Common Vulnerabilities and Exploits in Web Applications

What are Web Applications?

Web application is a piece of software that can be accessed by the browser. A Browser is an application that is used to browse the internet.
The web application uses a combination of server-side scripts and client-side scripts to present information. It requires a server to manage requests from the users.
Example: Google Apps, Amazon, YouTube.

Why are Web Applications So Vulnerable to Attacks?

Web apps can be attacked for various reasons, including system flaws resulting from incorrect coding, misconfigured web servers, application design flaws, or failure to validate forms. Any web application has at least some vulnerability that hackers can exploit at a higher level.

Such web application vulnerabilities allow hackers to gain direct and public access to databases that contain valuable information making them a frequent target of attacks.

The use of open source and reliance on application programming interfaces (APIs) also increase security concerns.

Cybercriminals use compromised sites for various purposes: to spread malware; steal sensitive data; implant unauthorized information; commit fraud; to infiltrate a company's internal infrastructure. All this threatens the organization's operation and reputation. Therefore, securing software and eliminating all potential vulnerabilities in a web application is necessary.

The **Open Web Application Security Project (OWASP**) is a nonprofit foundation that provides guidance on how to develop, purchase and maintain trustworthy and secure software applications. OWASP is noted for its popular Top 10 list of web application security vulnerabilities.

The **OWASP Top 10** list of security issues is based on consensus among the developer community of the top security risks. It is updated every few years as risks change and new ones emerge. The list explains the most dangerous web application security flaws and provides recommendations for dealing with them.

Common Vulnerabilities and Exploits in Web Applications are as follows:

1. Broken Access Control:

Broken access control is a type of vulnerability that allows unauthorized users to gain access to sensitive data or systems. This can happen when controls such as authentication and authorization are not properly implemented, or when there are weaknesses in the way these controls are enforced.

Broken access control can lead to a number of serious security issues, including data breaches, theft of sensitive information, and loss of system availability. In some cases, it can also enable an attacker to gain elevated privileges that allow them to perform malicious actions on the systems or data they have accessed.

Organizations need to be aware of the risks posed by broken access control and take steps to prevent it. This includes ensuring that all controls are properly implemented and enforcing least privilege principles to limit the permissions of users and groups. Additionally, regular monitoring of systems and data should be conducted to detect any unauthorized activity.

There are four common types of broken access control vulnerabilities:

- Insecure direct object references
- Lack of restriction on URL parameters
- Mass assignment
- Security misconfiguration

How to prevent broken access control:

1. Implement least privileges.
2. Segment the network.
3. Use MFA
4. Except for public resources, deny by default.
5. Log access control failures, alert admins when appropriate

2. Cryptographic Failure:

Cryptography encompasses the tools and techniques used to protect communication and information exchange to ensure confidentiality, non-repudiation, integrity, and authenticity. Modern cryptographic techniques involve converting plain-text messages into ciphertext that the intended recipient can only decode.

With the rapidly changing threat environment, traditional encryption and obfuscation techniques are susceptible to compromises, exposing sensitive data through a group of potential vulnerabilities known as **cryptographic failures**.

A cryptographic failure is a critical **web application security vulnerability** that exposes sensitive application data on a weak or non-existent cryptographic algorithm. Those can be passwords, patient health records, business secrets, credit card information, email addresses, or other personal user information.

Modern web applications process data at rest and in transit, which require **stringent security controls** for comprehensive threat mitigation. Some deployments employ weak cryptographic techniques that can be cracked within a reasonable time frame. Even with the perfect implementation of cryptographic techniques, users may avoid embracing data protection best practices, subsequently making sensitive information susceptible to sensitive data theft.

