CSE551: Advanced Computer Security

12. Client-side Security

Seongil Wi

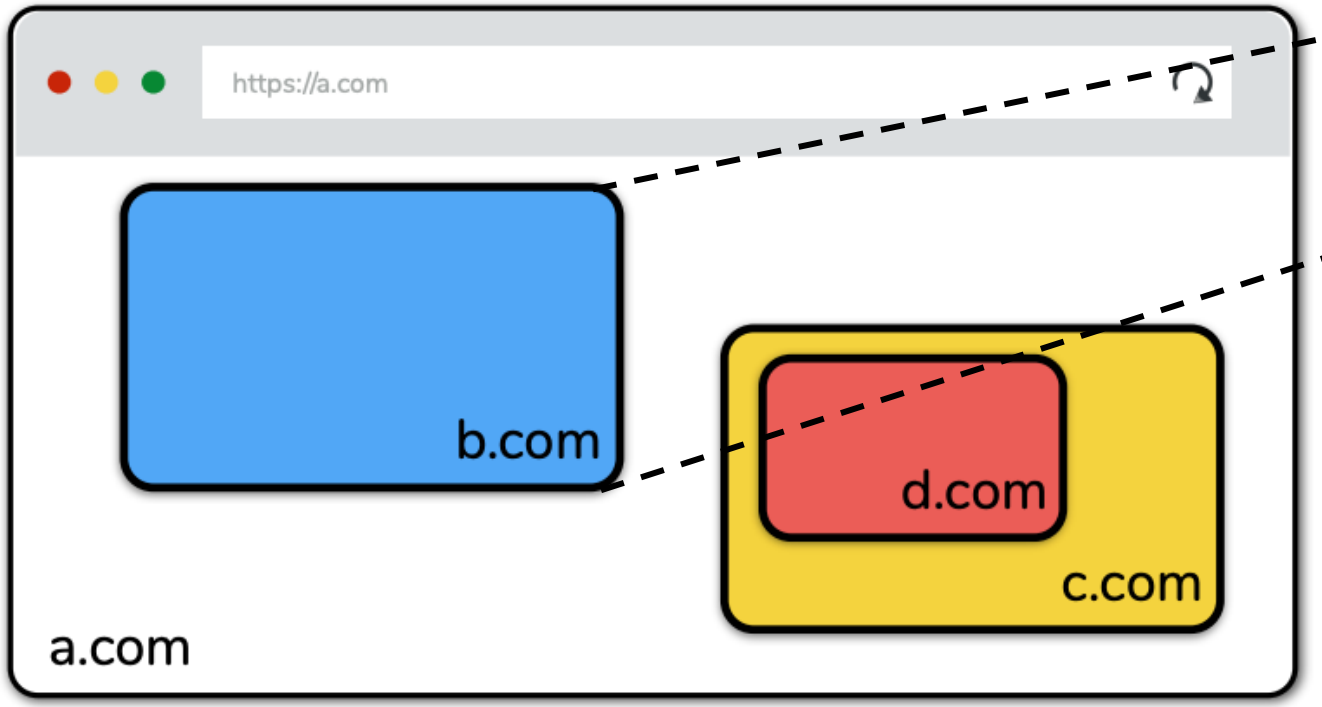
Department of Computer Science and Engineering

HW2

- Write a critique for the two papers:
  - WYSINWYX: What you see is not what you eXecute, ***TOPLAS 2005***
  - The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86), ***CCS 20107***
- Due: November 11, 11:59PM

Recap: Nested Execution Model

- Windows may contain frames from different sources
  - **Frame**: rigid visible division
  - **iFrame**: floating inline frame




```
<iframe src="b.com">
</iframe>
```

Recap: Web Threat Models

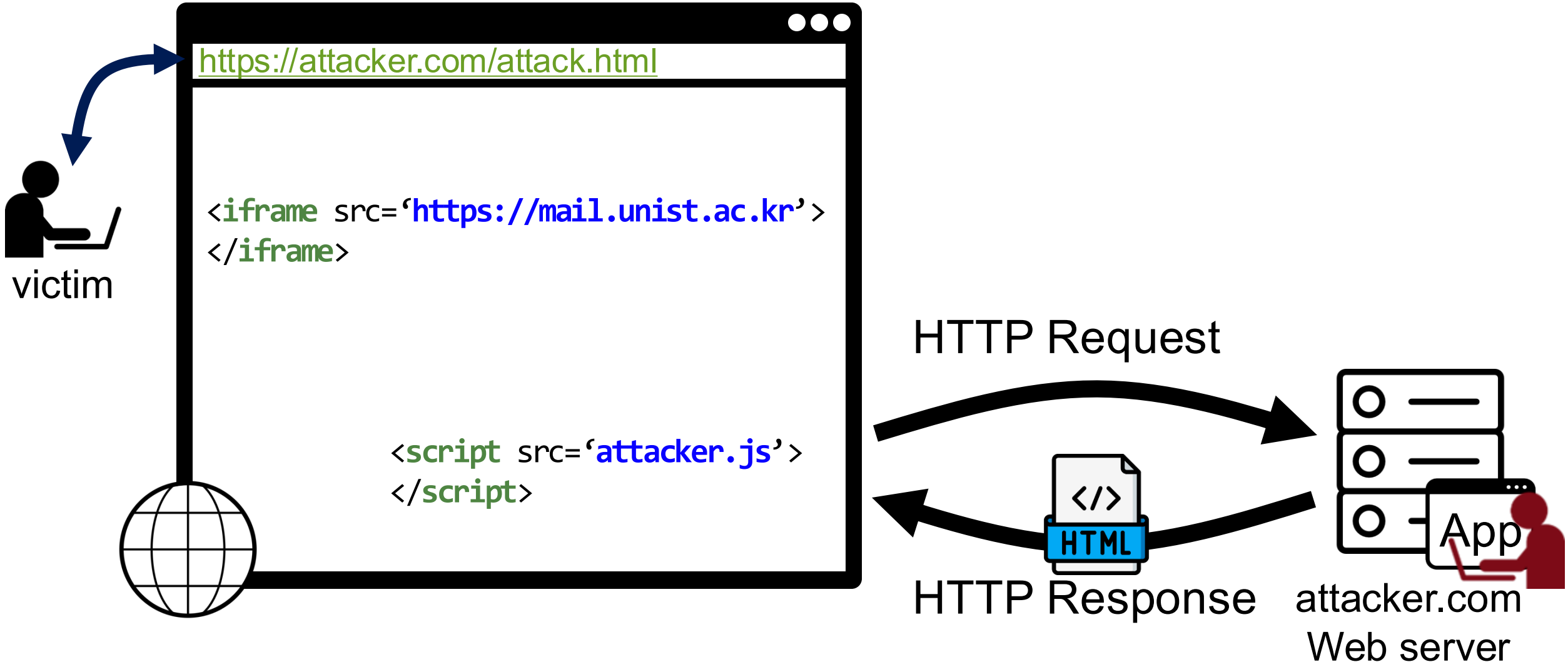
- **Network attacker**: resides somewhere in the communication link between client and server
  - Passive: evasdropping
  - Active: modification of messages, replay…

- **Remote attacker:** can connect to remote system via the network
  - Mostly targets the server

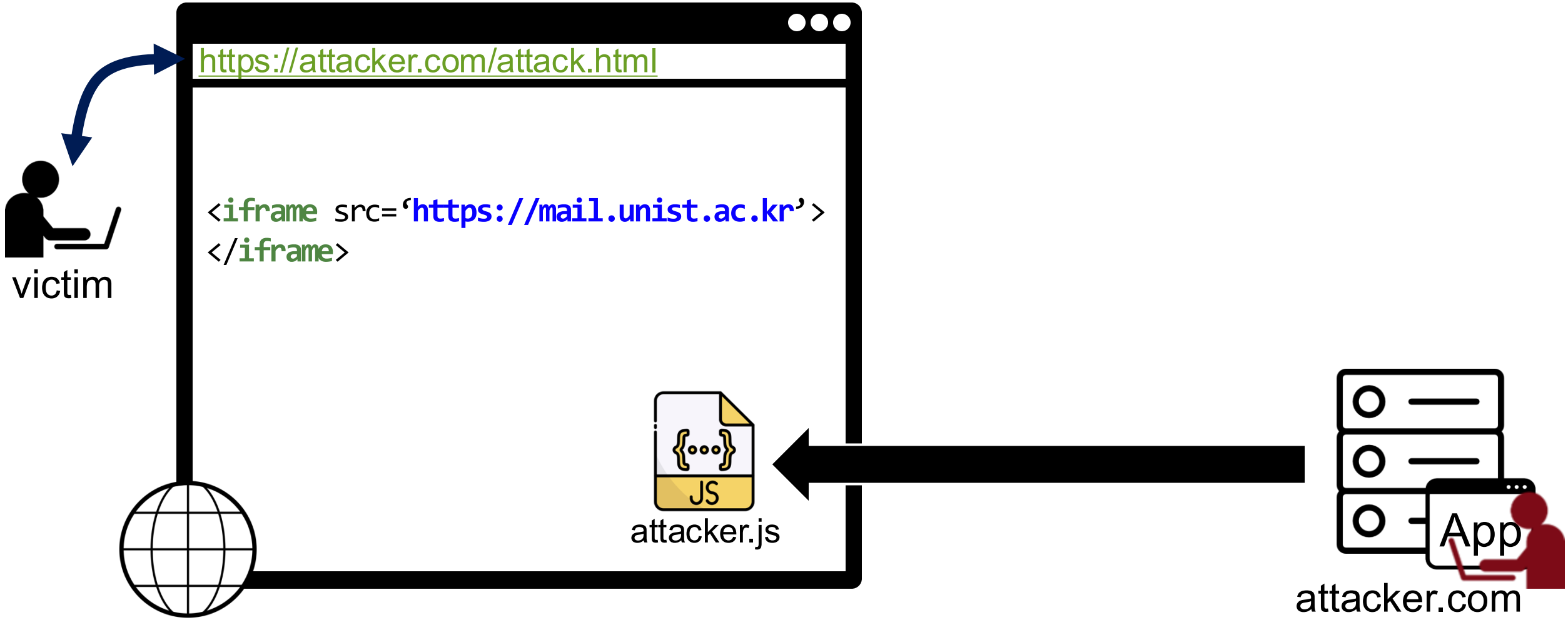
- **Web attacker**: controls attacker.com
  - Can obtain SSL/TLS certificates for attacker.com
  - Users can visit attacker.com

