Development Issues within AJAX Applications: How to Divert Threats

Lars Ewe
CTO
Cenzic
lars@cenzic.com

OWASP
July, 2009

The OWASP Foundation
http://www.owasp.org

Agenda

- What is AJAX?
- AJAX and Web App Security
- AJAX and Test Automation
- Vulnerability Examples:
  XSS, CSRF & JavaScript Hijacking
- AJAX Best Security Practices
- Demo
- Q & A

What is AJAX?

- Asynchronous JavaScript And XML
- AJAX allows for a new generation of more dynamic, more interactive, faster Web 2.0 applications
- AJAX leverages existing technologies, such as Dynamic HTML (DHTML), Cascading Style Sheets (CSS), Document Object Model (DOM), JavaScript Object Notation (JSON), etc., and the (a)synchronous XMLHTTPRequest (XHR)
- Not just a set of technologies, but a new Web application development approach and methodology

What is AJAX? (contd.)

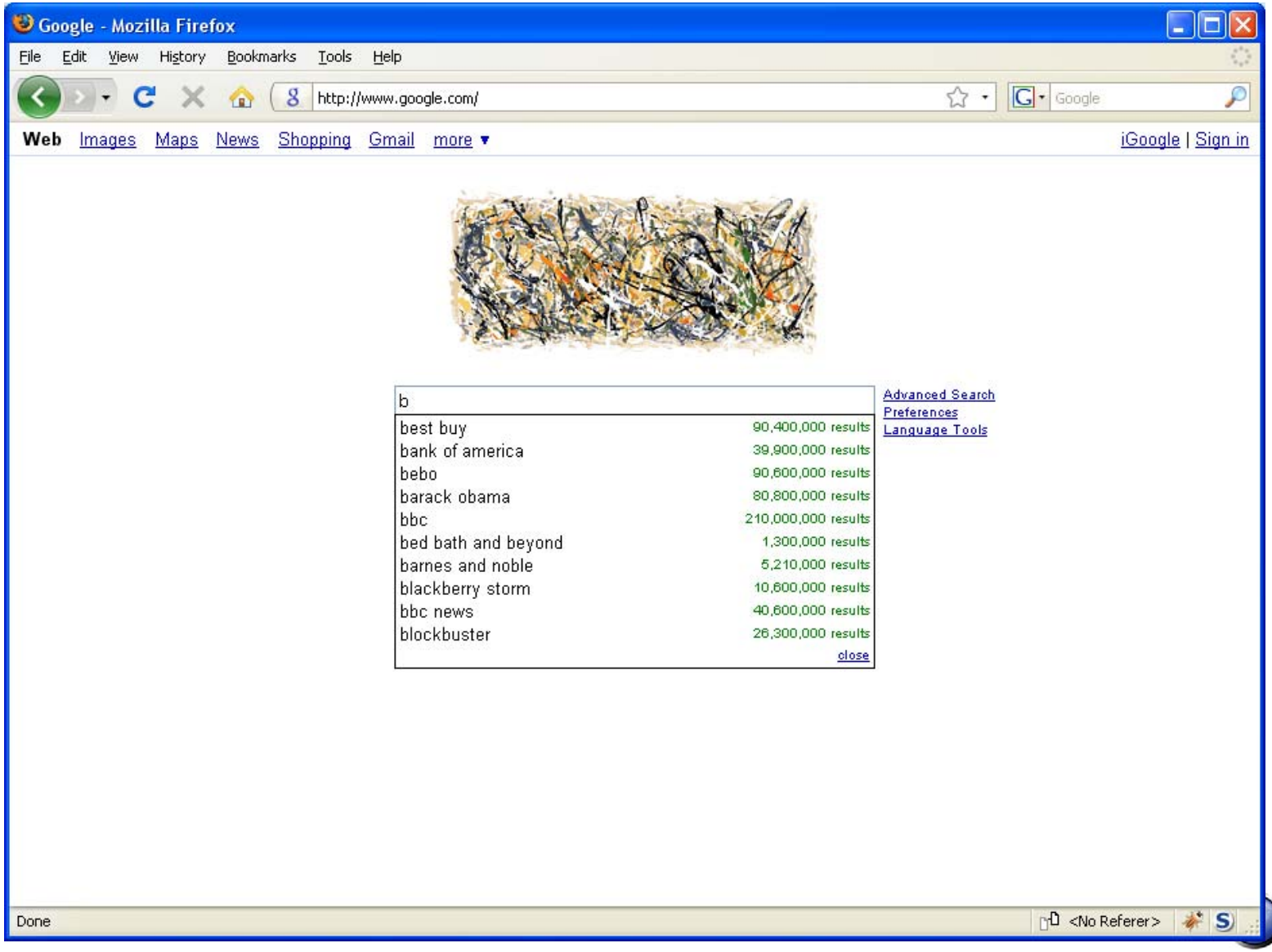
- XHR allows for (a)synchronous server requests without the need for a full page reload
- XHR “downstream” payload can be
  - XML, JSON, HTML/JS snippets, plain text, serialized data, basically pretty much anything…
- Responses often get further processed using JavaScript and result in dynamic web page content changes through DOM modifications