Application vulnerabilities that can lead to cryptographic failures include:

- Storing Passwords Using Simple/Unsalted Hashes
- Lack of Protection for Seeds in Pseudo-Random Number Generators
- Broken Chain of Trust
- Insufficient Transport Layer Security

How to prevent cryptographic failures:

1. Always use authenticated encryption instead of just encryption.
2. Classify data processed, stored, or transmitted by an application. Identify which data is sensitive according to privacy laws.
3. Don't store sensitive data unnecessarily. Discard it as soon as possible.
4. Make sure to encrypt all sensitive data at rest.
5. Ensure up-to-date and strong standard algorithms, protocols, and keys are in place; use proper key management.
6. Encrypt all data in transit with secure protocols such as TLS with forward secrecy (FS) ciphers, cipher prioritization by the server, and secure parameters.
7. Disable caching for response that contain sensitive data.

3. Injection:

An injection flaw is one of the security vulnerabilities in web applications that allow an attacker to slip in malicious code through an application all the way to another system. These injections come in various types, including SQL injections, command injections, CRLF injections, LDAP injections.

This type of attack allows an attacker to inject code into a program or query or inject malware onto a computer in order to execute remote commands that can read or modify a database, or change data on a web site.

For example, Many applications use Structured Query Language (SQL) to manage communications with the database. SQL vulnerabilities allow attackers to insert malicious SQL commands to exfiltrate, modify, or delete data. Some hackers use SQL to gain root access to the target system.

SQL injection attacks target servers that hold critical data used by web applications or services. They are particularly dangerous when they expose critical or sensitive data, such as user credentials and personal information. The most common vulnerability enabling SQL injection attacks is using unsanitized user inputs. It is important to strip out any element in user-supplied inputs that the server could execute as SQL code.

How to prevent injection attack:

1. Prevent injection attacks by validating or sanitizing the data submitted by users. Think of it as having a meticulous doorkeeper who rejects suspicious visitors (validation) or cleans up anything dubious they might bring (sanitizing).
2. Use an API that avoids the interpreter entirely or employ a parameterized API. Alternatively, you can shift towards tools that offer object-relational mapping. It's like opting for a safer route to your destination.
3. Implement positive server-side input validation. Consider it a double-check system, ensuring that only the correct data gets through.
4. Use constraints like LIMIT within SQL queries to prevent colossal data exposure during a SQL injection.
5. Lastly, avoid showing detailed error messages that could be a great use for an attacker.

4. Insecure Design:

Lack of effective security controls in the design phase often results in an application being susceptible to many weaknesses, collectively known as **insecure design vulnerabilities**.

Insecure design is the lack of security controls being integrated into the application throughout the development cycle. This can have wide ranging and deep-rooted security consequences as the application itself is not designed with security in mind.

Insecure design encompasses various risks that arise from **ignoring design** and **architectural best practices**, starting from the planning phase before actual implementation.

Insecure design vulnerabilities arise when developers, QA, and/or security teams **fail to anticipate** and **evaluate threats** during the code design phase. These vulnerabilities are also a consequence of the non-adherence of **security best practices** while designing an application. As the threat landscape evolves, mitigating design vulnerabilities requires consistent threat modeling to prevent known attack methods.

Without a secure design, it is difficult to detect and remediate architectural flaws such as:

- Unprotected Storage of Credentials
- Trust Boundary Violations
- Generation of Error Messages Containing Sensitive Information
- Improper Isolation or Compartmentalization

How to prevent insecure design:

1. Security specialists should be consulted at the beginning of a project and throughout the entire development lifecycle
2. Make heavy usage of threat modeling
3. Consider potential attack vectors and the level of exposure.
4. Analyze (and re-analyze) all data flows, particularly ones that resist the threat modeling
5. Establish and use a library of secure design patterns or paved road ready to use components
6. Use threat modeling for critical authentication, access control, business logic, and key flows
7. Integrate security language and controls into user stories.

5. Security Misconfiguration:

A security misconfiguration is when security options are not defined in a way that maximizes security, or when services are deployed with insecure default settings. This can happen in any computing system, software application, as well as in cloud and network infrastructure. Security misconfiguration is a common cause of cyber-attacks and successful data breaches.

