International Journal of Enterprise Computing and Business Systems

ISSN (Online): 2230-8849

http://www.ijecbs.com

Vol. 2 Issue 1 January 2012

WEB APPLICATION VULNERABILITY ASSESSMENT AND PREVENTING TECHNIQUES

Sreenivasa Rao B [1]
Dept. of Computer Science & Engineering
CMJ University, Shillong, India

Kumar N [2]
Dept. of Computer Science & Engineering
ACE college of Engineering and Management
Agra, India

Abstract: A Vulnerability Assessment (VA) is the process of identifying, quantifying, and prioritizing the vulnerabilities (security holes) in a system. VA has many things in common with risk assessment. The World Wide Web (WWW) is delivering a broad range of sophisticated web applications for business, net banking, news feeds, shopping etc., However, many web applications go through fast development phases with extremely short time, making it difficult to eliminate vulnerabilities.

Web applications provide access to increasing amounts of information, some of which is confidential. From an application perspective, vulnerability identification is absolutely

International Journal of Enterprise Computing and Business Systems

ISSN (Online): 2230-8849

http://www.ijecbs.com

Vol. 2 Issue 1 January 2012

critical and often over looked as a source of risk. Unverified parameters, broken access controls, and buffer overflows are just a few types of the many potential security vulnerabilities found in complex business applications. Here we analyse the design of web application security assessment mechanisms in order to identify poor coding practices.

Web applications vulnerable to attacks such as SQL injection and Cross-Site Scripting, Buffer Over Flows, Cross Site Request Forgery, Security Misconfiguration etc., we describe the use of a number of software testing techniques (including dynamic analysis, black-box testing, fault injection, and behaviour monitoring), and suggest mechanisms for applying these techniques to web applications.

Keywords: Vulnerability Assessment, Security analysis, penetration testing, application security preventing techniques

I. INRODUCTION

In computer security, vulnerability is a weakness which allows an attacker to reduce a system's information assurance. The Web Application Vulnerability Assessment (WAVA) is a method to test that assesses the security of interactive applications using web technologies such as e-banking, news and e-commerce web applications.

For the attack scenarios the organization wants to cover the Web Application Vulnerability Assessment and it will:

- Identify and analysis vulnerabilities
- Infrastructure, application and operational level;
- Identify root causes of weaknesses;

• Determine levels of business risk;

Web Application Vulnerability Assessment, taking into account the full range of layers ranging from “people and organization” down to the “physical environment” when is conducting the Vulnerability Assessment (VA). Depending on the scope and purpose of vulnerability assessment, it makes sense to start looking at the web security of crucial applications. Below is the list of Vulnerabilities reported by the web applications in the recent years.

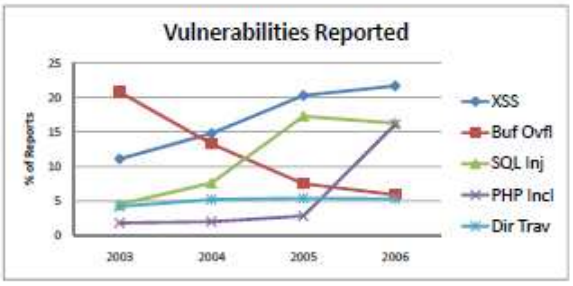

Fig 1: Web Application Vulnerabilities reported in the past recent years

A. Vulnerability Assessment: The First Steps

There are a number of reasons organizations may need to conduct a vulnerability assessment. It could be simply to conduct a check-up regarding overall web security risk posture. But if organization has more than a handful of applications and a number of servers, a vulnerability assessment of such a large scope could be overwhelming. The first thing needs to decide is what applications need to be assessed, and why. It could be part of PCI DSS requirements or the scope could be the web security of a single, ready-to-be deployed application etc.

