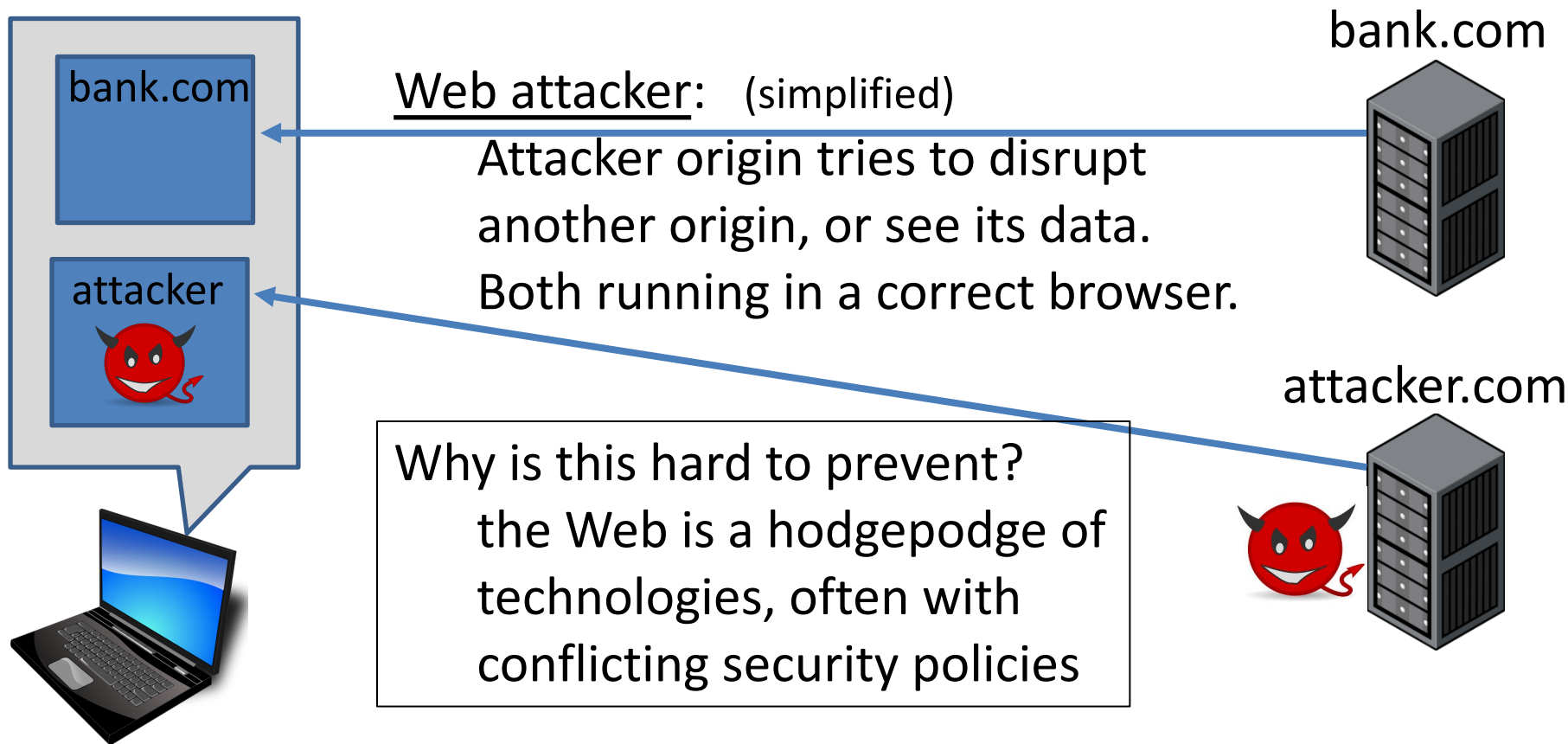




Web security

Web Session Management

Recap: Web attacker model



Recap: same origin policy

Review: Same Origin Policy (SOP) for DOM:

- Origin A can access origin B's DOM if match on
(scheme, domain, port)

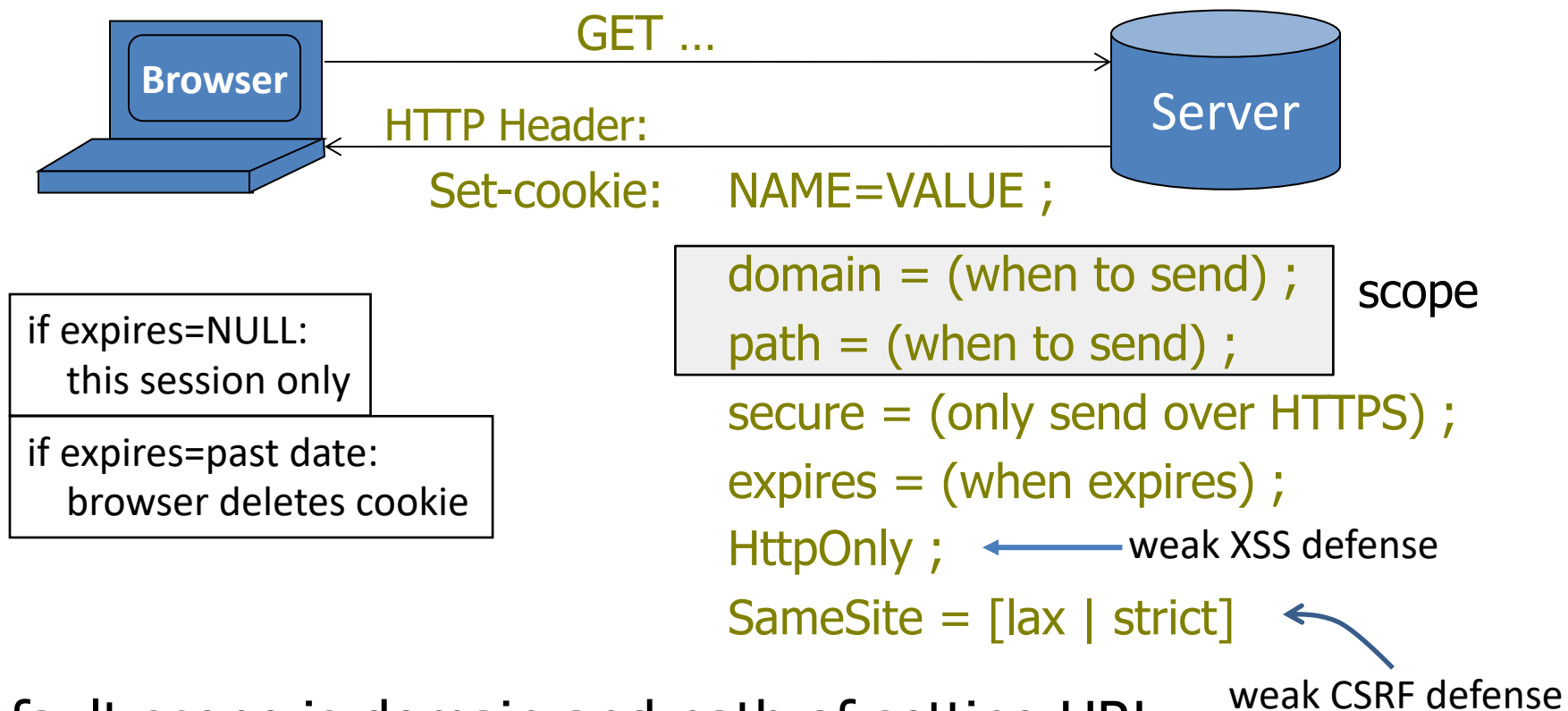
Review: Same Original Policy (SOP) for cookies:

- Based on: **([scheme], domain, *path*)**


optional

scheme://domain:port/path?params

Setting/deleting cookies by server



Remember: Cookies have no integrity

User can change and delete cookie values

- Edit cookie database (FF: cookies.sqlite)
- Modify Cookie header (FF: TamperData extension)

Silly example: shopping cart software

Set-cookie: **shopping-cart-total = 150** (\$)

User edits cookie file (cookie poisoning):

Cookie: **shopping-cart-total = 15** (\$)

Similar problem with localStorage and hidden fields:

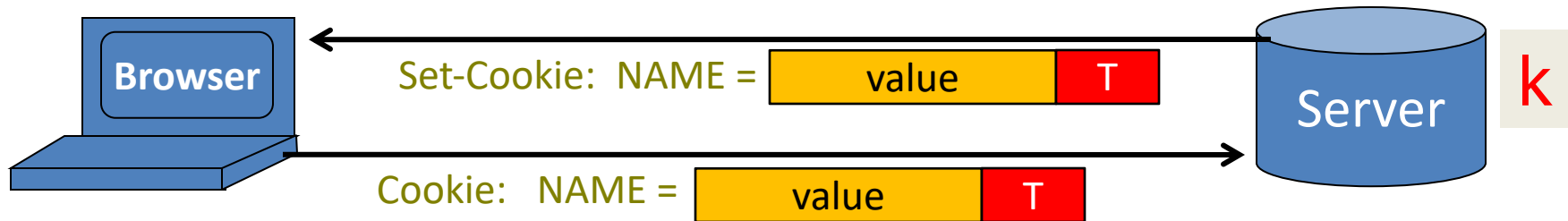
<INPUT TYPE="hidden" NAME=price VALUE="150">

Solution: cryptographic checksums

Goal: data integrity

Requires server-side secret key k unknown to browser

Generate tag: $T \leftarrow \text{MACsign}(k, (\text{SID}, \text{name}, \text{value}))$



Verify tag: $\text{MACverify}(k, (\text{SID}, \text{name}, \text{value}), T)$

Binding to session-id (SID) makes it harder to replay old cookies

Example: ASP.NET

`System.Web.Configuration.MachineKey`

- Secret web server key intended for cookie protection

Creating an encrypted cookie with integrity:

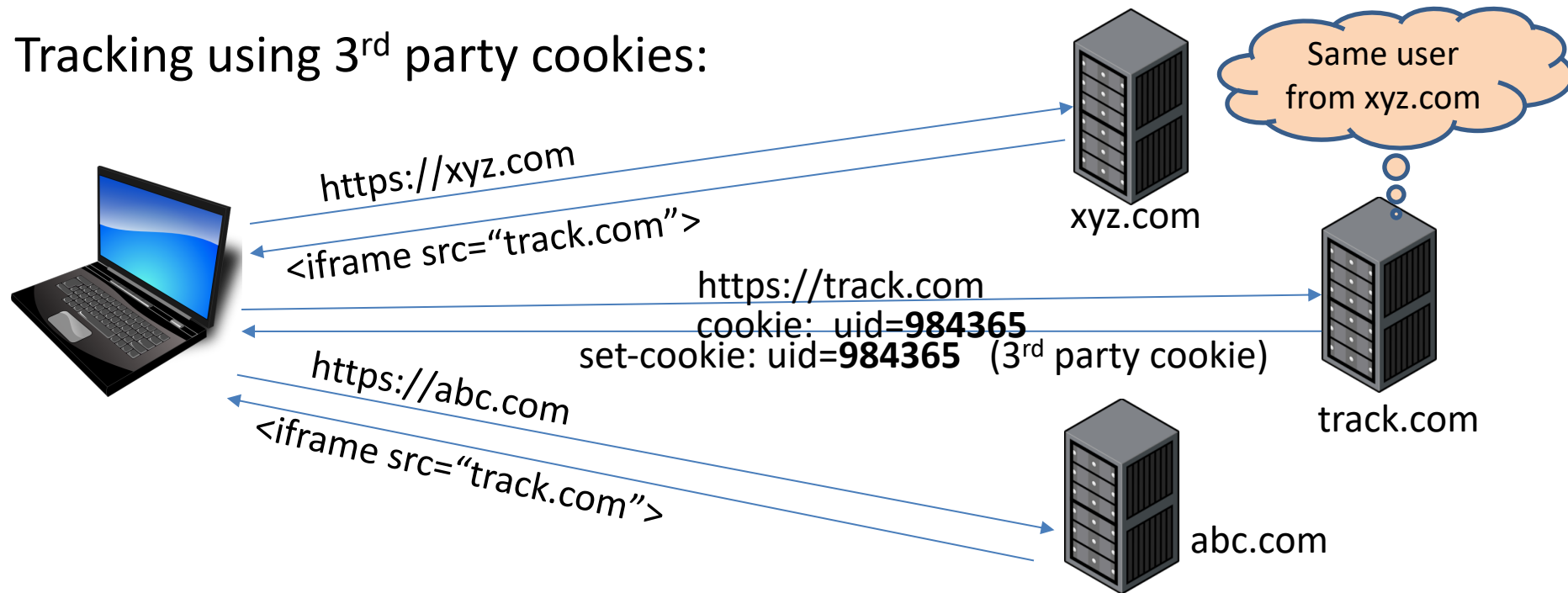
```
HttpCookie cookie = new HttpCookie(name, val);  
HttpCookie encodedCookie =  
    HttpSecureCookie.Encode (cookie);
```

Decrypting and validating an encrypted cookie:

```
HttpSecureCookie.Decode (cookie);
```

Chrome blocks 3rd party cookies

Tracking using 3rd party cookies:



2021: Chrome will block 3rd party cookies

Privacy Sandbox: alternative tech. for advertisers with better privacy properties