Web Attacker

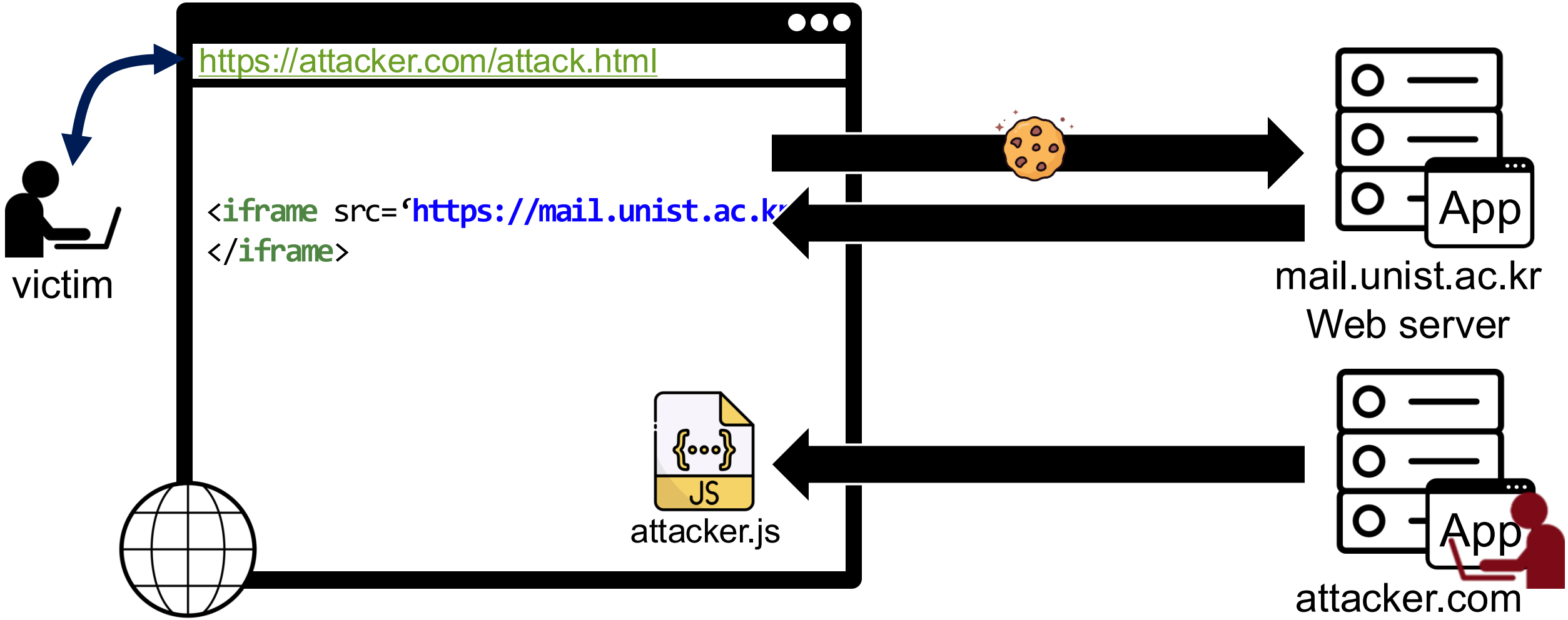
Motivation of the Client-side Security



Motivation of the Client-side Security



Motivation of the Client-side Security



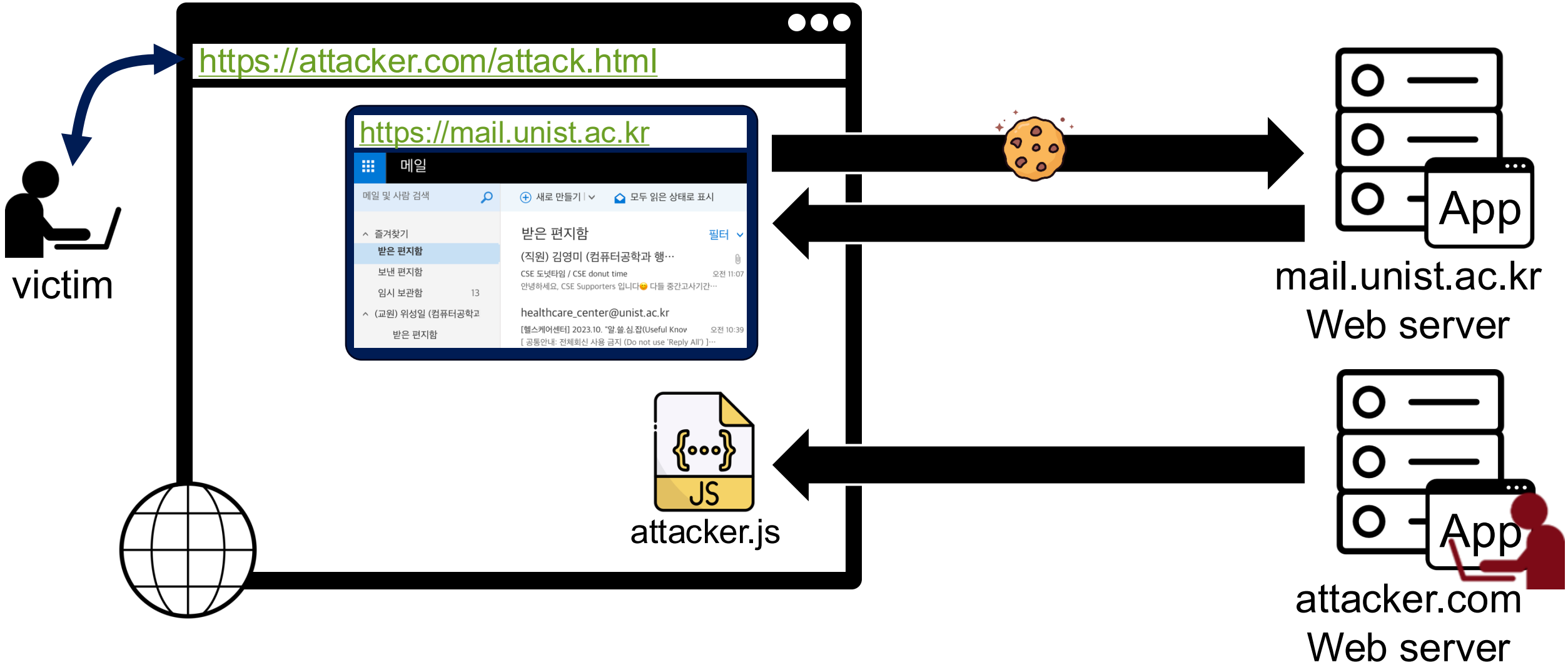
Cookie: Making HTTP Stateful

- Store a server-created file (cookie) in the browser

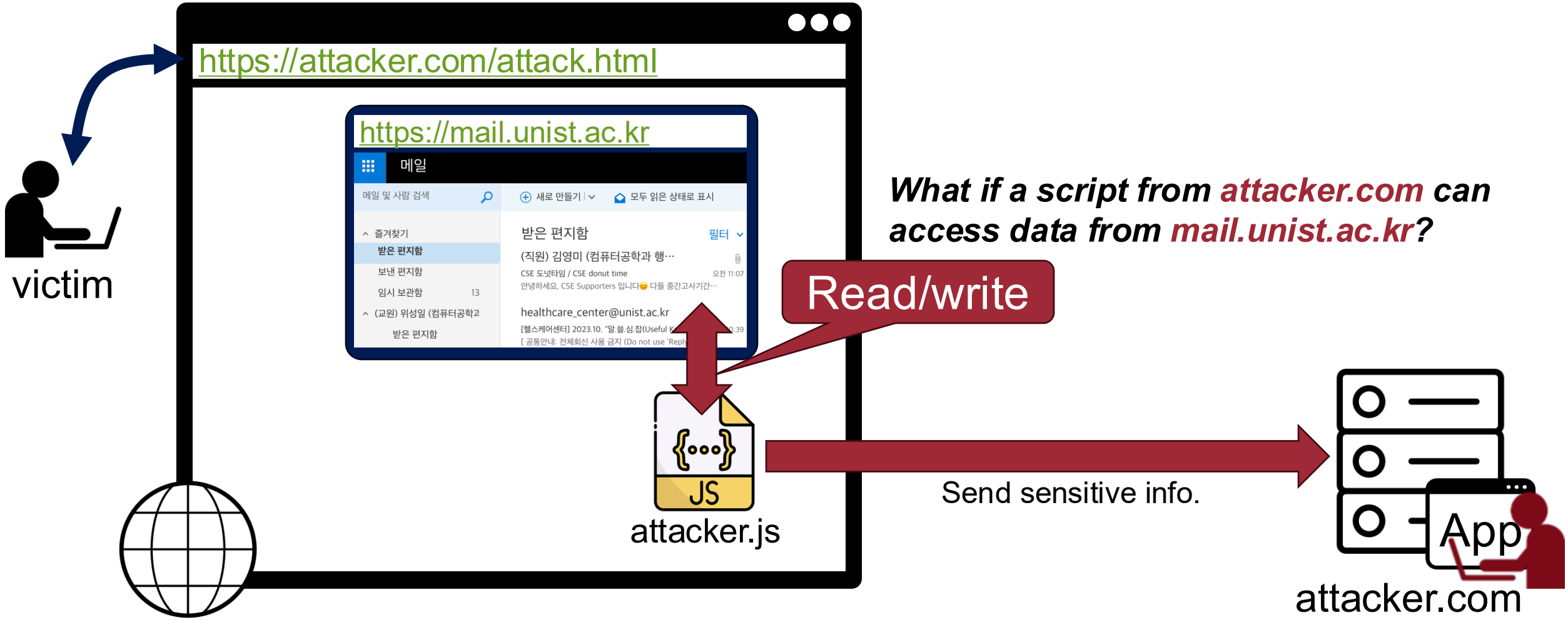
- Examples
  - Authentication (log in)
  - Personalization (language preference, shopping cart)
  - User tracking

- We can access all cookies for current document by `alert(document.cookie)`

Motivation of the Client-side Security



A World Without Separation between Sites

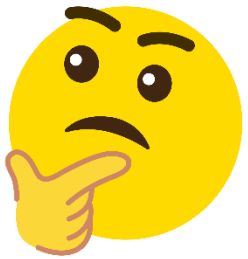


A World Without Separation between Sites

What if a script from attacker.com can access data from mail.unist.ac.kr?

It would be able to read your emails, private messages, authentication session cookies

Motivation of the Client-side Security

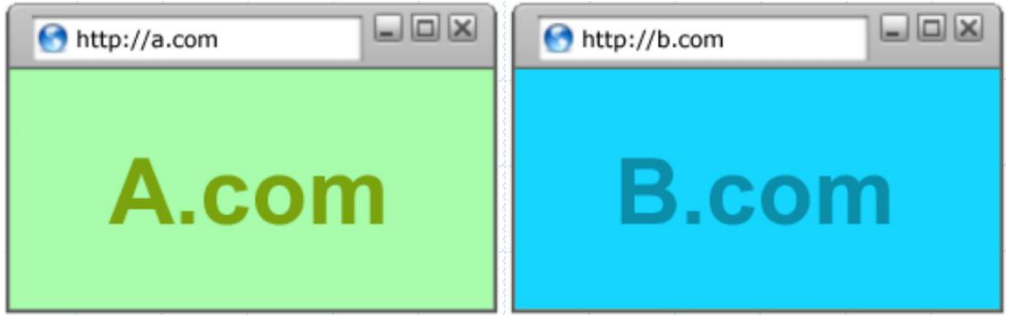


How can we prevent such malicious behaviors?

Policy Goals

- Safe to visit an evil website

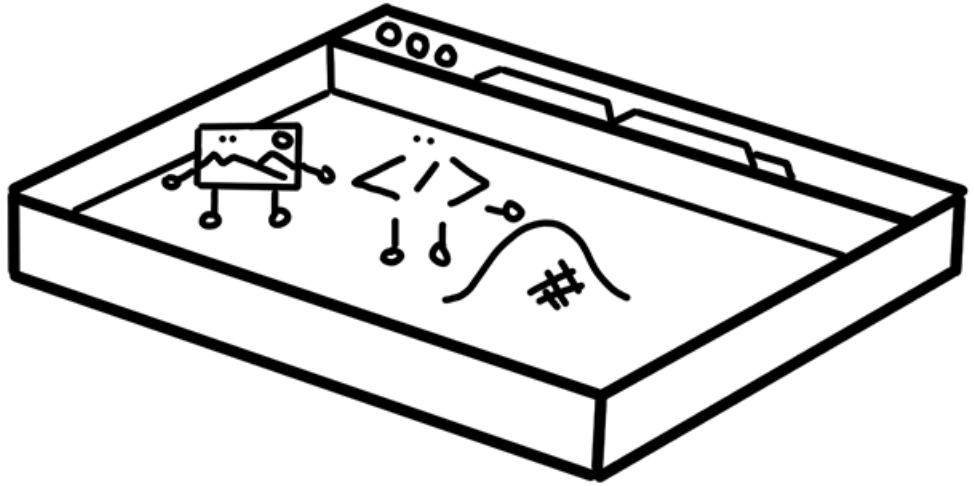
- Safe to visit two pages at the same time
  - Address bar distinguishes them



- Allow safe delegation

Browser Sandbox

- No direct file access, limited access to OS
- Goal: Safely execute JavaScript code provided by a remote website
  - Isolated process when HTML rendering and JavaScript execution



Browser Sandbox Escaping Vulnerabilities

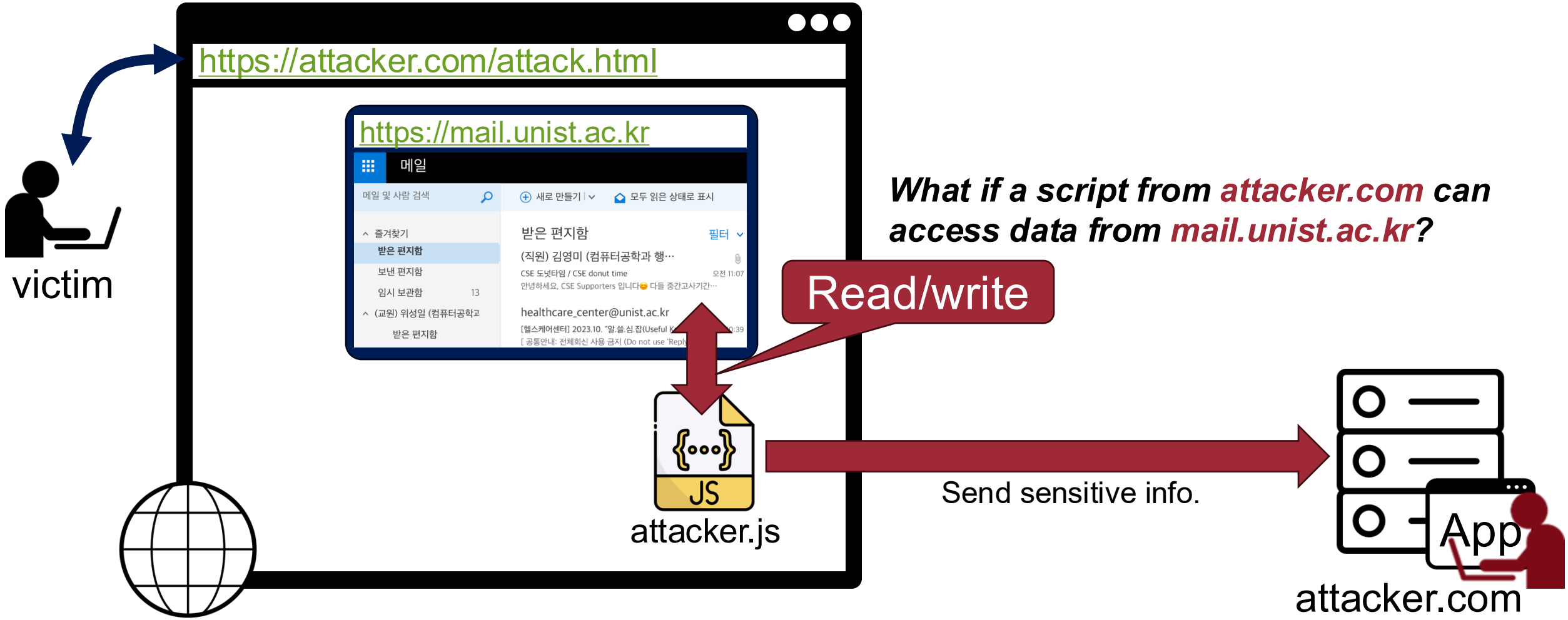
- Related to memory-level vulnerabilities, including Use-After-Free (UAF), heap overflow,…

- CVE-2013-6632
- CVE-2014-3188
- CVE-2015-6767
- CVE-2019-5850

Same Origin Policy (SOP)

- One of the browser sandboxing mechanism
- The basic security model enforced in the browser

A World Without Separation between Sites

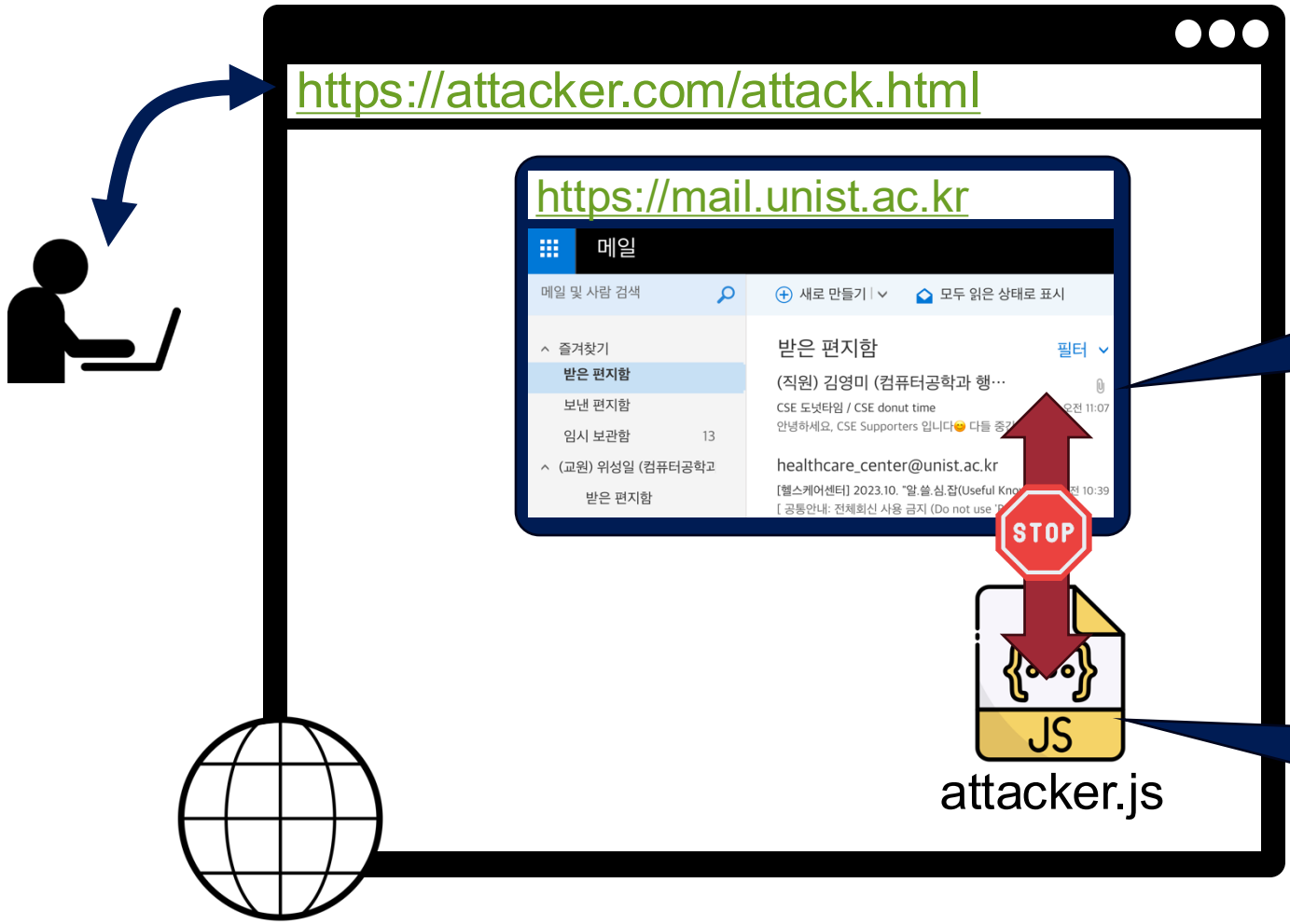


Same Origin Policy (SOP)