B. Web Application Vulnerability Types

Detecting web application vulnerabilities, we choose the following:

1. Cross-Site Request Forgery (CSRF),
2. SQL injection (SQLI)

3.Cross-site scripting (XSS) as our primary vulnerability detection targets for two reasons:

a) They exist in many Web Applications, and

b) Their avoidance and detection are still considered difficult. Here will give a brief description of vulnerability followed by our proposed detection and prevention models.

II. CROSS SITE REQUEST FORGERY (CSRF)

CSRF is a kind of attack which forces an end user to execute unwanted or unknown actions on a web application in which he/she is currently authenticated. With a little help of social engineering like sending a link via email/chat, an attacker may force the users of a web application to execute actions of the attacker's choosing. A successful CSRF exploit can compromise end user data and operation in case of normal user. If the targeted end user is the administrator account, this can compromise the entire web application.

CSRF attacks are also known by a number of other names, including XSRF, "Sea Surf", Session Riding, Cross-Site Reference Forgery, and Hostile Linking. Cross-Site Request Forgery (CSRF) attacks are considered useful if the attacker knows the target is authenticated to a web based system. They only work if the target is logged into the system, and therefore have a small attack footprint. Other logical weaknesses also need to be present such as no transaction authorization required by the user. In effect CSRF attacks are used by an attacker to make a target system perform a function (like Funds Transfer, Form submission etc.) via the target's browser without the knowledge of the target user, at least until the unauthorized function has been committed. A primary target is the exploitation of "ease of use" features on web applications (One-click purchase).

A. CSRF Detection

Web application offers messaging between users. Upon login it sets a large, unpredictable session ID cookie which is used to authenticate further requests by users. One of the features of website is that users can send each other links inside their text

messages. It uses HTTPS to keep messages, credentials, and session identifiers secret from network eavesdroppers.

An “incoming messages” frame is displayed to users who have logged in. This frame uses JavaScript or a refresh tag to execute the “Check for new messages” action every 10 seconds on behalf of the user. New messages appear in this frame, and include the name of the sender and the text of the message. Text that is formatted as an HTTP or HTTPS URL is automatically converted into a link. The “Send a message” action takes two parameters: recipient (a user name or “all”), and the message itself which is a short string. To determine if this application is susceptible to CSRF, examine the “Send a message” action (although the “Logout” action is also sensitive and might be targeted by attackers for exploitation). When I do this, we find the following simple HTML form is submitted to send messages:

```
<form action="GoatChatMessageSender"
      method="GET">
<INPUT type="radio” value="Bob"
       name="Destination">Bob<BR>
<INPUT type="radio" name="Destination" value="Alice">Alice<BR>
<INPUT type="radio" name="Destination" value="Malory">Malory<BR>
<INPUT type="radio" name="Destination" value="All">All<BR>
Message: <input type="text" name="message" value="" />
<br><input type="submit" name="Send" value="Send Message" />
</form>
```

Form for sending messages:

Here is what it looks like in its frame:

From looking at this form we can figure out that when users wish to send the message “Hi Alice” to Alice, the following URL will be fetched when the user clicks Send Message.

International Journal of Enterprise Computing and Business Systems

ISSN (Online): 2230-8849

http://www.ijecbs.com

Vol. 2 Issue 1 January 2012

Figure 2: Form for sending messages

https://localhost:8080/GoatChatMessageSender?Destination=Alice&message=Hi+Alice&Send=Send+Message

When the user performs an HTTP GET for URL 1, their browser includes the cookies appropriate for the goatchat.isecpartners.com site. These cookies are sent if the URL is typed in manually, if it is followed as part of loading a frame, clicking a link, due to an image request, or by submitting a form, even if it is loaded as the result of a 302 redirect or a meta□refresh tag. The only requirement for the cookie to be sent is that the logged in user is the one making a request. Attackers can exploit this.

In this example used the method POST rather than GET, then exploitation by link would be done differently. The attacker would send the victim a link that directed the victim to an attacker controlled site, and the site would contain (possibly within an iframe) a form pointing to the target site, but with hidden fields. The form could be submitted automatically with JavaScript, or when the user clicked on it. More sophisticated variations allow for the exploitation even of multi□stage forms.

