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

  (scheme, domain, port)

Today: Same Original Policy (SOP) for cookies:

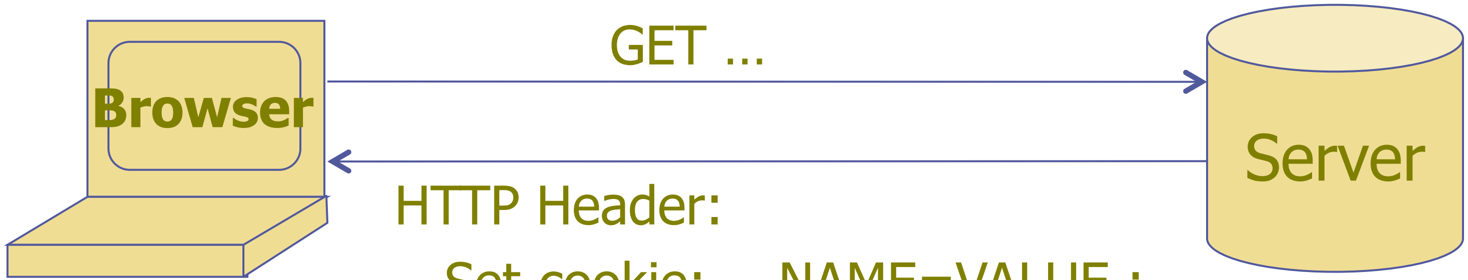
- Generally speaking, based on:

  ([scheme], domain, *path*)

optional

scheme://domain:port/path?params

Setting/deleting cookies by server



Set-cookie: NAME=VALUE ;

domain = (when to send) ;
path = (when to send)

scope

secure = (only send over SSL);
expires = (when expires) ;
HttpOnly

if expires=NULL:
this session only

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

Scope setting rules (write SOP)

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

example: host = "login.site.com"

allowed domains	disallowed domains
login.site.com	**user.site.com**
.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)

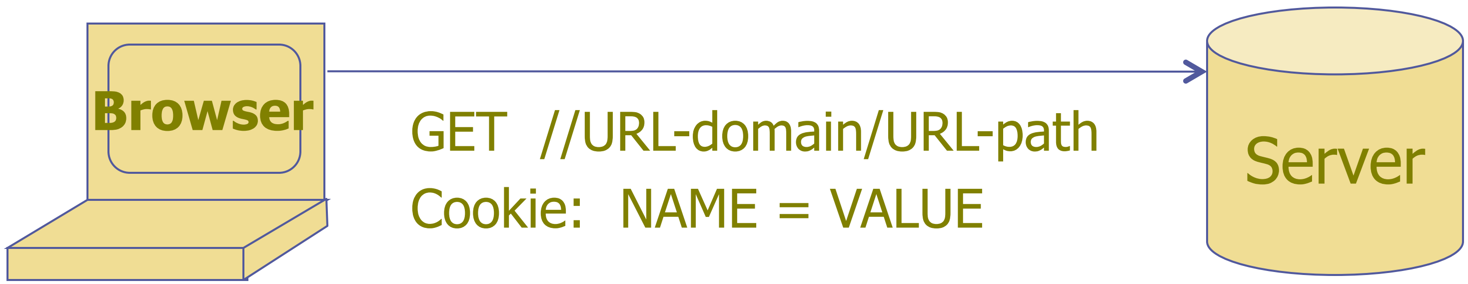
<u>cookie 1</u>
name = **userid**
value = test
domain = **login.site.com**
path = **/**
secure

<u>cookie 2</u>
name = **userid**
value = test123
domain = **.site.com**
path = **/**
secure

distinct cookies

- 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	cookie 2
name = **userid**	name = **userid**
value = u1	value = u2
domain = **login.site.com**	domain = **.site.com**
path = **/**	path = **/**
secure	non-secure

http://checkout.site.com/ cookie: userid=u2

http://login.site.com/ cookie: userid=u2

https://login.site.com/ **cookie: userid=u1; userid=u2**

(arbitrary order)

