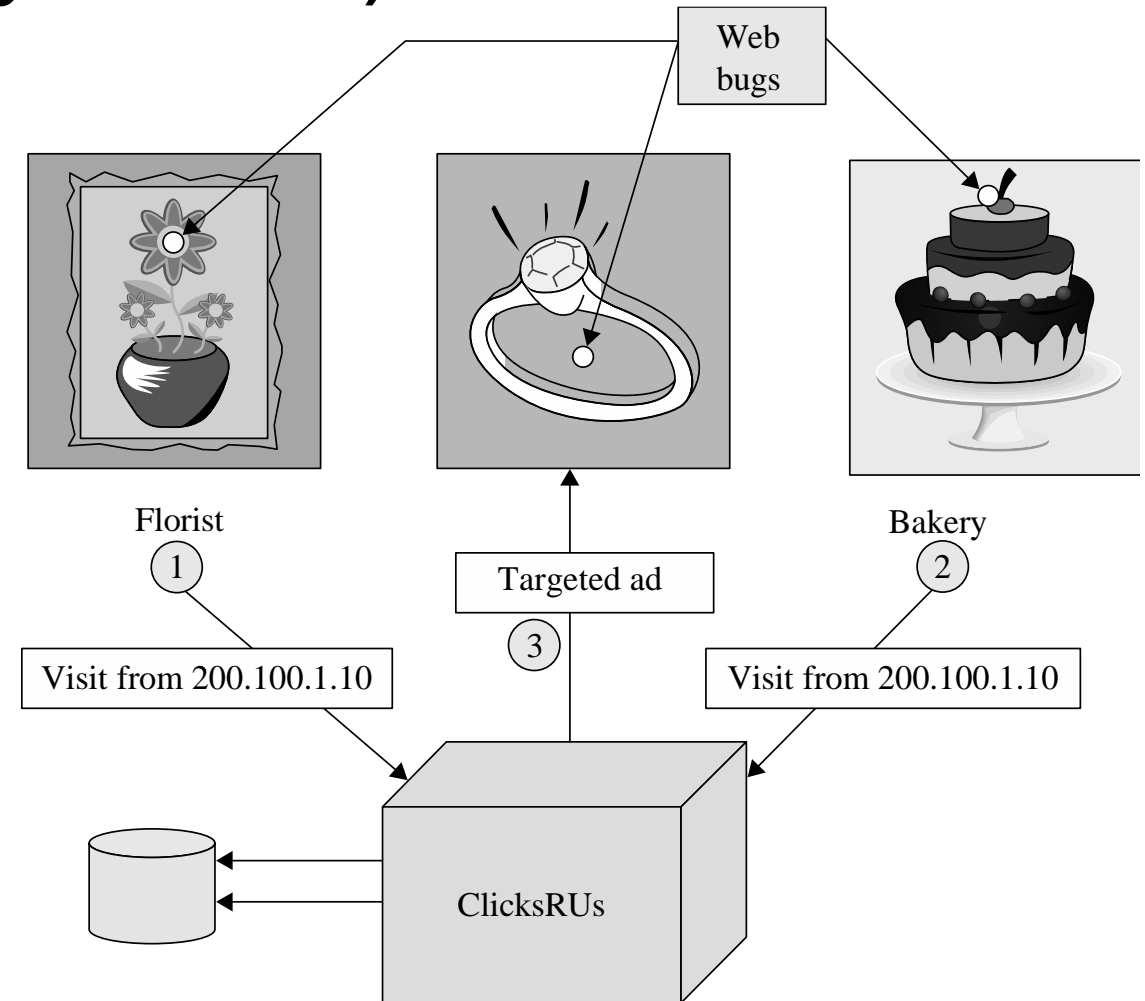


Tracking Bug (Privacy Issue)

- Tiny image unseen to web users
- URL linked to third-party site
- Can be used by an advertiser to track a buyer across shopping sites





