Task – 4

Name : C DEVAKI NANDAN REDDY

Understanding any Top 10 web applications Vulnerabilities (other than Top 10 OWASP) write a paragraph about that and add an image to the respective vulnerability

10 Common Web Security Vulnerabilities

Authentication and Authorization: A Cyber Security Primer

Programmers and IT professionals often express confusion regarding the distinction between authorization and authentication. The use of the abbreviation *auth* for both terms increases the haziness that surrounds them. Let's define and clarify the distinction:

- **Authentication:** Verifying that a user is (or at least appears to be) the person they say they are.
- **Authorization:** Granting a user access to a specific resource, or permission to perform a particular action.

Stated another way, *authentication* is knowing who an entity is, while *authorization* is what a given entity can do. With this in mind, let's explore 10 common internet vulnerability issues.

Injection Flaws

Injection flaws result from a classic failure to filter untrusted input. Injection flaws can happen when we pass unfiltered data to the SQL server (SQL injection), to the browser (via Cross Site Scripting), to the LDAP server (LDAP injection), or anywhere else. The problem here is that the attacker can inject commands to hijack clients' browsers, resulting in loss of data.
Anything that your application receives from an untrusted source must be filtered, preferably according to a whitelist. Using a blacklist to this end is not recommended, as it is difficult to configure properly. A blacklist is also considered easy for a hacker to bypass. Antivirus software products typically provide stellar examples of failing blacklists. Pattern matching does not work.

Prevention: Protecting against injection is "simply" a matter of filtering our input and considering which senders can be trusted. Filtering is quite an undertaking because we need to process all input unless it can unquestionably be trusted.
If we filter 999 inputs in a system with 1,000 inputs, we still have one field that can be the Achilles' heel that brings down our system.

Using Second Order SQL Injection to inject one SQL query result into another is also considered dangerous. It could seem like a good idea because the database is trusted. But if the perimeter is not, our input could originate indirectly from a malicious source.
Since filtering is pretty hard to get right, it is advisable to rely on our framework's filtering functions. They are proven to work and thoroughly scrutinized. If you do not already use a framework, consider the server security benefits of moving to one. .

Broken Authentication

Problems that might occur during broken authentication don't necessarily stem from the same root cause. Rolling your own authentication code is not recommended, as it is hard to get right. There are myriad possible pitfalls, and here are a few:

1. The URL might contain the session ID and leak it in the referer header.
2. Passwords might not be encrypted in storage and/or transit.
3. Session IDs might be predictable, making it a little too easy to gain unauthorized access.
4. Session fixation might be possible.
5. Session hijacking could occur if timeouts are not implemented correctly, or if using HTTP (no SSL security), etc.

Prevention: The most straightforward way to avoid the web security vulnerabilities related to broken authentication is to implement a framework. If you roll your own code, be extremely paranoid and educate yourself on the potential issues that could arise.

Cross-Site Scripting (XSS)

An attacker sends on input JavaScript tags to your web application. When this input is returned to the user unsanitized, the user's browser would execute it. This is a fairly widespread input sanitization failure, essentially a subcategory of injection flaws). CSS can be as simple as crafting a link and persuading a

user to click it, or it can be something much more sinister. For example, on page load, the script would run and be used to post your cookies to the attacker.
Prevention: Simply put, don’t return HTML tags to the client. This would also protect you from HTML injection, which is when an attacker injects plain HTML content (such as images or loud but invisible flash players). To implement this solution, convert all HTML entities to return something else. For example, convert `<script>` to return `&lt;script&gt;`. Alternatively, you can use regular expressions to strip away HTML tags using regular expressions on `<` and `>`. But this is dangerous because some browsers may not interpret severely broken HTML. Better to convert all characters to their escaped counterparts.

Insecure Direct Object References

This is a classic case of trusting user input and paying the price by inheriting a resultant security vulnerability. A direct object reference means that an internal object (e.g., a file or a database key) is exposed to the user, leaving us vulnerable to attack. The attacker can provide this reference, and if authorization is either not enforced or broken, the attacker gets in.
For example, the code has a `download.php` module that reads and lets the user download files, using a CGI parameter to specify the file name (e.g., `download.php?file=something.txt`). If the developer omitted authorization from the code, the attacker can now use it to download system files accessible to the user running PHP (e.g., the application code or random server data like backups).
Another example of insecure direct object reference vulnerability is a password reset function that relies on user input to determine their identity. After

clicking the valid URL, an attacker could modify the `username` field in the URL to say something like "admin."
Incidentally, I have seen both of these examples often "in the wild."

Prevention: Perform user authorization properly and consistently, and whitelist the choices. More often than not, the vulnerability can be avoided altogether by storing data internally and not relying on data being passed from the client via CGI parameters. Session variables in most frameworks are well suited to this purpose.

Security Misconfiguration

In my experience, it is common to encounter misconfigured web servers and applications. Some examples:

1. Running an application with debug enabled in production
2. Having directory listing (which leaks valuable information) enabled on the server
3. Running outdated software (think WordPress plugins, old PhpMyAdmin)
4. Running unnecessary services
5. Not changing default keys and passwords (which happens more frequently than you'd believe)
6. Revealing error handling information (e.g., stack traces) to potential attackers

Prevention: Have a good (preferably automated) “build and deploy” process, which can run tests on deploy. The poor man’s security misconfiguration solution is post-commit hooks, to prevent code from going out with default passwords and/or development stuff built in.

