CS 526: Information Security

Web Vulnerabilities

Security/Privacy Issues in Web Browsers

- How to securely run mobile code?
- How to provide access control?
- How to deal with privacy risks?

Background: HTTP

- HTTP is a stateless protocol
  - Servers are not required to retain information about users between requests
- Web applications may need to store state
  - E.g., user's progress when shopping (cart, etc)
- Many ways for state info to be stored
  - Server side sessions (most secure)
  - Client-side cookies (better if encrypted)
  - URL encoded parameters (many drawbacks)

Cookies

- Small files used by web applications to store information about state
- Purpose includes
  - Authentication (e.g., contains session ID)
  - Tracking
  - Maintaining session
  - Maintaining user information
- Cookie same-origin policy (enforced by browser)
  - Only the origin that created the cookie can access it
  - E.g., your cookie for shopping web site is not accessible to the social web site you're simultaneously visiting

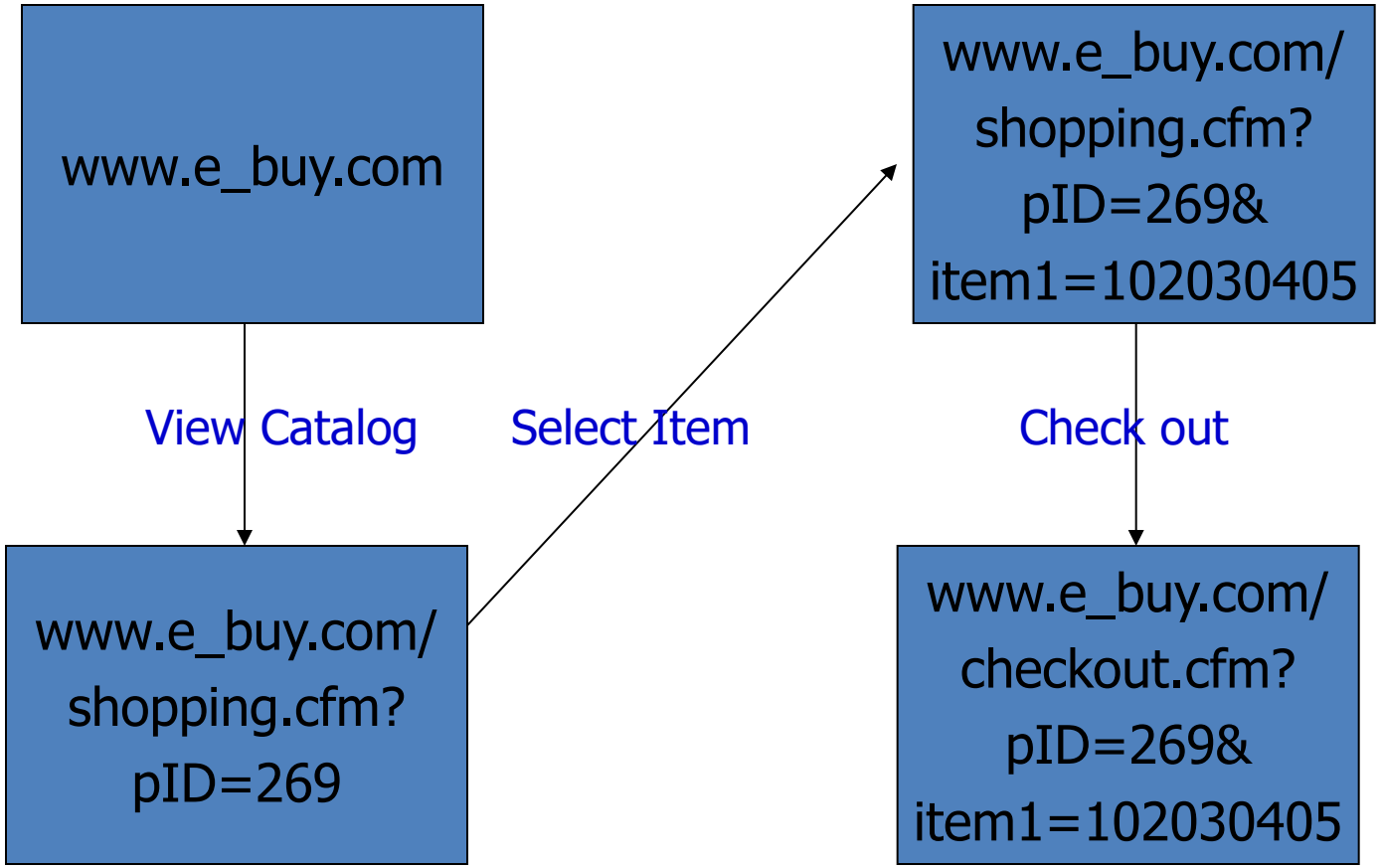
Cookies (cont'd)

