nccgroup

CSRF in the Modern Age

Sidestepping the CORS Standard

Tanner Prynn
@tannerprynn

nccgroup

In This Talk

- The State of CSRF
- The CORS Standard
- How *Not* To Prevent CSRF

The Fundamentals of HTTP

nccgroup

Without cookies:

```
GET / HTTP/1.1
Host: example.com
Connection: keep-alive
User-Agent: Mozilla/5.0 ...
Origin: attacker.example.com
Accept: */*
```

```
HTTP/1.1 200 OK
Server: nginx
Content-Type: text/plain

Please log in!
```

With cookies:

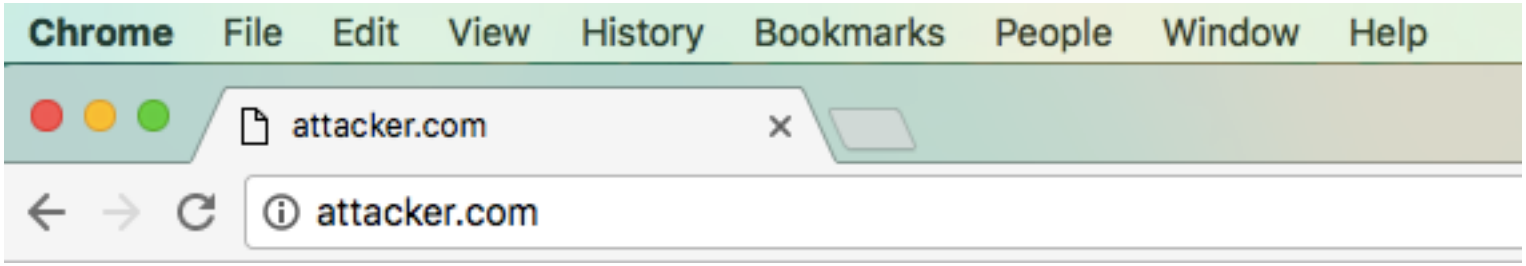
```
GET / HTTP/1.1
Host: example.com
Connection: keep-alive
User-Agent: Mozilla/5.0 ...
Origin: attacker.example.com
Accept: */*
Cookie: SessionID=blahblah
```

```
HTTP/1.1 200 OK
Server: nginx
Content-Type: text/plain

Your SSN is: 123456
```

The Fundamentals of CSRF

nccgroup

User visits attacker.com:



Browsers enforce **security boundaries**, so attacker.com cannot access information for example.com (one piece of the Same-Origin Policy):

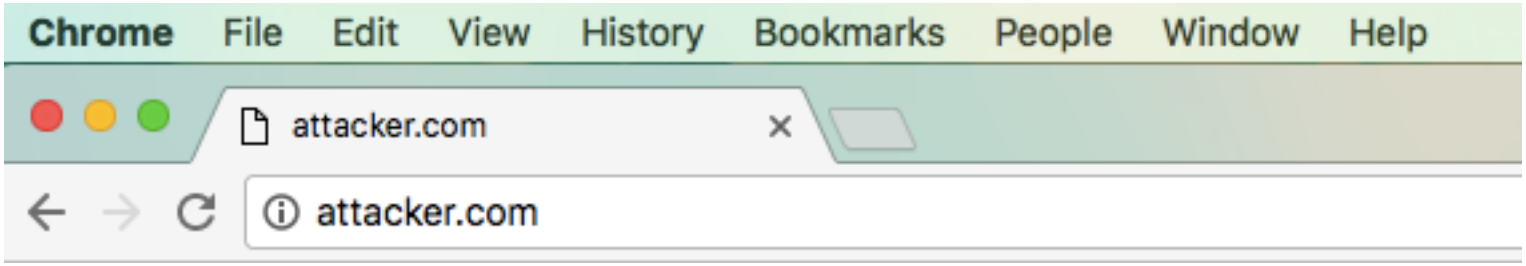
```
GET / HTTP/1.1
Host: example.com
Connection: keep-alive
User-Agent: Mozilla/5.0 ...
Origin: attacker.example.com
Accept: */*
Cookie: SessionID=blahblah
```

```
HTTP/1.1 200 OK
Server: nginx
Content-Type: text/plain

Your SSN is: 123456
```

The Fundamentals of CSRF

nccgroup

User visits attacker.com:



Javascript on attacker.com running in user's browser:

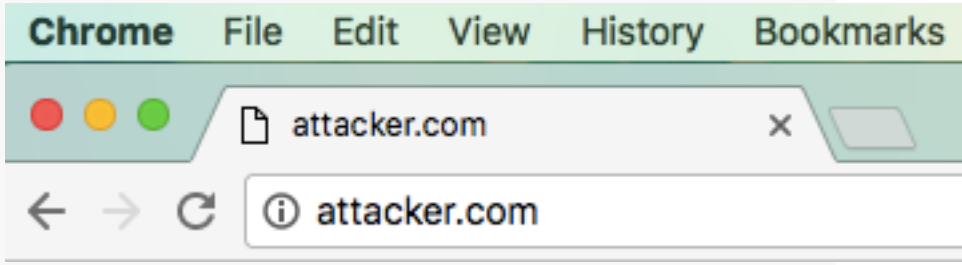
```
var x = new XMLHttpRequest();
x.open("GET", "http://example.com"); // GET request to example.com
x.withCredentials = true;            // If we have cookies, send 'em
x.send();                            // Actually submit the request
```

The Fundamentals of CSRF

nccgroup

attacker.com can *send a request* to example.com:

```
var x = new XMLHttpRequest();
x.open("GET", "http://example.com");
x.withCredentials = true;
x.send();
```




```
GET / HTTP/1.1
Host: example.com
Connection: keep-alive
User-Agent: Mozilla/5.0 ...
Origin: attacker.example.com
Accept: */*
Cookie: SessionID=blahblah
```

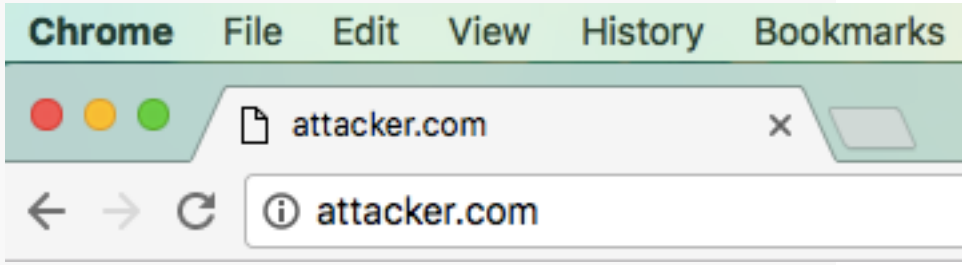
```
HTTP/1.1 200 OK
Server: nginx
Content-Type: text/plain

Your SSN is: 123456
```

The Fundamentals of CSRF

nccgroup

attacker.com can *send a request* to example.com:

```
var x = new XMLHttpRequest();
x.open("GET", "http://example.com");
x.withCredentials = true;
x.send();
```




```
GET / HTTP/1.1
Host: example.com
Connection: keep-alive
User-Agent: Mozilla/5.0 ...
Origin: attacker.example.com
Accept: */*
Cookie: SessionID=blahblah
```