Sensitive data exposure

This web security vulnerability is about crypto and resource protection. *Sensitive data should be encrypted at all times, including in transit and at rest. No exceptions.* Credit card information and user passwords should *never* travel or be stored unencrypted, and passwords should always be hashed. Obviously, the crypto/hashing algorithm must not be a weak one. When in doubt, web security standards recommend AES (256 bits and up) and RSA (2048 bits and up).
It cannot be overemphasized that session IDs and sensitive data should not travel in URLs. Cookies with sensitive data should have the “secure” flag on.

Prevention:

- *In transit:* Use HTTPS with a proper certificate and PFS (Perfect Forward Secrecy). Do not accept anything over non-HTTPS connections. Have the “secure” flag on cookies.
- *In storage:* Reduce your exposure to this vulnerability. If you don’t need sensitive data, virtually shred it. The data you don’t have can’t be stolen. Do not store credit card information, and you will not need to have to deal with being PCI compliant. Sign up with a payment processor like Stripe or Braintree. Store and encrypt sensitive data, and ensure all passwords are hashed using bcrypt.

> If you don’t use bcrypt, educate yourself on salting and rainbow tables.

And at the risk of stating the obvious, *do not store the encryption keys near their protected data*. That’s like storing your bike with a lock that has the key in it. Protect your backups with encryption and keep your keys private. And of course, don’t lose the keys!

Missing Function Level Access Control

This is a failure that happens if proper authorization is not performed when a function is called on the server. Developers tend to assume that since the server side generates the UI, the client would not be able to access functionality that is not supplied by the server. It is not as simple as that, as an attacker can always forge a request to the “hidden” functionality. An attacker will not be deterred by the fact that the desired functionality is not easily accessible. Imagine there’s an `/admin` panel, and the button is only present in the UI if the user is actually an admin. Nothing keeps an attacker from discovering and misusing this functionality if authorization is missing.
Prevention: On the server side, authorization must *always* be performed.

Cross-Site Request Forgery (CSRF)

In a CSRF—also referred to as a confused deputy attack—a malicious third party fools the browser into misusing its authority to do something for the attacker.
In the case of CSRF, a third-party site uses your browser, cookies, and session to issue a request to a target site (e.g., your bank). If on one browser tab you are logged in to your bank, and if your bank is vulnerable to this type of attack,

then another tab can be controlled to make your browser misuse its credentials on the attacker's behalf, which results in the confused deputy problem. The deputy is the browser that misuses its authority (session cookies) to perform the attacker's instructions.

Consider this example: Attacker Alice wants to lighten target Todd's wallet by transferring some of his money into her account.

To send money, Todd accesses the following URL: `https://example.com/app/transferFunds?amount=1500&destinationAccount=4673243243` at his bank which, incidentally, is vulnerable to CSRF attacks. After Todd performs his transaction, a success page displays and the transfer is complete.
Alice is aware that Todd frequently visits a site she controls at `https://blog.aliceisawesome.com`, so Alice places the following snippet on her site: `<img src=https://example.com/app/transferFunds?amount=1500&destinationAccount=4673243243 width=0 height=0 />`
When Todd next visits Alice's website, his browser wrongly thinks the snippet links to an image. The browser automatically issues an `HTTP GET` request to fetch the picture. But instead of getting an image to display in the browser, the request instructs Todd's bank to transfer $1,500 to Alice.
Incidentally, in addition to demonstrating the CSRF vulnerability, this example also demonstrates altering the server state with an idempotent (safe)`HTTP GET` request. This in itself a serious vulnerability. `HTTP GET` requests *must* be idempotent, meaning that they cannot alter the resource that is accessed. Never use idempotent methods to change the server state.

Fun fact: CSRF is also the method people used for cookie-stuffing in the past until affiliates got wiser.

Prevention: Store a secret token in a hidden form field, inaccessible to a third-party site. This, of course, requires you to verify the hidden field. Some sites may ask for a password before allowing you to modify sensitive settings (like a password reminder email). I suspect this could be to prevent the misuse of your abandoned sessions on public computers.

Using Components With Known Vulnerabilities

The title says it all. I'd classify this one as more of a maintenance/deployment issue. Before incorporating new code, do some research, and possibly some auditing. Using code from a random person on GitHub, for example, may be convenient, but it is not without risk of serious web security vulnerability.

I have seen many instances where sites got owned (i.e., where an outsider gains administrative access to a system), because third-party software (e.g., WordPress plugins) remained unpatched for years in production. If you think they will not find your hidden `phpmyadmin` installation, let me introduce you to DirBuster.

The lesson here is that software development does not end when the application is deployed. There has to be documentation, tests, and plans on how to maintain and keep the application updated, especially if it contains third-party or open source components.

Prevention:

- Do not be a copy-paste coder. Carefully inspect the piece of code you are about to put into your software, as it might be broken or, in some cases, intentionally malicious. Web security attacks are sometimes unwittingly invited in this way.

- Stay up-to-date with the latest versions of everything that you trust, and have a plan to update regularly. To stay on top of new security vulnerabilities, subscribe to your products' newsletters.

Unvalidated Redirects and Forwards

This is yet another input filtering issue. Suppose that the target site has a `redirect.php` module that takes a URL as a `GET` parameter. Manipulating the parameter can create a URL on `targetsite.com` that redirects the browser to `malwareinstall.com`. A user would see the link as `targetsite.com/blahblahblah`, which looks innocuous enough to trust and click. But clicking this link could transfer the user to a malware drop (or any other malicious) page. Alternatively, the attacker might redirect the browser to `targetsite.com/deleteprofile?confirm=1`.

It is worth mentioning that stuffing unsanitized user-defined input into an `HTTP` header might lead to header injection, which is pretty bad.

Prevention: Options include:

- Don't do redirects; these are seldom necessary.
- When a redirect is necessary, have a static list of valid redirect locations.
- Whitelist the user-defined parameter. Note this can be tricky.