**Aim: Study of OWASP vulnerabilities.**

**Theory:** OWASP Vulnerability.

**Injection**

An injection is a security risk that you can find on pretty much any target. Basically, it happens when a server-side interpreter processes untrusted user input as part of a command or a query. There are many vulnerabilities which cause injection. Here are some examples:

⦁SQL injection: You can find a SQL injection when the developer runs a SQL query that takes a parameter you control as an input. If you successfully exploit it, you steal data from the database, edit it or delete it altogether.

⦁OS command injection: It happens when user input is used as part of an insecure call to operating system commands. If you find one, you can run arbitrary operating system commands on the vulnerable server.

⦁XPATH injection: It targets the query language typically used in XML. When you can control part of the query. Therefore, you can bypass restrictions, read unauthorized XML nodes, etc.

⦁Server-Side Template Injection: This flaw affects applications which use template engines to render server-side data. If you can control variables passed into the template, you can achieve remote code execution.

⦁LDAP Injection: When your target insecurely uses some user input to query an LDAP directory, you can perform an injection to bypass restrictions, read unauthorized data, etc.

Injection Mitigation:

⦁Making use of Prepared Statements with Parameterized queries.

⦁Making use of Stored Procedures.

⦁Implement input validation and sanitization.

⦁Make sure you are escaping all user-supplied input.

Broken authentication and session management

Authentication is a feature which verifies an identity’s claims. For example, when you login into an application, it uses your username and password to verify that you are indeed who you are claiming to be. Upon authentication, and due to the stateless nature of HTTP, the application provides you with a session representing your identity, which your web browser sends on your subsequent requests.

Of course, you need to be able to sign up, log in, reset your password or enable Multi-Factor authentication. That’s why authentication is hard to implement without making any mistakes. Any flaw in one of those features can lead to broken authentication.

**Broken Authentication Mitigation:**

⦁Making use of captcha.

⦁Reduce the number of tries for a particular user based on the session ID or the IP.

⦁Blocking multiple requests coming from the same IP.

⦁Making the admin login page inaccessible to the public.

⦁Implement multi-factor authentication to prevent brute-forcing and credential theft.

**Sensitive data exposure**

If IT assets disclose data which is not meant to be publicly accessible, they suffer from sensitive data exposure. On the one hand, this data can be at rest, like databases or files. On the other hand, it can be in transit, especially if unencrypted or weak encrypted data is used for transmission.

**Sensitive Data Exposure Mitigation:**

⦁Always identify and classify which data is sensitive according to the privacy laws and the regulatory requirements.

⦁Immediately remove any data that is not needed to be stored.

⦁If you are going to store any sensitive data, make sure it is encrypted at rest.

⦁Use proper key management.

⦁Make sure you encrypt all data in transit with security protocols such as TLS and SSL.

⦁You can enforce encryption on your application simply by using HTTP Strict Transport Security (HSTS).

⦁Do not cache sensitive data.

⦁Always store passwords with different encryption methods.

**XML-External Entity (XXE)**

XXE is a flaw in the way XML parsers get configured. Specifically, this vulnerability happens when the XML parser can evaluate DTDs and external entities. It allows an attacker to achieve many exploits, like listing directories and reading files from the server. It can even provoke a Denial of Service.

*The XML external entities (XXE) attack can exploit these:*

⦁Vulnerable XML parser that allows an attacker to upload XML or include a hostile command in an XML document.

⦁Vulnerable integration.

⦁Vulnerable dependencies.

⦁Vulnerable code.

**XXE Injection Mitigation:**

⦁You must disable DTD and XML external entity features.

⦁All the XML processors and libraries used within the application must always be updated and patched.

⦁There is a need to put a proper validation in place for every user input.

⦁Use good XML parsers that are not vulnerable by default.

⦁Make use of a very good SAST tool that can help detect XXE in your source code.

**Broken access control**

Broken access control happens when the application allows a user to perform unauthorized actions. There are many vulnerabilities which contribute to this risk, For instance, if the developer forgets to validate permissions when dealing with identifiers, the application becomes vulnerable to Insecure Direct Object Reference (IDOR).Other vulnerabilities include Cross-site Request Forgery (CSRF), Cross-Origin Resource Sharing (CORS) misconfigurations and forced browsing. Read more about them in the dedicated blog post.

**Broken Access Control Mitigation:**

⦁Use a proper session management method.

⦁Use a token for authorization of users like JWT.

