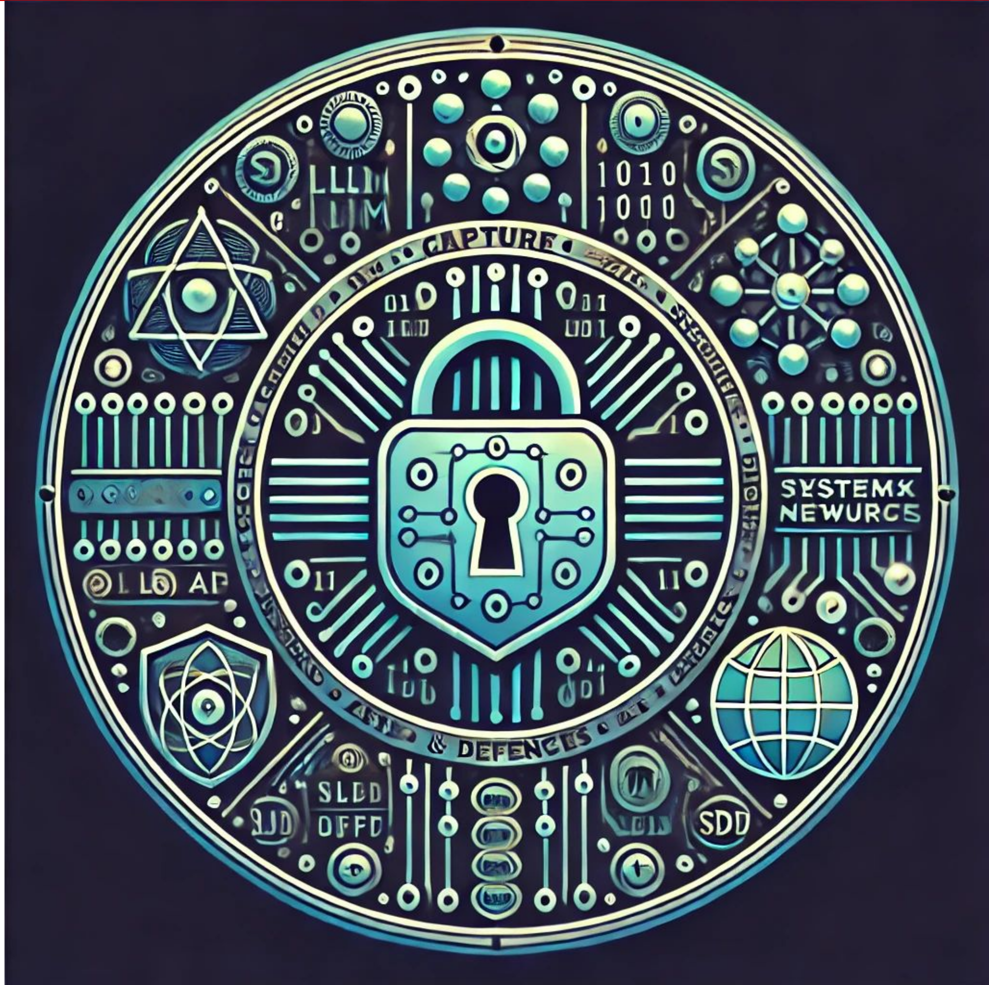
CSC-537 Systems Attacks and Defenses

Clickjacking, CSRF, and Sessions

Alexandros Kapravelos
akaprav@ncsu.edu



Clickjacking

What is Clickjacking

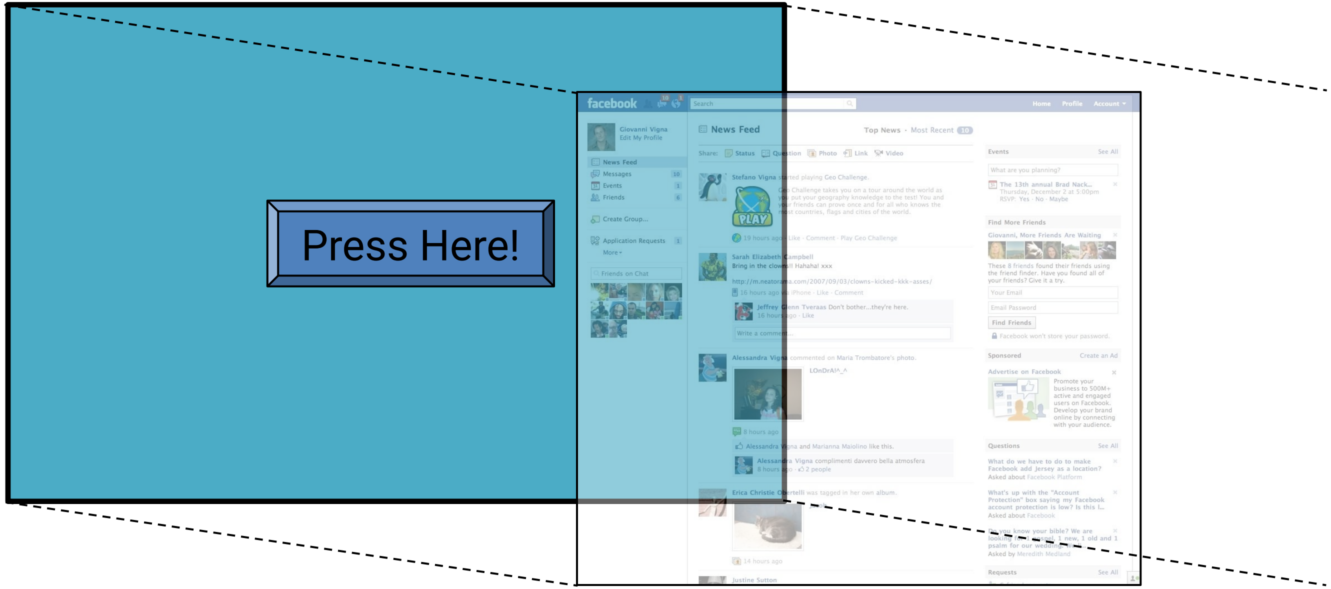
- UI redress attack using deceptive overlays
- Tricks users into clicking hidden or invisible elements
- Relies on framing techniques to mislead the user

Clickjacking Attack Mechanism

- Uses transparent iframes overlaying sensitive UI
- Displays decoy content to lure user clicks
- Captures clicks to trigger unauthorized actions

