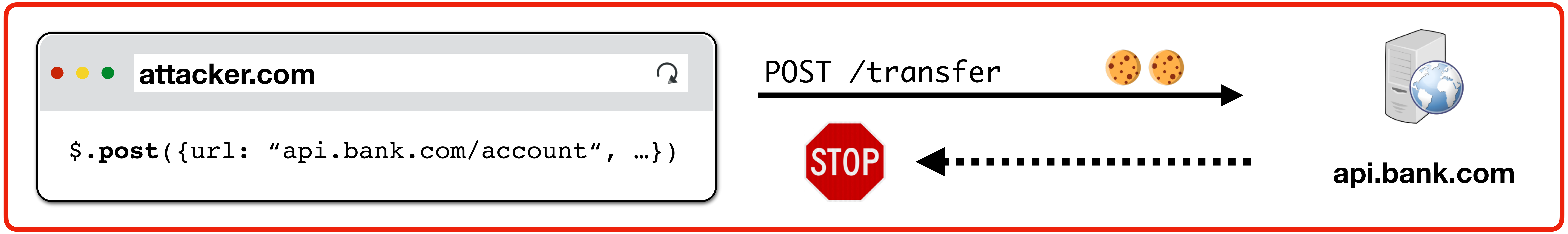
Building Secure Web Apps

CS155 Computer and Network Security

Stanford University

Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF)



Cross-site request forgery (CSRF) attacks are a type of web exploit where a website transmits unauthorized commands as a user that the server trusts

In a CSRF attack, a user is tricked into submitting an unintended (often unrealized) web request to a website — generally takes advantage of session cookies

<u>You need to actively build defenses into web apps to protect against CSRF attacks</u>

Options for Preventing CSRF Attacks

Do not trust cookies to indicate whether an authorized application submitted request since they're included in *every* (in-scope) request

We need another mechanism that allows us to ensure that a request is authentic (coming from a trusted page)

Three commonly used techniques to validate intent:

- Referer Header Validation
- Secret Validation Token
- Custom HTTP Header (forces CORS Pre-Flight Permissions Check)

Or, simply, don't send cookies to other domains:

- sameSite Cookies

Options for Preventing CSRF Attacks

Do not trust cookies to indicate whether an authorized application submitted request since they're included in *every* (in-scope) request

We need another mechanism that allows us to ensure that a request is authentic (coming from a trusted page)

~~Three~~ Two commonly used techniques to validate intent:

~~- Referer Header Valid~~ Form Submissions

- Secret Validation Token

Javascript Requests

- Custom HTTP Header (forces CORS Pre-Flight Permissions Check)

Or, simply, don't send cookies to other domains:

- sameSite Cookies

Options for Preventing CSRF Attacks

Do not trust cookies to indicate whether an authorized application submitted request since they're included in *every* (in-scope) request

W

a

T

What about GET Requests?

NEVER Change Application State based on a GET request

Or, simply, don't send cookies to other domains:

