Implementation of Web Application Security Solution using JAAS

[1]Dr Surjeet Dalal , [2]Dr Gundeep Tanwar
[1]SRM University Haryana
[2]BRCMCET , Bahal , Haryana
[1]surjeeetdalalcse@gmail.com , [2]mr.tanwar@gmail.com

Abstract :Web applications are one of the most prevalent platforms for information and service delivery over Internet today. As they are increasingly used for critical services, web applications became popular and valuable target for security attacks. Although a large number of techniques have been developed to fortify web applications and mitigate the attacks toward them, there is little effort devoted to drawing interaction among these techniques and building a big picture of web application security framework. As today's application's infrastructures are getting increasingly complex and interconnected, the difficulty of achieving application security is exponentially increasing. We present our experience of implementing OWASP protocol into large scale web application and the advantages gained thereof. These main security threats dealt in the current work are: Injection, Cross-Site Scripting, and Security misconfiguration. A quantitative analysis of the impact on various performance and security parameters is presented. We conclude that these security features are helpful in preventing the web-based attacks, and reduce security risks and development costs.

Keywords: OWASP, Web security, JAAS.

I.INTRODUCTION

We all know that web security is important. Certainly the cost of failures is high: a recent survey has found an average cost of $7.2 million per data breach event (or $214 per compromised customer record). It was also found that 88% of the organizations surveyed had at least one major data breach in 2010. The problem arises as most of the enterprises have invested in network and PC security but many have neglected to build adequate safeguards into their software applications. But nowadays, application security is rapidly being recognized as a top priority. Gartner has stated that: "Over 70% of security vulnerabilities exist at the application layer, not the network layer," [1] and other researchers have estimated this figure at 90%.

Recent research have demonstrated the pertinence and authority of the Open Web Application Security Project (OWASP) in defining standards for security over cloud and other platforms.[7,8,9] Since 2003, the OWASP publishes a list of the most critical web application security risks[2,10,11].This list represents a consensus among many of the world's leading information security experts about the greatest risks, based on both the frequency of the attacks and the magnitude of their impact on businesses. The objective of the OWASP project is not only to raise awareness about specific risks, but also to educate business managers and technical personnel on how to assess and protect against a wide range of application vulnerabilities. OWASP Model provides information about Web Application Security Risks in which following three web application security risks has been identified and tested in the presented work:

1. A1 - Injection Based Attacks: Injection flaws, such as SQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing unauthorized data.

2. A2 - Cross-Site Scripting (XSS): XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation and escaping. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.

3. A6 - Security Mis configuration: Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, and platform. All these settings should be defined, implemented, and maintained as many are not shipped with secure defaults. This includes keeping all software up to date, including all code libraries used by the application.

The OWASP organization suggests that the OWASP list can be used to "establish a strong foundation of training, standards and tools that makes secure coding possible." Enterprises who have implemented a successful application security program integrate the OWASP into each stage of their software development lifecycle (SDLC) to design, develop and test new software applications.

The paper is organized as follows: Section 2 discussed various phases of implementation along with the challenges faced.

II.DISCUSSION

A) Misconceptions against Security Standards

A frequent question in the technical community has been the need of existence of such standards like OWASP, when developers can implement security features themselves. But such statements contradict various surveys that very few developers have been educated on secure coding practices. Even with experienced developers, emerging threats [3] require refresher courses every year or two based on how attach methodologies continue to change. So educational programs built around the OWASP provide essential education that most developers might not seek to acquire on their own.

Another argument against security standards is to utilize software testing tools and let them detect vulnerabilities in applications. But software testing tools are almost useless unless developers learn how to use them and know where to point them. In fact, they can be worse than useless, because if not used properly they can generate large numbers of "false positives" that cause resources to be wasted hunting down non-existent bugs.

A third common misconception is that programs designed to improve application security are focused only on software coding. Many security and compliance requirements are missed during the requirements and design phases of the life cycle, and many vulnerabilities are created during the deployment and maintenance phases.