⦁Always deny public access by default except in rare cases for some resources that need to be accessed publicly.

⦁Regular audit and test access controls should be conducted to confirm its functionality.

⦁Disable the web-server directory listing and confirm backup files are not present in the web roots.

⦁Make sure you have an access control set up that will enforce the right to every user like what each user can perform and not that the user can create, update, delete and read any record.

⦁There is a need for domain models that will enforce business limit requirements.

**Security misconfiguration**

Security misconfigurations, as the name suggests, expose vulnerabilities due to weak configurations of an IT asset. It doesn’t affect web assets only. Any component which requires a configuration is subject to this vulnerability. This means that network devices, hardware, email services, etc. can suffer from this vulnerability. For instance, your smart door lock can have a predefined default administration PIN code. If you don’t change it, anyone can access and change your device configuration.

In the context of web applications, you can find things like directory listing enabled, which would allow you to list all files and directories. Or maybe the developer forgot to disable the debug mode, allowing you to get more insights on the inner-workings of the vulnerable application.

**Security Misconfiguration Mitigation:**

⦁A regular hardening of the application environment is very important, and it’s fast and easy to deploy another environment that is properly locked down. Each environment should be configured identically, but with different credentials.

⦁Make sure you review and update all the configuration settings appropriate to all security updates and patches that are part of the patch management process.

⦁Making sure you send security directives to clients, e.g. Security Headers.

⦁Create automated process environments to verify the effectiveness of the configuration settings.

**Cross-site Scripting (XSS)**

This is one of the famous client-side vulnerabilities. It allows an attacker to run arbitrary Javascript code on the victim’s web browser. XSS becomes possible when user input ends up inside an HTML page or a piece of Javascript code without proper encoding.

There are basically **three types of XSS**, all of them along with hands-on tutorials are explained further:

⦁Stored XSS happens when the user input gets stored in the application’s datastore, then retrieved back and rendered in a page without proper encoding.

⦁Reflected XSS happens when user input gets directly returned into the HTML page without proper encoding.

⦁DOM XSS happens when user input gets inside a Javascript code. Here, it is possible to exploit XSS even if there is no request made to the server.

**Cross Site Scripting Mitigation:**

⦁We can encode the following characters with HTML entity encoding to prevent any execution of any form.

⦁& –> & amp;

⦁< –> & lt;

⦁–> & gt;

⦁” –> & quot;

⦁‘ → &# x27;

⦁CSS encode and make sure it’s validated before Inputting untrusted data into HTML Style Property Values.

⦁Using frameworks like Ruby on Rails and React JS that escape XSS with ease.

⦁JavaScript encode Before Inputting untrusted data into JavaScript data values.

⦁HTML encodes JSON values in an HTML context and reads the data with JSON.parse.

⦁URL encode Before Inputting Untrusted Data into HTML URL Parameter Values.

⦁Implement Content Security Policy.

⦁Use the HTTPOnly cookie flag.

⦁Deploy firewall that protects against XSS.

**Insecure deserialization**

Insecure deserialization happens when the developer doesn’t check serialized data that a user sends to the application. This is another vulnerability where a lack of user input validation can lead to serious security problems. It is hard to exploit, but when it works, it can lead to either remote code execution or denial of service.

**Insecure Deserialization Mitigation:**

⦁Do not allow serialized objects from unreliable sources.

⦁Always carry out some integrity checks like digital signatures on serialized objects in order to prevent compromising of data and hostile object creation.

⦁Always carry out enforcement of strict type constraints when doing deserialization before the creation of the object.

⦁Make sure you isolate and run code that deserializes in low privilege environments when possible.

⦁Make sure that deserialization exceptions and failures are properly logged, like where the incoming type is not the same as the expected type.

**Using components with known vulnerabilities**

You might have totally secured your own code, but what about the dependencies you are using? Have you checked them or just imported them into your code? There is a high chance that one or more of them are vulnerable.

Unfortunately, using components with known vulnerabilities has led to many serious breaches in the past, and will still cause many breaches to come. But you already have the tools to check for them. For more in-depth knowledge of that, head to this dedicated article.

**Components with Known Vulnerability Mitigation:**

⦁Always remove any unused dependencies, unnecessary features, components, and files.

⦁Always obtain your application components from approved and official sources with secure links. This will reduce the chance of including any malicious component in your application.

⦁Always check and avoid frameworks, libraries, and components that are not maintained and do not have security patches for older versions.

