(lec 21) **WEB SECURITY : VULNERABILITIES & ATTACKS**

1> **COMMAND INJECTION** (first case of vulnerability)

A class of vulnerability in web apps.

>**Background**

Client browser sends URI to web server, which runs php prog (display.php)

↓

It parses to get the request to get parameter written in URI

↓ It is processed in server by display.php

↓

result is returned back to client browser.

→ Command injection is an attack in which the goal is execution of arbitary commands on the host operating sys via a vulnerable app.

→ Command injection ~~via~~ are possible when an app passes unsafe user supplied data (forms, cookies, header to a system shell.

- Client can send server different URI requests
  - →Any URI can ~~a~~ cause attack on server if attacker has injected a command in an original command
  - →Two commands will run (aik original aur doosra attacker ke injected)

(^ Eg. in slide)

Command Injection is a more general prob of INJECTION

- Caused when data & code share the same channel

Ways to Defend against attacks

① **INPUT VALIDATION** (whitelists untrusted inputs to a safe list)

when & check input against certain checks to ensure that input won't cause injection attack.

Two types of checks/ Two forms:

① Blacklisting — Block known attack values to make sure they aren't present in input val.

② Whitelisting — Only allow ~~un~~known good values.

①• Blacklists are easily bypassed b/c:
→ set of 'attack' inputs is potentially infinite.
→ set can change after you deploy your code.
• Only rely on blacklists as a part of defense in depth strategy.

} Blacklist not preferred!

DEB3/2.

- Blacklist Bypass

Blacklist	Bypass
→ Disallow semi colon	use pipes
→ Disallow pipes, semicolon, backticks	use $ operator
→ Disallow rm	use unlink
⋮	

} There are so many possibilities that attacker can launch an attack ∴ it's not preferred.

② **whitelisting**

→ check filename against whitelist before concatinating the filename (see the particular eg. in slide).

Developing a correct whitelist is enough & secure against attacks; as attacking can be challenging.

② **INPUT ESCAPING** (escape untrusted input so that it won't be treated as a command)

- we want whole parameter to be interpreted as one single string → we can accomplish this by escapeshellarg()

escapeshellarg() → adds single quotes around a string & quotes/escapes any existing single quotes allowing you to pass a string directly to a shell func & having it be treated as a single safe argument.

③ **USE LESS POWERFUL API** (we should follow this for strong security) – <u>preferred</u>

→ The sys command is too powerful eg. CAT.
- Executes the string argument in a new shell.
- If only need to read a file and output, <u>use simpler API</u>.

→ Similarly, proc_open ~~is~~ API is for executing commands – (also powerful)
- can only execute 1 command at a time.

} we should try to use simple & less powerful APIs.

→ use an API that only does what you want

2> **SQL INJECTION** (second case of vulnerability)

SQL → query language for db. ; ^case of injection in db queries – extremely common & exploiting alot.

Running Eg → A user tries to login a pg using name & pwd. It is checked in php login. through a query that if result exists, it logs in the user & redirect them to their control panel.

→ user tries to login through client Browser & sends a URL request (username, pwd) to webserver. Web server parses the request & calls login.php with 2 parameters (username, pwd). which connects to db & sent db the query which finds parameter. & process a result & resend the result to user control panel.

(see ex. in slide)

→ SQL injections occur b/c of the insecure way of login.php trying to create SQL query (trying to construct a string which contains SQL command & concatenate it with parameter from SQL query.
^chance of mixing channels.

PER 3/2.

→ Attack can inject a malicious command in SQL query.

→ One of the most exploited vulnerabilities on web. and has caused huge data Theft.

→ like command injection, caused when attacker controlled data is interpreted as a (SQL) command.

- Defenses

① INPUT VALIDATION FOR SQL

→ Put a check (on parameters) before performing a SQL query over it.

But in few cases (eg. in slides) — input validation needs to be done v. carefully b/c just validating one parameter (in username) can be insufficient & we need to validate other (in pwd) parameters too.

② INPUT ESCAPING FOR SQL

- lib 'pg_escape_string()' can be used to escape string
- pg_escape_string() → escapes a string for querying the PostgreSQL db. It returns an escaped literal in PostgreSQL format.

③ USE LESS POWERFUL API (preferred approach) —(use to prevent SQL injection)

→ Effective way in SQL injection is to use Prepared Statements.

→ Prepared Statement enables to create a template for SQL query, in which data values are substituted.
↓
This way db ensures that untrusted val. isn't interpreted as command.

→ less powerful:
  - Only allows query set in templates.