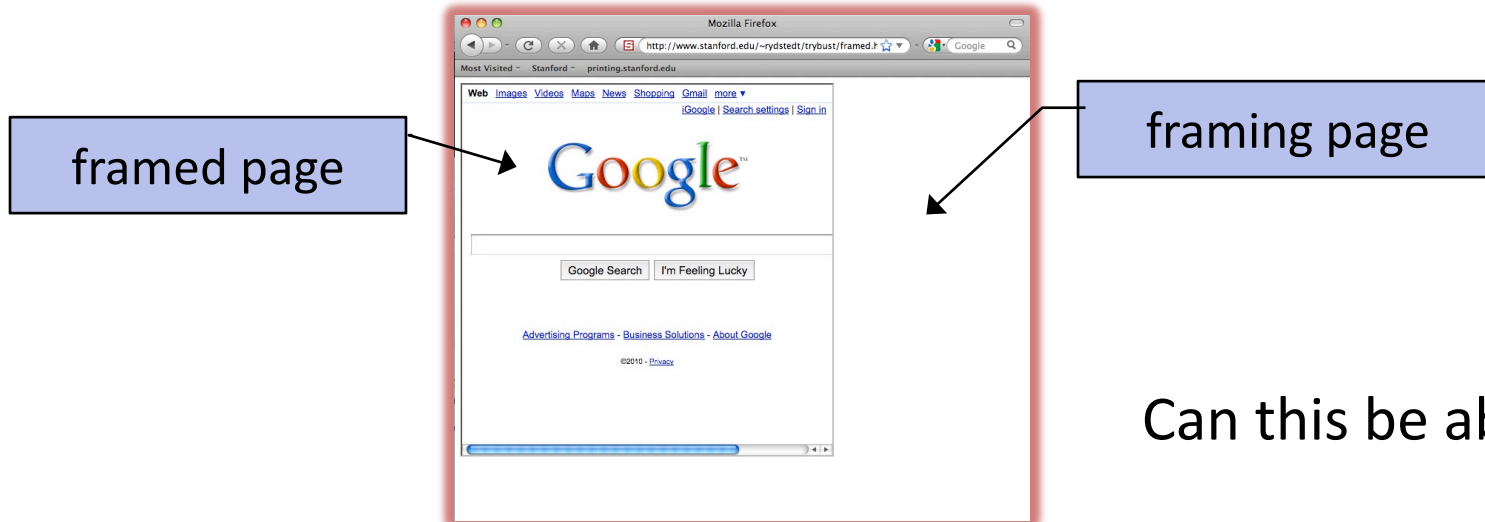
Clickjacking Attacks

Framing: one site can frame another

```
<iframe name="myframe" src="http://www.google.com/">
```

This text is ignored by most browsers.

```
</iframe>
```



Can this be abused?

Clickjacking

[rsnake'08]

User visits a gaming web site. Game frames Twitter site as:
(Twitter frame is occluded by game frame)



Tap-jacking attacks [RBB'10]

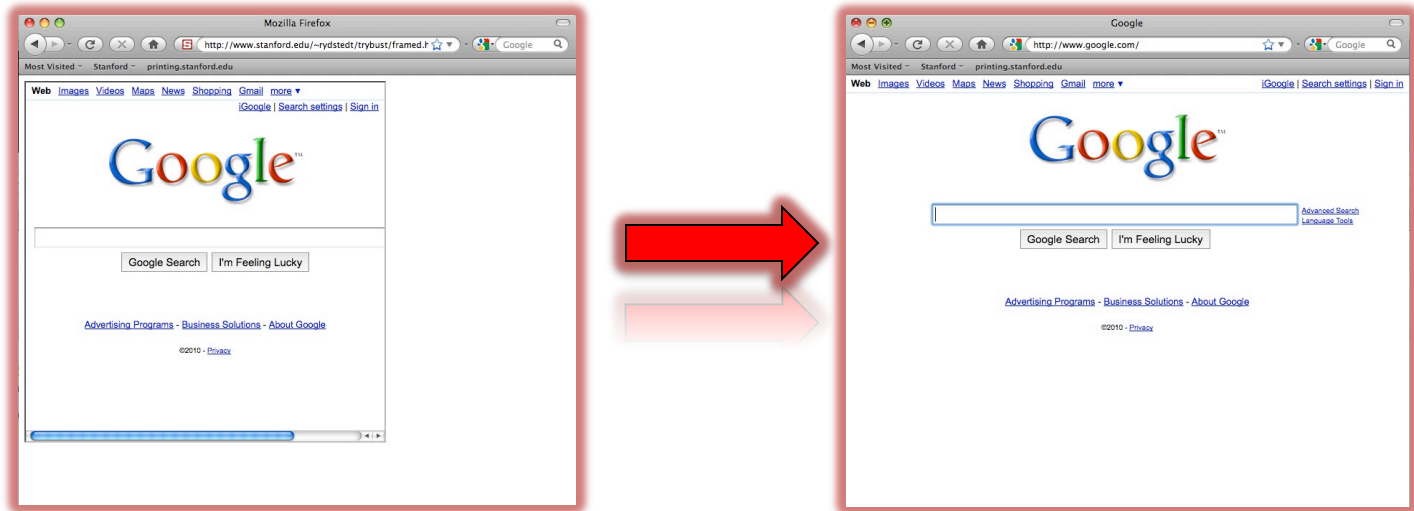
User visits a gaming web site:

- Can zoom, and auto scroll
- Web site displays an incoming text message screen, but frames Twitter



Incorrect solution: framebusting

Code on a page that prevents other pages from framing



```
if (top != self) { top.location = self.location; }
```

