
14
15        out.print("<h2>Search results for your query ");
16        out.print(cleanTerms);
17        out.print("':</h2>");
18
19        // ... finish response handling
20    }
21 }

```

Figure 4: Safe (filtered) inclusion of user input into generated HTML content

implementation of modern browsers. Any application which allows HTTP GET requests to modify protected resources can easily be exploited via XSRF, simply by constructing a URL which embeds the parameters the attacker wishes to pass, and using it as target of any eagerly-evaluated URL, such the SRC attribute of an HTML IMG tag.

XSRF attacks also utilize XSS vulnerabilities in order to conduct complex actions on web applications while masquerading as a trusted user. Because web browsers eagerly cache many types data, including HTTP Basic authentication credentials and session cookies, a request submitted from a JavaScript procedure will appear to the server to be coming from the authenticated user.

In order to minimize exposure to XSRF attacks, we need to insure that complete mediation [33] is applied before remote access of server resources. Rather than simply checking that a user has authenticated at some point in the past, we should insure that they are currently authorized to access the resource in question.

A common, coarse-grained approach to this problem is to generate a nonce for each page served to a user during a possibly-destructive transaction, and insure that the response sent back contains that nonce. This unfortunately only protects from a limited subset of XSRF vulnerabilities, as a clever attacker can perform multiple request/response cycles while masquerading as the user, not

just a single iteration.

More precise control over access to server resources can be accomplished using capability-based security [28]. By insuring that the active capability set for the user is no larger than that needed for their current transaction, and by revoking capabilities that are no longer needed, we can minimize the exposure to XSRF attacks.

Much as information flow has moved from being a theoretical model to being implemented at the language level, capability theory has also moved into the realm of multiple competing, pragmatic implementations.

The E language [34] implements capabilities in an object-oriented idiom by representing each capability as a distinct runtime object, and insuring that object references are “unforgeable” – that is, the capabilities available in a particular scope are precisely those objects over which the current context has references, and there is no way to gain unauthorized object references.

Joe-E [30] is a subset of the Java programming language which restricts conforming objects to only those operations which provide this same level of assurance, along with a static verifier that allows compilation and linking of these restricted objects.

Finally, Caja [31] is a very young but interesting project which aims to define a “safe” subset of the JavaScript programming language which allows for static object-capability-based confinement of untrusted code. Untrusted JavaScript code which has been certified by the Caja compiler can be safely embedded in website content without exposing the user to XSS or XSRF vulnerabilities.

## 4 Future Work

Because of the inter-dependence of client, server, and database-tier code, some researchers have begun focusing on implementation of the entire web application stack from within domain-specific languages which are easily subjected to static analysis.

The Links programming language [17] was implemented to simplify development of multi-tier web applications by providing a single surface syntax to code being executed at the client, application server, and database. The SELinks project [18] extends the type system of Links to add support for static checking of security labels on application data, providing protection against many types of cross-domain attacks, including XSS and SQL injection. Both rely on code

generation to produce executable JavaScript, Java, and SQL code for execution on existing platforms.

Sirer and Wang [36] implemented a domain-specific language for the security policy associated with web applications based on temporal logic. This allows for them to capture multiple request/response cycles within a single policy statement, and allows for the security state associated with a user to change over time.

Their implementation generates code for the OpenACS [8] and .NET Framework [2] as runtime platforms, and they were able to show that the runtime overhead of their automatically-generated security checks was minimal compared to hand-written access control code.

The SEPostgreSQL project [21] has extended the PostgreSQL open source RDBMS with security labels based on the SELinux [3] project’s model of domain and type enforcement (DTE). Previous [10] work [29] on DTE focused mainly on its interpretation at the single-host level, but more recent extensions, such as Labeled IPsec [26] extend security labels for information passing over untrusted networks.

## 5 Conclusion

The problems presented by SQL injection, XSS, and XSRF attacks are not going to disappear any time soon. Much of the promise of the Web is its ability to democratize content creation, where “content” is defined both in the traditional sense of words and pictures, and in the modern sense of code and behavior.

However, improved tools and techniques can help to protect inexperienced developers from easily-avoided mistakes, as well as insure that fewer vulnerabilities sneak escape detection in large projects. The continued adoption of best practices from the security community such as type safety should be encouraged, so that developers in industry can benefit from the continuing advances being made in the academic realm.

## References

- [1] Client-side javascript guide [online]. Available from World Wide Web: <http://devedge-temp.mozilla.org/library/manuals/2000/javascript/1.3/guide/index.html>.
- [2] .net framework [online]. Available from World Wide Web: <http://msdn.microsoft.com/en-us/library/w0x726c2.aspx>.
- [3] Security-enhanced linux [online]. Available from World Wide Web: <http://www.nsa.gov/selinux>.
- [4] Emcascript language specification, December 1999. Available from World Wide Web: <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf>.
- [5] Owasp top ten: The ten most critical web application security vulnerabilities [online]. 2007. Available from World Wide Web: [http://www.owasp.org/images/e/e8/OWASP\\_Top\\_10\\_2007.pdf](http://www.owasp.org/images/e/e8/OWASP_Top_10_2007.pdf).
