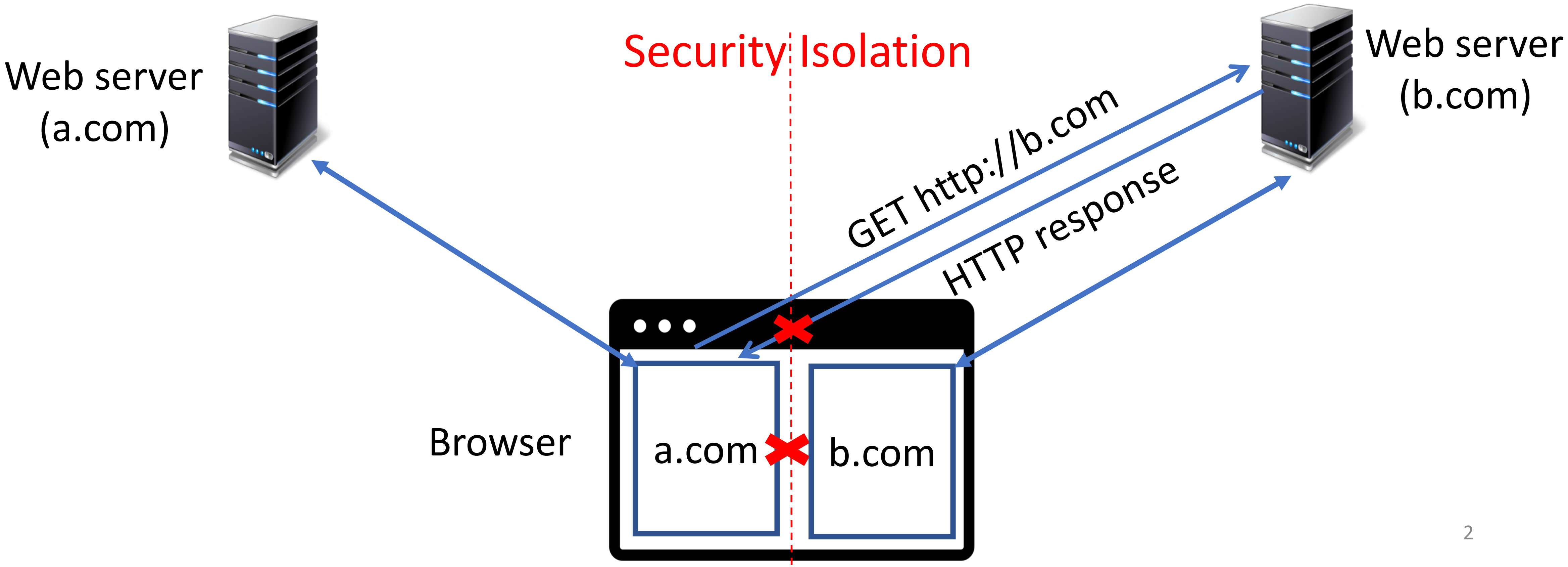
We Still Don’t Have Secure Cross-Domain Requests: an Empirical Study of CORS

Jianjun Chen, Jian Jiang, Haixin Duan, Tao Wan,
Shuo Chen, Vern Paxson, Min Yang

Tsinghua University, Shape Security, Huawei Canada,
Microsoft Research, UC Berkeley, Fudan University

Same Origin Policy (SOP)

- Isolate resources from different origins
- Cross origin network access: Can send, Can't Read



Developers need cross origin reading

- JSON with Padding (JSON-P)
  - A workaround to server the need
  - introduces many inherent security issues

- Cross Origin Resource Sharing (CORS)
  - A more disciplined mechanism
  - Browsers support(2009), W3C standard(2014)

Our work

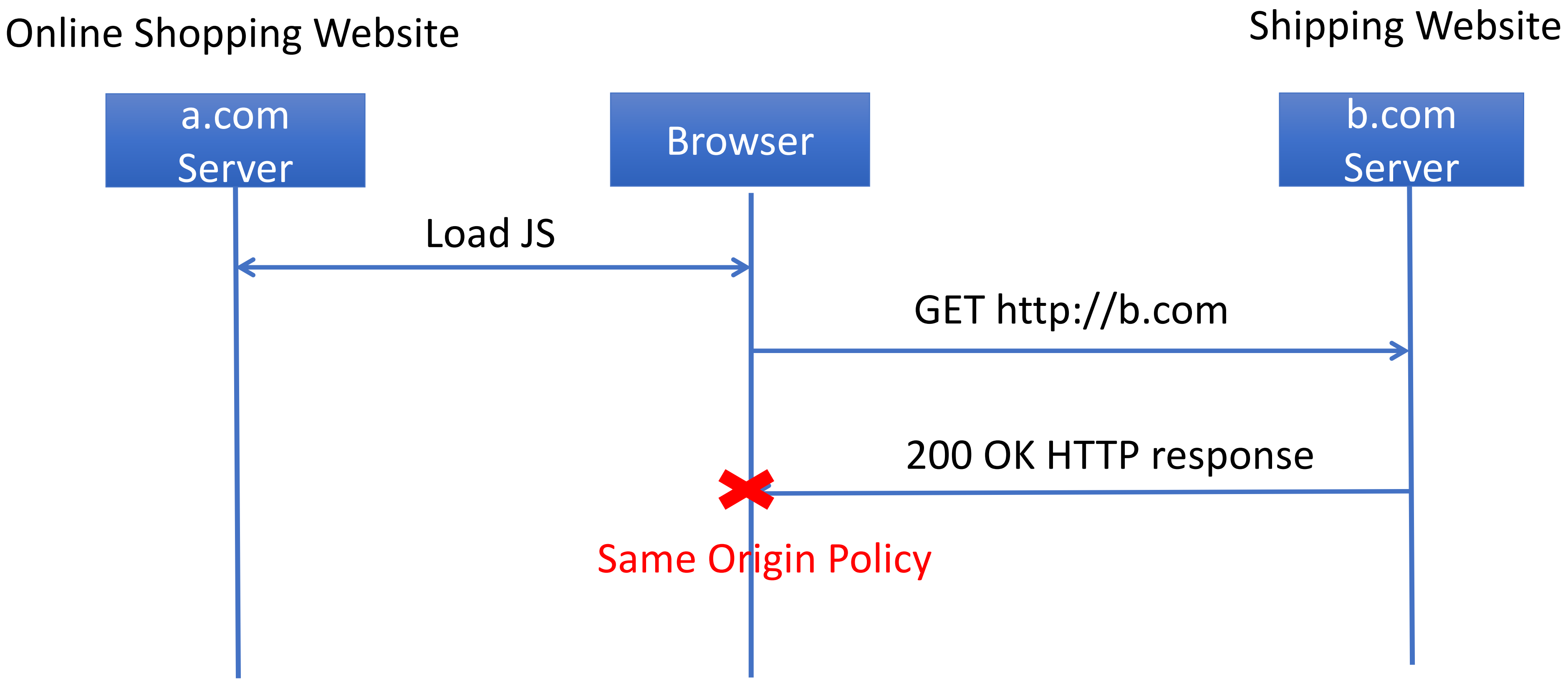
- Conducted an empirical study on CORS
  - Including its design, implementation and deployment
- Discovered a number of security issues
  - 4 categories of browser-side issues
  - 7 categories of sever-side issues
- Conducted a large-scale measurement on popular websites
  - 27.5% of CORS configured websites have insecure CORS configuration
- Proposed mitigations and some of them have been adopted by web standard and major browsers.