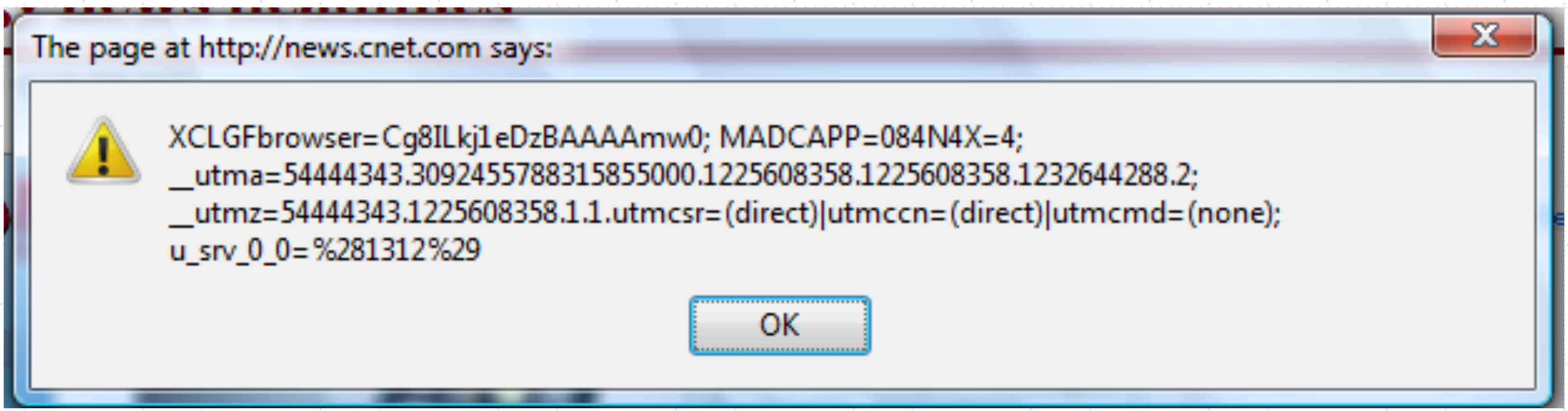
Client side read/write: document.cookie

- Setting a cookie in Javascript:
  document.cookie = “name=value; expires=…; ”
- Reading a cookie: alert(document.cookie)
  prints string containing all cookies available for document (based on [protocol], domain, path)
- Deleting a cookie:
  document.cookie = “name=; expires= Thu, 01-Jan-70”