⦁Always use library scanners to test for any vulnerabilities in the application packages you are using.

**Insufficient logging and monitoring**

When a hacker infiltrates a network, IT systems will generate traffic which usually doesn’t correspond to the normal one, unless you are dealing with highly skilled hackers who have time and money to go after your IT infrastructure. If you can’t detect this abnormal behavior as soon as possible, you are essentially giving them enough time to achieve their goal. Read more about this in this blog post.

Logging and monitoring should be part of your essential security infrastructure because you simply cannot defend what you don’t know.

**Insufficient Logging & Monitoring Mitigation:**

⦁Always ensure that every log is captured in a way that a management tool can easily use it.

⦁Every transaction on an application should have an audit trail with integrity controls in place so that transactions cannot be manipulated, even deleted.

⦁There is a need to implement a very effective and efficient monitoring and alerting system which will always inform us of any malicious or suspicious activities so that they can be immediately remediated.

⦁Make sure the audit log does not store your password and any other sensitive content, as this is very risky.

**⦁ the top 10 OWASP vulnerabilities**

**A01:2021**

Broken Access Control moves up from the fifth position to the category with the most serious web application security risk; the contributed data indicates that on average, 3.81% of applications tested had one or more Common Weakness Enumerations (CWEs) with more than 318k occurrences of CWEs in this risk category. The 34 CWEs mapped to Broken Access Control had more occurrences in applications than any other category.

**A02:2021**

Cryptographic Failures shifts up one position to #2, previously known as A3:2017-Sensitive Data Exposure, which was a broad symptom rather than a root cause. The renewed name focuses on failures related to cryptography as it has been implicitly before. This category often leads to sensitive data exposure or system compromise.

**A03:2021**

Injection slides down to the third position. 94% of the applications were tested for some form of injection with a max incidence rate of 19%, an average incidence rate of 3.37%, and the 33 CWEs mapped into this category have the second most occurrences in applications with 274k occurrences. Cross-site Scripting is now part of this category in this edition.

**A04:2021**

Insecure Design is a new category for 2021, with a focus on risks related to design flaws. If we genuinely want to "move left" as an industry, we need more threat modeling, secure design patterns and principles, and reference architectures. An insecure design cannot be fixed by a perfect implementation as by definition, needed security controls were never created to defend against specific attacks.

**A05:2021**

Security Misconfiguration moves up from #6 in the previous edition; 90% of applications were tested for some form of misconfiguration, with an average incidence rate of 4.5%, and over 208k occurrences of CWEs mapped to this risk category. With more shifts into highly configurable software, it's not surprising to see this category move up. The former category for A4:2017-XML External Entities (XXE) is now part of this risk category.\

**A06:2021**

Vulnerable and Outdated Components was previously titled Using Components with Known Vulnerabilities and is #2 in the Top 10 community survey, but also had enough data to make the Top 10 via data analysis. This category moves up from #9 in 2017 and is a known issue that we struggle to test and assess risk. It is the only category not to have any Common Vulnerability and Exposures (CVEs) mapped to the included CWEs, so a default exploit and impact weights of 5.0 are factored into their scores.

**A07:2021**

Identification and Authentication Failures was previously Broken Authentication and is sliding down from the second position, and now includes CWEs that are more related to identification failures. This category is still an integral part of the Top 10, but the increased availability of standardized frameworks seems to be helping.

**A08:2021**

Software and Data Integrity Failures is a new category for 2021, focusing on making assumptions related to software updates, critical data, and CI/CD pipelines without verifying integrity. One of the highest weighted impacts from Common Vulnerability and Exposures/Common Vulnerability Scoring System (CVE/CVSS) data mapped to the 10 CWEs in this category. A8:2017-Insecure Deserialization is now a part of this larger category.

**A09:2021**

Security Logging and Monitoring Failures was previously A10:2017-Insufficient Logging & Monitoring and is added from the Top 10 community survey, moving up from #10 previously. This category is expanded to include more types of failures, is challenging to test for, and isn't well represented in the CVE/CVSS data. However, failures in this category can directly impact visibility, incident alerting, and forensics.

**A10:2021**

Server-Side Request Forgery is added from the Top 10 community survey. The data shows a relatively low incidence rate with above average testing coverage, along with above-average ratings for Exploit and Impact potential. This category represents the scenario where the security community members are telling us this is important, even though it's not illustrated in the data at this time.

**Conclusion:**

Hence we have successfully studied OWASP vulnerabilities and how to mitigate them.