AJAX Code Example

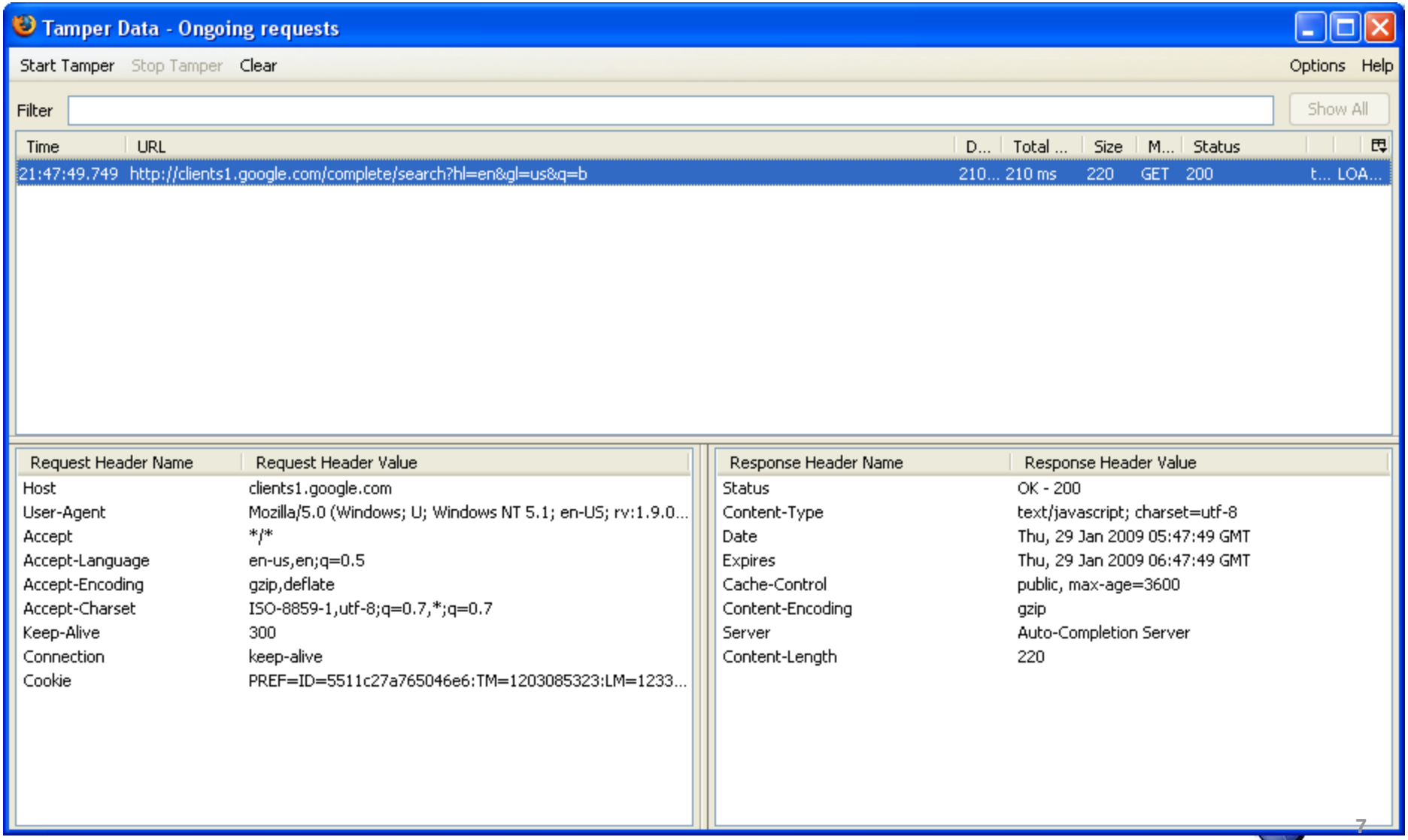
```
xhr = new XMLHttpRequest();
xhr.open("GET", AJAX_call?foo=bar, true);
xhr.onreadystatechange = processResponse;
xhr.send(null);

function processResponse () {
      if (xhr.readyState == 4) {
            if (request.status == 200) {
                  response =
                        xhr.responseText;
                  ........
            }
      }
}
```