document.cookie often used to customize page in Javascript

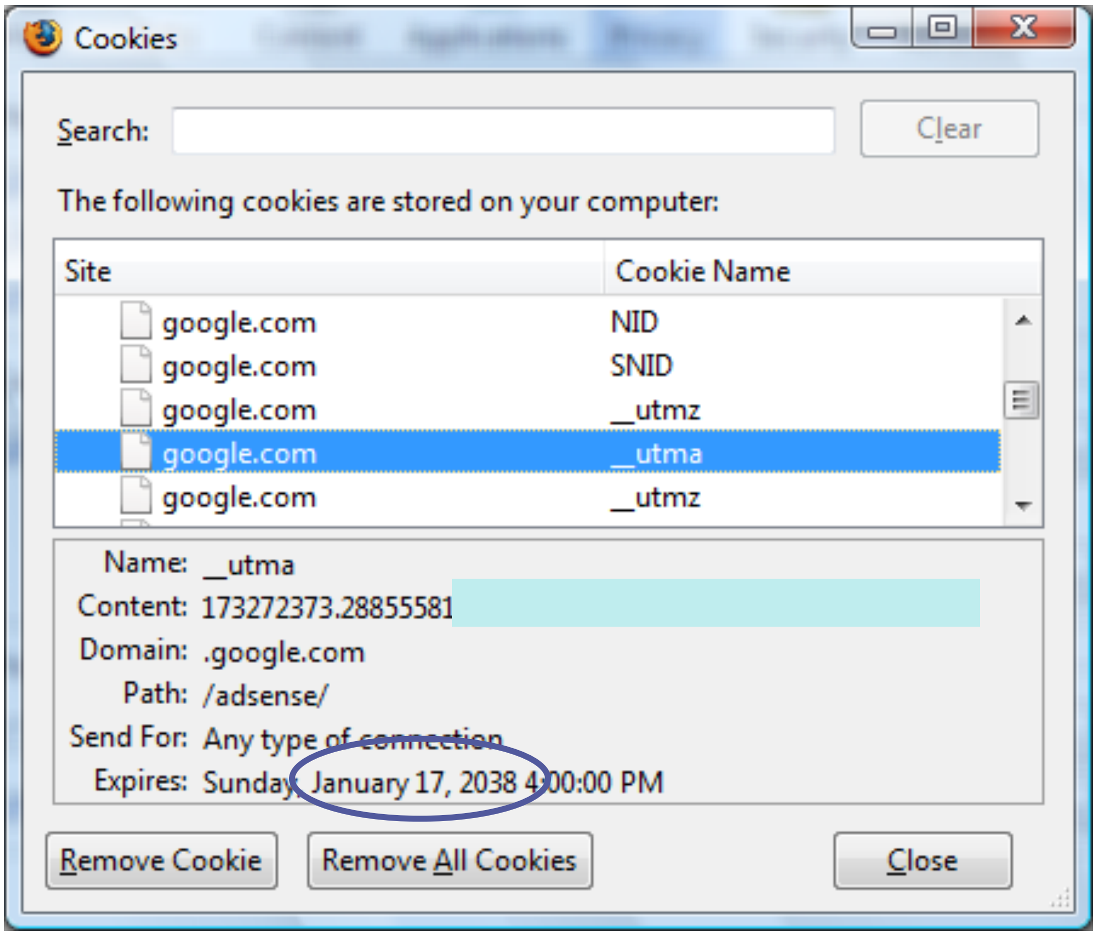
Javascript URL

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=cl:DQAAAHsAAACn3h7GCpKUNxckr79Ce3BUCJtlual9a7e5oPvByTr
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

x.com/A does not see cookies of **x.com/B**

Not a security measure:

DOM SOP: **x.com/A** has access to DOM of **x.com/B**

```
<iframe src=“x.com/B"></iframe>
alert(frames[0].document.cookie);
```

Path separation is done for efficiency not security:

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)

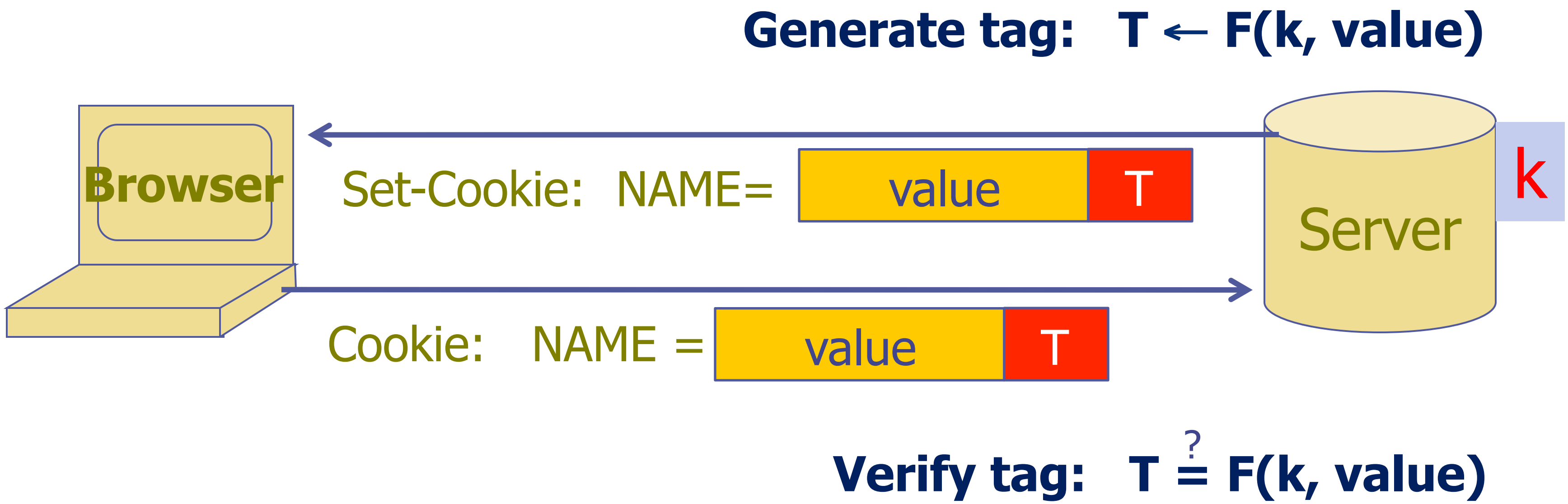
- D3.COM Pty Ltd: ShopFactory 5.8
- @Retail Corporation: @Retail
- Adgrafix: Check It Out
- Baron Consulting Group: WebSite Tool
- ComCity Corporation: SalesCart
- Crested Butte Software: EasyCart
- Dansie.net: Dansie Shopping Cart
- Intelligent Vending Systems: Intellivend
- Make-a-Store: Make-a-Store 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