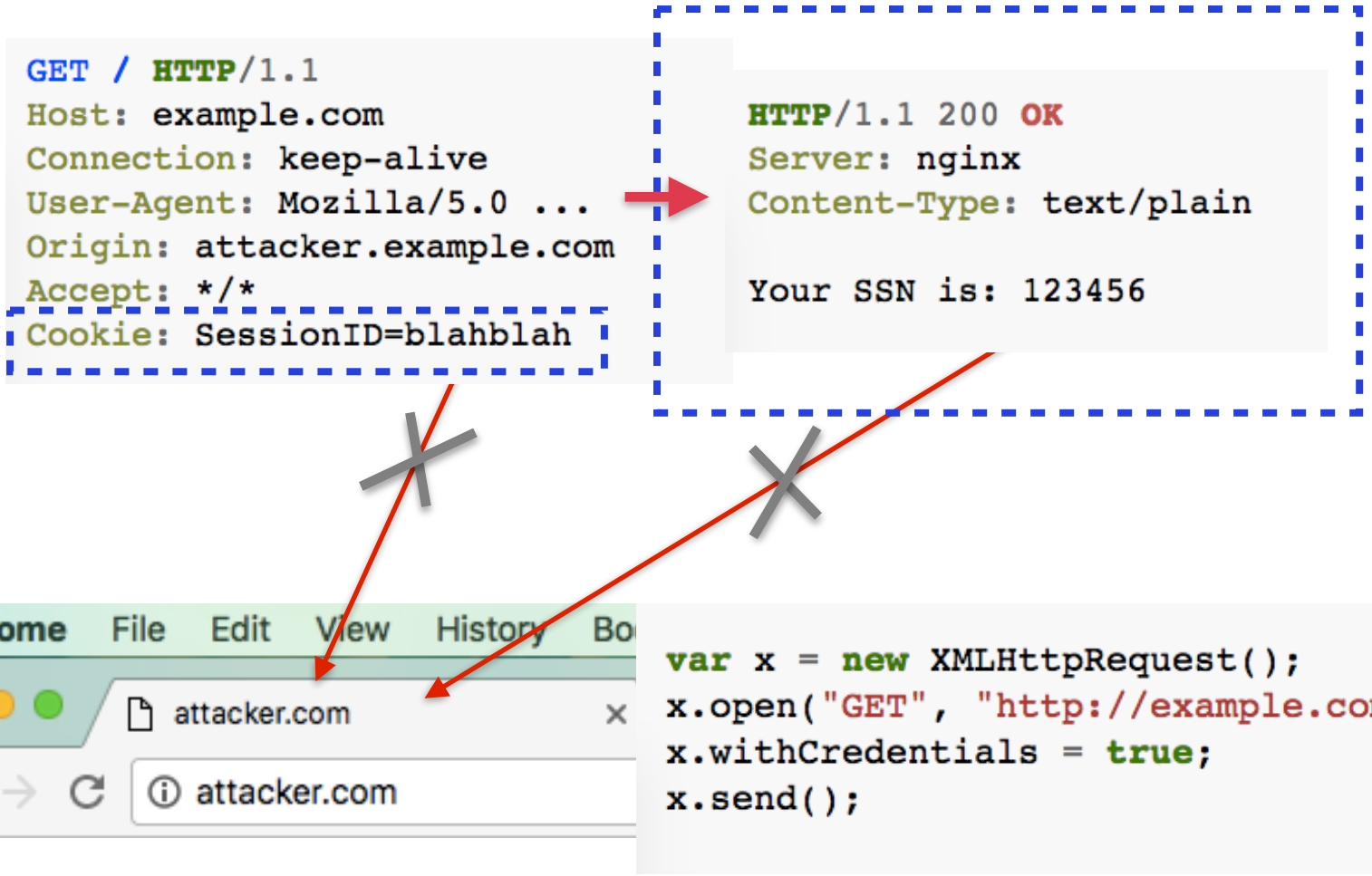
```
HTTP/1.1 200 OK
Server: nginx
Content-Type: text/plain

Your SSN is: 123456
```

This is a **cross-site request** because it goes between two sites (origins)

The Fundamentals of CSRF

nccgroup

Security boundaries are still enforced.



The Fundamentals of CSRF

nccgroup

What if the request causes a **State-Changing Action** on the server?

```
POST /DeleteAccount HTTP/1.1
Host: example.com
Connection: keep-alive
User-Agent: Mozilla/5.0 ...
Origin: attacker.example.com
Accept: */*
Cookie: SessionID=blahblah
```

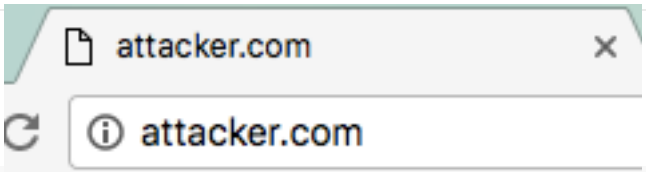
```
HTTP/1.1 200 OK
Server: nginx
Content-Type: text/plain

Account deleted!
```

Attacker would normally need a valid session ID for the target…

The Fundamentals of CSRF

nccgroup

But when a user visits attacker.com:




```
var x = new XMLHttpRequest();
x.open("POST", "http://example.com/DeleteAccount");
x.withCredentials = true; // Tell the browser to send cookies!
x.send();
```

The attacker can trigger the state-changing request, and the *browser will supply the cookie*!

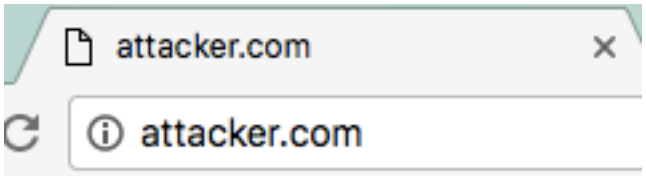
```
POST /DeleteAccount HTTP/1.1
Host: example.com
Connection: keep-alive
User-Agent: Mozilla/5.0 ...
Origin: attacker.example.com
Accept: */*
Cookie: SessionID=blahblah
```

```
HTTP/1.1 200 OK
Server: nginx
Content-Type: text/plain

Account deleted!
```

The Fundamentals of CSRF

nccgroup

This is **Cross-Site Request Forgery** (CSRF)




```
var x = new XMLHttpRequest();
x.open("POST", "http://example.com/DeleteAccount");
x.withCredentials = true; // Tell the browser to send cookies!
x.send();
```

```
HTTP/1.1 200 OK
Server: nginx
Content-Type: text/plain

Account deleted!
```

The Fundamentals of CSRF

nccgroup

What kind of state-changing actions can be CSRFed?

