Browser Security

Part 2

Recap

- XSS – 3 different types
  - Common aim/theme
    - Steal user credentials (session ID, cookies, etc.)

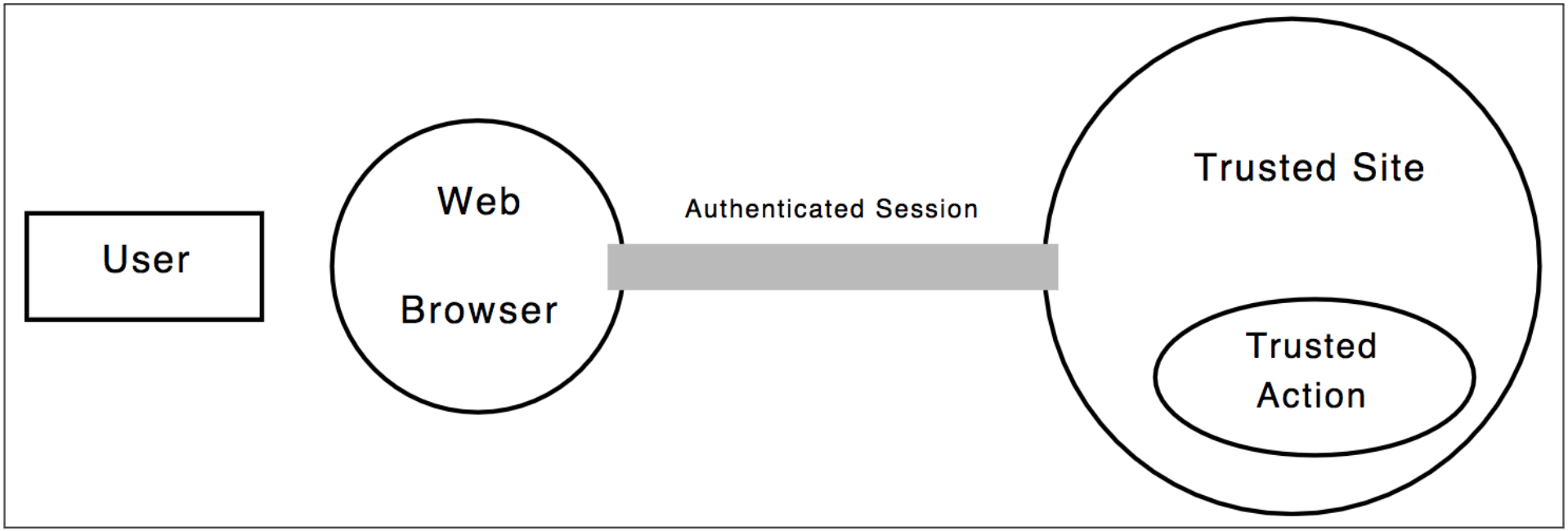
CSRF/XSRF

Cross-Site Request/Reference Forgery

CSRF

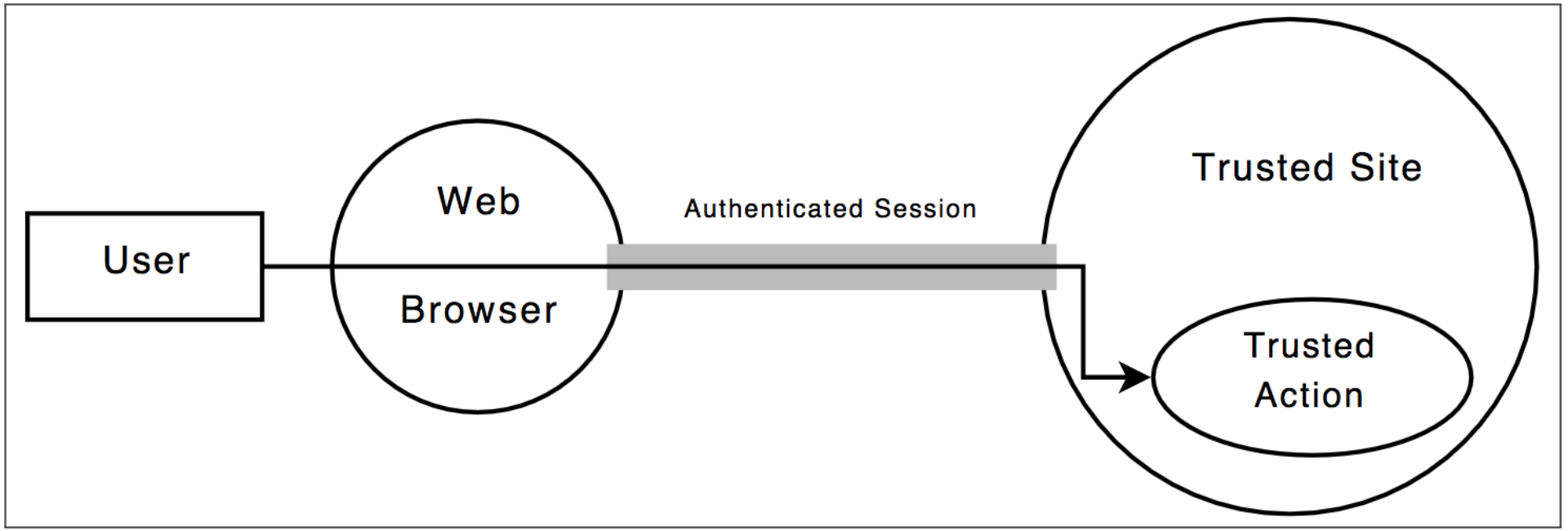
- Cross-Site Request Forgery is an attack that forces an end user to execute unwanted actions on a web application in which they are currently authenticated.