ClickJacking Example

Z-level: 1
Opaque

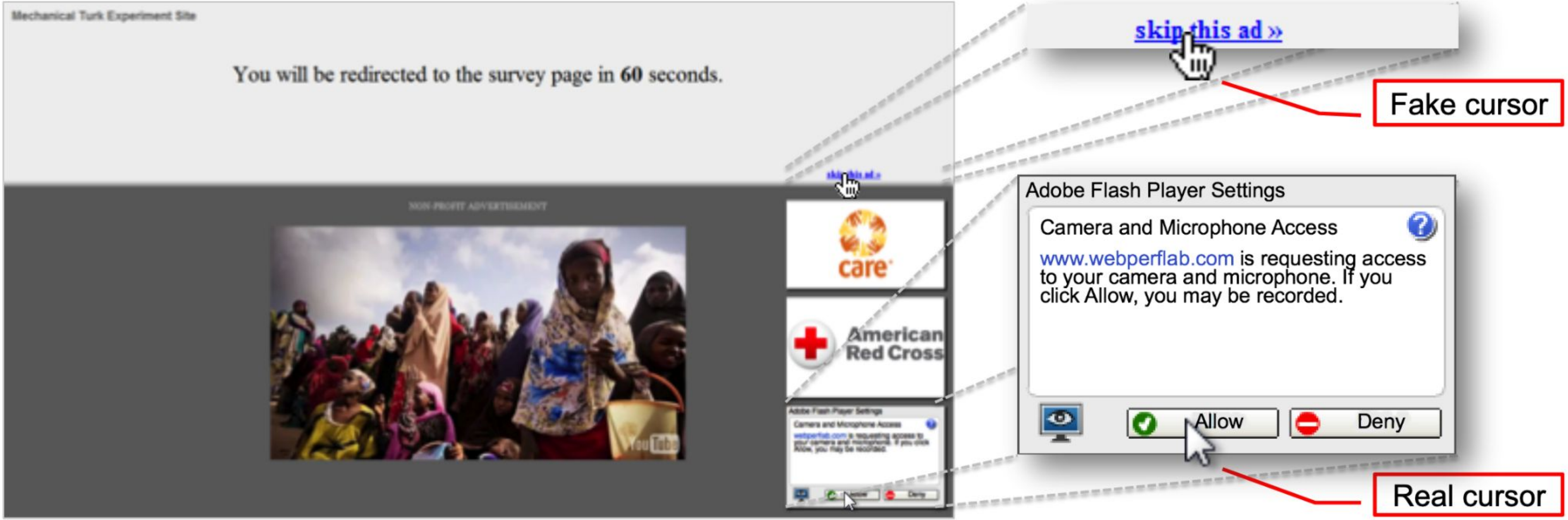


Z-level: 2
Transparent

Real-World Example of Clickjacking

- Early Flash vulnerabilities enabled unauthorized webcam access
  - attack against the Adobe Flash plugin settings page
  - by loading this page into an invisible iframe, an attacker could trick a user into altering the security settings of Flash
  - giving permission for any Flash animation to utilize the computer’s microphone and camera

Cursor spoofing attack page



Mechanical Turk Experiment Site

You will be redirected to the survey page in **60** seconds.

NON-PROFIT ADVERTISEMENT

Adobe Flash Player Settings

Camera and Microphone Access

www.webperflab.com is requesting access to your camera and microphone. If you click Allow, you may be recorded.

source: https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final39.pdf

Live Demo: Clickjacking Setup

- Create a decoy page with a visible button for a fake prize
- Overlay an invisible iframe targeting a sensitive action
- Use HTML and CSS to position and style the elements

Code Snippet: Clickjacking HTML (Part 1)

```
<!-- Decoy Page -->
<button id="bait-btn">Click here for a
free kitten</button>
<iframe id="target-frame"
src="http://bank.example.com/transfer?acc
t=attacker&amount=1000"></iframe>
```

Code Snippet: Clickjacking CSS (Part 1)

```
#target-frame {
  opacity: 0
  position: absolute
  top: 0
  left: 0
  width: 800px
  height: 600px
}
#bait-btn {
  position: absolute
  top: 300px
  left: 200px
}
```

Defenses Against Clickjacking

- Use `X-Frame-Options` HTTP header to block framing
- Apply Content Security Policy with `frame-ancestors` directive
- Set `SameSite` attributes on sensitive cookies

X-Frame-Options Header Defense

- Deny all framing using DENY
- Allow same-origin framing with SAMEORIGIN
- Goal: prevent attackers from loading your page in an iframe

```
X-Frame-Options: DENY
X-Frame-Options: SAMEORIGIN
X-Frame-Options: ALLOW-FROM https://trusted.example.com
```

nginx configuration:

```
add_header X-Frame-Options SAMEORIGIN always;
```

Content Security Policy for Clickjacking

- Use the directive `frame-ancestors ‘self’`
- Restricts which parent URLs can embed the current resource in frames
- Provides flexible control over allowed framing sources
- Enhances security when combined with `X-Frame-Options`

Using SameSite Cookies

- Prevent cookies from being sent on cross-site requests
- Mitigates risk if the attack relies on session credentials
- Works in conjunction with token-based defenses

```
Set-Cookie: session=0F8tgdOhi9ynR1M9wa3ODa; SameSite=Strict
```

Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF)

- CSRF exploits a logged-in user’s session
- Forces the browser to send unauthorized requests
- Can result in unintended state changes in web applications

