Stony Brook University

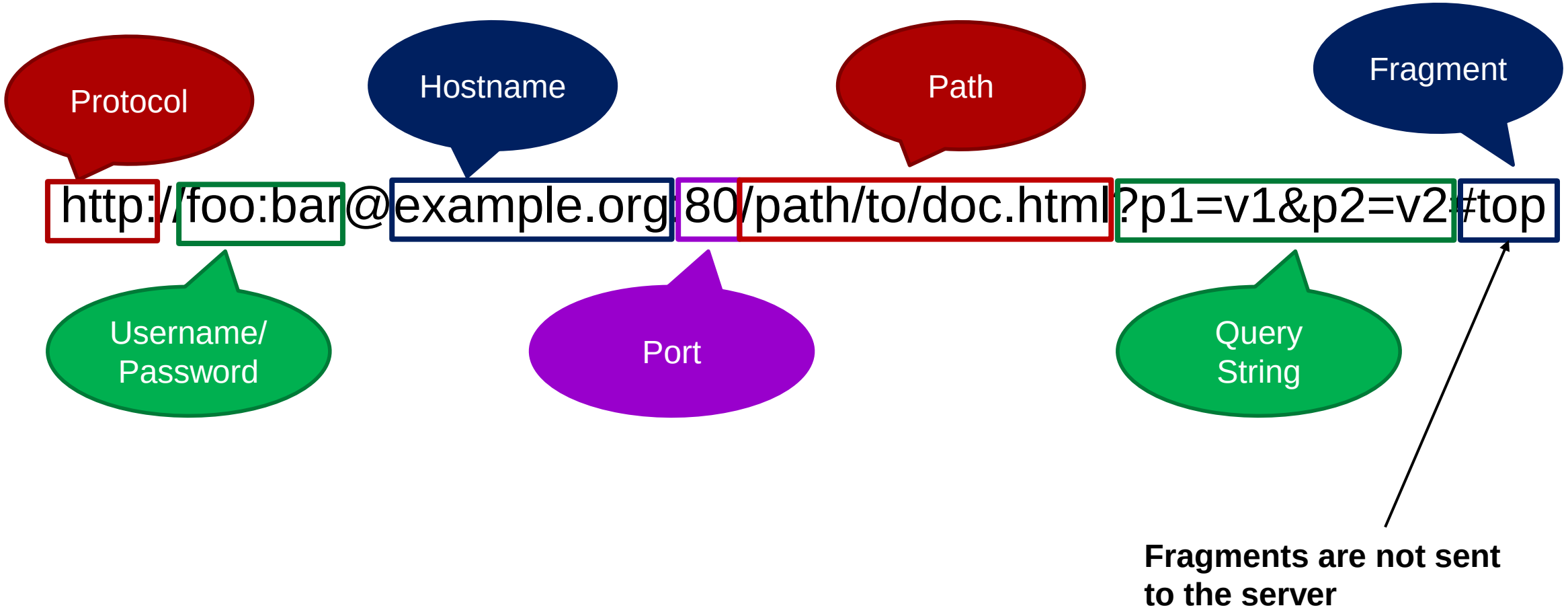
CSE 361: Web Security

Midterm Recap

Nick Nikiforakis

HTTP BASICS

Uniform Resource Locator (URL)



HTTP Evolution over Time: HTTP 1.0 (1991-1995)

- Requirements
  - serve content other than plain text documents
  - allow for authentication
  - allow for transmission of meta information, e.g., age of file
  - transmit data to the server (via forms)
- Result
  - Mandatory HTTP version in request
  - Optional headers in request and response
  - Status Line in response
  - New methods: POST and HEAD

```
GET / HTTP/1.0
Host: example.org
```

```
HTTP/1.0 200 OK
Content-Length: 123

<html>…
(connection closed)
```

HTTP Requests (since HTTP/1.0)

- Consists of several, partially optional components
- Request Line with *Verb*, Path, and Protocol
- List of HTTP headers, as header:value
- Empty line to end headers
- Optional body message (used, e.g., with POST requests)

```
GET /index.html HTTP/1.0
Host: stonybrook.edu
Cookie: hello=1
```

HTTP GET request

- Purpose: retrieve resource from server
- Should not cause side effects on Web server's state
  - dubbed "idempotent" in W3C standard
  - although it does often cause side effects in practice, due to developers
- Should not carry a message body
- Parameters passed via URL
  - Special characters percent-encoded (hex value of char, e.g., ? = %3F)
  - **Usually logged on server side together with requested file**

```
GET /index.html?name=value%3F HTTP/1.0
Host: stonybrook.edu
```

HTTP POST request

- Purpose: send data to the server
  - for storage or processing
  - should be used for state-changing operations
- Can be combined with GET parameters
- Message body contains data
  - Depending on content-type, percent-encoded or plain

```
POST /index.html?name=value%3F HTTP/1.0
Host: stonybrook.edu
Content-Length: 10
Content-Type: application/json

{"a": "?"}
```

```
POST /index.html?name=value%3F HTTP/1.0
Host: stonybrook.edu
Content-Length: 5
Content-Type: application/x-www-form-urlencoded

a=%3F
```

HTTP Response (since HTTP/1.0)

- Status Line: Protocol, Status Code, and *Status Text*
- List of HTTP headers, as header:value
- Empty line to end headers
- Response Body

```
HTTP/1.0 200 OK
Server: nginx
Content-Type: text/html
Content-Length: 123

<html>…</html>
```

HTTP Response Codes

- 2xx Success
  - 200 OK
  - 206 Partial Content (for range requests)
- 3xx Redirection
  - 301 Moved Permanently (always redirect to new URL)
  - 302 Found (redirect once, don’t store redirect)
  - 304 Not Modified (not changed since last client request, not transferred)
  - 307 Moved Temporarily (only redirect to new URL this time)

HTTP Response Codes

- 4xx Client errors
  - 400 Bad Request (e.g., no carriage return in HTTP request)
  - 401 Unauthorized (used for HTTP authentication)
  - 403 Forbidden
  - 404 Not Found
  - 405 Method Not Allowed
  - 418 I’m a teapot (April Fool’s Joke, see RFC 2324)
- 5xx Server errors
  - 500 Internal Server Error
  - 502 Bad Gateway (e.g., timeout in reverse proxies)

HTTP Evolution over Time: HTTP 1.1 (finalized 1999)

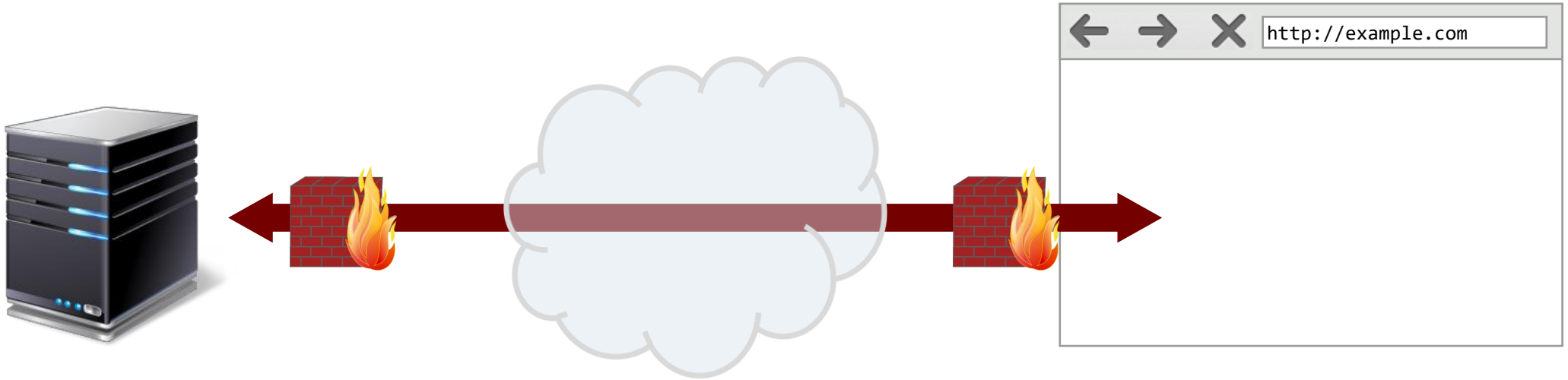
- Requirements
  - Increased resource size requires other transport and caching strategies
  - Fix some ambiguities in the previous protocol versions
  - Assess server's capabilities to handle requests
- Result
  - New methods: PUT (similar to POST), DELETE, TRACE, CONNECT (proxies), OPTIONS
  - Keep-Alive connections
  - Accept-Encoding info for the server
  - Chunked transfers, range transfers
  - Standardized in RFC 2616

```
GET / HTTP/1.1
Host: example.org
```

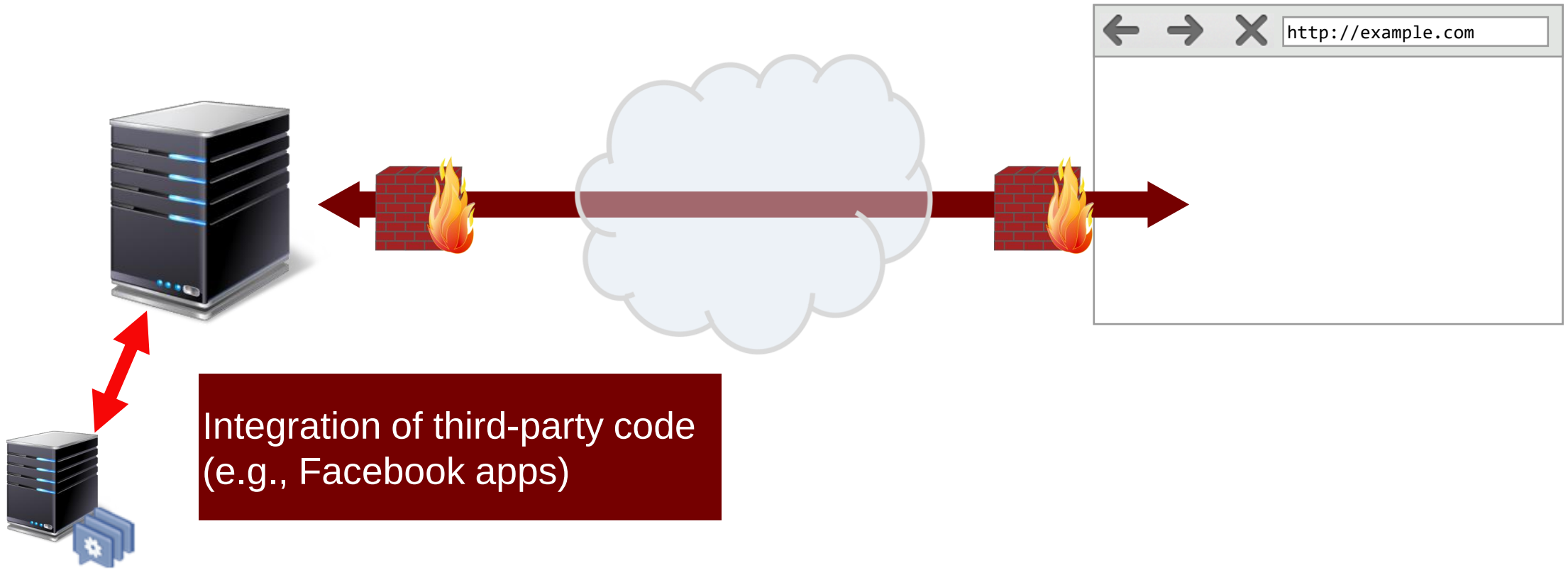
```
HTTP/1.0 200 OK
Transfer-Encoding: chunked

7b
<html>…
0
(connection closed)
```

