

Preventing XSRF attacks

CS487

Slides adapted from "Foundations of Security: What Every Programmer Needs To Know" by Neil Daswani, Christoph Kern, and Anita Kesavan (ISBN 1590597842; <http://www.foundationsofsecurity.com>). Except as otherwise noted, the content of this presentation is licensed under the Creative Commons 3.0 License.

HTTP Request Authentication

- HTTP is stateless, so web apps have to associate requests with users themselves
- *HTTP authentication*: username / passwd automatically supplied in HTTP header
- *Cookie authentication*: credentials requested in form, after POST app issues session token
- Browser returns session cookie for each request
- *Hidden-form authentication*: hidden form fields transfer session token
- Http & cookie authentication credentials cached

Lifetime of Cached Cookies and HTTP Authentication Credentials

- Temporary cookies cached until browser shut down, *persistent* ones cached until expiry date
- HTTP authentication credentials cached in memory, shared by all browser windows of a single browser instance
- Caching depends only on browser instance lifetime, not on whether original window is open

Credential Caching Scenario

- (1) Alice has browser window open, (2) creates new window (3) to visit our site, HTTP authentication credentials stored
- (4) She closes the window, but original one still open (5) later, she's lured to the hacker's site which causes a surreptitious request to our site utilizing the cached credentials

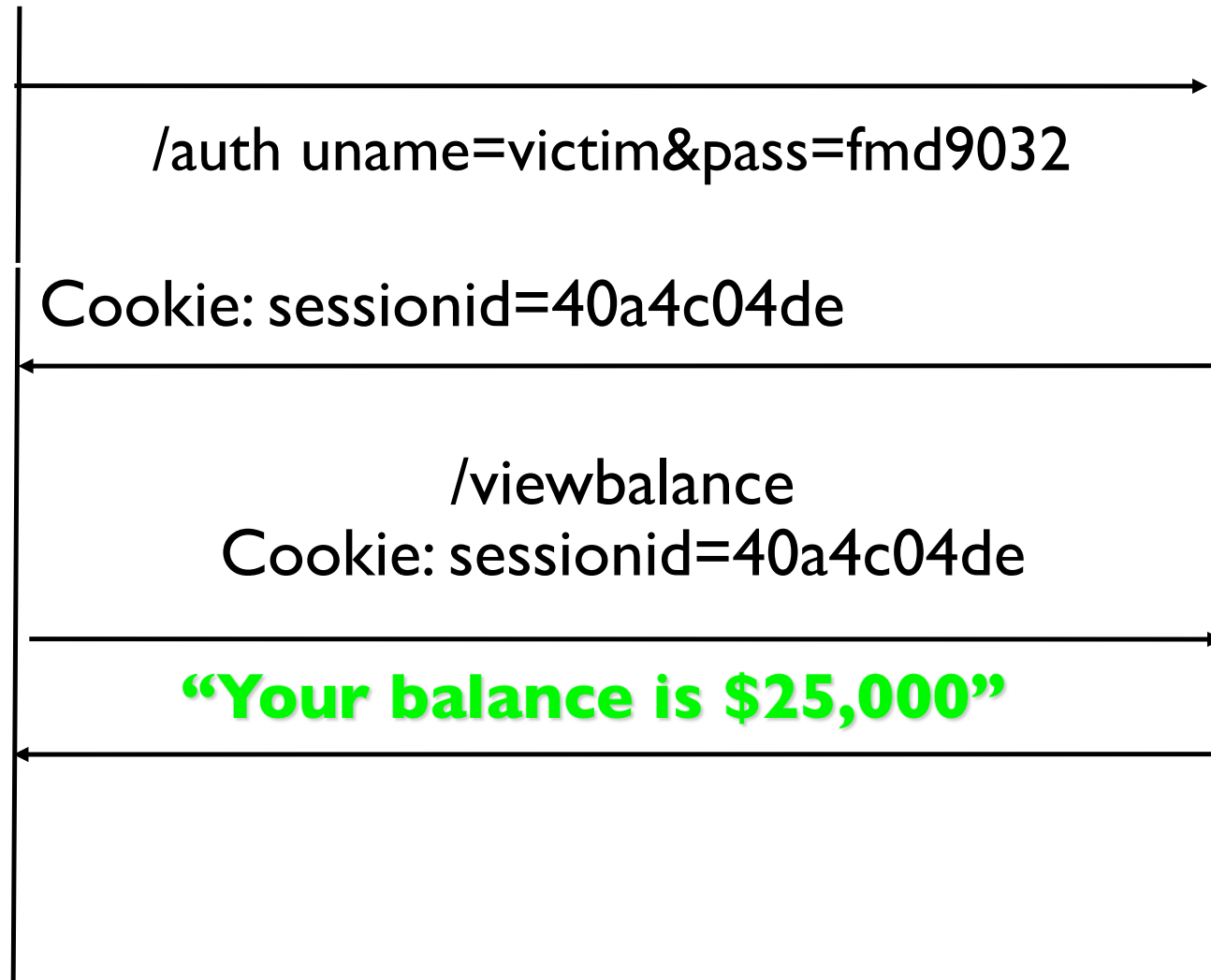
Cross-Site Request Forgery (XSRF)