- Account creation & modification
- Admin functions: SQL / `run_command?cmd=rm /`
- Vulns only exploitable by admins: XSS, SQLi, etc
- Self-XSS *becomes exploitable* through CSRF

CSRF is, by nature, an authorization bypass

nccgroup

The State of CSRF

nccgroup

The State of CSRF

- **CSRF is everywhere**

nccgroup

The State of CSRF

- **CSRF is everywhere**
- **Developers don’t understand how to protect against it**

nccgroup

The State of CSRF

- **CSRF is everywhere**
- **Developers don’t understand how to protect against it**
- **CSRF enables attacks that don’t exist otherwise**

nccgroup

The State of CSRF

- **CSRF is everywhere**
- **Developers don’t understand how to protect against it**
- **CSRF enables attacks that don’t exist otherwise**
- **Users don’t have a good way to protect against it**

The State of CSRF 0x00

CVE-ID	
CVE-2013-6343	Learn more at National Vulnerability Database (NVD) • Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings
Description	
Multiple buffer overflows in web.c in httpd on the ASUS RT-N56U and RT-AC66U routers with firmware 3.0.0.4.374_979 allow remote attackers to execute arbitrary code via the (1) apps_name or (2) apps_flag parameter to APP_Installation.asp.	

CSRF -> code exec in popular ASUS routers

Every major router/modem vendor had (has?) their own version of this

(from Jacob Holcomb - link)

image from Asus

The State of CSRF 0x01

Helpful local HTTP service:

```
https://localhost:49155/api/
openUrlInDefaultBrowser?url=c:/
windows/system32/calc.exe
```

CSRF -> code exec in TrendMicro Antivirus/Password Manager

See also: password disclosure CSRF in Lastpass

(from Tavis Ormandy - link)

image from TrendMicro

The State of CSRF 0x02

CSRF -> Full Control of Monero (Digital Currency) “SimpleWallet”

```
POST http://127.0.0.1:18082/json_rpc

{
  "jsonrpc":"2.0",
  "id":"0",
  "method":"transfer",
  "params":{
    "destinations":[{
      "amount":100000000000,
      "address":"49FuXtv..."
    }],
    ...
  }
}
```

From MWR InfoSec (link)

image from Monero

nccgroup

The CORS Standard

Cross-Origin Resource Sharing

"CORS defines a way in which a browser and server can interact to determine whether or not it is safe to allow [a] cross-origin request."

- Wikipedia

The CORS Standard

Problem:

api.example.com wants to provide an API that another site on the Internet can talk to through Javascript

The CORS Standard

Problem:

api.example.com wants to provide an API that another site on the Internet can talk to through Javascript

Solution:

Allow browsers to do more with requests (headers, content-types) and read the full response

The CORS Standard

Problem:

Servers *cannot* distinguish between legitimate requests from their own sites, and CSRF - the HTTP request looks the same

nccgroup

The CORS Standard

Problem:

Servers *cannot* distinguish between legitimate requests from their own sites, and CSRF - the HTTP request looks the same

Solution:

Browser tells the server when it wants to make a cross-site request, and the server can say yes or no

nccgroup

The CORS Standard

We ask the browser to send a request like the following:

```
PATCH /my_account HTTP/1.1
Host: account.example.com
Connection: keep-alive
User-Agent: Mozilla/5.0 ...
Origin: manage.example.com
Accept: */*
Cookie: SessionID=deadbeef
Content-Type: application/json

{"password": "my_new_password"}
```

nccgroup

The CORS Standard

Preflight Request + Response

```
OPTIONS /my_account HTTP/1.1
Host: account.example.com
Access-Control-Request-Method: PATCH
Access-Control-Request-Headers: Content-Type
Origin: http://manage.example.com
```

```
HTTP/1.1 200 OK
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: http://manage.example.com
Access-Control-Allow-Methods: GET, OPTIONS, POST, PATCH, DELETE
Access-Control-Allow-Headers: Content-Type
Access-Control-Allow-Credentials: true
```

nccgroup

CORS: Preflight Request

```
OPTIONS /my_account HTTP/1.1
Host: account.example.com
Access-Control-Request-Method: PATCH
Access-Control-Request-Headers: Content-Type
Origin: http://manage.example.com
```

Origin: Where did the request come from?