Threat models

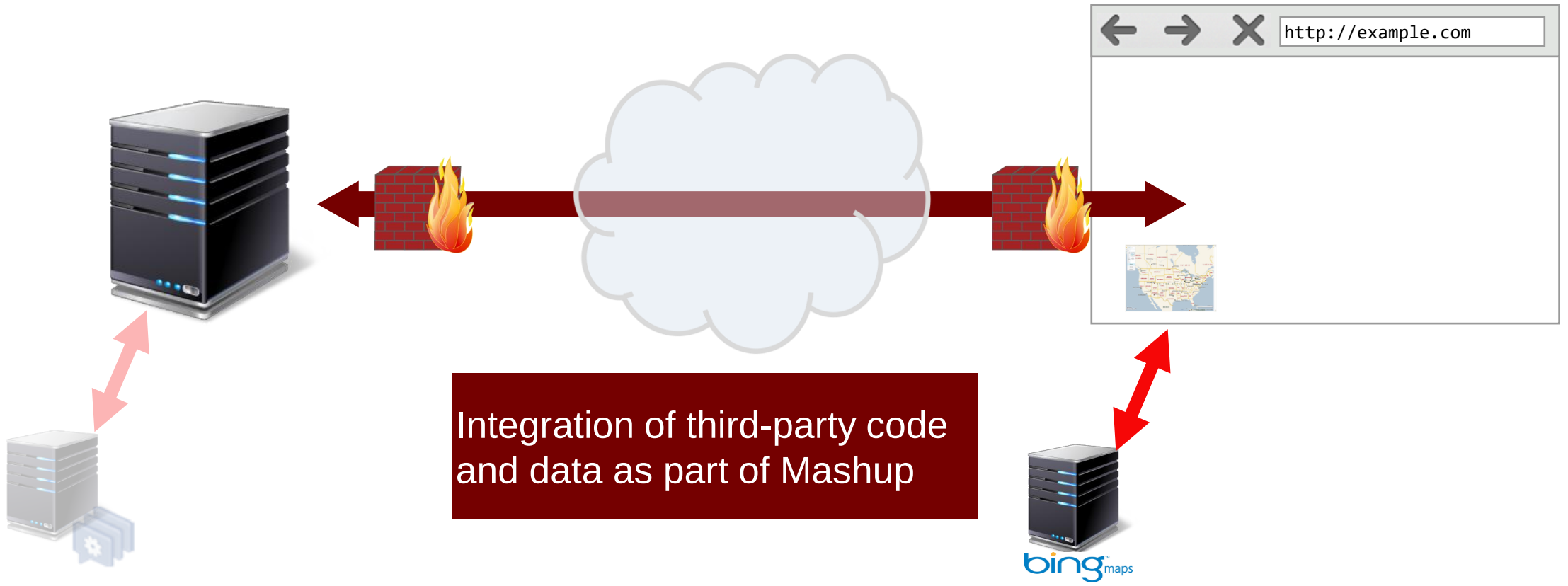
Basic Web Paradigm



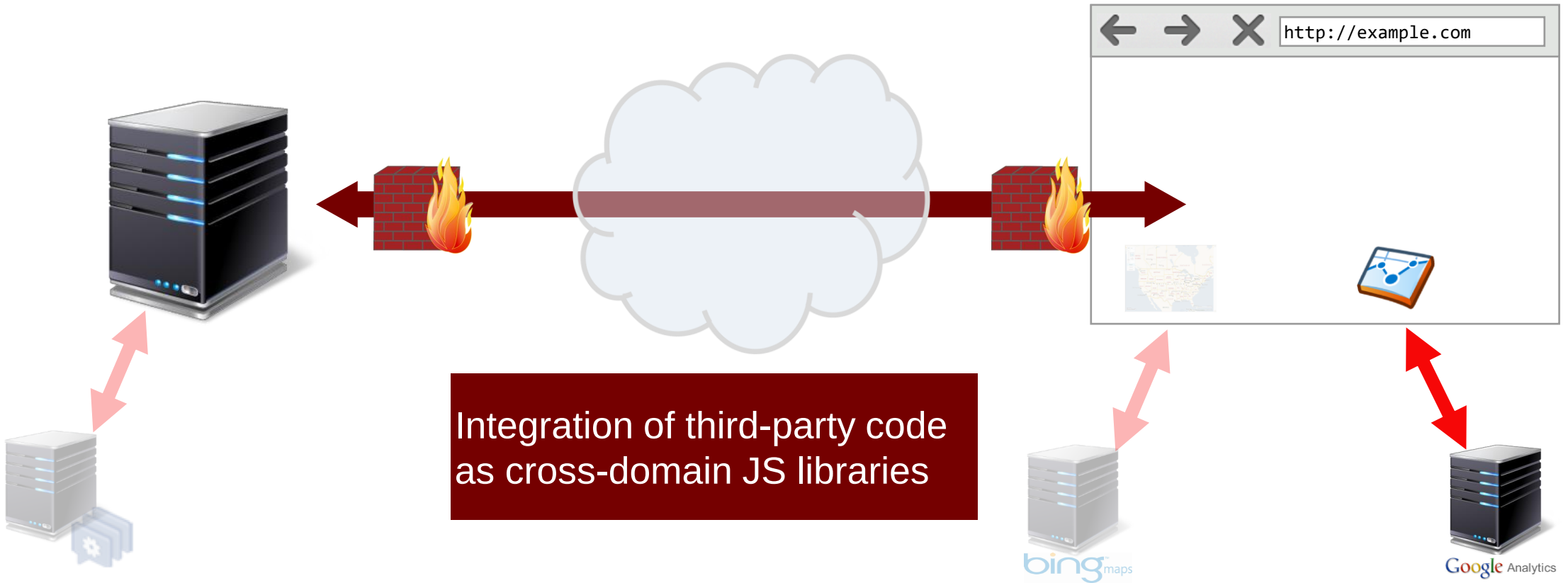
Modern Web Applications



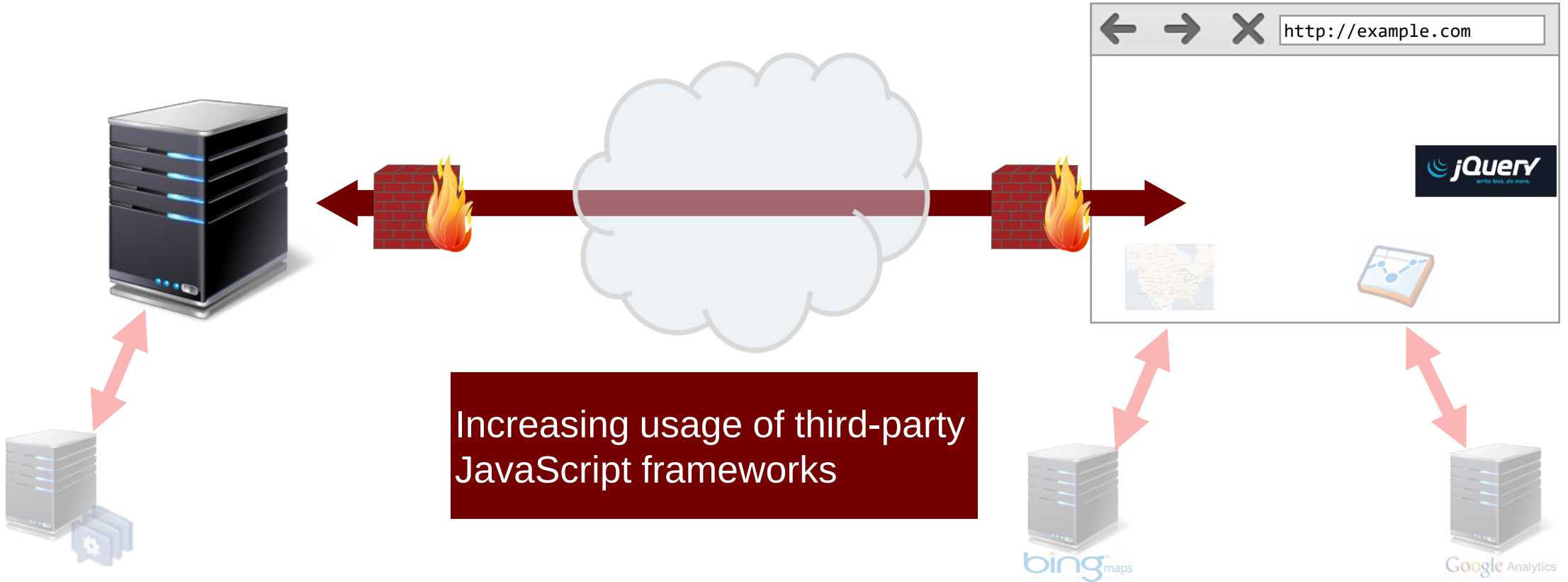
Modern Web Applications



Modern Web Applications

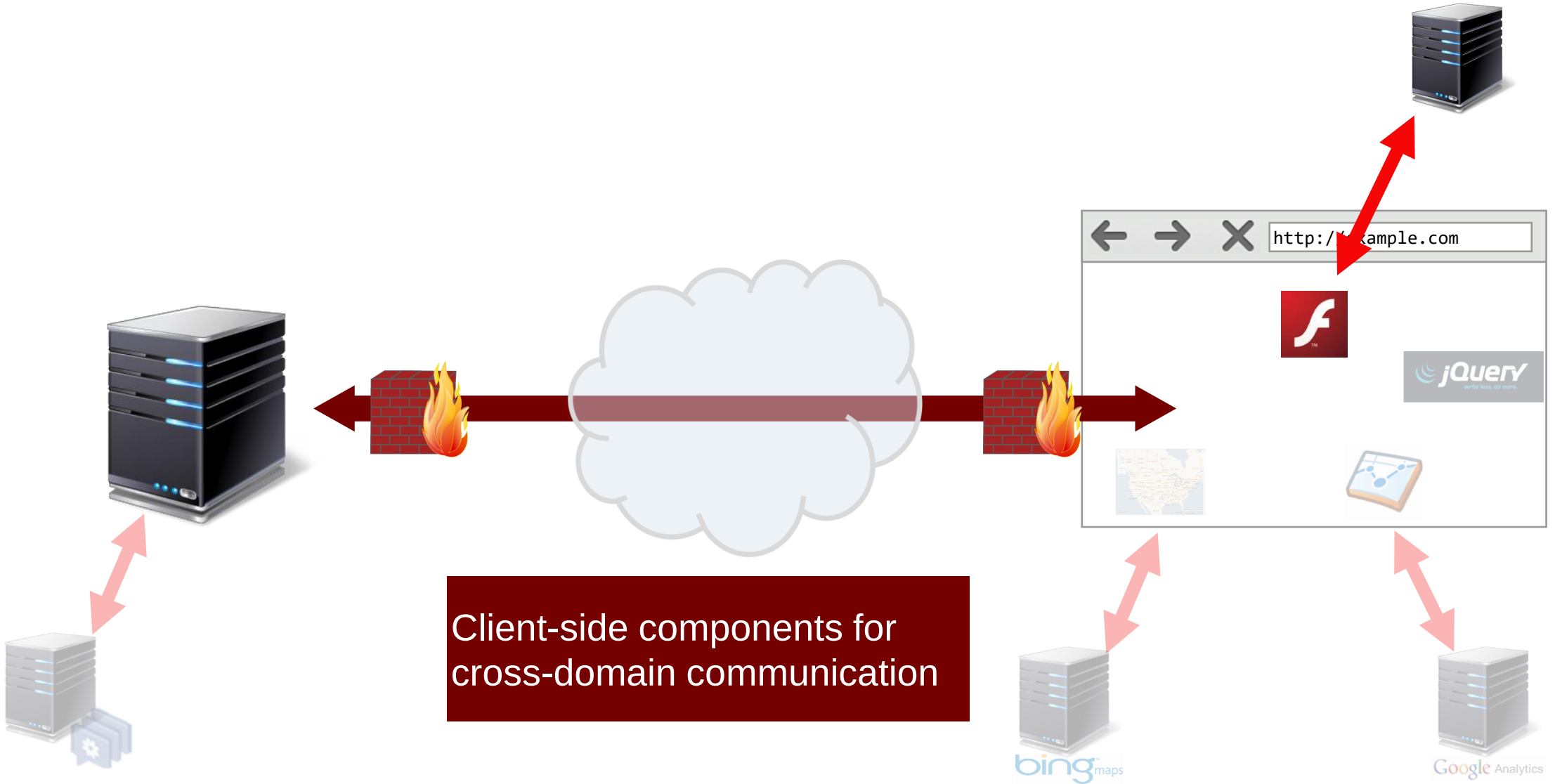


Modern Web Applications