- Restricts scripts on one origin from accessing data from another origin

Same Origin Policy (SOP)

- Restricts scripts on one origin from accessing data from another origin



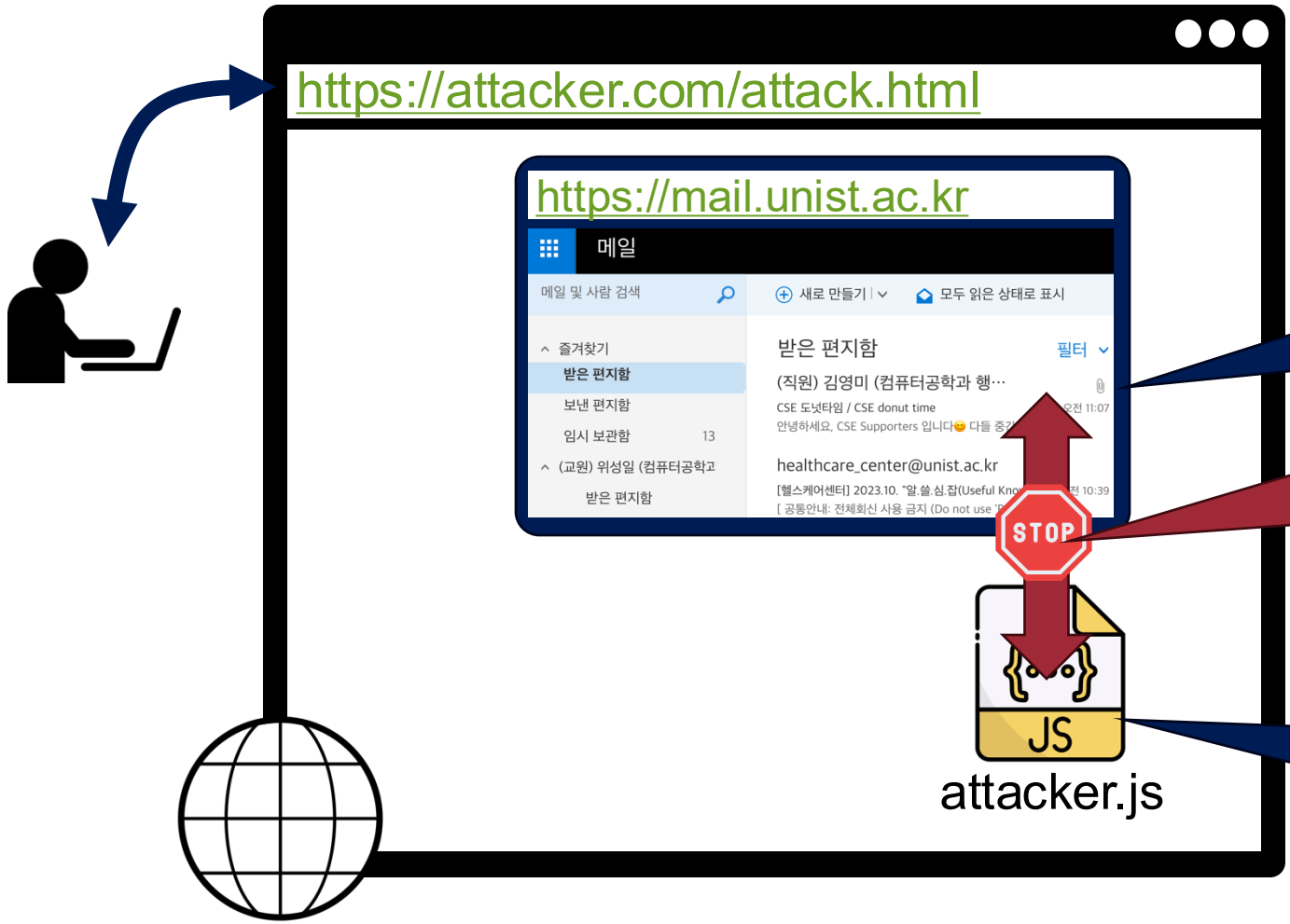
Any resource has its own origin

Resources located at the *https://mail.unist.ac.kr* origin

JavaScript runs with a *https://attacker.com* origin

Same Origin Policy (SOP)

- Restricts scripts on one origin from accessing data from another origin



Any resource has its own origin

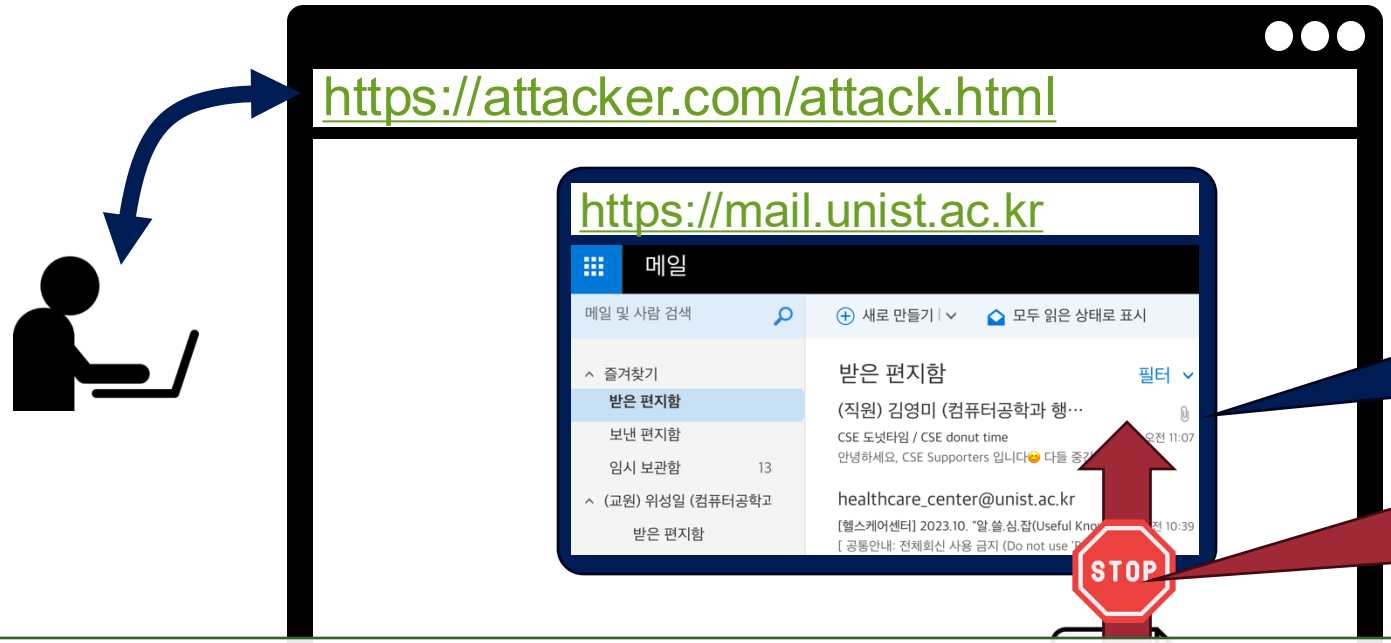
Resources located at the *https://mail.unist.ac.kr* origin

A JS runs with an origin cannot access other origin resources

JavaScript runs with a *https://attacker.com* origin

Same Origin Policy (SOP)

- Restricts scripts on one origin from accessing data from another origin



Any resource has its own origin

Resources located at the *https://mail.unist.ac.kr* origin

A JS runs with an origin cannot access other origin resources

Uncaught DOMException: Permission denied to access property "document" on cross-origin object

Same Origin Policy (SOP)

- Restricts scripts on one origin from accessing data from another origin

- The basic security model enforced in the browser

- Basic access control mechanism for web browsers
  - All resources such as DOM, cookies, JavaScript has their own origin
  - SOP allows a subject to access only the objects from the same origin

What is an Origin?

- **Origin = Protocol + Domain Name + Port**
  - `origin = protocol://domain:port`
- Any resource has its own origin (owner)

- Two URLs have the same origin if the **protocol**, **domain name** (not subdomains), **port** are the same for both URLs
  - All three must be equal origin to be considered the same

Quiz – Same Origin?

- Consider this URL:

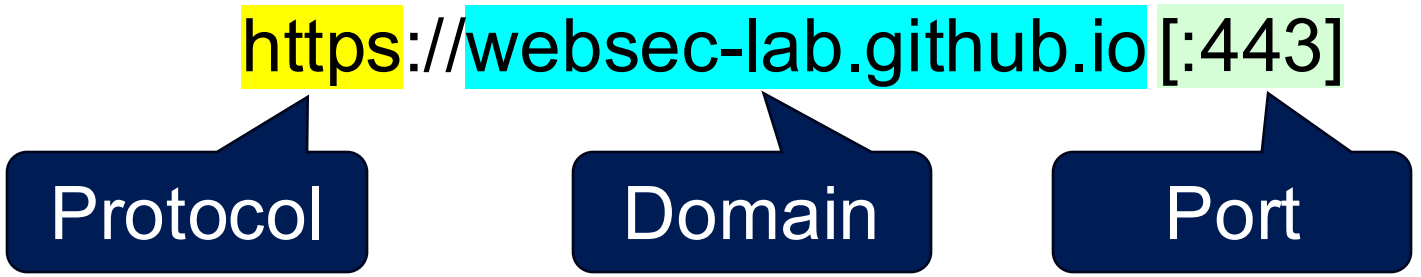
https://websec-lab.github.io

Origin = Protocol + Domain Name + Port

Idx	URL	Same Origin? ✓ (yes) **or** ✗ (No)
1	http://websec-lab.github.io	
2	https://www.websec-lab.github.io	
3	https://websec-lab.github.io:443	
4	https://websec-lab.github.io:8081	
5	https://websec-lab.github.io/cse610	

Quiz – Same Origin?

- Consider this URL:



Origin = Protocol + Domain Name + Port

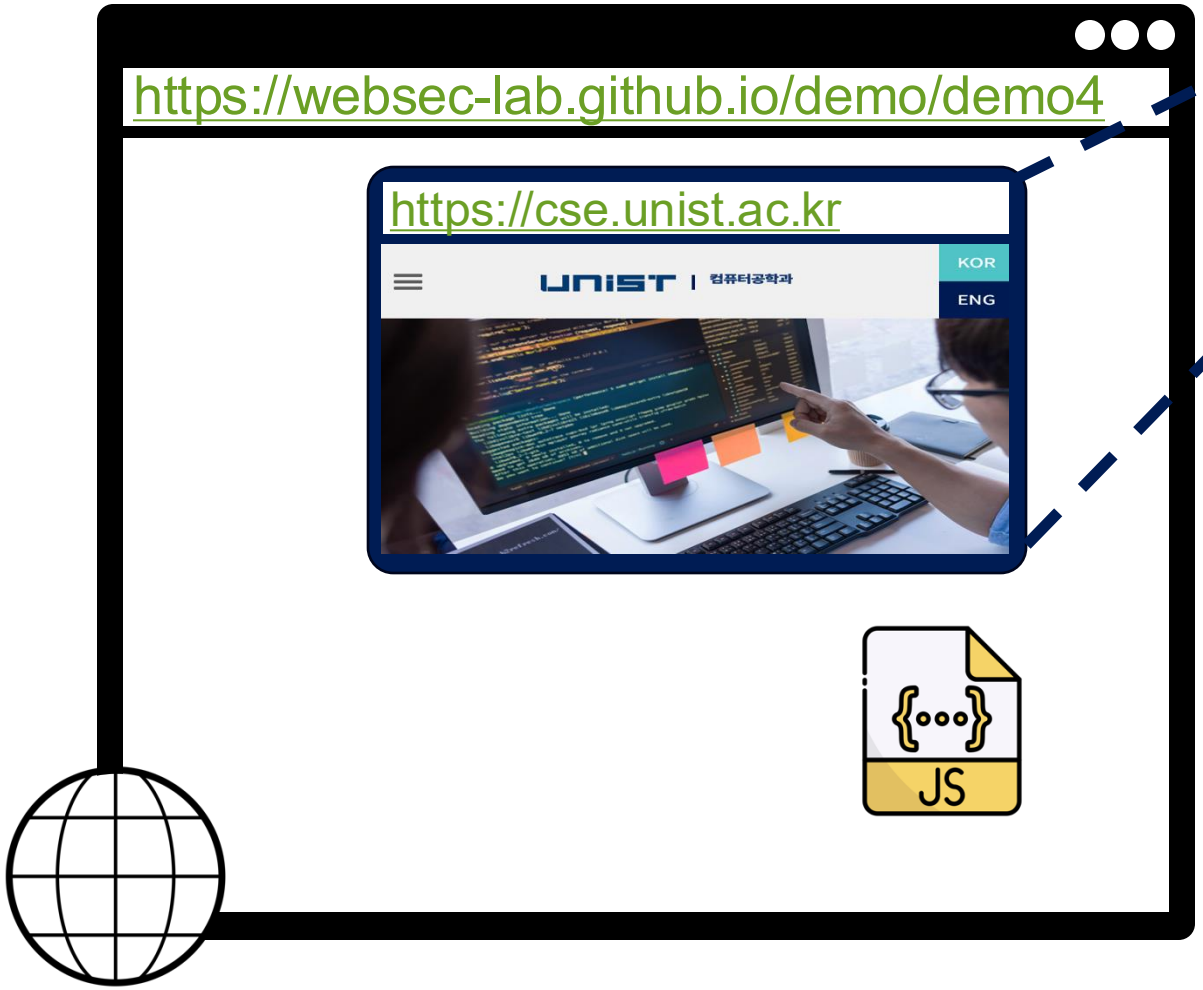
Idx	URL	Same Origin? ✓ (yes) or ✗ (No)
1	http://websec-lab.github.io[:80]	✗ (different protocol and different port)
2	https://www.websec-lab.github.io	✗ (different domain)
3	https://websec-lab.github.io:443	✓ (same protocol, domain, and port)
4	https://websec-lab.github.io:8081	✗ (different port)
5	https://websec-lab.github.io/cse610	✓ (same protocol, domain, and port)

What is an Origin?

