Cybersecurity Assignment 4

Name: Deep Ghadiyali

Roll Number: CS17B011

Q1. Create at least five different patterns of queries which indicate an SQL Injection attack. Explain how.

A1. SQL injection is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database.

1. Retrieving hidden data:

Using SQL injection, attacker can reveal some hidden data that should not be displayed. For example, consider an ecommerce website which has many categories of products. One of the browser request URL:

https://sqli-vulnerable-website.com/products?category=Gifts

This causes the application to make an SQL query to retrieve details of the relevant products from the database:

*SELECT * FROM products WHERE category = 'Gifts' AND released = 1*

Now if we want to see unreleased products in database, we can comment out the part after gift parameter and make released=0.

https://sqli-vulnerable-website.com/products?category=Gifts' AND released = 0 --

So, the injected query will be:

SELECT * FROM products WHERE category = 'Gifts' AND released = 0 -- ' AND released = 1

Even if we don't know about released parameter, we can use *category=Gifts' OR 1=1 –*

It will give all the products in database irrespective of category.

2. Subverting application logic:

Consider an application that lets users log in with a username and password. If a user submits the username user1 and the password passwd1, the application checks the credentials by performing the following SQL query:

*SELECT * FROM users WHERE username = 'user1' AND password = 'passwd1'*

If the query returns the details of a user, then the login is successful. Otherwise, it is rejected.

Here, an attacker can log in as any user without a password simply by using the SQL comment sequence -- to remove the password check from the WHERE clause of the query. For example, submitting the username admin'-- and a blank password results in the following query:

*SELECT * FROM users WHERE username = 'admin'--' AND password = ''*

This query returns the admin user and successfully logs the attacker in as that admin.

3. Retrieving data from other database tables:

In cases where the results of an SQL query are returned within the application's responses, an attacker can leverage an SQL injection vulnerability to retrieve data from other tables within the database. This is done using the UNION keyword, which lets you execute an additional SELECT query and append the results to the original query.

For example, if an application executes the following query containing the user input "Gifts":

SELECT name, description FROM products WHERE category = 'Gifts'

then an attacker can submit the input:

' UNION SELECT username, password FROM users--

This will cause the application to return all usernames and passwords along with the names and descriptions of products.

4. Examining the database:

Following initial identification of an SQL injection vulnerability, it is generally useful to obtain some information about the database itself. This information can often pave the way for further exploitation.

You can query the version details for the database. The way that this is done depends on the database type, so you can infer the database type from whichever technique works. For example, on Oracle you can execute:

*SELECT * FROM v$version*

We can also determine what database tables exist, and which columns they contain. For example, on most databases you can execute the following query to list the tables:

*SELECT * FROM information_schema.tables*

We can use UNION operator to take out information. For example, if an application executes the following query containing the user input "Gifts":

SELECT name, description FROM products WHERE category = 'Gifts'

then an attacker can submit the input:

' UNION SELECT table_schema,table_name FROM information_schema.tables--

This will cause the application to return all table name with corresponding schema from MySQL database.

5. Exploiting blind SQL injection by triggering time delays:

It is often possible to exploit the blind SQL injection vulnerability by triggering time delays conditionally, depending on an injected condition. Because SQL queries are generally processed synchronously by the application, delaying the execution of an SQL query will also delay the HTTP response. This allows attacker to infer the truth of the injected condition based on the time taken before the HTTP response is received.

The techniques for triggering a time delay are highly specific to the type of database being used. On Microsoft SQL Server, input like the following can be used to test a condition and trigger a delay depending on whether the expression is true:

'; IF (1=2) WAITFOR DELAY '0:0:10'--
'; IF (1=1) WAITFOR DELAY '0:0:10'—

The first of these inputs will not trigger a delay, because the condition 1=2 is false. The second input will trigger a delay of 10 seconds, because the condition 1=1 is true.
Using this technique, attacker can retrieve data in the way already described, by systematically testing one character at a time:

'; IF (SELECT COUNT(Username) FROM Users WHERE username = 'Admin' AND SUBSTRING (Password, 1, 1) > 'm') = 1 WAITFOR DELAY '0:0: {delay}'--

Q2. Demonstrate the effectiveness of same-origin policy. What attacks does it prevent?

