

## Cross Origin Resource Sharing(CORS)

- Cors is a set of browser controls set by web servers CORS defines what responses are allowed from where
  - Origins
  - Credentials
  - Methods
  - Headers

### Why use CORS?

- You have an API or web server with something to project
  - User data
  - Intellectual property
  - Branding

### Common CORS Scenario

- Create a new server for an API
- The server is set with some basic security in place
- Developers get a CORS allow-origin error from a localhost test
- Developer “Fixes” the error by bypassing CORS
- This hole is never plugged for production

```
~ > npm install cors
```

```
var express = require('express')
var cors = require('cors')
var app = express()

app.use(cors())

app.get('/product/:id',function (req,res,next){
  res.json({msg:'this is a CORS-enabled for all origins! '})
})

app.listen(80,function(){
  console.log('Cors-enabled web server listening on port 80')
})
```

## CORS Solution

- Set the headers, define what is allowed

```
import express from "express";
import helmet from "helmet";

const app = express();

// Use Helmet!
app.use(helmet());

app.get("/", (req, res) => {
  res.send("Hello world!");
});

app.listen(8000);
```

By default, Helmet sets the following headers:

- **Content-Security-Policy:** A powerful allow-list of what can happen on your page which mitigates many attacks
- **Cross-Origin-Opener-Policy:** Helps process-isolate your page
- **Cross-Origin-Resource-Policy:** Blocks others from loading your resources cross-origin
- **Origin-Agent-Cluster:** Changes process isolation to be origin-based
- **Referrer-Policy:** Controls the `Referer` header
- **Strict-Transport-Security:** Tells browsers to prefer HTTPS
- **X-Content-Type-Options:** Avoids MIME sniffing
- **X-DNS-Prefetch-Control:** Controls DNS prefetching
- **X-Download-Options:** Forces downloads to be saved (Internet Explorer only)
- **X-Frame-Options:** Legacy header that mitigates clickjacking attacks
- **X-Permitted-Cross-Domain-Policies:** Controls cross-domain behavior for Adobe products, like Acrobat
- **X-Powered-By:** Info about the web server. Removed because it could be used in simple attacks
- **X-XSS-Protection:** Legacy header that tries to mitigate XSS attacks, but makes things worse, so Helmet disables it

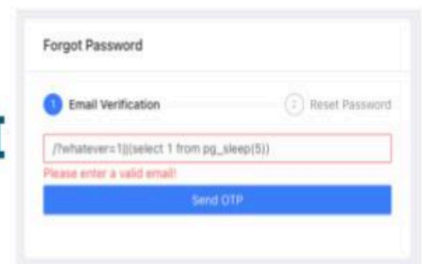
# Error Disclosure

Error Disclosure = revealing technology and code in an error

- Erroring out is necessary
- Intentional error handling is good
- Developers and customers are separate error message audiences

## Example

UI



Spring

API

```
{
  "message": "Validation Failed",
  "details": "org.springframework.validation.BeanPropertyBindingResult: 1
errors\n" +
  "Field error in object 'forgotPassword' on field 'email': rejected value
[/?whatever=1||(select 1 from pg_sleep(5))]; codes
[Email.forgotPassword.email,Email.email,Email.java.lang.String,Email]; arguments
[org.springframework.context.support.DefaultMessageSourceResolvable: codes
[forgotPassword.email,email]; arguments []; default message
[email],[Ljavax.validation.constraints.Pattern$Flag;@261d836,*]; default message
[must be a well-formed email address]"
}
```

How does this happen ?

## Python

```
try:
    print(x)
except E:
    print("An exception occurred", E)
```

# Node.js

```
try{
  fs.readFile('sample.txt',
    function(err, data) {
      if (err) throw err;
    });
} catch(err){
  console.log("In Catch Block")
  console.log(err);
}
```

## Error Handling Best Practices

### DO

- Error early
- Include useful information for developers in an error
- Log the error
- Display something specifically crafted for customers on error

### DO NOT

- Bypass Errors
- Expose Developer information to customers

## Server Information Leak

- Server information Leak = tech stack advertisement in headers
- Web Servers advertise in headers by default
- Appliances can also add headers
- These Headers have no benefit
- Detecting and removing them is easy