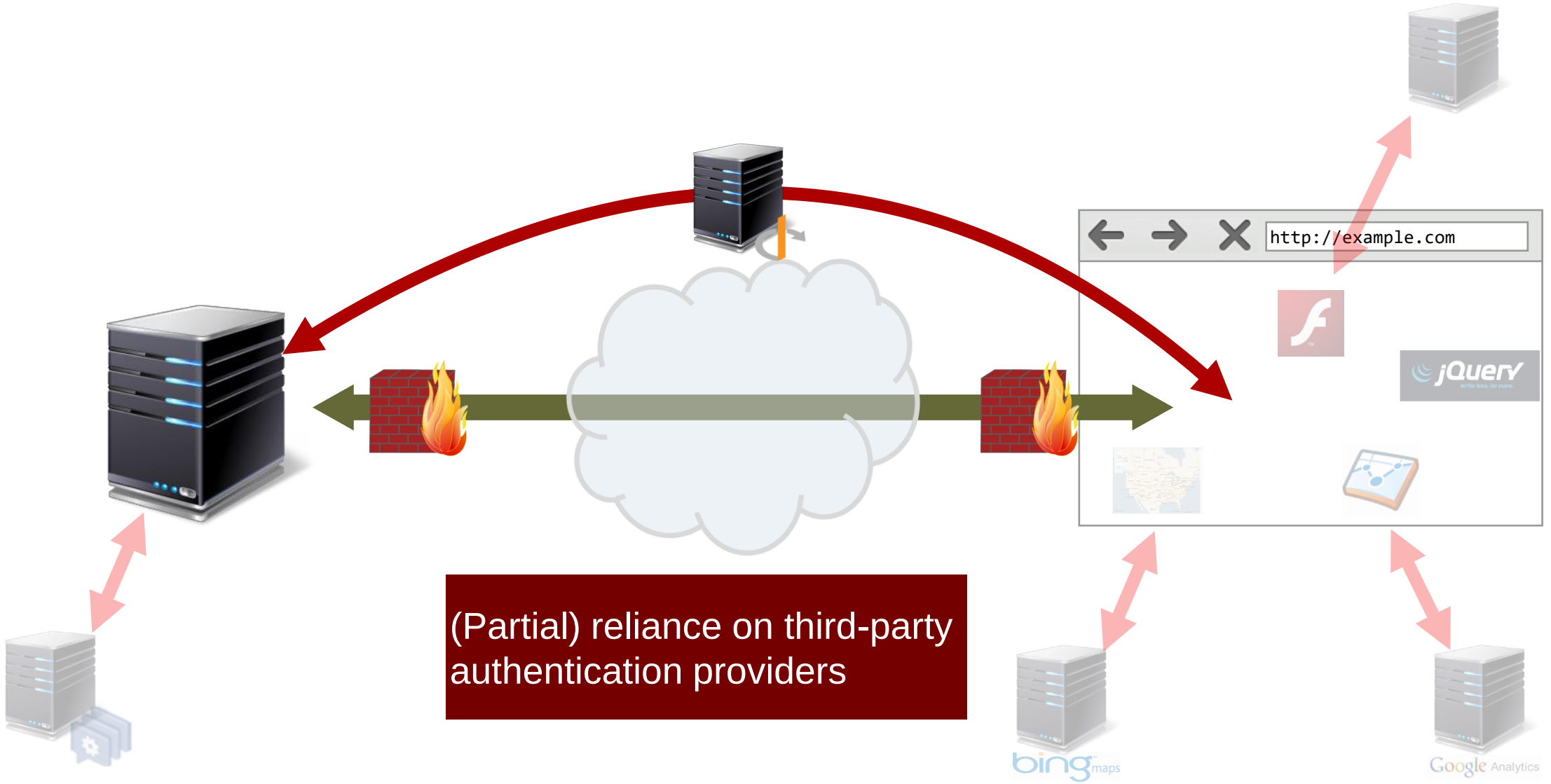
Increasing usage of third-party JavaScript frameworks

Modern Web Applications



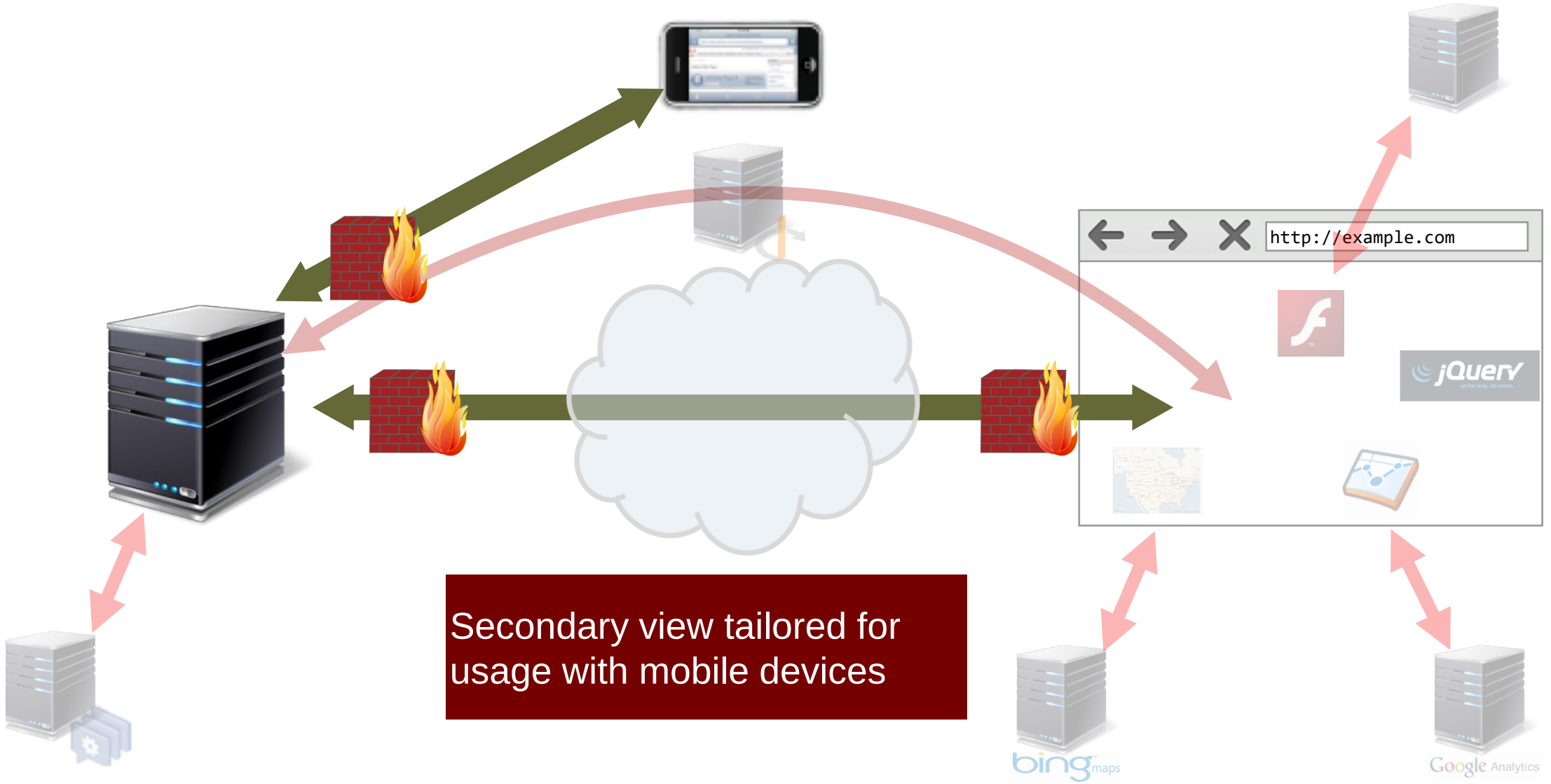
Client-side components for cross-domain communication

Modern Web Applications



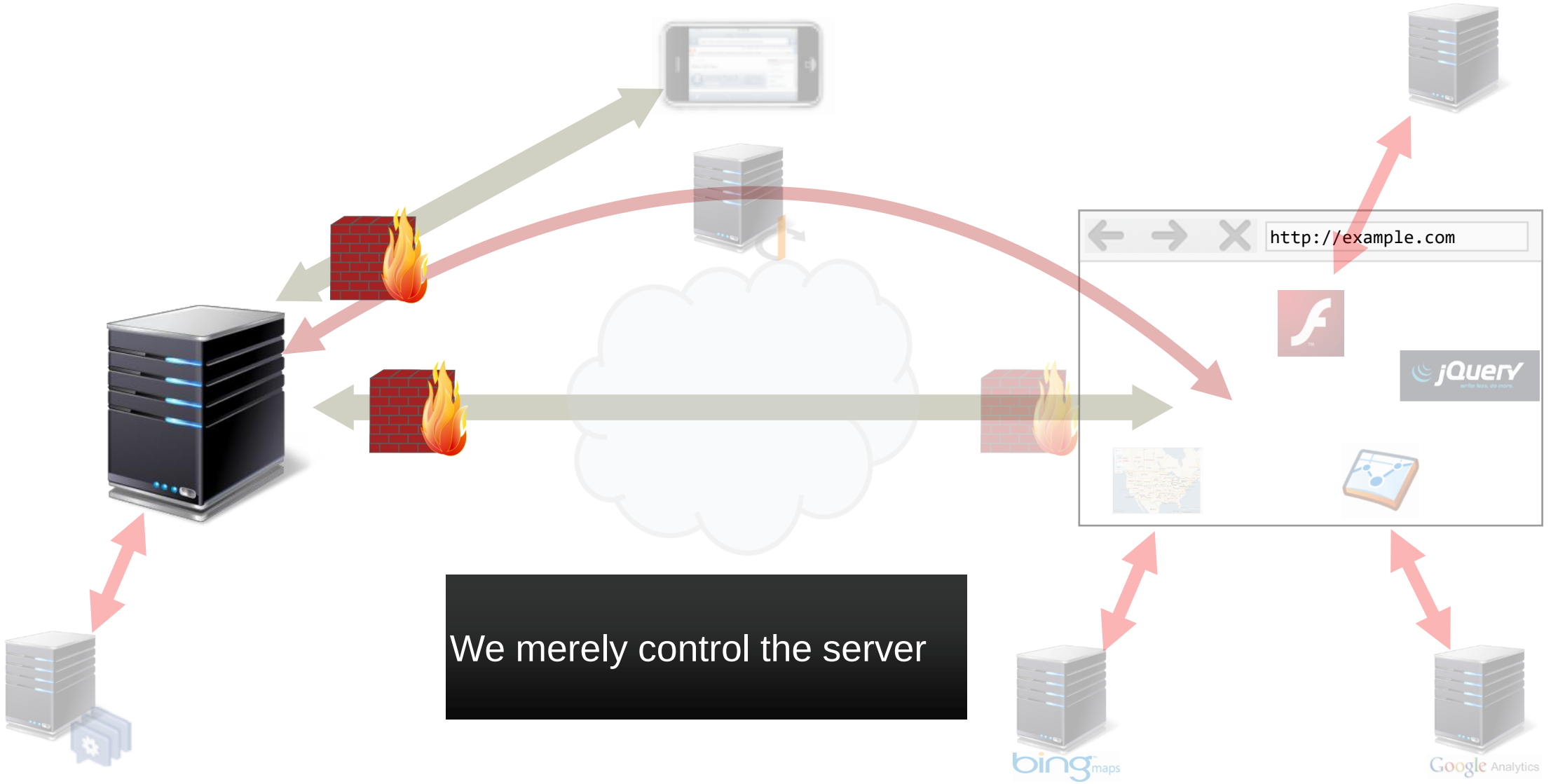
(Partial) reliance on third-party authentication providers