B. Hidden Form Based CSRF Exploit

The exploitation of a system which allows password changes, but does not require the user old password. In this example the target servlet requires an HTTP POST, and the attacker creates a self-submitting form to full fill this requirement. Note that this exploit is not as reliable as the image based request. Because the user's browser (or at least the tiny frame this exploit is placed in) is actually directed to the targeted site. Users with disabled browser scripting won't be exploited, and depending on the user's browser and configuration form submissions to other sites may result in a security popup box.

```
<HTML>
<BODY>
<form method="POST" id="evil" name="evil"
action="https: //www.yahoo.com/ VictimApp/PasswordChange">
<input type=hidden name="newpass" value="badguy">
</form>
                    <script>document.evil.submit()</script>
</BODY>
</HTML>
```

Even if scripting is disabled however a "close this window" link that is actually a submit button may trick a user into submitting the form on the attacker's behalf.

C. How to prevent CSRF

Preventing CSRF requires the inclusion of a unpredictable token as part of each transaction. Such tokens should at a minimum be unique per user session, but can also be unique per request.

1. The preferred option is to include the unique token in a hidden field. This causes the value to be sent in the body of the HTTP request, avoiding its inclusion in the URL, which is subject to exposure.
2. The unique token can also be included in the URL itself, or a URL parameter. However, such placement runs the risk that the URL will be exposed to an attacker, thus compromising the secret token.

International Journal of Enterprise Computing and Business Systems

ISSN (Online): 2230-8849

http://www.ijecbs.com

Vol. 2 Issue 1 January 2012

III. SQL INJECTION (SQLI)

For web applications, one common class of security problems is the so-called SQL command injection attacks. We use a simple example to illustrate the problem. Many applications include code that looks like the following:

String query = "SELECT * FROM employee WHERE name = '" + name + "'";

The user supplies the value of the name variable, and if the user inputs \John" (an expected value), then the query variable contains the string: "SELECT * FROM employee WHERE name = 'John'". A malicious user, however, can input \John' or 1=1--," which results in the following query being constructed:

"SELECT * FROM employee WHERE name = 'John' OR 1=1--'".

The \--" is the single-line comment operator supported by many relational database servers, including MS SQL Server, IBM DB2, Oracle, PostgreSQL, and MySQL. In this way, the attacker can supply arbitrary code to be executed by the server and exploit the vulnerability.

Although the source language, e.g., Java, may have a strong type system, it provides no guarantee about the dynamically generated SQL queries. Certainly direct string manipulation is a low-level programming model, but it is still widely used, and command injections do pose a serious threat both to legacy systems and to new code. A recent web-search easily revealed several sites susceptible to such attacks.

At the heart of SQL injections is an input validation problem, i.e. to accept only certain expected inputs. Proper input validation turns out to be very difficult to achieve injection attack. Several techniques to address it, and we give an overview here. At a low level, input can either be altered, so that inputs are rejected, or altered with the design of

making all inputs are good. One suggested technique is to enumerate the strings that the programmer believes are necessary for an injection attack but not for normal use. If any of those strings appear as substrings in the input, either the input can be rejected, or they can be cut out, leaving usually nonsense or harmless code.

Another common practice is to restrict the length of input strings. More generally, inputs can be altered by matching them against a regular expression pattern and rejecting them if they do not match. An alternative is to alter input by adding slashes in front of quotes in order to prevent the quotes that surround literals from being closed within the input. Common ways to do this are with PHP's add slashes function and PHP's magic quotes setting. Recent research efforts provide ways of systematically specifying and enforcing constraints on user inputs. Power- Forms provides a domain specific language to generate both client-side and server-side checks of constraints expressed as regular expressions. One recent project proposes a type system to ensure that all data is \trusted"; that type system considers input to be trusted once it has passed through a Perl's \tainted mode" has a similar goal, but it operates at runtime.

The best way to find out if an application is vulnerable to injection is to verify that all use of interpreters clearly separates un-trusted data from the command or query. For SQL calls, this means using bind variables in all prepared statements and stored procedures, and avoiding dynamic queries.