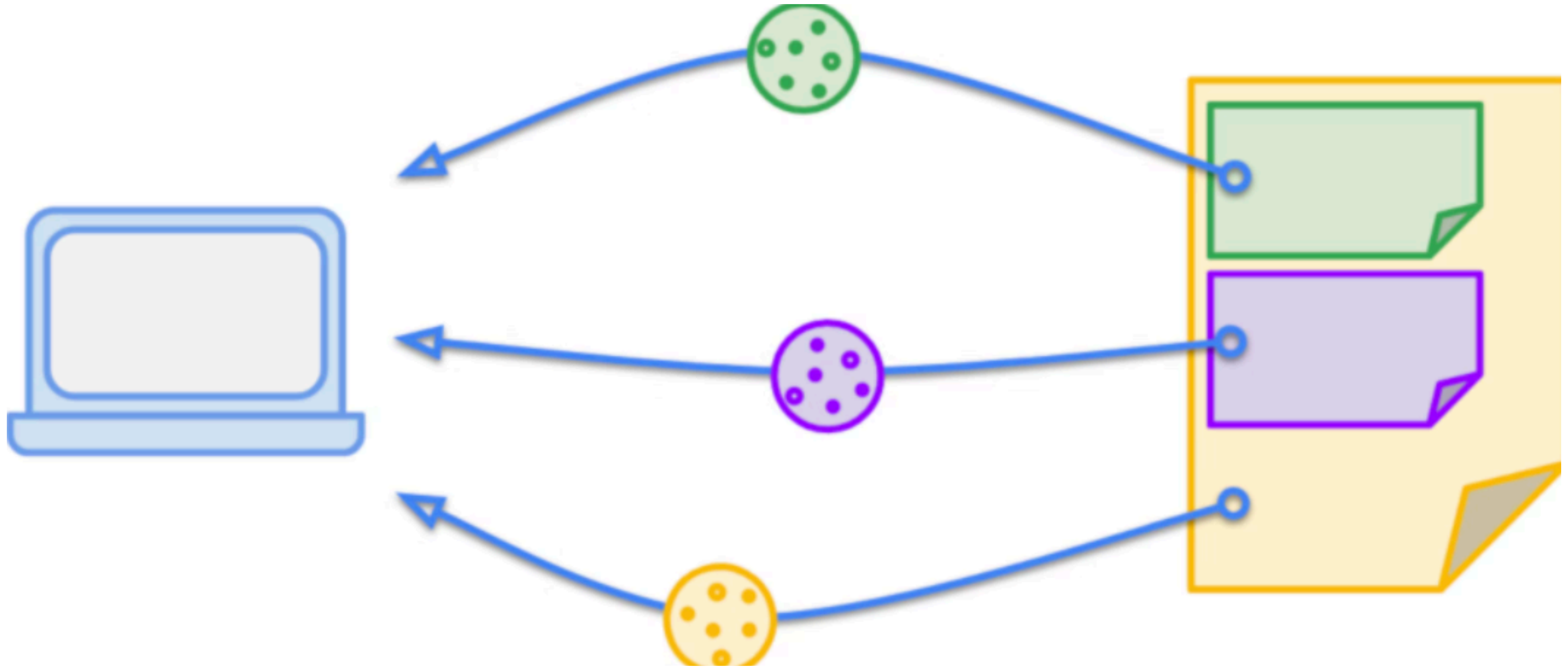
- sameSite Cookies

sameSite Cookies

Cookies that match the domain of the current site, i.e. what's *currently* displayed in the browser's address bar, are referred to as **first-party cookies**

Cookies from domains other than the current site are **third-party cookies**

Cookies marked as **sameSite are only sent** if first party



Will not be sent for image, form post if URL bar != origin of resource

Two Modes

sameSite cookie setting can be in two modes:

Strict Mode (SameSite=Strict): The cookie will only be sent if the site for the cookie matches the site currently shown in the browser's URL bar.

Problem: If you're on **Site A**, click on a link to **Site B**, then **Site B** won't receive cookie because when you clicked on the link, URL bar said **Site A** (or, if you simply typed the site into the URL bar

Lax Mode (SameSite=Lax): Allows cookie to be sent with these top-level navigations.

A Properly Secured Cookie

1. Don't set domain, unless you need to (increases scope)

2. Add Necessary Security Restrictions

Only Allowed Over HTTPS

```
Set-Cookie: key=value; Secure; HttpOnly;
SameSite=Lax;
```

Don't Allow Javascript Access through DOM

Prevent CSRF Attacks

Cross Site Scripting
(XSS)

Command Injection

Cross Site Scripting: Attack occurs when application takes untrusted data and sends it to a web browser without proper validation or sanitization.

Command/SQL Injection	Cross Site Scripting (XSS)
attacker's malicious code is executed on app's <u>server</u>	attacker's malicious code is executed on victim's <u>browser</u>

Both due to mixing untrusted user content and code to be executed

Content Security Policy (CSP)

You're always safer using a whitelist- rather than blacklist-based approach

`Content-Security-Policy` is an HTTP header that servers can send that declares which dynamic resources (e.g., Javascript) are allowed to execute

Good News: CSP eliminates XSS attacks by whitelisting the origins that are trusted sources of scripts and other resources and preventing all others

Bad News: CSP headers are complicated and folks frequently get the implementation incorrect.