Tracking with Browser Cookies

- Cookies are a small bit of information stored on a computer associated with a specific server
 - When you access a website, it might store information as a cookie
 - Every time you visit that server, the cookie is **re-sent** to the server
 - Effectively used to hold state information over sessions
- Cookies can hold any type of (sensitive) information
 - Passwords, credit card information, social security number, etc.
 - Session cookies, non-persistent cookies, persistent cookies



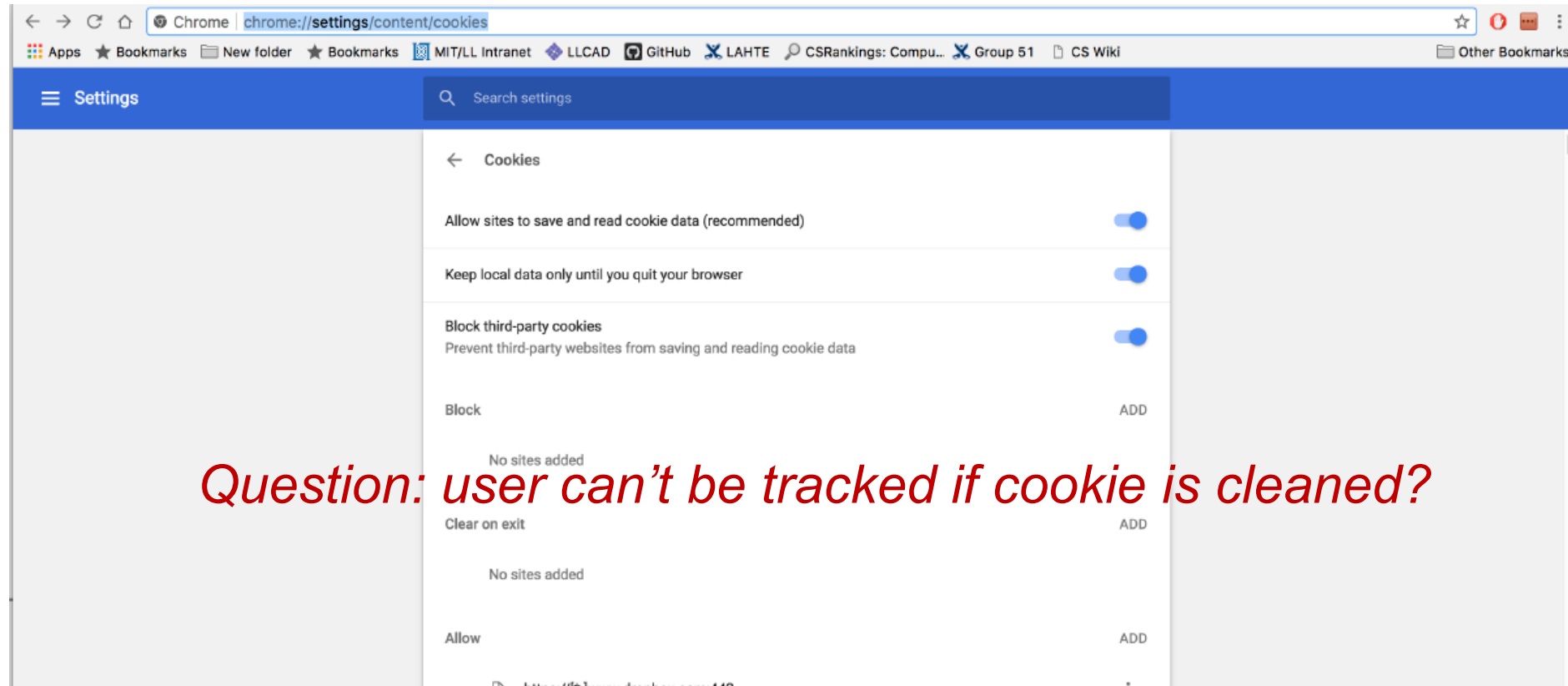
More on Cookies

- Cookies are stored on **your computer** and can be controlled
 - Many sites require that you enable cookies in order to use the site
 - Their storage on your computer naturally lends itself to exploits
 - You can (and probably should) clear your cookies on a regular basis
- Cookies expire
 - The expiration is set by the sites' session by default, which is chosen by the server (e.g., Jan. 1, 2036)
 - This means that cookies will probably stick around for a while



Taking Care of Your Cookies

- Managing your cookies in Chrome:



Question: user can't be tracked if cookie is cleaned?

