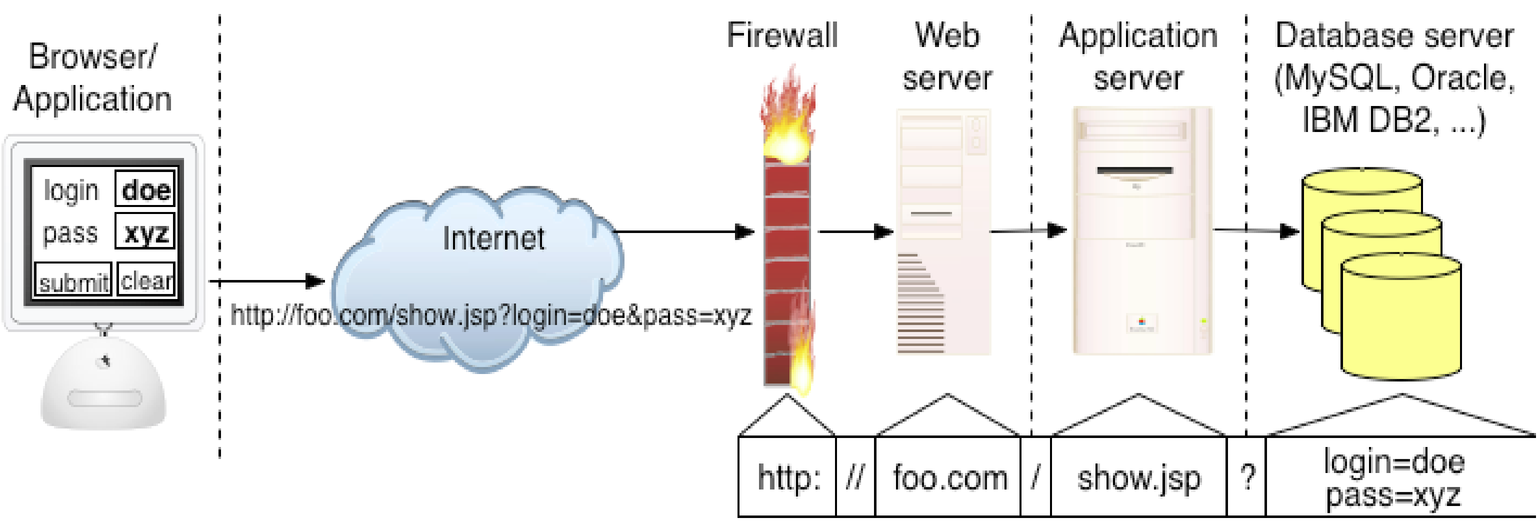
Web Security

Historical Web

- Historically, the web was just a request response protocol
- HTTP is stateless, which means that the server essentially processes a request independent of prior history
- Envisioned as a way for exchanging information

Current Web

- Evolving into a platform for executing programs that support day-to-day tasks
- A lot of state needs to be maintained
- Distributed computation, and trust model



Structure of HTTP GET request

- Connect to: www.example.com
  - TCP Port 80 is the default for http, others may be specified explicitly in the URL.
- Send: GET /index.html HTTP/1.1
- Server Response:

HTTP/1.1 200 OK

Date: Mon, 23 May 2005 22:38:34 GMT

Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)

Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT

Etag: "3f80f-1b6-3e1cb03b"

Accept-Ranges: bytes

Content-Length: 438

Connection: close Content-Type: text/html; charset=UTF-8

GET with parameters

- GET /submit_order?sessionid=79adjadf888888768&
  pay=yes
  HTTP/1.1
- User Inputs sent as parameters to the request

POST Requests

- Another way of sending requests to HTTP servers
- Commonly used in FORM submissions
- Message written in the BODY of the request
- Sending links with malicious parameter values is difficult when a web site accepts only POST requests.
- But a script running on a malicious web site can as easily send a POST request (as a GET request) to another web site.

Cookies

- HTTP is stateless, therefore client needs to remember state and send this with every request
- Cookies are the common way of keeping state

Client:

```
GET /index.html HTTP/1.1
Host: www.example.org
```

Server:

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: sess-id=3773777adbdad

(content of page)
```

Cookies…

- Browsers send cookie with every subsequent request

