View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Northumbria Research Link

Northumbria Research Link

Roberts-Morpeth, P. and Ellman, J. (2010) 'Some security issues for web based frameworks', in Proceedings of the 7th international symposium on Communication systems networks and digital signal processing (CSNDSP), University of Northumbria, Newcastle, UK, 21-23 July, IEEE Xplore, pp. 726-731.

This paper was originally published by IEEE, 2010. Further details are available on the publisher's Website:

http://www.ieee.org/index.html

University Library, Sandyford Road, Newcastle-upon-Tyne, NE1 8ST

SOME SECURITY ISSUES FOR WEB BASED FRAMEWORKS.

P. Roberts-Morpeth, J. Ellman
School of Computing, Engineering, and Information Sciences, Northumbria University, Newcastle upon Tyne, UK
Jeremy. Ellman @unn.ac.uk

Abstract—**This report investigates whether a vulnerability found in one web framework may be used to find a vulnerability in a different web framework. To test this hypothesis, several open source applications were installed in a secure test environment together with security analysis tools. Each one of the applications were developed using a different software framework. The results show that a vulnerability identified in one framework can often be used to find similar vulnerabilities in other frameworks. Cross-site scripting security issues are the most likely to succeed when being applied to more than one framework.**

I. INTRODUCTION

Web Frameworks such as Ruby on Rails [1] and Microsoft's ASP.NET, are technologies designed to support the development of web applications, web sites and web services. [2] believe that a benefit of their use is to improve software quality. If this were to be the case, it is not unreasonable to expect this to be reflected in a reduction in the number of security issues.

[3] discusses maintainability as a major deciding factor when choosing between Ruby on Rails and the .NET framework. However there is no mention of security issues, or the possibility of any built-in vulnerability. The question is then whether web application frameworks can be relied upon to manage every aspect of security. To this end we have investigated some current security vulnerabilities relating to web based applications and frameworks.

Several Web frameworks were selected that use different underlying technologies. These were ASP.Net, Ruby on Rails, and Cake PHP. These frameworks were then used to implement test applications whose security could be investigated in an isolated network lab. Web framework vulnerabilities are published in several web sites. These sites were used to find published security issues together with the Security Focus Bugtraq. We then tested our experimental hypothesis; that a vulnerability reported in one framework could be used to identify a similar vulnerability in an application built using a completely different framework.

The paper proceeds as follows: Firstly we briefly review and describe the most common web application security failures [4]. Then we discuss how these apply to web framework based applications. Next we describe security testing environment used. Finally we present results that offer good support for our experimental hypothesis before moving on to discussion and conclusions.

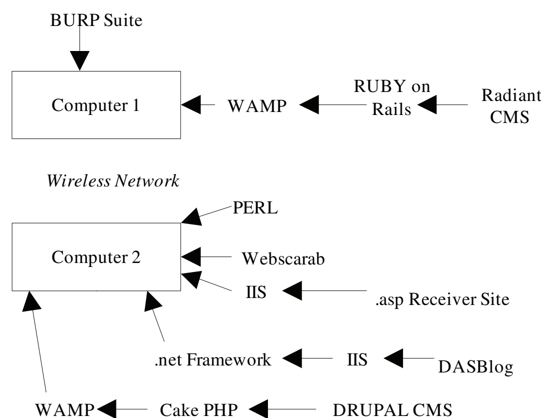

Figure 1. The Security Testing Configuration

II. BACKGROUND TO THE PROBLEM

As network firewalls and security updates have become more common, network boundaries have become more secure. For this reason computer security hackers and crackers have moved their attention from the Network Layer [5] to the Application Layer [5], focusing on the websites themselves. Over seventy percent of attacks now occur at the application layer [6] and research by WhiteHat Security has found eight in ten websites have serious vulnerabilities [6].