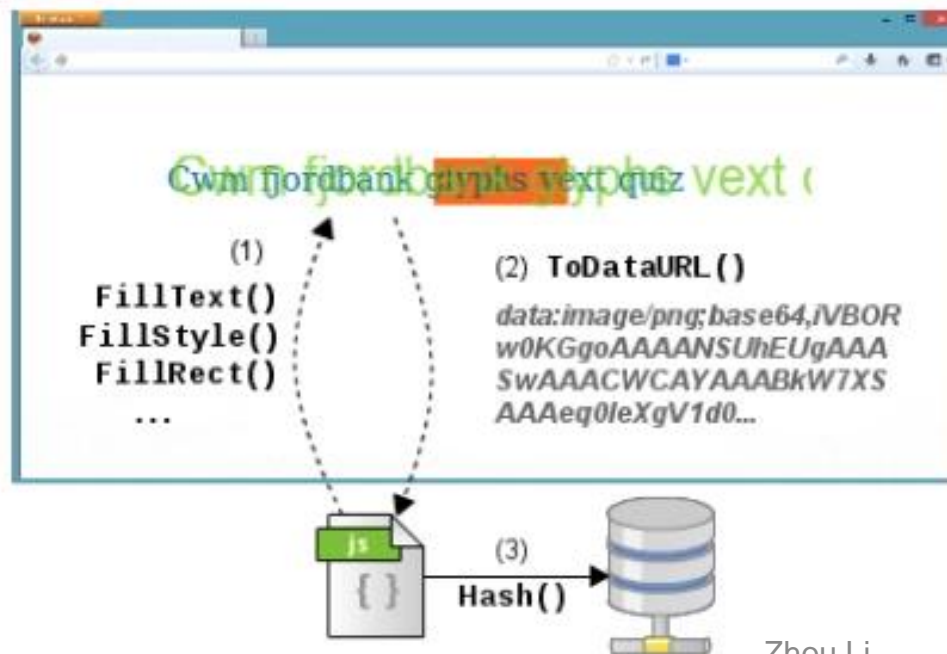


Persistent User Tracking

- Tracking users **over time**, **even after users clean cookies**
- Tracking (sharing) users **across trackers**
- Tracking users over time
 - Canvas fingerprinting
 - Ever cookie (flash cookies)
 - Cookie syncing: different trackers share user identifiers

1. Canvas Fingerprinting

- Tracker seeks to uniquely identify a user/browser
 - Let the browser render some text (can be invisible to users)
 - Generate browser signatures based text rendering



Very subtle differences for different browsers

- Operating system
- Font library
- Graphics card
- Graphics driver
- Browser implementation
- ...



Obtaining user or website data

- Server-side attacks
 - Dot-Dot-Slash (or directory traversal)
 - Server-Side Include (SSI)
- Client-side attacks
 - Cross-site Scripting (XSS)
 - Cross-site Request Forgery (XSRF)