```
GET /spec.html HTTP/1.1
Host: www.example.org
Cookie:  sess-id=3773777adbdad
```

- Now server can look up stored state through sess-id
- Alternative to cookies: hidden form fields.

Lifetime of Cached Cookies and HTTP Authentication Credentials

- Temporary cookies cached until browser shut down, persistent ones cached until expiry date

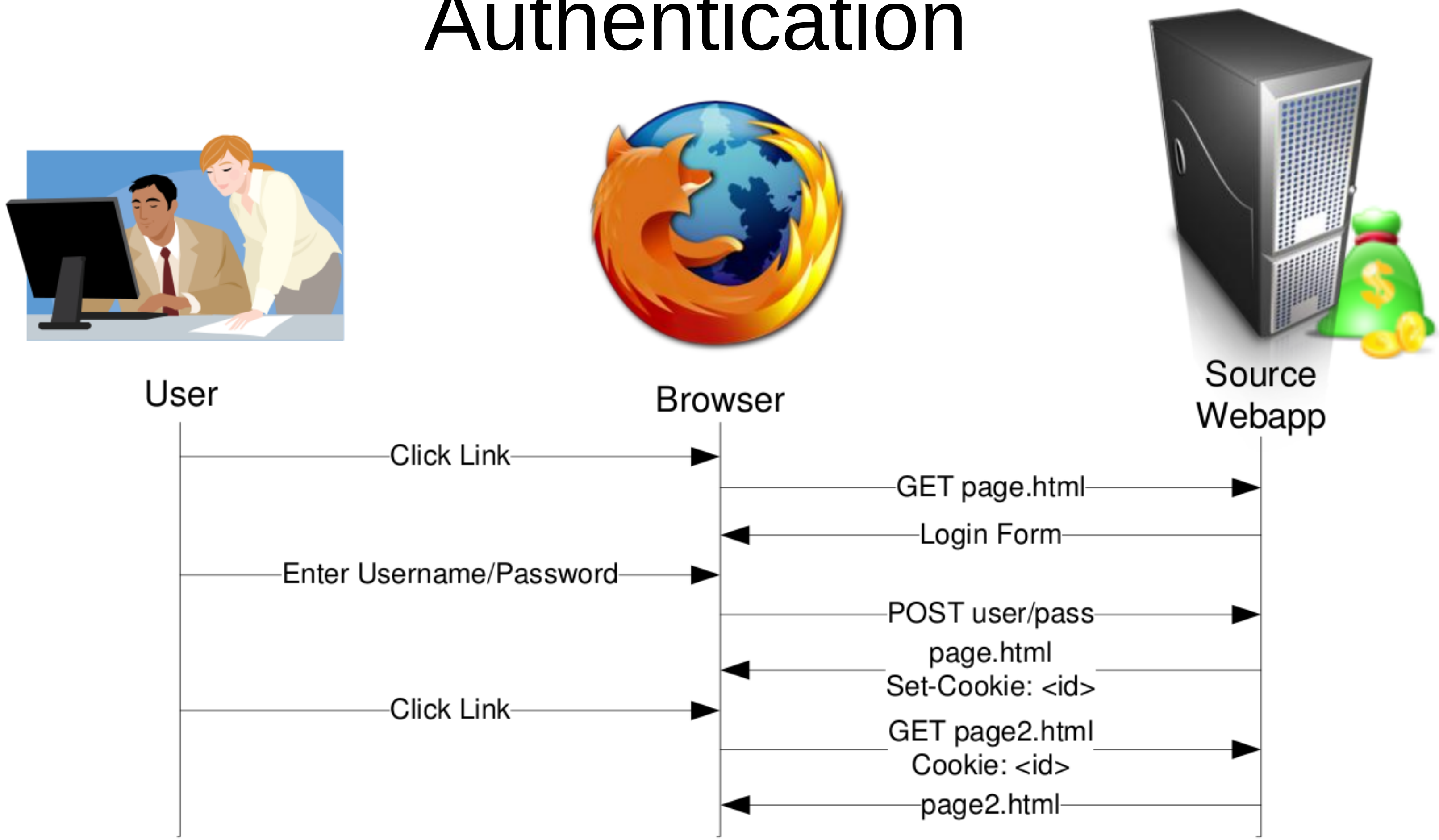
- HTTP authentication credentials cached in memory, shared by all browser windows of a single browser instance

- Caching depends only on browser instance lifetime, not on whether original window is open

Web Security

- Web Security is concerned with ensuring the following 3 properties for web applications:
    - **Authentication:** securely identify users on top of HTTP, which is a stateless protocol.
    - **Confidentiality:** protect any sensitive data that websites serve to the browser from other websites, and the user's own sensitive data outside the browser from any website.
    - **Integrity:** ensure that the data and the code served to users cannot be tampered with.

Authentication

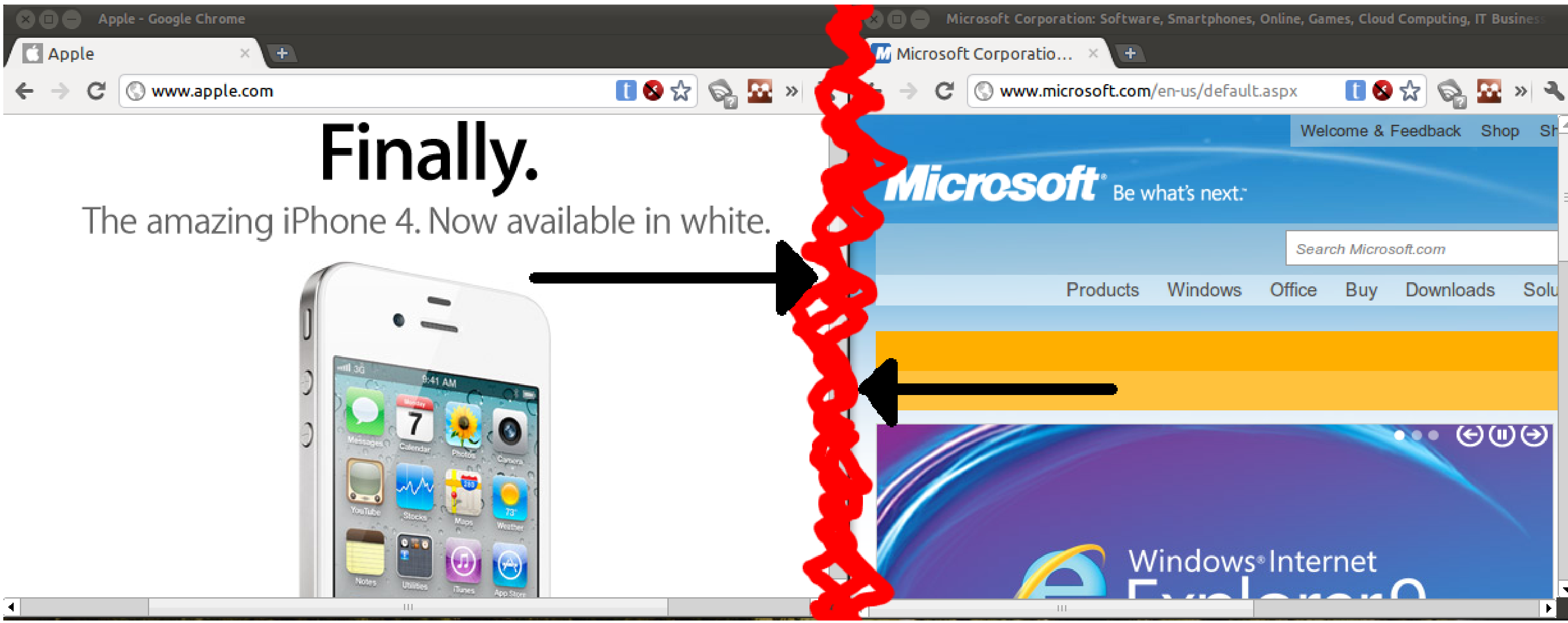


- HTTP is a stateless protocol.
  - User Authentication: Use cookies and send them implicitly for convenience.
  - Server Authentication: SSL + Certification Authorities

HTTP Request Authentication