A web application is a piece of software written in a browser supported language such as HTML, Java, PHP, C#.NET that is accessed using the http protocol. Http is stateless. Thus, when a resource is requested and a response received, the next request is dealt with by the server as a new, unique request. To achieve the appearance of state, web applications can use a Session Identifier (SID). This simulates a session on top of http [7]. Following a client request, the server creates and returns an SID. The client then includes this SID in all subsequent requests within the same session.

We define a vulnerability as a weakness in a web application, web service or web site which can be caused by a programming error, design flaw or an implementation bug. Web applications and web services are vulnerable to a number of attacks [8]. Input manipulation attacks mainly occur via the application interface to exploit vulnerabilities in the application server. A number of different types of attack and security issue are discussed by [8] with the view of developing a validation framework for checking input before it is sent to the application

server. This would however only protect against people sending input to the application server via the input form. It would not protect against attacks that bypass client side validation by sending data directly to the application server.

Web application security analysis tools are designed to target an application from an attacker's point of view [9]. Several programs are readily available and well known (e.g. BURP Suite[10]; OWASP WebScarab[4]; and Paros Proxy [11]). These have common features such as the ability to manipulate requests and responses, spider an application to obtain a list of files in the site, and utilities for encoding and decoding sensitive data. One feature of the BURP Suite program is to allow all hidden html form fields to be displayed and updated on the web page. This demonstrates that the use of such fields is completely insecure.

A. Web Application Vulnerabilities

The Open Web Application Security project, OWASP, [4] tracks the most common failures in the area and has identified Injection attacks, Cross Site Scripting (XSS), and Broken Authentication and Session Management, as three most common areas of weakness. Consequently, we further describe these as relevant to our investigation.

Cross-site scripting (XSS) is an injection attack by one user of a web site on other users [12]. The typical attack pattern involves a malicious user including JavaScript code in a web page such as a blog post or other user editable field such the web server then delivers the page to other users. The embedded JavaScript may then be used to steal the victim's session, or other data stored in cookies.

A variation on XSS is cross site request forgery (XSRF, or CSRF). Here the payload JavaScript causes the victim to execute web requests with their network access and security privileges.

[13] discuss how most XSS scanners check input and rely on either a testing or static taint analysis approach (a list of functions designated as input cleaners) to detect illicit JavaScript. [13] found that in most cases, even flawed validation code catches such exploits and the perpetrators of such attacks know this and create exploits to target known validation weaknesses. [13] therefore proposed an approach based firstly on an adapted string analysis which tracks untrusted string values. They secondly propose checking for untrusted scripts. This check is based on formal language techniques. [13] also discuss the limitations of such a proposal as it does not work for Domain-Object-Model (DOM) based Cross Site Scripting.

In a further variation on XSS, an attacker could intercept web application traffic using an 'evil twin' rogue wireless access point, or comprised proxy server. Then using applications such as Burp Proxy [10,15] or Webscarab[16, 17] the attacker could inject JavaScript into the victims session in a 'man in the middle' attack. This would be equivalent in outcome to an XSS where JavaScript is stored on the server, and is worthy of further consideration.

SQL injection is a well known attack style [18] to which all web applications could be vulnerable. The general principle is that attackers input SQL code in addition to the input expected from a normal user (such as name, or any user data).

Using SQL injection techniques, a software hacker could take advantage of errors or vulnerabilities and use a web application to execute SQL statements against a database or to gain access to data files [18]. An attacker using SQL injection to insert code either directly in the URL or via form fields, could result in the system either giving access or returning an error supplying information about the system.

SQL injection and cross-site scripting techniques belong to a type of security issue known as “taint-style vulnerabilities” [19]. Those authors suggested that issues of this type share a “source-sink” characteristic. This is explained by the fact the user entered values or “tainted values” enter a program at certain points and then are propagated throughout the program. Microsoft offer a tool to find SQL injection vulnerabilities in Active Server Pages [20].

SQL injection attacks are however somewhat generic in that most web applications could be potential targets. Consequently, we considered it them be out of the scope of this investigation as we are concentrating on web application frameworks.