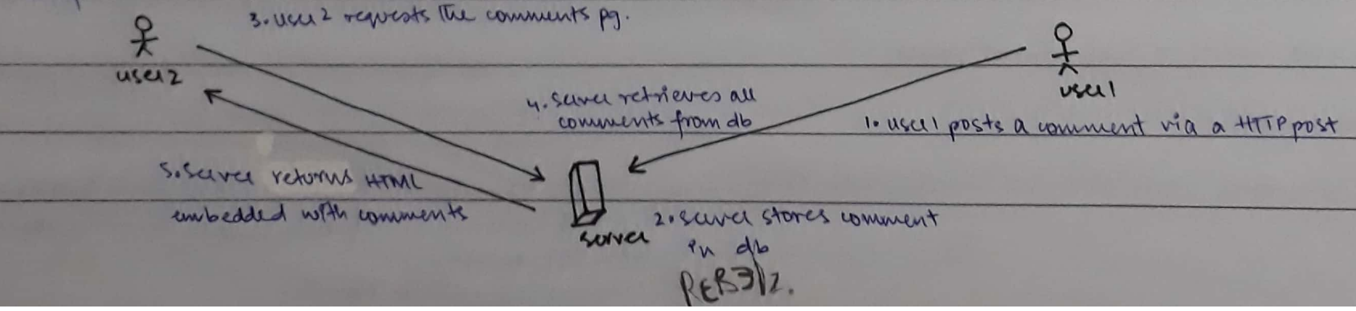
- When developer makes the Query complex, its hard for db to know the real intention of developer.

3> CROSS SITE SCRIPTING (XSS) (third case of vulnerability) → aka HTML/Script Injection

Eg. Application

→ consider a blog supporting anonymous comments. Anyone can post comments & user can see the comments.

Workflow


3. user 2 requests the comments pg.
user 2
user 1
4. server retrieves all comments from db
1. user 1 posts a comment via a HTTP post
5. server returns HTML embedded with comments
server
2. server stores comment in db

REB3/2.

(from eg. in slides) — It's clear that php code generates the webpg to be returned which displays the comments

- Attacker can perform Script Injection by putting a malicious piece of code in HTML code which will cause an untrusted code execution.

Script Injection — A security vulnerability, a serious threat that enables attacker to inject malicious code in the user interface element of your Web form of data-driven websites.

→ How is workflow exploited? — In previous diag. of workflow, user 1 ki jagah attacker & user 2 ki jagah victim.
And attacker post malicious comments (aur woh comments pooray process mein use hotta

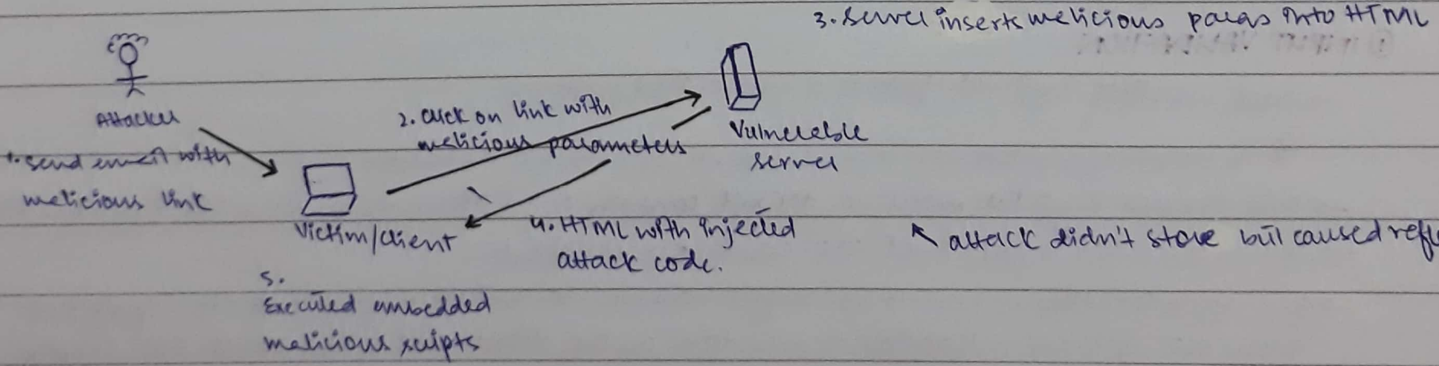
Three types of XSS

- **Type 2 : Persistent or Stored**
  → The attack vector is stored at the server & attack is triggered when user/victim tries to view comment.
  Eg. the comment based (discussed above)

- **Type 1 : Reflected** → Attack value is reflected back by the server.

- **Type 0 : DOM Based** → The vulnerability is in the client side code.

Example App : blog also has a search ~~#~~ interface search.php
→ search.php accepts a query and shows results.
(<script>doEvil() </scripts) — only injects code in attackers pg. Attack needs to make sure click on this link, for attackers to be effective.