- Temporary cookies
  - Stored until you logout or quit your browser
  - Or until session times out
- Persistent cookies
  - Remain until deleted (or at expiration date)
  - If user chooses "Remember me", "Stay Logged In"
- Third-party cookies
  - Originate on a web site other than the one that provided the page you visited
  - Can configure some browsers to disallow them

Example of Third-Party Cookies

- Alice gets a page from www.merchant.com
  - Contains <img src=http://doubleclick.com/advt.gif>
  - Image fetched from DoubleClick.com (DoubleClick knows the page Alice was looking at)
- DoubleClick sends back an ad tailored to Alice
  - Stores a cookie that identifies Alice to DoubleClick
- Next time Alice gets page with a DoubleClick img
  - Alice's DoubleClick cookie is sent back to DoubleClick
  - DoubleClick server can maintain list of sites she viewed
  - Send back targeted advertising (and a new cookie)
- Sites cooperate with DoubleClick
  - E.g., pass information to DoubleClick in URL

Example of Session State in URL



Changes undergone by URL as a shopping session progresses

Security Risks Posed by Mobile Code

- Compromise host on which the browser runs
  - Write to file system
  - Interfere with other processes in the browser's environment
- Steal information from host or browser
  - Read host file system
  - Read session authentication information
  - Read info associated with other browser windows
  - Send stolen info to attacker (e.g., encoded in URL as a "request")

Mitigating Mobile Code Risks

- Sandboxing
  - Code executed in browser has only restricted access to OS, network
- Isolation: Same-origin principle
  - Only the site that stores some information in the browser may later read or modify that information (or depend on it in any way)
- Establish trust in the code
  - Digitally signed code
  - Use of automated reasoning about code safety

Mechanisms for Effective Sandboxing

- Examine code before executing it
  - Code verifier performs critical tests on code (uses automated theorem-proving techniques)
- Interpret code and trap risky operations
  - Interpreter does run-time tests
- Security manager applies local access policy
  - Same-origin policy
  - Signed code

Same Origin Policy for Code

- Browsers restrict what scripts can access
  - Necessary to prevent a www.buystuff.com script from accessing open page www.mybank.com info
- Allow a script running on a web page to access the content of another web page only if both pages have the *same origin*
- Origin = domain, protocol, port
  - Example: www.buystuff.com, http, 80

Examples of Same Origin Check

URL of Target Window	Result of Same Origin Check with www.example.com	Reason
http://www.example.com/ index.html	Passes	Same domain and protocol
http://www.example.com/ other1/other2/index.html	Passes	Same domain and protocol
http://www.example.com :8080/dir/page.html	Does not pass	Different port
http://www2.example.co m/ dir/page.html	Does not pass	Different server
http://otherdomain.com/	Does not pass	Different domain
ftp://www.example.com/	Does not pass	Different protocol

Same-origin check applies to access to window object of other frames, etc.

Controlled Bypassing of Same Origin

- There are many ways for pages from different origins to communicate in a controlled way
- HTML <script> element
  - <script src> loads script code from elsewhere on the Internet
- Lowering two pages' domain property values to a common subdomain
  - foo1.buystuff.com, foo2.buystuff.com => buystuff.com
- Cross-origin policy files
  - Target web site specifies who can access its data

Risks Associated with Cookies

- Cookies maintain record of your browsing habits
  - Store information as set of name/value pairs
  - May include *any* information a site has on you
  - Track your activity from multiple visits to site
- Sites can share this information
  - E.g., DoubleClick
- Cookies vulnerable to attack
  - Including authentication cookies

DNS Rebinding Attack

- Malicious web page causes visitors to run scripts that attack other machines on network
  - Based on abuse of Domain Name System (makes possible subversion of same origin policy)
- Attacker registers a domain and delegates it to a DNS server that the attacker controls
  - That DNS server responds with a very short time-to-live record (so the record does not get cached – this is important for the attack to succeed)

DNS Rebinding (cont'd)

1. Alice browses to the attacker's web site, runs malicious client-side script (attacker as origin)
2. Malicious script makes a *new* DNS request for attacker site, to which the response is the IP address of the target of the attack: Tammy
   - Alice's browser has been fooled into taking Tammy's IP address to be that of the origin of the script: It allows the script to attack Tammy (more damage is done if Tammy trusts Alice)