Frameworks have made programming easy, reducing the time and effort spent building an application. However, these frameworks have complex configurations, increasing the risk of security misconfigurations. Similarly, open-source code is widely used, and might come with default configurations that compromise security and make the application insecure.

Security misconfiguration is an easy-to-target vulnerability. It is typically easy to detect misconfigured web servers and applications, and hackers can exploit the vulnerabilities they discover to cause significant damage.

This risk of misconfiguration can pose a threat to the entire application stack – network services, platform, web server, application server, database, frameworks, custom code, and pre-installed virtual machines, containers, or storage.

The most common security misconfiguration that occur are listed below:

- Unpatched systems
- Default account settings
- Unencrypted files
- Unsecured devices
- Web application and cloud misconfiguration
- Insufficient firewall protection

How to Prevent Security Misconfiguration:

1. Adopt Repeatable Hardening Processes
2. Automate Repetitive Tasks
3. Conduct Frequent Audits
4. Build Segmented Architecture
5. Avoid Unused Features
6. Regularly Update Software

6. Vulnerable and Outdated Components:

A vulnerable and outdated component is a software component that is no longer being supported by the developer, making it susceptible to security vulnerabilities. Many times, a component has known vulnerabilities that don't get fixed due to a lack of maintainer.

Applications often become vulnerable to attacks because they use outdated software components with known security vulnerabilities. Hackers can exploit these vulnerabilities to gain access to the application's data or to take control of the application entirely.

Outdated software components are also more likely to contain security vulnerabilities, as timely patching is a vital part of security posture.

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 severe data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine the app.

These attacks have become commonplace because it is far easier for an attacker to use a known weakness than create a specific program or attack methodology to search out vulnerabilities themselves.

Components with known vulnerabilities and outdated:

- These components can be defined as the third-party apps or software or platforms that are outdated and contain bugs that are public to all, that is- sites like https://www.exploit-db.com contain the full detail as to how to exploit the bugs to put the security of the whole website under severe threat.
- This vulnerability arises with the fact that a website finds it difficult to code everything while making its website functional like -transaction, location, chats, etc.
- So, in order to ease the process of building the website, many websites use third-party apps which do the tasks. But the main problem is that those apps can be harmful to their system if they are not updated regularly. Components with known vulnerabilities are considered to be one of the top 10 web application vulnerabilities listed by OWASP.
- Many websites security gets compromised when they use components having known vulnerabilities.

How to prevent Vulnerable and Outdated Components:

1. Regular updates of the components of software will prevent this kind of attack.
2. Also, if any threat is detected in any component, immediately isolate that module and release security patches for that.
3. Implement regular monitoring and security assessment testing.
4. You can check vulnerability assessment, and you can perform it periodically, and you can do penetration testing both internal and external to confirm the security of your application in depth.
5. Remove unused dependencies, unnecessary features, components, files, etc.
6. It is recommended to obtain components from official sources over secure links only to ensure security.

7. Identification and Authentication Failures:

Identification failures, from the name itself, suggest the system's lack of ability to identify the user. Likewise, authentication failures connote the application's incapability to validate the user's identity as their own.

Identification and authentication failures are vulnerabilities related to applications' authentication schemes. Such failures can lead to serious and damaging data breaches. Any vulnerability related to an application's authentication scheme, whether it is related to how strong it is or how it is implemented, is called an Identification and Authentication Failure.

Types of Authentication Failure Vulnerabilities:

- Permits automated attacks such as credential stuffing, where the attacker has a list of valid usernames and passwords.
- Permits brute force or other automated attacks.
- Permits default, weak, or well-known passwords, such as"Password1″ or "admin/admin.″
- Uses weak or ineffective credential recovery and forgot-password processes.
- Uses plain text, encrypted, or weakly hashed passwords.
- Has missing or ineffective multi-factor authentication (MFA).
- Exposes session IDs in the URL (e.g., URL rewriting).
- Does not rotate session IDs after successful login.
- Does not properly invalidate session IDs. User sessions or authentication tokens (particularly single sign-on (SSO) tokens) aren't properly invalidated during logout or a period of inactivity.