Checking the code is a fast and accurate way to see if the application uses interpreters safely. Code analysis tools can help a security analyst find the use of interpreters and trace the data flow through the application. Manual penetration testers can confirm these issues by crafting exploits that confirm the vulnerability.

A. Checking Access Control Policies

Access control policies (ACP) grant entities permissions on resources. Our analysis checks the generated queries against some given access control policy for the

database. DBMSs usually use role-based access control (RBAC), in which the entities are roles (e.g., administrator, manager, employee, customer, etc.) and users act as one of these roles when accessing the database. The active role for each hotspot is an input to our analysis. The permissions include, for example, SELECT, INSERT, UPDATE, DELETE and DROP etc.

The resources are database tables and columns. As we know for each column transition, all contexts (e.g., SELECT, INSERT, etc.) in which it may appear in the generated queries. We use this to discover access control violations. For example, if the role customer does not have the INSERT permission on table, even if table is mentioned in a SELECT sub-query of an INSERT statement, we will discover and flag the violation.

B. Detecting SQL Injections

SQL Injection refers to the technique of inserting SQL meta-characters and commands into web-based input fields in order to manipulate the execution of the back-end SQL queries.

An important point to keep in mind while choosing regular expression(s) for detecting SQL Injection attacks is that an attacker can inject SQL into input taken from a form, as well as through the fields of a cookie. Input validation logic should consider each and every type of input that originates from the user -- be it form fields or cookie information -- as suspect. Also if discover too many alerts coming in from a signature that looks out for a single-quote or a semi-colon, it just might be that one or more of these characters are valid inputs in cookies created by your Web application. Therefore, you will need to evaluate each of these signatures for your particular Web application.

As mentioned earlier, a trivial regular expression to detect SQL injection attacks is to watch out for SQL specific meta-characters such as the single-quote (') or the double-dash (--). In order to detect these characters and their hex equivalents, the following regular expression may be used:

International Journal of Enterprise Computing and Business Systems

ISSN (Online): 2230-8849

http://www.ijecbs.com

Vol. 2 Issue 1 January 2012

/(\%27)|(\')|(\-\-)|(\%23)|(#)/ix

Regex for typical SQL Injection attack:

/\w*((\%27)|(\'))((\%6F)|o|(\%4F))((\%72)|r|(\%52))/ix

Explanation:
\w* - zero or more alphanumeric or underscore characters
(\%27)|\' - the ubiquitous single-quote or its hex equivalent
(\%6F)|o|(\%4F))((\%72)|r|(\%52) - the word 'or' with various combinations of its upper and lower case hex equivalents.

The use of the 'union' SQL query is also common in SQL Injection attacks against a variety of databases. If the earlier regular expression that just detects the single-quote or other SQL meta characters results in too many false positives, you could further modify the query to specifically check for the single-quote and the keyword 'union'. This can also be further extended to other SQL keywords such as 'select', 'insert', 'update', 'delete', etc.

The second main check we perform on the generated SQL queries is to verify the absence of tautologies from all WHERE clauses. Generally, if an honest user wants to return all tuples (rows) for a query, the query will not have a WHERE clause. In the context of web applications, a tautology in a WHERE clause is an almost-certain sign of an attack, in which the attacker attempts to circumvent limitations on what web users are allowed to do.

C. How to Prevent SQL Injection

Preventing injection requires keeping un-trusted data separate from commands and queries.

- The preferred option is to use a safe API which avoids the use of the interpreter entirely or provides a parameterized interface. Beware of APIs, such as stored procedures that appear parameterized, but may still allow injection under the hood.
- If a parameterized API is not available, you should carefully escape special characters using the specific escape syntax for that interpreter.
- Positive or “white list” input validation with appropriate canonicalization also helps protect against injection, but is not a complete defense as many applications require special characters in their input.

