Session Management and User Authentication

Dan Boneh
Same origin policy: “high level”

Review: Same Origin Policy (SOP) for DOM:
- Origin A can access origin B’s DOM if match on 
  \((\text{scheme}, \text{domain}, \text{port})\)

Today: Same Original Policy (SOP) for cookies:
- Generally speaking, based on:
  \(([[\text{scheme}]], \text{domain}, \text{path})\)

optional

scheme://domain:port/path?params
Setting/deleting cookies by server

- Delete cookie by setting “expires” to date in past
- Default scope is domain and path of setting URL

HTTP Header:
Set-cookie: NAME=VALUE ; domain = (when to send) ; path = (when to send) ; secure = (only send over SSL) ; expires = (when expires) ; HttpOnly

if expires=NULL: this session only
Scope setting rules  (write SOP)

domain: any domain-suffix of URL-hostname, except TLD

example: host = “login.site.com”

allowed domains
  login.site.com
  .site.com

disallowed domains
  user.site.com
  othersite.com
  .com

⇒ login.site.com can set cookies for all of .site.com
   but not for another site or TLD

   Problematic for sites like .stanford.edu

path: can be set to anything
Cookies are identified by (name, domain, path)

- **cookie 1**
  - name = `userid`
  - value = `test`
  - domain = `login.site.com`
  - path = `/`
  - secure

- **cookie 2**
  - name = `userid`
  - value = `test123`
  - domain = `.site.com`
  - path = `/`
  - secure

Both cookies stored in browser’s cookie jar; both are in scope of `login.site.com`
Reading cookies on server (read SOP)

Browser sends all cookies in URL scope:
- cookie-domain is domain-suffix of URL-domain, and
- cookie-path is prefix of URL-path, and
- [protocol=HTTPS if cookie is “secure”]

Goal: server only sees cookies in its scope
Examples

both set by login.site.com

cookie 1
name = userid
value = u1
domain = login.site.com
path = /
secure

cookie 2
name = userid
value = u2
domain = .site.com
path = /
non-secure

http://checkout.site.com/
http://login.site.com/
https://login.site.com/

cookie: userid=u2
cookie: userid=u2

cookie: userid=u1; userid=u2
(arbitrary order)
Client side read/write:  

- Setting a cookie in Javascript:
  ```javascript
  document.cookie = "name=value; expires=...;"
  ```

- Reading a cookie:
  ```javascript
  alert(document.cookie)
  ```
  prints string containing all cookies available for document (based on [protocol], domain, path)

- Deleting a cookie:
  ```javascript
  document.cookie = "name=; expires= Thu, 01-Jan-70"
  ```

document.cookie often used to customize page in Javascript
Javascript URL

```
javascript: alert(document.cookie)
```

Displays all cookies for current document
Viewing/deleting cookies in Browser UI
Cookie protocol problems

Server is blind:
- Does not see cookie attributes (e.g. secure)
- Does not see which domain set the cookie

Server only sees: Cookie: NAME=VALUE
Example 1: login server problems

- Alice logs in at login.site.com
  login.site.com sets session-id cookie for .site.com

- Alice visits evil.site.com
  overwrites .site.com session-id cookie with session-id of user “badguy”

- Alice visits cs155.site.com to submit homework.
  cs155.site.com thinks it is talking to “badguy”

Problem: cs155 expects session-id from login.site.com; cannot tell that session-id cookie was overwritten
Example 2: “secure” cookies are not secure

Alice logs in at https://www.google.com/accounts

Set-Cookie: LSID=EXPIRED; Domain=.google.com; Path=/; Expires=Mon, 01-Jan-1990 00:00:00 GMT
Set-Cookie: LSID=EXPIRED; Path=/; Expires=Mon, 01-Jan-1990 00:00:00 GMT
Set-Cookie: LSID=EXPIRED; Domain=www.google.com; Path=/accounts; Expires=Mon, 01-Jan-1990 00:00:00 GMT
Set-Cookie: LSID=ci:DOAAHsAAACn3h7GCpKUNxcr79Ce3BUCJtiual9a7e5oPvByTr
Set-Cookie: GAUSR=dabo123@gmail.com; Path=/accounts; Secure

Alice visits http://www.google.com (cleartext)
- Network attacker can inject into response
  
  Set-Cookie: LSID=badguy; secure

  and overwrite secure cookie

Problem: network attacker can re-write HTTPS cookies!
⇒ HTTPS cookie value cannot be trusted
Interaction with the DOM SOP

Cookie SOP: path separation

\texttt{x.com/A} does not see cookies of \texttt{x.com/B}

Not a security measure:

DOM SOP: \texttt{x.com/A} has access to DOM of \texttt{x.com/B}

\begin{verbatim}
<iframe src="x.com/B"></iframe>
alert(frames[0].document.cookie);
\end{verbatim}

Path separation is done for efficiency not security:

\texttt{x.com/A} is only sent the cookies it needs
Cookies have no integrity !!
Storing security data on browser?