Access-Control-**Request-Method**: What method do we want to send?

Access-Control-**Request-Headers**: What headers do we want to modify?

CORS: Preflight Response

```
HTTP/1.1 200 OK
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: http://manage.example.com
Access-Control-Allow-Methods: GET, OPTIONS, POST, PATCH, DELETE
Access-Control-Allow-Headers: Content-Type
Access-Control-Allow-Credentials: true
```

Access-Control-**Allow-Origin**: Only this origin is allowed (can be a wildcard *)

Access-Control-**Allow-Methods**: Any of these HTTP verbs are allowed

CORS: Preflight Response

```
HTTP/1.1 200 OK
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: http://manage.example.com
Access-Control-Allow-Methods: GET, OPTIONS, POST, PATCH, DELETE
Access-Control-Allow-Headers: Content-Type
Access-Control-Allow-Credentials: true
```

Access-Control-**Allow-Headers**: These headers can be modified

Access-Control-**Allow-Credentials**: The browser can send cookies (or other credentials)

The CORS Standard

If the server responded with valid Access-Control-Allow headers:

Security boundary around the response is no longer enforced (!)

```
PATCH /my_account HTTP/1.1
Host: account.example.com
Connection: keep-alive
User-Agent: Mozilla/5.0 ...
Origin: manage.example.com
Accept: */*
Cookie: SessionID=deadbeef
Content-Type: application/json

{"password": "my_new_password"}
```

```
HTTP/1.1 200 OK
Server: Apache/2.0.61 (Unix)
Content-Type: application/json
Content-Length: 20

{"Result": "Password changed"}
```

*Most response headers are not readable unless Access-Control-Expose-Headers is set

nccgroup

CORS Issues: Legacy Support

Browsers can’t make any changes that break sites

=

Browser security standards are dead before they’re implemented

nccgroup

CORS Issues: Legacy Support

HTTP Methods

```
GET
HEAD
POST
PUT
PATCH
DELETE
```

`GET`, `HEAD`, and `POST` worked before CORS, so they have to keep working after

Can't change how they work at all

CORS Issues: Legacy Support

Thus, we get **Simple Requests**

GET
HEAD
POST

+

Content-Type:
`application/x-www-form-urlencoded`
`multipart/form-data`
`text/plain`

nccgroup

CORS Issues: Legacy Support

Thus, we get **Simple Requests**

GET
HEAD
POST

+

Content-Type:
`application/x-www-form-urlencoded`
`multipart/form-data`
`text/plain`

Classic CSRF: No Preflight Required

CORS Issues: Legacy Support

Everything else:

PUT	HTTP Headers
PATCH	Any other Content-Type
DELETE	

Requires CORS preflight

nccgroup

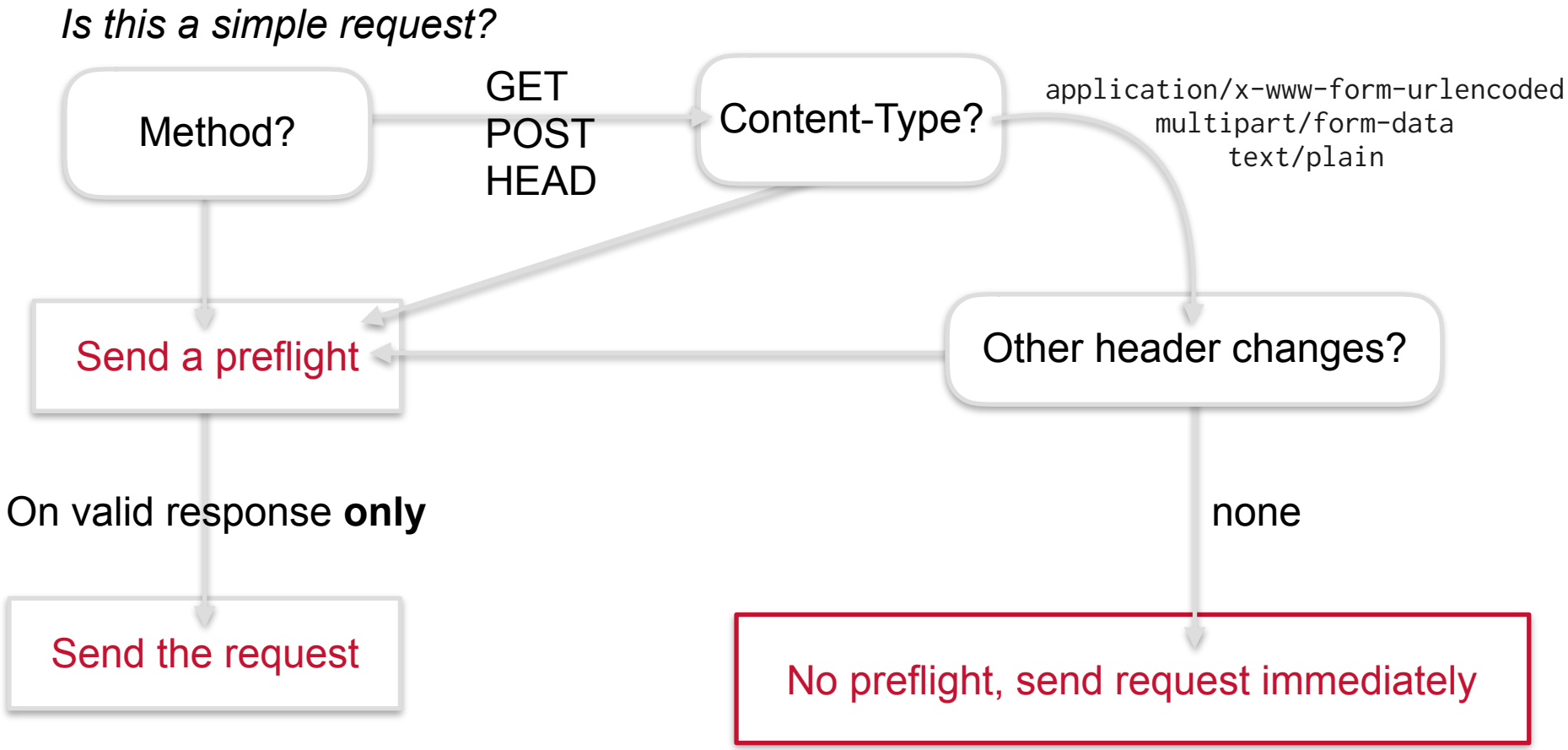
CORS In Practice

Rule Zero, the **Same-Origin Policy**:

Any requests to the **same domain**
(that the request is initiated from)
are **not affected** by CORS

- No Preflight for any request
- Can change most headers
- Can read response body
- Can read most response headers

CORS In Practice



CORS In Practice

```
GET / HTTP/1.1
Host: info.gov
Connection: keep-alive
User-Agent: Mozilla/5.0 ...
Origin: example.com
Accept: */*
Cookie: SessionID=blahblah
```

For simple requests: returning Valid Access-Control headers means the requesting site can read the response

```
HTTP/1.1 200 OK
Server: nginx
Access-Control-Allow-Origin: example.com
Access-Control-Allow-Credentials: true
Content-Type: text/plain

Your SSN is: 123456
```

nccgroup

CORS In Practice

We ask the browser to send a request like the following:

```
PATCH /my_account HTTP/1.1
Host: account.example.com
Connection: keep-alive
User-Agent: Mozilla/5.0 ...
Origin: manage.example.com
Accept: */*
Cookie: SessionID=deadbeef
Content-Type: application/json

{"password": "my_new_password"}
```

nccgroup

CORS In Practice

Preflight Request + Response

1

```
OPTIONS /my_account HTTP/1.1
Host: account.example.com
Access-Control-Request-Method: PATCH
Access-Control-Request-Headers: Content-Type
Origin: http://manage.example.com
```

2

```
HTTP/1.1 200 OK
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: http://manage.example.com
Access-Control-Allow-Methods: GET, OPTIONS, POST, PATCH, DELETE
Access-Control-Allow-Headers: Content-Type
Access-Control-Allow-Credentials: true
```

The CORS Standard

After a valid preflight response, the browser sends the original request

3

```
PATCH /my_account HTTP/1.1
Host: account.example.com
Connection: keep-alive
User-Agent: Mozilla/5.0 ...
Origin: manage.example.com
Accept: */*
Cookie: SessionID=deadbeef
Content-Type: application/json

{"password": "my_new_password"}
```