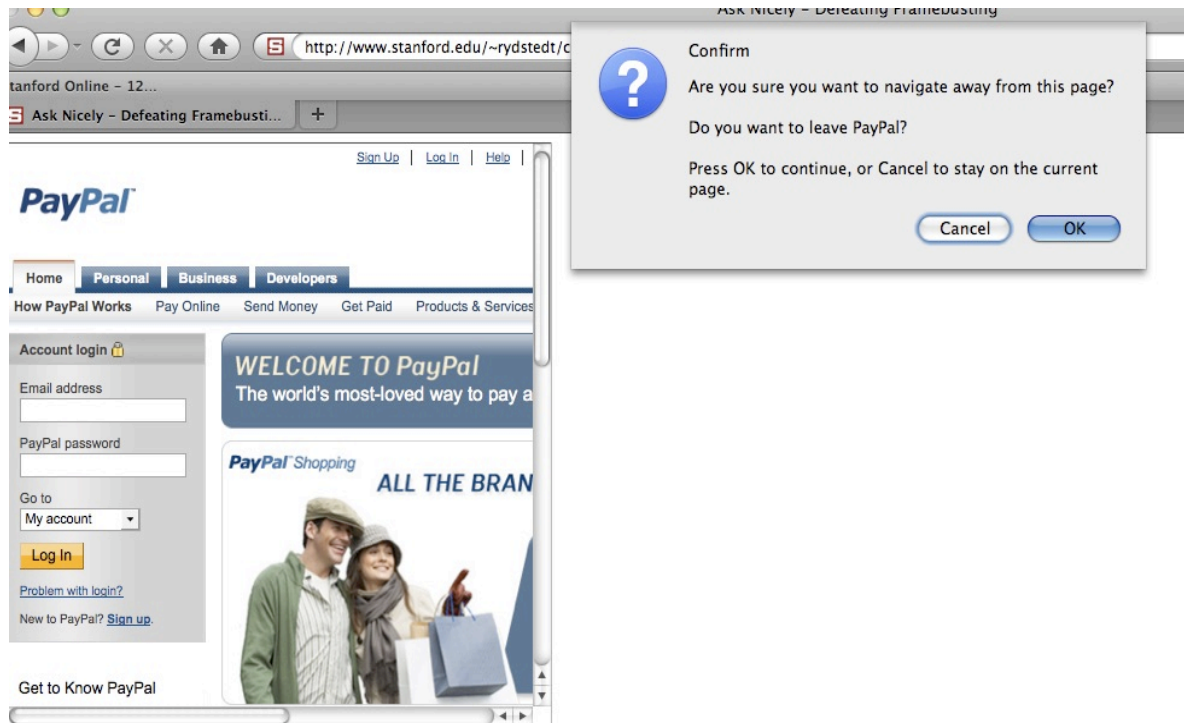
Many attacks

Example: on framing page:

```
<script>
  window.addEventListener('beforeunload', function(e) {
    e.preventDefault();
  })
</script>
<iframe src="http://www.paypal.com"> </iframe>
```

User likely to hit cancel and cancel framebusting

What the user will see

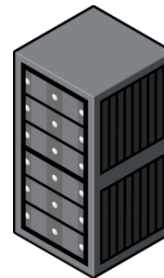


Correct defense: CSP

web browser



example.com



HTTP response from server:

HTTP/1.1 200 OK

...

Content-Security-Policy: frame-ancestors 'none';

...

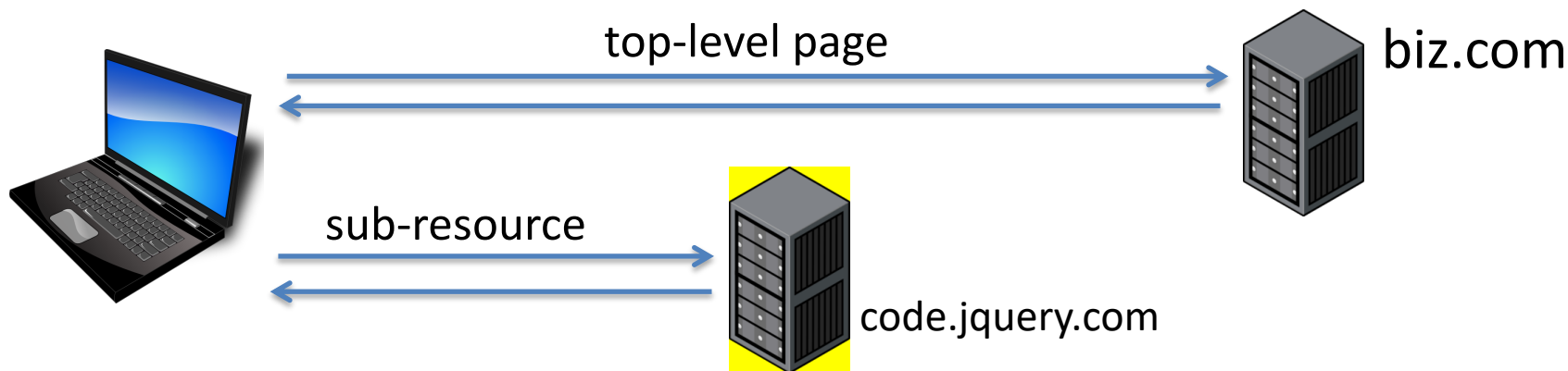
`<iframe src='example.com'>`
will cause an error

frame-ancestors 'self' ;
means only example.com
can frame page



Sub Resource Integrity (SRI)

The sub-resource integrity problem



in top-level page:

```
<script src=https://code.jquery.com/jquery-3.5.1.min.js> </script>
```

What if the sub-resource site serves bad content?

Sub-resource integrity (SRI)

SRI attribute can be added to scripts and style sheets:

```
<script src="https://code.jquery.com/jquery-3.5.1.min.js"  
        integrity="sha256-9/aliU8dGd2tb6OSsuzixeV4y/faTqgFtohetphbbj0=""  
        crossorigin="anonymous">  
</script>
```

```
<link rel='stylesheet' type='text/css' href='https://example.com/style.css'  
      integrity="sha256-9/aliU8dGd2tb6OSsuzixeV4y/faTqgFtohetphbbj0=""  
      crossorigin="anonymous">
```

integrity attribute: precomputed hash of the the target sub-resource

What the browser does

```
<script    src="https://code.jquery.com/jquery-3.5.1.min.js"  
          integrity="sha256-9/aliU8dGd2tb6OSsuzixeV4y/faTqgFtohetphbbj0=""  
          crossorigin="anonymous">  
</script>
```

Browser: (1) load sub-resource, (2) compute hash of contents,
(3) compare value to the integrity attribute.

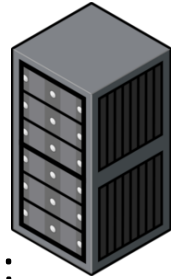
- if hash mismatch: script or stylesheet are not executed and an error is raised.

Enforcing SRI using CSP

web browser



example.com



HTTP response from server:

HTTP/1.1 200 OK

...

Content-Security-Policy: **require-sri-for** script style;

...

Requires SRI for all scripts and style sheets on page



WebAuthn

The Universal Second Factor (U2F) Standard

(and WebAuthn)

Goals:

- **Browser malware cannot steal user credentials**
- U2F should not enable tracking users across sites
- U2F uses counters to defend against token cloning



U2F token
(holds user credentials)



