[CROSS-SITE SCRIPTING & SQL INJECTION]

Cross-Site Scripting (XSS)

occurs when app ✕ sanitize user-supplied input
↳ allows attcr to execute JS in context of user's browser
↳ ex. weaponry:
① fetch(): starts process of fetching resource frm server
② document.cookie: allows reading & writing cookies assoc. w/ document (getter & setter for cookies' actual vals.)

DOM XSS

possible if web app writes data to Document Object Model (DOM) w/out proper sanitization

XSS Types

① HTML context

* XSS injected w/in HTML tags
* introduce new HTML tags designed to trigger execution of JS

```
<p>No results found for <USER SUPPLIED INPUT></p>
<p>No results found for Cats</p>
<p>No results found for <script>alert(1)</script></p>
```

② Attribute context

* XSS injected w/in HTML attribute
* terminate attr. value, close tag, or introduce new one

```
<input name="search" value="<USER SUPPLIED INPUT>" />
<input name="search" value="cats" />
<input name="search" value="" onfocus="alert(1)" autofocus="" />
```

③ Javascript context

* XSS injected directly into JS
* inject arbitrary JS possible so long can harmonize syntax to avoid errors

```
<script>
document.body.style="background:#<USER-SUPPLIED-INPUT>";
</script>
```

```
<script>
document.body.style="background:#ccc;";
</script>
```

```
<script>
document.body.style="background:#ccc;";alert(1); var i="";
</script>
```

XSS Context

Defenses Against XSS

① Input validation & Encoding (apply in following order)
* any user called data should be HTML output encoded as backup for when input validation fails / special chars needed
* validate everything!
↳ validate input on server side w/ context of their finality
↳ use whitelist of chars

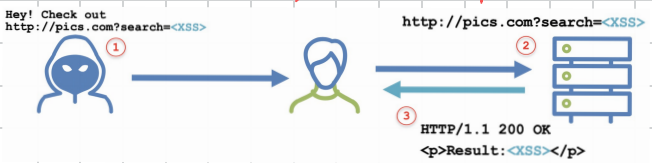
② Content Security Policy (CSP) headers
* allows browser to be aware & protect users frm dynamic calls that will load content into page currently being visited
* CSP directives:
↳ default-src: fallback for other CSP directives
↳ script-src: specifies valid srcs for JS
↳ object-src: specifies valid srcs for <object> & <embed> elements
↳ style-src: specifies valid srcs for stylesheets
↳ img-src: specifies valid srcs for imgs & favicons
↳ media-src: specifies valid srcs for loading media using <audio> & <video> elements
↳ font-src: specifies valid srcs for fonts loaded using @font-face
↳ connect-src: restricts URLs which can be loaded using script interfaces
* CSP directive vals: 'none', 'self', hostnames, * (wildcard)

ex., Content-Security-Policy: default-src 'self'; img-src *; media-src example.org example.net; script-src userscripts.example.com

↳ images may load frm anywhere ('*' wildcard)
↳ media only allowed frm example.org & example.net (excluding subdomains)
↳ executable scripts only frm userscripts.example.com

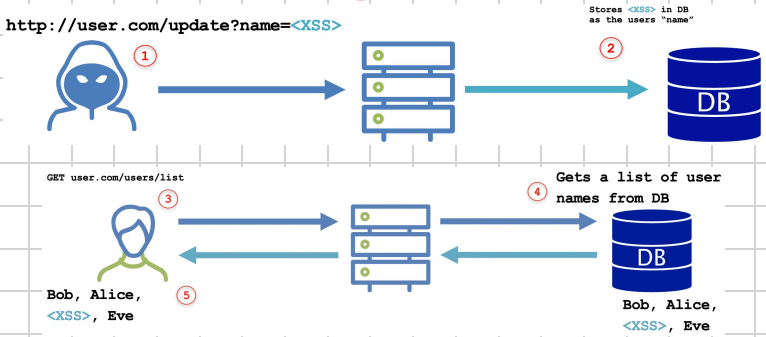
Reflected XSS

malicious payload sent to server is returned (reflected) w/in its response



Stored XSS

malicious payload sent to server is persistently stored w/in app & returned to any user subsequently accessing affected content