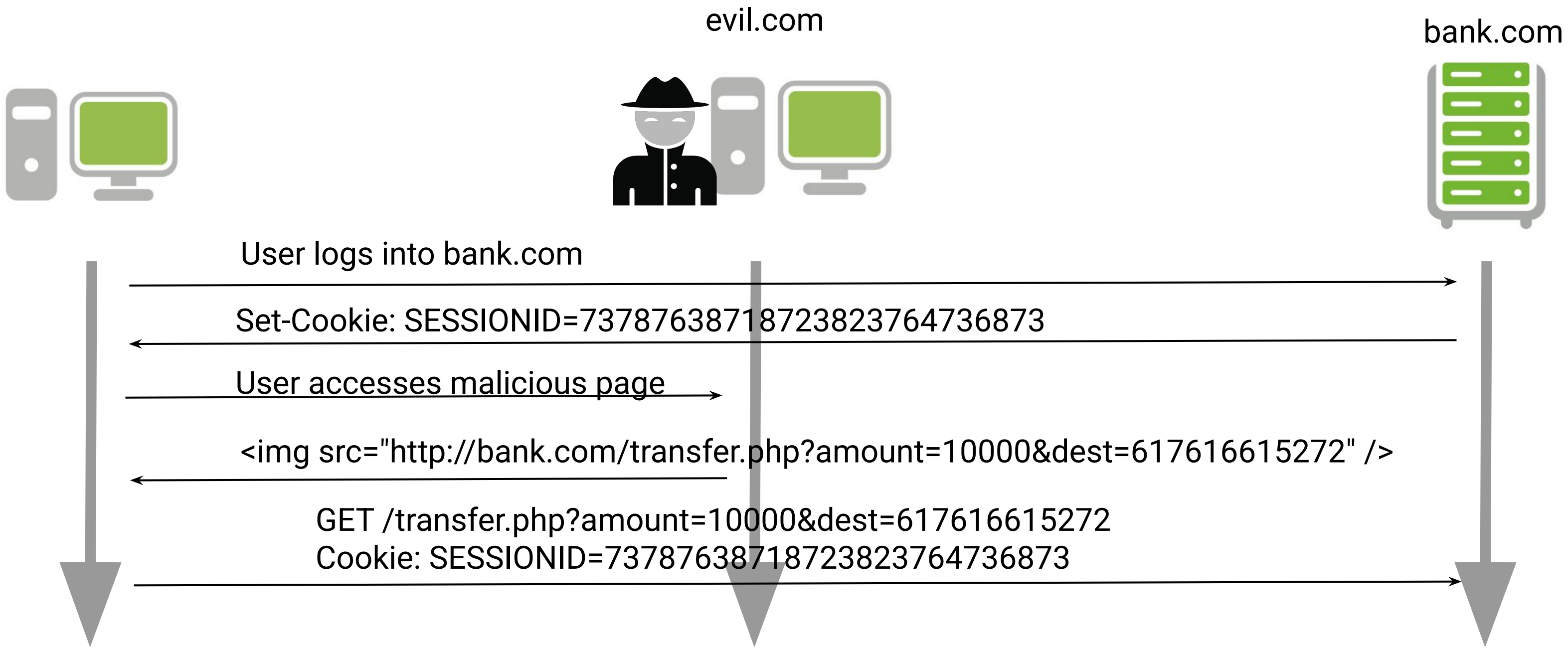
What is CSRF

- Attack forces a user’s browser to make an unintended request
- Exploits the automatic inclusion of session cookies
- Results in actions performed without the user’s intent

How CSRF Works

- Victim visits a malicious page
- The page sends a forged request to a trusted site
- Browser includes valid session cookies automatically

Cross-Site Request Forgery



evil.com

bank.com

User logs into bank.com

Set-Cookie: SESSIONID=73787638718723823764736873

User accesses malicious page

<img src="http://bank.com/transfer.php?amount=10000&dest=617616615272" />

GET /transfer.php?amount=10000&dest=617616615272
Cookie: SESSIONID=73787638718723823764736873

Real-World CSRF Case Study

- ING Direct Bank CSRF (2008)
  - vulnerability allowed unauthorized fund transfers
- Attack on MetaFilter changed user emails without consent
- More real-world examples here

Live Demo: CSRF with Auto-submitting Form

- Create a vulnerable PHP endpoint performing a sensitive action
- Build an attacker page with a hidden auto-submitting form

Code Snippet: CSRF Auto-submitting Form

```
<form id="attackForm"
action="http://vulnerable-bank.test/transfer
.php" method="POST">
  <input type="hidden" name="acct"
value="attacker">
  <input type="hidden" name="amount"
value="1000">
</form>
<script>

document.getElementById('attackForm').submit
()
</script>
```

CSRF Using Image Tag GET Request

- Exploit GET requests that trigger state changes
- Use an image tag with the target URL as src
- Browser sends cookies automatically with the request

Code Snippet: CSRF via Image Tag

```
<img
src="http://vulnerable-bank.test/transfer
.php?acct=attacker&amount=1000"
style="display:none">
```

Defense: Anti-CSRF Tokens

- Generate a unique token per session or request
- Include token in every state-changing form
- Verify token on the server side to validate request

Code Example: PHP Anti-CSRF Token (Form)

```
<?php
session_start()
if (empty($_SESSION['csrf_token'])) {
  $_SESSION['csrf_token'] =
bin2hex(random_bytes(32))
} ?>
<form action="/transfer.php" method="POST">
  <input type="hidden" name="csrf_token"
value="<?= $_SESSION['csrf_token'] ?>">
  <input type="text" name="acct">
  <input type="number" name="amount">
  <button type="submit">Transfer</button>
</form>
```