Dot-Dot-Slash

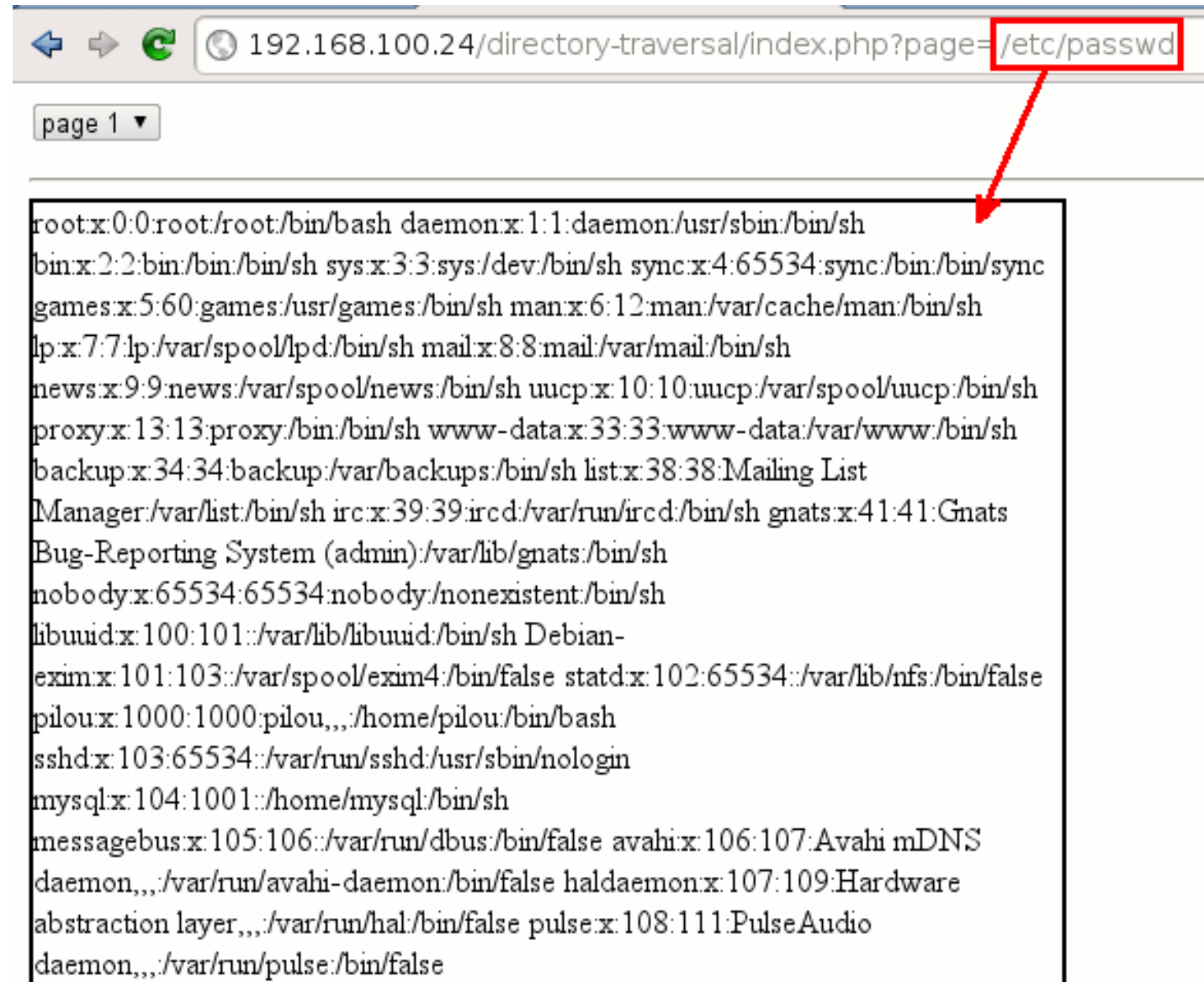
- Also known as “**directory traversal**,” this is when attackers use the term “../” to access files that are on the target web server but not meant to be accessed from outside
- Most commonly entered into the URL bar but may also be combined with other attacks, such as XSS
- **Root cause: inconsistent/missing access control**

```
http://yoursite.com/webhits.htw?CiWebHits&File=../../../../winnt/system32/autoexec.nt
```