- [6] Mass sql injection [online]. April 2008. Available from World Wide Web: <http://www.f-secure.com/weblog/archives/00001427.html>.
- [7] Myspace worm explanation [online]. June 2008. Available from World Wide Web: <http://namb.la/popular/tech.html>.
- [8] Openacs documentation [online]. 2008. Available from World Wide Web: <http://openacs.org/doc/>.
- [9] Jon Allen. Perl 5.10 language reference, 2008. Available from World Wide Web: <http://perldoc.perl.org/index-language.html>.
- [10] Lee Badger, Daniel F. Sterne, David L. Sherman, and Kenneth M. Walker. A domain and type enforcement UNIX prototype. *Computing Systems*, 9(1):47–83, 1996. Available from World Wide Web: [citeseer.ist.psu.edu/badger96domain.html](http://citeseer.ist.psu.edu/badger96domain.html).
- [11] David E. Bell and Leonard J. LaPadula. Secure computer system: Unified exposition and multics interpretation. *MTR-2997 Rev. 1*, 1976.
- [12] Tim Berners-Lee. Web architecture from 50,000 feet. 1998. Available from World Wide Web: <http://www.w3.org/DesignIssues/Architecture.html>.

- [13] Tim Bunce. Dbi - database independent interface for perl.
- [14] National Computer Security Center. Trusted computer system evaluation criteria. 1985.
- [15] Steve Christey and Robert A. Martin. Vulnerability type distributions in cve. Technical Report 1.1, Mitre Corporation, May 2007. Available from World Wide Web: <http://cve.mitre.org/docs/vuln-trends/index.html>.
- [16] Fred Cohen. Computer viruses: theory and experiments. *Computers and Security*, 6(1):22–35, 1987.
- [17] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *5th International Symposium on Formal Methods for Components and Objects*, 2006.
- [18] Brian J. Corcoran, Nikhil Swamy, and Michael Hicks. Cross-tier, label-based security enforcement for web applications, March 2008. Submitted for publication.
- [19] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: Automatic protection from printf format string vulnerabilities. pages 191–200. Available from World Wide Web: [citeseer.ist.psu.edu/cowan01formatguard.html](http://citeseer.ist.psu.edu/cowan01formatguard.html).
- [20] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.
- [21] SE-PostgreSQL development team. The security-enhanced postgresql security guide. 2007.
- [22] ECMA International. *C# Language Specification*, 4th edition.
- [23] Network Working Group. Rfc 2616 - hypertext transfer protocol – http/1. *Internet RFC/STD/FYI/BCP Archives*, June 1999. Available from World Wide Web: <http://www.faqs.org/rfcs/rfc2616.html>.
- [24] Network Working Group. Rfc 2965 - http state management mechanism. *Internet RFC/STD/FYI/BCP Archives*, October 2000. Available from World Wide Web: <http://www.faqs.org/rfcs/rfc2965.html>.

- [25] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, 1988.
- [26] Trent Jaeger, David H. King, Kevin R. Butler, Serge Hallyn, Joy Latten, and Xiaolan Zhang. Leveraging ipsec for mandatory per-packet access control. In *Proceedings of the Second International Conference on Security and Privacy in Communication Networks*. Systems and Internet Infrastructure Lab, Pennsylvania State University and IBM Linux Technology Center, IBM T.J. Watson Research Center, 2006.
- [27] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973. Available from World Wide Web: <http://citeseer.ist.psu.edu/lampson73note.html>.
- [28] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984. Available from World Wide Web: <http://www.cs.washington.edu/homes/levy/capabook/>.
- [29] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. In *USENIX Annual Technical Conference*. National Security Agency and NAI Labs, 2001.
- [30] Adrian Mettler and David Wagner. The joe-e language specification. February 2008.
- [31] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized javascript, January 2008.
- [32] A. Sabelfeld and A. Myers. Language-based information-flow security, 2003. Available from World Wide Web: [citeseer.ist.psu.edu/sabelfeld03languagebased.html](http://citeseer.ist.psu.edu/sabelfeld03languagebased.html).
- [33] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, 1975.
- [34] Mark Siegler. *The E Language in a Walnut*, 2004. Available from World Wide Web: <http://skyhunter.com/marcs/ewalnut.html>.
- [35] Vincent Simonet. The flow caml system (version 1.00): Documentation and user’s manual. Available from World Wide Web: <http://crystal.inria.fr/~simonet/soft/flowcaml/flowcaml-manual.pdf>.

- [36] E. Sizer and K. Wang. An access control language for web services, 2002. Available from World Wide Web: [citeseer.ist.psu.edu/sirer02access.html](http://citeseer.ist.psu.edu/sirer02access.html).
- [37] Jif Implementation Team. Jif reference manual, 2008. Available from World Wide Web: <http://www.cs.cornell.edu/jif/doc/jif-3.2.0/manual.html>.

## 6 Glossary

**CVE** Common Vulnerabilities and Exposures

**OWASP** Open Web Application Security Project

**XSS** cross-site scripting

**XSRF** cross-site request forgery

**RDBMS** relational database management system

**SQL** structured query language

**HTTP** hypertext transfer protocol

**HTML** hypertext markup language

**DTE** domain and type enforcement