- User can change and delete cookie values!!
  - Edit cookie file (FF3: 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 to problem with hidden fields

<INPUT TYPE="hidden" NAME=price VALUE="150">
Not so silly … (as of 2/2000)

- D3.COM Pty Ltd: ShopFactory 5.8
- @Retail Corporation: @Retail
- Adgrafix: Check It Out
- Baron Consulting Group: WebSite Tool
- ComCity Corporation: SalesCart
- Crested Butte Software: EasyCart
- Dansie.net: Dansie Shopping Cart
- Intelligent Vending Systems: Intellivend
- Make-a-Store: Make-a-Store OrderPage
- McMurtrey/Whitaker & Associates: Cart32 3.0
- pknutsen@nethut.no: CartMan 1.04
- Rich Media Technologies: JustAddCommerce 5.0
- SmartCart: SmartCart
- Web Express: Shoptron 1.2

Source: http://xforce.iss.net/xforce/xfdb/4621
Solution: cryptographic checksums

Goal: data integrity
Requires secret key $k$ unknown to browser

Generate tag: $T \leftarrow F(k, \text{value})$

Verify tag: $T = F(k, \text{value})$

"value" should also contain data to prevent cookie replay and swap
Example: .NET 2.0

- **System.Web.Configuration.MachineKey**
  - Secret web server key intended for cookie protection
  - Stored on all web servers in site

Creating an encrypted cookie with integrity:

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

Decrypting and validating an encrypted cookie:

- ```csharp
  HttpSecureCookie.Decode(cookie);
```
Session management
Sessions

- A sequence of requests and responses from one browser to one (or more) sites
  - Session can be long (Gmail - two weeks) 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"

Browsers send hashed password on all subsequent HTTP requests:

Authorization: Basic ZGFddfibzsdfgkjheczI1NXRleHQ=
HTTP auth problems

- Hardly used in commercial sites
  - User cannot log out other than by closing browser
    - What if user has multiple accounts?
    - What if multiple users on same computer?
  - Site cannot customize password dialog
  - Confusing dialog to users
  - Easily spoofed
  - Defeated using a TRACE HTTP request (on old browsers)
Session tokens

Browser

GET /index.html
set anonymous session token

GET /books.html
anonymous session token

POST /do-login
Username & password
elevate to a logged-in session token

POST /checkout
logged-in session token

Web Site

check credentials (later)

Validate token
Storing session tokens:  
Lots of options  (but none are perfect)

- Browser cookie:
  
  Set-Cookie: SessionToken=fduhye63sfdb

- Embedd in all URL links:
  
  https://site.com/checkout ? SessionToken=kh7y3b

- In a hidden form field:
  
  <input type="hidden" name="sessionId" value="kh7y3b">

- Window.name DOM property
Storing session tokens: problems

- Browser cookie:
  browser sends cookie with every request, even when it should not (CSRF)

- Embed in all URL links:
  token leaks via HTTP Referer header

- In a hidden form field: short sessions only

Best answer: a combination of all of the above.
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
SESSION HIJACKING

Attacker waits for user to login; then attacker obtains user’s Session Token and “hijacks” session.
1. Predictable tokens

Example: counter (Verizon Wireless)

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

Example: weak MAC (WSJ)

- token = \{userid, MAC_k(userid) \}
- Weak MAC exposes \( k \) from few cookies.

Session tokens must be unpredictable to attacker:
Use underlying framework.

Rails: token = MD5( current time, random nonce )
2. Cookie theft

Example 1: login over SSL, but subsequent HTTP
- What happens at wireless Café? (e.g. Firesheep)
- Other reasons why session token sent in the clear:
  - HTTPS/HTTP mixed content pages at site
  - Man-in-the-middle attacks on SSL

Example 2: Cross Site Scripting (XSS) exploits

Amplified by poor logout procedures:
- Logout must invalidate token on server
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.