- HTTP is stateless, so web apps have to associate requests with users themselves
  - HTTP authentication: username/passwd automatically supplied in HTTP header
  - Cookie authentication: credentials requested in form, after POST app issues session token
    - Browser returns session cookie for each request
  - Hidden-form authentication: hidden form fields transfer session token
- Http & cookie authentication credentials are cached, so they don’t have to be supplied with each request

Confidentiality (Browser)



- No mutual trust among parties.
- Confidentiality through Isolation: Same-Origin Policy (SOP)
  - Partition the Web into domains and isolate sensitive data such as cookie, network data and DOM nodes.

Confidentiality (OS)

- Users do not trust the websites they visit.
- Again: Confidentiality through Isolation
  - Sandboxing: only expose a safe API to web application that limits their interaction with the browser
    - DOM manipulation, cookie storage, drawing inside the browser window, etc.
    - Recent developments: HTML5, WebGL, NaCL. Web developers need more capabilities for dynamic applications.

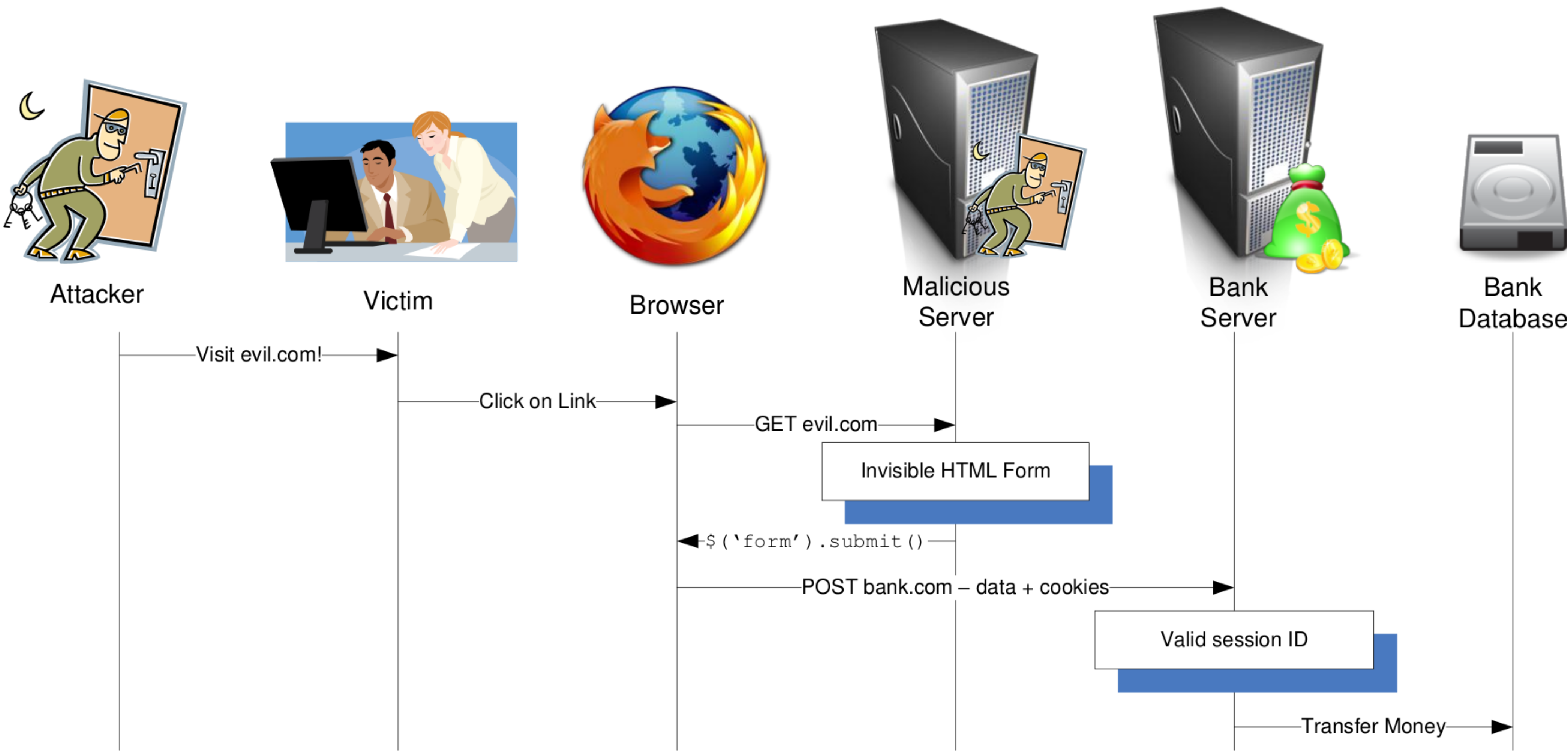
Integrity

- Network data integrity: HTTPS/DNSSEC
  - Also used to authenticate the server (e.g Banks) and ensure network confidentiality.
  - Public-key protocol used to establish a session key to encrypt traffic.
- Browser data integrity: SOP