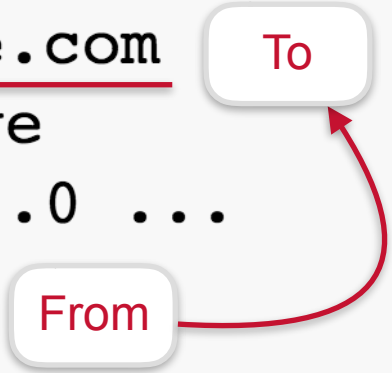
4

```
HTTP/1.1 200 OK
Server: Apache/2.0.61 (Unix)
Content-Type: application/json
Content-Length: 20

{"Result": "Password changed"}
```

CORS In Practice

We ask the browser to send a request like the following:

```
PATCH /my_account HTTP/1.1
Host: account.example.com
Connection: keep-alive
User-Agent: Mozilla/5.0 ...
Origin: attacker.com
Accept: */*
Cookie: SessionID=deadbeef
Content-Type: application/json

{"password": "attacker_password"}
```

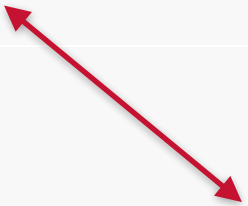


nccgroup

The CORS Standard

Request is not sent until **valid** preflight response is received

```
OPTIONS /my_account HTTP/1.1
Host: account.example.com
Access-Control-Request-Method: PATCH
Access-Control-Request-Headers: Content-Type
Origin: attacker.com
```



```
HTTP/1.1 200 OK
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: http://manage.example.com
Access-Control-Allow-Methods: GET, OPTIONS, POST, PATCH, DELETE
Access-Control-Allow-Headers: Content-Type
Access-Control-Allow-Credentials: true
```

Origin does not match, so browser will **not** send the request

CORS: Fundamental Issue 1

Problem:

Servers *cannot* distinguish between legitimate requests from their own sites, and CSRF - the HTTP request looks the same

~~**Solution:**~~

Is not solved by CORS in many cases!

CORS: Fundamental Issue 2

Security boundary around the response is no longer enforced (!)

```
PATCH /my_account HTTP/1.1
Host: account.example.com
Connection: keep-alive
User-Agent: Mozilla/5.0 ...
Origin: manage.example.com
Accept: */*
Cookie: SessionID=deadbeef
Content-Type: application/json

{"password": "my_new_password"}
```

```
HTTP/1.1 200 OK
Server: Apache/2.0.61 (Unix)
Content-Type: application/json
Content-Length: 20

{"Result": "Password changed"}
```

Relaxed Security Boundaries == Potential for New Vulnerabilities

CORS: Fundamental Issue 3

It’s really, REALLY confusing!

nccgroup

CORS Issues: It's Just Really Confusing

Which of the following **does not** trigger a CORS preflight*?

A) http://example.com to https://example.com

B) example.com to example.com:80

C) example.com:80 to example.com:3000

D) subdomain.example.com to example.com

E) a.example.com to b.example.com

*For example, with a PUT request

nccgroup

CORS Issues: It's Just Really Confusing

Which of the following **does not** trigger a CORS preflight*?

A) http://example.com to https://example.com

B) example.com to example.com:80

C) example.com:80 to example.com:3000

D) subdomain.example.com to example.com

E) a.example.com to b.example.com

*For example, with a PUT request

CORS Issues: It's Just Really Confusing

What happens if…

- A site returns

```
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: True
```

(This violates the CORS specification)

nccgroup

CORS Issues: It's Just Really Confusing

What happens if…

- A site returns

```
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: True
```

The browser protects you

CORS Issues: It's Just Really Confusing

Am I protected from CSRF if…

- My site uses client certificates (or HTTP Basic Auth)?

nccgroup

CORS Issues: It's Just Really Confusing

Am I protected from CSRF if…

- My site uses client certificates (or HTTP Basic Auth)?

NOPE

nccgroup

CORS Issues: It's Just Really Confusing

What does the browser do if…

- A site returns a `Set-Cookie` header
- But doesn't return valid CORS headers?

nccgroup

CORS Issues: It's Just Really Confusing

What does the browser do if…

- A site returns a `Set-Cookie` header
- But doesn't return valid CORS headers?

The browser will still set the cookies

nccgroup

CORS Issues: It's Just Really Confusing

Do different browser behave weirdly?