IV. CROSS SITE SSCRIPTING (XSS)

Cross-site scripting (XSS) is a type of computer security vulnerability typically found in web applications that enables attackers to inject client-side script into web pages viewed by other users. A cross-site scripting vulnerability may be used by attackers to bypass access controls (AC) such as the same origin policy.

Cross site scripting web application vulnerability which allows attackers to bypass client-side security mechanisms normally imposed on web content by modern browsers. By finding ways of injecting malicious scripts into web pages, an attacker can gain elevated access privileges to sensitive page-content, session cookies, and a variety of other information maintained by the browser on behalf of the user. Cross-site scripting attacks are a special case of code injection. There are three known types of XSS flaws: 1) Persistent (or stored) XSS, 2) Non-persistent (or reflected) XSS and 3) DOM based XSS.

When hackers are using website to attack customers, you are probably dealing with a Cross Site Scripting attack. Hackers can inject JavaScript (a routinely used scripting solution that gets executed on the user’s web browser) which is normally used for legitimate functionality on websites, but in the hands of a hacker can be used for malicious purposes. Here are but a few examples:

- Steal cookies which can then be used to impersonate your customer and have access to their data and privileges. This is also known as Session Hijacking;
- Redirect the user to another website of their choosing. Maybe one that may be quite offensive, or one that attempts to install malware onto users computer;
- Display alternate content on your own website;
- Do ports scan of the customer's internal network, which may lead to a full intrusion attempt.

As with SQL injection, cross-site scripting is also associated with undesired data flow. To illuminate the basic concept, offer the following scenario.

A web site for selling computer-related merchandise holds a public on-line forum for discussing the newest computer products. Messages posted by users are submitted to a CGI program that inserts them into the Web application's database. When a user sends a request to view posted messages, the CGI program retrieves them from the database, generates a response page, and sends the page to the browser. In this scenario, a hacker can post messages containing malicious scripts into the forum database. When other users view the posts, the malicious scripts are delivered on behalf of the web application. Browsers enforce a same origin policy that limits scripts to accessing only those cookies that belong to the server from which the scripts were delivered. In this scenario, even though the executed script was written by a malicious hacker, it was delivered to the browser on behalf of the Web application. Such scripts can therefore be used to read the Web application's cookies and to break through its security mechanisms.

A. Persistent (or stored) XSS

The persistent (or stored) XSS vulnerability is a more devastating variant of a cross-site scripting flaw: it occurs when the data provided by the attacker is saved by the server, and then permanently displayed on "normal" pages returned to other users in the course of regular browsing, without proper HTML escaping. A classic example of this is with

online message boards where users are allowed to post HTML formatted messages for other users to read.

B. Non-persistent (or reflected) XSS

The non-persistent (or reflected) cross-site scripting vulnerability is by far the most common type. These holes show up when the data provided by a web client, most commonly in HTTP query parameters or in HTML form submissions, is used immediately by server-side scripts to generate a page of results for that user, without properly sanitizing the request. Example of non-persistent XSS Non-persistent XSS vulnerabilities in Google could allow malicious sites to attack Google users who visit them while logged in.

C. DOM-based XSS

DOM-based vulnerabilities occur in the content processing stages performed by the client, typically in client-side JavaScript. The name refers to the standard model for representing HTML or XML contents which is called the Document Object Model (DOM). JavaScript programs manipulate the state of a web page and populate it with dynamically-computed data primarily by acting upon the DOM.

A typical example is a piece of JavaScript accessing and extracting data from the URL via the location.* DOM, or receiving raw non-HTML data from the server via XMLHttpRequest, and then using this information to write dynamic HTML without proper escaping, entirely on client side.

Example Attack Scenario:

The application uses un-trusted data in the construction of the following HTML snippet without validation or escaping:

(String) page += "<input name='creditcard' type='TEXT‘value='" + request.getParameter("CC") + "'>";

The attacker modifies the ‘CC’ parameter in their browser to:

'><script>document.location='http://www.attacker.com/cgi-bin/cookie.cgi?'%20+document.cookie</script>.