- Once user logs in, token changes to value unknown to attacker.
  ⇒ Attacker’s token is not elevated.

- In the limit: assign new SessionToken after every request.
  - Revoke session if a replay is detected.
Generating session tokens

Goal: prevent hijacking and avoid fixation
Option 1: minimal client-side state

- SessionToken = [random unpredictable string]
  (no data embedded in token)

- Server stores all data associated to SessionToken: userid, login-status, login-time, etc.

- Can result in server overhead:
  - When multiple web servers at site, lots of database lookups to retrieve user state.
Option 2: lots of client-side state

- SessionToken:

  \[
  \text{SID} = [\text{userID}, \text{exp. time}, \text{data}]
  \]

  where \( \text{data} = (\text{capabilities, user data, ...}) \)

  \[
  \text{SessionToken} = \text{Enc-then-MAC (k, SID)}
  \]

  (as in CS255)

  \( k \): key known to all web servers in site.

- Server must still maintain some user state:
  - e.g. logout status (should be checked on every request)

- Note that nothing binds SID to client’s machine
Binding SessionToken to client’s computer; mitigating cookie theft

approach: embed machine specific data in SID

- **Client IP Address:**
  - Will make 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:**
  - A weak defense against theft, but doesn’t hurt.

- **SSL session key:**
  - Same problem as IP address (and even worse)
The Logout Process

Web sites provide a logout function:
- Functionality: let user to login as different user
- Security: prevent other 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)

Note: on a kiosk, logout can be disabled
⇒ enables session hijacking after logout.
User Authentication with passwords

OPTIONAL MATERIAL
Identification protocol

- **User P (prover)**
- **Server V (verifier)**

- Algorithm $G$
- Secret key $sk$
- Public key $vk$

- No key exchange

- Typically runs over a one-sided SSL channel

- $vk$ either public or secret
Basic Password Protocol (incorrect version)

- **PWD**: finite set of passwords

- Algorithm G (KeyGen):
  - choose rand pw in PWD.
  - output sk = vk = pw.

User P (prover) → sk → Server V (verifier)

sk

yes

iff sk = vk
Basic Password Protocol (incorrect version)

- **Problem:** VK must be kept secret
  - Compromise of server exposes all passwords
  - Never store passwords in the clear!

Password file on server:

<table>
<thead>
<tr>
<th>Alice</th>
<th>( pw_{alice} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>( pw_{bob} )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Basic Password Protocol: version 1

H: one-way hash function from PWD to X

“Given H(x) it is difficult to find y such that H(y)=H(x)”

User P (prover) \( \rightarrow \) sk \( \rightarrow \) Server V (verifier)

\( sk \) \( \rightarrow \) vk = H(sk)

yes iff H(sk)=vk

password file on server

<table>
<thead>
<tr>
<th>Alice</th>
<th>H(pw(_A))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>H(pw(_B))</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>
Weak Passwords and Dictionary Attacks

- People often choose passwords from a small set:
  - The 6 most common passwords (sample of $32 \times 10^6$ pwds): 123456, 12345, Password, iloveyou, princess, abc123
    ('123456' appeared 0.90% of the time)
  - 23% of users choose passwords in a dictionary of size 360,000,000

- Online dictionary attacks:
  - Defeated by doubling response time after every failure
  - Harder to block when attacker commands a bot-net
Offline Dictionary Attacks

- Suppose attacker obtains \( vk = H(pw) \) from server
  - **Offline** attack: hash all words in Dict until a word \( w \) is found such that \( H(w) = vk \)
  - Time \( O(|Dict|) \) per password

- Off the shelf tools
  - 2,000,000 guesses/sec
  - Scan through 360,000,000 guesses in few minutes
    - Will recover 23% of passwords
Password Crackers

Many tools for this
- John the ripper
- Cain and Abel
- Passware(Commercial)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Speed/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES</td>
<td>2 383 000</td>
</tr>
<tr>
<td>MD5</td>
<td>4 905 000</td>
</tr>
<tr>
<td>LanMan</td>
<td>12 114 000</td>
</tr>
</tbody>
</table>
Batch Offline Dictionary Attacks

- Suppose attacker steals pwd file F
  - Obtains hashed pwds for all users

- Batch dict. attack:
  - Build list L containing \((w, H(w))\) for all \(w \in \text{Dict}\)
  - Find intersection of L and F
- Total time: \(O(\mid \text{Dict} \mid + \mid F \mid)\)