B) Phases of Implementing Security Standards

Requirements analysis and Design:

In the Requirements and Analysis phase, analysts consider the requirements and goals of the application, as well as possible problems and constraints. Part of this process involves threat modeling, which identifies threats and vulnerabilities relevant to the application.

The OWASP can be used as guides to potential attacks. A thorough examination of which of those risks could affect the software will suggest ways the application design can be shaped to achieve security objectives, and where resources could be applied to develop countermeasures.

Development:

a) In the Development phase, specific coding standards that have been proven to defend against the risks can be adopted. As an example, developers could be required to have their software encode user-supplied input; that is, to tell the database "these characters come from a user screen, so they are definitely data and should never be executed as commands."
b) To address some of the "Failure to Restrict URL Access" issues, coding standards might require that every web page be protected by role-based permissions. For example, special logon screens for managers could be added to prevent attackers (and non-management employees) from accessing management screens.
c) Code reviews are another activity that typically occurs during the Development phase. Most developers review code only to make sure that it has the features and functions described in the specification. But developers trained to look also for vulnerabilities in the code related to the OWASP will find many types of security issues.

Testing:

When the quality assurance group builds the test plan, it can ensure that specific tests are run to simulate attacks related to the risks. Static analysis tools which read through software code can be programmed to look for clues in the code that the application may be vulnerable to risks. Web scanning tools can be programmed to simulate attacks based on vulnerabilities. For example, they could be set up to attempt injection attacks on all customer input screens.

C) Deployment

Computer systems and software that are not configured with security in mind can open up systems to attacks. That is why the OWASP can be very helpful in the Deployment phase of the software life cycle. For example, many problems can be prevented by ensuring that unnecessary utility software is shut off on servers, and that auditing and logging services are always turned on.

D) Maintenance:

Finally, in the maintenance phase of the life cycle, a focus on the OWASP model will ensure that organizations conduct ongoing reviews and code scanning; to find out if changes to the application over time might have created any new vulnerabilities.

In short, integrating the OWASP into every phase of the software development life cycle forces development organizations to adopt security best practices and learn how to use software testing tools. These best practices and testing tools help eliminate mitigate the risks, not just of the OWASP project, but for many types of security risks.

III.SECURITY RISKS

It is also necessary to discuss about three security risks and how to prevented application from these risks:

A) A1-Injection [4]:

Threat Agents:

Consider anyone who can send untrusted data to the system, including external users, internal users, and administrators.

Attack Vectors:

Attacker sends simple text-based attacks that exploit the syntax of the targeted interpreter. Almost any source of data can be an injection vector, including internal sources.

Security Weakness:

Injection flaws occur when an application sends untrusted data to an interpreter. Injection flaws are very prevalent, particularly in legacy code. They are often found in SQL queries. Injection flaws are easy to discover when examining code, but frequently hard to discover via testing. Scanners and fuzzers can help attackers find injection flaws.

Impacts on application:

Injection can result in data loss or corruption, lack of accountability, or denial of access. Injection can sometimes lead to complete host takeover. All data could be stolen, modified, or deleted.

Vulnerable To 'Injection:

The best way to find out if an application is vulnerable to injection is to verify that all use of interpreters clearly separates untrusted 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. Penetration testers can validate these issues by crafting exploits that confirm the vulnerability.

Automated dynamic scanning which exercises the application may provide insight into whether some exploitable injection flaws exist. Scanners cannot always reach interpreters and have difficulty detecting whether an attack was successful. Poor error handling makes injection flaws easier to discover

Example Attack Scenarios:

Scenario #1: The application uses untrusted data in the construction of the following vulnerable SQL call:

String query = "select * from userreg where name='" + request.getParameter ("name") + "'";
Scenario #2: Similarly, an application's blind trust in frameworks may result in queries that are still vulnerable, (e.g., Hibernate Query Language (HQL)):
Query HQLQuery = session.createQuery ("FROM userreg where name='" + request.getParameter ("name") + "'");
In both cases, the attacker modifies the 'id' parameter value in her browser to send: ' or '1'='1. For example: http://example.com/app/accountView?id=' or '1'='1
This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could modify data or even invoke stored procedures.