A2. The same-origin policy is a web browser security mechanism that aims to prevent websites from attacking each other.

The same-origin policy restricts scripts on one origin from accessing data from another origin. An origin consists of a URI scheme, domain and port number.

When a browser sends an HTTP request from one origin to another, any cookies, including authentication session cookies, relevant to the other domain are also sent as part of the request. This means that the response will be generated within the user's session, and include any relevant data that is specific to the user. Without the same-origin policy, if you visited a malicious website, it would be able to read your emails from Gmail, private messages from Facebook, etc. Thus, it precents in leaking of user’s other sensitive data.

Q3. What are the top ten latest OWASP API application security attacks in API level? Explain each with example.

A3.

I. Broken Object Level Authorization:

It is also known as Insecure Direct Object Reference (IDOR) vulnerability, is amongst the topmost API security risks. Improper access controls for assets accessible from the internet make it an easy target for attacker. This is largely due to the lack of strict authorisation controls implementation or no authorisation controls. This issue impacts in varied ways, from data disclosure to full account takeover.
For example, an API call with parameters using value (ID) associated with the resource i.e.

/customers/user/user1/profile

An attacker attempts different usernames guessing through correct accounts

/customers/user/user2/profile
/customers/user/ user3/profile

If the API does not check current user permissions and allows the current call, the current user shall access user2 and user3 user profiles. Based on the user permissions, the resulting access would relate to horizontal or vertical privilege escalation vulnerability.

II. Broken user authentication:

Insecure or missing protection mechanisms for API endpoints causes broken user authentication flaws. The authentication endpoint being always exposed makes it an attractive target for cybercriminals due to the way authentication mechanisms work. Attackers can exploit this vulnerability to steal sensitive data, take over accounts, or impersonate users.

For example, flawed implementation of authentication or session management, weak passwords, authentication susceptible to brute-force attacks etc.

III. Excessive Data Exposure:

APIs disclose varied types of data based on who is requesting. The purpose of what information will be served and to whom this will be

served are the core concepts while designing an API. Insecure implementation or lack of implementation of data filtering leads to disclosing more than required information. Data exposure issues are a common sight, ranging from technology or version information disclosure to high-risk issues related to user preferences or account information.
For example, API returns full data objects as they are stored in the backend database.

IV. Lack of Resources & Rate Limiting:

Attackers can easily abuse an API leading it to consume available resources and making the service unavailable to legitimate users. This could lead to Denial of Service (DoS) attacks.

Public APIs are often the target as the common misconception that there is no authentication removes risks to the application applies commonly. This is also an issue that could be exploited using multiple attack vectors such as brute force attacks, credential stuffing attacks, user enumeration, or other fuzzing techniques. Bots and other automated techniques could end up using the victim's precious computing resources**.**

For example, overloading the API by sending more requests or requests exceeding the field size that it can handle and leads to crash the API service.

V. Broken Function-Level Authorization:

Deploying unique strategies for different services yields complexity that leads to vulnerabilities or increased attack surface. Every API function call should require authorization to verify that the user requesting the data is authorized to send the request on those objects. Attackers use this flaw to find administrative functions that may be hidden otherwise and allow access to higher privilege functions.

For example, in a social media site's API user should be authorized before creating, editing or deleting an account but the request for deleting user profile is misconfigured and authentication is not checked.

GET /social-media/user1 => Authentication checked

DELETE /social-media/user1/profile-photo => Authentication not checked as called internally while deleting profile

VI. Mass Assignment:

Mass assignment issue occurs where the lack of properties filtering allows exposure of internal variables or objects. Due to a lack of restrictions, an endpoint may provide access to other properties outside the authorized scope allowing an attacker to modify objects related to those parameters. This also illustrates the API security risk of how unauthorised users can exploit direct mapping client inputs to internal variables.