  - It inherits the identity & privileges of the victim to perform an undesired function the the victim's behalf
- CSRF attacks specifically target state-changing requests, *not theft of data*, since the attacker has no way to see the response to the forged request.

Setup



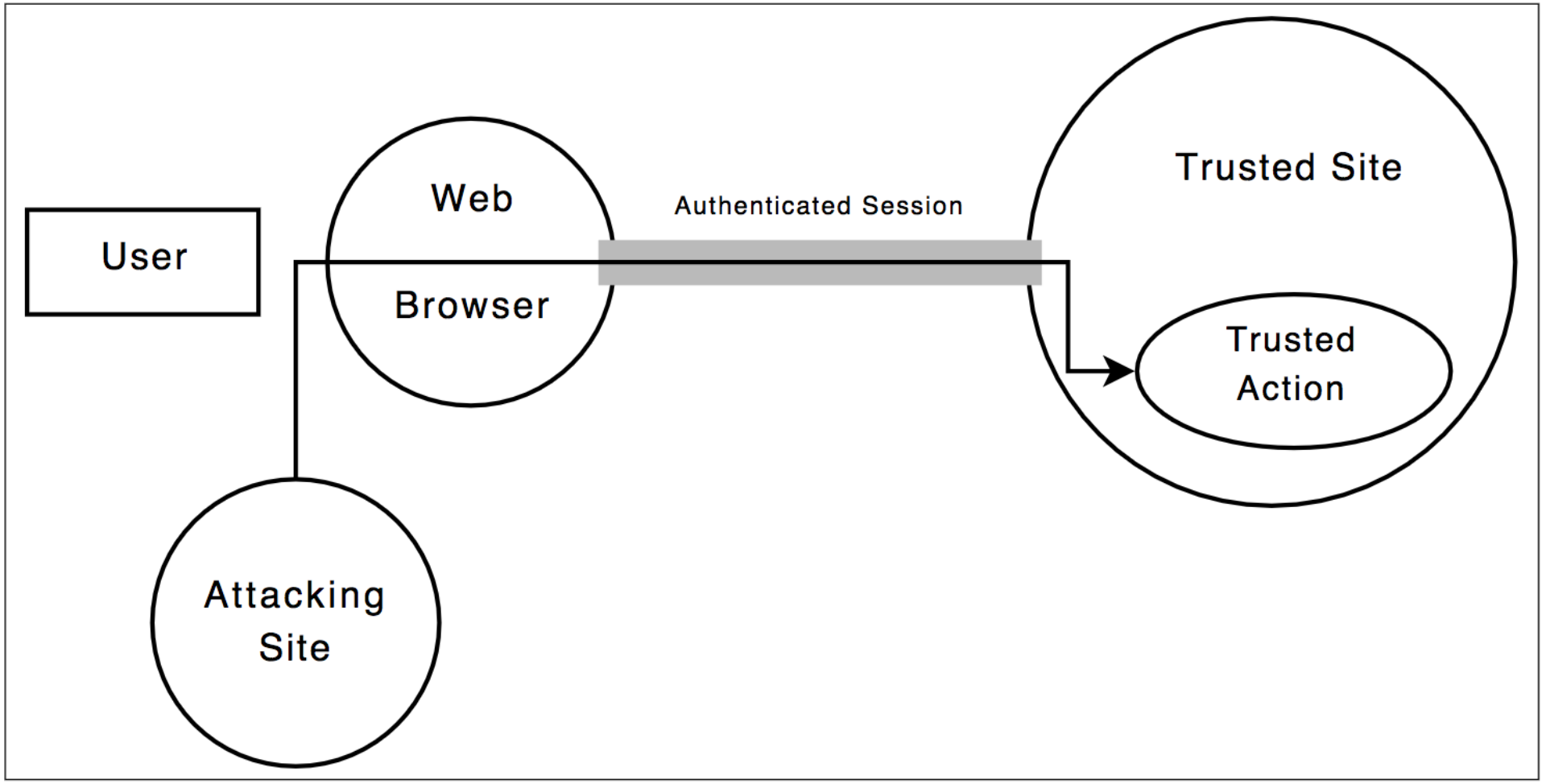
Authenticated session by an authorized user