Several HTML tags may be used to inject cross site scripting using their attributes (e.g. *href*, *style* and *src*) and are also potentially dangerous. These tags at risk include *applet*, *body*, *embed*, *frame*, *script*, *frameset*, *html*, *iframe*, *img*, *style*, *layer*, *link*, *ilayer*, and *object* [21].

Session Hijacking involves swapping a unique identifier belonging to one user with the unique identifier of another. Each user has a unique identifier which is used during their use of a web application. This starts with the server issuing a unique identifier to each user when the user logs in or navigates via the home page of a site [22]. Future requests include this unique number so web applications can identify users and associate them with distinct sessions. Sessions are hijacked to gain greater privileges in the application than those to which users would be entitled. Thus application administrators are potentially targets.

A session could be hijacked by calculating the alphanumeric sequence of identifiers allocated by the application [23]. Indeed, WebScarab [16] will attempt to predict session keys based on the analysis of a large quantity of session data.

A further way that a session could be stolen would be to use security analysis software to launch a man-in-the-middle attack and copy the session information to another location for future use. Session identifiers can also be stored in cookies by applications and loaded each time the application starts. [24] considers session hijacking and how attacks performed by people eavesdropping via a network are commonly known as "sidejacking". [24] suggests that while Secure Sockets Layer (SSL) [25] can help protect such attacks. The use of SSL is not an option in every case as it can affect the performance of the application. Also, SSL may not be considered when dealing with non sensitive data. [24] therefore proposes a JavaScript client/server solution which uses the benefits of SSL to encrypt the session key but the remaining data in the page would be transmitted using http sessions. Every http request from that point on would include an encrypted authentication code. For this solution to work a JavaScript library has to be included in each web page and this is used to generate the secret key.

[24] discusses two limitations of the proposed solution. Firstly that a Session Lock is totally dependent upon JavaScript being enabled. Secondly that Session Lock does not protect against an attacker either amending or adding code to the plain http request which could be used to steal the session key and to use it elsewhere. Consequently, whilst Session Lock is a good idea and protects against a certain type of attack, it is still susceptible to attackers changing or stealing data from the page, including the session key itself.

In summary there are numerous types of attacks that can be performed against web applications [8]. XSS attacks are still considered to be a major threat against web applications and the security of web application data.

III. The Experiment

Here we describe the isolated lan environment used to investigate the vulnerabilities described. A new application designed to receive data from the target applications is also discussed. Following this a description of the known security issues that were tested for is provided, and the application of the issues is then performed. Each test is described and a result of either success or failure allocated depending upon the test outcome. A summary of the findings is then given.

The test environment was based on Microsoft Windows technologies. This was configured to safely test a cross section of available, modern technologies. This consisted of a laptop and a pc both running MS windows. Two installations of each application were required for Session Hijacking tests to facilitate the production of fake session ids.

The applications chosen for testing were DASBlog. A blogger application based on Microsoft .NET framework ; RADIANT. A CMS application based on the Ruby on Rails, and DRUPAL, a CMS application based on the Cake PHP framework.

A. *Software Tools used in this experiment*

1) *BURP-Proxy:*

Burp Proxy is a Java based web proxy server that allows user interaction. It is used for attacking and testing web applications [10, 15]. It functions as a man-in-the-middle between the user's browser and the target web server. This allows the user to intercept, and modify the raw http traffic between their browser and web server. Burp Proxy supports http request modification and can be used for attacks such as SQL injection, cookie interception, privilege escalation, Session Hijacking and buffer overflows.

2) *OWASP Webscarab:*

OWASP is the Open Web Application Security Project whose aim is the improvement of security of applications [4]. It offers an environment for testing and learning about security issues is using a Free, Open Source Software (FOSS) model. OWASP includes tools such as Webscarab for analyzing and discovering security issues in applications. Webscarab has a built in HTTP Proxy (with HTTPS interception), web-site crawler, session ID analysis, a script interface allowing for automation and a Base64 and MD5 encoder/decoder [4].