Cross-Site Scripting (XSS)

- Victim Bob uses shopping web site
  - www.buystuff.com allows user to search, echoes back user's input to the search along with the outcome of the search (most search engines do this)
- Vulnerability: What is echoed back to Bob is rendered as HTML (rather than merely as text)
  - If Bob's search string contains a malicious script, it will run in Bob's browser with buystuff as "origin"
  - Malicious script can now access Bob's info of origin buystuff (including Bob's authentication cookie)

Example of XSS Vulnerability

- Social web site provides Bob with a search facility
- Bob types the following search string:

```
<i>eggplant ice cream recipes</i>
```

- Bob expects the response to be:

```
“Your search for ‘<i>eggplant ice cream recipes</i>’ has found 0 results”
```

but instead the response is:

“Your search for ‘*eggplant ice cream recipes*’ has found 0 results”

Example of XSS Vulnerability (cont'd)

- Instead of the harmless italicization, the search string could have contained a harmful script
  - The harmful string would have been echoed back to Bob rendered as HTML, and the harmful script would have run on Bob's browser with the social web site as origin (therefore would have had access to sensitive information like Bob's authentication cookie for the social web site)
- But why is this dangerous?
  - "Surely Bob would not launch such an attack on himself !"

Why it is Dangerous

- Because Bob can be tricked into clicking on a link that triggers the attack, for example:
  - Attacker creates a link with the harmful string:

    http://www.buystuff.com/search?searchTerm=...
  - Attacker tricks Bob into clicking on such a link, causing Bob's browser to run the malicious script
  - The malicious script runs with buystuff as origin, sends to attacker Bob's authentication cookie for the buystuff session

Why it is Dangerous (cont'd)

- Malicious script can send to attacker stolen info by (e.g.) hiding it in URL of request to attacker:

  http://www.attacker.com/… stolen info ...

- Theft of authentication cookie allows attacker to impersonate the victim
  - Change password, steal money, …
- XSS vulnerabilities are very common
  - Estimates of % servers affected range from 40 to 70

XSS Attack: Getting Victim to Click

- How is Bob tricked into clicking a poisoned link?
  - Social engineering (e.g., link is in email, pop-up)
  - URL shortening (e.g., TinyURL)
- Social engineering
  - "You've won! Click here to claim your prize!"
  - "Your child Alice's school principal called with an urgent message, click here to hear it"
- URL shortening: Use short link to redirect to long
  - Otherwise hovering over hyperlink with mouse, prior to clicking, may reveal the attack to the victim

XSS Attacks: Reflected, Stored

- XSS as just described is called *reflected* XSS
  - Because it is based on echo'ing ("reflecting") the user's own input immediately back to that user
  - Only that user is affected by the attack
- In *stored* XSS, the malicious script resides permanently on the vulnerable web site
  - Through malicious user input that is stored there
  - E.g., posted as a user review of a product or service
- Both cases caused by lack of input validation

Stored XSS Attack Example

- Malicious user posts, on buystuff web site, a product review that contains a malicious script
- Victim Bob accesses and reads the review
  - This immediately causes the malicious script to run on Bob's browser, with buystuff as its origin, hence it can do the same damage as in the case of a reflected XSS
- No need for social engineering
  - No need for spam email, or popups: Attack succeeds against anyone who accesses and reads the review

Stored XSS Attack (cont'd)

- The attack exploits the trust that users like Bob have in the vulnerable web site
  - Many web sites are not worthy of such trust
- The attack occurs because the vulnerable web site does not properly sanitize the HTML content posted by its subscribers
  - Bob could not have done much to prevent attack
  - It is not the browser's fault either

HTML Injection

- What is injected need not be a script that sends the attacker an authentication cookie
- Instead of injecting a script, inject HTML
  - E.g., a frame or form that points to the attacker's server, says "Session timed out, please re-login"
  - Steals passwords (which attackers prefer to authentication cookies that expire quickly)
  - Like phishing, but without the trouble of creating a fake web site (the attack uses the real web site)

Defending against XSS

- Does using htpps protect from XSS attacks?
  - No
- Preventing all user input?
  - No, too drastic (users need to search, post, …)
  - Many sites depend heavily on user contributions
- Better: Restrict what users can input
  - No scripts
  - Only “safe” HTML

Defense: Only Plain Text Input

- Cannot prevent users from including in their inputs “<“, “>”, and other special characters
  - Legitimate user input may involve using them
- Instead, process user input to encode them
  - So they are properly displayed by browser but lose their special meaning and become harmless
  - E.g., “<“ is encoded as “&lt”, “>” as “&gt”, …etc
  - Attack script is displayed to user as harmless text

Defense: Sanitizing Input

- Social web sites need to let user contributions include highlighted text, links, …
  - Such user inputs need to be sanitized
  - Sanitize HTML = filter out “bad”, keep “good”
- Sanitizing is tricky
  - Use keyword “script” to guide the filtering?