Contents

- Web SOP and CORS background
- Our discovery: CORS security issues
  - Browser-side: overly permissive sending
  - Server-side: CORS misconfigurations
- CORS real-world deployments
  - Our large scale measurement
- Disclosure and Mitigation

Web & CORS background

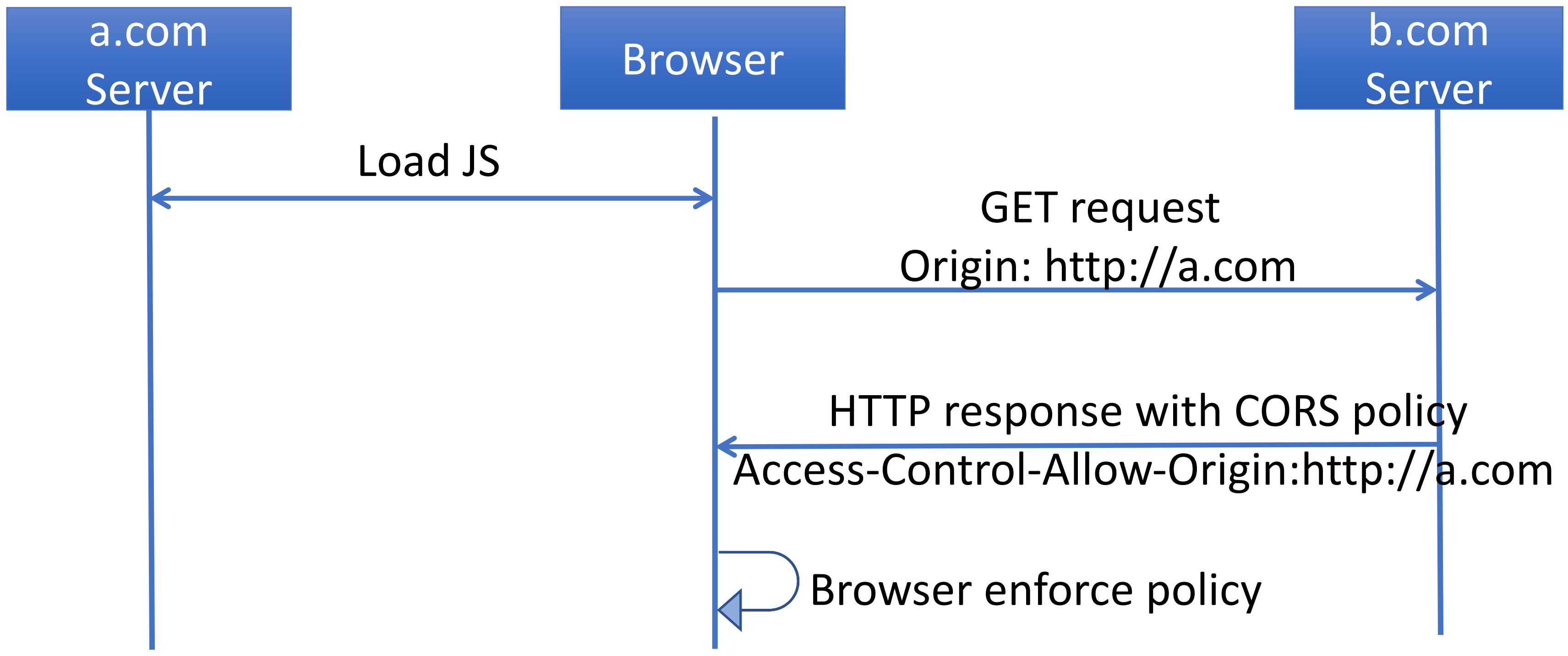
The default SOP prevents cross origin reading



Developers need cross origin reading!

Cross origin resource sharing (CORS)

- Explicit authorization access control mechanism
  - Browsers support(2009), W3C standard(2014)



CORS JavaScript interfaces (e.g. XHR)

- CORS allows JS to customize method, header and body

```
var xhr=new XMLHttpRequest();
xhr.open("PATCH", "http://b.com/r", true);
xhr.setRequestHeader("X-Requested-With", "XMLHttpRequest ");
xhr.withCredentials = true;

xhr.send("any data");
```

Document of a.com

But this interface is very powerful, and may break CSRF defense of many websites.

Simple requests in CORS standard

- Two categories of requests
  - Simple request: can be sent directly
  - Non-simple request: not to cover this in this talk (refer to the paper)

- A simple request must satisfy all of the three conditions：
  1. Request method is *HEAD*, *GET* or *POST*.
  2. Request headers are not customized, except for 9 whitelisted headers: *Accept, Accept-Language, Content-Language, Content-Type*, etc.
  3. Content-Type header value is one of three specific values: *“text/plain”, “multipart/form-data”,* and *“application/x-form-uri-encoded”.*

Browser-side Issues: Overly Permissive Sending Permissions

(4 categories of issues)

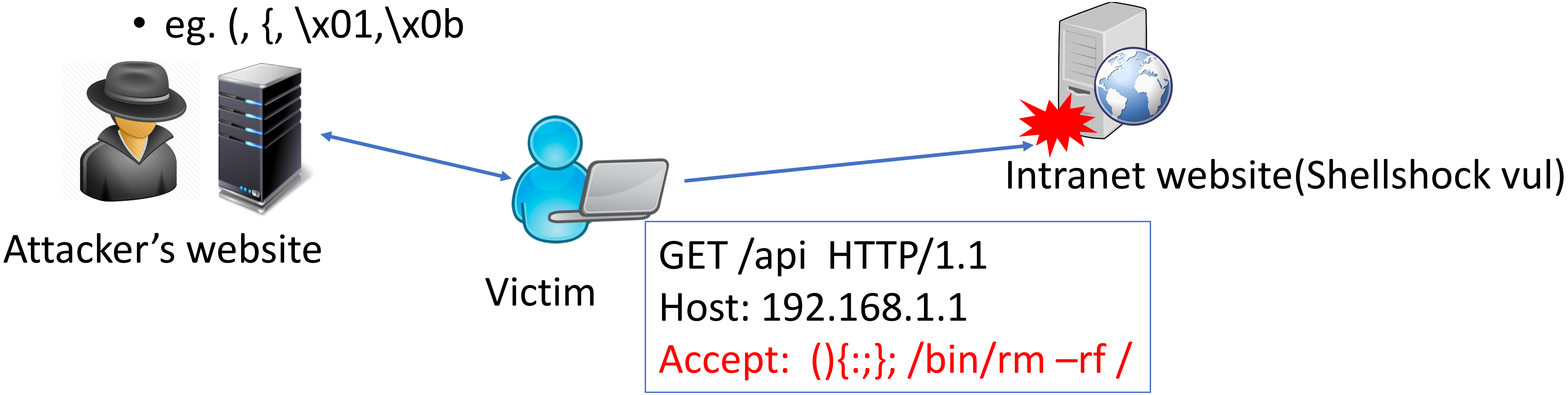
Overly permissive request headers and bodies

- CORS relax send restrictions unintentionally, allowing malicious customization of HTTP headers and bodies
- The relaxation can be exploited by attackers

Problems	Attacks
P1. Overly permissive header values	RCE attack on intranet servers
P2. Few limitations on header size	Infer cookie presence for ANY website
P3. Overly flexible body values	Attack MacOS AFP server
P4. Few limitations on body format	Exploit previously unexploitable CSRF

P1. Overly permissive header values

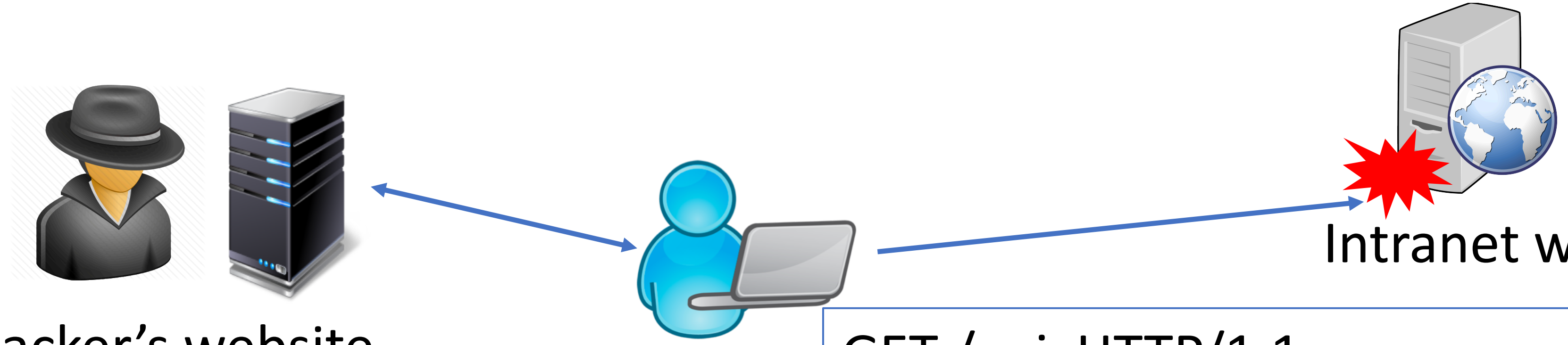
- CORS allows JavaScript to modify 9 whitelisted headers.
- CORS imposes few limitations on header values except "Content-Type"
  - eg. (, {, \x01,\x0b