On form submission, verify token

```
<?php
if ($_POST['csrf_token'] !==
$_SESSION['csrf_token']) {
die("CSRF validation failed");//illegitimate
}
// else proceed with the transfer...
?>
```

Defense: SameSite Cookie Attribute for CSRF

- Set cookies with `SameSite=Lax` or Strict
- Prevents cookies from being sent in cross-site requests
- Reduces risk of CSRF if implemented correctly

Defense: Referer and Origin Validation

- Check the Origin or Referer header on incoming requests
- Reject requests that do not originate from trusted domains
- Acts as an additional layer of defense

Session Management

Session Management

- Sessions maintain user authentication state
- Vulnerabilities include hijacking and fixation
- Attackers exploit weak session handling to impersonate users

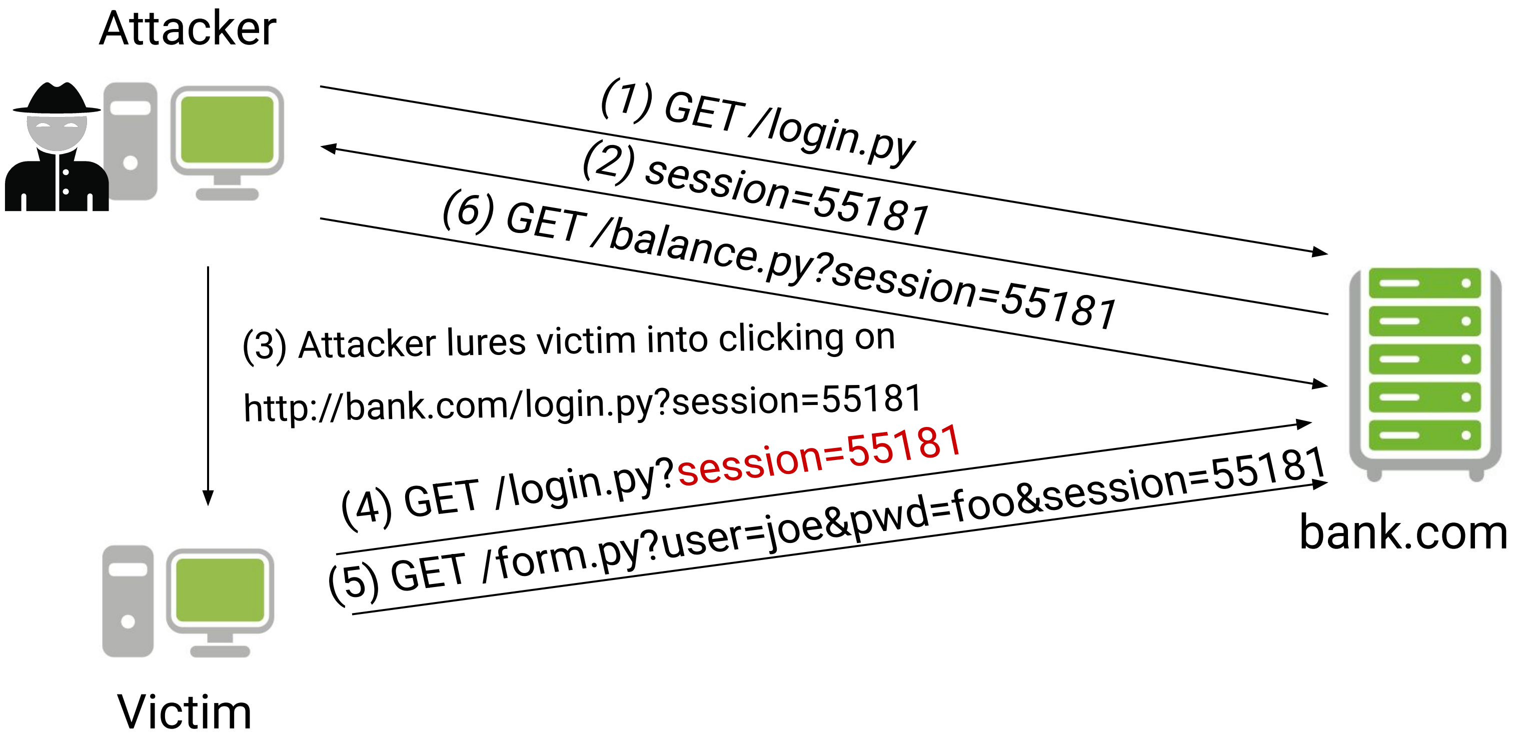
What is Session Hijacking

- Attackers steal valid session tokens
- Use intercepted or guessed cookies to impersonate users
- Bypasses authentication entirely