Example

- Passwd not protected
- User ID leaked



```
root:x:0:0:root:/root:/bin/bash daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh sys:x:3:3:sys:/dev:/bin/sh sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh list:x:38:38:Mailing List
Manager:/var/list:/bin/sh irc:x:39:39:ircd:/var/run/ircd:/bin/sh gnats:x:41:41:Gnats
Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
libuuid:x:100:101:/var/lib/libuuid:/bin/sh Debian-
exim:x:101:103:/var/spool/exim4:/bin/false statd:x:102:65534:/var/lib/nfs:/bin/false
pilou:x:1000:1000:pilou,,/home/pilou:/bin/bash
sshd:x:103:65534:/var/run/sshd:/usr/sbin/nologin
mysql:x:104:1001:/home/mysql:/bin/sh
messagebus:x:105:106:/var/run/dbus:/bin/false avahi:x:106:107:Avahi mDNS
daemon,,/var/run/avahi-daemon:/bin/false haldaemon:x:107:109:Hardware
abstraction layer,,/var/run/hal:/bin/false pulse:x:108:111:PulseAudio
daemon,,/var/run/pulse:/bin/false
```



Server-Side Include (SSI)

- SSI is an interpreted server-side scripting language that can be used for basic web server directives, such as including files and executing commands, like sending email after user click button on web page
- SSI attack allows the exploitation of a web application by injecting scripts in HTML pages or executing arbitrary codes remotely
- The web server parses and executes the directives before supplying the page. **Then, the attack result will be viewable the next time that the page is loaded for the user's browser.**
- ```
<!--#exec cmd="/usr/bin/telnet &"-->
```



# Client-side Security: Browser Policies

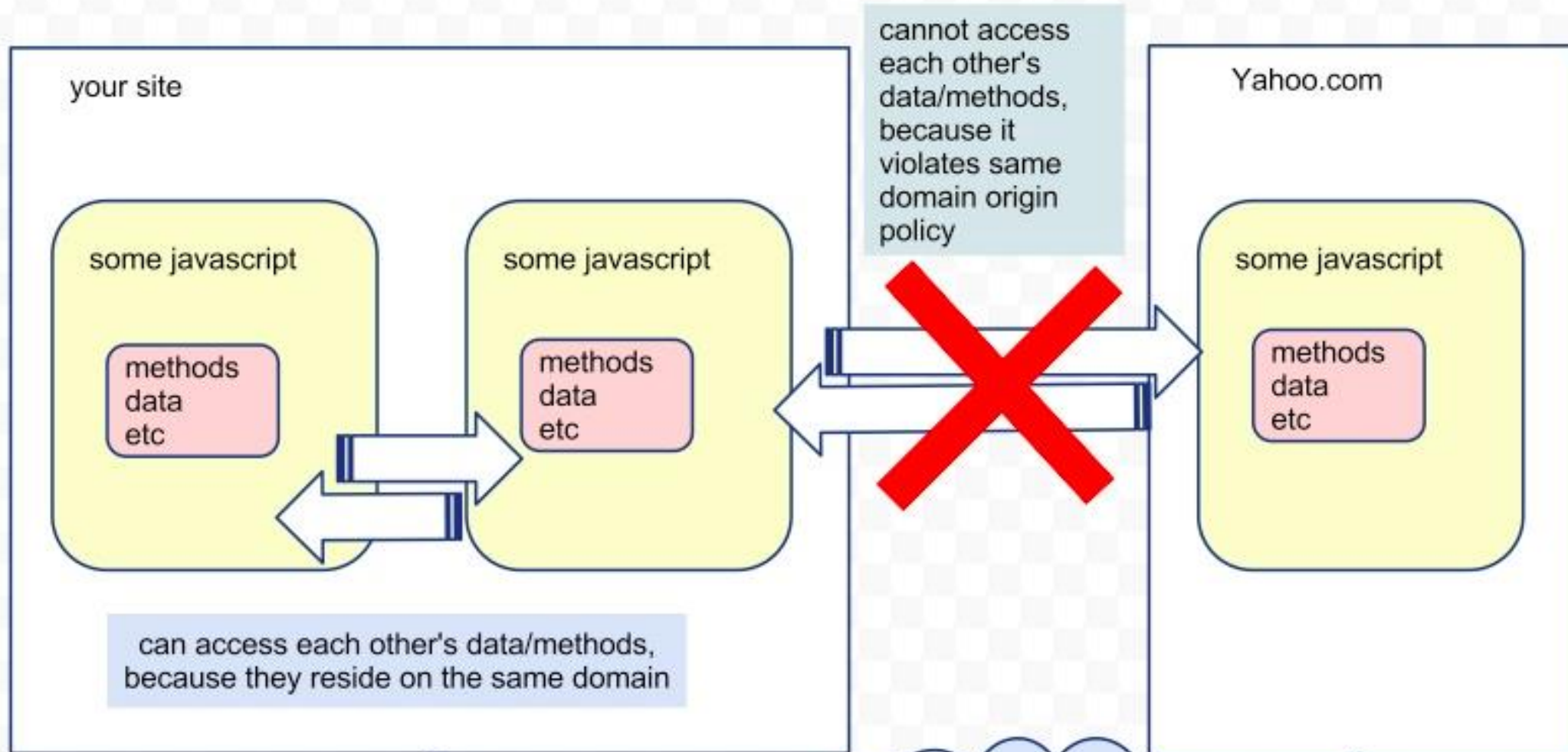
- Key mechanism: *Same-origin policy (SOP)*
  - **SOP is a sandbox model**: only the site that stores the information in the browser can read or modify that information
- An **untrusted** page cannot corrupt the user's actions at other sites nor can it issue transactions on behalf of the user
- Applies to cookies, JavaScript access to DOMs, and plugins
  - **Cookies**: cookie from origin A not visible to origin B
  - **DOM (Document Object Model)** : script from origin A cannot read or set properties for origin B