3) *The Receiver Application*

The receiver is a web application written for this project as a repository of site specific data sent from the target applications. The receiver includes a number of web pages for processing and writing data to log files. This is done using JavaScript functions designed to be called from within the target applications. The functions are also used to insert data or JavaScript code into the application pages.

B. *Vulnerabilities and issues subject to testing*

We examined for three vulnerabilities[1], Cross Site Scripting using JavaScript injection, Cross Site Scripting with HTML Object tag insertion, and Session Hijacking.

For XSS vulnerabilities, the first test was to insert a JavaScript file from a remote site containing functions to extract information from within the applications. The JavaScript file was inserted into the login page of each application and then executed. Information sent from this function was sent to the receiver and logged.

The second test was for XSS HTML Object tag insertion. Here we attempted to insert an html object tag into a field using the Burp Proxy intercept option. This was then saved by the application as part of the user data. This object was configured to point to the receiver application where JavaScript would be used to obtain data from the target application.

The third test was Session Hijacking. Here we attempted to manipulate session information to give an ordinary user priveleged access.

C. *Applying the vulnerabilities to the target frameworks*

1) *Test 1: XSS JavaScript injection.*

The purpose of this test was to investigate whether JavaScript from the receiver could be inserted into the login page and executed. Success would be achieved by including and executing a reference to a JavaScript file in the receiver site. Failure would be concluded if the JavaScript file could not be included successfully, or if the page did not subsequently function correctly, or if the JavaScript function was not executed.

a) *JavaScript injection in to the DasBlog application.*

The BURP proxy intercept option was used to add JavaScript to the DasBlog login page and to change the flow of the site by replacing the form submit button with one under our control. This was achieved by intercepting the message between the client and the server and inserting an html tag into the header section of the page. The purpose of this html was to call a JavaScript function in the receiver application. That is, to effect a cross site request forgery.

Next, again using BURP Proxy, the original submit button used by the page was hidden by setting the type attribute in the control to hidden. An html image tag was then added to the page as the new submit button. Finally an onclick event was added to the new submit button with a reference to a JavaScript function 'getFormValues' in the newly added informerFunctions.js.

The DasBlog Site was started and the login page displayed containing the newly inserted submit button. When the new submit button was pressed the page was submitted, the getFormValues JavaScript function sent the

[1] For details see [20-26]

form data sent to the receiver application. The Event Target, Event Argument, ViewState, Event Validation, User Name and Password fields were also logged. This test was successful as the JavaScript was added to the login page, executed and the target data collected by the receiver.

b) JavaScript injection in to the Radiant CMS application.

The BURP proxy intercept option was used, as with DasBlog above, to replace the login form submit button with one connected to the inserted JavaScript library. Here the User Name, Password and Authenticity_Token (used by Radiant CMS for security purposes) fields were collected by the receiver. This test was again successful as the injected JavaScript allowed the receiver to collect confidential data.

c) JavaScript injection in to the Drupal CMS application.

The BURP proxy intercept option was again used as with DasBlog and Radiant CMS above to hide the original submit button and replace this with our custom JavaScript functions. The test was again successful as the JavaScript executed and the receiver collected the target data.

The test was successful for all three applications, as the JavaScript function was added to the login pages in all three of the applications. The JavaScript was then executed and login information was sent to the receiver application and recorded in a log file.

2) Test 2: XSS: HTML Object tag insertion.

The purpose of this test was to see whether an html object tag could be added to a user input field. Here BURP Proxy's intercept option was used in the test environment (see fig 1) to intercept the message between the client and the server and insert an html object tag. This was added to a new content or blog entry depending on the application. Success would be achieved if the html object tag were added to an application input field.

a) HTML object tag insertion in to the DasBlog application.

Two new blog entries were added, allowing a different version of the object tag to be added to the first line of each blog. These were added to the first line of the blog entry as this line is also displayed on the blogger site home page. The first entry was a plain text version titled Blog 2. The second was an encoded version added to an entry entitled encoded object.