### Analyze your headers

1. Use a client to access the API(nothing cached)
2. Headers will be returned in any client or programming language
3. Look for:
  - Server
  - X-Powered-By
  - X-Version

## Insecure Cookies

- Insecure cookies = Cookies stored without restrictive security settings
- Secure Cookies
- HTTP only
- SameSite,Domains,SubDomains

## Cookie Decoding

- Find interesting keys/fields
- Sort Data Types
  - Unique ID string
  - Numeric ID
  - Booleans
  - Encoded Data
- Decoding Data
  - Delimiters
  - Base64 code and similar

## The Problem

1. Cookie forging / Fuzzing (Trusting cookie data)
2. Data Harvesting, different site (httponly + no Domains)
3. Data Harvesting, through XSS (httponly)
4. Data Harvesting in transit (Secure flag)

## Solutions

1. Treat cookies as untrusted user data
2. Be restrictive on what data is stored in cookies
3. Analyze your cookies from an offensive mindset

```
res.cookie('exampleCookieName', 'client side persistence', {  
  maxAge: 86400 * 1000000, // 24 hours  
  httpOnly: true, // http only, prevents JavaScript cookie access  
  secure: true // cookie must be sent over https / ssl  
});
```

## Path Traversal

Path Traversal = Allowing access to unintended paths on a server

- Directory Traversal
- Direct file reference
- Server Configuration
- Defensive coding
- Spec

### Malicious Cookie

```
GET /vulnerable.php HTTP/1.0  
Cookie:  
TEMPLATE=../../../../../../../../etc/passwd
```

### Dynamic Path Sourcecode

```
<?php  
$template = 'blue.php';  
if ( is_set( $_COOKIE['TEMPLATE'] ) )  
  $template = $_COOKIE['TEMPLATE'];  
include ( "/home/users/phpguru/templates/" .  
  $template );  
?>
```

## The Problem

- Input Data validation
- Server config (Directory listings, and allowed include paths)
- Dynamics path in source code
- Loose Spec “String” allows attacks

## Solutions

1. Scrub input data.
  - a. Specify what’s allowed. Enum/Array/List
  - b. Don’t try filtering what’s bad ?
2. Disable server directory listings
3. Don’t store anything sensitive at web root.
4. Put the web root on a separate drive than system
5. Remove variables from paths in source code
6. When in doubt, error

## Rate Limiting

Rate Limiting = Setting inbound limitations to ensure a service level and reliability

- RateLimit Headers
  - RateLimit-Limit: 10
  - RateLimit-Remaining: 1
  - RateLimit-Reset: 7
- HTTP status = 429
- Scope = user, origin, global, resource, endpoint
- Server Side Throttling

## Why Rate Limit ?

- OWASP-API-4: Unrestricted Resource Consumption
- Service Availability
- Security
- Budget

## Limiting Layers

1. Gateway/Network Compliance
2. Service
3. Server
4. Endpoint
5. User
6. Client Signature
7. Quota
8. Logic

## Scenario 1: The Application Layer DDOS

### Botnet:

Bots are coordinating bursts of requests to an endpoint

### XSS:

Requests come through people clicking on crafted links that reflect to your endpoint

Requiring Authentication helps.

throttle by User OR client signature after that.

## Scenario 2: The Impatient User

Easily identified by identical requests

Manageable through a browser “debounce”

### 420 Enhance Your Calm (Twitter)

Returned by version 1 of the Twitter Search and Trends API when the client is being rate limited; versions 1.1 and later use the **429 Too Many Requests** response code instead.



### **Scenario 3: The brute force**

1. 4 digit Pin = 10,000 combinations
2. User Enumeration
3. Scraping
  - Attempts within a session can be easily counted
  - Across a session use a client signature to keep a count of attempts

### **Scenario 4: The Budget Server**

Hosting Resources cost \$\$

Don't get hit in the wallet!

The more processing or scaling invoked by an endpoint the higher the cost will be of not limiting.

limit concurrency or scope of queries

limit by quotas

### **Tech Tips**

- Avoid SQL operations to manage throttling
- Avoid disk operations
- Include IP with User
- Use cache where appropriate, (same query)

Example throttle Solution = memcache style

- key = user
- value = scope

- TTL = time frame of throttle

Example cache solution = memcache or key value store:

- key = hash of the query
- value = result
- short TTL