- Malicious site can initiate HTTP requests to our app on Alice's behalf, w/o her knowledge
- Cached credentials sent to our server regardless of who made the request
- Ex: change password feature on our example
 - authenticates because cookies are sent

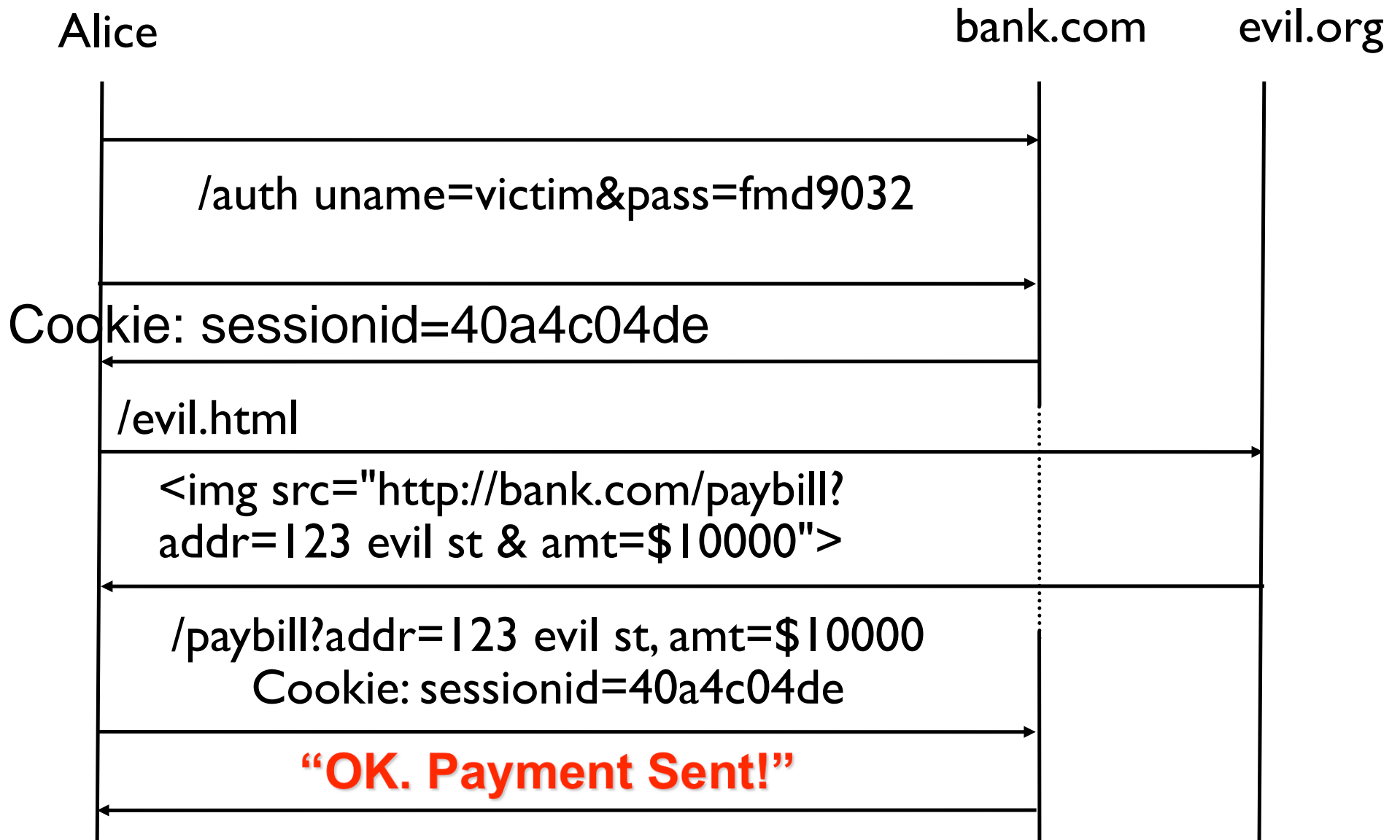
Example: Normal Interaction

Alice

bank.com



XSRF Attack



XSRF Example with scripts



evilform

1. Alice's browser loads page from `hackerhome.org`
2. Evil Script runs causing `evilform` to be submitted with a password-change request to our "good" form:
`www.mywwwservice.com/`
`update_profile` with a
`<input type="password"`
`id="password">` field