"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:

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

Decrypting and validating an encrypted cookie:

- **HttpSecureCookie.Decode** (cookie);

Session managemnt

Sessions

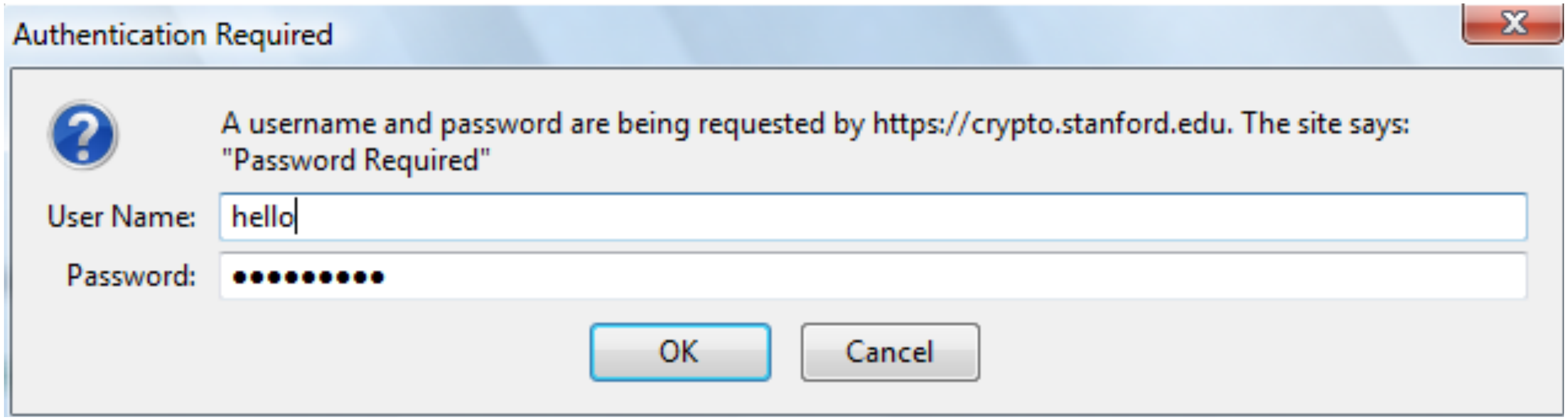
- A sequence of requests and responses from one browser to one (or more) sites
  - Session can be long (Gmail - two weeks) or short
  - without session mgmt:
    users would have to constantly re-authenticate
- Session mgmt:
  - Authorize user once;
  - All subsequent requests are tied to user

Pre-history: HTTP auth

HTTP request: GET /index.html

HTTP response contains:

WWW-Authenticate: Basic realm="Password Required“



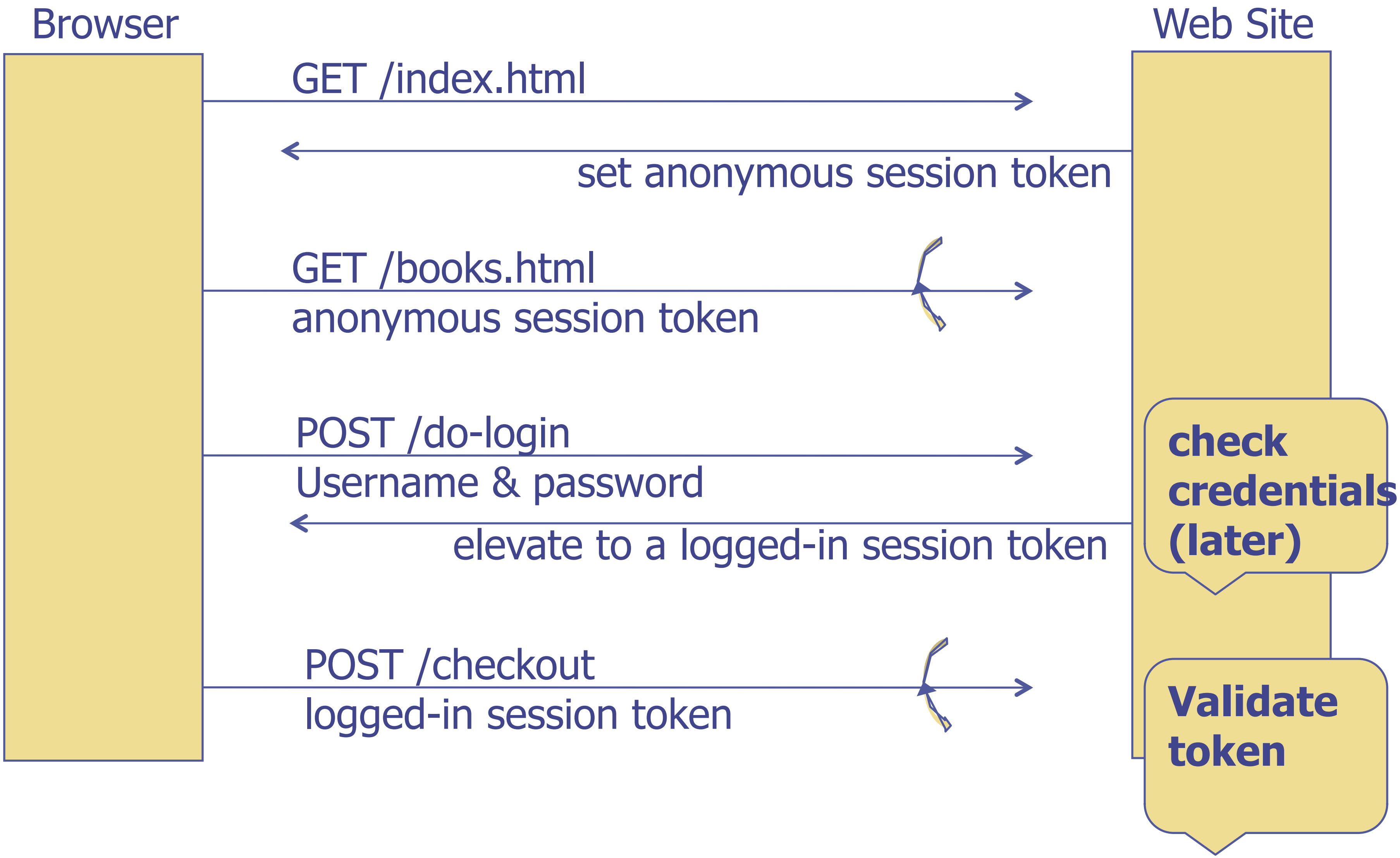
Browsers sends hashed password on all subsequent HTTP requests:

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



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, $\mathbf{MAC_k}$(userid) }**
  - Weak MAC exposes **k** from few cookies.

Session tokens must be unpredicatble 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

- <u>Attack</u>: (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:

  SID = **[userID, exp. time, data]**

  where data = (capabilities, user data, ...)

  SessionToken = **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) !!