Example CSP — Javascript

Policies are defined as a set of directives for where different types of resources can be fetched. For example:

```
Content-Security-Policy: script-src 'self'
```

→ Javascript can only be loaded from the same domain as the page

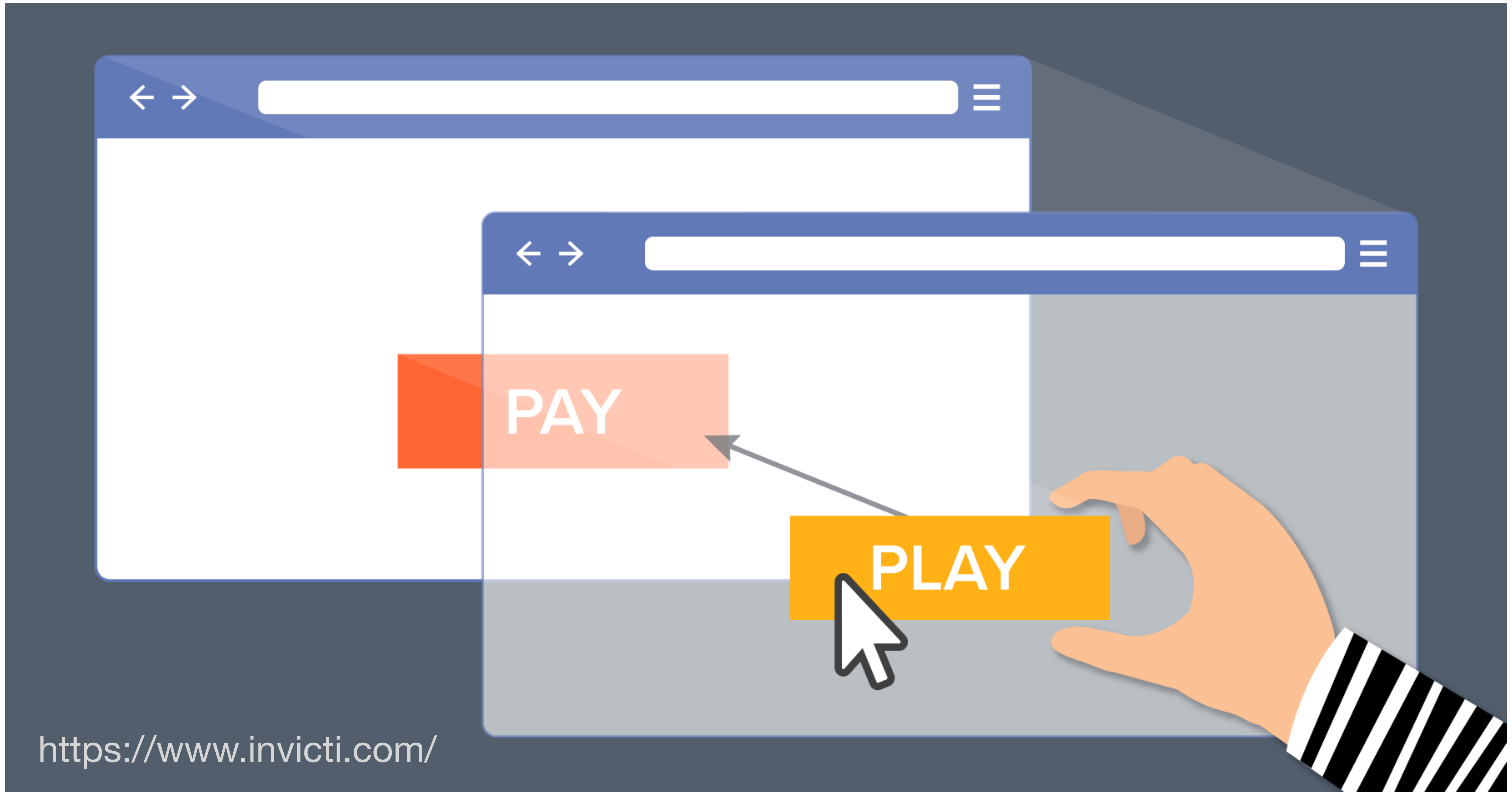
→ No Javascript from any other origins will be executed