XSRF Example with scripts



```
<form method="POST" name="evilform"
  target="hiddenframe" action="https://
www.mywwwservice.com/update_profile">
  <input type="hidden" id="password"
    value="evilhax0r">
```

```
</form>
```

```
<iframe name="hiddenframe" style="display: none">
  </iframe>
```

```
<script>document.evilform.submit();</script>
```

3. Browser sends authentication cookies to our app. We're hoodwinked into thinking the request is from Alice. Her password is changed to **evilhax0r!**

XSRF Impacts

- Malicious site can't read info, but can make *write* requests to our app!
- In Alice's case, attacker gained control of her account with full read/write access!
- Who should worry about XSRF?
 - Apps w/ server-side state: user info, updatable profiles such as username/passwd (e.g. Facebook)
 - Apps that do financial transactions for users (e.g. Amazon, eBay)
 - Any app that stores user data (e.g. calendars, tasks)

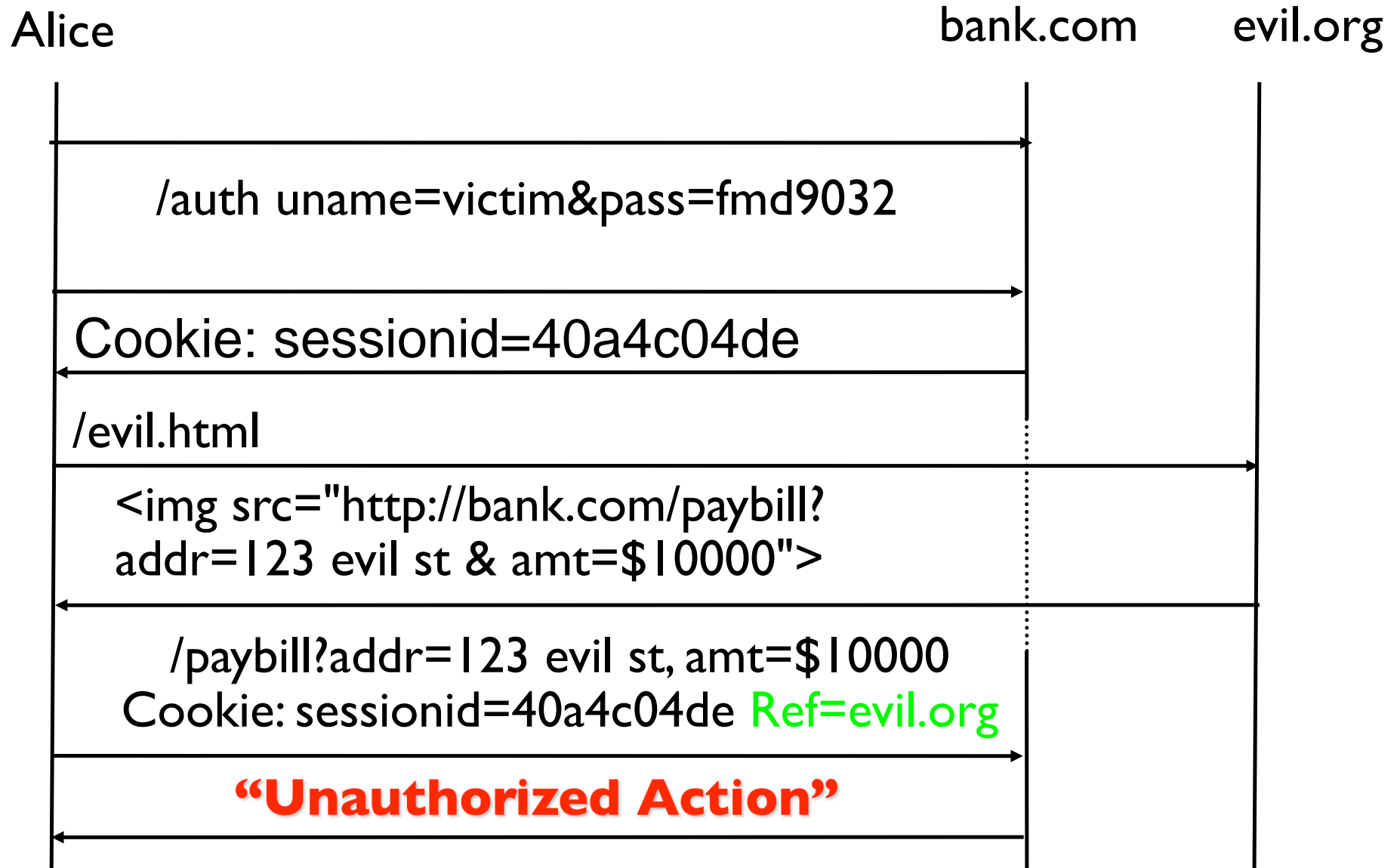
Preventing XSRF

- HTTP requests originating from user action are indistinguishable from those initiated by attacker
- Need own methods to distinguish valid requests
- Inspecting `Referer` Headers
- Validation via User-Provided Secret
- Validation via Action Token

10.3.1. Inspecting Referer Headers

- `Referer` header specifies the URI of document originating the request
- Assuming requests from our site are good, don't serve requests not from our site

Another XSRF Attack



10.3.1. Inspecting Referer Headers

OK, but not practical since it could be forged or blanked (even by legitimate users)

- For well-behaved browsers, reasonable to expect `Referer` headers to be accurate, if present
- But if blank, we can't tell if it's legitimate or not

Validation via User-Provided Secret

- Can require user to enter secret (e.g. login password) along with requests that make server-side state changes or transactions
- Ex: The change password form (10.2.1) could ask for the user's current password
- Balance with user convenience: use only for infrequent, "high-value" transactions
 - Password or profile changes
 - Expensive commercial/financial operations

Validation via Action Token

- Add special *action tokens* as hidden fields to “genuine” forms to distinguish from forgeries
- Same-origin policy prevents 3rd party from inspecting the form to find the token
- Need to generate and validate tokens so that
 - Malicious 3rd party can't guess or forge token
 - Then can use to distinguish genuine and forged forms
 - How? We propose a scheme next.

Generating Action Tokens

- Concatenate value of timestamp or counter C with the Message Authentication Code (MAC) of C under secret key K :