Affected browser(4/5):

P1. Overly permissive header values

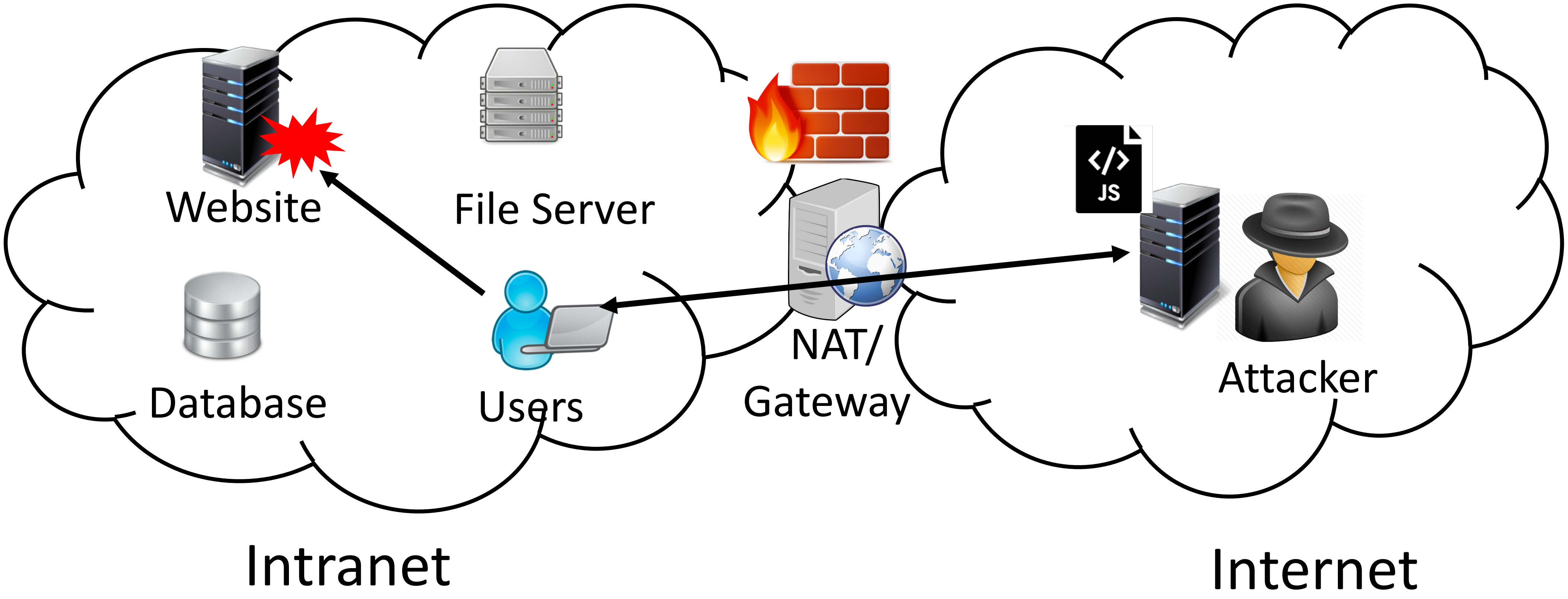
- CORS restricts “Content-Type” to three specific values
  - But the restriction can be bypassed due to browsers’ implementation flaws.



GET /api HTTP/1.1
Host: 192.168.1.1
Content-Type: text/plain ; %{(apache struts exploit)}

Affected browsers(5/5):

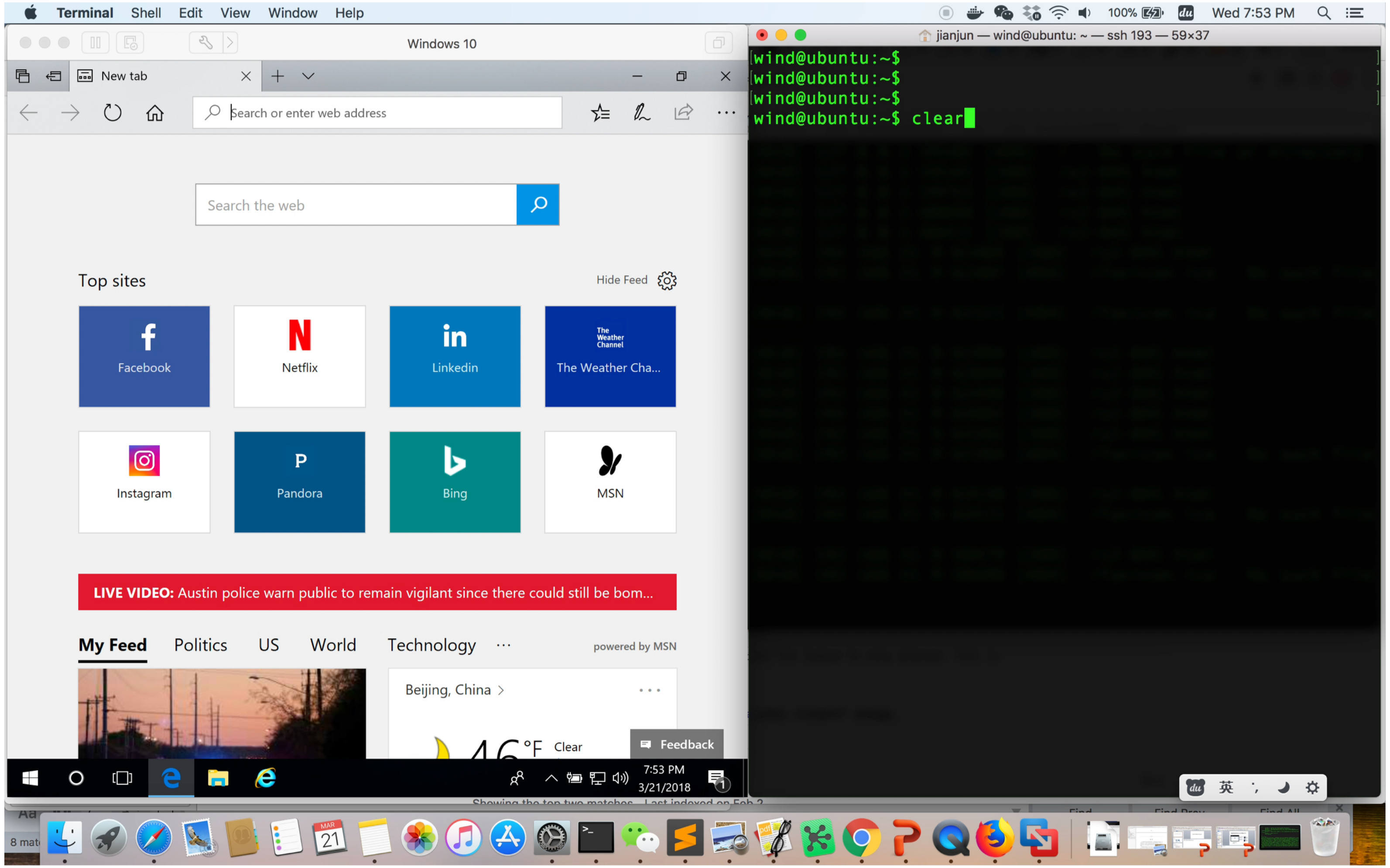
Case study: obtain a shell on Intranet server by exploiting browsers