- **Origin = Protocol + Domain Name + Port**
  - `origin = protocol://domain:port`
- Any resource has its own origin (owner)

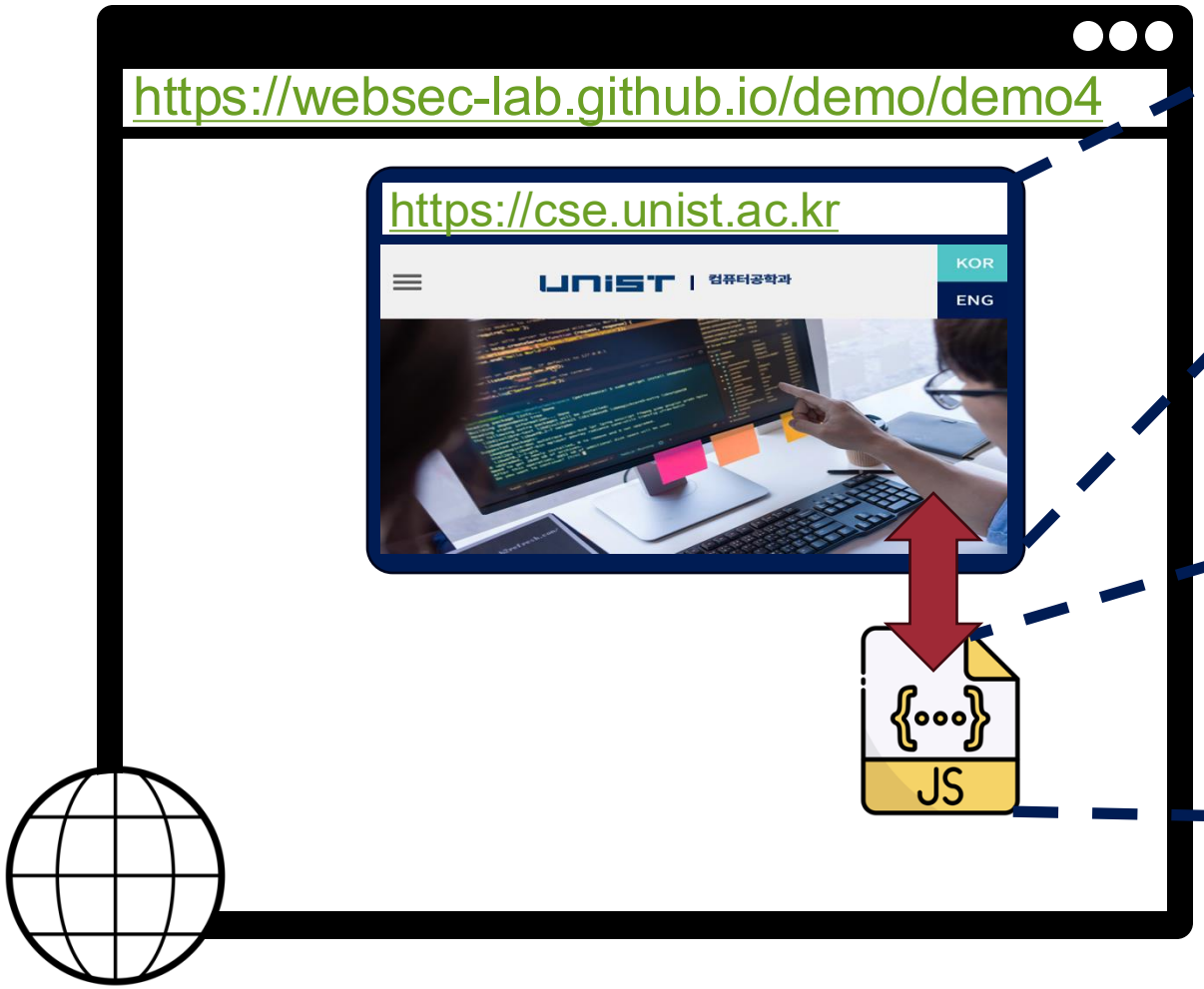
- Two URLs have the same origin if the **protocol**, **domain name** (not subdomains), **port** are the same for both URLs
  - All three must be equal origin to be considered the same

Demo: Same Origin Policy




```
<iframe id="UNIST_CSE”
        src=https://cse.unist.ac.kr/>
</iframe>
```

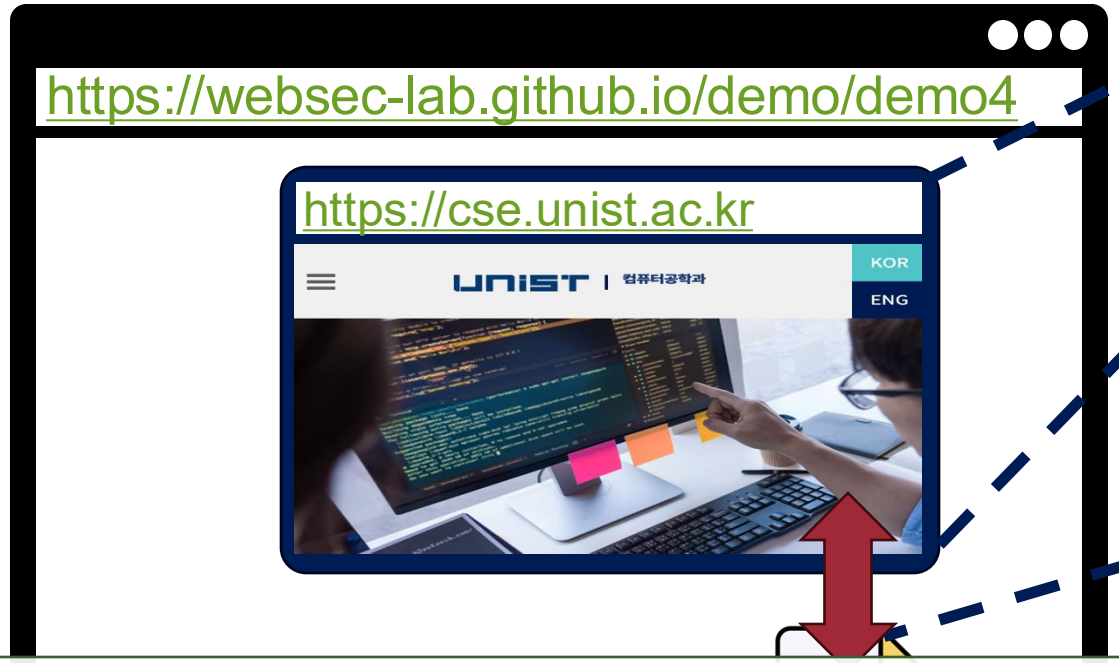
Demo: Same Origin Policy




```
<iframe id="UNIST_CSE”
        src=https://cse.unist.ac.kr/>
</iframe>
```

```
cookie =
   document.getElementById(‘UNIST_CSE’).
        contentWindow.document.cookie;
console.log(cookie)
```

Demo: Same Origin Policy



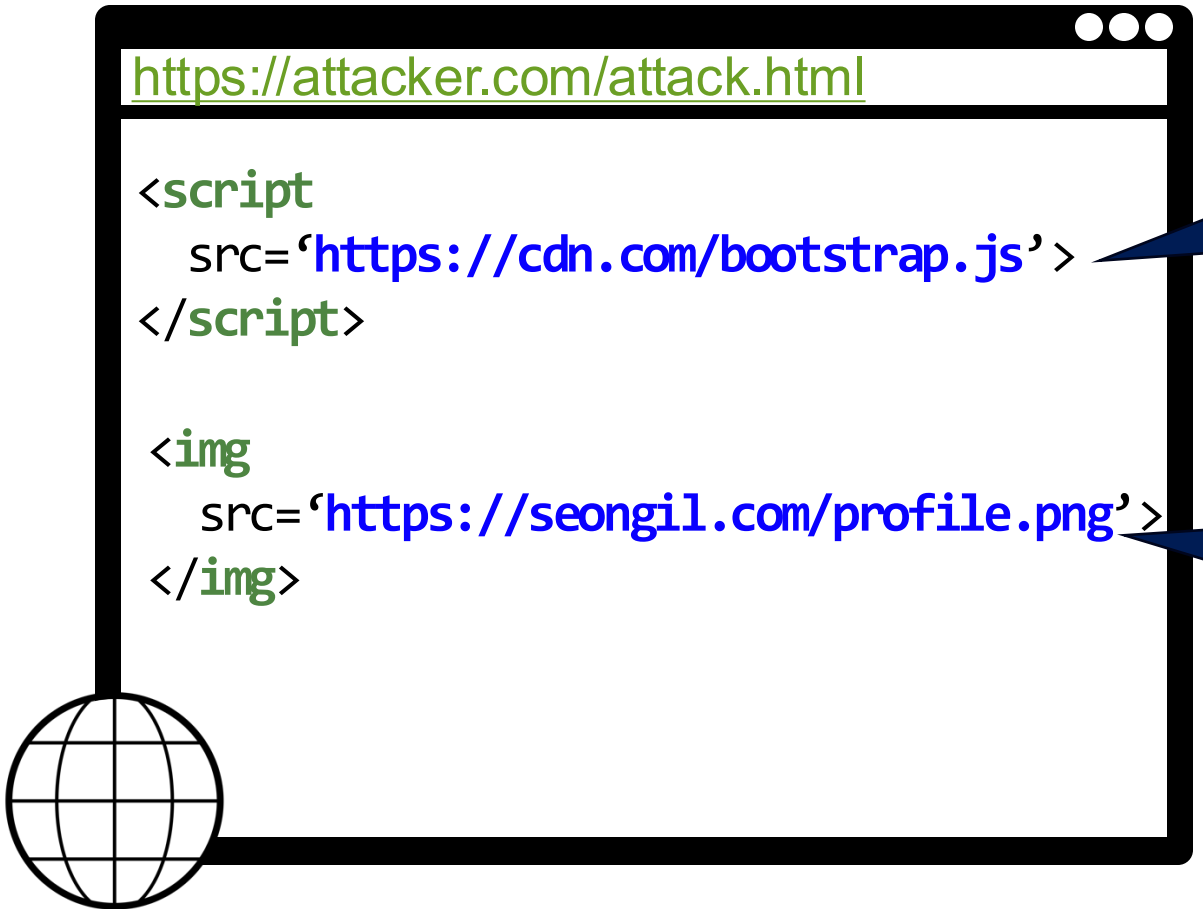

```
<iframe id="UNIST_CSE”
        src=https://cse.unist.ac.kr/>
</iframe>
```

```
cookie =
    document.getElementById(‘UNIST_CSE’).
             contentWindow.document.cookie;
console.log(cookie)
```

Uncaught DOMException: Blocked a frame with origin "https://websec-lab.github.io" from accessing a cross-origin frame

For Your Information...

- **Cross-origin loading** of page resources is generally permitted
  - E.g., the SOP allows embedding of external resources via HTML tags (e.g., <img>, <video>, <script>, ...)



The origin of the loaded script is https://attacker.com

The origin of the loaded image is https://attacker.com

Analogy

- **Operating system**
  - Primitives (Resources)
    - System calls
    - Processes
    - Disk
  - Principals: Users
    - Discretionary access control
  - Vulnerabilities
    - Buffer overflow
    - Root exploit

- **Web browser**
  - Primitives (Resources)
    - Document object model
    - Frames
    - Cookies / localStorage
  - Principals: “Origins”
    - Mandatory access control
  - Vulnerabilities
    - Cross-site scripting
    - Cross-site request forgery
    - Cache history attacks
    - …

If I need to communicate with other websites, what methods should be used?

Cross-Origin Resource Sharing (CORS)

PostMessage (PM)

Cookie Security  

How to make HTTP stateful securely?

Same Origin Policy: "High Level"

- Recap: Same Origin Policy (SOP) for DOM:

Origin A can access origin B's DOM if match on:
(protocol, domain, port)

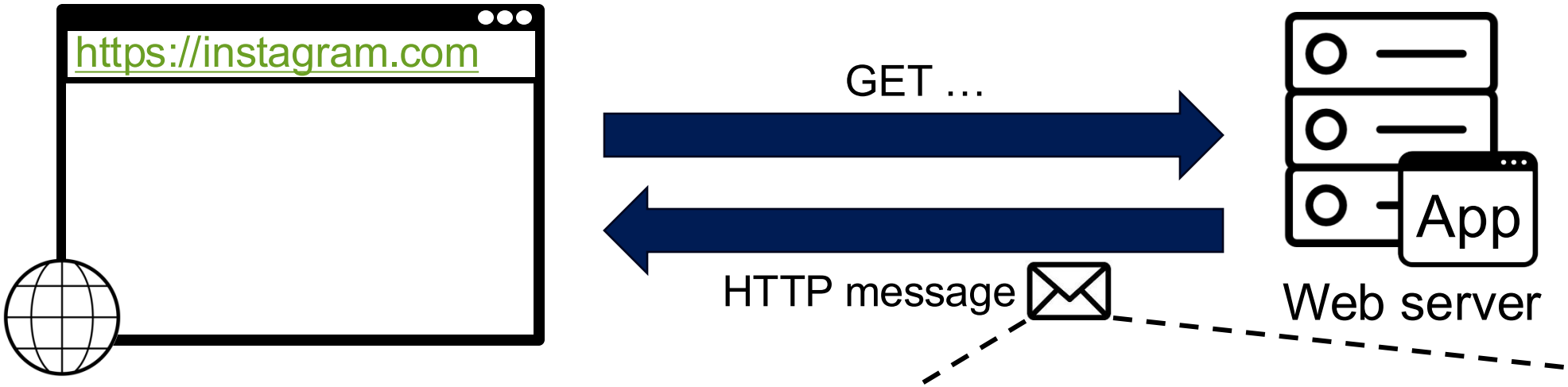
- Today: Same Origin Policy (SOP) for cookies: 

Generally speaking, based on:
([protocol], domain, *path*)

Optional

```
protocol://domain:port/path?params
```

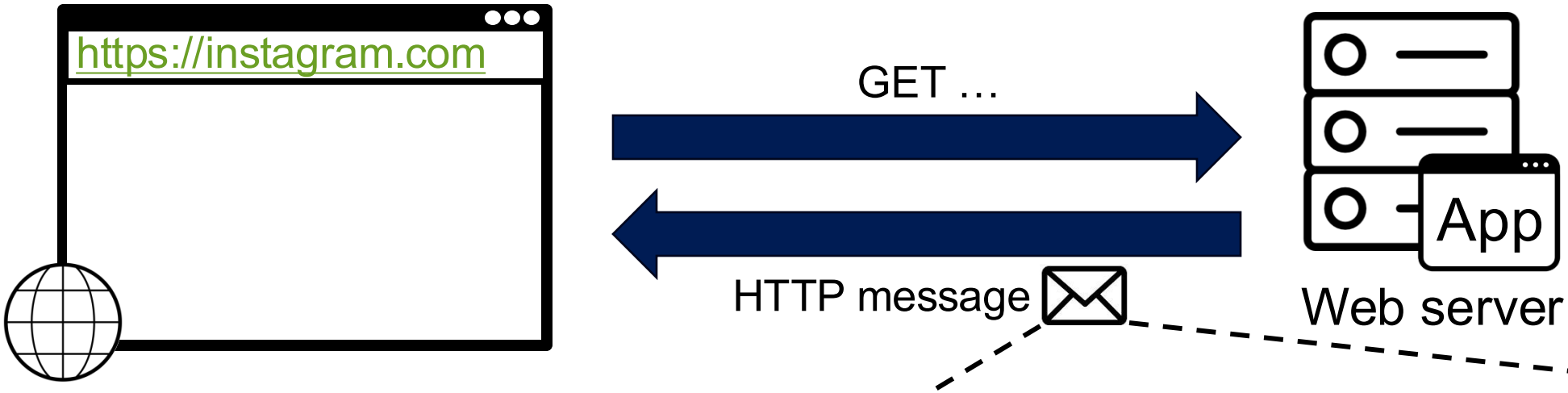
Setting Cookies by Server



HTTP Header:

```
Set-cookie: NAME = VALUE;
            Domain = (where to send);
            Path = (where to send);
            Expires = (when expires);
            Secure; (only send over SSL)
            HttpOnly; (later)
```