How to prevent Identification and Authentication Failures:

1. **Implement multi-factor authentication (MFA)** to verify the consumer's identity. Examples include One-Time Password (OTP) messaged or emailed to the user. This step will prevent brute force attacks, credential stuffing, and stolen credential reuse attacks.
2. **Use weak-password checks** by forcing users to include a mix of small letters, capital letters, alphanumeric symbols, and special characters while creating passwords.
3. Place a limit on **failed login attempts** to 3 or a maximum of 5. Alert the system admin if you detect an attack — brute force, credential stuffing, or any other attack.
4. **Ensure that credential recovery**, registration, and API pathways are not vulnerable to account enumeration attacks by using the same message for each outcome.
5. **Generating new random session IDs** with high entropy after login protects against hackers. Remember, those session IDs should not be present in the URL and invalidated after logout.

8. Software and Data Integrity Failures:

Modern software development is characterized by agile principles that encourage the rapid release and update cycles. Agile methodology's core components require strict integrity checks, without which attackers can inject malicious inputs that can potentially impact all deployment pipeline stages. In most instances, insecure design is one of the most commonly attributed causes that lead to a broad category of application security vulnerabilities – collectively known as **software and data integrity failures**.

The complexity of architectures in modern update-release cycles often forces developers to use plugins, modules, and libraries from public repositories, untrusted sources, and content delivery networks.

Due to such complexities, software and data integrity failures (categorized as **design flaws**) occur when critical data and software updates are added to the delivery pipeline without verifying their integrity.

In the absence of adequate validation, software and data integrity failures make applications susceptible to unauthorized information disclosure, system compromise, or insertion of malicious code.

Few scenarios leading to integrity failures include:

- Faulty assumptions of the server-side and client-side components in use
- Outdated or unsupported third-party software
- Insufficient vulnerability scanning
- Erroneous input validation across the pipeline
- Missing framework/platform patches
- Missing unit tests
- Insecure component configurations

How to Prevent Integrity Failure:

1. Use software that was digitally signed by a trusted vendor
2. Use trusted software repositories, or your own repository
3. Verify that your extensions contain no known vulnerabilities
4. Verify checksums and file hashes
5. Ensure there is a review process for code changes/updates
6. Ensure proper access control to ensure data integrity
7. Digital signatures can serve as a trusted stamp, confirming that the data or software comes from the original sources without any meddling.
8. Secure your CI/CD workflow and ensure adequate segmentation, access control, and parameterization.
9. Audit the code before it makes it to the production
10. Get your software pentested frequently to ensure high security levels.

9. Security Logging And Monitoring Failures:

Security event logging and Monitoring is a procedure that associations perform by performing electronic audit logs for signs to detect unauthorized security-related exercises performed on a framework or application that forms, transmits, or stores secret data. Lack of such functionalities can make malicious activities harder to detect and in turn affects the incident handling process. When it comes to exploitation of cybersecurity, insufficient logging and monitoring have been the major cause of incidents. Attackers are always on a lookout for opportunities like lack of monitoring and timely detection and response to an incident.

Security Logging & Monitoring Failures is not a direct vulnerability or threat, but rather the organisation is blind to current active attacks, previous attacks, and the information needed in the forensics process to determine the impact of the attack.

Without this insight the organisation is vulnerable to future attacks through the same methods or backdoors planted in previous attacks which might be even more difficult to detect.

Log monitoring can be split up into three parts:

- **Log collection**: this includes log enrichment like parsing of logs, converting of logs, filtering of logs, etc.
- **Log management**: keeping data retention policies, keeping shards/indexes for better performance, implementing access control as logs contain sensitive information, etc.
- **Log monitoring/analysis**: visualization, alerting, reporting.