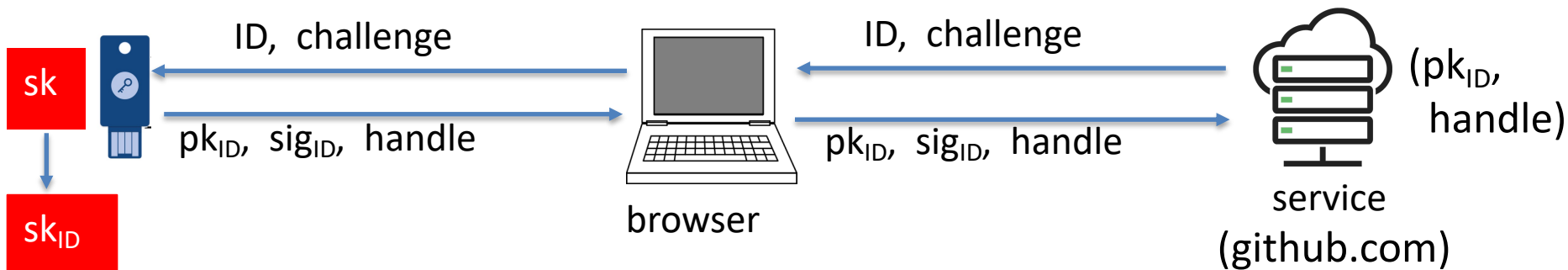
browser



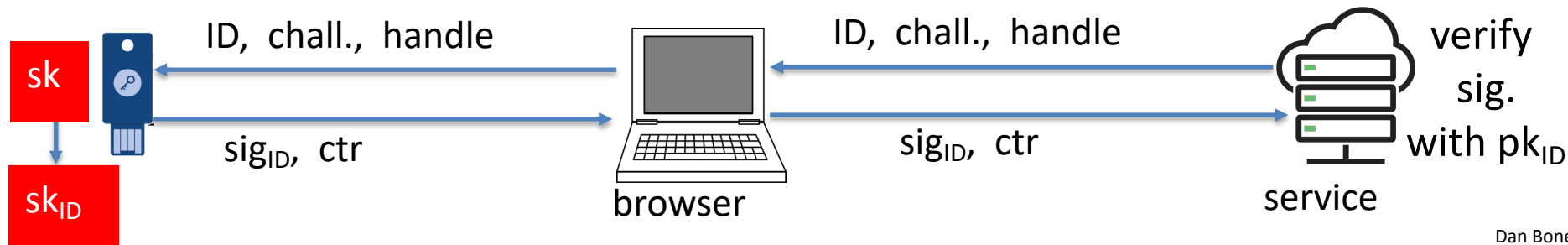
service (github.com)

The U2F protocol: two parts (simplified)

Device registration:

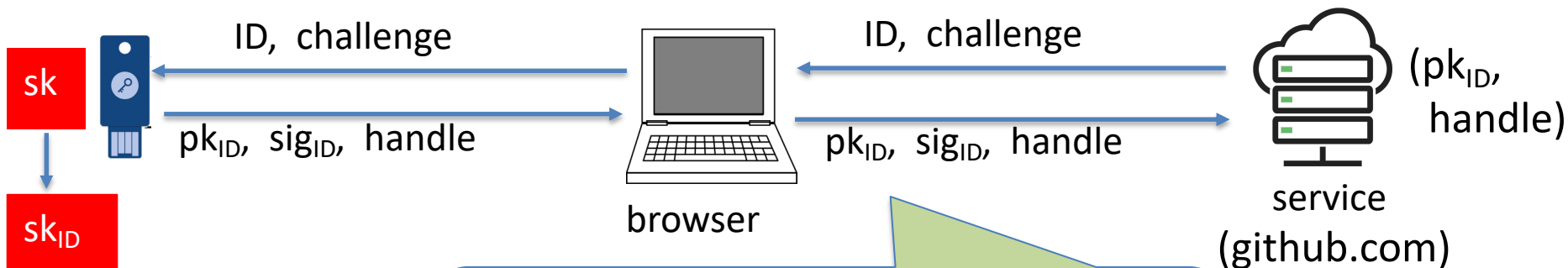


Authentication:

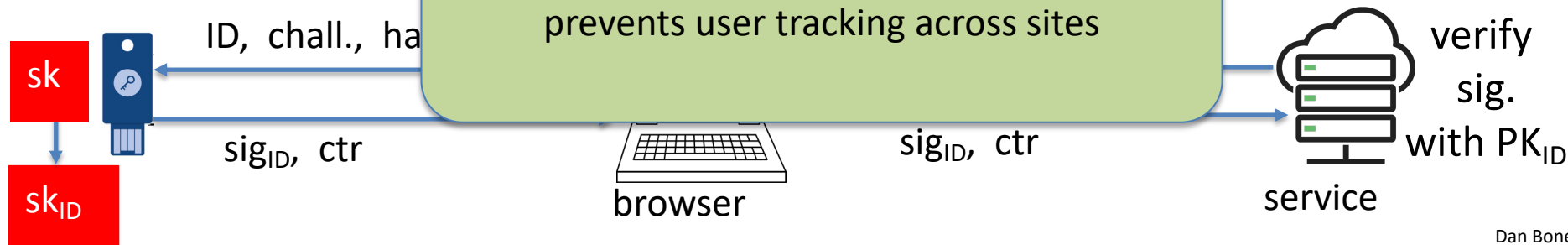


The U2F protocol: two parts (simplified)

Device registration:



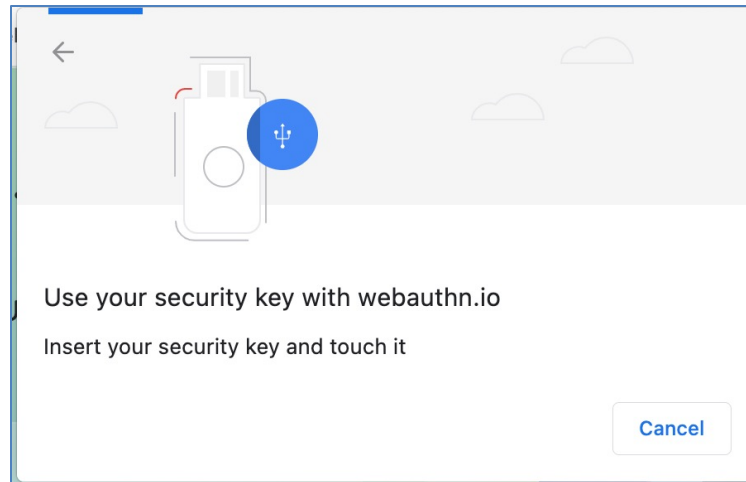
Authentication:



WebAuthn

A browser Javascript API:

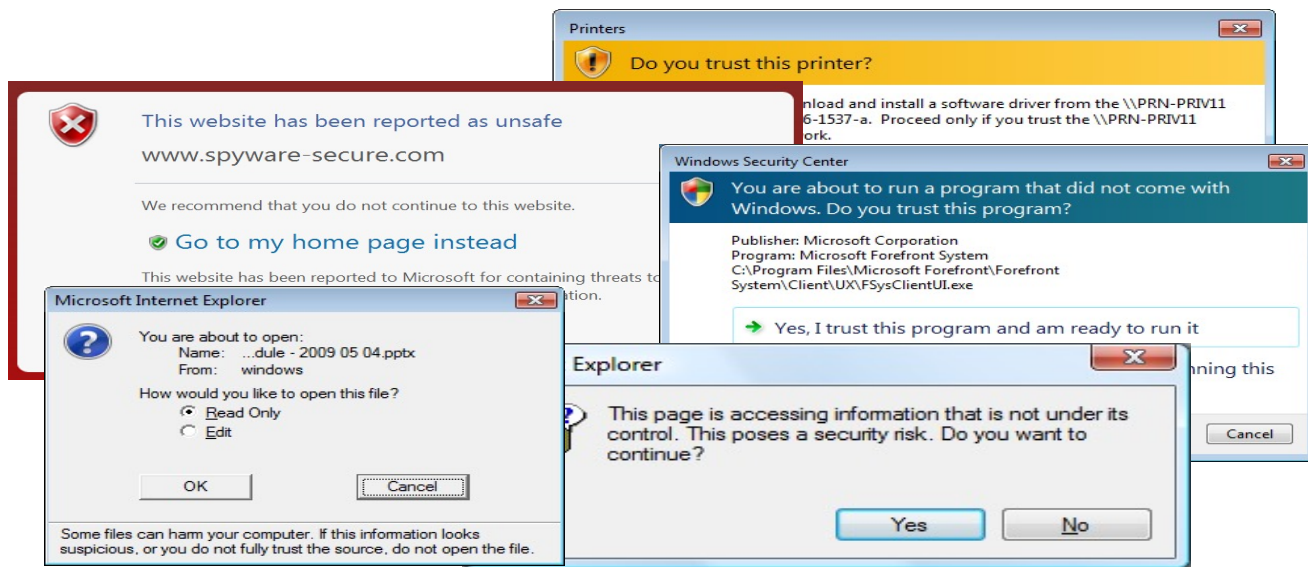
- Javascript from web site can register a U2F device, and later user can authenticate to web site with U2F device



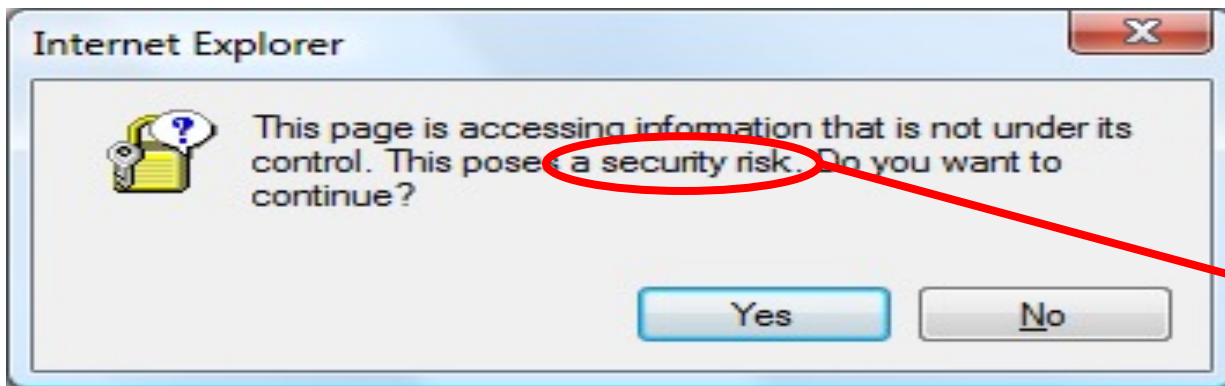


Interlude: Designing Security Prompts

Users are faced with a lot of challenging trust-related decisions



An example problem: IE6 mixed context

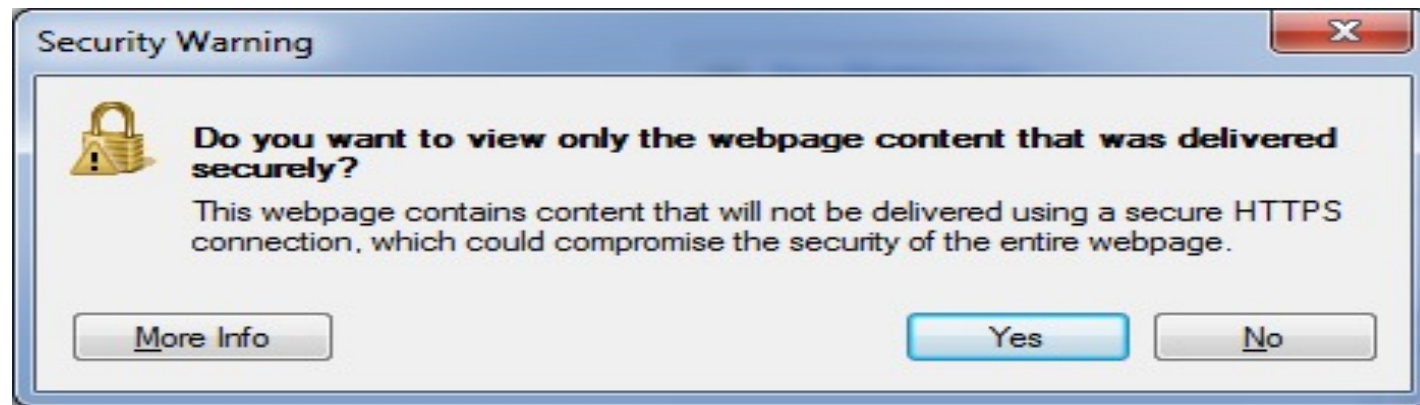


Vague threat.
What's the
risk? What
could happen?

"Yes", the possibly less
safe option, is the default

How should the user make
this decision? No clear
steps for user to follow.

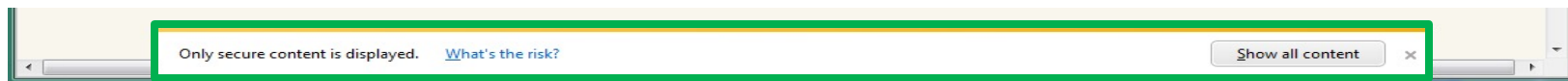
Better



(IE8)

Even better: load the safe content, and use the address bar to enable the rest

(IE9)

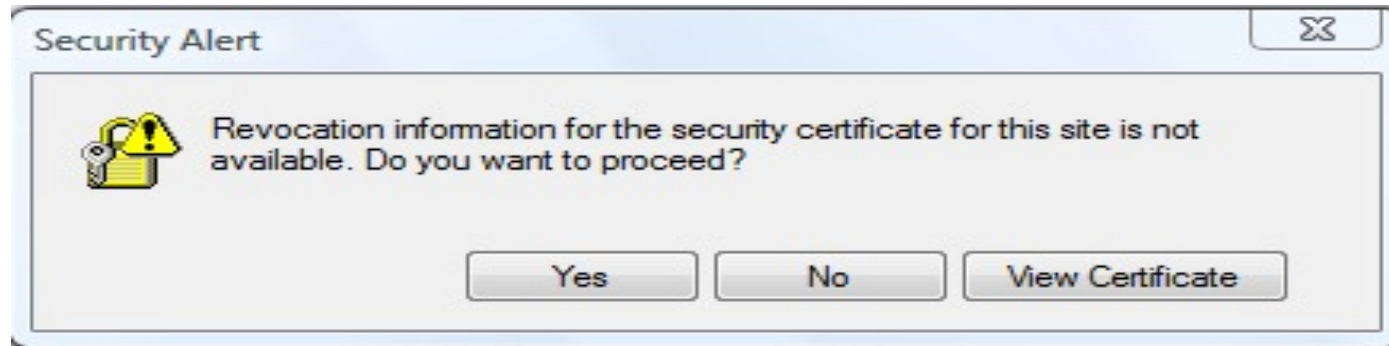


Guidelines

- Philosophy:
 - Does the user have unique knowledge the system doesn't?
 - Don't involve user if you don't have to
 - If you involve the user, enable them to make the right decision
- Make sure your security dialogs are NEAT:
 - **Necessary:** Can the system take action without the user?
If the user has no unique knowledge, redesign system.
 - **Explained:** *see next slides*
 - **Actionable:** Can users make good decisions with your UI in both malicious and benign situations?
 - **Tested:** Test your dialog on a few people who haven't used the system before -- both malicious and benign situations.