→ no inline **`<script></script>`** will be executed

Clickjacking Attacks

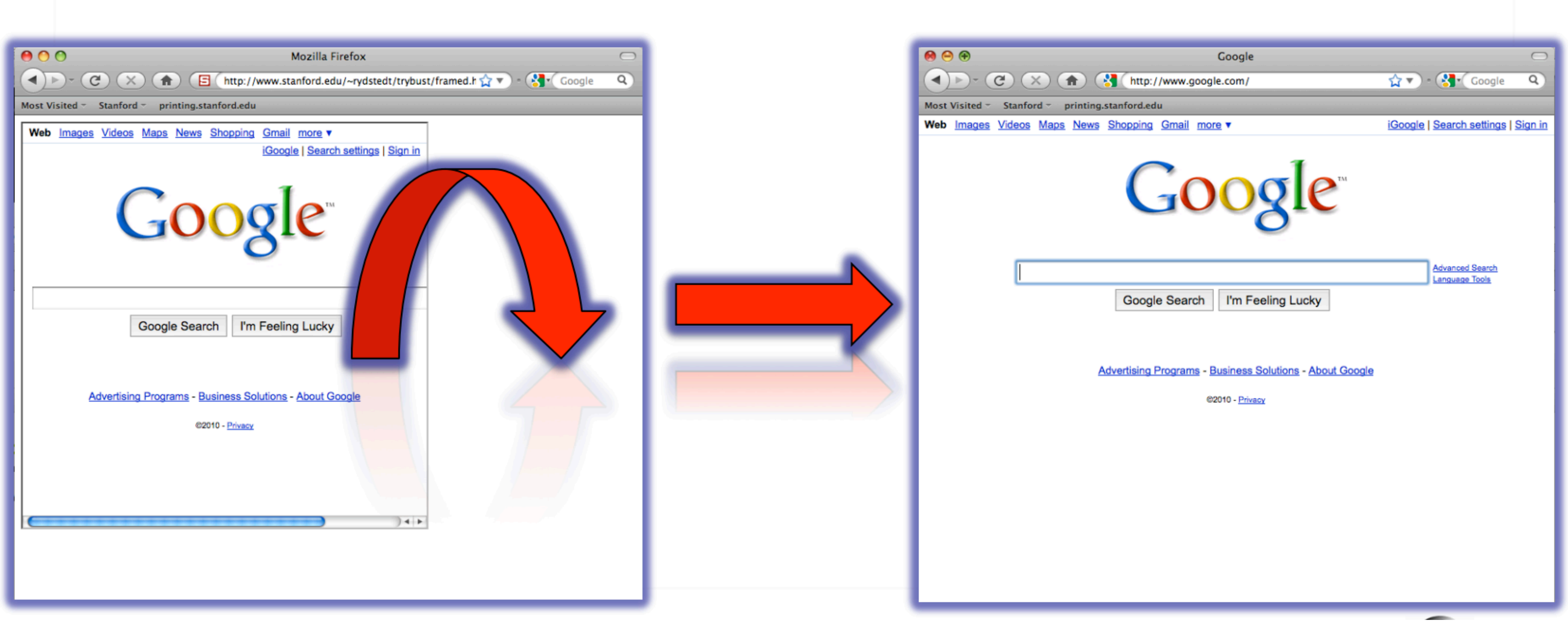
Clickjacking

Attacker uses a transparent frame to trick a user into clicking on a button or link on another page when they were intending to click on the top level page.