Deleting Cookies by Server

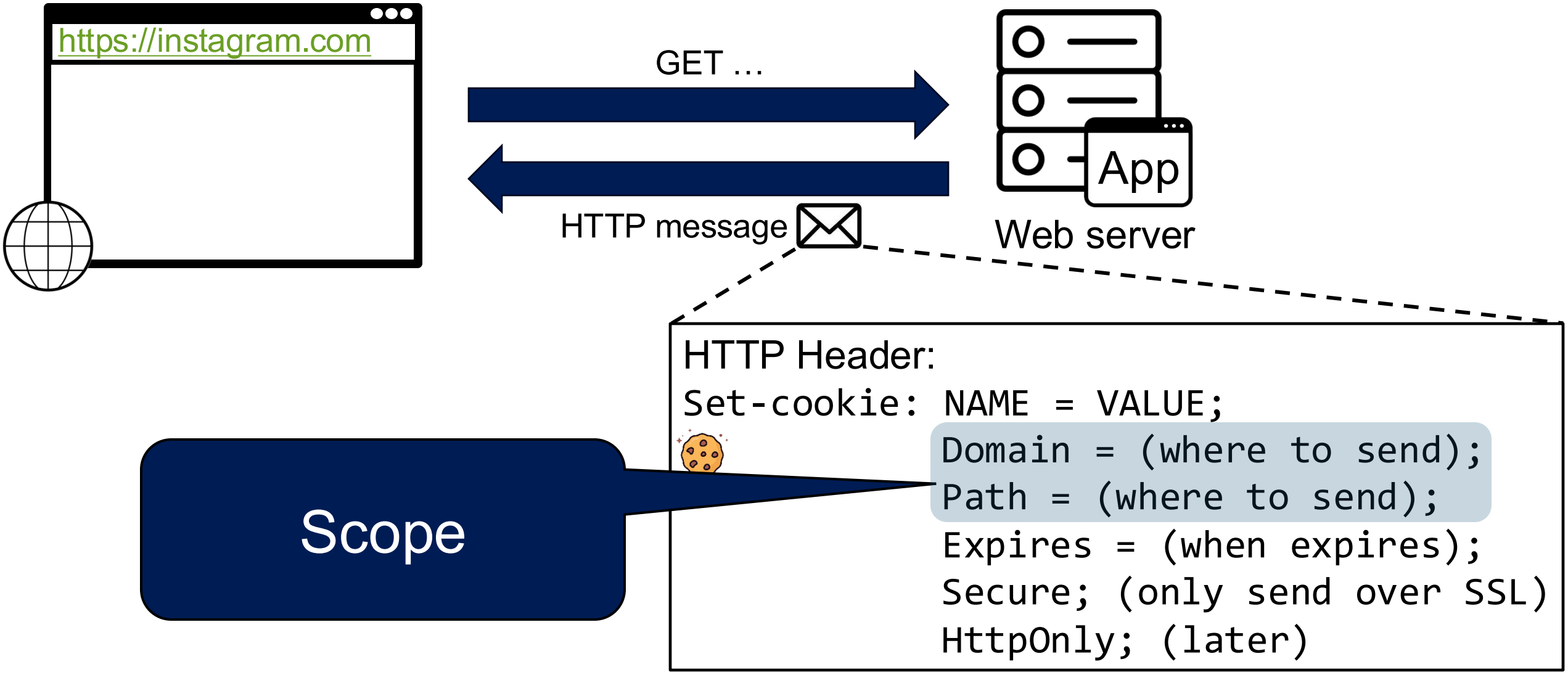


HTTP Header:

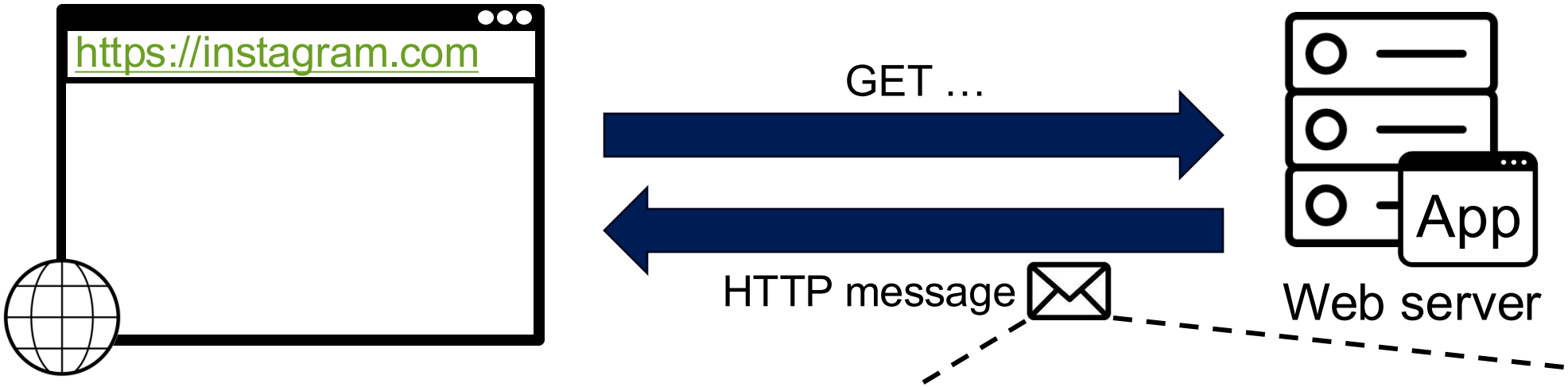
```
Set-cookie: NAME = VALUE;
            Domain = (where to send);
            Path = (where to send);
            Expires = (when expires);
            Secure; (only send over SSL)
            HttpOnly; (later)
```

Delete cookie by setting "Expires" to date in past

Cookie Scope



Cookie Scope

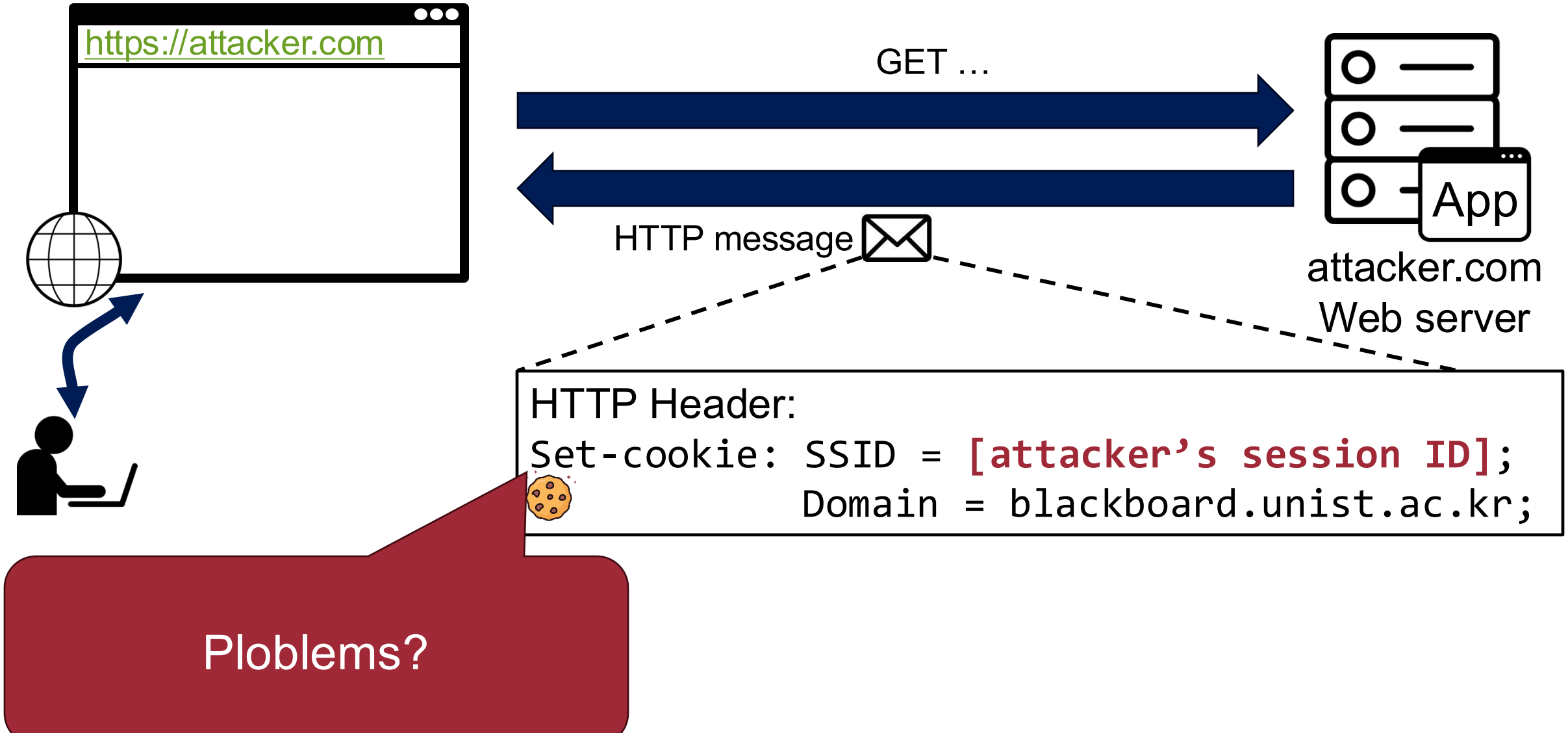


HTTP Header:

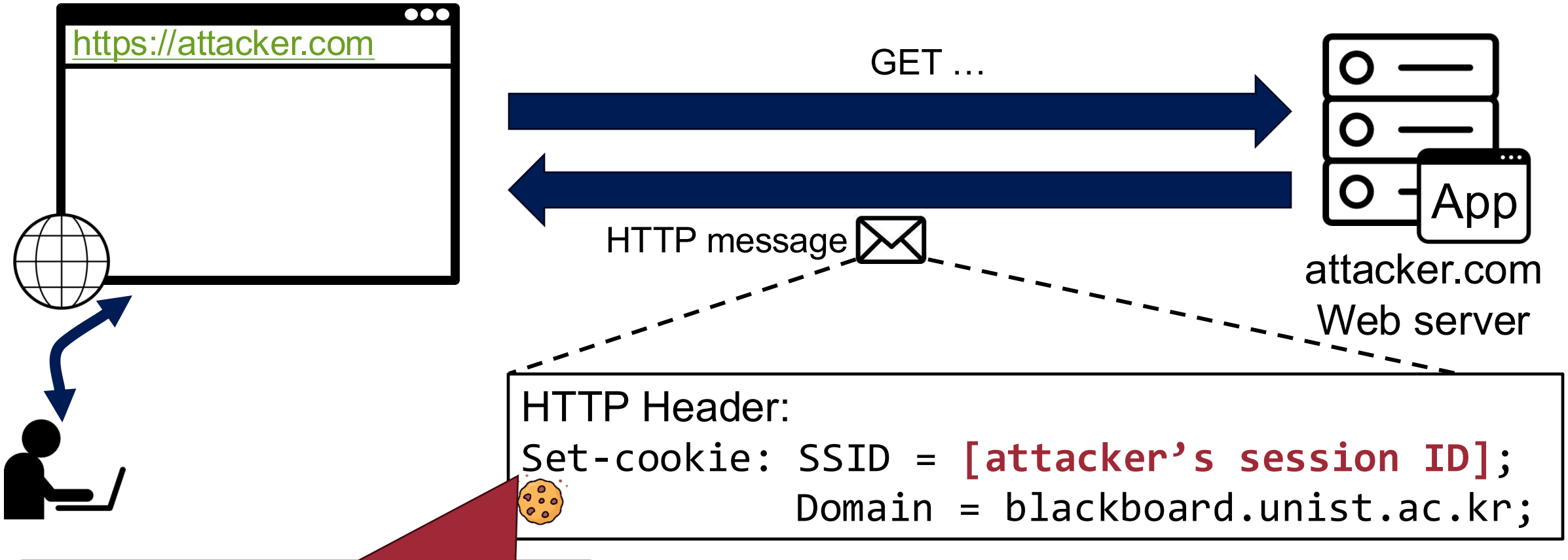
```
Set-cookie: NAME = VALUE;
            Domain = (where to send);
            Path = (where to send);
            Expires = (when expires);
            Secure; (only send over SSL)
            HttpOnly; (later)
```

Default scope is domain and path of setting URL

Motivating Example of the Cookie Write SOP



Motivating Example of the Cookie Write SOP



The attacker can write arbitrary values to the arbitrary cookies

Scope Setting Rules (Write SOP)

- **Domain**: any domain-suffix of URL-hostname, except Top Level Domain (TLD)

- **Path**: can be set to anything

Scope Setting Rules (Write SOP)

- **Domain**: any domain-suffix of URL-hostname, except Top Level Domain (TLD)

  Question: which cookies can be set by login.site.com?

Idx	Cookie's domain	Write Allowed?
1	login.site.com	
2	.site.com	
3	.com	
4	alice.site.com	
5	othersite.com	

- **Path**: can be set to anything

Scope Setting Rules (Write SOP)

- **Domain**: any domain-suffix of URL-hostname, except Top Level Domain (TLD)

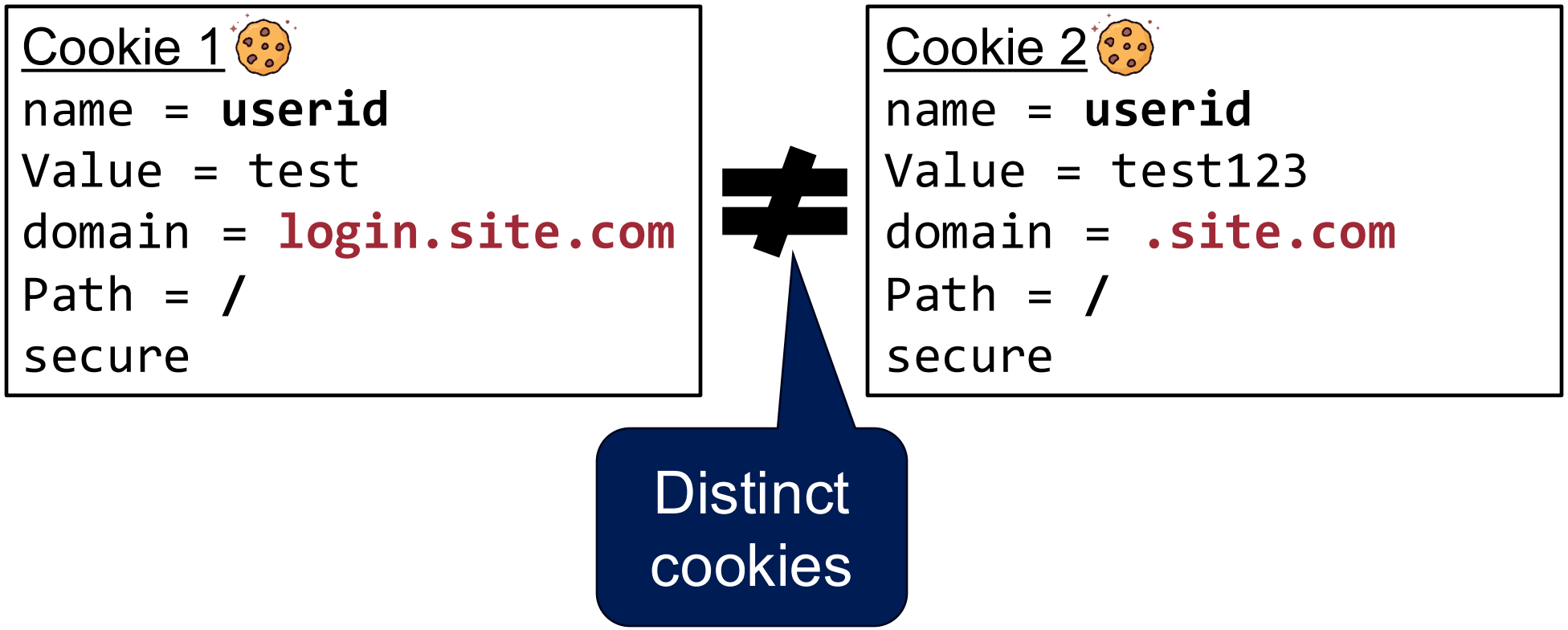
  Question: which cookies can be set by login.site.com?