Demo: Obtain a shell on Intranet server by exploiting browsers(https://youtu.be/jO6hoXyXVqk)

Victim’s browser in Intranet

Attacker in Internet



P2. Few limitations on header size

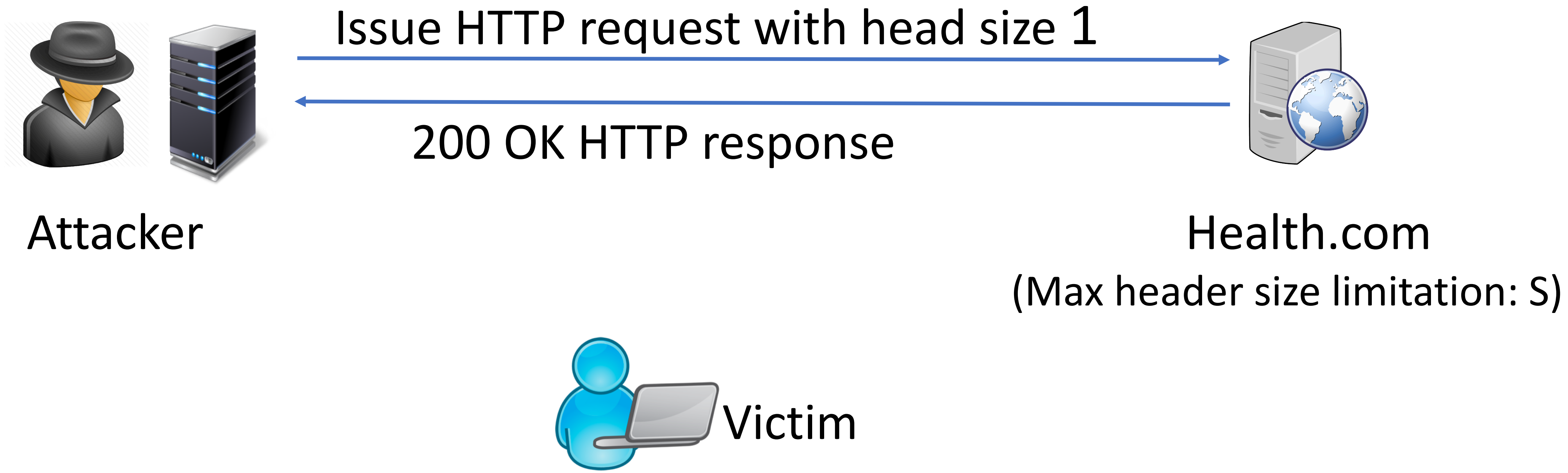
- Both HTTP and CORS standards have no explicit limit on request header sizes.
- Browsers’ header size limitation are more relaxed than servers.

Browser	Limitation	Server	Limitation
Chrome	>16MB/>16MB	Apache	8KB/<96KB
Edge	>16MB/>16MB	IIS	16KB/16KB
Firefox	>16MB/>16MB	Nginx	8KB/<30KB
IE	>16MB/>16MB	Tomcat	8KB/8KB
Safari	>16MB/>16MB	Squid	64KB/64KB

- Case study 2: Remotely infer cookie presence for ANY website.

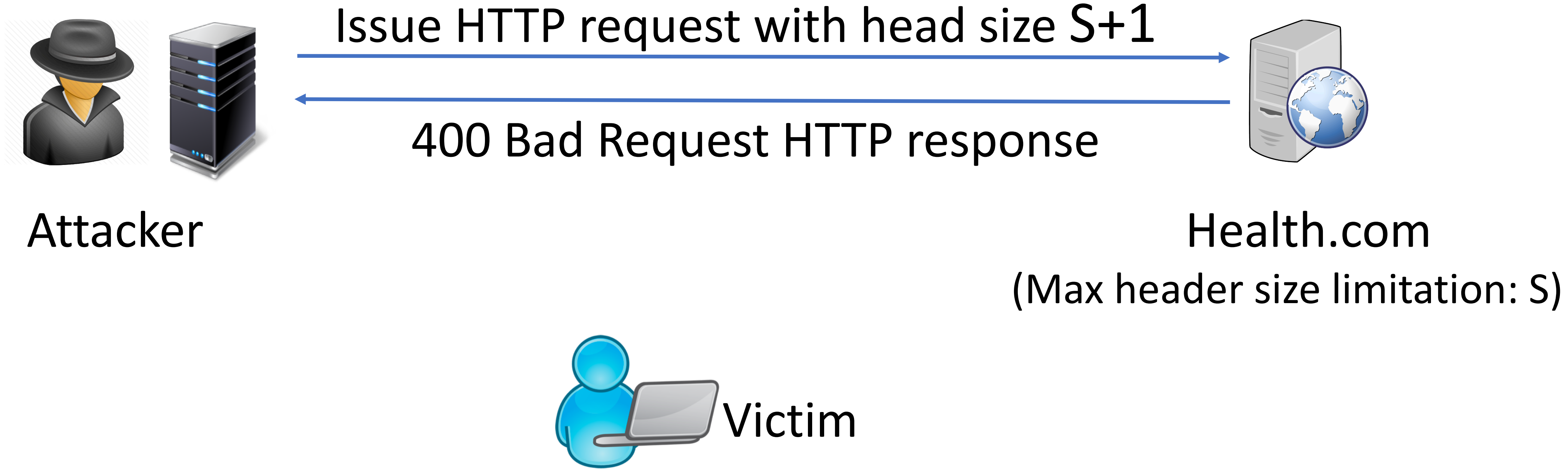
Remotely infer cookie presence for ANY website

Step 1: Measure the header size limit of target server



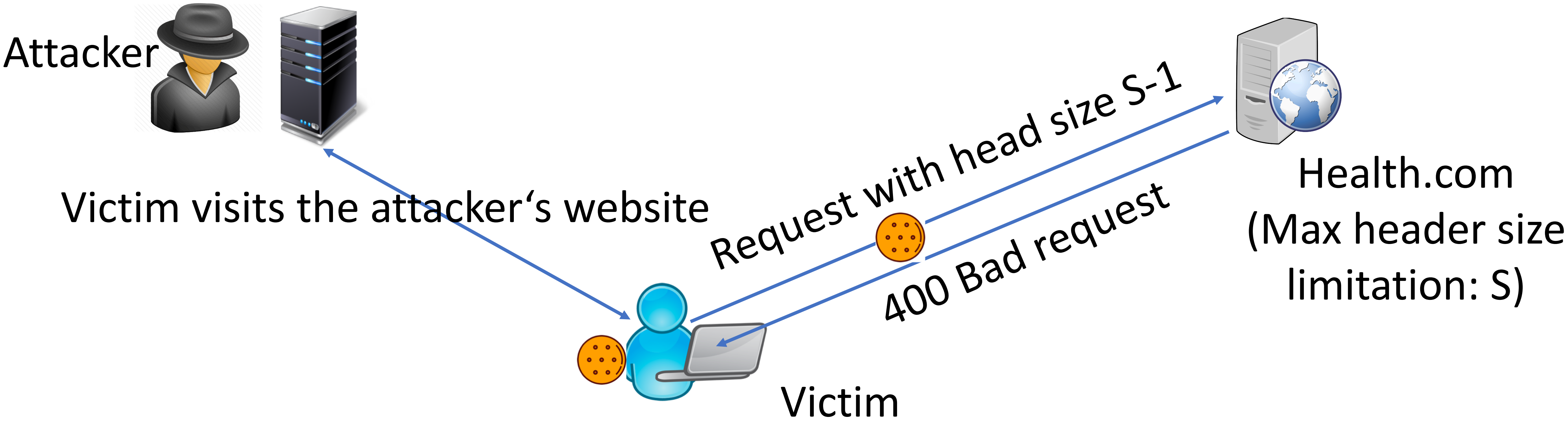
Remotely infer cookie presence for ANY website