 - Token: $T = MAC_K(c) || c$
 - Security dependent on crypto algorithm for MAC
 - $||$ denotes string concatenation, T can be parsed into individual components later

Validating Action Tokens

- Split token T into MAC and counter components
- Compute expected MAC for given C and check that given MAC matches
- If MAC algorithm is secure and K is secret, 3rd party can't create $\text{MAC}_K(c)$, so can't forge token

Problem with Scheme

- Application will accept *any* token we've previously generated for a browser
- Attacker can use our application as an *oracle!*
 - Uses own browser to go to page on our site w/ form
 - Extracts the token from hidden field in form
- Need to also verify that incoming request has action token sent to the *same* browser (not just *any* token sent to *some* browser)

Fixing the Problem

- Bind value of action token to a cookie
 - Same-origin policy prevents 3rd party from reading or setting our cookies
 - Use cookie to distinguish between browser instances
- New Scheme
 - Cookie C is unpredictable, unique to browser instance
 - C can be session authentication cookie
 - Or random 128 bits specifically for this purpose

Validation in New Scheme

- Extract request URL L' (w/o query part for GET request) and cookie C' .
- Compute expected value of action token:
 - $T_{expected} = MAC_K(C' || d || L')$
- Extract actual $T_{request}$ of action token from appropriate request parameter
- Verify $T_{expected} = T_{request}$, otherwise reject
- Occasionally legitimate request may fail
 - Ex: user leaves page w/ form open and initiates new session in different window; action token for original form becomes “stale”

Security Analysis of the Action Token Scheme

- Value of token chosen to be unguessable
 - Output of cryptographically strong MAC algorithm
- Only way to obtain token (w/o key) is to use our app as an oracle
 - This also requires the user's session cookie
 - Assume attacker doesn't have this otherwise he could already directly hijack the session anyway
 - Session cookies are also hard to guess

Security Analysis: Leakage of Tokens

- For GET requests, action token visible as query parameter in request URL
 - Would appear in proxy and web server logs
 - Could be leaked in `Referer` header if page contains references (images, links) to 3rd party documents
- HTTP spec recommends POST instead of GET
- Scheme incorporates target action URL into MAC computation
 - If one URL is leaked, can't be used against another
 - Use fresh cookie for each browser instance, so stolen action token not usable for future sessions