Idx	Cookie's domain	Write Allowed?
1	login.site.com	✓
2	.site.com	✓
3	.com	*X* (TLD)
4	alice.site.com	*X* (Not the domain suffix of the hostname)
5	othersite.com	*X* (Not the domain suffix of the hostname)

login.site.com can set cookies for all of .site.com

- **Path**: can be set to anything

Cookies are Identified by (name, domain, path)



Browser's Cookie Jar

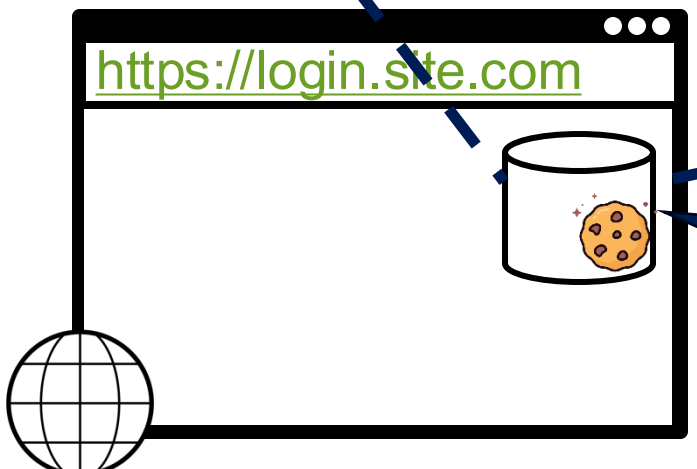
Cookie 1

name = **userid**
Value = test
domain = **login.site.com**
Path = /
secure

≠

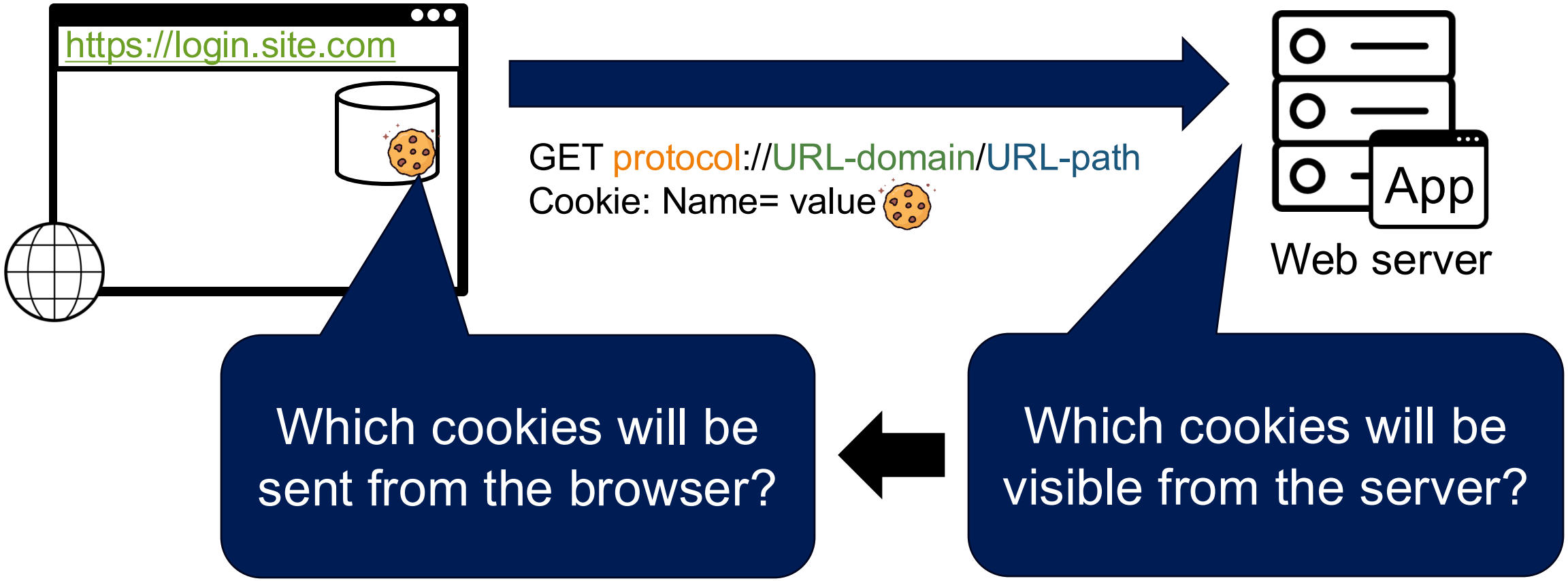
Cookie 2

name = **userid**
Value = test123
domain = **.site.com**
Path = /
secure

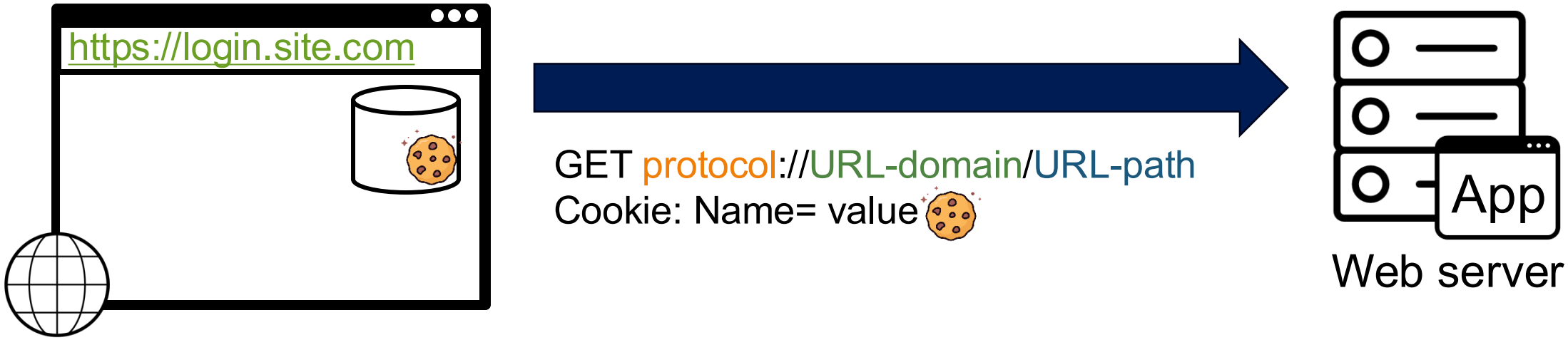


Both cookies are stored in browser's cookie jar;
Both are in scope of login.site.com

Reading Cookies on Server (Read SOP)



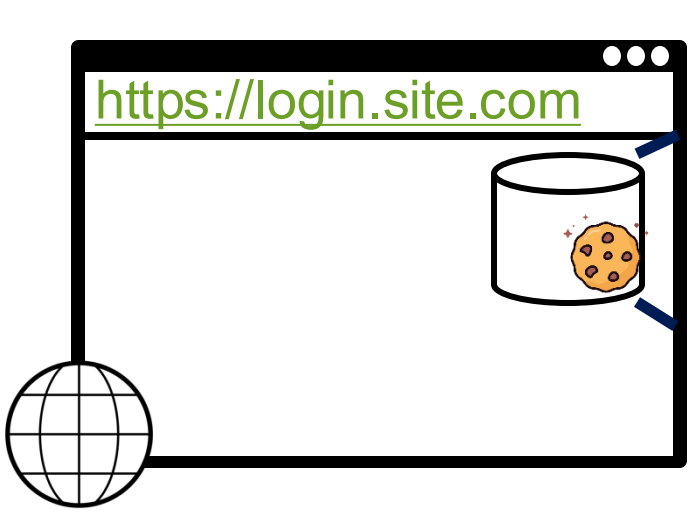
Reading Cookies on Server (Read SOP)



- **Browser sends all cookies in URL scope:**
  - Cookie `domain` is domain-suffix of URL-domain, and
  - Cookie `path` is prefix of URL-path, and
  - [protocol=HTTPS if cookie is "`secure`"]

- **Goal**: server only sees cookies in its scope

Quiz: Read SOP



Cookie 1

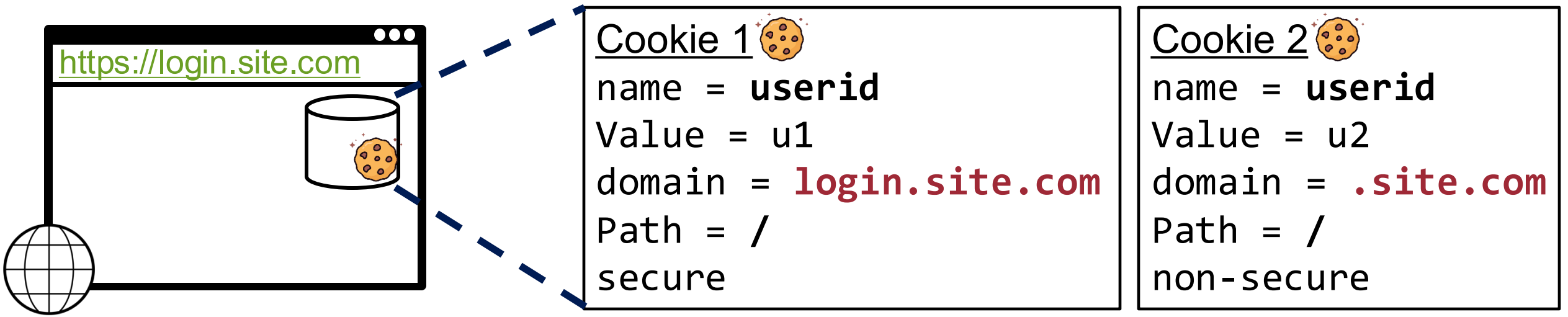
```
name = userid
Value = u1
domain = login.site.com
Path = /
secure
```

Cookie 2

```
name = userid
Value = u2
domain = .site.com
Path = /
non-secure
```

Idx	Request Domains	Sent Cookies
1	http://checkout.site.com/	
2	http://login.site.com/	
3	https://login.site.com/	

Quiz: Read SOP



Idx	Request Domains	Sent Cookies
1	http://checkout.site.com/	cookie: userid=u2
2	http://login.site.com/	cookie: userid=u2
3	https://login.site.com/	**cookie: userid=u1; userid=u2** (arbitrary order)

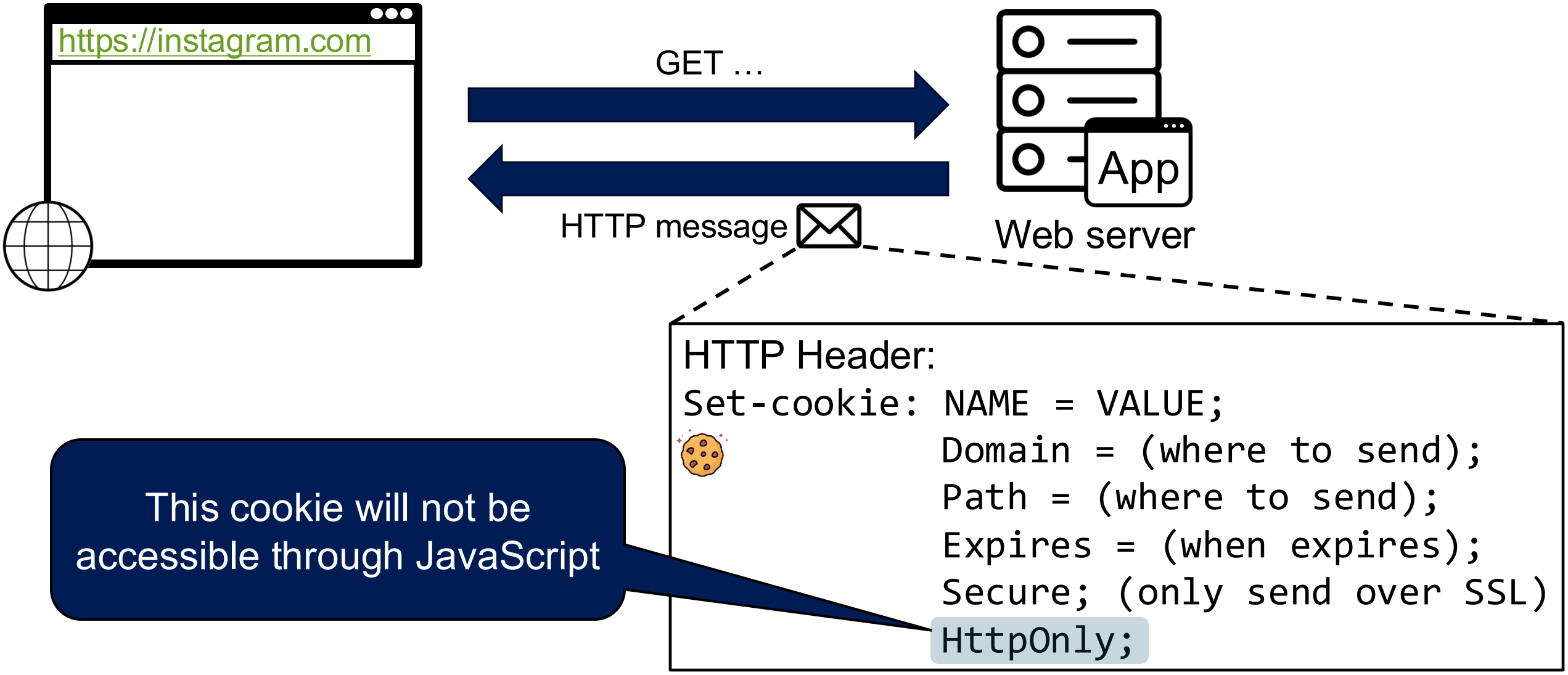
Client-side Read/Write: `document.cookie`

- Setting a cookie in JavaScript:
  - E.g., `document.cookie = “name=value; expires=…;”`
  - E.g., `document.cookie = “ssid=AB31FBS5; domains=.unist.ac.kr”`
- Reading a cookie: `alert(document.cookie)`
  - Prints string containing all cookies available for document
- Deleting a cookie:
  - Document.cookie = “name=; expires=Thu, 01-Jan-70”