Preventing Injection Attack:
Preventing injection requires keeping untrusted data separate from commands and queries.
1. The preferred option is to use a safe API, which avoids the use of the interpreter entirely or provides a parameterized interface. Be careful with APIs, such as stored procedures that are parameterized, but can still introduce injection under the hood.
2. If a parameterized API is not available, you should carefully escape special characters using the specific escape syntax for that interpreter. OWASP's ESAPI provides many of these escaping routines.
3. Positive or "white list" input validation is also recommended, but is not a complete defense as many applications require special characters in their input. If special characters are required, only approaches 1 and 2 above will make their use safe.
To prevent web application from SQL Injection parameters is used in SQL Query as:
PreparedStatement ps=con.prepareStatement ("select * from userreg where name=? and pass=?");
ps.setString (1, name);
ps.setString (2, pass);
ResultSet rs=ps.executeQuery ();

B)A3-Cross-Site Scripting (XSS) [5]:

Threat Agents:
Consider anyone who can send untrusted data to the system, including external users, internal users, and administrators

Attack Vectors:
Attacker sends text-based attack scripts like javascripts that exploit the interpreter in the browser. Almost any source of data can be an attack vector, including internal sources such as data from the database.

Security Weakness:
XSS is the most prevalent web application security flaw. XSS flaws occur when an application includes user supplied data in a page sent to the browser without properly validating or escaping that content. There are two different types of XSS flaws: 1) Stored and 2) Reflected, and each of these can occur on the a) Server or b) on the Client.
Detection of most Server XSS flaws is fairly easy via testing or code analysis. Client XSS is very difficult to identify.

Impacts on application:
Attackers can execute scripts in a victim's browser to hijack user sessions, deface web sites, insert hostile content, redirect users, hijack the user's browser using malware, etc.

Vulnerable To 'Cross-Site Scripting':
A system is vulnerable if it does not ensure that all user supplied input is properly escaped, or it does not verify it to be safe via input validation, before including that input in the output page. Without proper output escaping or validation, such input will be treated as active content in the browser. If Ajax is being used to dynamically update the page, are you using safe JavaScript APIs? For unsafe JavaScript APIs, encoding or validation must also be used.
Automated tools can find some XSS problems Automated tools can find some XSS problems automatically. However, each application builds output pages differently and uses different browser side interpreters such as JavaScript, ActiveX, Flash, and Silver light, making automated detection difficult. Therefore, complete coverage requires a combination of manual code review and penetration testing, in addition to automated approaches. Web 2.0 technologies, such as Ajax, make XSS much more difficult to detect via automated tools.

Example Attack Scenarios:
The application uses untrusted 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 ?foo='+document.cookie</script>'.
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 automated CSRF defense the application

Preventing 'Cross-Site Scripting' :
Preventing XSS requires separation of untrusted data from active browser content.
1. The preferred option is to properly escape all untrusted data based on the HTML context (body, attribute, JavaScript, CSS, or URL) that the data will be placed into. See the OWASP XSS Prevention Cheat Sheet for details on the required data escaping techniques.
2. Positive or "white list" input validation is also recommended as it 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, validate the length, characters, format, and business rules on that data before accepting the input.
3. For rich content, consider auto-sanitization libraries like OWASP's AntiSamy or the Java HTML Sanitizer Project.
4. Consider Content Security Policy (CSP) to defend against XSS across your entire site.
To prevent web application from Cross-Site Scripting AntiSamy filter is used in application. AntiSamy is a library for HTML and CSS encoding. The OWASP AntiSamy

project is a few things. Technically, it is an API for ensuring user-supplied HTML/CSS is in compliance within an

application's rules .t's an API that helps you make sure that clients don't supply malicious cargo code in the HTML they supply for their profile, comments, etc., that get persisted on the server. The term "malicious code" in regards to web applications usually mean "JavaScript." Cascading Stylesheets are only considered malicious when they invoke the JavaScript engine.