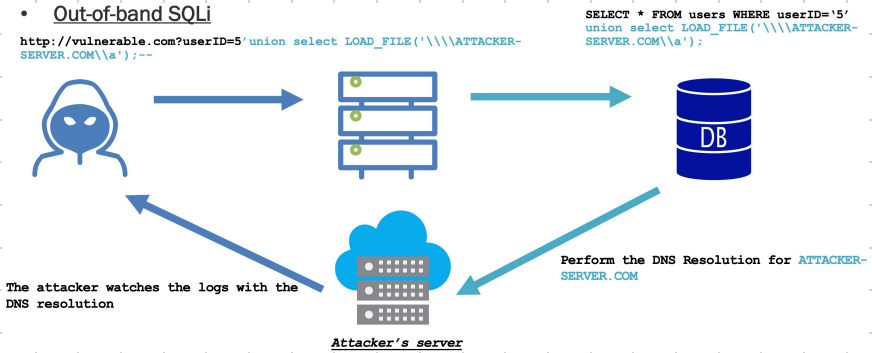
SQL Injection (SQLi)

code injection technique in which malicious SQL stmts are inserted into SQL query for execution

* can occur w/in diff. parts of query
  - ↳ UPDATE stmts (w/in updated vals. or WHERE clause)
  - ↳ INSERT stmts (w/in inserted vals.)
  - ↳ SELECT stmts (w/in table or column name or ORDER BY clause)
* SQLi performed thru stacked queries possible
  - ↳ stacked queries: perform multiple queries one after the other, separated w/ ";"

Out-of-band SQLi

* rely on — whether query returns data
  - whether db error occurs
  - time taken to execute query
* requires triggering out-of-band network interaction to ctrled server



In-band SQLi (Classic SQLi)

occurs when attacker able to use same communication channel to both launch atk & gather results

↓

* union-based SQLi
  - ↳ steps: ① get # columns ..?id=1' order by 1 -- → .. order by 2 -- → etc.
    ② perform union query ..?id=1' union select database(), version() --
  - NOTE: UNION query must have same # columns as original query & data type must be compatible w/ string data

→ * error-based SQLi
  - relies on error msgs thrown by db server to obtain info
  - ex., localhost/sqli/?id=1'&submit=Submit#
    localhost/sqli/?id=1'or1=1--
    SELECT * FROM users WHERE username = 'admin'--
    AND password = ''

Inferential SQLi (Blind SQLi)

HTTP responses for server ✗ contain results of relevant SQL query or details of any db errors

→ * boolean-based: app response (text, length, layout, status) different depending on whether query returns any data
→ * time-based: triggers time delays conditionally, depending on an injected condition
  - ↳ allows to infer truth of injected condition based on time taken before HTTP response is received

SQLi Remediation Technique

Prepared stmts

* pre-compile SQL template stmts first, then bind them w/ param values for execution

SQLi Methodology

* identify & test inputs (using quotation marks or semicolon)
* encoding (URL-encode in Burpsuite)
* db fingerprint (identify db ver.)
* exploit — list tables for db
  - dump data
  - perform RCE if possible
* evasion (bypass Web App Firewalls) — use SQL comments, etc.

SQLi in diff. contexts:

* w/ payloads like JSON or XML
* 2nd order SQLi
  - ↳ malicious input processed elsewhere by app

* SQLi for remote code execution (RCE)
  - ↳ create web shells on server (SELECT... INTO OUTFILE) ↓
  - ↳ leverage xp_cmdshell on MSSQL servers

```
SELECT "<?php system($_GET['Param']); ?>" into outfile
"/var/www/html/malicious.php"
```