For example, if a user account balance value is transmitted in user profile information, an attacker might be able to send a request that changes the account balance.

VII. Security Misconfiguration:

Security misconfiguration vulnerabilities arise due to the use of insecure configuration or misconfiguration in the web application components. This could be either application framework, web server hosting the application or third-party libraries in use. Exploiting configuration weakness in an application leads to security misconfiguration attacks.

For example, environment variables are exploited by other application running on server because of misconfiguration of system variables in server. Lack of patch deployment, Missing security headers etc.

VIII. Injection:

Like web application injection attacks, all injections, i.e., OS Command injection, SQL, NoSQL, and LDAP injections, fall under one of the notable API security risks. Due to the lack of strict input validation on the server-side, malicious input can make way as a query or command to enumerate backend information.

For example, SQL injection exploitation in an web application.

IX. Improper Assets Management:

Like other business assets, APIs go through a lifecycle, and the lack of asset management is related to legacy security vulnerabilities. By not retiring older versions, APIs expose more endpoints, exposing endpoints that are no more needed, not known or forgotten by the teams. This increases the attack surface due to older and newer API versions running alongside.

For example, Desire to maintain backward compatibility forces to leave old APIs running. Old or non-production versions are not properly maintained, but these endpoints still have access to production data.

X. ***Insufficient Logging & Monitoring:***
Logging and monitoring of data is a vital step for audit trials and incident response teams. Due to the lack of logging and monitoring controls, an attacker finding it difficult to attack the victim API has more time by his side, leading to a bigger and high impact attack. This can be easily avoided by logging and analysing the suspicious events early in the attack chain due to stricter monitoring processes. Unfortunately, this is one of the reasons data breaches go unnoticed early in the attack stage.

For example, logging is not properly done in web application and due to which manual analysis is done instead of automatic analysis of attacks and data breaches. This also leads to unnoticed data breaches and attacks at API level.

Q4. Explain how broken authentication leads to attacks.

A4. Authentication is "broken" when attackers are able to compromise passwords, keys or session tokens, user account information, and other details to assume user identities. Due to poor design and implementation of identity and access controls, the prevalence of broken authentication is widespread. Several broken authentication attack examples are listed below:

1. ***Session Hijacking:***
   Verified Session IDs may be hijacked impersonate user identities. If a user forgets to log off from a public computer, any other individual can continue that session using the same Session ID that was previously created for the original user. If the same ID is issued before and after authentication, it may lead to a type of broken authentication attack, known as Session Fixation attacks.

2. ***Session ID URL:***
   The Session ID appears in the website URL, and any individual who accesses the URL through a wired or wireless network, can use it to impersonate the user's identity.

3. ***Credential Stuffing:***
   Sometimes, hackers access a database containing users' user-passwords that are unencrypted, and may often employ tactics to determine if the passwords are valid and functional. This is called credential stuffing, and a secure web application must have protocols that guard against such attempts.

4. ***Password Spraying:***
   Password spraying refers to the use of the most common and weak passwords, such as 'password' or '123456' by hackers trying to access secure accounts. Consequently, minimum password requirements have been introduced to avoid such attacks.

5. ***Phishing Attacks:***
   Hackers phish by sending users links to a website that resembles the original web application, to get users to divulge their login credentials. Phishing attacks can be easily prevented, however, with proper diligence and by verifying the web application in use.

Q5. Look at the history of AWS attacks on amazon cloud. Take any five of them. Analyse them from perspective of what was the attack? What vulnerability led to the same. What can be done to mitigate the occurrence of same again.

A5. These are some of the common vulnerability and possible solution to prevent attacks on AWS. Amazon faced largest DDOS attack in Q1 of 2020.