    <scr~~<script>wifwifwoof</script>~~ipt>
  - Iterate until no more? (still possible to attack)

Sanitizing Input (cont'd)

- Possible to attack without using string "script"
- Attacker can use an event handler
  - E.g., "onmouseover"
  - Moving the mouse over a word can trigger attack
  - Strip out all event handlers?
- Sanitizing HTML is tricky and hazardous
  - Better: Restrict input to weaker markup languages that allow italic, bold, ... (*but nothing dangerous*)

Extra Precaution: Hide the Cookie

- Make authentication cookie invisible to client-side scripts
  - So that any malicious script running on the client is unable to send that cookie to the attacker
- In most applications, only the server side code needs to read and write that cookie
  - Client scripts don't need to access it anyway
- Cookie-hiding is supported by all browsers
  - Apply the HttpOnly property to the cookie

Content Security Policy (CSP)

- Browser ignores all inline scripts (including event handlers)
  - Script injected by attacker is ignored by browser
- All scripts are pulled in from separate URLs
  - CSP allows specification of which URLs can be used for this (a whitelisting approach)
- CSP is very configurable
  - But difficult to retrofit existing applications that rely on the now-ignored functionality

Cross-Site Request Forgery (CSRF)

1. Victim browses to a malicious website (possibly upon following a link in an email)
2. Malicious website entices victim to submit a form with the action pointing to (e.g.) victim's online bank
   - Victim's browser automatically sends to the bank any existing cookies for the online bank
   - If the victim is already authenticated to the bank's server, the authentication cookies would be sent and the form submission would be accepted (and do much damage – change the victim's password, steal $, …)

Cross-Site Request Forgery (cont'd)

- The attack exploits the online bank's trust in the victim's browser
  - The trust exists because the victim is already logged in
- Bank's audit logs would not contain evidence of the attack, because nothing is logged about the attacker
  - The attack appears to be coming from the victim's own IP address

Simple DIY Precautions Against CSRF

- When you visit your online bank, do nothing else with your browser
- Make sure you explicitly logout when done
- Beware of features like “Remember Me” and “Stay Logged In”
  - They set the authentication cookie as persistent (it is no longer just a temporary session cookie, it stays even if you quit the browser and reboot)

Defending against CSRF

- Browser denies all cross-site requests?
  - Probably not – it would interfere with the normal use of many web sites
- Browser omits all authentication info from cross-site requests sent by non-whitelisted sites?
  - Yes, but attacker might use a zombie on the whitelist
- Server checks “referer” header?
  - Ineffective: Referer header is not sent for all requests (e.g., clicking email), and it can easily be spoofed

Defending against CSRF (cont'd)

- Cookieless authentication?
  - URL rewritten to contain authentication token
  - Attacker does not know that token, so the server never receives it (and malicious request fails)
  - A dangerous cure, better avoided (it is not safe to use URLs to pass around authentication info)
- Server asks user to re-authenticate?
  - Yes, for sensitive requests (like money transfers)
  - Password is needed, defeats CSRF

Defending against CSRF (cont'd)

- Embedding additional data into requests, that allows the web application to detect forgeries
- Example 1: Synchronizer Token Pattern (STP)
  - Embed a token with each request, that is verified by server; next token = hash of previous token
  - Attacker is unable to place the correct token in a forged request
  - Increases load on server, that must verify the token for every request
  - Forces sequencing of events, which harms usability

Defending against CSRF (cont'd)

- Example 2: Cookie-to-Header token
  - On login, web app sets a cookie containing a random token for that whole session (no forced sequencing)
  - The token is included with every request, in a hidden form field (script running on client, within the same origin as the token, can include the token in requests)
  - The server expects to see that token with all requests for this session
  - Attacker is unable to place correct token in a forged request (rogue script is unable to read the cookie, because of same origin policy)

Defending against CSRF (cont'd)