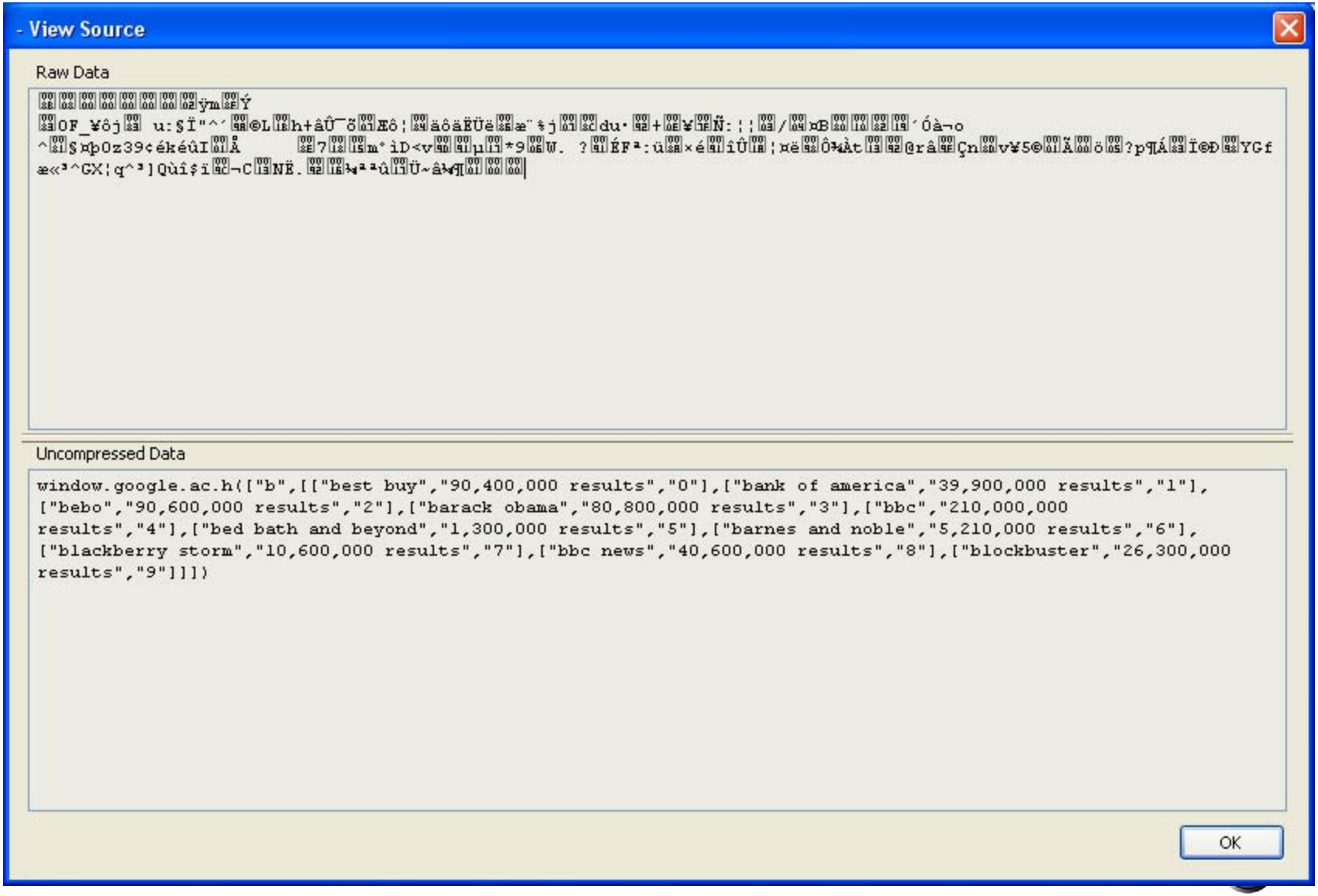
AJAX Example #1



AJAX Example #1

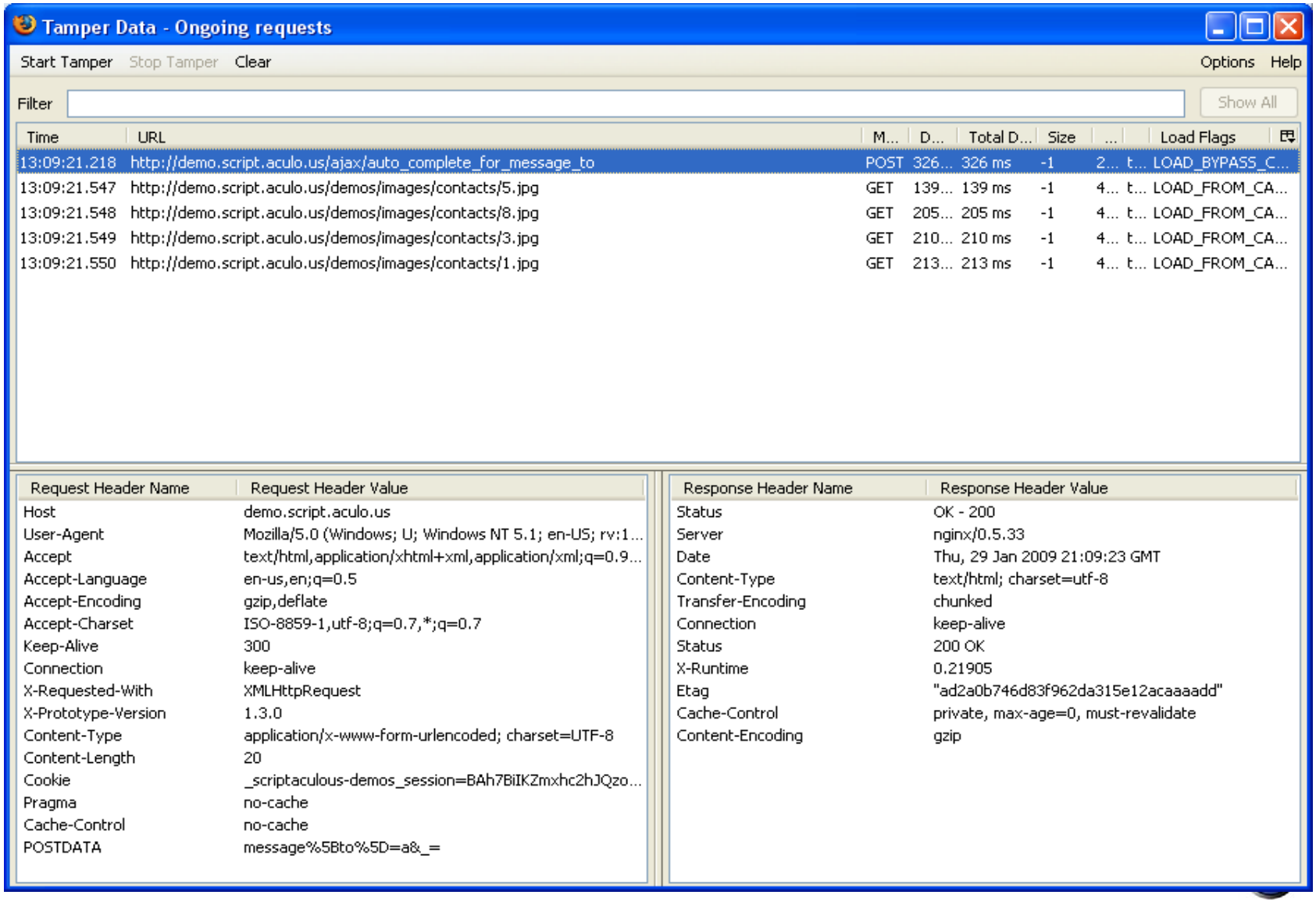


AJAX Example #1

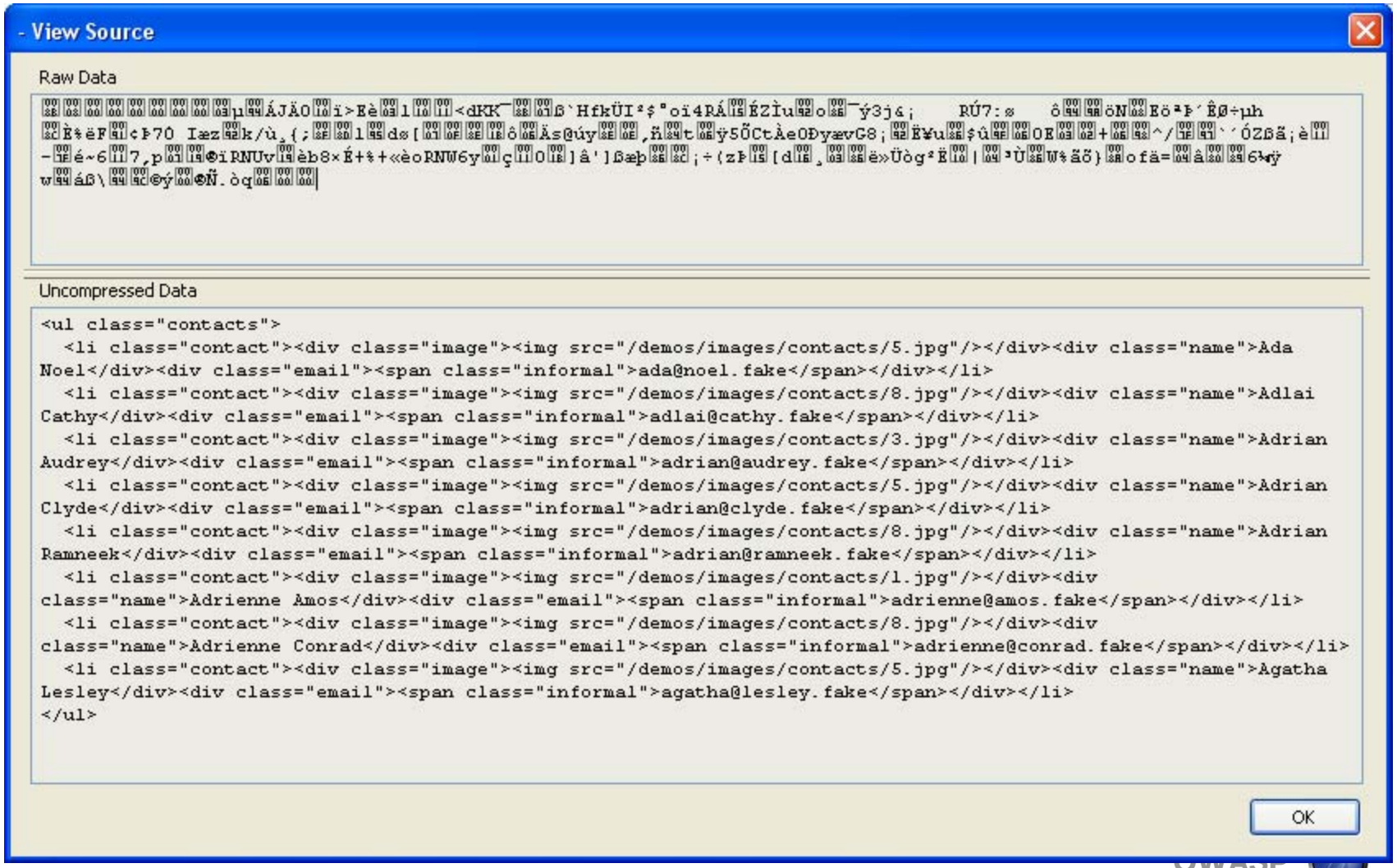


AJAX Example #2

AJAX Example #2



AJAX Example #2



AJAX Deployment Statistics

- Cenzic CTS (SaaS): ~30% of recently tested applications use AJAX
- >50% AJAX developer growth year-over-year – Evans Data, 2007
- ~3.5 million AJAX developers worldwide – Evans Data, 2007
- 60% of new application projects will use Rich Internet Application (RIA) technologies such as AJAX within the next three years – Gartner, 2007

AJAX and the Same Origin Policy

- Same origin policy is a key browser security mechanism
  - To prevent any cross-domain data leakage, etc.
  - With JavaScript it doesn't allow JavaScript from domain A to access content / data from domain B
- In the case of XHR, the same origin policy does not allow for any cross-domain XHR requests
  - Developers often don't like this at all!

Common Cross Domain Workarounds