1. ***Overuse of Public Subnets:***
   Many organizations use the default Virtual Private Cloud (VPC) built into AWS, making few changes to the configuration. When they need to spin up an application, they take the simplest approach: using the VPC's default public subnet.
   However, this approach can be extremely dangerous. Public subnets are routed to Internet gateways, making them accessible via the public Internet. As a result, any sensitive information hosted on the subnet is placed in harm's way.
   Solution: Public subnets are suitable for blogs or simple websites, but not critical applications or databases. Instead, use private subnets and ensure all subnets are properly configured for the security needs of the assets involved.
   If an application needs to be public-facing, you can use a combination of public and private subnets to ensure back-end functions and databases are kept out of the public domain

2. ***IAM Issues:***
   IAM issues are far from being unique to AWS. However, because cloud infrastructure is designed to be accessed remotely, IAM issues can become an even greater risk and present a valid AWS security concern.
   Many organizations don't enable multi-factor authentication for privileged accounts. Worse, insecure privileged accounts are often overused — including for trivial tasks that could be performed from any account — putting them further at risk from social engineering and other credential theft attacks.
   Solution: Use an identity solution provider to centralize authentication and use single sign-on so you don't have to manually create IAM users and attach policies to them.
   For sensitive assets, use a tool to create short term keys that expire after a predetermined period. That way, if access and secret keys are leaked, nobody will have persistent access to your AWS environment.
   From a structural perspective, create different AWS accounts for different environments, and use a single AWS organization to enforce policies for sub environments.

3. ***Misconfigured S3 buckets:***
   The most common problem with S3 buckets is permission misconfiguration. Organizations create S3 buckets for various purposes, and even though the buckets are private by default, we as humans make the buckets publicly writeable and readable to make our jobs easier. With this, there is an inevitable threat of malicious users finding the misconfigured S3 buckets and either dumping all the contents or deleting it. Misconfigured S3 buckets can be disastrous if there are sensitive files, db backups, or application logs in it.
   Solution: If you want to store content in the bucket, that should be accessed publicly, use AWS CloudFront. Keep the bucket private and use it as the origin of CloudFront distribution. With this configuration, you will be allowing only CloudFront to access content from the bucket. Any entities requiring access to the content from the bucket would need a full URI path that includes CloudFront domain and doesn't expose your S3 bucket for enumeration.
   There are other hardening techniques provided by AWS such as signed URLs and signed cookies for serving private contents.

4. ***Exposed Database Origin Servers:***
   The IP addresses of database origin servers should never be available to any person or application unless specifically required.

Unfortunately, these IPs are often available as a result of improperly configured CDNs and other solutions.
One of our pen-testers gave an example of how this can lead to a security breach. While testing an HR system, the tester discovered that origin server IPs were available due to a misconfigured CDN. Once he had the IPs, he discovered that the AWS instance wasn't set up to restrict inbound access to specific applications. Within a short period of time, he was able to access the system directly as a privileged user.
Solution: Secure configuration can be a challenge, because applications and infrastructure can each potentially have thousands of configuration options. pen-testers and cyber criminals use tools — many of which are free and open source — to help them identify when configuration issues exist.
To avoid falling victim to configuration issues, it makes sense for security teams to use a similar approach to identify and resolve configuration issues pre-emptively.

5. ***Server-Side Request Forgery (SSRF):***
   SSRF is an attack that abuses legitimate AWS functionality to gain access to instance metadata. If successful, an attacker may be able to extract credentials for an IAM role attached to the instance, and gain privileged access to the target application.
   An attack similar to this was used in the 2019 CapitalOne breach.
   Solution: Thankfully, preventing SSRF attacks against AWS is straightforward. Only version one of the Instance Metadata Service (IMDSv1) is vulnerable, so simply update to version two (IMDSv2).

6. ***Hanging DNS Records:***
   In the course of working with AWS, most organizations create and delete plenty of S3 buckets. However, when an S3 bucket is deleted, many organizations forget to remove references to it across all subdomains.
   This is a problem, because if a subdomain's DNS records stay pointed to a deleted S3 bucket, an attacker can use subdomain enumeration to identify the issue and simply take over the subdomain.
   Solution: Simply keep track of all S3 buckets using a spreadsheet, and record all references to each bucket. Whenever an S3 bucket is deleted, use the spreadsheet to ensure all references to it are removed.