Modern Web Applications

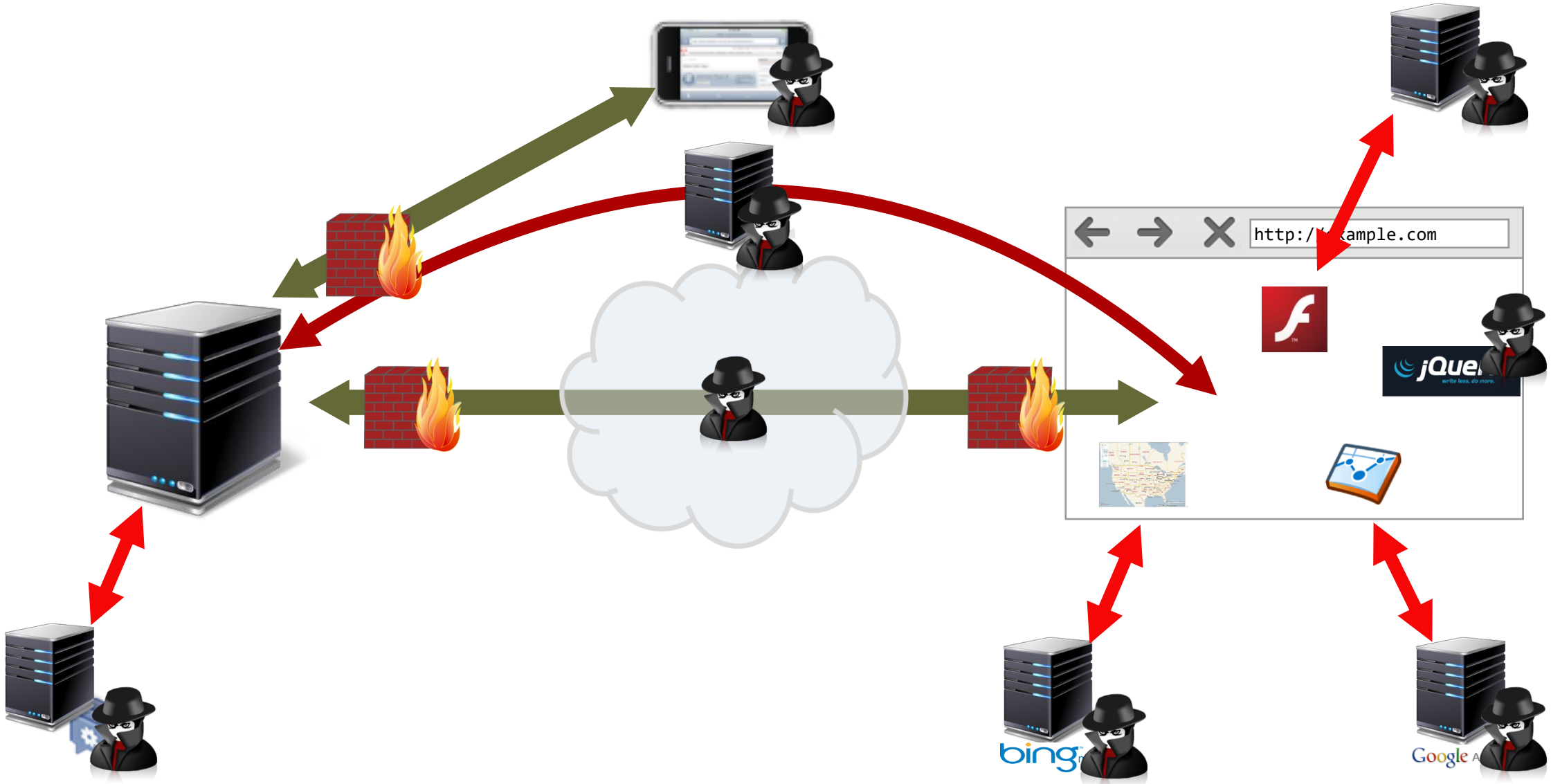


Secondary view tailored for usage with mobile devices

Security Implications



Possible Attackers on the Web



Network Attacker

- Resides somewhere in the communication link between client and server
- Tries to disturb the confidentiality, integrity, and authenticity of the connection
  - Observation of traffic (passive eavesdropper)
  - Fabrication of traffic (e.g., injecting fake packets)
  - Disruption of traffic (e.g., selective dropping of packets)
  - Modification of traffic (e.g., changing unencrypted HTTP traffic)
- "Man in the middle" (MITM)

Remote Attacker

- Can connect to remote system via the network
  - mostly targets the server
- Attempts to compromise the system (server-side attacks)
  - Arbitrary code execution
  - Information exfiltration (e.g., SQL injections)
  - Information modification
  - Denial of Service

Web Attacker

- Attacker specific to Web applications
- "Man in the browser"
  - can create HTTP requests within user's browser
  - can leverage the user's state (e.g., session cookies)
  - Case of "confused deputy"
- Examples
  - Cross-Site Scripting attacker: can execute arbitrary JavaScript in authenticated user's context
  - Cross-Site Request Forgery attacker: can force user's browser to execute certain operations on vulnerable site

Social Engineering Attacker

- No real technical capabilities
  - Abusing users rather than software vulnerabilities
- Can lure victim to perform certain tasks
  - Clickjacking
- May use technical measures to ease his task
  - Unicode URLs to easily fake
  - Use well-known icons to suggest "secure" sites

Adding State to HTTP

- Recall: no inherent state in HTTP
  - server does not keep any state after TCP connection is closed
- For static content sites, no problem
  - developing "applications" is impossible though
  - e.g., shopping cart on Amazon
- Need to introduce state in HTTP
  - in the form of "sessions"

Option 1: HTTP Authentication

- Associate user with state on server
  - unclear when the "sessions" ends
- Authentication done by Web server
  - not by application itself, impossible to use in multi-tenant architectures
- Implements "pulling" of credentials
  - User: "Please give me resource X"
  - Server: "No, please tell me who you are"
  - User: "Ok, I am *alice* and my password is *nu7^yjUtasw* "
- Logout non-trivial
  - browser always sends along authentication header

Cookie directives

- `HttpOnly`, disallows access from JavaScript via document.cookie
- `Secure`, only transmit cookie over secure connection
  - Can only be set from HTTPS connections
- `SameSite=None/Strict/Lax`
  - `Strict:` do not transmit cookies on **any** cross-site request
  - `Lax:` only transmit cookies on "safe" top-level navigation
    - Safe methods (per RFC 7231): GET, HEAD, OPTIONS, (TRACE)
  - `None:` explicit opt-in for cross-site requests, requires `Secure`
  - Browsers will default to `SameSite=Lax` soon (Chrome already does so, FF and Edge warn)

JavaScript in Web documents

- JavaScript can be included in script tags or event handlers
  - `<script>var hello="world";</script>`
  - `<script src="http://hello.world"></script>`
  - `<a onclick='var hello="world";'>Click me</a>`
- Each script tag or event handler is separate parsing block
  - code not executed when parsing error occurs
  - other scripts' execution is not interrupted
- Rendering of document stops until script is executed
  - especially important when HTML is written by JavaScript
- **All scripts run in same global space (of including page)**

JavaScript Variable Scoping

- Variables without *var* keyword always in global scope
- Variables with *var* keyword as specified in current scope (function-level)
  - Gotcha: in top-level script code, that is the global scope
- Public members of object use `this` keyword, private members `var`

```
function Container(param) {
    var member = param;
}



var a = new Container(1);
a.member
// > undefined
```

```
function Container(param) {
    this.member = param;
}



var a = new Container(1);
a.member
// > 1
```

```
function Container(param) {
 var member = param;
 this.getmember = function() {
  return member; }
}

var a = new Container(1);
a.getmember()
// > 1
```

(Almost) everything in JavaScript can be overwritten/deleted

```
eval("var a='hello'")
a
// > "hello"

eval = alert;

eval("var a='hello'");
// opens alert box
```

```
var oAlert = alert;
alert = function(x) {
    console.log(x);
    oAlert(x);
}
alert(1);
// log 1 to console
// opens alert box
```

```
var oAlert = alert;
delete alert;

alert(1);
// Uncaught ReferenceError: alert is not defined

oAlert(1)
// opens alert box
```

Document Object Model (DOM) and Browser APIs

- Exposed to JavaScript through global objects
  - `document`: Access to the document (e.g., cookies, head/body)
  - `navigator`: Information about the browser (e.g., UA, plugins)
  - `screen`: Information about the screen (e.g., dimension, color depth)
  - `location`: Access to the URL (read and modify)
  - `history`: Navigation
- Global object is called `window`, current object is `self`

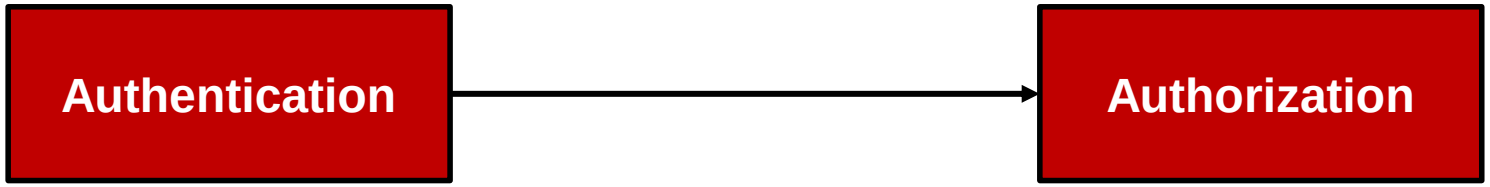
```
a = "Hello";
a === window.a;
> true
```

```
document.location === location;
> true
```

```
self === window;
> true
```

Password-based Authentication

- Passwords are key to the process of **authentication**
  - Authentication is at the heart of security



How do I know you are, who you say your are?

Are you allowed to do the thing you are asking?

Password-based Authentication

User has a secret password.

System checks it to authenticate the user.

- How is the password communicated?
  - Eavesdropping risk (We will see later how crypto can be used)
- How is the password stored?
  - In the clear? Encrypted? Hashed?
- How does the system check the password?
- How easy is it to guess the password?
  - Easy-to-remember passwords tend to be easy to guess

Attackers

- What is the threat model?
  - Online attacker
    - Tries to login to a service by iteratively trying passwords and looking whether he was successful
  - Offline attacker
    - Stole password database and tries to recover the, hopefully protected, passwords
      - Also known as a “dictionary attack”
  - Against one user
  - Against all/any user

How do attackers use passwords?

- Once a database of credentials is leaked, attackers can use them in multiple ways
  - Extract emails and usernames
    - Chances are that users are reusing the same username/email address in other unrelated services
  - Learn what are the most common passwords that most users use
  - Learn what are the passwords that specific users use

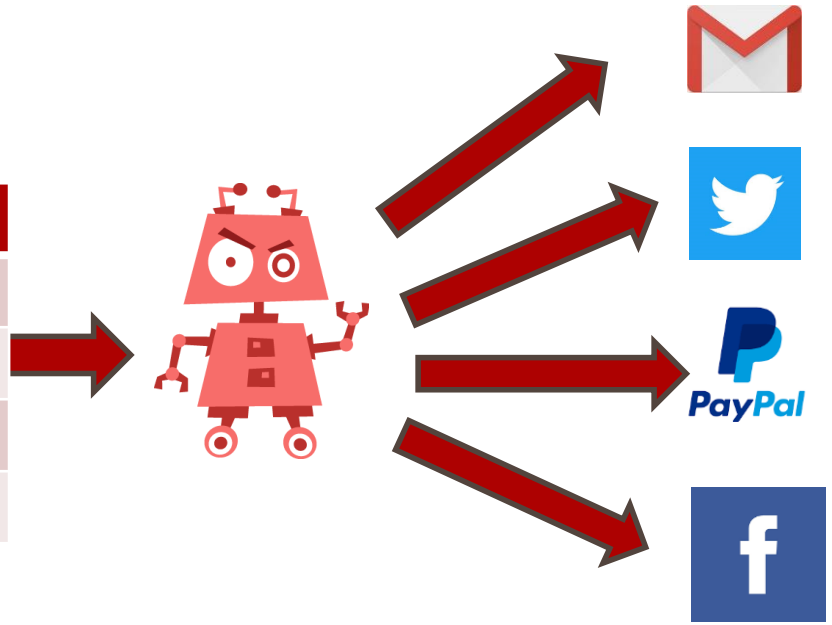
Username	Password
alice@gmail.com	ilovedogs
bob@yahoo.com	Password!
eve@outlook.com	1q2w3e4r
john@stonybrook.edu	g@rfield1

Credential stuffing

- Attackers build programs that try these credentials against other services
  - These programs act like regular users trying to log in
  - Attackers bet on users reusing their passwords

Username	Password
alice@gmail.com	ilovedogs
bob@yahoo.com	Password!
eve@outlook.com	1q2w3e4r
john@stonybrook.edu	g@rfield1

supercutecats.com



Sample Cryptographic hash functions

Name	Year of release	Digest size (output size)
MD5 (Media Digest 5)	1992	128-bit
SHA-1 (Secure Hash Algorithm 1)	1995	160-bit
SHA-256 (Part of the SHA-2 family)	2001	256-bit

MD5(“helloworld”) = d73b04b0e696b0945283defa3eee4538
SHA-1(“helloworld”) = e7509a8c032f3bc2a8df1df476f8ef03436185fa
SHA-256(“helloworld”) = 8cd07f3a5ff98f2a78cfc366c13fb123eb8d29c1ca37c79df190425d5b9e424d

Salting

- Instead of just hashing the user's password, hash the user's password when concatenated with a per-user random value

