International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 07 Issue: 09 | Sep 2020 www.irjet.net p-ISSN: 2395-0072

Critical Web Application Security Risks

Sarthak Javeri[1] and Prof. Meenakshi Garg[2]

[1]Student, [2]Associate Professor
[1]Department of MCA, [2]Department of MCA
[1]Vivekanand Education Society's Institute of Technology (VESIT), Mumbai, India.
[2]Vivekanand Education Society's Institute of Technology (VESIT), Mumbai, India.

---***---

Abstract - *Web applications are one among the foremost prevalent platforms for information and services delivery over Internet today. As they're increasingly used for critical services, web applications become a popular and valuable target for security attacks. There's little effort dedicated to drawing connections among these techniques and building an enormous picture of web application security research. This paper surveys the planet of web application security, with the aim of systematizing the prevailing techniques into an enormous picture that promotes future research. We first present the vulnerabilities that can lead to an unsecured web application and how application can be exploited. Finally, we discuss the technique which can be used for preventing those vulnerabilities and make application more secured*

Key Words — Web applications, Security testing, Vulnerabilities

1. INTRODUCTION

In Recent years, we've witnessed rapid diffusion of internet which produces significant demand of web applications with enforced security. Thanks to which there's a rise within the number of vulnerabilities in web applications which may be exploited by attackers so on gain unauthorized access to the online sites and web applications. Modern Web systems are really complex, distributed and heterogeneous, interactive and responsive, ever evolving, and rapidly changed. Web domain is pervasive and dynamic in nature which makes it more susceptible to malevolent actions like security breaches, Trojans etc. within the light of diversification of the online applications, security becomes a critical issue and is said to the standard of the online application. So we will say that security becomes an elusive goal. Understanding vulnerabilities like cross-site scripting, SQL injection, broken authentication helps us understand the critical risks. Thanks to the big increase within the web application vulnerabilities, there are various threats and challenges being faced which may cause a severe setback to the integrity, confidentiality and security of the online applications. So as to plan any effective methodology or techniques for web security testing, we should always first understand the various web application security risks.

The goal of the paper is to debate about various web application security risk and the way to stop them to make a secure web application.

2. INJECTION

2.1 Am I vulnerable to Injection?

•User supplied data isn't validated, filtered or sanitized by the appliance.

Hostile data is directly used or concatenated, or in stored procedures.

•Hostile data is used within ORM search parameters such the search evaluates bent include sensitive or all records.

•Hostile data is directly used or concatenated and hostile data in dynamic queries, commands, or in stored procedures.

2.2 How Do I Prevent Injection?

•Positive or "white list" input validation, but this is often often not a whole defense as many applications require special characters, like text areas or APIs for mobile applications

•For any residual dynamic queries, escape special characters using the precise escape syntax for that interpreter. Java Encoder provide such escaping routines. NB: SQL structure like table names, column names, then on cannot be escaped, and thus user-supplied structure names are dangerous. This is often often a typical issue in report writing software.

•Use LIMIT and other SQL controls within queries to prevent mass disclosure of records just in case of SQL injection.

3. Broken Authentication

3.1 Am I susceptible to Broken Auth?

•Permits credential stuffing, which is where the attacker features an inventory of valid usernames and passwords.

•Permits brute force or other automated attacks.

•Permits default, weak or well-known passwords, like "Password1" or "admin/admin“.

•Uses weak or ineffectual credential recovery and forgot password processes, like "knowledge-based answers", which cannot be made safe.

•Uses plain text, encrypted, or weakly hashed passwords permit the rapid recovery of passwords using GPU crackers or brute force tools.

•Has missing or ineffective multi-factor authentication.

3.2 How Do I Prevent This?

•Don't deploy with any default credentials, particularly for admin users

•Store passwords employing a contemporary a way hash function with sufficient work factor to prevent realistic GPU cracking attacks.

•Implement weak password checks, like testing new or changed passwords against a listing of the very best 10000 worst passwords.

•Align password length, complexity and rotation policies with NIST 800-63 B's, evidence based password policies

•Where possible, implement multi-factor authentication to prevent credential stuffing, brute force, automated, and stolen credential attacks

•Log authentication failures and alert administrators when credential stuffing, brute force, other attacks are detected.

4. Sensitive Data Exposure

4.1 Am I susceptible to Data Exposure?

•Is any data of a site transmitted in clear text, internally or externally? Internet traffic is particularly dangerous, but from load balancers to web servers or from web servers to rear systems are often problematic.