Next the entries were viewed on the home page of the site using the Firefox web browser. The new entries were visible on the DasBlog home page but the Encoded Object entry had been translated into a plain text format. The original plaintext version in object 2 was not visible (which is expected as the source object did not exist). The entries on the home page were then viewed using the Internet Explorer 7 browser.

Launching the DASBlog web-site and logging in through IE7 revealed that the home page could no longer be viewed. Each time the home page was launched it forwarded itself on to the test address http://god/dasblog/www.mysite.com. This effectively made the application unusable in IE7. We considered this test to be successful as the object tag was added to the blog page and saved by the system. However, the application was unusable in Internet Explorer as the home page could not be reached and every action forwarded the application on to an unreachable url. Since this was a serious bug it was logged in the DASBlogIssue Tracker on codePlex (issue number 4183).

b) HTML object tag insertion in to the Radiant application.

As with DasBlog, the BURP proxy intercept option was used to add the html object tag to a new content entry on page submission. Next the entry was viewed on the home page of the site using the Firefox web browser. The new entry was visible on the Radiant home page pointing to the the receiver application application which has been inserted in to the page. An html tag defining an object with an invalid Url was then inserted. However, the Invalid object was ignored by the Radiant application and neither the target application nor the html tag were displayed. This test was successful as the object tag was added to the content page and saved by the application. The object displayed the target Receiver Application in the page. The application did handle incorrectly configured object tags.

c) HTML object tag insertion in to the Drupal application.

Two new content pages were added, allowing a different version of the object tag to be added to the first line of each. On Test 1 the object is not displayed. On Test 2 the encoded text has been converted and the object tag displayed. This test was consequently a failure as the object tag was handled correctly by the application and disallowed.

3) Test 3: Session Hijacking.

This method of attack was reported on a Ruby site forum and had been used to attack the Radiant CMS application [27]. The purpose of this test was to investigate whether the session information used by the applications to manage security levels and system access could be manipulated to give a standard user a higher level of access. Again the BURP Proxy security analysis tool was used in the test environment shown in fig 1, using the receiver application to record data. Success would be demonstrated by gaining a higher level of access in the system or by gaining access without using a user credentials for example, bypassing the login screen.

a) Session Hijacking: DASBlog.

The BURP proxy intercept option was used to change the session information used by the DASBlog application and the options available to the user were then observed.

The administrator's session information can be gained by either accessing the administrator users stored cookie information, which in the case of the Windows operating system is located within their roaming profile. Or, as used in this case a separate test version of the application was used to generate valid administrator session details which were used within the live application.

The test DASBlog site was launched and the administrator's account used to login. On login the form details including the session value were passed and logged by the receiver application. The live DASBlog application was launched using a direct url pointing to a page within the site rather than the login page. The BURP Proxy intercept option was then used to locate and replace the

DASBlog ASP session value with the value obtained by the receiver application. This was repeated for each message that contained the session value.

The DasBlog application was launched and the user was presented with a blog view page. It was then possible to select, edit and delete any blog entry. This type of functionality can normally only be performed by the site administrator account. This test was successful as the site was accessed with administrator access without entering a user name or password. Radiant: Thus, for DasBlog type of functionality can normally only be performed by the site administrator account whilst access had been enabled for an ordinary user.

b) Session Hijacking. Radiant

Here we proceeded as for DasBlog using a test and live application. The test Radiant site was launched and the administrators account used to login. On login the form details including the session value were passed and logged by the receiver application. The Live Radiant application was launched using a direct url pointing to a page within the site rather than the login page. The BURP Proxy intercept option was then used to locate and replace the Radiant session value with the value obtained by the receiver application. This was repeated for each message that contained the session value.

The Radiant application was launched and the user was presented with a content administration page for maintaining users and configuring site preferences. This functionality can normally only be performed by the site administrator account, and is consequently a significant security vulnerability.

c) Session Hijacking: Drupal.