Cross-domain access is often still implemented by various means, such as

- Open / Application (server-based) proxies
- Flash & Java Applets (depending on `crossdomain.xml`)
  - E.g. `FlashXMLHttpRequest` by Julien Couvreur
- RESTful web service with JS callback and JSON response
  - E.g. `JSONscriptRequest` by Jason Levitt

AJAX Frameworks

- AJAX frameworks are often categorized as either “Client” or “Proxy/Server” framework
- “Proxy/Server” frameworks sometimes result in unintended method / functionality exposure
- Beware of any kind of “Debugging mode”
- Remember: Attackers can easily “fingerprint” AJAX frameworks
- Beware of JavaScript Hijacking
  - Don't use HTTP GET for “upstream”
  - Prefix “downstream” JavaScript with `while(1);`

AJAX and Web App Security

- AJAX potentially increases the attack surface
  - More “hidden” calls mean more potential security holes
- AJAX developers sometimes pay less attention to security, due to it’s “hidden” nature
  - Basically the old mistake of security by obscurity
- AJAX developers sometimes tend to rely on client side validation
  - An approach that is just as flawed with or without AJAX

AJAX and Web App Security (contd.)

- Mash-up calls / functionality are often less secure by design
  - 3rd party APIs (e.g. feeds, blogs, search APIs, etc.) are often designed with ease of use, not security in mind
  - Mash-ups often lack clear security boundaries (who validates, who filters, who encodes / decodes, etc.)
  - Mash-ups often result in untrusted cross-domain access workarounds
- AJAX sometimes promotes dynamic code (JavaScript) execution of untrusted response data

The Bottom Line…

AJAX adds to the problem of well-known Web application vulnerabilities, such as XSS, CSRF, etc.

AJAX and Test Automation

- Spidering is more complex than just processing ANCHOR HREF's; various events need to be simulated (e.g. mouseover, keydown, keyup, onclick, onfocus, onblur, etc.)
- Timer events and dynamic DOM changes need to be observed
- Use of non-standard data formats for both requests and responses make injection and detection hard to automate
- Page changes after XHR requests can sometimes be delayed
- In short, you need to have browser like behavior (JS engine, DOM & event management, etc.)

Cross-Site Scripting (XSS)

- **What is it?:** The Web Application is used to store, transport, and deliver malicious active content to an unsuspecting user.
- **Root Cause:** Failure to proactively reject or scrub malicious characters from input vectors.
- **Impact:** Persistent XSS is stored and executed at a later time, by a user. Allows cookie theft, credential theft, data confidentiality, integrity, and availability risks. Browser Hijacking and Unauthorized Access to Web Application is possible.
- **Solution:** A global as well as form and field specific policy for handling untrusted content. Use whitelists, blacklists, and regular expressions to ensure input data conforms to the required character set, size, and syntax.