International Journal of Enterprise Computing and Business Systems

ISSN (Online): 2230-8849

http://www.ijecbs.com

Vol. 2 Issue 1 January 2012

This causes the victim's session ID to be sent to the attacker's website, allowing the attacker to hijack the user's current session. Note that attackers can also use XSS to defeat any CSRF defense the application might employ.

D. Cross-Site Scripting (XSS) detection

Indications of cross-site scripting are detected during the reverse engineering phase, when a crawler performs a complete scan of every page within a web application. A crawler with the functions of a full browser results in the execution of dynamic content on every crawled page (e.g., JavaScript, ActiveX controls, Java Applets, and Flash scripts). Any malicious script that has been injected into a web application via cross-site scripting will attack the crawler in the same manner that it attacks a browser, We used the detours package to create a SEE that intercepts system calls made by a crawler. Calls with malicious parameters are rejected. The SEE operates according to an anomaly detection model.

During the initial run, it triggers a learning mode. It crawls through predefined links that are the least likely to contain malicious code that induces abnormal behaviour. Well-known and trusted pages that contain ActiveX controls, Java Applets, Flash scripts, and JavaScript are carefully chosen as crawl targets. As they are crawled, normal behaviour is studied and recorded. Our results reveal that during start-up, Microsoft Internet Explorer (IE)

1. Locates temporary directories.
2. Writes temporary data into registry.
3. Loads favourite links and history lists.
4. Loads the required DLL and font files.
5. Creates named pipes for internal communication.

During page retrieval and rendering, IE

1. Checks registry settings.
2. Writes files to the user's local cache.
3. Loads a cookie index if a page contains cookies.

4. Loads corresponding plug-in executables if a page contains plug-in scripts.

E. XSS Exploit (attack) scenarios

Attackers intending to exploit cross-site scripting vulnerabilities must approach each class of vulnerability differently. For each class, a specific attack vector is described here. The names below are technical terms, taken from the cast of characters commonly used in computer security.

Reflect XSS: 1.Alice often visits a particular website, which is hosted by Bob. Bob's website allows Alice to log in with a username/password pair and stores sensitive data, such as billing information.

2. Mallory observes that Bob's website contains a reflected XSS vulnerability.

3. Mallory crafts a URL to exploit the vulnerability, and sends Alice an email, enticing her to click on a link for the URL under false pretenses. This URL will point to Bob's website (either directly or through an iframe or Ajax call), but will contain Mallory's malicious code, which the website will reflect.

4. Alice visits the URL provided by Mallory while logged into Bob's website.

5. The malicious script embedded in the URL executes in Alice's browser, as if it came directly from Bob's server (this is the actual XSS vulnerability). The script can be used to send Alice's session cookie to Mallory. Mallory can then use the session cookie to steal sensitive information available to Alice (authentication credentials, billing info, etc.) without Alice's knowledge.

Stored XSS: 1.Mallory posts a message with malicious payload to a social network.

2. When Bob reads the message, Mallory's XSS steals Bob's cookie.

3. Mallory can now hijack Bob's session and impersonate Bob

F. Cookie Security

Other imperfect methods for cross-site scripting mitigation are also commonly used. One example is the use of additional security controls when handling cookie-based user

authentication. Many web applications rely on session cookies for authentication between individual HTTP requests, and because client-side scripts generally have access to these cookies, simple XSS exploits can steal these cookies. To mitigate this particular threat (though not the XSS problem in general), many web applications tie session cookies to the IP address of the user who originally logged in, and only permit that IP to use that cookie. This is effective in most situations (if an attacker is only after the cookie), but obviously breaks down in situations where an attacker spoofs their IP address, is behind the same NATed IP address or web proxy—or simply opts to tamper with the site or steal data through the injected script, instead of attempting to hijack the cookie for future use.