Authentication/Authorization flow



A valid request by authorized user

CSRF attack


User
Web
Browser
Authenticated Session
Trusted Site
Trusted
Action
Attacking
Site

An Example

- Let's consider a hypothetical example of a site vulnerable to a CSRF attack.
- This site is a web-based email site that allows users to send and receive email.
- The site uses implicit authentication to authenticate its users.
- One page, http://example.com/compose.htm, contains an HTML form allowing a user to enter a recipient's email address, subject, and message as well as a button that says, "Send Email."

```
<form
action="http://example.com/send_email.htm"
method="GET">
Recipient's Email address:  <input
type="text" name="to">
Subject:  <input type="text" name="subject">
Message:  <textarea name="msg"></textarea>
<input type="submit" value="Send Email">
</form>
```

CSRF Example contd.

- When a user of example.com clicks “Send Email”, the data he entered will be sent to http://example.com/send_email.htm as a GET request.
- Since a GET request simply appends the form data to the URL, the user will be sent to the following URL (assuming he entered “bob@example.com” as the recipient, “hello” as the subject, and “What’s the status of that proposal?” as the message):

```
http://example.com/send_email.htm?to=bob%
40example.com&subject=hello&msg=What%27s+the+
status+of+that+proposal%3F
```
[3]

CSRF Example contd.

- The page send_email.htm would take the data it received and send an email to the recipient from the user.
- Note that send_email.htm simply takes data and performs an action with that data.
- It does not care where the request originated, only that the request was made.
- This means that if the user manually typed in the above URL into his browser, example.com would still send an email!

```
http://example.com/send_email.htm?to=bob%
40example.com&subject=hi+Bob&msg=test
http://example.com/send_email.htm?to=alice%
40example.com&subject=hi+Alice&msg=test
http://example.com/send_email.htm?to=carol%
40example.com&subject=hi+Carol&msg=test
```

<img src=“ ”> loads whatever URI is set as the “src” attribute, even if the URI is not an image (because the browser can only tell the URI is image after loading it

```
<img src="http://example.com/send_email.htm?
to=mallory%40example.com&subject=Hi&msg=My+
email+address+has+been+stolen">
```

Authentication and CSRF

- CSRF attacks often exploit the authentication mechanisms of targeted sites.

- The root of the problem is that Web authentication normally assures a site that a request came from a certain user's browser; but it does not ensure that the user actually requested or authorized the request.

Site Authentication Mechanism

- For example, suppose that Alice visits a target site *T* .
- *T* gives Alice's browser a cookie containing a pseudorandom session identifier *sid*, to track her session.
- Alice is asked to log in to the site, and upon entry of her valid username and password, the site records the fact that Alice is logged in to session *sid*.
- When Alice sends a request to *T*, her browser automatically sends the session cookie containing *sid*.
- *T* then uses its record to identify the session as coming from Alice.

Exploiting the authentication

- Now suppose Alice visits a malicious site *M*.
- Content supplied by *M* contains Javascript code or an image tag that causes Alice's browser to send an HTTP request to *T*.
- Because the request is going to *T*, Alice's browser "helpfully" appends the session cookie *sid* to the request.
- On seeing the request, *T* infers from the cookie's presence that the request came from Alice, so *T* performs the requested operation on Alice's account.
- This is a successful CSRF attack.