Cross-Site Request Forgery (CSRF)

- **What is it?:** Basic Web application session management behavior is exploited to make legitimate user requests without the user's knowledge or consent.
- **Root Cause:** Basic (cookie-based) session id management that is vulnerable to exploitation.
- **Impact:** Attackers can make legitimate Web requests from the victim's browser without the victim's knowledge or consent, allowing legitimate transactions in the user's name. This can results in a broad variety of possible exploits.
- **Solution:** Enhance session management by using non-predictable "nonce" or other unique one-time tokens in addition to common session identifiers, as well as the validation of HTTP Referrer headers.

JavaScript Hijacking

- **What is it?:** An attack vector specific to JavaScript messages. Confidential data contained in JavaScript messages is being accessed by the attacker despite the browser's some origin policy.
- **Root Cause:** The `<script>` tag circumvents the browser's same origin policy. In some cases the attacker can set up an environment that lets him observe the execution of certain aspects of the JavaScript message. Examples: Override/implement native Object constructors (e.g. Array) or callback function. This can result in access to the data loaded by the `<script>` tag.
- **Impact:** Data confidentiality, integrity, and availability with the ability to access any confidential data transferred by JavaScript.
- **Solution:** Implement CSRF defense mechanisms; prevent the direct execution of the JavaScript message. Wrap your JavaScript with non-executable pre- and suffixes that get stripped off prior to execution of the sanitized JavaScript message. Example: Prefix your JavaScript with `while(1);`

JavaScript Hijacking

Example #1: Override Array Constructor

Attacker code (override Array constructor)

```
<script type="text/javascript">
function Array(){
/* Put hack to access Array elements here */
}
</script>
```

AJAX Call

```
<script src="http://AJAX_call?foo=bar"
  type="text/javascript"></script>
```

Example AJAX response

```
["foo1","bar1"],["foo2","bar2"]
```

JavaScript Hijacking

Example #2: Implement Callback

Attacker code (implement callback)

```
<script type="text/javascript">
function callback(foo){
/* Put hack to access callback data here */
}
</script>
```

AJAX Call

```
<script src="http://AJAX_call?foo=bar"
  type="text/javascript"></script>
```

Example AJAX response

```
callback(["foo","bar"]);
```

Preventing JavaScript Hijacking

A simple code example

```
var object;
var xhr = new XMLHttpRequest();
xhr.open("GET", "/object.json", true);
xhr.onreadystatechange = function () {
            if (xhr.readyState == 4) {
                  var txt = xhr.responseText;
                  if (txt.substr(0,9) == "while(1);") {
                        txt = txt.substring(10);
                        Object = eval("(" + txt + ")");
                  }
            }
};
xhr.send(null);
```

Remember, the attacker cannot sanitize the JavaScript, since they are relying on the `<script>` tag

AJAX Best Security Practices

Pretty much all the usual Web app security best practices apply:

- Analyze and know your security boundaries and attack surfaces
- Beware of reliance on client-side security measures
  - Always implement strong server side input & parameter validation (black & whitelisting)
  - Test against a robust set of evasion rules
  - Remember: The client can never be trusted!
- Assume the worst case scenario for all 3rd party interactions
  - 3rd parties can inherently not be trusted!

AJAX Best Security Practices (contd.)

- Be extremely careful when circumventing same origin policy
- Avoid / limit the use of dynamic code / `eval()`
- Beware of JavaScript Hijacking (prefix JavaScript with `while(1);`)
- Implement anti-CSRF defenses
- Escape special characters before sending them to the browser (e.g. `<` to `<`)
- Leverage HTTPS for sensitive data, use `HTTPOnly` & `Secure` cookie flags
- Use parameterized SQL for any DB queries
- Also see owasp.org and OWAP dev guide

XSS AJAX & JavaScript Hijacking Demo