  - ``Integrity” as write access on confidential resources.

Attacks on Authentication

- CSRF and Clickjacking
  - Confused deputy attacks that cause the victim browser to send authenticated requests for the attacker's benefit
  - CSRF: Cross-site request forgery: attacker sends requests to another web site, impersonating browser user
  - Clickjacking: User intends to click on one link, but the browser recognizes a link on another site
    - Achieved using overlaid frames and by manipulating visibility related attributes

CSRF


Attacker
Victim
Browser
Malicious Server
Bank Server
Bank Database
Visit evil.com!
Click on Link
GET evil.com
Invisible HTML Form
$('form').submit()
POST bank.com – data + cookies
Valid session ID
Transfer Money

Cross-site Request Forgery (CSRF)

<form method=“POST” action=“/changepass”>

…

New Password: <input type=“password” name=“password”>

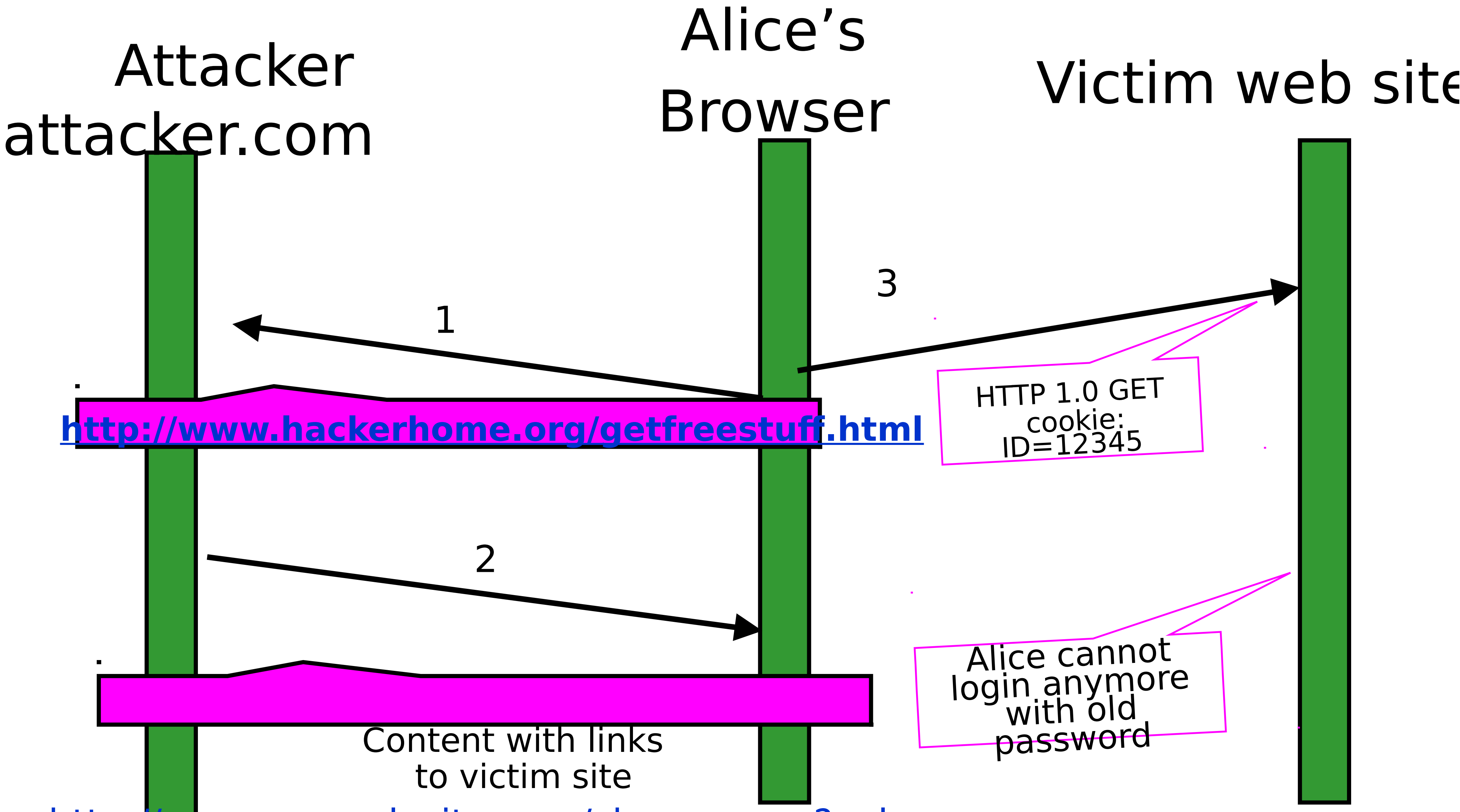
</form>

- Browser makes the following request :

  GET http://www.examplesite.com/changepass?val=newpassword HTTP 1.1

- Let’s say the application didn’t authenticate password change request using any other means
- An attacker can easily forge request!

Forged Requests



http://www.examplesite.com/changepass?val=newpassword

POST Example

- POST requests can also be forged
- Attacker lures the client to visit his /her web page