3. Server inserts malicious paras into HTML
Attacker
1. Send email with malicious link
2. Click on link with malicious parameters
Vulnerable server
Victim/client
4. HTML with injected attack code.
5. Executed embedded malicious scripts

↖ attack didn't store but caused reflexed s

^Type 1 & Type 2 (both) have vulnerabilities on server side. (traditional XSS) & fix involves improving sanitization at the server side.

PER3/2.

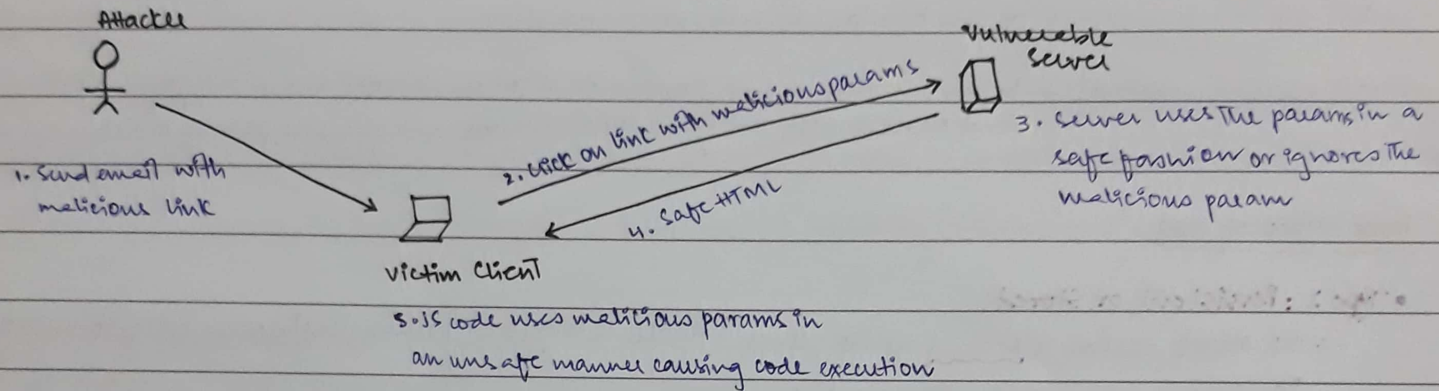
- Type 0 : DOM Based XSS

→ web 2.0 app include significant processing ~~at~~ logic at client side, writtens in JS.

→ Similar to server, code can also be vulnerable.

→ In type 0, vulnerability occurs at client side code.

consider eg. that uses client side code to display a welcome to user. (code in slide)



Attacker
Vulnerable server
1. Send email with malicious link
2. click on link with maliciousparams
3. server uses the params in a safe fashion or ignores the malicious param
4. safe HTML
Victim Client
5. JS code uses malicious params in an unsafe manner causing code execution

^ the attack payload (URI) is still sent to server, where it might be logged. } what this means?!!

→ In some web apps, URI fragment is used to pass arguments (eg. fb)

- Contexts in HTML

→ XSS is more complex than command & SQL injections.

→ Main reason → large no. of contexts present in HTML.

① HTML Attribute context → URI context ie. href = "http://xyz.com"

→ Event handler context ie onclick = "call()"

② HTML context

- XSS Injection Defenses

① INPUT VALIDATION

→ Check whether input val. follows a whitelisted pattern

eg. if accepting a phone no. by user, use JS code to validate it.

→ This ensures that the phone no. doesn't contain a XSS attack vector or a SQL injection attack

→ Only works for inputs that are easily ~~&~~ restricted.

→ Even for validation we need to be careful how we do that b/c attacker still can exploit and lauch XSS if input validation is only done on the client side & the server side simply accepts the inputs.

↓

server side cannot assume that JS validation has been done correctly on client side. ~~b/c for malicious attacker~~, he can be sending any arbitary request to server without input being validated or attack may even compromise the client and skip the JS check & hence when server recieves req. from client side, server will need to do seperate input valid. without relying on input valid. of client side.

PES312

→ So input validation on client side are only done for convinience whereas for security purpose it is a must to be done at server side.

^ This attack is known as Parameter Tampering.

∴ Input validation can be sufficient in few cases, but not all.

② INPUT ESCAPING or SANITIZATION

→ Sanitize untrusted data before outputting it to HTML

Context Sensitive Sanitization

→ sanitization needs to be done depending on context.

③ USE A LESS POWERFUL API (Preferred!)

→ current HTML API is too powerful as it allows arbitary scripts to execute at any point in HTML.

→ Content Security Policy is one policy that allows you to disable all inline scripting & restrict external script loads

→ Disabling inline scripts & restricting script loads to 'self' (own domain) makes XSS a lot harder.

Moreover,

→ To protect against DOM based XSS, use a less powerful JS API.

ie. If you want to only insert untrusted text, consider using INNERText API in JS.

↓

It ensures that the argument is only used as a Text.