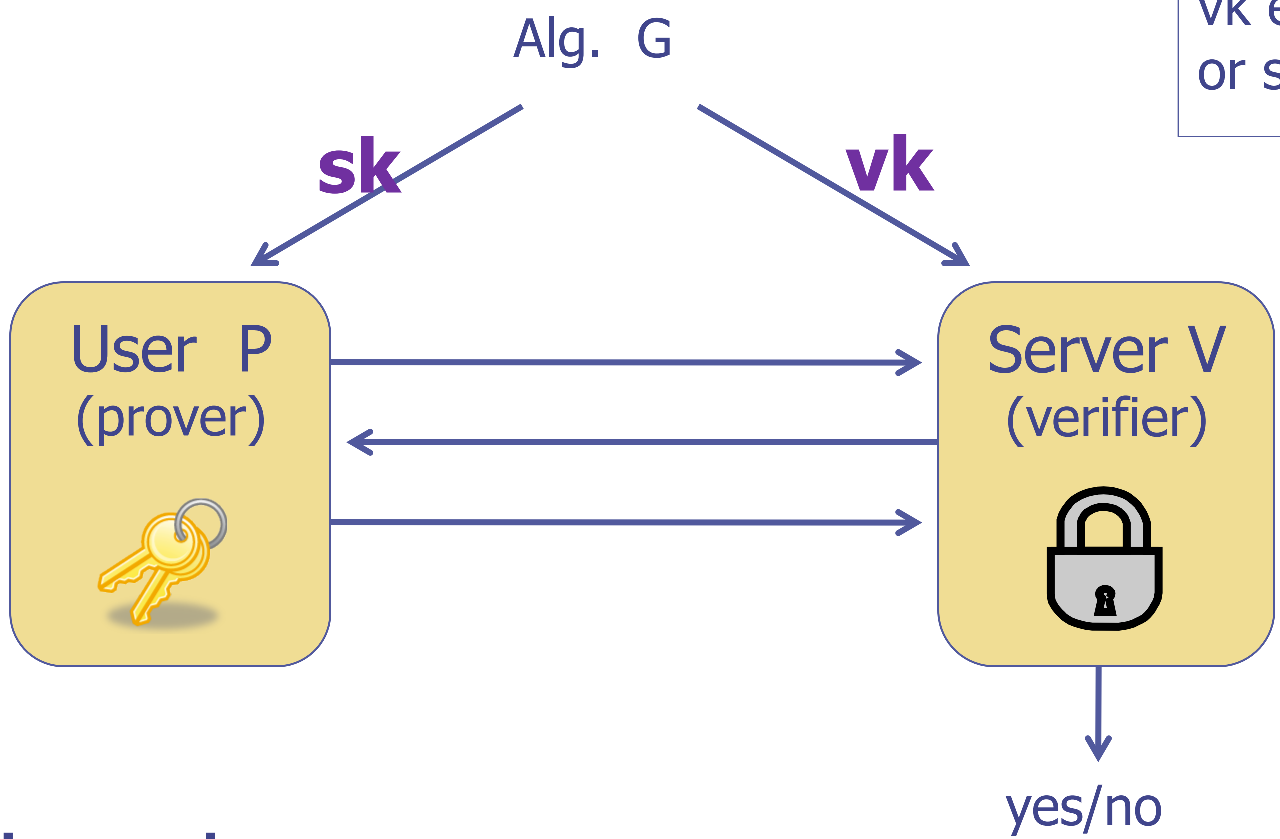
Note: on a kiosk, logout can be disabled

$\Rightarrow$ enables session hijacking after logout.