```
<iframe name=“hiddenframe” style=“display:none”>
<form method=“POST” name=“evilform”
  target=“hiddenframe” action=
  http://www.examplesite.com/update_password>
  <input type=“hidden” name=“password”
  value=“evilhax0r”>
</form>
<script>document.evilform.submit()</script>
</iframe>
```

Possible targets of CSRF

- Banks
  - Attacker can issue a request to transfer money from victim’s bank account to attacker’s
- E-commerce sites
  - Purchase items using victim’s account, ship to attacker
- Forums and Social network sites
  - Post articles using victim’s identity
- Home/Intranet firewall
  - Reconfigure firewall to permit connections from the Internet to a host behind the firewall
  - Note that victim user’s location is exploited: the attacker (typically) cannot communicate with the firewall, but the user’s browser can

CSRF Impacts

- Malicious site can’t read info, but can make write requests to our app!
- In Alice’s case, attacker gained control of her account with full read/write access!

Preventing CSRF

- HTTP requests originating from user action are indistinguishable from those initiated by attacker
- Need own methods to distinguish valid requests
  - Inspecting Referer Headers
  - Validation via User-Provided Secret
  - Validation via Action Token

Inspecting Referer Headers

- Referer header specifies the URI of document originating the request

- Assuming requests from our site are good, don’t serve requests not from our site
- Unfortunately, Referrer information may be suppressed by browsers (or firewalls) for privacy reasons

Validation via User-Provided Secret

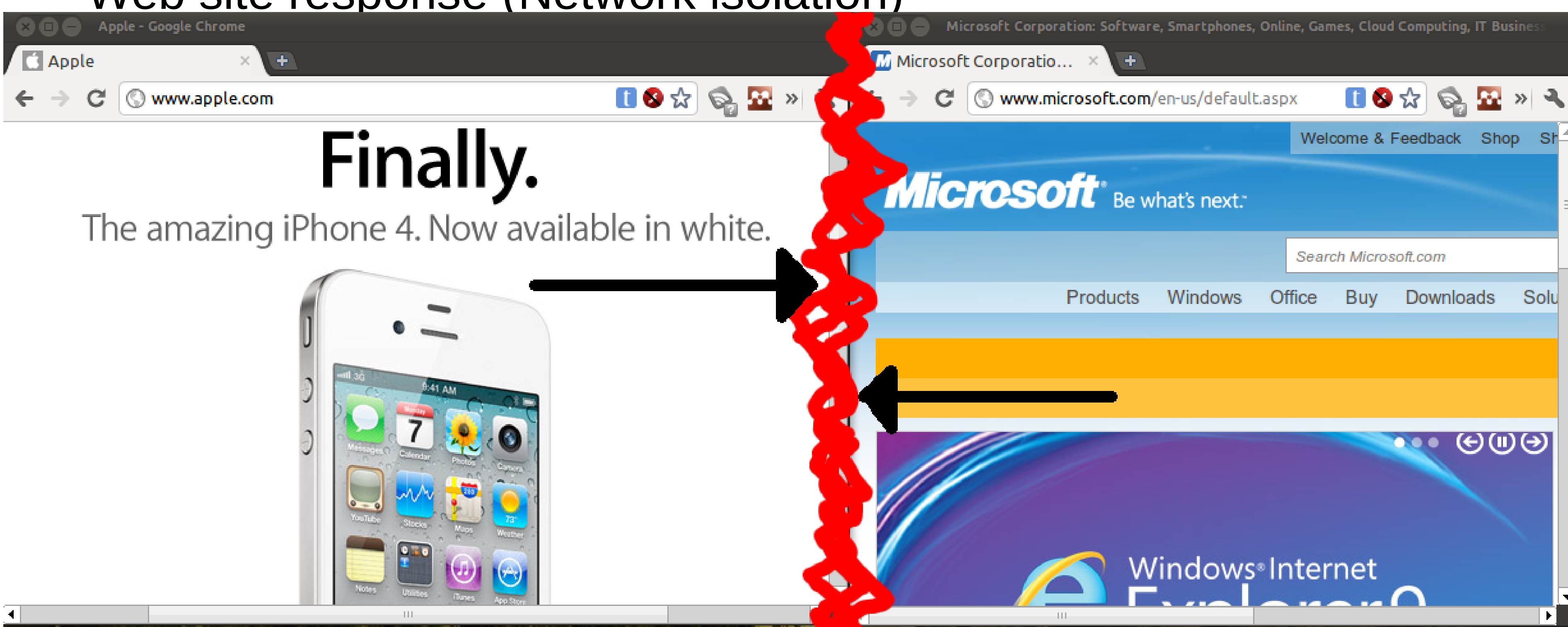
- Can require user to enter secret (e.g. login password) along with requests that make server-side state changes or transactions
- Ex: The change password form could ask for the user’s current password
- Security vs convenience: use only for infrequent, “high-value” transactions
  - Password or profile changes
  - Expensive commercial/financial operations

Validation via Action Token

- Add special action tokens as hidden fields to “genuine” forms to distinguish from forgeries
- Same-origin policy prevents 3rd party from inspecting the form to find the token
- Need to generate and validate tokens so that
  - Malicious 3rd party can’t guess or forge token
    - Browser’s Same Origin Policy prevents attacker from “reading” the token
  - Then can use to distinguish genuine and forged forms

Same-Origin Policy (SOP)

- The SOP partitions the web into domains (according to their DNS origin) and isolates sensitive data from scripts running in other domains.
  - What is sensitive data?
    - Cookies
    - Web page content (DOM isolation)
    - Web site response (Network isolation)



SOP: Cookie Isolation

- Each domain has its own set of independently managed cookies, and these are embedded only in requests to the same domain.
- Only scripts running from the same domain and responses from the same domain can read and write cookies
  - HTTP-Only cookies

SOP: Page content isolation

- Basic unit of isolation in a browser is a <frame>
  - document.write – refers to the current frame
- DOM Isolation
  - Scripts only have access to DOM elements on the same domain.
  - Frames embedded in a page are part of the DOM tree of the parent, but the policy still applies:
    - `document.frames[0].title`
    - Only accessible if the parent is from the same origin.

SOP: Network isolation

- Script can send requests to arbitrary sites
- But scripts cannot read responses from any server
  - They can still send blind requests to other domains.
  - Is it safe for a malicious script to issue a request if it cannot read the response?