G. Safely Validating Untrusted HTML input

Many operators of particular web applications (e.g. forums and webmail) wish to allow users to utilize some of the features HTML provides, such as a limited subset of HTML markup. When accepting HTML input from users (say, very large), output encoding (such as very large) will not suffice since the user input needs to be rendered as HTML by the browser (so it shows as "very large", instead of "very large"). To stop XSS when accepting HTML input from users is much more complex in this situation. Untrusted HTML input must be run through an HTML policy engine to ensure that it does not contain XSS.

H. How to prevent XSS:

Preventing XSS requires keeping un-trusted data separate from active browser content.

1. The preferred option is to properly escape all un-trusted data based on the HTML context (body, attribute, JavaScript, CSS, or URL) that the data will be placed into. Developers need to include this escaping in their applications unless their UI framework does this for them.

2. Positive or “white list” input validation with appropriate canonicalization (decoding) also helps protect against XSS, but is not a complete defense as many applications require special characters in their input. Such validation should, as much as possible,

decode any encoded input, and then validate the length, characters, format, and any business rules on that data before accepting the input.

5. CONCLUSION

We have presented for verifying a class of security properties for web applications. In particular, we have presented techniques for the detection of Cross Site Request Forgery, Cross Site Scripting and SQL command injection vulnerabilities in these applications. Our analysis is sound. Based on encourage on this paper work on syntactic and semantic checking of dynamically generated database queries (SQL), XSRF, XSS and properties of the constructions presented in this paper.

6: ACRONYMS

[1] VA-Vulnerabilities Assessment

[2] SQLI–Structured Query Language Injection

[3] XSS – Cross Site Scripting

[4] DOM – Document Object Model

[5] HTML – Hyper Text Mark-up Language

[6] CSS – Cascading Style Sheet

[7] URL – Uniform Resource Locater

[8] DLL – Dynamic Link Library

[9] IE – Internet Explorer

[10] CSRF – Cross Site Request Forgery

[11] PCI DSS – Payment Card Industry Data Security Standard

[12] WWW – World Wide Web

[13] RBAC – Role Based Access Control

[14] DBMS – Database Management System

[15] ACP - Access control policies

[16] ACL - Access control list

[17] CC - Carbon copy

[18] IE - Internet Explorer

[19] WAVA - Web Application Vulnerability Assessment
[20] AC - Access Control
[21] UI – User Interface

7: REFERENCES

[1] L. O. Andersen. Program Analysis and Specialization for the C Programming Language. PhD thesis,University of Copenhagen, May 1994.

[2] M. Bishop. Computer Security: Art and Science. Addison Wesley Professional, 2002.

[3] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL injection attacks. In ACNS, 2004.

[4] C. Brabrand, A. Mller, M. Ricky, and M. I.Schwartzbach. Powerforms: Declarative client-side form field validation. World Wide Web, 2000.

[5] A. S. Christensen, A. Mller, and M. I. Schwartzbach.Precise analysis of string expressions. In Proc. SAS'03, pages 1{18, 2003. URL: http://www.brics.dk/JSA/.

[6] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In Proc. ICSE'04, May 2004.
[7] J. E. Hopcroft and J. D. Ullman. Introduction to Automata Theory, Language, and Computation. Addison-Wesley, Reading, MA, 1979.

[8] M. Howard and D. LeBlanc. Writing Secure Code. Microsoft Press, 2002.

[9] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai.Web application security assessment by fault injection and behavior monitoring. In World Wide Web, 2003.

[10] 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 World Wide Web, pages 40{52, 2004.

[11] Kavado, Inc. InterDo Vers. 3.0, 2003.

[12] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In Conference on LISP and Functional Programming, 1986.

[13] Y. Matiyasevich. Solution of the tenth problem of hilbert. Mat. Lapok, 21:83{87, 1970.

[14] D. Melski and T. Reps. Interconvertbility of set constraints and context-free language reachability. In Proc. PEPM'97, pages 74{89, 1997.

[15] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural data analysis via graph reachability. In Proc. POPL'95, pages 49{61, 1995.

[16] Sanctum Inc. Web Application Security Testing-Appscan 3.5.
URL: http://www.sanctuminc.com.