OWASP Security Threats / Vulnerabilities Document

Project D8Finder

Introduction

With awareness of the importance of cyber security continues to steadily increase, more and more high profile cyber security incidents being made public. Therefore, we should prepare and analyse the cyber threats which makes the project insecure. In this report, we will introduce the most top 10 security threats of web application based on OWASP and point out which of threats makes our project at risk. We will also outlines the point why it is vulnerable in our project and how to resolve the issue.

Top 10 most critical threats for web application by OWASP

1. A1 Injection : Injection flaws, such as SQL, NoSQL, 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 data without proper authorization.

2. A2 Broken Authentication : Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities temporarily or permanently.

3. A3 Sensitive Data Exposure : Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.

4. A4 XML External Entities (XXE) : Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.

5. A5 Broken Access Control : Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.

6. A6 Security Misconfiguration : Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating

systems, frameworks, libraries, and applications be securely configured, but they must be patched and upgraded in a timely fashion.

7. A7 Cross-Site Scripting (XSS) : XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. 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.

8. A8 Insecure Deserialization : Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.

9. A9 Using Components with Known Vulnerabilities : Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.

10. A10 Insufficient Logging & Monitoring : Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

The most critical security threats in our application

1. A1 Injection threat - Injection threat is vulnerable to our application because we use SQL command and it is exposed without special protection method. Changing SQL command is the most common threat and the example of this threat is as follow:

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

   String query = "SELECT * FROM accounts WHERE custID='" + request.getParameter("id") + "'";

   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 accounts WHERE custID='" + request.getParameter("id") + "'"); In both cases, the attacker modifies the 'id' parameter value in their 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 or delete data, or even invoke stored procedures. Therefore, we need to prepare for this threat to prevent data changes or any harmful actions.

- Solution : the best method of detecting if application are vulnerable is source code review. Automated testing of all parameters, header, URL, cookies, JSON, SOAP and XML data inputs are needed. Prefered options is to use a safe API which avoids the use of the interpreter entirely or provides a parameterized interface or migrate to uses Object Relational Mapping Tools (ORMs). For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter. Lastly, use LIMIT and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection.

2. A2 Broken Authentication - Our application is vulnerable to this threat. This is because we have weak combination of password, weakly hashed passwords and exposed session ID. The example attack scenarios which can caused from this as follow:

Scenario #1: Credential stuffing, the use of lists of known passwords, is a common attack. If an application does not implement automated threat or credential stuffing protections, the application can be used as a password oracle to determine if the credentials are valid.
Scenario #2: Application session timeouts aren't set properly. A user uses a public computer to access an application. Instead of selecting "logout" the user simply closes the browser tab and walks away. An attacker uses the same browser an hour later, and the user is still authenticated.

- Solution : Implementing multi-factor authentication to prevent automated, credential stuffing, brute force, and stolen credential re-use attacks in possible situation. Align password length, complexity and rotation policies with NIST 800-63 B's guidelines in section 5.1.1 for Memorized Secrets or other modern, evidence based password policies. Limit or increasingly delay failed login attempts. Log all failures and alert administrators when credential stuffing, brute force, or other attacks are detected.

3. A3 Sensitive Data Exposure - Our application is vulnerable to this threat because our app have user's private information such as their education level, preference type, birthdate, email address etc. All this information is stored in clear text in our database therefore we need to solve this problem to prevent any further issues. The example attack scenarios which can caused from this as follow:

Scenario #1 : The password database uses unsalted or simple hashes to store everyone's passwords. A file upload flaw allows an attacker to retrieve the password database. All the unsalted hashes can be exposed with a rainbow table of pre-calculated hashes. Hashes generated by simple or fast hash functions may be cracked by GPUs, even if they were salted.

Scenario #2 : An application encrypts credit card numbers in a database using automatic database encryption. However, this data is automatically decrypted when retrieved, allowing an SQL injection flaw to retrieve credit card numbers in clear text.

- Solution: Classify data processed, stored, or transmitted by an application. Identify which data is sensitive according to privacy laws, regulatory requirements, or business needs. Make sure to encrypt all sensitive data at rest. Ensure up-to-date and strong standard algorithms, protocols, and keys are in place; use proper key management.Store passwords using strong adaptive and salted hashing functions with a work factor (delay factor), such as Argon2, scrypt, bcrypt, or PBKDF2.

4. A9 Using Components with Known Vulnerabilities - Our application has not implemented any testing method to check if our application is up-to-date and supported. Also we have not scanned for vulnerabilities regularly and subscribe to security bulletins related to the components we use. The example attack scenarios which can caused from this as follow:

Scenario #1: Components typically run with the same privileges as the application itself, so flaws in any component can result in serious impact. Such flaws can be accidental (e.g. coding error) or intentional (e.g. backdoor in component).

Some example exploitable component vulnerabilities discovered are:

• CVE-2017-5638, a Struts 2 remote code execution vulnerability that enables execution of arbitrary code on the server, has been blamed for significant breaches.
• While internet of things (IoT) are frequently difficult or impossible to patch, the importance of patching them can be great (e.g. biomedical devices).

There are automated tools to help attackers find unpatched or misconfigured systems. For example, the Shodan IoT search engine can help you find devices that still suffer from the Heartbleed vulnerability that was patched in April 2014.

- Solution : Remove unused dependencies, unnecessary features, components, files, and documentation. Continuously inventory the versions of both client-side and server-side components (e.g. frameworks, libraries) and their dependencies using tools like versions, DependencyCheck, retire.js, etc. Continuously monitor sources like CVE and NVD for vulnerabilities in the components. Use software composition analysis tools to automate the process. Subscribe to email alerts for security vulnerabilities related to components you use.

Conclusion

In this report, we identified and outlined the most critical security threats to our application and found out its solutions. The secure environment is the most important thing to users. Therefore, we must create a secure and up-to-date web application. We will follow the guideline to prevent any issues cause by security threats and make a secure and better environment for our users.

References

1 . OWASP Top 10 - 2017 The Ten Most Critical Web Application Security Risks [Internet]. OWASP; 2017 [cited 10 October 2018]. Available from: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf

2. OWASP [Internet]. Owasp.org. 2018 [cited 10 October 2018]. Available from: https://www.owasp.org/index.php/Main_Page