User Authentication with passwords

OPTIONAL MATERIAL

Identification protocol



vk either public or secret

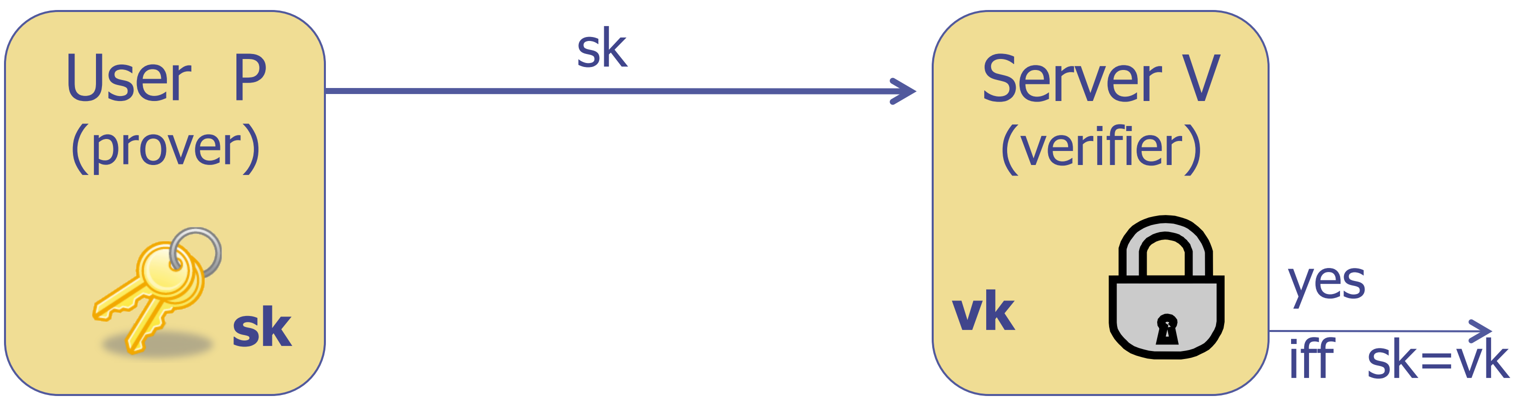
no key exchange

Typically runs over a one-sided SSL channel

Basic Password Protocol (incorrect version)

- **PWD**: finite set of passwords

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



Basic Password Protocol (incorrect version)

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

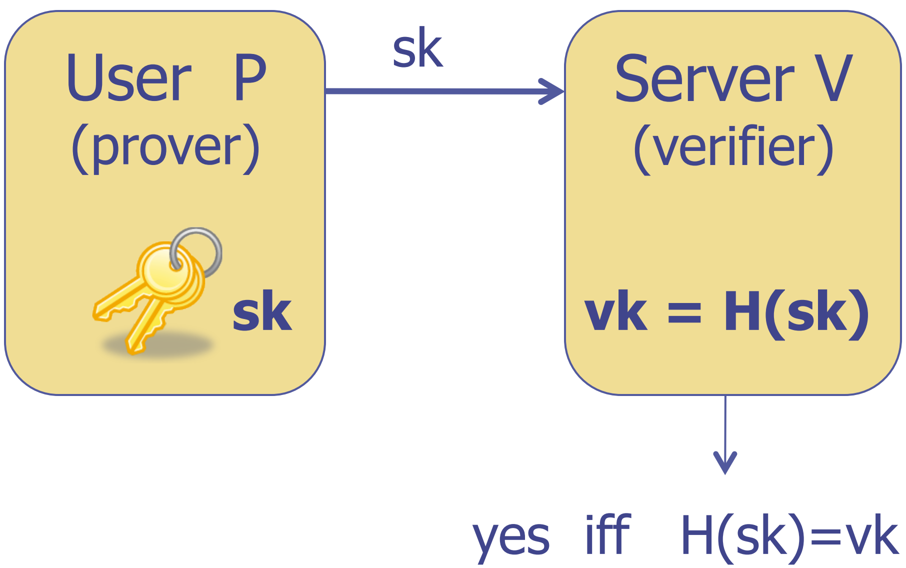
password file on server

Alice	pw_{alice}
Bob	pw_{bob}
…	…

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)"



password file on server

Alice	$H(pw_A)$
Bob	$H(pw_B)$
…	…

Weak Passwords and Dictionary Attacks

- People often choose passwords from a small set:
  - The 6 most common passwords (sample of 32×10^6 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

Algorithm	Speed/sec
DES	2 383 000
MD5	4 905 000
LanMan	12 114 000

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

Batch Offline Dictionary Attacks

Alice	$H(pw_A)$
Bob	$H(pw_B)$
…	…

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

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

- Much better than a dictionary attack on each password

Preventing Batch Dictionary Attacks

id	S	h
Alice	$\mathbf{S_A}$	$H(pw_A, \mathbf{S_A})$
Bob	$\mathbf{S_B}$	$H(pw_B, \mathbf{S_B})$
…	…	…