What is Session Fixation

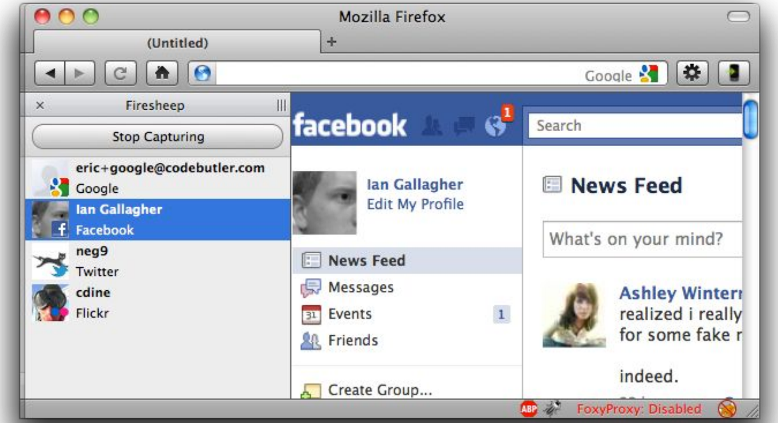
- Attacker sets a known session ID for the victim
- Victim logs in with the fixed session ID
- Attacker uses the known ID to access the victim’s account

Session Fixation


Attacker
(1) GET /login.py
(2) session=55181
(6) GET /balance.py?session=55181
(3) Attacker lures victim into clicking on
http://bank.com/login.py?session=55181
(4) GET /login.py?session=55181
(5) GET /form.py?user=joe&pwd=foo&session=55181
bank.com
Victim

Real-World Session Hijacking: Firesheep

- Firesheep enabled hijacking over open Wi-Fi networks
- Captured session cookies from popular websites
- Led to widespread adoption of HTTPS and secure cookie practices



Live Demo: Session Hijacking with Python

- Simulate stealing a session cookie from a vulnerable application
- Use Python requests to mimic a hijacked session
- Show retrieval of a user profile using a stolen PHPSESSID

```
import requests
cookies = {'PHPSESSID':
'KNOWN_SESSION_ID_VALUE'}
resp =
requests.get('http://localhost/demo/profi
le.php', cookies=cookies)
print(resp.text)
```

Live Demo: Session Fixation with PHP

- Demonstrate session fixation by allowing session ID in URL
- Victim logs in with attacker-controlled session ID
- Attacker uses the same ID to access the victim’s account

```
// Vulnerable login page example
session_id($_GET['PHPSESSID'])
session_start()
// Process login and do not regenerate
session
```

Defense: Use HTTPS

- Encrypt all traffic using TLS
- Prevent session cookie interception over insecure networks
- Essential for protecting session integrity

Defense: HttpOnly Cookies

- Set cookies with HttpOnly attribute
- Prevents access to cookies via JavaScript
- Mitigates theft through XSS

Defense: Regenerate Session IDs on Login

- Issue a new session ID after successful login
- Prevents session fixation attacks
- Use built-in functions to regenerate securely

Code Example: PHP Session Regeneration

```php
// After successful login
session_regenerate_id(true)
$_SESSION['username'] = $user
```

Defense: Short Session Lifetimes

- Expire sessions after periods of inactivity
- Invalidate sessions on logout immediately
- Reduces the window for attackers to hijack sessions

Defense: SameSite Attribute for Sessions

- Configure `SameSite=Lax` or Strict on session cookies
- Prevents cross-site sending of session cookies
- Complements other session security measures

Interconnection of Attacks

- Clickjacking, CSRF, and Session Attacks exploit trust layers
- XSS can lead to session hijacking which in turn enables CSRF
- A layered defense-in-depth approach is necessary

Summary and Takeaway Points

- Clickjacking tricks users with deceptive UI overlays
- CSRF forces unauthorized actions by exploiting session cookies
- Secure session management prevents hijacking and fixation
- Use a combination of HTTP headers, tokens, HTTPS, and proper cookie settings for robust defense
- Always adopt a defense-in-depth strategy and stay updated with best practices