Here we proceeded as above taking an administrator session id from a correctly authorized session on a test site, and using this key in a live Drupal site. However, the 'create content' page was exactly the same as would be presented to an anonymous user. Changing the session information had no effect. Thus, for Drupal this test failed.

IV. EXPERIMENT RESULTS

We chose three common vulnerabilities to investigate. The first was to apply a XSS vulnerability to each of the applications. Weakness in application security allowed JavaScript to be injected into a web page before the page was rendered by the browser on the client machine.

For the first test, an XSS, JavaScript injection vulnerability was found for all three applications tested, as JavaScript was added and displayed within the client web browser.

The second test was to apply a XSS, HTML Object tag insertion vulnerability. Application weakness allowed JavaScript to be injected into an input field in a web page. The injected JavaScript was then saved by each application to either a database or an XML file. This vulnerability was successful for two of the three applications tested.

The third test Session Hijacking used a second installation of each application to generate a session key that was usable to gain access to the first installation. The vulnerability gave administrator access to two of the applications, removing the need to login using a user name and password. The DRUPAL application alone did not allow access via this method.

These results are summarized in Table 1 below.

Table 1: Summary of Vulnerabilities

Application	Did the expected vulnerabilities exist?		
	XSS JavaScript injection	XSS HTML tag injection	Session Hijacking
DASBlog	Yes	Yes	Yes
Radiant CMS	Yes	Yes	Yes
DRUPAL CMS	Yes	No	No

V. DISCUSSION AND CONCLUSION

This paper has reported an investigation into the security issues of web based frameworks. Its hypothesis was that a vulnerability found in one web based framework can be used to uncover a similar vulnerability in a different framework. For example, the Session Hijacking issue tested against the Radiant CMS application was originally reported on the Radiant Issue log site. This was then used to uncover the same issue with the DASBlog application.

New vulnerabilities were also found as a result of the testing. Using the BURP Proxy application allowed an html object tag to be inserted into the content field of a new blog in the DASBlog application. This highlighted a serious issue, if the path specified in the object tag was invalid, the application was rendered unusable in Internet Explorer. This was reported on the DASBlog issue tracker as a number of sites use this application and are potentially vulnerable.

All web based frameworks provide and recommend security features to protect application data from malicious code. However, the majority expect malicious code to be inserted in to form fields. Using XSS, JavaScript injection and a proxy-server it is possible to inject the malicious code in to any section of the web page, or remove any existing protective code from a web page.

This paper has shown that any person with a proxy server can read, extract and manipulate the data sent. This would allow a man-in-the-middle to change web pages sent from a web server, before the page arrived on the client machine. Unfortunately people with this ability are constantly around us as whenever we visit a hotel, a coffee shop or other location using a proxy server.

There are currently internet sites that inform the users of the site that a built in proxy server is used to protect their identity. As all of the messages are travelling through the proxy server, the site could be used to manipulate the data and pages in the ways described in this paper.

If there are sites that advertise the use of built in proxy servers, there will be sites or more importantly search engines that do not advertise the fact, leaving the users of the site vulnerable.

The underlying frameworks do provide security features to test and clean scripts from input and hidden

fields, but the page content in its entirety is not validated. Would it be possible to develop a security technique to ensure that the page displayed within the client browser is exactly as expected? If any extra lines of code exist could a warning be displayed to inform the user? For example, could a Cyclic Redundancy Check (CRC) be performed on the data to ensure that data received had not been changed in anyway? [28] discuss and demonstrate a method used to manipulate data in a way that is not detectable to a CRC. Although [28] demonstrates the theory on business application data, It does bring into doubt the usability of CRC to guarantee successful delivery of web based communications.

This report has shown that there is fair evidence to support our hypothesis: that a vulnerability found in one web framework may be used to find a vulnerability in another, entirely separate web framework.

Our future work will consider potential defenses against these types of attacks against web framework based applications.