# Confining the Power of JavaScript Scripts

- Given all that power, browsers need to make sure JS scripts don't abuse it
- For example, don't want a script sent from **hackerz.com** web server to read cookies belonging to **bank.com**...
- ... or alter layout of a **bank.com** web page
- ... or read keystrokes typed by user while focus is on a **bank.com** page!

# SOP





# Same-origin Examples

- Origin: protocol, hostname, port, but not path
- Same Origin
  - <http://www.example.org/here>
  - <http://www.example.org/there>
  - same protocol: http, host: example, default port 80
- How about these?
  - <http://www.example.org/here>
  - <https://www.example.org/there>
  - <http://www.example.org:8080/hello>
  - <http://www.hacker.org/you>





# More on SOP

- Origin comparisons with <http://store.company.com/dir/page.html>

| URL                                                          | Outcome | Reason             |
|--------------------------------------------------------------|---------|--------------------|
| <code>http://store.company.com/dir2/other.html</code>        | Success | -                  |
| <code>http://store.company.com/dir/inner/another.html</code> | Success | -                  |
| <code>https://store.company.com/secure.html</code>           | Failure | Different protocol |
| <code>http://store.company.com:81/dir/etc.html</code>        | Failure | Different port     |
| <code>http://news.company.com/dir/other.html</code>          | Failure | Different host     |



# SOP protection scope

- Content (e.g. JavaScript) from site A and site B are under different **windows, frames and iframes, separated by SOP**
  - `<iframe src="siteB"></iframe>`
- If site A includes site B through **script tag, unprotected by SOP**
  - `<script src="siteB">...</script>`
- If site A includes code in **URL parameters, unprotected by SOP**
  - **XSS, XSRF**