Step 1: Measure the header size limit of target server



Remotely infer cookie presence for ANY website

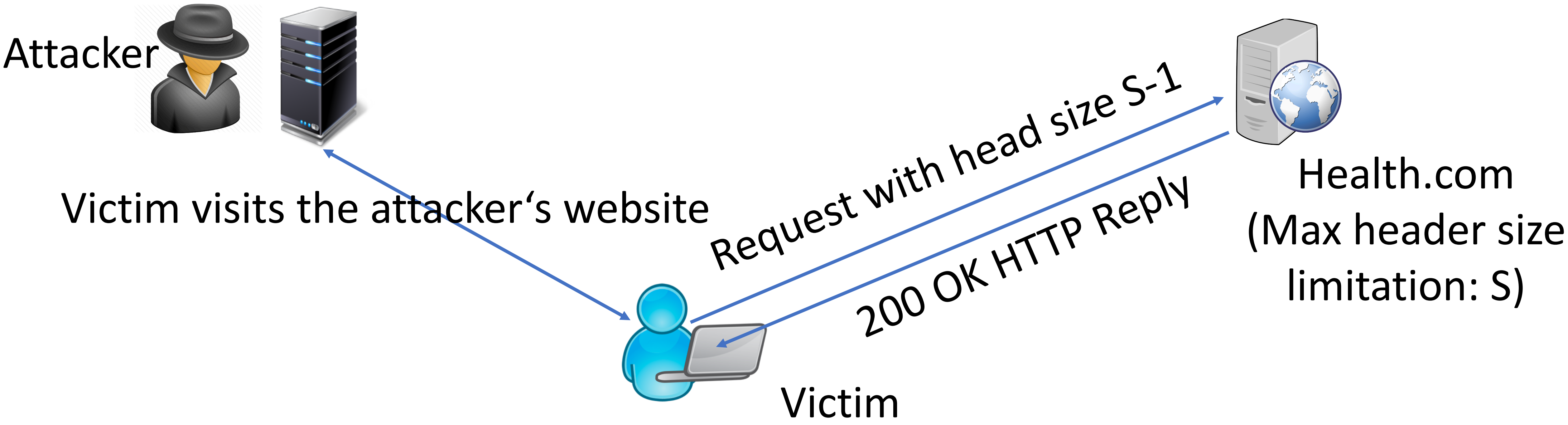
Step 2: Send request from the victim's browser with header size slightly smaller than the measured limit.



When Cookie is present, "400 Bad request" is returned

Remotely infer cookie presence for ANY website

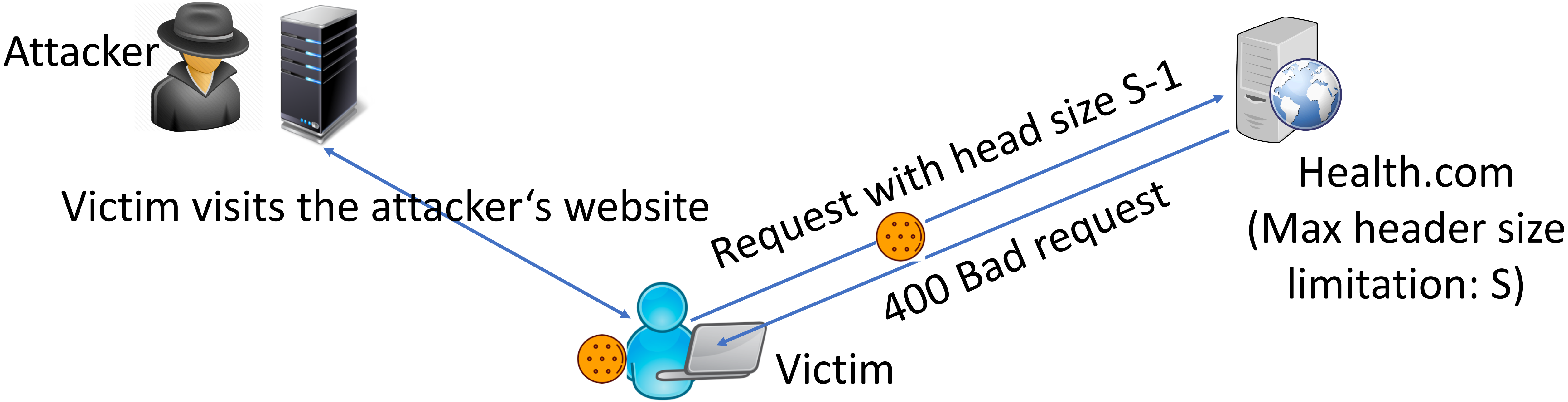
Step 2: Send request from the victim's browser with header size slightly smaller than the measured limit.



When Cookie is not present, "200 OK" is returned

Remotely infer cookie presence for ANY website

Step 3: Infer the response status through timing channel.



- One general timing channel is response time.
- In Chrome, *Performance.getEntries()* directly exposes it.

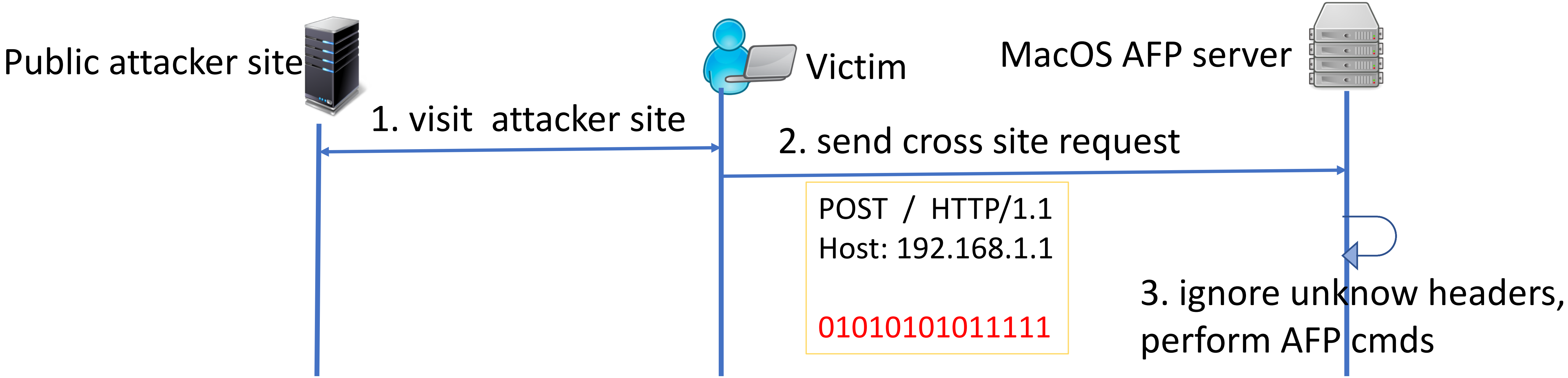
Remotely infer cookie presence for ANY website

- The presence of a cookie can leak private information.
  - victim’s health conditions
  - Financial considerations
  - Political preferences

Affected browsers(5/5):