Example 1: bad explanation

IE6 CRL check failure notification



Most users will not understand “revocation information” .

Choices are unclear, consequence is unclear.

Better explanation

Source



There is a problem with this website's security certificate.

The security certificate presented by this website was not issued by a trusted certificate authority.


Risk

Security certificate problems may indicate an attempt to fool you or intercept any data you send to the server.

We recommend that you close this webpage and do not continue to this website.

Choices

 [Click here to close this webpage.](#)

 [Continue to this website \(not recommended\).](#)

 [More information](#)

Process

- If you arrived at this page by clicking a link, check the website address in the address bar to be sure that it is the address you were expecting.
- When going to a website with an address such as `https://example.com`, try adding the 'www' to the address, `https://www.example.com`.
- If you choose to ignore this error and continue, do not enter private information into the website.

For more information, see "Certificate Errors" in Internet Explorer Help.

In Chrome (2019)

Risk Explanation

Choices



Your connection is not private

Attackers might be trying to steal your information from **expired.badssl.com** (for example, passwords, messages, or credit cards). [Learn more](#)

NET::ERR_CERT_DATE_INVALID

- ☒ Help improve Safe Browsing by sending some [system information and page content](#) to Google.
[Privacy policy](#)

Advanced

Back to safety

In Chrome (2019)

☒ Help improve Safe Browsing by sending some system information and page content to Google.
[Privacy policy](#)

Hide advanced

Back to safety

This server could not prove that it is **expired.badssl.com**; its security certificate expired 1,483 days ago. This may be caused by a misconfiguration or an attacker intercepting your connection. Your computer's clock is currently set to Saturday, May 4, 2019. Does that look right? If not, you should correct your system's clock and then refresh this page.

[Proceed to expired.badssl.com \(unsafe\)](#)

(expired certificate)

Process

Choice

Example 2: bad explanation

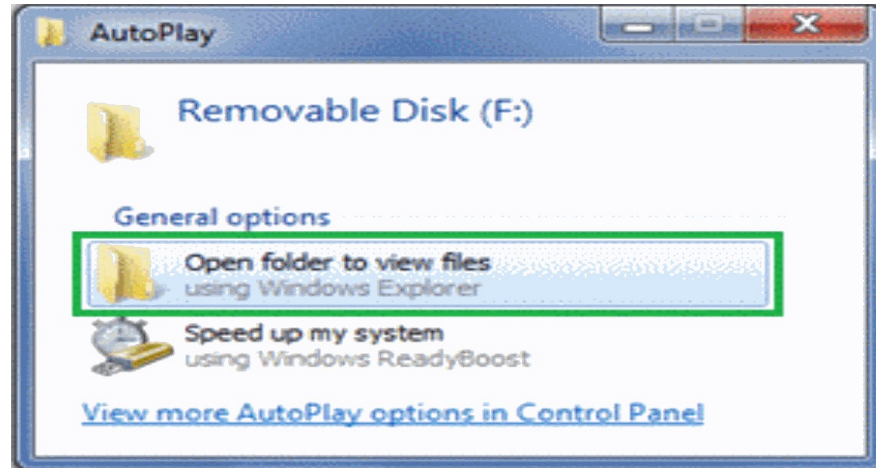
AutoPlay dialog in Vista



Attacker can abuse explanation causing bad user decisions.

Used by Conficker spread through USB drives.

A better design



Windows 7 AutoPlay removed the auto-run option



... back to Web security

Session Management and
User Authentication on
the Web

Sessions

A sequence of requests and responses from one browser to one (or more) sites

- Session can be long (e.g. Gmail) or short
- without session mgmt:
users would have to constantly re-authenticate

Session mgmt: authorize user once;

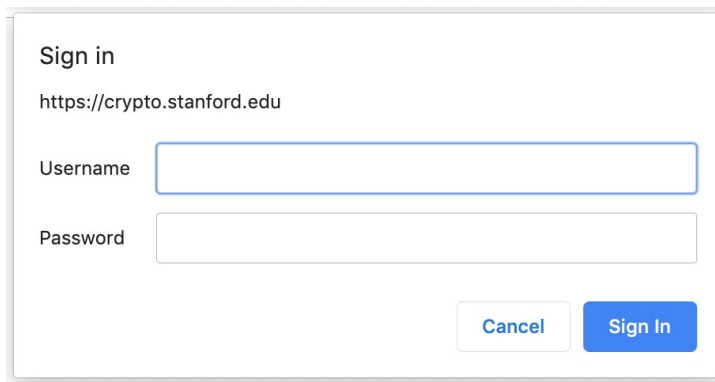
- All subsequent requests are tied to user

Pre-history: HTTP auth

HTTP request: GET /index.html

HTTP response contains:

WWW-Authenticate: Basic realm="Password Required"



Sign in

https://crypto.stanford.edu

Username

Password

Cancel Sign In

Browsers sends hashed password on all subsequent HTTP requests:

Authorization: Basic ZGFddfibzsdffgkjheczI1NXRleHQ=

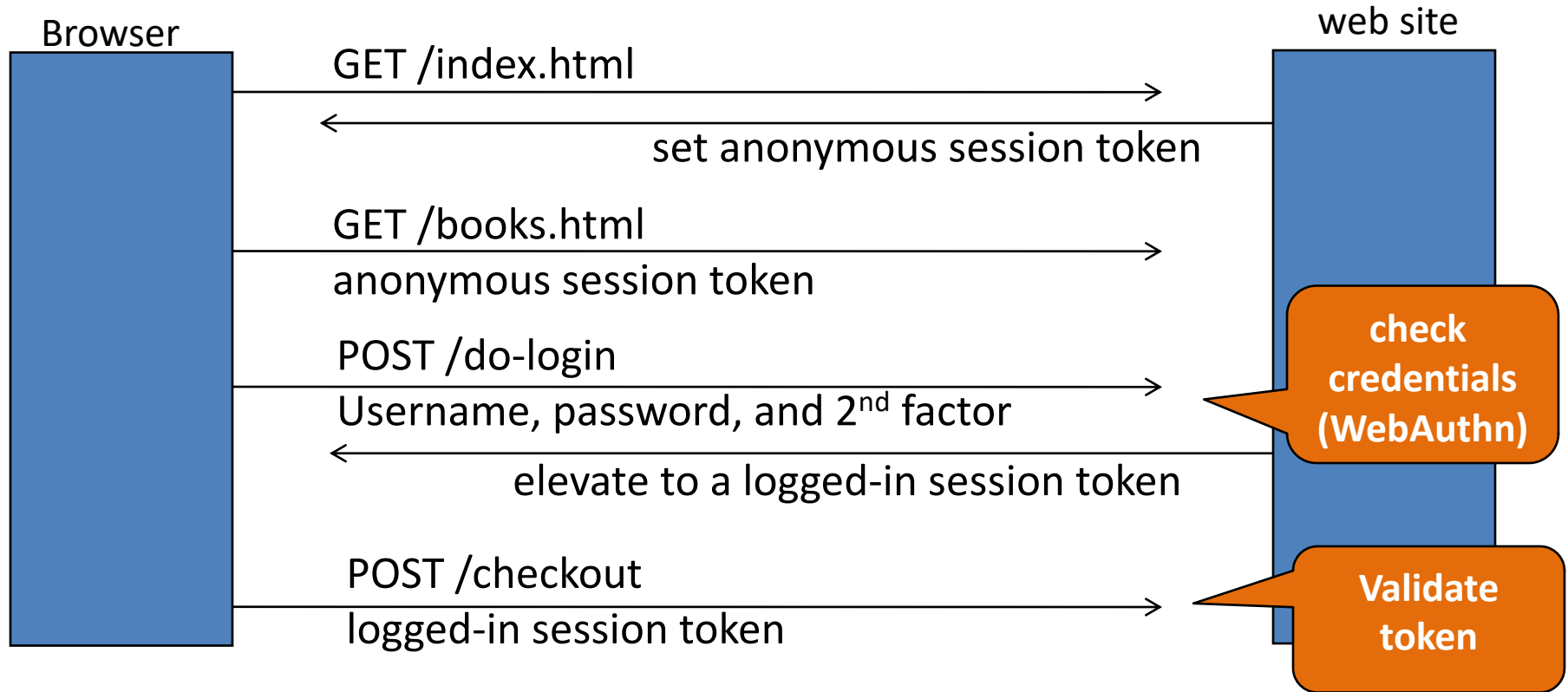
HTTP auth problems

Hardly used in commercial sites:

- User cannot log out other than by closing browser
 - What if user has multiple accounts?
multiple users on same machine?
- Site cannot customize password dialog
- Confusing dialog to users
- Easily spoofed

Do not use ...

Session tokens



Storing session tokens:

Lots of options (but none are perfect)

Browser cookie:

Set-Cookie: SessionToken=fduhye63sfdb

Embed in all URL links:

<https://site.com/checkout?SessionToken=kh7y3b>

In a hidden form field:

<input type="hidden" name="SessionToken" value="uydh735">

Storing session tokens: problems

Browser cookie: browser sends cookie with every request,
even when it should not (CSRF) [note: SameSite attribute]

Embed in all URL links: token leaks via HTTP **Referer** header
(or if user posts URL in a public blog)

In a hidden form field: does not work for long-lived sessions

Best answer: a combination of all of the above

Supported in most frameworks

PHP ex: **output_add_rewrite_var(name, value)**

The HTTP referer header

GET /wiki/John_Ousterhout HTTP/1.1

Host: en.wikipedia.org

Keep-Alive: 300

Connection: keep-alive

Referer: http://www.google.com/search?q=john+ousterhout&ie=utf-8&oe

Referer leaks URL session token to 3rd parties

Referer suppression:

- not sent when HTTPS site refers to an HTTP site
- in HTML5: ``

The Logout Process

Web sites must provide a logout function:

- Functionality: let user to login as different user
- Security: prevent others from abusing account

What happens during logout:

1. Delete SessionToken from client
2. Mark session token as expired on server

Problem: many web sites do (1) but not (2) !!

⇒ Especially risky in case of XSS vulnerability

Session hijacking

Session hijacking

Attacker waits for user to login

then attacker steals user's Session Token
and “hijacks” session

⇒ attacker can issue arbitrary requests on behalf of user

Example: **FireSheep**

Firefox extension: hijacks HTTP session tokens over WiFi

Solution: always send session tokens over HTTPS!

Beware: Predictable tokens

Example 1: counter

⇒ user logs in, gets counter value,
can view sessions of other users

Example 2: weak MAC. token = { userid, $\text{MAC}_k(\text{userid})$ }

- Weak MAC exposes k from few cookies.

Apache Tomcat: generateSessionId()

- Returns random session ID [server retrieves client state based on sess-id]

Session tokens must be unpredictable to attacker

To generate: use underlying framework (e.g. ASP, Tomcat, Rails)

Rails: token = SHA256(current time, random nonce)

Beware: Session token theft

Example 1: Cross Site Scripting (XSS) exploit

Example 2: use of HTTP after login over HTTPS

- Token sent over HTTP ... readable by network adversary
- Alternatively: Man-in-the-middle attack on HTTPS

Amplified by poor logout procedures:

- Logout must invalidate token on server

Mitigating SessionToken theft by binding SessionToken to client's computer

A common idea: embed machine specific data in SID

Client IP addr: makes it harder to use token at another machine

- But honest client may change IP addr during session
 - client will be logged out for no reason

Client user agent: weak defense against theft, but doesn't hurt.

TLS session id: same problem as IP address (and even worse)

Session fixation attacks

Suppose attacker can set the user's session token:

- For URL tokens, trick user into clicking on URL
- For cookie tokens, set using XSS exploits

Attack: (say, using URL tokens)

1. Attacker gets anonymous session token for site.com
2. Sends URL to user with attacker's session token
3. User clicks on URL and logs into site.com
 - this elevates attacker's token to logged-in token
4. Attacker uses elevated token to hijack user's session.

Session fixation: lesson

When elevating user from anonymous to logged-in:

always issue a new session token

(e.g. in PHP by calling `session_regenerate_id()` in PHP)

After login, token changes to value unknown to attacker

⇒ Attacker's token is not elevated.

Summary

- Session tokens are split across multiple client state mechanisms:
 - Cookies, hidden form fields, URL parameters
 - Cookies by themselves are insecure (CSRF, cookie overwrite)
 - Session tokens must be unpredictable and resist theft by network attacker
- Ensure logout and timeout invalidates session on server

THE END

Timing attacks

Observation: a web site response time depends on secret state

⇒ Attacker can observe response time to learn secret

Example: dictionary attack