•Is sensitive data stored in clear text, including backups?

•Are unspecified or weak cryptographic algorithms used either by default or in older code? (see A6:2017 Security Misconfiguration)

•Are given crypto keys in use, generic crypto keys generated or re-used, or is proper key management or rotation missing?

•Is encryption not enforced, e.g. are any user agent security directives or headers missing?

4.2 How Do I Prevent This?

•Classify data processed, stored or transmitted by a system. Apply controls as per the classification.

•Understand the privacy laws or regulations applicable to sensitive data, and protect as per regulatory requirements

•Don't store sensitive data unnecessarily .Discard it as soon as possible or use PCI DSS compliant tokenization or maybe truncation. Data you don't retain can't be stolen.

•Make sure that critical data is encrypted

•Encrypt all data in transit, like using TLS. Enforce this using directives like HSTS

•Ensure up-to-date and powerful standard algorithms or ciphers, parameters, protocols and keys are used, and proper key management is in situ. Think about using crypto modules.

•Ensure passwords are stored with a robust adaptive algorithm appropriate for password protection, like Argon2, scrypt, bcryptand PBKDF2. Configure the work factor (delay factor) as high as you'll tolerate.

•Disable caching for response that contains sensitive data.

•Verify independently the effectiveness of your settings.

5. XML External Entities

5.1 Am I susceptible to XXE?

• Application accepts XML directly or XML uploads, untrusted sources, or inserts untrusted data into XML documents, which is then parsed by an XML processor

•Any SOAP based web services has document type definitions (DTDs) enabled. Because the exact mechanism for disabling DTD processing varies by processor, it's recommended that you simply consult a reference like the OWASP XXE Prevention Cheat Sheet.

•If your application uses SOAP before version 1.2, it's likely vulnerable to XXE attacks if XML entities are being passed to the SOAP framework.

•SAST tools can help detect XXE in ASCII text file, although manual code review is that the best alternative in large, complex apps with much integration.

•Being susceptible to XXE attacks likely means you're susceptible to other billion laughs denial-of-service attacks.

5.2 How Do I Prevent This?

•Disable XML external entity and DTD processing altogether XML parsers in your application, as per the OWASP XXE Prevention Cheat Sheet.

•Implement white listing input validation, filtering, or sanitization to stop hostile data within XML documents, headers, or nodes.

•Patch or upgrade all the most recent XML processors and libraries in use by the app or on the underlying OS. The utilization of dependency checkers is critical in managing the danger from necessary libraries and components in not only your app, but any downstream integrations.

•Upgrade SOAP to the newest version.

6. Broken Access Control

6.1 Am I susceptible to Broken Access Ctl?

•Bypassing access control checks by modifying the URL, internal app state, or the HTML page, or just employing a custom API attack tool.

•Allowing the first key to be changed to another's user's record, like viewing or editing someone else's account.

•Elevation of privilege. Performing users tasks without being logged in, or acting as an admin when logged in as a user.

•Metadata manipulation, like tampering with a JWT access control token or a cookie or hidden field manipulated to escalate privileges.

•Force browsing to authenticated pages as an unauthenticated user, or to privileged pages as a typical user or API not enforcing access controls for POST, PUT and DELETE

6.2 How Do I Prevent This?

•Implement access control mechanisms once and re-use them throughout the appliance.

•Enforce record ownership, instead of accepting that the user can perform all actions on file

•Domain access controls are unique to every application, but business limit requirements should be enforced by domain models

•Disable web server directory listing, and ensure file metadata such (e.g. .git) isn't present within web roots

• Alert admins when repeated failures happen in log

•Rate limiting API and controller access to attenuate the harm from automated attack tooling.

7. Security Misconfiguration

7.1 Am I susceptible to Security Misconfiguration?

•Are any unnecessary features enabled or installed (e.g. ports, services, pages, accounts, privileges)?

•Are default accounts and their passwords still in use ?

•Does your error handling reveal informative error messages ?

•Do still you user older configs with updated software? Does one continue to support obsolete backward compatibility?

•Are the safety settings in your application servers, application frameworks (e.g. Struts, Spring, ASP.NET), libraries, databases, etc. not set to secure values?

•Is any of your software out of date? (see A9:2017Using Components with Known Vulnerabilities)

7.2 How Do I Prevent This?

•A repeatable hardening process that creates it fast and straightforward to deploy another environment that's properly locked down. QA, and production environments should all be configured identically (with different credentials utilized in each environment). This process should be automated to attenuate the difficulty required to setup a replacement secure environment.