How to prevent Against Security Logging And Monitoring Failures:

1. Ensure that you log all login and failed attempts
2. Ensure that the logs contains all the relevant data and are well-formatted to be consumed by other tools or log management solutions
3. Test if your monitoring systems that can suspicious activity and ensure the alerting is done in near real time.
4. Ensure that your logs are tamper-proof.
5. How we log is also partially decided by the log management systems since the logs we create should be able to be consumed by said systems.
6. The formats of our logs should match the expected format for the management solution.
7. Log should be treated with the utmost respect towards people's privacy in mind, they should be kept safe and transported safe.
8. Logging is needed but we should also set up a 24/7 monitoring system that monitors our logs, infrastructure and API endpoints. We should get an alert from this system if a breach occurs.
9. Security Information and Event Management (SIEM) systems can be used to aggregate logs from all components of the API technology stack and the virtual hosts.

10. Server-Side Request Forgery (SSRF):

SSRF flaws occur whenever a web application is fetching a remote resource without validating the user-supplied URL. It allows an attacker to coerce the application to send a crafted request to an unexpected destination, even when protected by a firewall, VPN, or another type of network access control list (ACL).

As cloud services increase in usage and popularity as well as their complexity, the prevalence and risk of SSRF attacks increase too. As cloud services increase in usage and popularity as well as their complexity, the prevalence and risk of SSRF attacks increase too.

In this attack, attackers target a vulnerable application's backend server and coax it to execute malicious requests for performing unintended actions. Through SSRF attacks, hackers can infiltrate other systems connected to the webserver or target internal resources they cannot reach from outside the network.

Types Of SSRF Attacks:

1. Server SSRF Attacks:

In a server SSRF attack, attackers exploit a process in which a browser or other client system directly accesses a URL on the server.

The attacker will replace the original URL with another, typically using the IP 127.0.0.1 or the hostname "localhost", which point to the local file system on the server.

Under this hostname the attacker finds a file path that leads to sensitive data.

2. Back-End SSRF attacks:

It is when a server has a trusted relationship with a back-end component.

If, when the server connects to that component, it has full access rights, an attacker can forge a request and gain access to sensitive data, or perform unauthorized operations.

Back-end components often have weak security because they are considered to be protected inside the network perimeter.

How to prevent SSRF:

1. Divide your network into different sections, each with its specific role. Thus, by separating the remote access features into distinct networks, you can efficiently mitigate the damage an SSRF attack can inflict.
2. Also, set your firewall settings to a "deny by default" stance or establish network access control rules that block all web traffic, save for necessary internal exchanges.
3. Finally, always stay alert and question the legitimacy of URLs. This vigilance can provide a safeguard against insidious attacks such as DNS redirection and "time of check, time of use" situations.
4. Sanitize all user input
5. Use a positive allow list rather than a punitive block list
6. Do not send raw responses to users/clients
7. Disable unencrypted (HTTP) redirections

Other common web applications vulnerabilities are:

Cross-Site Request Forgery (CSRF):

A Cross-Site Request Forgery (CSRF) attack is when a victim is forced to perform an unintended action on a web application they are logged into. The web application will have already deemed the victim and their browser trustworthy, and so executes an action intended by the hacker when the victim is tricked into submitting a malicious request to the application. This has been used for everything from harmless pranks on users to illicit money transfers.

One way website owners can help cut down on their chance of attack is to have advanced validation techniques in place for anyone who may visit pages on their site or app, especially when it comes to social media or community sites. This will enable them to identify the user's browser and session to verify their authenticity.

While there are a variety of ways a hacker may infiltrate an application due to web application vulnerabilities, there are also a variety of ways to defend against it. There are web application security testing tools specially designed to monitor even the most public of applications. Using these scanners reduce your chances of being the victim of a hack by showing you exactly where to make the changes needed for more secure applications.

Cross-Origin Resource Sharing (CORS) Policy:

Every web-based application uses a URL as a way to connect the user's browser to its server. One common protection is called a Same Origin Policy. According to this, the server will only respond to a URL that has the same protocol, top-level domain name, and path schema. This means that you can access http://company.com/page1 and http:/company.com/page2 because they both have the following in common:

- Protocol: HTTP
- Domain: Company.com
- Path schema: /page#

Although secure, the Same Origin Policy becomes restrictive when working with web-based applications that need access to resources that connect to subdomains or third-parties.