- Example 3: Double-submit
  - Submit with every request a hash of the session ID in a hidden form field
  - Called "double submit" because authentication cookie with session ID is also sent (automatically)
  - Server expects both authentication cookie and the mirrored hash of the session ID, checks for match
  - Attacker is unable to place correct info in a forged request (so the server rejects forged requests)

Clickjacking attacks

- Trick victim into clicking on a button Y
  - Victim thought they were clicking on something else, would never have knowingly clicked on Y
- Examples of Y
  - Runs a malicious script (as discussed in previous lectures)
  - Causes a “one click buy” at Amazon
  - Causes a “like” at Facebook
  - Causes a “Follow” on Twitter
  - … etc
  - Requests look legitimate to servers at the legitimate sites

Clickjacking (cont'd)

- Y is usually in a frame of very small size (just a few pixels) and low opacity (= transparent and invisible to the victim)
- Example 1: Attacker makes use of a visible fake "button" X, a lure that is likely to appeal to visitors to the malicious page (X can say "play", "download", …)
  - The real button Y is invisible and also at position X
- Example 2: Y follows victim's mouse
  - No need for a lure X
  - Attack succeeds no matter where the user clicks (even an accidental click works for the attacker)
  - More general than Example 1 (works for all merchants)

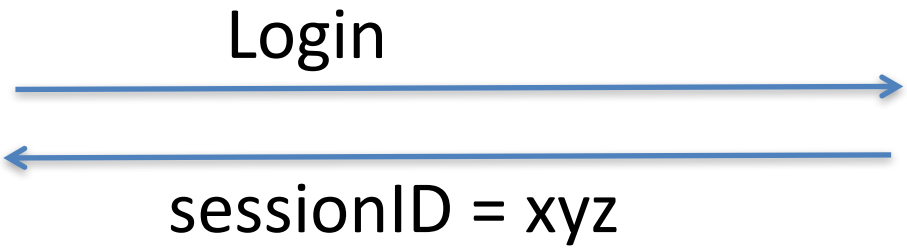
Forged Login Request

- Attacker forges request to log the victim into a target website *using the attacker's credentials*
  - Attack succeeds if user doesn’t realize they’re using someone else’s account
- Attacker later logs into the same site using own credentials
  - Views private information created by victim, saved activity history, etc
  - Can use the info for carrying out other attacks on victim

Session Fixation Attack

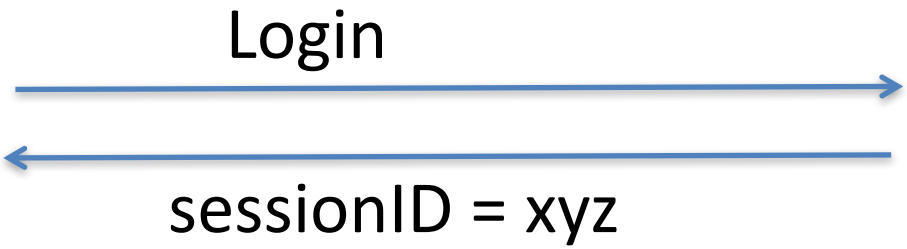
- Vulnerability: Server does not automatically assign a new session ID to a user who logs in with password
  - It allows user who logs in, the optional re-use of an existing session ID previously issued by the server
  - A user who does not specify an existing session ID is assigned a random new one by the server
- Vulnerability allows attackers to impersonate users
- To eliminate the vulnerability, eliminate the re-use
  - Every time a user authenticates with password, the server automatically assigns a new random session ID

Example 1 of Session Fixation Attack

1. Attacker 

1. Attacker tricks Alice into following a link that points to Server with sessionID of xyz in URL
2. Server sees that sessionID xyz already exists
   - Does not send new sessionID to Alice, expects Alice to validate the existing xyz (which she does)
3. Attacker can use xyz to access Alice's account

Example 2 of Session Fixation Attack

1. Attacker 

1. Attacker tricks Alice … with sessionID of xyz ~~in URL~~ in cookie via malicious script run on victim's browser

… (the rest as in Example 1)

Phishing Attacks

- Rely on human weakness and gullibility
  - Obtaining victim's credentials by pretending to be a trusted entity (e.g., the victim's bank)
  - Possible if (i) victim follows link to malicious web site, and (ii) pays no attention to absence of htpps or to wrong (but misleading) URL (such as www.amazon.badsite.com)
- Many related variants
  - Spear phishing targets a specific individual
  - Whaling targets high-value victims (CEO, rich, …)
  - Clone phishing modifies a previous legitimate email ("Update:") by including malicious link
- We review some defenses deployed by banks