Client-side Read/Write: `document.cookie`

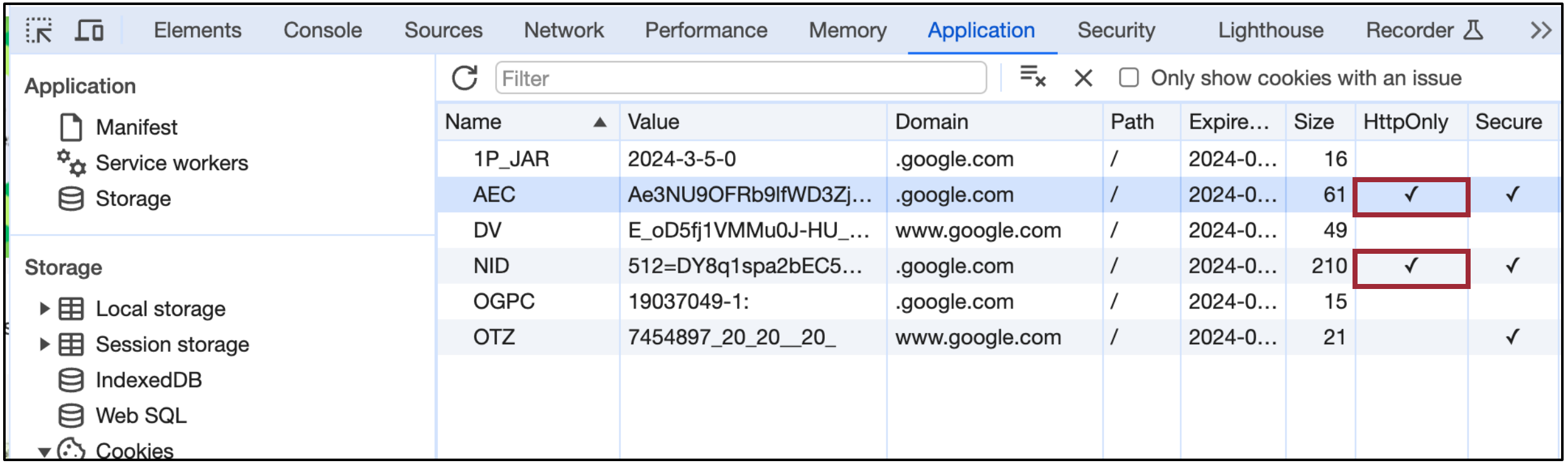
We can access all cookies for current document by `alert(document.cookie)`

HttpOnly Cookies



HttpOnly Cookies

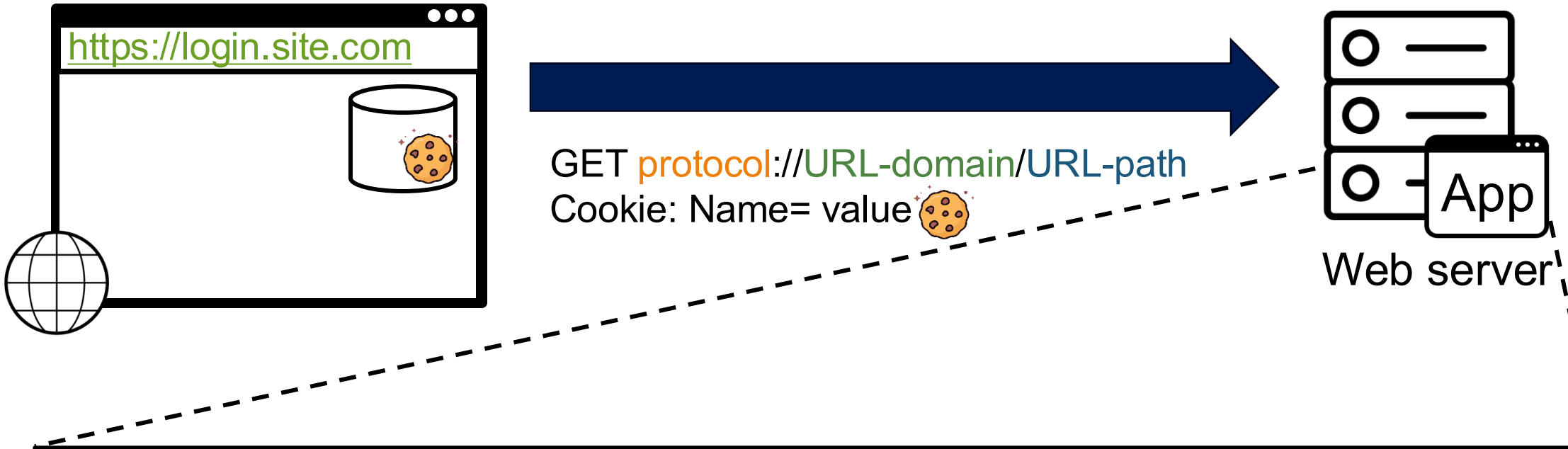
- Cookie sent over HTTP(s), but not accessible to scripts
  - Cannot be read via `document.cookie`
  - Helps prevent cookie theft attacks



HttpOnly Cookie Demo

Cookie Protocol Problems

Cookie Protocol Problems



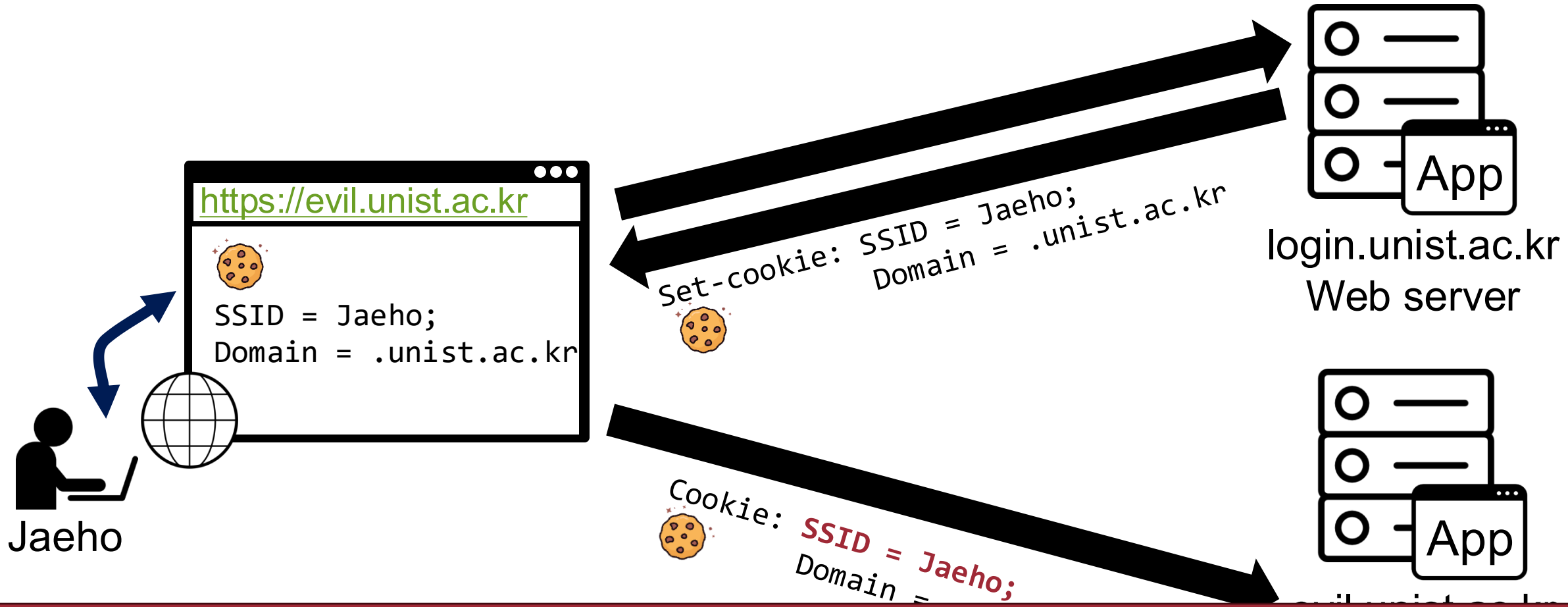
- Server is blind:
  - Does not see cookie attributes (e.g., `secure`, `HttpOnly` attributes)
  - Does not see which domain set the cookie

  - **Server only sees**: cookie: NAME=VALUE

Example 1: Cookie Exfiltration

Example 1: Cookie Exfiltration

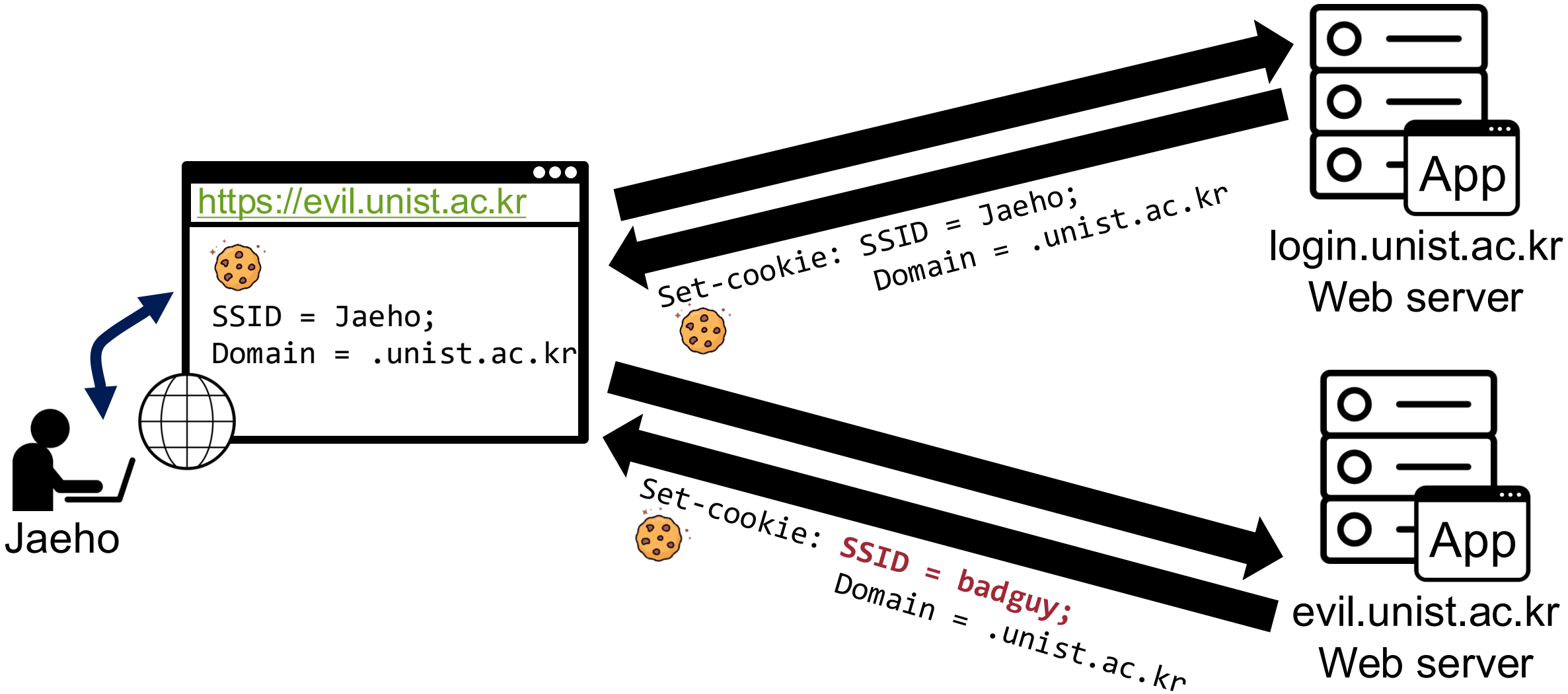
Example 1: Cookie Exfiltration



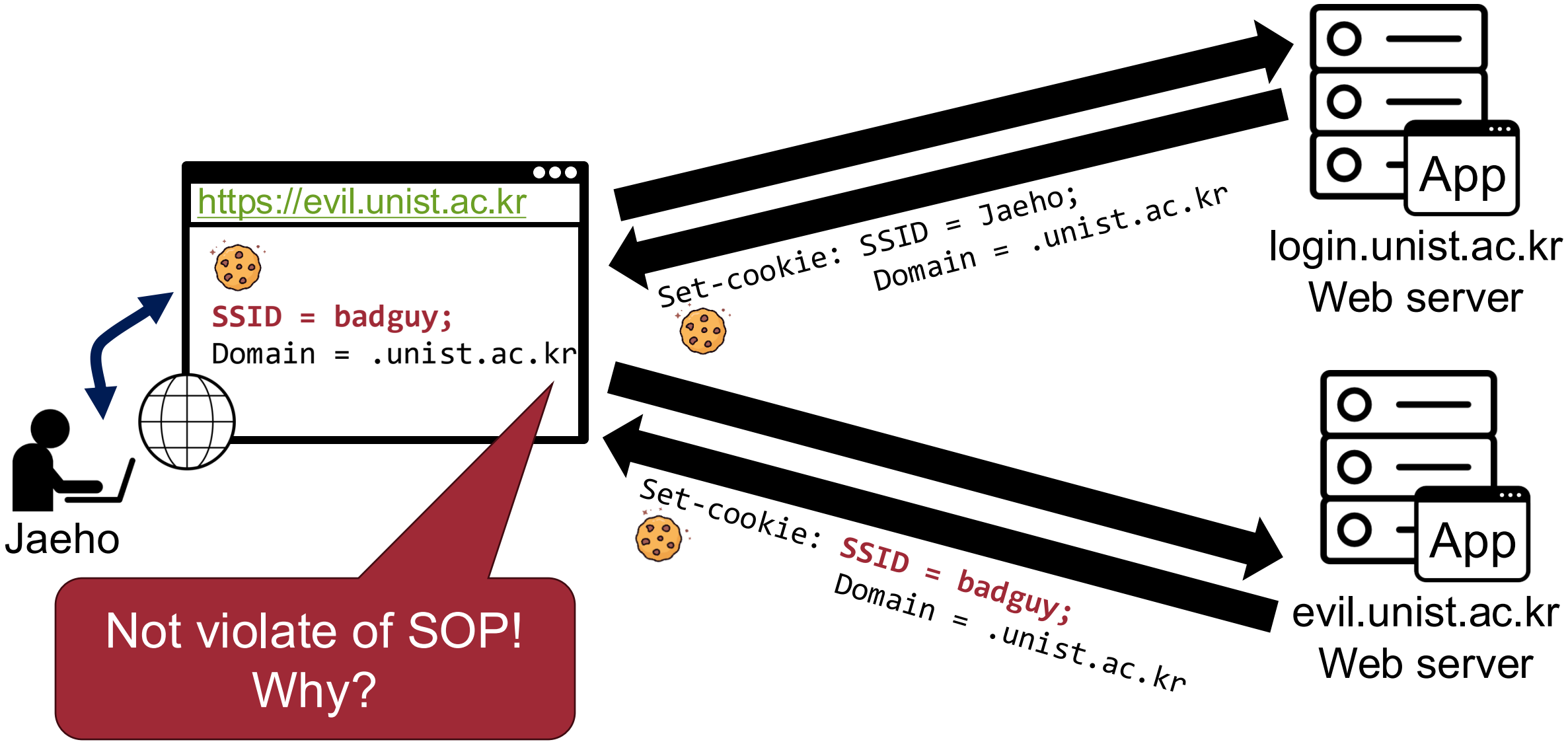
An attacker obtained Jaeho's session cookie