- Public salt:
  - When setting password, pick a random n-bit salt S
  - When verifying pw for A, test if $\mathbf{H(pw, S_A) = h_A}$
- Recommended salt length, $n = 64$ bits
  - Pre-hashing dictionary does not help
- Batch attack time is now: $O(\ |\mathbf{Dict}| \times |\mathbf{F}|\ )$

Further Defenses

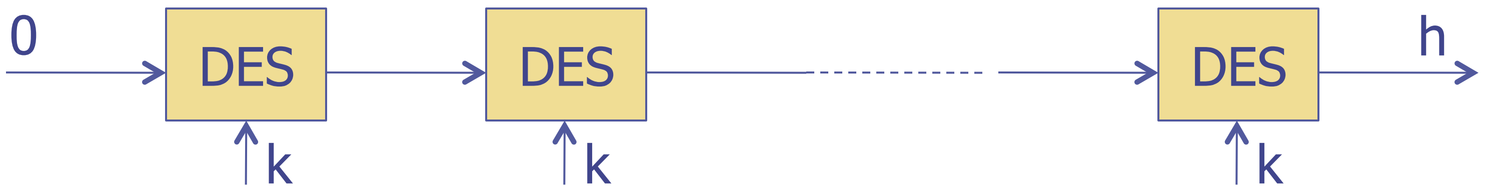
- **Slow hash function** H: (0.1 sec to hash pw)
  - Example: H(pw) = SHA1(SHA1(… SHA1(pw) …))
  - Unnoticeable to user, but makes offline dictionary attack harder

Alice	**S_A**	H(pw_A , S_A , $\mathbf{r_A}$)
Bob	**S_B**	H(pw_B , S_B , $\mathbf{r_B}$)
…	…	…

- **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:



- **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

- $\Rightarrow$ 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)

vasco

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

- Identification:

prover

sk = (k,0)

↓

sk = (k,1)

⋮

$r_0 \leftarrow F(k,0)$ →

$r_1 \leftarrow F(k,1)$ →

verifier

vk = (k,0)

↓

vk = (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

vasco

- RSA SecurID uses a custom PRF:

64 bit key → F → 6 digit output
24 bit ctr → F

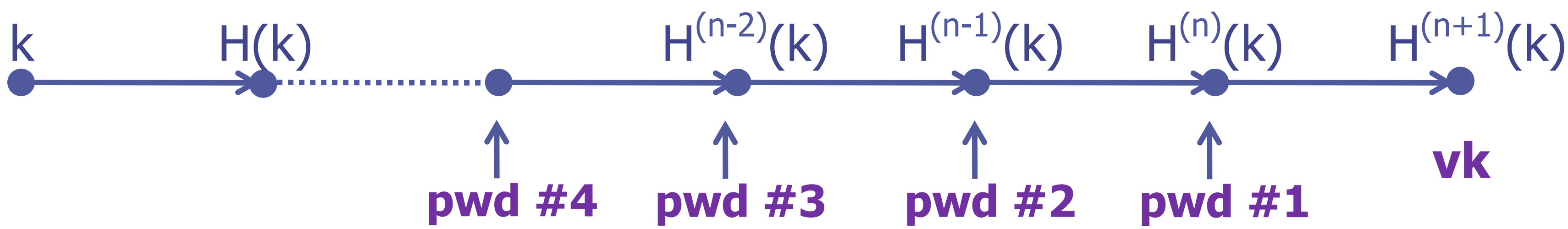
- Advancing state: sk ← (k, i+1)
  - Time based: every 60 seconds
  - User action: every button press
- Both systems allow for skew in the counter value

The S/Key system (public vk, stateful)

- Notation: $H^{(n)}(x) = \underbrace{H(H(\ldots H(x)\ldots))}_{\text{n times}}$

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

- <u>Identification</u>:



The S/Key system (public vk, stateful)

- Identification (in detail):
  - Prover (**sk=(k,i)**): send $\mathbf{t} \leftarrow \mathbf{H^{(i)}(k)}$; set $\mathbf{sk} \leftarrow \mathbf{(k, i-1)}$
  - Verifier($\mathbf{vk=H^{(i+1)}(k)}$): if $\mathbf{H(t)=vk}$ then $\mathbf{vk \leftarrow t}$, 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