REFERENCES

[1] Bachle M & Kirchberg P. 2007 Ruby on Rails. IEEE Software, v 24, n 6, pp 105-108

[2] Fayad M & Schmidt D.1997 Object-oriented application frameworks. CACM, pp32-38,

[3] Lok F, Fang S, Stan J & Bimlesh W.A Comparative Study of Maintainability of Web Applications on J2EE, .NET and Ruby on Rails. Nat. Uni. Singapore, 2008

[4] OWASP(b) http://www.owasp.org/index.php/Main_Page Accessed 23/01/10)

[5] Peters L, De Turck F, Moerman I, Dhoedt B & Demeester P. Network Layer Solutions forWireless Shadow Networks. Proc. Mobile Comms and Learn. Tech (2006)

[6] Grossman J 2007 10 "Things You Should Know about Website Security." http://www.whitehatsec.com/home/resource/whitepapers/website_security.html

[7] Gollmann D. Securing Web applications. Hamburg Uni. Technology, Hamburg 21071, Germany. Information on Security Technical Report, pp 1-9, 2008.

[8] Brinhosa R, Westphall CB and Westphall MC. 2008 A.Security framework for Input Validation. Network & Management Lab, Fed Uni. Santa Catarina, Brasil

[9] Fonseca, J, Vieira, M & Madeira, H. Testing and Comparing Web vulnerability Scanning Tools for SQL Injection and XSS Attacks. Proc.13th PRDC 2007, pp 365-372

[10] Portswigger, http://portswigger.NET/proxy/ Accessed (22/06/2009)

[11] Scambray J, Shema M & Sima C. 2006 Hacking Web Applications Exposed. McGraw

[12] Braganza R. 2006 Cross-site scripting - an alternative view. Network Security no 9

[13] Wassermann G & Su GW Static Detection of cross-site scripting vulnerabilities, Uni. California, pp 171-180, 2008

[14] Krebs B 2006 'Hacked Ad Seen on MySpace Served Spyware to a Million' The Washington Post "http://blog.washingtonpost.com/securityfix/2006/07/myspace_ad_served_adware_to_mo.html" accessed Feb 2010

[15] Stuttard Dafydd; Marcus Pinto 2007 "The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws" John Wiley & Sons

[16] OWASP 2010 " OWASP WebScarab Project" accessed Feb 2010 http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project

[17] Hope Paco; Walther Ben 2008 "Web Security Testing Cookbook" (Cookbook) O'Reilly Media, Inc.

[18] Bisson R 2005. SQL Injection, The Computer Bulletin, No 47, pp25

[19] Jovanovic N and Kruegel C and KirdaA E. Static Analysis Tool for Detecting Web Application vulnerabilities, Technical Uni. Vienna, 2006.

[20] Microsoft 2008 "The Microsoft Source Code Analyzer for SQL Injection tool is available to find SQL injection vulnerabilities in ASP code" http://support.microsoft.com/kb/954476

[21] Microsoft(a) http://msdn.microsoft.com/en-us/library/ms998274.aspx (2009)

[22] Andrews M and Whittaker J.A 2006 "How to Break Web Software." pp 59 Pearson Ed.

[23] Endler D. 2001 Brute-Force Exploitation of Web Application Session Ids. iDefense Labs,

[24] Adida B 2008 SessionLock: Securing Web Sessions against Eavesdropping, CRCS Harvard University, Refereed Track: Security and Privacy - Web Client Security

[25] Dierks T & Allen C 1999 The TLS protocol version 1.0, RFC 2246, www.ietf.org

[26] Roberts-Morpeth Paul 2009 An investigation into security vulnerabilities of Web based frameworks. MSc Thesis, Northumbria University. Available on request.

[27] Radiant CMS Accessed (21/06/2009). http://www.ruby-forum.com/topic/116043

[28] Schiller F and Mattes T and Weber U 2009 Undetectable Manipulation of CRC Checksums for Communication and Data Storage, 1st International Business Conference, ChinacomBiz, Communications and Networking in China, Vol 26, pp 1-9,