Example 2: Login as Attacker

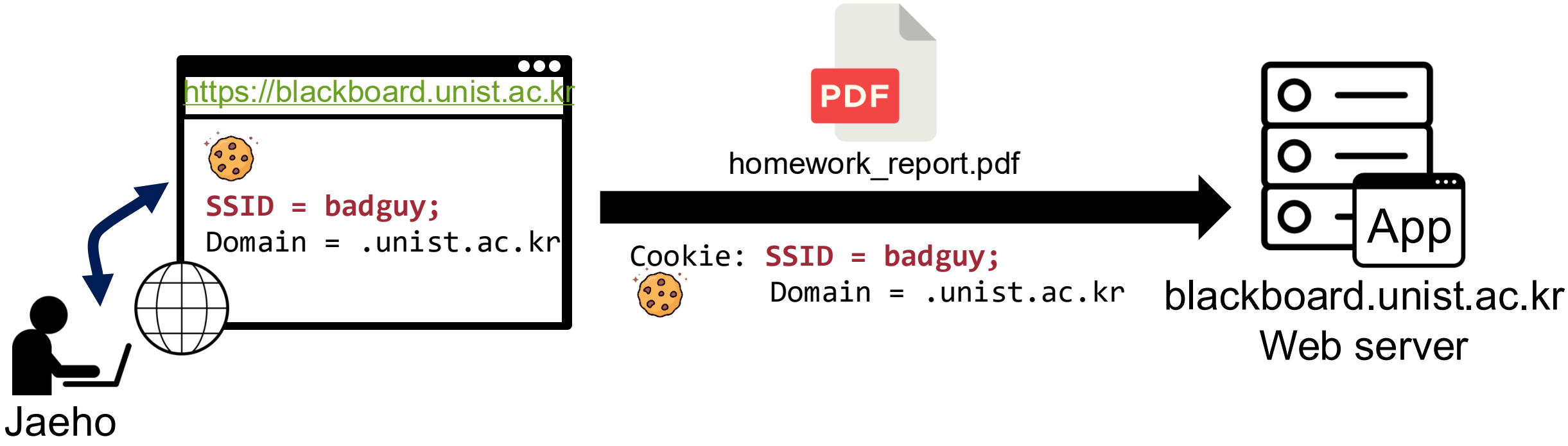
Example 2: Login as Attacker



Example 2: Login as Attacker



Example 2: Login as Attacker



Jaeho's homework will be submitted under the name "badguy"

Example 2: Login as Attacker

1. Jaeho logs in at **login.unist.ac.kr**
   login.site.com sets session-id cookie for **.unist.ac.kr**

2. Jaeho visits **evil.unist.ac.kr**
   overwrites **.unist.ac.kr** session-id cookie with session-id of user “badguy”

3. Jaeho visits **blackboard.unist.ac.kr** to submit homework
   Jaeho’s homework will be submitted under the name “badguy”

The web server **blackboard.unist.ac.kr** expects session-id from **login.unist.ac.kr**; But it cannot know that session-id cookie was overwritten

Example 3: "secure" Cookies are not Secure

1. Alice logs in at **https**://accounts.google.com
   - `Set-Cookie: SSID=Au7_ESAgDpKY5TGnf; Domain=.google.com; Path=/; Expires=Wed, 09-Mar-2026; Secure; HttpOnly`

2. Alice visits **http**://www.google.com (cleartext)
   - Network attacker can inject into response **`Set-Cookie: SSID=badguy; secure`** and overwrite secure cookie

Network attacker can re-write HTTPS cookies
⇒ HTTPS cookie value cannot be trusted

Recap: Web Threat Models

- **Network attacker**: resides somewhere in the communication link between client and server
  - Passive: evasdropping
  - Active: modification of messages, replay…

- **Remote attacker:** can connect to remote system via the network
  - Mostly targets the server

- **Web attacker**: controls attacker.com
  - Can obtain SSL/TLS certificates for attacker.com
  - Users can visit attacker.com

Example 3: Path Separation is not a Security Measure

- Cookie SOP: path separation

  x.com/A does not see cookies of **x.com/B**

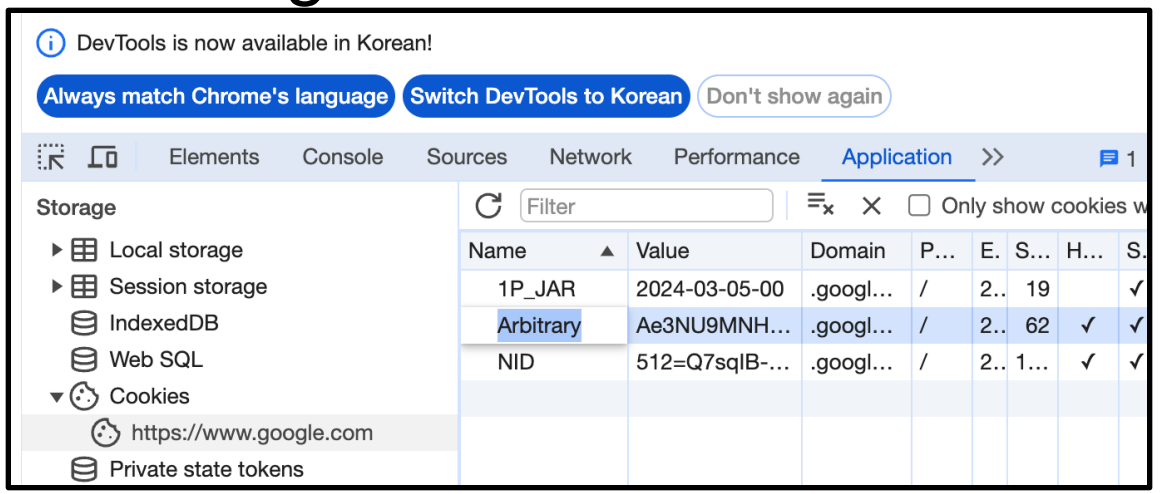
- Not a security measure:
  - DOM SOP: **x.com/A** has access to DOM of **x.com/B**

```
<iframe src="x.com/B"></iframe>
alert(frames[0].document.cookie);
```

- Path separation is done for efficiency not security:
  - x.com/A is only sent the cookies it needs

Cookies have No Integrity

- End users and attackers can change and delete cookie values



- Simple example: shopping cart software

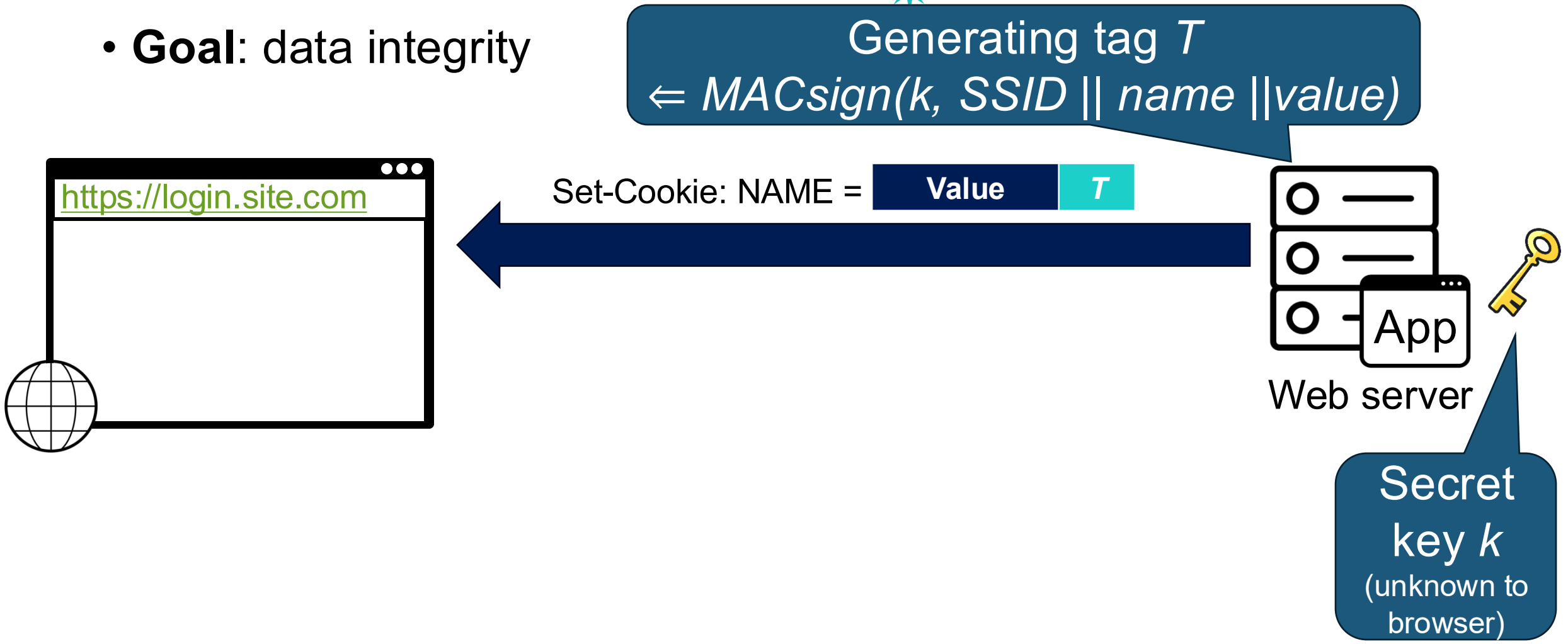
```
Set-cookie: shopping-cart-total = 150 ($)
```

User edits cookie contents (cookie poisoning):

```
Cookie: shopping-cart-total = 15 ($)
```

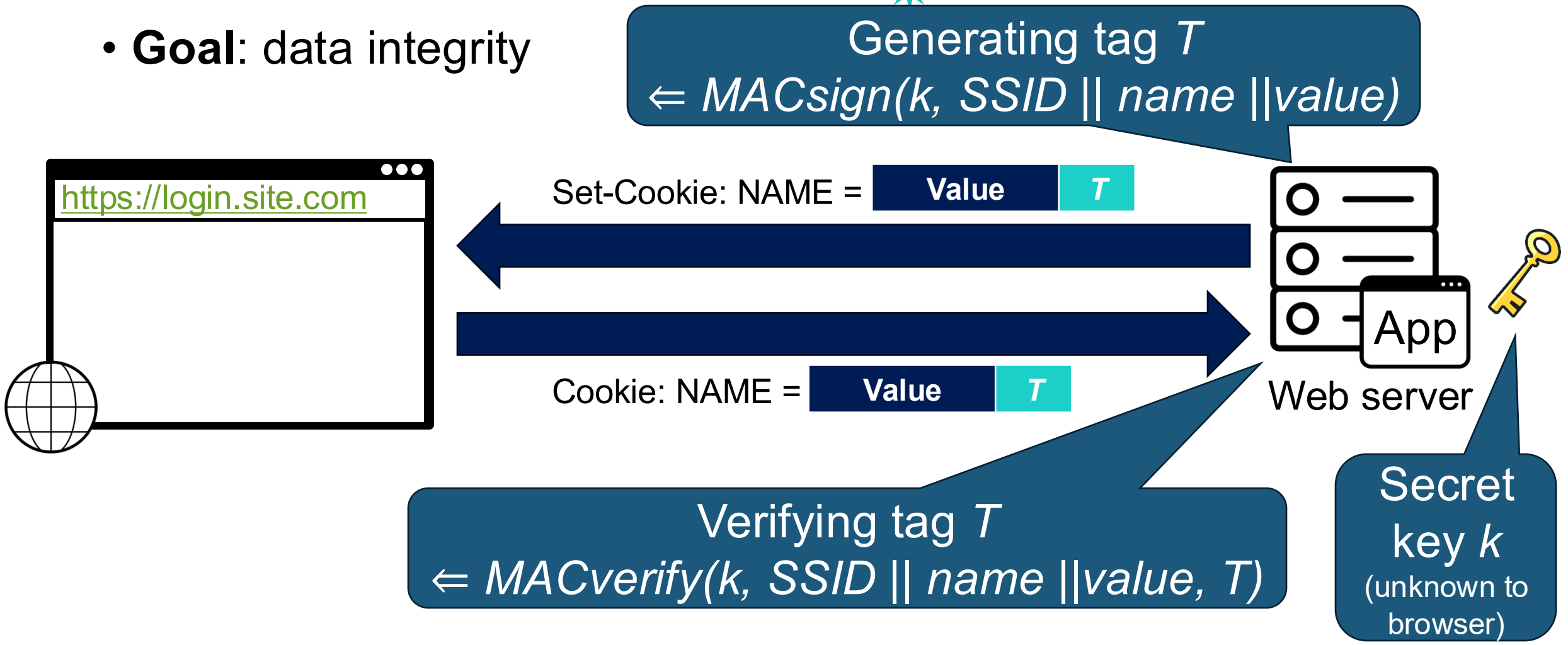
Solution: Cryptographic Checksums

- **Goal**: data integrity



Solution: Cryptographic Checksums

- **Goal**: data integrity



Binding to session-id (SSID) makes it harder to replay old cookies

Example: ASP.NET

- `System.Web.Configuration.MachineKey`
  - Secret web server key intended for cookie protection

- Creating an encrypted cookie with integrity:

```
HttpCookie cookie = new HttpCookie(name,val);
HttpCookie encodedCookie = HttpSecureCookie.Encode(cookie);
```

- Decrypting and validating an encrypted cookie:

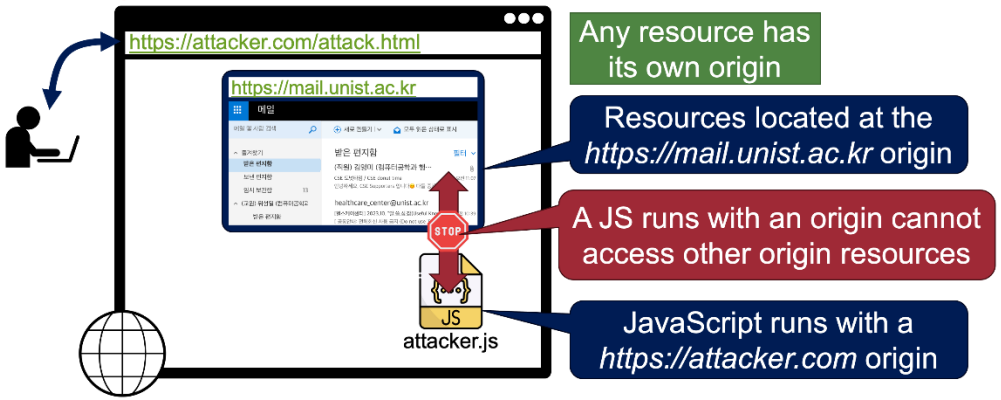
```
HttpSecureCookie.Decode(cookie);
```

Conclusion

Same Origin Policy (SOP)

23

- Restricts scripts on one origin from accessing data from another origin



Same Origin Policy: "High Level"

38

- Recap: Same Origin Policy (SOP) for DOM:

Origin A can access origin B's DOM if match on:
(protocol, domain, port)

- Today: Same Origin Policy (SOP) for cookies:

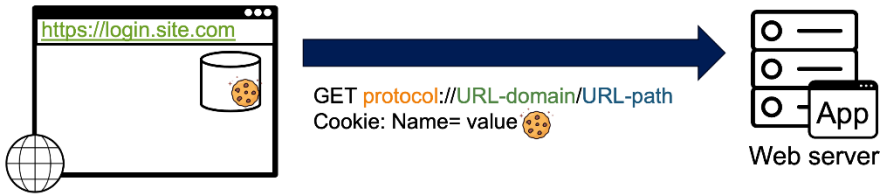
Generally speaking, based on:
([protocol], domain, *path*)

Optional

`protocol://domain:port/path?params`

Reading Cookies on Server (Read SOP)

51



- **Browser sends all cookies in URL scope:**
  - Cookie `domain` is domain-suffix of URL-domain, and
  - Cookie `path` is prefix of URL-path, and
  - [protocol=HTTPS if cookie is "`secure`"]

- **Goal**: server only sees cookies in its scope

Scope Setting Rules (Write SOP)

45

- **Domain**: any domain-suffix of URL-hostname, except Top Level Domain (TLD)

- **Path**: can be set to anything

Question?