C)A6 - Security Misconfiguration [6]:

Threat Agents:

Consider anonymous external attackers as well as users with their own accounts that may attempt to compromise the system. Also consider insiders wanting to disguise their actions.

Attack Vectors:

Attacker accesses default accounts, unused pages, unpatched flaws, unprotected files and directories, etc. to gain unauthorized access to or knowledge of the system.

Security Weakness:

Security misconfiguration can happen at any level of an application stack, including the platform, web server, application server, database, framework, and custom code. Developers and system administrators need to work together to ensure that the entire stack is configured properly. Automated scanners are useful for detecting missing patches, misconfigurations, use of default accounts, unnecessary services, etc.

Impacts on application:

The system could be completely compromised without you knowing it. All of your data could be stolen or modified slowly over time.

Recovery costs could be expensive

Vulnerable To 'Security Misconfiguration':

Following questions needs to be answered for analyzing vulnerability to security misconfiguration:

1. Is any of the software is out of date? This includes the OS, Web/App Server, DBMS, applications, and all code libraries (see new A9).
2. Are any unnecessary features enabled or installed (e.g., ports, services, pages, accounts, privileges)?
3. Are default accounts and their passwords still enabled and unchanged?
4. Does error handling reveal stack traces or other overly informative error messages to users?
5. Are the security settings in your development frameworks (e.g., Struts, Spring, ASP.NET) and libraries not set to secure values?

Without a concerted, repeatable application security configuration process, systems are at a higher risk.

Example Attack Scenarios:

Scenario #1: The app server admin console is automatically installed and not removed. Default accounts aren't changed. Attacker discovers the standard admin pages are on your server, logs in with default passwords, and takes over.

Scenario #2: Directory listing is not disabled on your server. Attacker discovers she can simply list directories to find any file. Attacker finds and downloads all your compiled Java classes, which she decompiles and reverse engineers to get all your custom code. She then finds a serious access control flaw in your application.

Scenario #3: App server configuration allows stack traces to be returned to users, potentially exposing underlying flaws. Attackers love the extra information error messages provide.

To Prevent 'Security Misconfiguration' :

The primary recommendations are to establish all of the following:

1. A repeatable hardening process that makes it fast and easy to deploy another environment that is properly locked down. Development, QA, and production environments should all be configured identically (with different passwords used in each environment). This process should be automated to minimize the effort required to setup a new secure environment.
2. A process for keeping abreast of and deploying all new software updates and patches in a timely manner to each deployed environment. This needs to include all code libraries as well (see new A9).
3. A strong application architecture that provides effective, secure separation between components.
4. Consider running scans and doing audits periodically to help detect future misconfigurations or missing patches.

IV.CONCLUSION

This paper presents a comprehensive survey of recent research results in the area of web application security. We described about security threats in web applications, and implementation of some of OWASP security properties like SQL Injection, Cross-Site Scripting, and Security misconfiguration to make secure Web application. The outcome of the above research has been implemented in **http://serbonline.in** web application. In future, other security features of OWASP [2] will be implemented.

REFERENCES

[1] https://www.owasp.org/images/c/c4/OWASP- Italy Day E Gov09_04_Morana.pdf
[2] https://www.owasp.org/index.php/Top_10_2013-Top_10
[3] http://www.darkreading.com/vulnerabilities---threats/10-web-threats-that-could-harm-your-business/d/d-id/1139318
[4] https://www.acunetix.com/websitesecurity/sql-injection/
[5] https://www.acunetix.com/websitesecurity/cross-site-scriptin/
[6] https://www.owasp.org/index.php/Insecure_Configuration_ Management
[7] Boyd, Stephen W., and Angelos D. Keromytis. "SQLrand: Preventing SQL injection attacks." Applied Cryptography and Network Security. Springer Berlin Heidelberg, 2004.
[8] Morana, Marco. "2013 AppSec Guide and CISO Survey: Making OWASP Visible to CISOs." AppSec USA 2013. Owasp, 2013.
[9] Thompson, Bill. "Leveraging OWASP in Open Source Projects-CAS AppSec Working Group." AppSec USA 2013. Owasp, 2013.