The “Familiar Picture” Defense

- When Alice enrolled, she selected a picture P
  - She selected P out of many choices displayed to her
  - The selected P is not known to the phishing adversary
- The login process requires Alice to see P:
  - Alice enters her username
  - Based on her username, bank’s server displays P
  - She enters her password only after she recognizes P
- This works only against crude phishing web sites
  - Those designed to non-adaptively harvest passwords

“Familiar Picture” Defense (cont’d)

- It fails against a phishing site X that carries out an adaptive (“man in the middle”) attack:
  1. Victim Alice enters username on X (by definition, “victim” pays no attention to the lack of htpps)
  2. X mirrors Alice’s login to the bank server
  3. Bank server displays P to X
  4. X mirrors P to Alice, gets Alice’s password
  5. X can now impersonate Alice (possibly multiple times)

“Text to Cell Phone” Defense

- The login process at the bank requires Alice to enter a one-time password sent by the bank to her cell phone:
  - Alice enters her username
  - Bank’s server sends to Alice’s cell phone a text message containing a random one-time password
  - Alice enters the password she just received in the text msg
- This too fails against a phishing site X that carries out a “man in the middle” attack
  - Alice enters the one-time password on X, X mirrors it to the bank and gains access to Alice’s account (a one-time access, because the password cannot be re-used)

Typo-squatting attacks

- Attacker obtains domain names similar to those of legitimate entities, e.g.,
  - amazoon.com, pudrue.edu, porshe.com, ferari.com, paypaal.com, …etc
- Attacker buys certificates for those names, then waits for victims who mis-type the legitimate domain name
  - Certificates match, so there is no browser security warning to alert the victim

Typo-squatting attacks (cont'd)

- As in phishing attacks, the attacker needs the fake web site to mimic the genuine one
  - So as to trick the victims into entering their authentication credentials
- Sometimes they only seek the high traffic
  - The goal is then to charge for advertising
  - Legitimate web sites typically sue them for trademark infringement (civil lawsuits – unlike the password-theft cases which are a criminal activity)

More on Validation

- The discussion here is mainly for the validation of user inputs
  - Some of the points made also apply to other kinds of validations
- Whitelisting approach
  - Specifying what is allowed in some formal language (such as regular expressions)
- Blacklisting approach
  - Similar except that it specifies what is *not* allowed
- In general, input whitelisting is preferred

Why is input whitelisting preferred?

- In both blacklisting and whitelisting, there is a multiplicity of ways in which an input can be provided
  - All these different ways are functionally equivalent to each other, but some are more complex (“sneaky”) than others
  - But one or more may escape detection by the formal verification system
- In whitelisting, honest users have no reason to use that multiplicity (they don’t want to be misclassified)
  - They'd use the most straightforward formulation
- In blacklisting, malicious users have every reason to make use of multiplicity (they want to be misclassified)

Which Inputs to Validate

- Which inputs are untrusted?
- Any input in a user request
  - Query string parameters
  - Web form values
  - Header values
  - Cookie values
- Input from the server's own database?
  - Yes if it came from users, another organization, …

When to Validate

- Validate-early approach
  - Validate data as it enters server's system, e.g.,
  - When server first receives a database from a partner web site (even if it is not to be used much)
  - When a new user creates a record in the server's database (even if it will never be used thereafter)
- Validate-late approach
  - Validate data just before it is used
- Which one? (Probably both)
  - Safer to wash the "pre-washed" fruit before eating it

Other Client DIY Defenses

- De-activate any application features you do not use
  - More features => more vulnerabilities
  - Many features are activated by default (must be explicitly switched off)
- Activate higher security features
  - Many are off by default (must be activated)
  - E.g., second factor for authentication (preferably using out-of-band communication like cell phone)

Other Server-Side Controls

- IP address blacklisting
  - Against known sources of attack
- IP address whitelisting
  - Not common for public-facing applications, but useful in private networks – corporate intranets where subnets have meaning in the organizational structure (e.g., the HR subnet)
  - Useful in case of an application meant for use only by a specific corporate partner

More Server-Side Controls (cont'd)

- Limitations based on URL
  - Server can specify which users and groups can access which URLs
- Use native OS security framework
  - Web server and apps are subject to the OS constraints for the context within which they run
  - File and directory permissions, access control, …
  - In case web app or server is compromised, a lower privilege limits the extent of the resulting damage