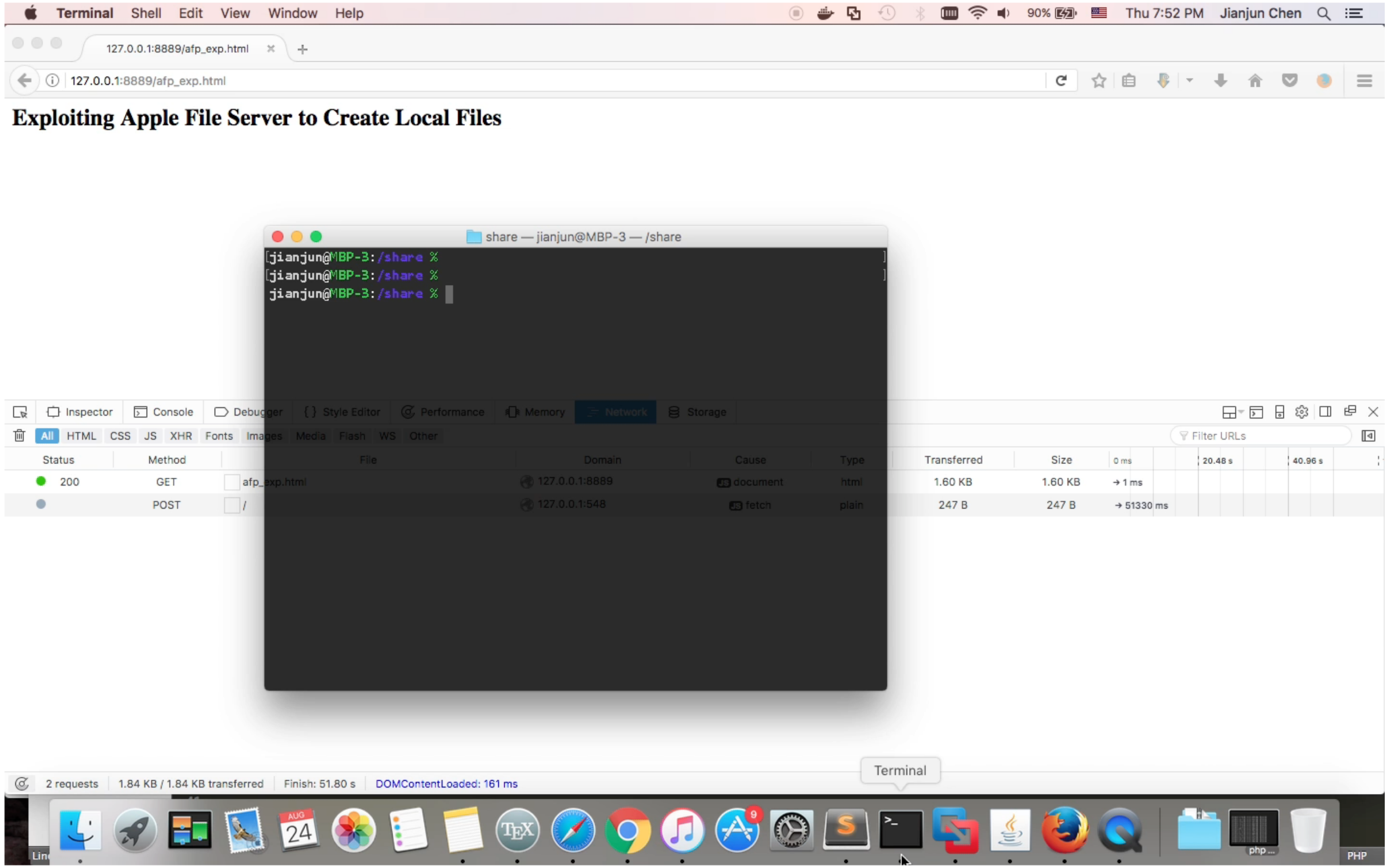
P3. Overly flexible body values

- CORS impose no limitations on the values of request body
  - CORS allows JavaScript to construct ANY binary data in request body



Affected browsers(5/5):

Demo: exploiting MacOS built-in Apple file server to create local files(https://youtu.be/WXIy94prfvs)



Server-side issues: CORS misconfigurations
(7 categories of issues)

Inspired by these previous work:

[1] James Kettle, “Exploiting CORS misconfigurations for Bitcoins and bounties”, AppSecUSA 2016

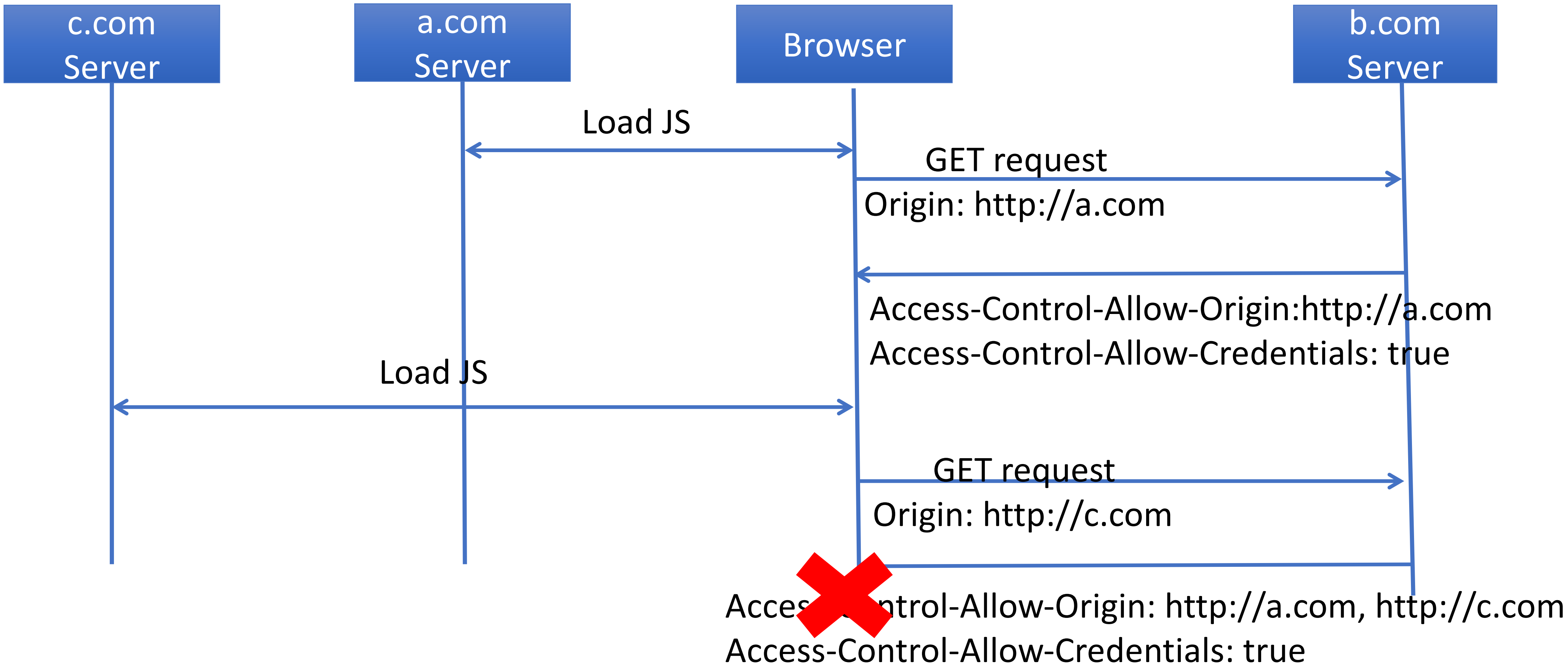
[2] Evan Johnson, “Misconfigured CORS and why web appsec is not getting easier”, AppSecUSA 2016

[3] Von Jens Müller, "CORS misconfigurations on a large scale"