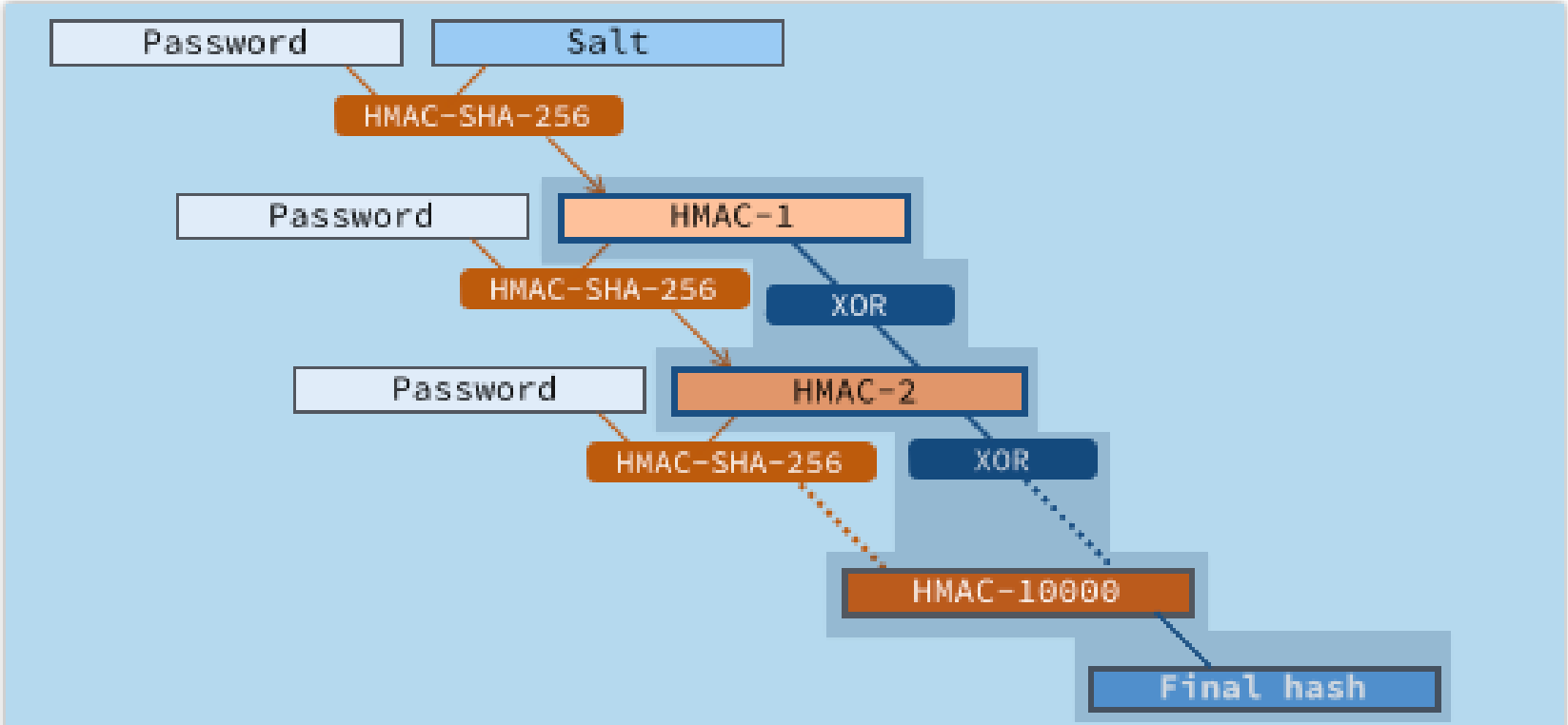
SHA256("mysecretpassword")

Username	Password
nick	94AEFB8BE78B2B7C344D11D 1BA8A79EF087ECEB19150881 F69460B8772753263

SHA256("199654mysecretpassword")

Username	Salt	Password
nick	199654	1C8622F514E7BB8B86210FE8 83D48CC55C5BEDA849DAF74 6AFFFDEC757952F77

PBKDF2 + HMAC-SHA-256



PBKDF2 using XOR to combine 10,000 successive HMAC-SHA-256 outputs into a final hash

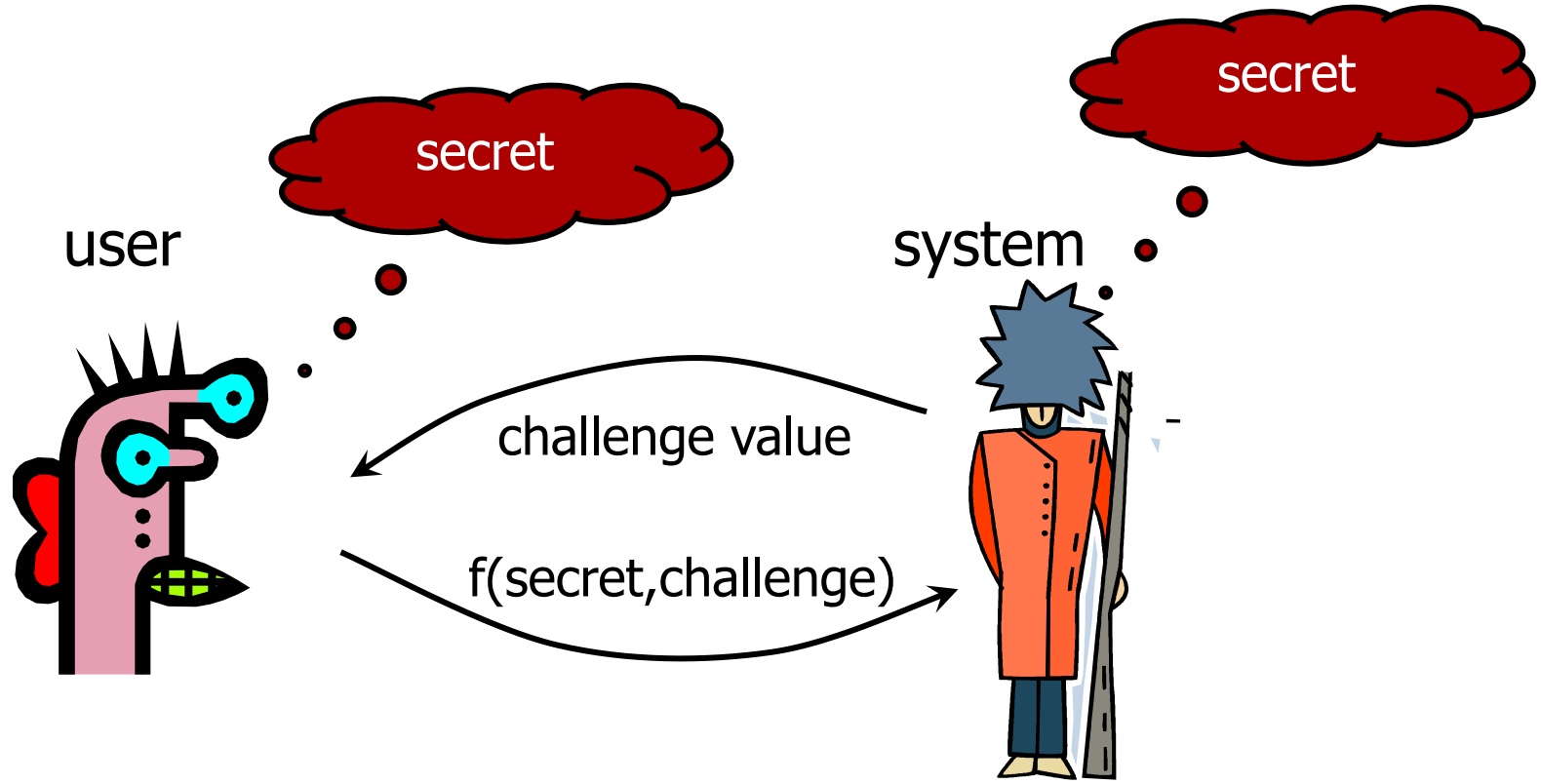
Image source: https://nakedsecurity.sophos.com/2013/11/20/serious-security-how-to-store-your-users-passwords-safely/

Password Managers

- One place where all your passwords are stored
  - This place is protected with one master password
  - Flavors:
    - Online versus Offline (e.g. LastPass versus KeePass)

- Benefits
  - No need to remember any more passwords (other than the master phrase)
  - Unique password per website (no more password reuse)
  - Most password managers also have their own password generators to automatically create strong passwords
- Disadvantages
  - Single-point of failure
    - This can be easily mitigated by storing multiple copies of the database
  - Lock yourself out
    - If you forget your master password, there is no way to recover passwords
  - Cannot authenticate to services if you don’t have access to the password manager

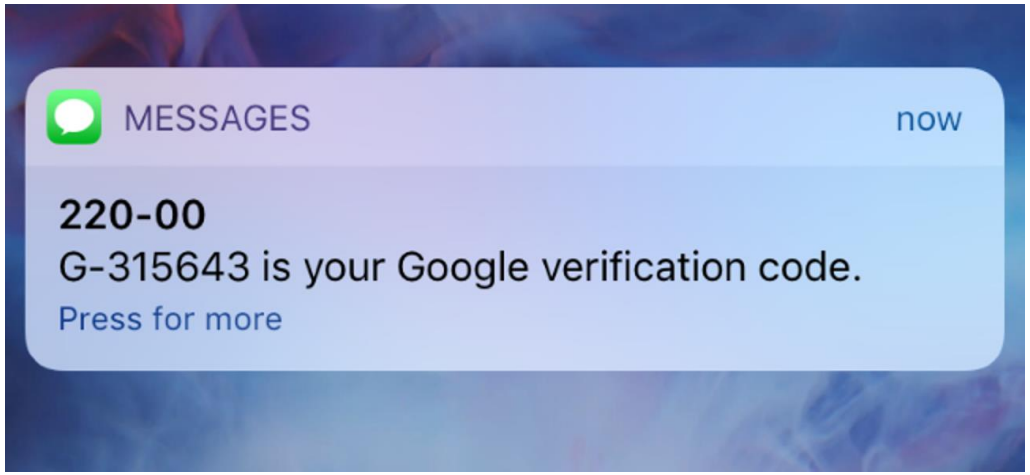
Challenge-Response



Why is this better than the password over a network?

Something you have - SMS

- Text messages (SMS) as a 2-factor authentication method is falling out of favor.
  - NIST has mentioned that it is deprecated and when possible, services should use hardware tokens or smartphone apps to deliver codes
- Reasons
  - Too many incidents of attackers social engineering phone companies into sending them SIM cards because the real owner “lost their phone”
  - Telcos in authoritarian governments can cooperate with their governments
  - Phone networks and their protocols are not exactly the most secure ones



Time-based One Time Passwords (TOTP) apps

- TOTP(K,C) = Truncate(HMAC-SHA-1(K,T))
  - K: Shared secret key
    - One copy in your app, one copy on the server
  - T: Current time (in specific steps)
    - Default time step of 30 seconds

- Resynchronization options
  - Allow for client-clocks being slightly slower / slightly faster
  - Potentially ask for additional codes

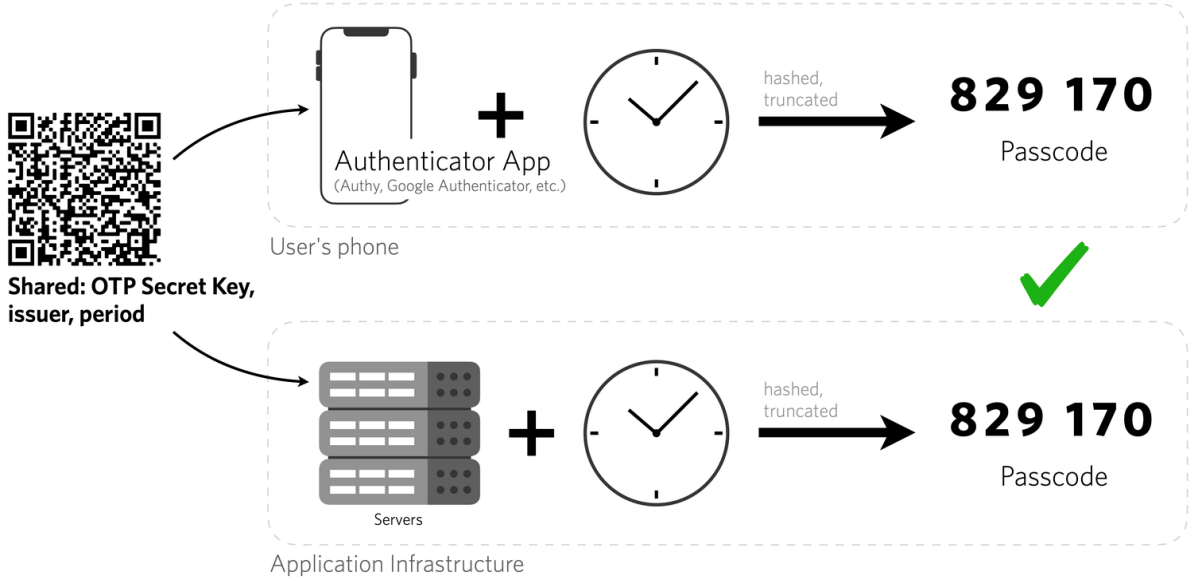
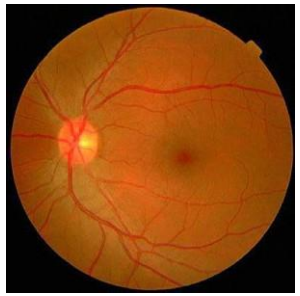
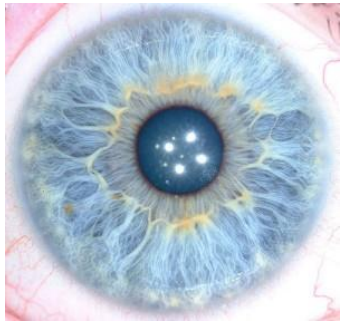


