Security Model

Threat model

Threat modeling can be done by anyone and at any stage of development. Here are 5 basic steps:

- **Define security requirements** so you know what to protect from threats and what these threats could target.
- **Map out the application structure** to get an idea of all the data flows and actors involved. A data flow diagram is the usual way of doing this.
- **Identify threats and threat categories** that could pose a risk to your application based on defined security criteria. This is the main threat identification and analysis step.
- **Mitigate threats** to ensure that identified risks can't translate into real-life attacks.
- **Check that threats have been mitigated** to make doubly sure that everything specified in your threat model is now a purely theoretical risk.

Cross site scripting (XSS) - XSS is a vulnerability that allows an attacker to inject client-side scripts into a webpage in order to access important information directly, impersonate the user, or trick the user into revealing important information. The goal of XSS attacks is to send this malicious code to other users, sometimes infecting their devices with malware or stealing sensitive information. This type of	1. **Modern frameworks have made it a lot easier to escape untrusted user input and mitigate XSS attacks. AngularJS, React JS, and Ruby on Rails are some of the latest, most effective frameworks to prevent these web application vulnerabilities. These frameworks can automatically escape user input and help mitigate XSS attacks by design, although they do have limitations.** 2. **Avoid implementing a blacklist, instead favor of a whitelist, because blacklists are less effective at preventing web security vulnerabilities. An attacker who knows what they're doing can easily bypass a blacklist filter.** 3. **The ultimate solution to prevent these web application vulnerabilities is output**

website application vulnerability can give the attacker full control of the user's browser and can be extremely dangerous to any website.	**encoding. This involves converting untrusted user input into a safe form so the input is displayed to the user as data without being executed as code in the browser. This means that special characters will be translated into an equivalent form that the browser will no longer find significant.** **4. It's also important to understand that output encoding depends on the context of where data is being output. For instance, you may have HTML contexts, HTML entity contexts, HTML attribute contexts, CSS contexts, JavaScript contexts, and more. As such, you will need to apply context-sensitive encoding when render the page for the browser.** **5. Enable a Content Security Policy (CSP), which can be very effective to help mitigate Cross-Site Scripting vulnerabilities.**
SQL injection (SQi) – SQi is a method by which an attacker exploits vulnerabilities in the way a database executes search queries. Attackers use SQi to gain access to unauthorized information, modify or create new user permissions, or otherwise manipulate or destroy sensitive data.	**1. Prepared statements with parameterized queries can mitigate SQL-related web application vulnerabilities. A prepared statement helps to sanitize the input and ensures that it is considered as a string literal in SQL rather than as part of the SQL query. In other words, the database can tell the difference between SQL data and SQL code. So the code is no longer vulnerable to SQL injection attacks as the query is less vulnerable to tampering.** **2. Migrating to Object Relational Mapping Tools (ORMs) is another excellent option. However, most ORMs allow non-parameterized queries in addition to performing parameterized queries. As such, it's crucial to carefully use the frameworks keeping this in mind.** **3. Make the most of LIMIT and other SQL controls within your queries so that even if an SQL injection attack does occur, it can prevent the mass disclosure of**

	records.
Denial-of-service (DoS) and distributed denial-of-service (DDoS) attacks - Through a variety of vectors, attackers are able to overload a targeted server or its surrounding infrastructure with different types of attack traffic. When a server is no longer able to effectively process incoming requests, it begins to behave sluggishly and eventually deny service to incoming requests from legitimate users.	**DDoS mitigation** A commonly used method for disrupting a web application is the use of distributed denial-of-service or DDoS attacks. Cloudflare mitigates DDoS attacks through a variety of strategies including dropping volumetric attack traffic at our edge, and using our Anycast network to properly route legitimate requests without a loss of service.
Memory corruption - Memory corruption occurs when a location in memory is unintentionally modified, resulting in the potential for unexpected behavior in the software. Bad actors will attempt to sniff out and exploit memory corruption through exploits such as code injections or buffer overflow attacks.	
Buffer overflow - Buffer overflow is an anomaly that occurs when software writing data to a defined space in memory	

known as a buffer. Overflowing the buffer's capacity results in adjacent memory locations being overwritten with data. This behavior can be exploited	
Application attack from malicious HTTP traffic	**WAF - Protected against Application Layer attacks** A web application firewall or WAF helps protect a web application against malicious HTTP traffic. By placing a filtration barrier between the targeted server and the attacker, the WAF is able to protect against attacks like cross site forgery, cross site scripting and SQL injection.
Bad actors will attempt to hijack this DNS request process through DNS cache poisoning, on-path attacks and other methods of interfering with the DNS lookup lifecycle. If DNS is the phonebook of the Internet, then DNSSEC is unspoofable caller ID.	**DNS Security - DNSSEC protection** The domain name system or DNS is the phonebook of the Internet and represents the way in which an Internet tool such as a web browser looks up the correct server.
Authentication Failure **Authentication-related web application vulnerabilities occur when there's an improper implementation of adequate user authentication controls. This puts user**	**How to Prevent It** 1. **One of the essential steps to avoid these web application vulnerabilities is to allow enough time for developers to test the code before it gets deployed to production. External security audits can also help ensure that you apply the best**