- Much better than a dictionary attack on each password
Preventing Batch Dictionary Attacks

- **Public salt:**
  - When setting password, pick a random n-bit salt $S$
  - When verifying pw for A, test if $H(pw, S_A) = h_A$

- **Recommended salt length, $n = 64$ bits**
  - Pre-hashing dictionary does not help

- **Batch attack time is now:** $O(|Dict| \times |F|)$

<table>
<thead>
<tr>
<th>id</th>
<th>S</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>$S_A$</td>
<td>$H(pw_A, S_A)$</td>
</tr>
<tr>
<td>Bob</td>
<td>$S_B$</td>
<td>$H(pw_B, S_B)$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Further Defenses

**Slow hash function** \( H: \) (0.1 sec to hash pw)
- Example: \( H(pw) = \text{SHA1(SHA1( ... SHA1(pw) ...))} \)
- Unnoticeable to user, but makes offline dictionary attack harder

**Secret salts:**
- When setting pwd choose short random \( r \) (8 bits)
- When verifying pw for A, try all values of \( r_A \): 128 times slow down on average
- 256 times slow down for attacker
Case study: UNIX and Windows

- **UNIX:** 12-bit public salt
  - Hash function $H$:
    - Convert pw and salt and a DES key $k$
    - Iterate DES (or DES') 25 times:

  ![Diagram](image)

- **Windows:** NT and later use MD4
  - Outputs a 16 byte hash
  - No public or secret salts
Biometrics

✦ Examples:
  • Fingerprints, retina, facial recognition, ...
  • Benefit: hard to forget

✦ Problems:
  • Biometrics are not generally secret
  • Cannot be changed, unlike passwords

⇒ Primarily used as a second factor authentication
The Common Password Problem

- Users tend to use the same password at many sites
  - Password at a high security site can be exposed by a break-in at a low security site

- Standard solution:
  - Client side software that converts a common password $pw$ into a unique site password
    $$ pw' \leftarrow H(pw, \text{user-id}, \text{server-id}) $$
    $pw'$ is sent to server
Attempts at defeating key-loggers

Bank of Adelaide

Swivel PinSafe
One-time Passwords: security against eavesdropping
The SecurID system (secret vk, stateful)

Algorithm G: (setup)
- Choose random key \( k \leftarrow K \)
- Output \( sk = (k,0) \); \( vk = (k,0) \)

Identification:

- prover
  - \( sk = (k,0) \)
  - \( sk = (k,1) \)

- verifier
  - \( vk = (k,0) \)
  - \( vk = (k,1) \)

\[ r_0 \leftarrow F(k,0) \]
\[ r_1 \leftarrow F(k,1) \]

Yes iff \( r = F(k,0) \)
The SecurID system (secret vk, stateful)

- "Thm": if F is a secure PRF then protocol is secure against eavesdropping

- RSA SecurID uses a custom PRF:
  - 64 bit key → F → 6 digit output
  - 24 bit ctr

- Advancing state: sk ← (k, i+1)
  - Time based: every 60 seconds
  - User action: every button press

- Both systems allow for skew in the counter value
The S/Key system  (public vk, stateful)

- **Notation:** \( H^{(n)}(x) = H(H(...H(x)...)) \) for \( n \) times

- **Algorithm G: (setup)**
  - Choose random key \( k \leftarrow K \)
  - Output \( sk = (k,n) ; \quad vk = H^{(n+1)}(k) \)

- **Identification:**

\[ k \xrightarrow{H(k)} H^{(n-2)}(k) \xrightarrow{H^{(n-1)}(k)} H^{(n)}(k) \xrightarrow{H^{(n+1)}(k)} \]

pwd #1(pwd #2(pwd #3(pwd #4))))
The S/Key system (public vk, stateful)

Identification (in detail):

- Prover (sk=(k,i)): send \( t \leftarrow H^{(i)}(k) \); set \( sk \leftarrow (k, i-1) \)

- Verifier (vk=H^{(i+1)}(k)): if \( H(t)=vk \) then \( vk \leftarrow t \), output “yes”

Notes: vk can be made public; but need to generate new sk after n logins (\( n \approx 10^6 \))

“Thm”: S/Key

\( n \) is secure against eavesdropping (public vk) provided H is one-way on n-iterates
SecurID vs. S/Key

S/Key:
- **public** vk, **limited** number of auths
- often implemented using pencil and paper

SecurID:
- **secret** vk, **unlimited** number of auths
- often implemented using secure token