•Remove or don't install any unnecessary features, components, documentation and samples. Remove unused dependencies and frameworks.

•A strong application architecture that provides effective, secure separation between components or cloud security groups.

8. Cross-Site Scripting (XXS)

8.1 Am I Vulnerable XSS?

Reflected XSS: Your app or API includes unvalidated and unescaped user input as a neighborhood of HTML output or there is no content security policy (CSP) header. A successful attack can allow the attacker to execute arbitrary HTML and JavaScript within the victim's browser. Typically the user will need to interact with a link, or another attacker controlled page, sort of a watering hole attack, malvertizing, or similar.

Stored XSS: Your app or API stores unsanitized user input that's viewed at a later time by another user or an administrator. Stored XSS is typically considered a high or critical risk.

DOM XSS: JavaScript frameworks, single page apps, and APIs that dynamically include attacker-controllable data to a page are vulnerable to DOM XSS. Ideally, you'd avoid sending attacker-controllable data to unsafe JavaScript APIs.

8.2 How Do I Prevent This?

•Use safer frameworks that automatically escape for XSS intentionally, like in Ruby 3.0 or React JS.

•Escaping untrusted HTTP request data supported the context within the HTML output will resolve Reflected and Stored XSS vulnerabilities..

•Enabling CSP could also be a defense thorough mitigating control against XSS, assuming no other vulnerabilities exist which may allow placing malicious code via local file include like path traversal overwrites, or vulnerable libraries in permitted sources, like content delivery network or local libraries.

•An automated process to verify the effectiveness of the configurations and settings altogether environments.

9. Insecure Deserialization

9.1 Am I vulnerable to Insecure Deserialization?

•The serialization mechanism allows for the creation of arbitrary data types, AND

•There are classes available to the appliance which can be chained together to vary application behavior during or after deserialization, or unintended content are often used to influence application behavior, AND

•The application or API accepts and deserializes hostile objects supplied by an attacker, or an application uses serialized opaque client side state without appropriate tamper resistant controls. OR

•Security state sent to an untrusted client without some kind of integrity control is perhaps going vulnerable to deserialization

9.2 How Do I Prevent This?

•Implement encryption of the serialized objects to prevent data tampering

•Isolate code that deserializes, such it runs in very low privilege environments, like temporary containers.

•Log deserialization exceptions and failures, like where the incoming type is not the expected type, or the deserialization throws exceptions.

•Monitor deserialization, alerting if a user deserializes constantly.

10. Using Components with Known Vulnerabilities

10.1 Am I susceptible to Known Vulnerabilities?

•If you are doing not know the versions of all components you employ (both client-side and server-side).

•If any of your software out of date? This includes the OS, Web/App Server, DBMS, applications, APIs and every one components, runtime environments and libraries.

•If you are doing not know if they're vulnerable. Either if you don't research for this information or if you don't scan them for vulnerabilities on a daily base.

•If you are doing not fix nor upgrade the underlying platform, frameworks and dependencies during a timely fashion. This happens is environments where patching isn't done regularly, which leaves organizations hospitable many days or months of unnecessary exposure to fixed vulnerabilities. This is often likely the basis explanation for one among the most important breaches of all time.

•If you are doing not secure the components' configurations (seeA6:2017-Security Misconfiguration).

10.2 How Do I Prevent This?

•Remove unnecessary data components

•Continuously inventory the versions of both client-side and server-side components and their dependencies using tools like versions, Dependency Check, retire.js, etc.

•Continuously monitor sources for vulnerabilities in your components. Use automated software composition analysis tools

•Only obtain your components from official sources and, when possible, prefer signed packages to scale back the prospect of getting a modified, malicious component.

11. References:

1. https://www.lifelock.com/education/history-of-databreaches/
2. https://www.ibm.com/security/data-breach
3. https://info.whitehatsec.com/rs/675-YBI674/images/WHS%202017%20Application%20Sec urity%20Report%20FINAL.pdf
4. Source: Building a web application security program from Securosis.com
5. Source: Open Web Application Security Project
6. https://usa.kaspersky.com/about/pressreleases/2017_kaspersky-lab-report-on-ddosattacks-in-q1-2017-the-lull-before-the-storm
7. Source: Gartner presentation on SAST and DAST