Image:https://www.twilio.com/docs/glossary/totp

Something you are

- **Biometrics**
  - Fingerprints
  - Palms
  - Face
  - Iris/Retina scanning
  - Voice
  - How you walk? How you type? How you swipe?
    - Research in continuous authentication
- Benefits
  - Nothing to remember
  - Passive (nothing to type, always carrying them around)
  - Can’t share
  - Can be fairly unique





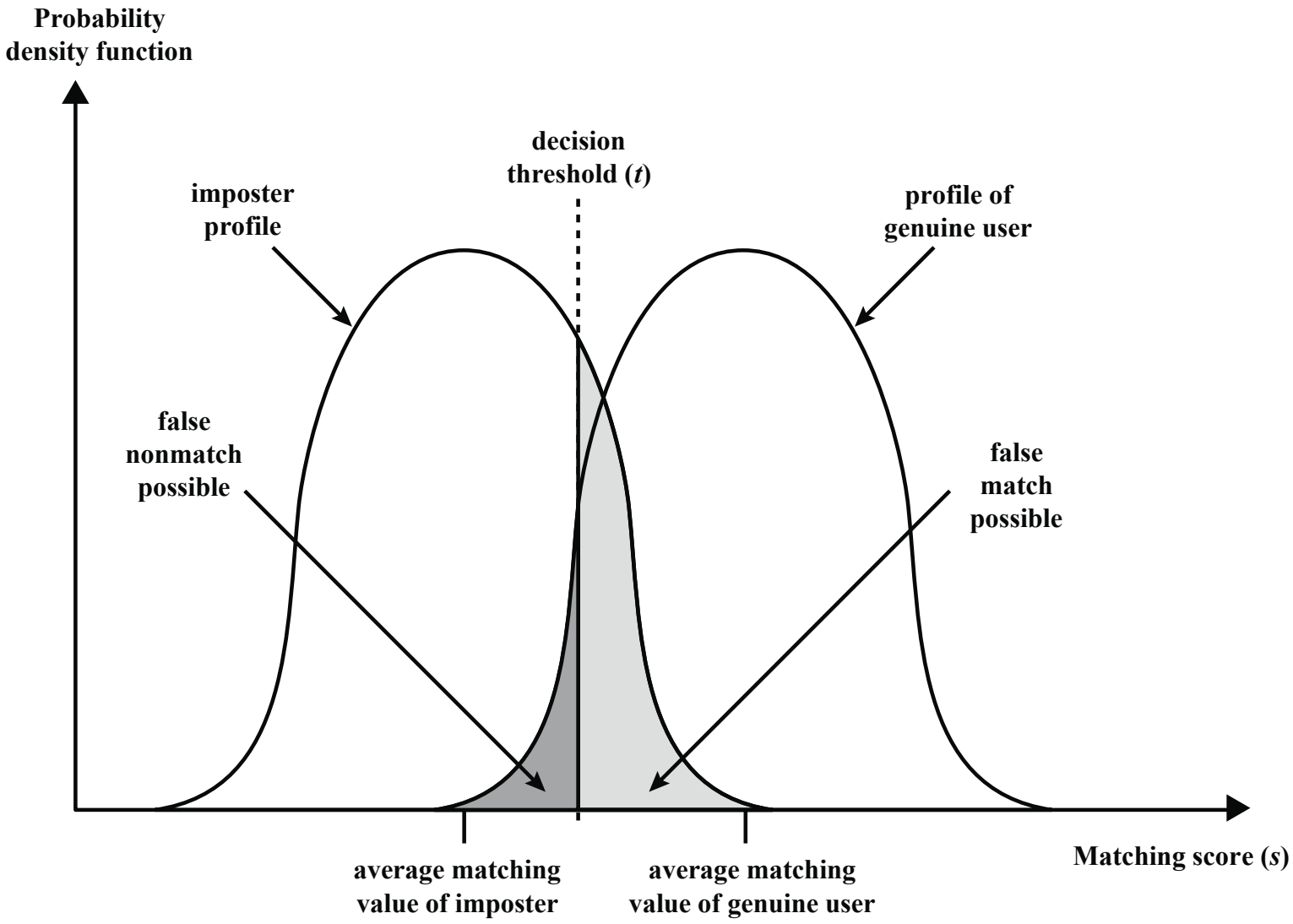


Figure 3.9 Profiles of a Biometric Characteristic of an Imposter and an Authorized Users In this depiction, the comparison between presented feature and a reference feature is reduced to a single numeric value. If the input value (*s*) is greater than a preassigned threshold (*t*), a match is declared.

***Image Source**: Computer Security: Principles and Practice*

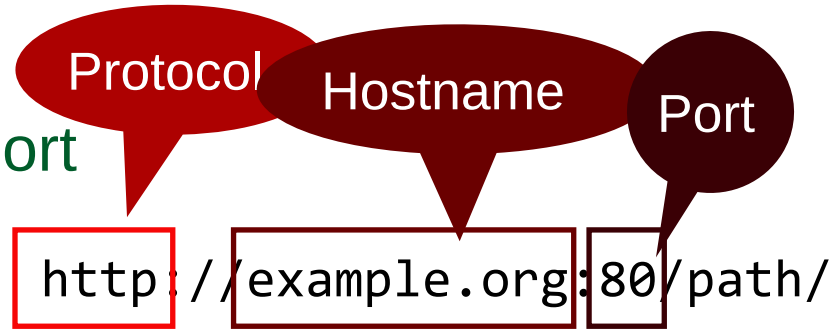
Communication between different websites

The Same-Origin Policy for JavaScript

- Most basic access control policy
  - controls how active content can access resources
- Same-Origin Policy for JavaScript for three actions
  - Script access to other document in same browser
    - frames/iframes
    - (popup) windows
  - Script access to application-specific local state
    - cookies, Web Storage, or IndexedDB
  - Explicit HTTP requests to other hosts
    - XMLHttpRequest

The Same-Origin Policy for JavaScript

- Only allows access if origins match
  - Origin defined by protocol, hostname, and port

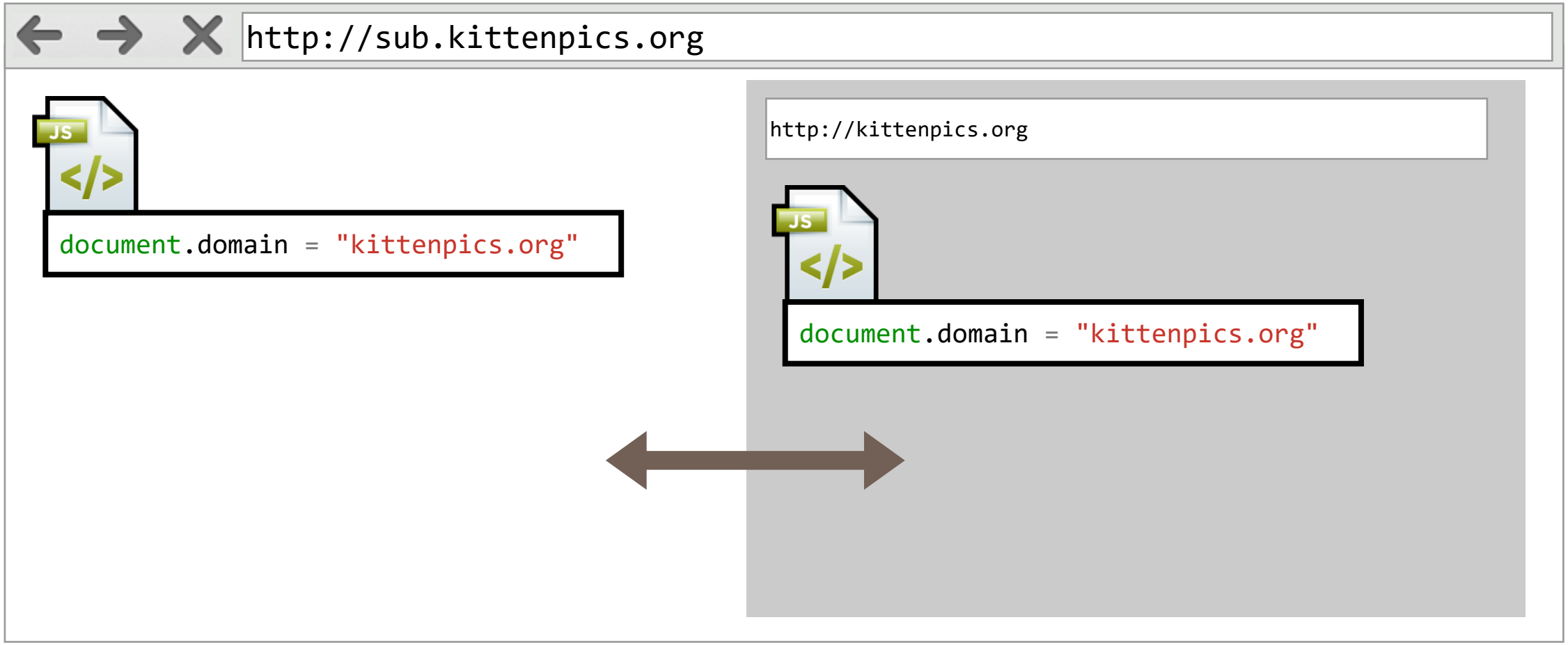


Originating document	Accessed document	Non-IE Browser	Internet Explorer
http://example.org/a	http://example.org/b	✓	✓
http://example.org	http://www.example.org	⊘	⊘
http://example.org	https://example.org	⊘	⊘
http://example.org	http://example.org:81	⊘	✓

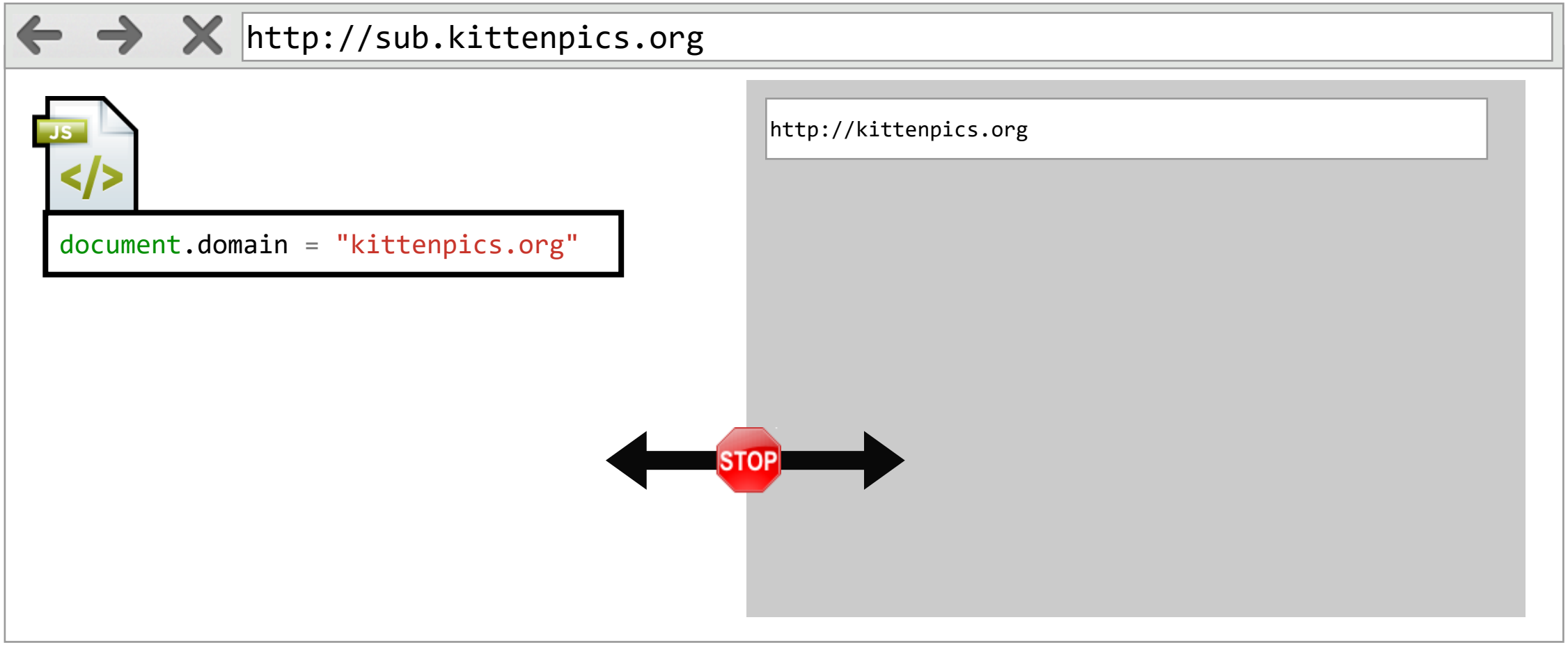
Domain Relaxation

- Two sub-domains of a common parent domain want to communicate
  - Notably: can overwrite different port!
- Browsers allow setting `document.domain` property
  - Can only be set to valid suffix including parent domain
  - `test.example.org -> example.org` ok
  - `example.org -> org` forbidden
- When first introduced, relaxation of single sub-domain was sufficient
- Nowadays: both (sub-)domains must explicitly set `document.domain`