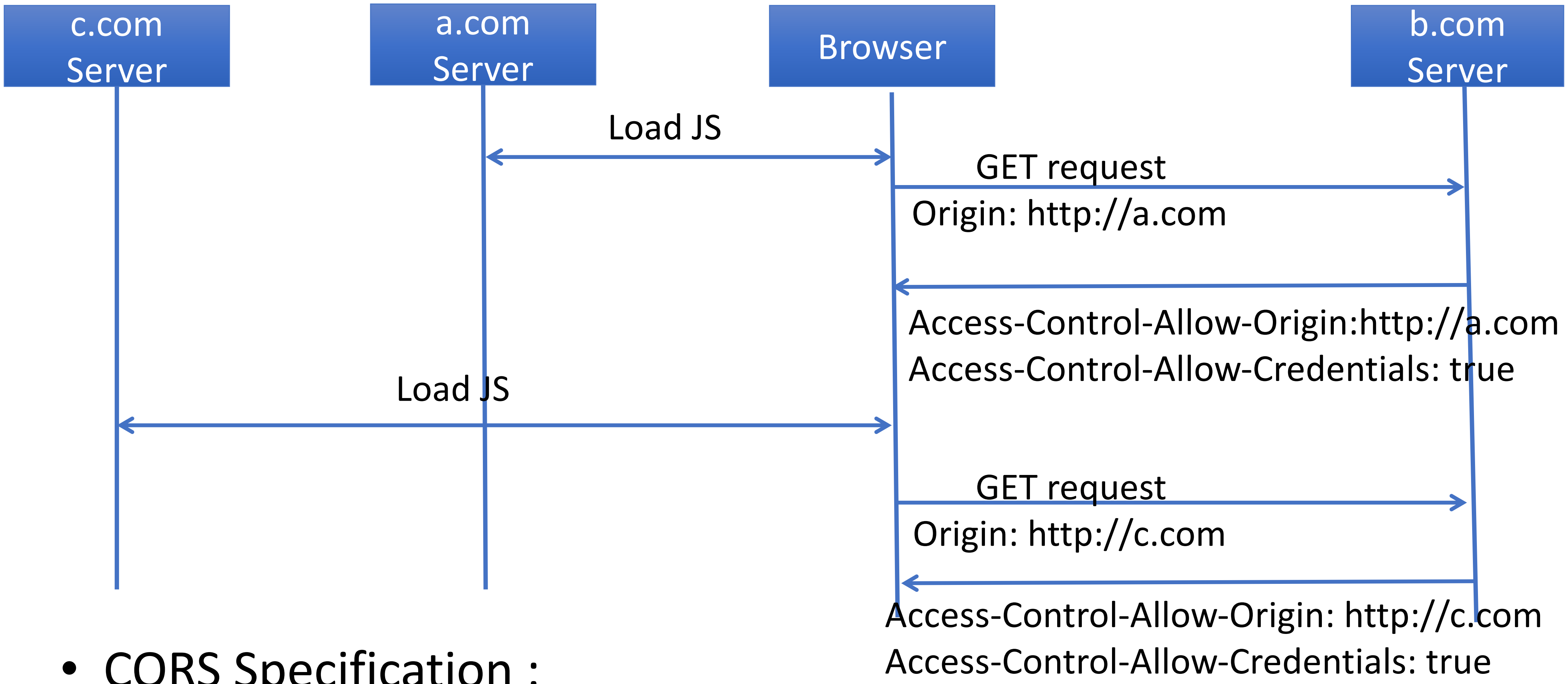
CORS misconfigurations

1. Origin reflection
2. Validation mistakes
3. HTTPS trust HTTP
4. Trust null
5. Wildcard origin with credentials
6. Trust all of its own subdomains
7. Lack of “Vary: Origin”

How does CORS policy work?

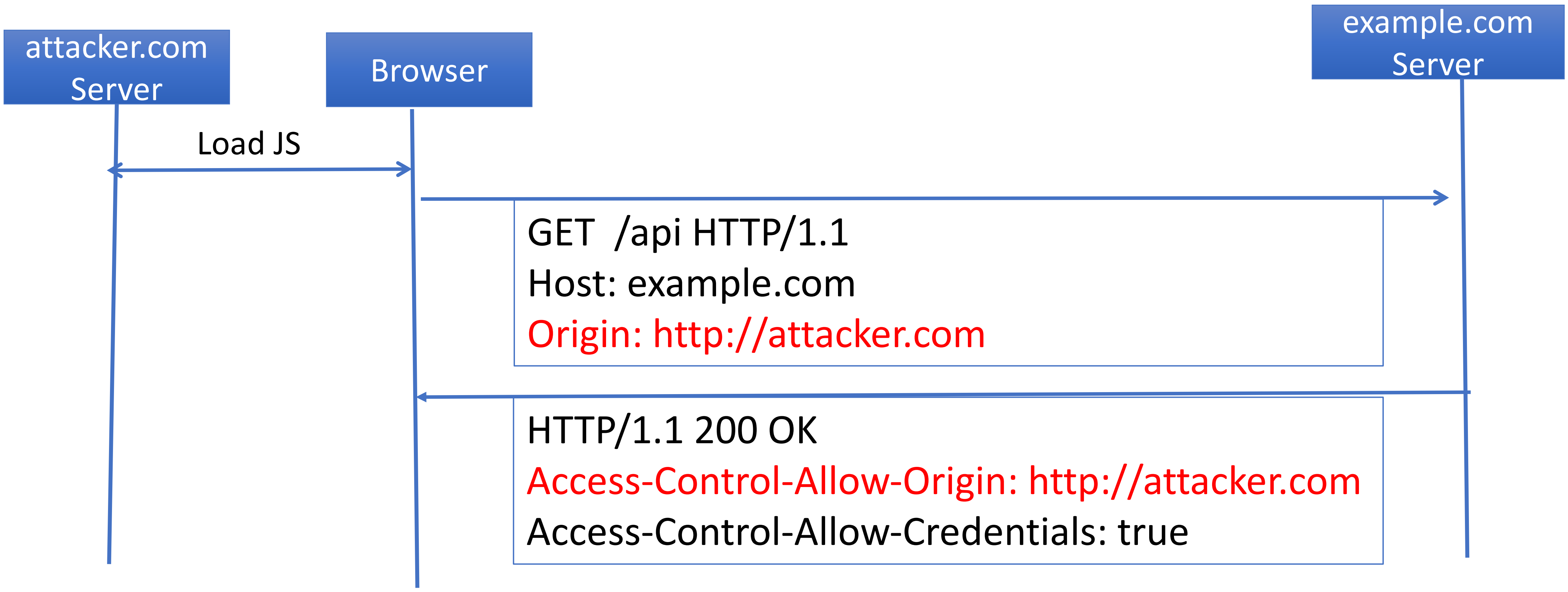


How does CORS policy work?



- CORS Specification :
  - Access-Control-Allow-Origin = single origin, null or *

P1: Origin reflection



P2: Validation mistakes

1) Prefix Match:

- A example of insecure Nginx configuration: $

```
if ($http_origin ~ “http://(example.com|foo.com)”) {
    add_header "Access-Control-Allow-Origin" $http_origin;
}
```

```
GET  /api HTTP/1.1
Host: www. example.com
Origin: http://example.com.evil.com
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http:// example.com.evil.com
Access-Control-Allow-Credentials: true
```

P2: Validation mistakes

2) Suffix Match

- A example of insecure CORS policy generation :

```
if (reqOrigin.endswith(“example.com”) ) {
    respHeaders[“Access-Control-Allow-Origin”] = reqOrigin
}
```

```
GET  /api HTTP/1.1
Host: www.example.com
Origin: http://attackexample.com
```

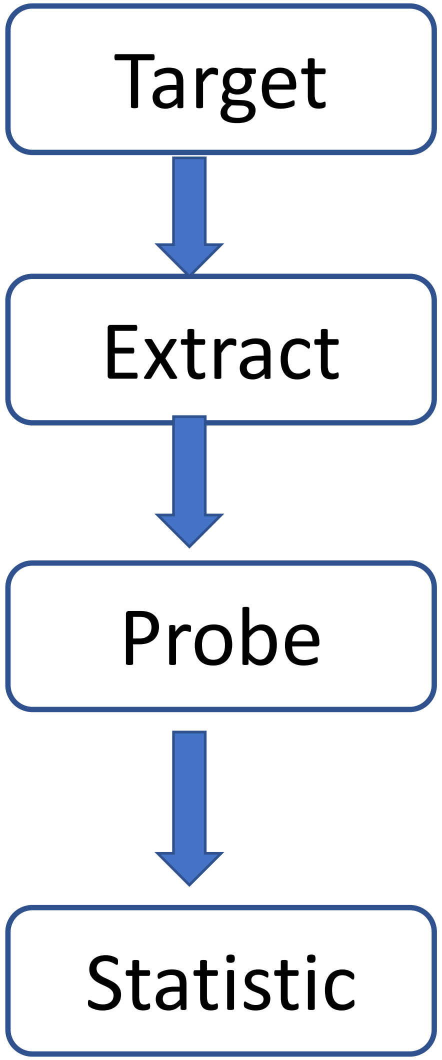
```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://attackexample.com
Access-Control-Allow-Credentials: true
```