- Android 2.X is totally broken
- Safari requires `Access-Control-Allow-Headers: Origin`
- Internet Explorer 8/9 don't support everything, but are surprisingly secure

How Not to Prevent CSRF

nccgroup

Pre-CORS Checklist

- Does the site have CSRF protection (tokens)?
- Is protection applied to all state-changing routes?
- Are the CSRF tokens predictable?
- Could a token be leaked to an attacker? (e.g. token in URL)

nccgroup

Post-CORS

Scenario: We want to make use of CORS

Assume all requests trigger preflight (e.g. server requires a custom header to be sent with requests)

What can go wrong?

How Not to Prevent CSRF: `Allow-Origin`

The `Access-Control-Allow-Origin` header is really limited, it can only contain one of two things:

- Exactly one fully-specified domain
- A wildcard: `*` (incompatible with `Allow-Credentials`)

This means that most implementations will automatically generate a response based on the request's `Origin` header.

nccgroup

How Not to Prevent CSRF: Reflect Origin

```
function cors_middleware(req, res, next) {
    res.header("Access-Control-Allow-Origin", req.origin)

    res.header("Access-Control-Allow-Credentials", "true")
    res.header("Access-Control-Allow-Headers", "X-Custom-Header")
}
```

Any site which can get back a valid origin header for itself can make arbitrary requests, read responses

Reflecting origin == Removing Same-Origin Policy

nccgroup

How Not to Prevent CSRF: Reflect Subdomain

```
function cors_middleware(req, res, next) {
  if(req.origin.endsWith("example.com")) {
    res.header("Access-Control-Allow-Origin", req.origin)

    res.header("Access-Control-Allow-Credentials", "true")
    res.header("Access-Control-Allow-Headers", "X-Custom-Header")
  }
}
```

"We control all subdomains, so all of them should be allowed access!"

XSS in any subdomain == XSS in the main domain

nccgroup

The CORS Footgun

The combination of these two headers encompasses 90% of CORS security issues in the wild:

- `Access-Control-Allow-Origin`
- `Access-Control-Allow-Credentials`

nccgroup

The CORS Footgun

The combination of these two headers encompasses 90% of CORS security issues in the wild:

- `Access-Control-Allow-Origin`
- `Access-Control-Allow-Credentials`

Takeaway: Returning these two headers gives whatever origin is returned the ability to make requests and read the responses as the victim - just like XSS.

nccgroup

The CORS Footgun

Returning `Allow-Origin` and `Allow-Credentials` can allow attackers to bypass CSRF protection in otherwise secure sites

For any page that returns a CSRF token in the body:

- GET the page with a CSRF token
- CORS allows you to read the response, with the token
- Submit the token with a chosen request

How Not to Prevent CSRF: Method-Override

HTTP Method Override: functionality in many web frameworks that allows the HTTP method to be specified as a header, query param, or body param, which *overrides* the actual HTTP verb that was sent

Common ways to invoke:

- Headers: `X-HTTP-Method-Override` or `X-Method-Override`
- Query or Body Parameter: `_method`

Turn `GET` or `POST` into `PUT/PATCH/DELETE`

How Not to Prevent CSRF: Method-Override

HTTP Method Override:

Turn `GET` or `POST` into `PUT/PATCH/DELETE`

CSRF protection only applies to POST/PUT/etc

- Turn a GET into a POST to bypass CSRF protection*

Server returns `Access-Control-Allow-Methods`

- Send an allowed method, override to a banned method

Endpoint requires PUT/PATCH/DELETE, no CORS headers returned

- Send a simple request, override to PUT/PATCH/DELETE

*This is relatively uncommon, most frameworks have default protection against these types of attacks

How Not to Prevent CSRF: Content-Type Confusion

Some sites rely on CORS to prevent CSRF by:

- Making all requests with a header or content-type which prevents the request from being simple.
- Requests from their own site do not need to perform a preflight, so everything works there.
- Don’t return any preflight responses, so no cross-site requests will come through.

How Not to Prevent CSRF: Content-Type Confusion

Example: All requests use JSON, which makes the request non-simple. Browsing the site, you will never see a `text/plain` or `form-urlencoded` request.

But these requests will often still be parsed, and from perspective of the code, the objects look the same.

```
// parse application/x-www-form-urlencoded
app.use(bodyParser.urlencoded())

// parse application/json
app.use(bodyParser.json())
```

nccgroup

So what do I do?

Don’t return CORS headers

Use your framework’s CSRF protection



North America

Atlanta
Austin
Chicago
New York
San Francisco
Seattle
Sunnyvale

Europe

Manchester - Head Office
Amsterdam
Cheltenham
Copenhagen
Edinburgh
Glasgow
Leatherhead
London
Luxembourg
Milton Keynes
Munich
Zurich

Australia

Sydney