Domain Relaxation



Domain Relaxation

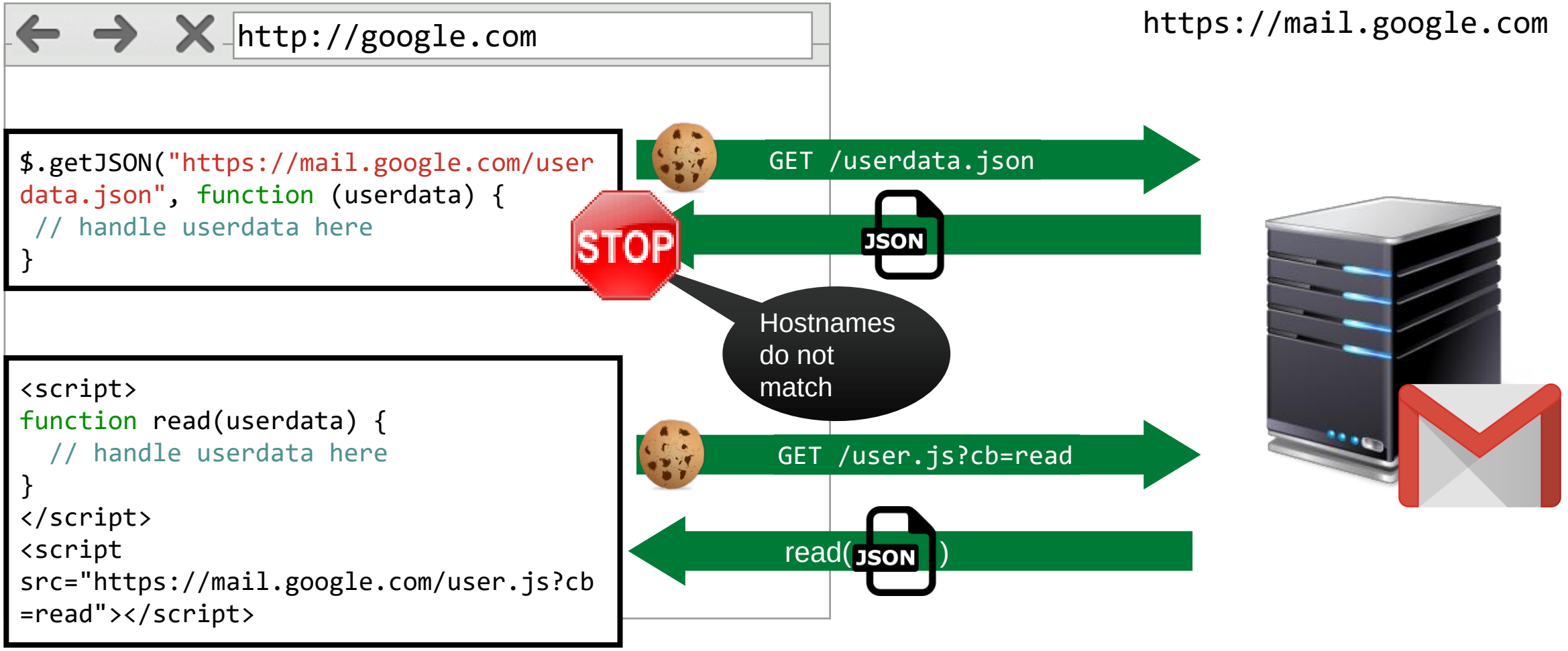


Cross-Origin Communication

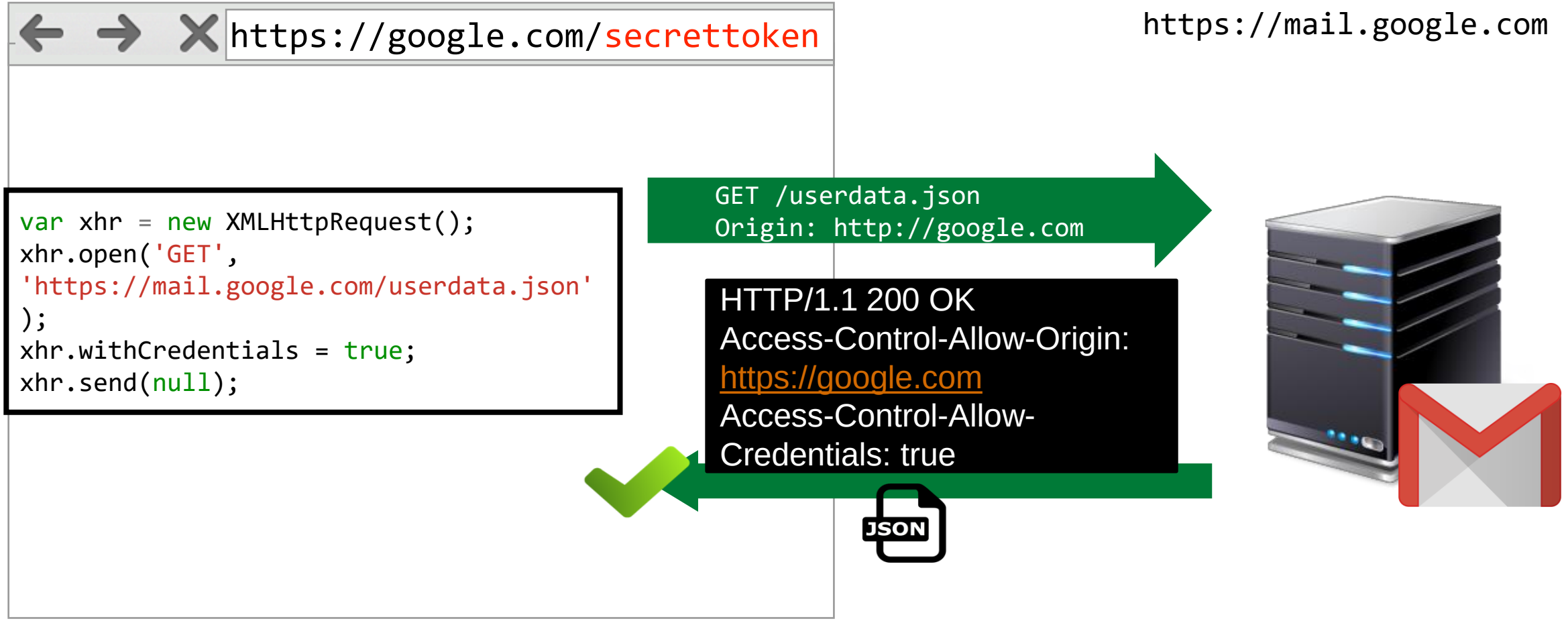
Cross-Domain Communication: JSONP

- Recall Web model: may include resources from remote origins
  - access from JavaScript to cross-domain resources is restricted though
- Weird case: scripts
  - can be included from remote origin
  - execute in **including** origin (side effects observable on global scope)
  - source code not accessible from including page
- JSONP ("JSON with Padding") (ab)uses this
  - callback function as parameter
  - creates script code dynamically

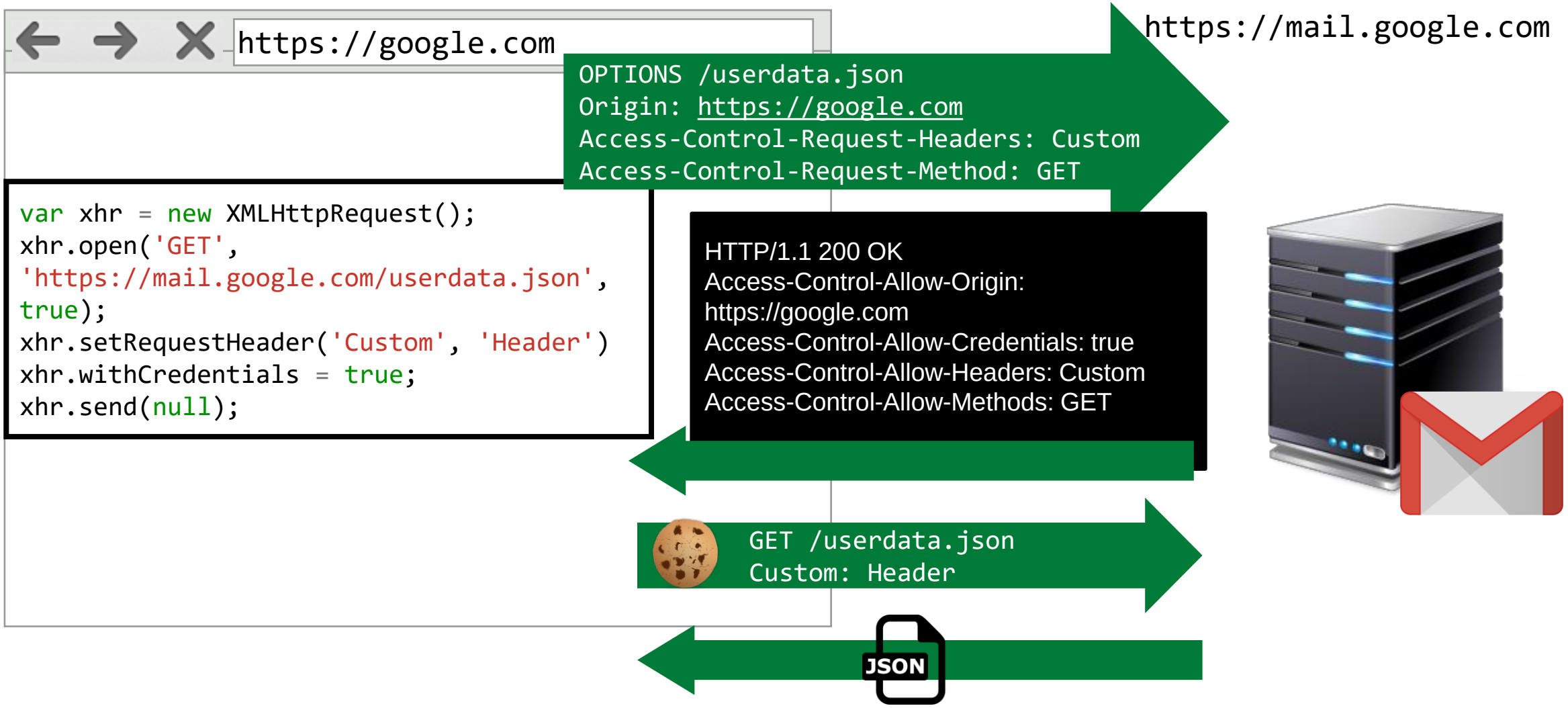
JSONP Concept



CORS Concept (simple request)