    - CSRF
- Exception: XmlHttpRequests permit a script to read from its origin server

Embedding and SOP: Caveats

- For embedded content, origin of the content may be different from the domain used for SOP checks
  - Scripts retrieved from B and embedded in A run with A privileges.
    - Akin to user A running an executable written by B in a UNIX environment.
    - Plugins implement their own SOP-like policies.
      - Flash keeps its server origin.
  - Cross-site scripting attacks exploit this

Same-Origin Policy: Exceptions

- Some resources are not considered sensitive and can be accessed across domains
  - Browser History: CSS allows website to use different rules for visited and unvisited links.
  - CSS rules: they can be read even when importing a cross-origin stylesheet
  - Unsurprisingly, two attacks use these exceptions for information leaks
    - Cross-origin CSS and CSS history hacks exploit these exceptions

A web site vulnerable to XSS

- Host: www.vulnerable.site
- GET /welcome.cgi?name=value

HTTP/1.0

- Displays name submitted in the web page
- Example

GET /welcome.cgi?name=Joe%20Hacker HTTP/1.0

Web site response

<HTML>

<Title>Welcome!</Title>

Hi Joe Hacker

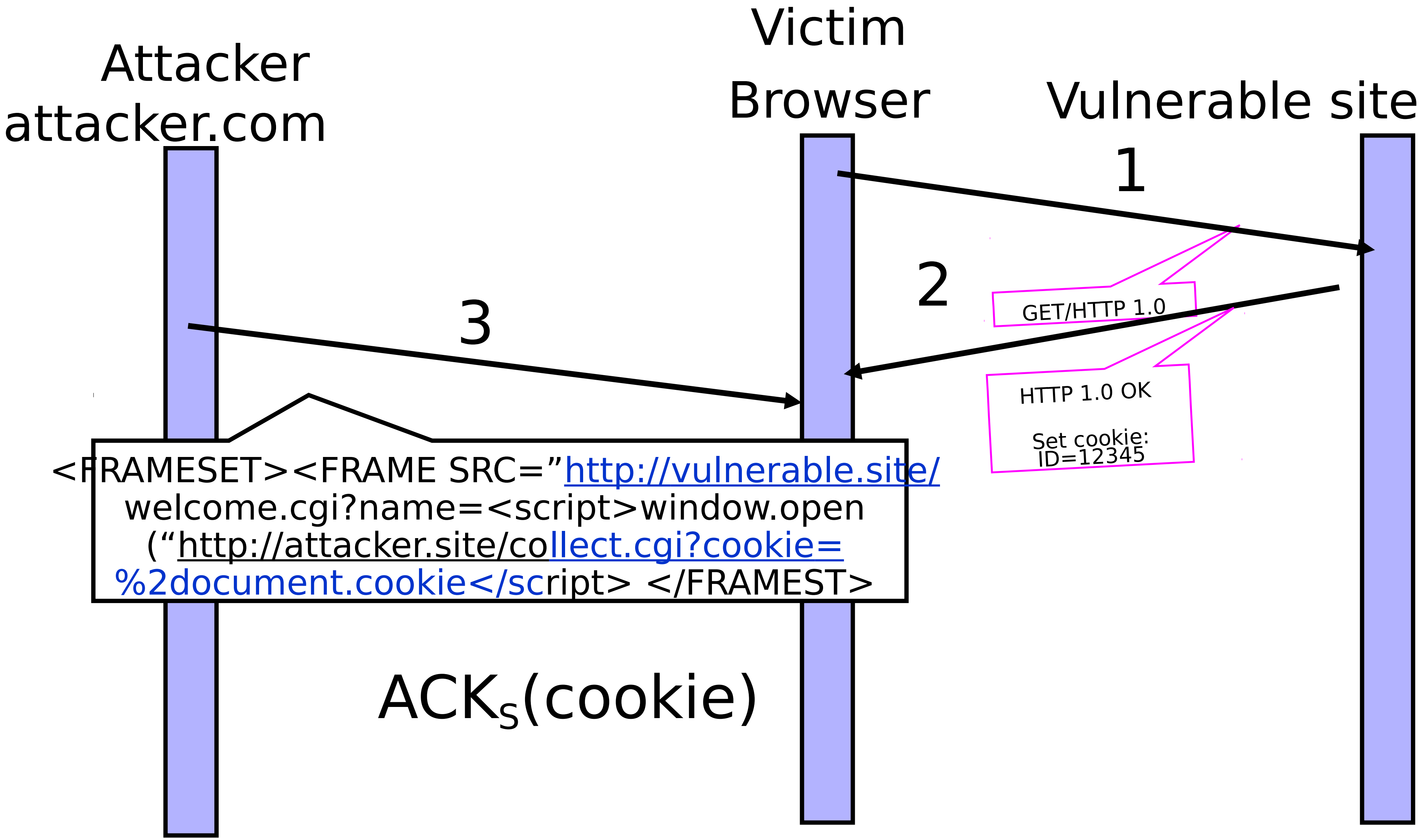
Welcome to our system

...

</HTML>

How can this be abused??

Reflected XSS attacks



Summary

- Attacker causes victim to click on maliciously crafted link
- request goes to vulnerable web site
- web site does not perform input filtering
- returns a page that contains executable code that sends private information to attacker

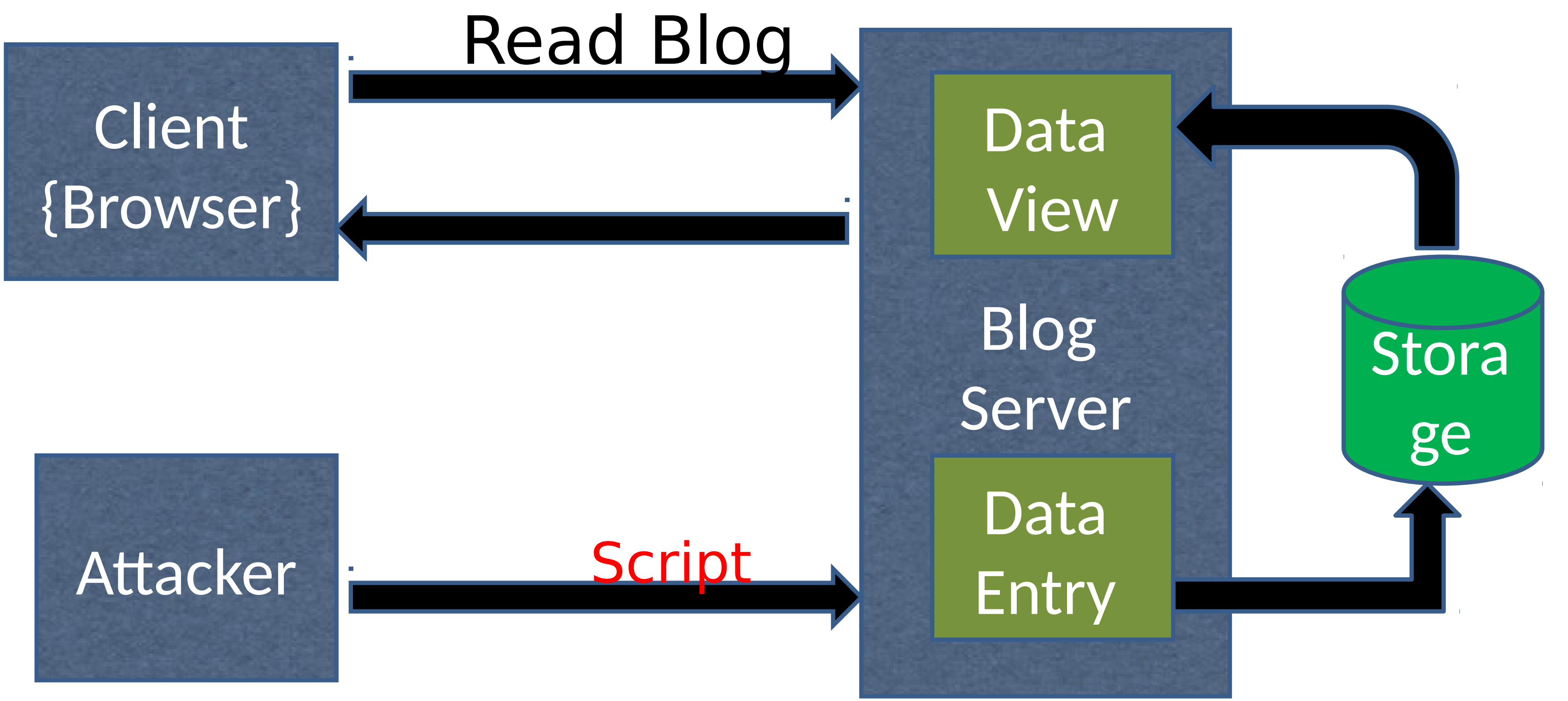
Attack details

- Above attack requires victim to click on attacker link
  - Easy way: use email messages with enticing information
  - victim clicks on link
  - Variation: Attacker provides scripting code as input to vulnerable web application

How to run passive attacks?

- These are attacks where user will not perform explicit actions
- How can this be possible?
- Think of a blog, where user input becomes part of the page’s comments
- Stealthy, and mostly unknown to user browsing the page

Problem Context



XSS

- Unauthorized scripts come from user input
- Can we identify scripts that are legitimate vs. those that are injected?
- If so, the web site can reject any script content that did not come from it
- This requires “tracking” user input as it flows through the application

References

1. XSS (Cross Site Scripting) Cheat Sheet Esp: for filter evasion http://ha.ckers.org/xss.html
2. Technical Explanation of The MySpace Worm

http://namb.la/popular/tech.html

3. Malicious Yahooligans www.symantec.com/avcenter/reference/**malicious.yahooligans.pdf**

Preventing XSS

- Never send untrusted data to browser
  - Such that data could cause execution of script
  - Usually can just suppress certain characters
- We show examples of various contexts in HTML document as *template snippets*
  - Variable substitution placeholders: `%(var)s`
  - `evil-script;` will denote what attacker injects
  - Contexts where XSS attack is possible

General Considerations

- Input Validation vs. Output Sanitization
  - XSS is not just a input validation problem
  - Strings with HTML metachars not a problem until they’re displayed on the webpage
  - Might be valid elsewhere, e.g. in a database, and thus not validated later when output to HTML
  - Sanitize: check strings as you insert into HTML doc
- HTML Escaping
  - a.k.a entity reference encoding
  - escape some chars with their literals
    - e.g. & = & < = < > = &rt; “ = "
    - Library functions exist

Simple Text

- Most straightforward, common situation
- Example Context:

```
<b>Error: Your query '%(query)s' did not return any results.</b>
```

  - Attacker sets `query = <script>evil-script;</script>`
  - HTML snippet renders as

```
<b>Error: Your query '<script>evil-script;</script>'
          did not return any results.</b>
```

- Prevention: HTML-escape untrusted data
- Rationale: If not escaped
  - `<script>` tags evaluated, data may not display as intended

Tag Attributes (e.g., Form Field Value Attributes)

- Contexts where data is inserted into tag attribute
- Example HTML Fragment:

```
<form ...><input name="query" value="%(query)s"></form>
```

  - Attacker sets

```
query = cookies"><script>evil-script;</script>
```

  - Renders as

```
<form ...>
<input name="query" value="cookies">
<script>evil-script;</script>">
</form>
```

- Attacker able to “close the quote”, insert script

More Attribute Injection Attacks

- Image Tag: `<img src=%(image_url)>`
- Attacker sets `image_url = http://www.examplesite.org/ onerror=evil-script;`
- After Substitution: `<img src=http://www.examplesite.org/ onerror=evil-script;>`
  - Lenient browser: first whitespace ends `src` attribute
  - `onerror` attribute sets handler to be desired script
  - Attacker forces error by supplying URL w/o an image
  - Can similarly use `onload`, `onmouseover` to run scripts
  - Attack string didn’t use any HTML metacharacters!

Preventing Attribute Injection Attacks

- HTML-escape untrusted data as usual
  - Escape &, ', ", <, >
- Also attribute values must be enclosed in " "
- Must escape the quote character to prevent “closing the quote” attacks as in example
- Decide on convention: single vs. double quotes
  - But escape both anyway to be safe

URL Attributes (href and src)

- Dynamic URL attributes vulnerable to injection
- Script/Style Sheet URLs: `<img src="%(script_url)s">`
  - Attacker sets `script_url = http://hackerhome.org/evil.js`
- javascript: URLS - `<img src="%(img_url)s">`
  - By setting `img_url = javascript:evil-script;` we get

    `<img src="javascript:evil-script;">`
  - And browser executes script when loading image

Preventing URL Attribute Injection

- Escape attribute values and enclose in " "
  - Follow earlier guidelines for general injection attacks
- Only serve data from servers you control
  - For URLs to 3rd party sites, use absolute HTTP URLS (i.e. starts with http:// or https://)
- Against `javascript:` injection, whitelist for good URLs (apply positive filter)
  - Not enough to just blacklist, too many bad URLs
  - Ex: even escaping colon doesn’t prevent script
  - Could also be `data:text/html,<script>evil-script;</script>`

Style Attributes

- Dangerous if attacker controls style attributes
  - Attacker injects:
  - Browser evaluates:

```
<div style="background: %(color)s;">I like colors.</div>

color = green; background-image:
  url(javascript:evil-script;)

<div style="background: green;
background-image: url(javascript:evil-script;);">
I like colors. </div>
```

- In IE 6 (but not Firefox 1.5), script is executed!
- Prevention: whitelist through regular expressions
  - Ex: `^([a-z]+)|(#[0-9a-f]+)$` specifies safe superset of possible color names or hex designation
  - Or expose an external param (e.g. `color_id`) mapped to a CSS color specifier (lookup table)

Within Style Tags

- Injections into `style=` attributes also apply for `<style>` tags
- Validate data by whitelisting before inserting into HTML document `<style>` tag
- Apply same prevention techniques as in earlier.

In JavaScript Context

- Be careful embedding dynamic content
  - `<script>` tags or handlers (`onclick`, `onload`, …)

```
<script>
var msg_text = '%(msg_text)s';
// do something with msg_text
</script>
```

  - Attacker injects: `msg_text = oops'; evil-script; //`
  - And `evil-script;` is executed!