# Cross-site Scripting (XSS)

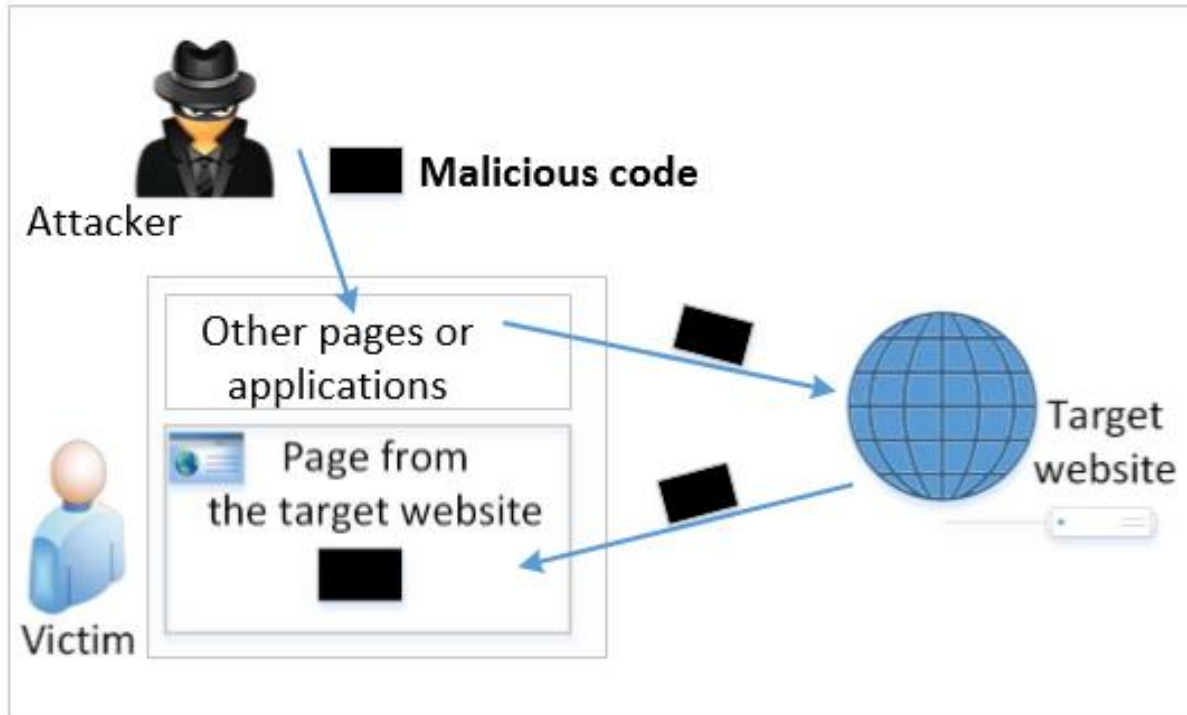
- **Problem:** Lack of input sanitization on a trusted website
- **Attack:** attacker submits code as data to a **trusted website**; later, the trusted website serves that malicious script to users
- **Outcome:** allows the attacker to have their scripts run as if they were a part of the trusted site



# Types of XSS Attacks

- Non-persistent (Reflected) XSS Attack
- Persistent (Stored) XSS Attack

# Non-persistent (Reflected) XSS Attack



If a website with a reflective behavior takes user inputs, then :

- Attackers can put JavaScript code in the input
- When the input is reflected back, the JavaScript code will be injected into the web page



# Vulnerable HTML page

<http://www.example.com/search?input=word>


```
<HTML>
<script>
function getUrlVars() {
 var vars = {};
 var parts = window.location.href.replace(/[?&]+([^\&]+)=([^\&]*)/gi, function(m,key,value) {
 vars[key] = value;
 });
 return vars;
}
var input = getUrlVars()["input"];
</script>
<div id="output"></div>
<script> document.getElementById("output").innerText = "User searched " + input; </script>
</HTML>
```

# Non-persistent (Reflected) XSS Attack

[http://www.example.com/search?input=<script>alert\("attack"\);</script>](http://www.example.com/search?input=<script>alert('attack');</script>)

*After JavaScript runs*

```
<HTML>
<script>
function getUrlVars() {
 var vars = {};
 var parts = window.location.href.replace(/[?&]+([^\&]+)=([^\&]*)/gi, function(m,key,value) {
 vars[key] = value;
 });
 return vars;
}
var input = getUrlVars()["input"];
</script>
<div id="output"> User searched <script> alert("attack"); </script> </div>
<script> document.getElementById("output").innerText = "User searched " + input; </script>
</HTML>
```





# How to attack user

- The attacker sends the following URL to the victim
  - [http://www.example.com/search?input=<script>alert\("attack"\);</script>](http://www.example.com/search?input=<script>alert("attack");</script>)
- The victim clicks on this link
- Malicious JavaScript code executed on example.com
  - This example pops up the a dialog
  - But real-world attacker can do much more, like **leaking cookies to attacker's website, installing keylogger**
- This example is also named DOM XSS