A CORS policy gives the browser permission to access these shared resources by creating a set of allowed HTTP headers considered "trusted."

For example, an application may need to pull data from two databases on different web servers. Creating a specific "allowed" list becomes too much work as you add more servers.

Since the application is "shared" by both servers, the organization creates a CORS policy that lets browsers connect to both. However, if a CORS policy is not well defined, then the policy might allow the servers to provide access when a malicious actor requests it.

Credentials management:

User credentials consist of a user ID and password. To gain access to an application, the user must input both pieces of information into the login page. The application compares this data to that stored in its database. If both pieces match, then it grants the user access.

However, databases often store this information in plaintext or use weak encryption. Poor credentials management makes it easy for attackers to steal credentials and use them to gain access to web applications.

Directory indexing:

Web servers often list all the files stored on them in a single directory. If a user is trying to locate a specific file in a web application, they normally include the file name as part of the request. If that file is not available, the application will return a list of all indexed files, giving the user a way to choose something else.

However, web servers automatically index the files. If the application returns a list of all files stored, a malicious actor exploiting vulnerabilities in the directory index can gain access to information that can tell them more about the system.

For example, it can tell them about naming conventions or personal user accounts. Both of these data points can be used to locate sensitive information or engage in credential theft attacks.

Improper certificate validation:

SSL certificates bind a domain name, server name, or hostname to a company and location. For example, GoodSecureCo installs the SSL certificate data files on its US web servers.

Every time a browser asks for data from the US web server, the SSL certificate checks to make sure that the user's browser connects with an approved owner. The two securely connect if the answer is yes.

When software refuses to validate or incorrectly validates the certificate, it has an improper certificate validation vulnerability. Most often, attackers create a false trusted entity that tricks the server or application into thinking the certificate is valid so it accepts the data transfer as legitimate.

Often, malicious actors use improper certificate validation vulnerabilities as a way to install malware on endpoints.

Session ID leakage:

Session IDs are the unique identifiers that authenticate users and track their activities when they use a web application. Web application vulnerabilities that lead to session leakage include:

- Storing the session ID in the query string. By storing the session ID in the part of the URL that asks the application to retrieve information from the database, sharing of that URL allows the recipient to inherit that session without new authentication.

- Storing the session ID in HTTP cookies: By storing the session ID in the small data files that let a web server remember a web browser and using the unencrypted HTTP protocol, the application gives the attacker the ability to steal the session ID and impersonate the user.

Unrestricted File Upload:

Web applications often incorporate file upload capabilities. For example, if you want to input data in bulk, you might upload a CSV file to a database. An unrestricted file upload vulnerability can be a lack of authentication OR authorization when someone tries to upload a file.

This means that the application fails to verify the user, giving malicious actors the ability to upload compromised files. Additionally, the application may fail to sanitize files prior to uploading, thus giving attackers a way to leave malicious content in the files, like macros that hide malware.

Additional file upload vulnerabilities include:

- Allows all file extensions
- Fails to authorize or authenticate users
- Fails to scan content to ensure the file type is expected
- Allows webserver to fetch files
- Stores files in a publicly accessible directory

Conclusion: In conclusion, Web applications commonly have vulnerabilities that can pose risks, to users and organizations alike. Malicious actors can exploit these vulnerabilities to compromise data, disrupt services. Even gain access to sensitive information. To address these risks it is crucial for web developers, security professionals and organizations to take an comprehensive approach, towards ensuring the security of web applications.

Some key takeaways from the discussion on common vulnerabilities in web applications include:

- Awareness and Education
- Secure Coding Practices
- Regular Testing and Scanning
- Patch Management
- Web Application Firewalls (WAFs)
- Access Control and Authentication
- Monitoring and Incident Response
- Security by Design
- Compliance and Regulations