```
<script>
var msg_text = 'oops';
evil-script; //';
// do something with msg_text
</script>
```

Preventing JavaScript Injection

- Don't insert user-controlled strings into JavaScript contexts
  - `<script>` tags, handler attributes (e.g. `onclick`)
  - within code sourced in `<script>` tag or using `eval()`
  - Exceptions: data used to form literal (strings, ints, …)
  - Enclose strings in `' '` & backslash escape (\n, \t, \x27)
  - Format non-strings so that string rep is not malicious
  - Backslash escaping important to prevent "escape from the quote" attack where notions of "inside" and "outside" string literals is reversed
  - Numeric literals ok if from `Integer.toString()`, …

Another JavaScript Injection Example

- From previous example, if attacker sets

```
msg_text = foo</script><script>evil-script;</script><script>
```

  - the following HTML is evaluated:

```
<script>var msg_text = 'foo</script>
<script>evil-script;</script>
<script>'// do something with msg_text</script>
```

- Browser parses document as HTML first
  - Divides into 3 `<script>` tokens before interpreting as JavaScript
  - Thus 1st & 3rd invalid, 2nd executes as `evil-script`

JavaScript-Valued Attributes

- Handlers inside `onload`, `onclick` attributes:
  - HTML-unescaped before passing to JS interpreter
  - Ex: `<input ... onclick='GotoUrl("%(targetUrl)s");'>`
  - Attacker injects: `targetUrl = foo");evil_script("`
  - Browser Loads: `<input ... onclick='GotoUrl("foo");evil_script("");'>`
  - JavaScript Interpreter gets `GotoUrl("foo");evil_script("");`
- Prevention: Two Rounds of Escaping
  - JavaScript escape input string, enclose in `''`
  - HTML escape entire attribute, enclose in `""`

Redirects, Cookies, and Header Injection

- Need to filter and validate user input inserted into HTTP response headers
- Ex: servlet returns HTTP redirect

```
HTTP/1.1 302 Moved
Content-Type: text/html; charset=ISO-8859-1
Location: %(redir_url)s

<html>
<head><title>Moved</title></head>
<body>Moved <a href='%(redir_url)s'>here</a></body>
</html>
```

  - Attacker Injects: (URI-encodes newlines)

```
oops:foo\r\nSet-Cookie: SESSION=13af..3b;
domain=mywwwservice.com\r\n\r\n
<script>evil()</script>
```

Non-HTML Documents & IE Content-Type Sniffing

- Browsers may ignore MIME type of document
  - Specifying `Content-Type: text/plain` should not interpret HTML tags when rendering
  - But not true for IE: mime-type detection
- AKA Content-Type Sniffing: ignores MIME spec
  - IE scans doc for HTML tags and interprets them
  - Even reinterprets image documents as HTML!

Java Security

- With binary code, memory and type safety issues complicate the problem of untrusted code
- Java and Javascript rely on safe languages, thereby avoiding most low-level issues we studied so far
  - Code can be created and executed only through sanctioned pathways, e.g., class loader
  - Access-control restrictions associated with classes will be strictly and fully enforced
    - No way to circumvent public/private restrictions by casting etc.
    - No buffer overflows
    - …

Java Security (Basics)

- Java permits remote code execution
  - In JDK 1.0, the picture was very simple:
    - Local (“trusted”) code ran without restrictions
    - Untrusted code was confined within a sandbox
      - Sandbox enforced access controls, e.g., whether files can be accessed, and if so, which ones
      - Sandbox policy was configurable
    - Caveats
      - Several significant (but perhaps not disastrous) errors were found in default policies, making users reluctant to permit running any code
      - Native code interface can negate type safety
    - Result
      - Java has come to be used primarily with trusted code

Java Security (Continued)

- JDK 1.1 permitted one more option
  - Signed code could be run outside the sandbox
- J2SE provides more flexibility
  - Any code (unsigned local, unsigned remote, signed remote) can be run in a sandbox with a custom policy.
  - Code from one source can invoke code from another source
    - What policy to enforce?
      - Java enforces the intersection of policies applicable to the current function and all its callers --- uses stack-walking to compute this info
      - Provides a doPrivileged primitive by which a piece of code can choose to use more permissive policies: namely, run a operation with the privileges available to that piece of code, regardless of who invoked it.

Java Security (Continued)

- Class loaders
  - Need to watch out for attacks that may subvert language restrictions: use a verification process for this process
    - Similar in spirit to the checks performed by SFI or NaCl
  - Ensure that appropriate security managers are loaded and restrictions enforced

Java Vs Javascript

- Java originally developed to support “active web pages”
  - Applets were intended to allow local execution of untrusted code
  - Security was achieved by restricting access to local resources, e.g., files
  - Drawbacks
    - did not provide good integration with the browser environment
    - focus was more on integrity rather than confidentiality
    - these factors led to the development of Javascript
  - Today, Adobe flash is closer in many ways to Java than Javascript

Java Vs Javascript

- Javascript takes a different approach
  - Language safety is still the basis
  - Use this basis to provide safe interface to the browser environment
    - Browser is the platform, not the underlying OS
      - It is not about whether untrusted code can access local files, but whether the browser permits it to do so (“trusted dialogs”)
    - The security model is object-oriented
      - What are the browser resources, which ones are accessible to untrusted code
    - Cookie-based model of browser security evolved in conjunction with Javascript, leading to excellent support for the same.