P3: HTTPS trust HTTP

- HTTPS provides confidentiality protection
  - Prevent man-in-the-middle(MITM) attackers

- When a HTTPS site configured to trust its HTTP site
  - eg. Access-Control-Allow-Origin: http://example.com
- A MITM attacker can first hijack HTTP site, and then steal secrets on HTTPS by issuing cross origin requests

CORS measurement



Alexa Top 50,000 websites

Extract 97,199,966 subdomains

- From Qihoo 360 network security lab

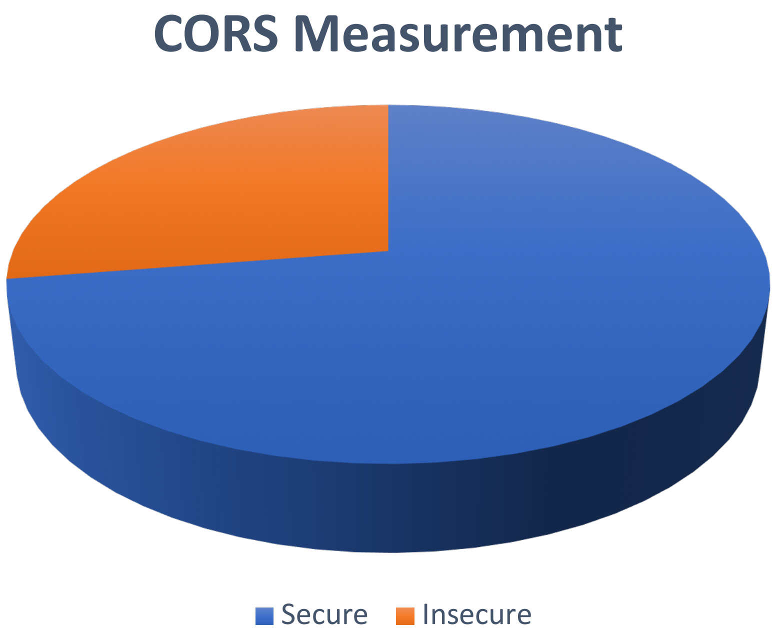
Actively probe CORS configurations

```
GET  /api HTTP/1.1
Host: www.example.com
Origin: example.com.attacker.com
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://example.com.attacker.com
Access-Control-Allow-Credentials: true
```

Measurement results

- 481,589 subdomains configured CORS
- 132,476 subdomains(27.5%) have insecure configurations



Categories	Count	Percentage
HTTPS trust HTTP	61,347	12.7%
Trust all subdomains	84,327	17.5%
Reflecting origin	15,902	3.3%
Prefix match	1,876	0.4%
Suffix match	32,575	6.8%
Substring match	430	0.1%
Not escaping "."	890	0.2%
Trust *null*	3,991	0.8%
Total	132,476	27.5%

Disclosure & Response

Response by CORS standard organization

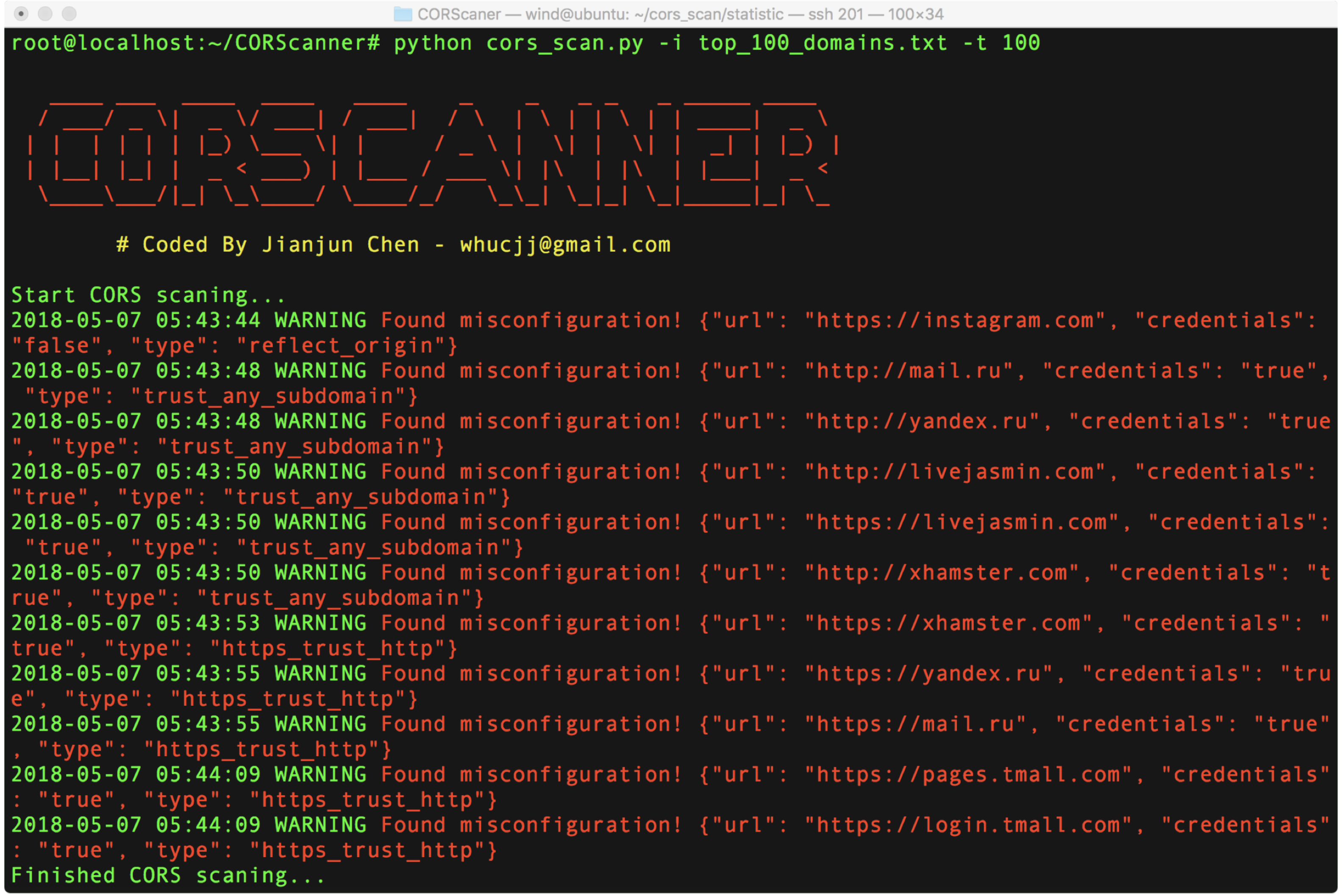
- For cross origin sending attacks
  - Accepted some of our suggestions and made corresponding changes to the CORS specification
  - Added more restrictions on CORS simple requests, e.g. restricting header length, restricting access to unsafe ports
  - Acknowledged us in the CORS specification.
- For CORS misconfigurations issues
  - Misconfigured websites should fix those issues by themselves.
  - Agreed to add a security consideration section in the standard

Response by vendors

- Browsers
  - **Chrome and Firefox:** have blocked port 548 and 427, and are implementing specification changes.
  - **Safari:** are testing those changes with a beta testing program.
  - **Edge/IE:** acknowledged our report.
- CORS frameworks and Websites
  - Tomcat(CVE-2018-8014), Yii and Go-CORS fixed
  - Some(e.g., nasdaq.com, sohu.com, mail.ru) have fixed the issues.
- We provide an open-source tool for automatic CORS configuration checking.

https://github.com/chenjj/CORScanner

CORScanner (https://github.com/chenjj/CORScanner)



Summary

- An empirical security study on CORS
- Discovered multiple security issues in browsers and specs
  - 4 categories of browser-side issues
  - 7 categories of server-side issues
- Conducted a large-scale measurement
  - 27.5% of CORS configured websites have insecure CORS configuration
- Proposed mitigations
  - Some of them have been adopted by web standard and major browsers.

Thank you!

Twitter: whucjj

Blog: https://www.jianjunchen.com