accounts at risk of being breached. Attackers may exploit these web security vulnerabilities to gain control over any user account or even over the entire system. **One of these vulnerabilities is Credential Stuffing, where an attacker will test a list of valid passwords and usernames gleaned from another breach or attack until they manage to find a valid combination and gain access.** **Another common vulnerability is a Brute Force attack, in which the attacker tries every possible character combination until they find a valid one.** **Session hijacking is another common attack that can occur as a result of authentication failure. This is when there is a failure to properly invalidate session IDs, allowing attackers to exploit an authenticated session of a legitimate user.**	**practices of website security.** **2. Avoid deploying with default credentials, especially for admins.** **3. Wherever possible, make sure you implement multi-factor authentication to make your system less vulnerable to the attacks mentioned above.** **4. Put a limitation or delay on failed login attempts. Make sure you log all failures and notify administrators when there's an attack attempt.** **5. Avoid unnecessarily restricting input size. If you allow more characters, there are fewer chances for attackers to guess the right password.** **6. Have some form of lockout in place to prevent brute force attacks and minimize these web application vulnerabilities.** **7. Use adaptive hashing algorithms like bcrypt, pbkdf2, argon2, etc. to salt passwords and hash them before storing them in the database** **8. Implement weak-password checks for better password security. This would include testing new or changed passwords and comparing them against a list of compromised or weak passwords. Use of a service to check for compromised passwords (such as Have I Been Pwned) helps to automate this functionality and keeps the list of compromised passwords up to date as new attacks occur.**
Security Misconfiguration **Security misconfigurations provide attackers with an easy way into your website, making it one of the most critical web application vulnerabilities that you need to prevent.**	**1. Make sure you use encrypted (HTTPS) connections to transfer data and information between the users and the application.** **2. Have a repeatable hardening process that you can quickly and easily deploy on another environment. This will save time in setting up a new and secure environment as you'll be able to automate the process.** **3. Perform all remote admin tasks through**

Unused pages, unpatched flaws, unprotected files and directories, and default configurations, are some of the security misconfigurations that attackers can leverage to gain unauthorized access. **Every level of your application stack can be vulnerable to security misconfigurations. This includes your web server, platform, database, network services, storage, frameworks, application server, and more.** **If attackers manage to exploit these web application vulnerabilities, they can access sensitive information and take control of user and admin accounts.**	**secured channels to minimize these web application vulnerabilities. Even if you do have to use protocols that don't support strong encryption, make sure you activate them over a secondary encryption channel such as IPSEC, or TLS.** **4. Regularly conduct file integrity checking to ensure that there haven't been any unauthorized changes to critical files. Use file integrity checking tools that can accept routine and expected changes while alerting you of any unexpected or unusual changes.** **5. Have an automated process in place to regularly verify the effectiveness of your settings and configurations in every environment. You can use an automated configuration monitoring tool that can alert you of any unauthorized changes. This will help you identify these web application vulnerabilities before they cause any damage.** **6. Keep your platform minimal and avoid adding unnecessary features, samples, documentation, and components. If you have unused features and frameworks in the platform, it's best to remove them to prevent web application vulnerabilities.**
XML External Entities **An XML external entity attack, also known as an XXE, or an XML injection attack, is another class of vulnerabilities you should watch out for. These types of attacks occur when attackers exploit a weakly-configured XML parser. Through such attacks, attackers can inject additional data, access confidential data, and execute applications**	**How to Prevent It** **1. Completely disabling Document Type Definitions (DTD's), also known as External Entities, is the safest way to prevent XXE attacks. This secures the parser against DoS attacks. However, it may not always be possible to completely disable DTDs. In this case, you need to disable external document type declarations and external entities in a specific way for each parser.** **2. Whenever possible, you should try using less complex data formats like JSON. It's also good to avoid serialization of**

and create remote tunnels (shells). **XML external entity attacks can also result in remote code execution, Server Side Request Forgery (SSRF), and more. By default, most XML parsers are prone to these attacks. This leaves it up to the developers to ensure that their web application is free from these web application vulnerabilities.**	**sensitive data to avoid these website application vulnerabilities.** **3. Implement a positive, server-side method to validate, sanitize, and filter input. This will help prevent the occurrence of hostile data within your XML documents, nodes, and/or headers and help you avoid XXE-related web application vulnerabilities.** **4. Patch or upgrade all the XML processors and libraries that the application or its underlying OS is using.** **5. While the best solution is to use manual code review for critical functionality in large and complex applications, you should also use SAST tools to detect XXE in source code.** **6. Use XSD validation or an equivalent alternative to validate the incoming XML structure.**
Broken Access Control (Authorization Failure) **Access control helps you control what sections of a website and what application data different visitors can access.** **For instance, if your website is a platform for different sellers to list their products, they will need some kind of access to add new products and manage their sales. However, not every visitor will need that level of access since most of them are visiting your site to buy products.** **As such, having a broken access control opens up your site to web application**	**1. It's crucial to maintain a security-first policy when developing and configuring software.** **2. Except for public resources, deny access by default.** **3. Make sure you apply protection horizontally (across all data) and vertically (across all levels of access privileges). Vertical protection involves employing the least privilege concept wherein access is granted only in accordance to their** **4. Centralize all authorization decisions to minimize the occurrence of access-related web application vulnerabilities.** **5. Instead of letting users freely create,**

vulnerabilities, which attackers can exploit to access sensitive information or unauthorized functionality. They might even use these attacks to make modifications to access rights and user data.	**read, modify, or delete any record, use model access controls to enforce record ownership. Remove their ability to read or modify data from other users.** **6. Implement a one-time access control mechanism that you can reuse through the 1.** **7. Set limitations to API and controller access to reduce the risk of attacks by automated tools.** **8. Invalidate JWT tokens and user sessions on the server after logout.** **9. Disable webserver directory listing and also prevent web roots from storing backup files and file metadata.**