- In general, *whenever authentication happens implicitly—there is a danger of CSRF attacks*.

- In principle, this danger could be eliminated by requiring the user to take an explicit, *unspoofable* action (such as re-entering a username and password) for each request sent to a site, but in practice this would cause major usability problems (inconvenience!)
- The most standard and widely used authentication mechanisms fail to prevent CSRF attacks, so a practical solution must be sought elsewhere.

CSRF Attack Vectors

- For an attack to be successful, the user must be logged-in to the target site and must visit the attacker's site or a site over which the attacker has partial control.
- If a server contains CSRF vulnerabilities and also accepts GET requests (as in the example shown), CSRF attacks are possible *without the use of JavaScript* (for example, a simple <img> tag can be used).
- If the server only accepts POST requests, JavaScript is required to automatically send a POST request from the attacker's site to the target site.

XSS attack (recap)

- A XSS attack occurs when an attacker injects malicious code (typically JavaScript) into a site for the purpose of targeting other users of the site.
- Malicious JavaScript embedded in a target site would be able to send and receive requests from any page on the site and access cookies set by that site.
- Protection from XSS attacks requires sites to carefully filter any user input to ensure that no malicious code is injected.

CSRF vs. XSS

- CSRF and XSS attacks differ in that XSS attacks require JavaScript, while CSRF attacks do not.
- XSS attacks require that sites accept malicious code, while with CSRF attacks malicious code is located on third-party sites.
- Filtering user input will prevent malicious code from running on a particular site, but it will not prevent malicious code from running on third-party sites.
- Since malicious code can run on third-party sites, protection from XSS attacks does not protect a site from CSRF attacks.

- If a site is vulnerable to XSS attacks, then it *is vulnerable* to CSRF attacks.
- If a site is completely protected from XSS attacks, it is most likely *still vulnerable* to CSRF attacks.

Preventing CSRF

- Allow GET requests to only *retrieve* data, not modify any data on the server
- Require all POST requests to include a pseudorandom value
- Use a pseudorandom value that is independent of a user's account

CSRF Defense

1. Check standard headers to verify the request is same origin, and
2. Check CSRF token

SOP (Same Origin Policy)

1. Determine the origin the request is coming from (source origin)
2. Determine the origin the request is going to (target origin)

- Referer header
- Origin header

CSRF Tokens

- **Synchronizer (CSRF) Tokens**
- Double cookie defense
- Encrypted token pattern
- Custom Header
  - E.g., X-Requested-With: XMLHttpRequest

Synchronizer Token

- Any state changing operation requires CSRF token
- Characteristics of CSRF token:
  - Unique per user session
  - Large random value
  - Generated by cryptographically secure RNG
- The CSRF token is added as a hidden field
- The server rejects the requested action if token validation fails

Alternate CSRF Defenses

- Re-authentication
- OTP
- CAPTCHA

Example

```
<form action="/transfer.do" method="post">
<input type="hidden" name="CSRFToken"
value="OWY4NmQwODE4ODRjN2Q2NTlhMmZlYWE...
wYzU1YWQwMTVhM2JmNGYxYjJiMGI4MjJjZDE1ZDZ...
MGYwMGEwOA==">
…
</form>
```

Reference

- Cross-Site Request Forgeries: Exploitation and Prevention
  http://www.cs.utexas.edu/~shmat/courses/cs378_spring09/zeller.pdf

- Robust defenses for Cross-site Request Forgery
  https://seclab.stanford.edu/websec/csrf/csrf.pdf

- RequestRodeo: Client Side Protection against Session Riding
  https://www.owasp.org/images/4/42/RequestRodeo-MartinJohns.pdf

Curiosity Exercise

- ClickJacking
- LikeJacking
- XSHM

Exercises

1. What are Cookies?
2. Browser cache & Browser fingerprint
3. Session ID
4. GET vs POST methods
5. HTTP is a stateless protocol, why is it called stateless?
6. HTTP Request Header values
7. XMLHttpRequest
8. Forbidden Headers list (what is their purpose)
9. X-Forwarded-Host header
10. Is HTTPS a stateful protocol?
11. Try Burpe Suite or Arachni or OWASP ZAP