CORS Preflight requests



postMessage Concept

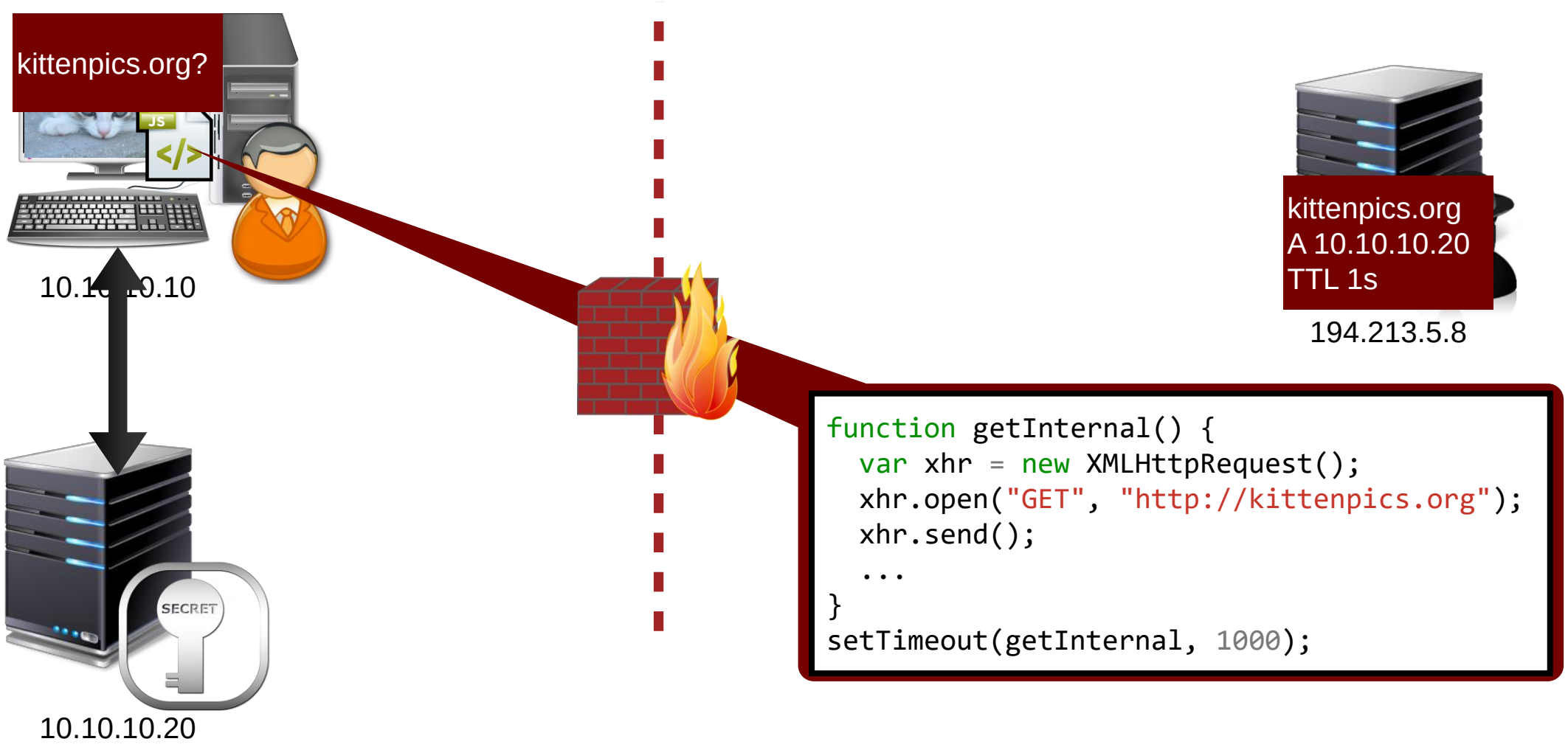

```
window.addEventListener("message",
    receiveMessage);

function receiveMessage(event)
{
  if (event.origin !== "http://main.site")
    return;
  var message = event.data;
  process(message);
}
```

Bypassing SOP

DNS Rebinding - Concept


```
function getInternal() {
  var xhr = new XMLHttpRequest();
  xhr.open("GET", "http://kittenpics.org");
  xhr.send();
  ...
}
setTimeout(getInternal, 1000);
```

DNS Rebinding - Concept



Dimensions of Cross-Site Scripting

	Server	Client
Reflected	`echo "Welcome ". $_GET["name"];`	`document.write("Welcome " + location.hash.slice(1));`

Persistent	`mysql_query("INSERT INTO posts ...");`
	`// ..`
	`$res = mysql_query("SELECT * FROM posts");`
	`while ($row = mysql_fetch_array($res)) {`
	`  print $res[0];`
	`}`
		`localStorage.setItem("name", location.hash.slice(1));`
		`// ..`
		`document.write("Welcome " + localStorage.getItem("name"));`

Preventing Server-Side Cross-Site Scripting

- Option 1: Input Validation/Sanitization
- Check input against list of allowed/expected characters
  - Is this a number? Is this an email?
- Can only be considered first line of defense
  - Usage of data might not be known at that point
  - Hard to get right, for the general case
- (bad) alternative: removing unwanted elements
  - Known as blacklisting/blocklisting
  - e.g., all script tags
  - simple replace does not suffice: `<scr<script>ipt>`

```
foreach ($_REQUEST as $key => value) {
  $_REQUEST[$key] = preg_replace("[^0-9a-zA-Z]",
                                 "", $value);
}
// ....
$username = base64_decode($_REQUEST["user"]);
```

Preventing Server-Side Cross-Site Scripting

- Option 2: Output Encoding
- When using the data, encode it
  - depending on context, different encoders might be necessary

HTML Encoding **PHP**

```php
<?php
 function noHTML($input, $encoding = 'UTF-8'){
  return htmlentities($input, ENT_QUOTES | ENT_HTML5, $encoding)
 }
 ...
 echo '<div> You searched for ' . noHTML($_GET['q']) . ' </div>';
?>
```

Preventing Server-Side Cross-Site Scripting

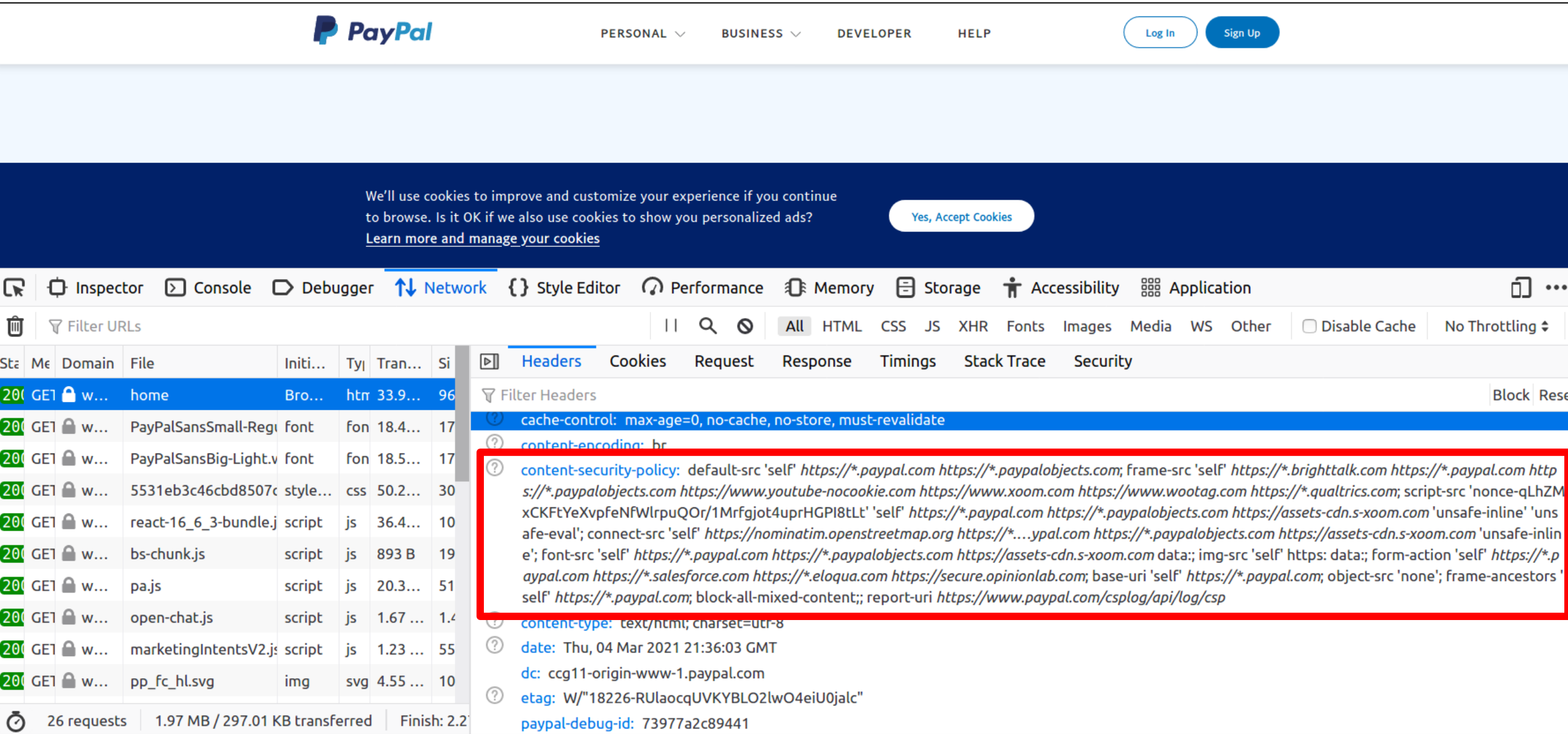
- Option 2: Output Encoding
- When using the data, encode it
  - depending on context, different encoders might be necessary

URL Encoding

PHP

```php
<?php

function sanitizeParam(){
 return urlencode($param);
}

echo '<a href="https://example.com/article?input="' . sanitizeParam($_GET['q']) . '">...</a>';

?>
```

Example policy on paypal.com



CSP Level 1 - Controlling scripting resources

- `script-src` directive
  - Specifically controls where scripts can be loaded from
  - **If provided, inline scripts and eval will not be allowed**

- Many different ways to control sources
  - **'none' -** no scripts can be included from any host
  - **'self'** - only own origin
  - **https://domain.com/specificscript.js**
  - **https://*.domain.com** - any subdomain of domain.com, any script on them
  - **https**: - any origin delivered via HTTPS
  - **'unsafe-inline' / 'unsafe-eval'** - reenables inline handlers and eval

CSP Level 1 - Controlling additional resources

- `img-src, style-src, font-src, object-src, media-src`
  - Controls non-scripting resources: images, CSS, fonts, objects, audio/video
- `frame-src`
  - Controls from which origins frames may be added to a page
- `connect-src`
  - Controls XMLHttpRequest, WebSockets (and other) connection targets
- `default-src`
  - Serves as fallback for all fetch directives (all of the above)
    - Only used when specific directive is absent

Content Security Policy (CSP)

- XSS boils down to execution of attacker-created script in vulnerable Web site
  - Browser cannot differentiate between intended and unintended scripts
- Proposed mitigation: Content Security Policy
  - explicitly **allow resources** which are trusted by the developer
  - disallow dangerous JavaScript constructs like eval or event handlers
  - delivered as HTTP header or in meta element in page (only subset of directives supported)
  - **enforced by the browser** (**all policies** must be satisfied)
- First candidate recommendation in 2012, currently at Level 3
- Important: does not stop XSS, tries to mitigate its effects
  - similar to, e.g., the NX bit for stacks on x86/x64

CSP Level 2 - Allowed hosts with Nonces or Hashes

```
script-src 'self' https://cdn.example.org
'nonce-d90e0153c074f6c3fcf53'
'sha256-5bf5c8f91b8c6adde74da363ac497d5ac19e4595fe39cbdda22cec8445d3814c'
```

```
<script>
alert('My hash is correct');
</script>
```

```
<script>
 alert('My hash is correct');
</script>
```

SHA256 matches value of CSP header

SHA256 does not match

CSP Level 2 - Allowed hosts with Nonces or Hashes

```
script-src 'self' https://cdn.example.org
'nonce-d90e0153c074f6c3fcf53'
'sha256-5bf5c8f91b8c6adde74da363ac497d5ac19e4595fe39cbdda22cec8445d3814c'
```

```
<script nonce=“d90e0153c074f6c3fcf53”>
alert("It’s all good");
</script>
```

```
<script nonce=“nocluehackplz”>
 alert(‘I will not work');
</script>
```

Script nonce matches CSP header

Script nonce does not match CSP header

End of recap