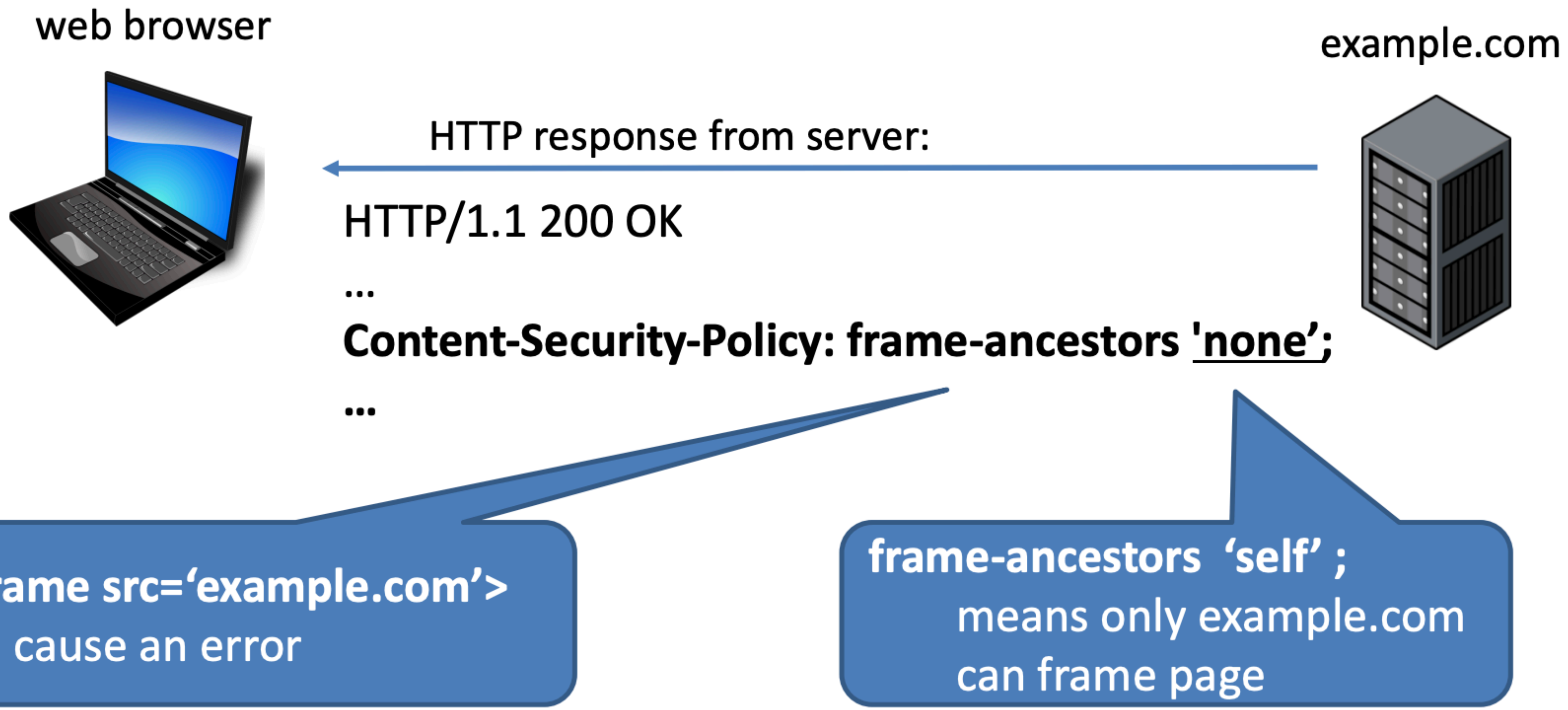
Incorrect solution: framebusting

if (top != self) { top.location = self.location; }



Easy for parent to intercept and block call to change URL of page

Correct Solution: CSP



Sub-Resource Integrity

Third-Party Content Safety

Question: how do you safely load an object from a third party service?

```
<script src="https://code.jquery.com/jquery-3.4.0.js"></script>
```

If **code.jquery.com** is compromised, your site is too!

MaxCDN Compromise

2013: MaxCDN, which hosted bootstrapcdn.com, was compromised

MaxCDN had laid off a support engineer having access to the servers where BootstrapCDN runs. The credentials of the support engineer were not properly revoked. The attackers had gained access to these credentials.

Bootstrap JavaScript was modified to serve an exploit toolkit

Sub-Resource Integrity (SRI)

SRI allows you to specify expected hash of file being included

```
<script
  src="https://code.jquery.com/jquery-3.4.0.min.js"
  integrity="sha256-BJeo0qm959uMBGb65z40ejJYGSgR1fNKwOg="
/>
```

Sub-Resource Integrity (SRI)

```
<script    src="https://code.jquery.com/jquery-3.5.1.min.js"
           integrity="sha256-9/aliU8dGd2tb6OSsuzixeV4y/faTqgFtohetphbbj0="
           crossorigin="anonymous">
</script>
```

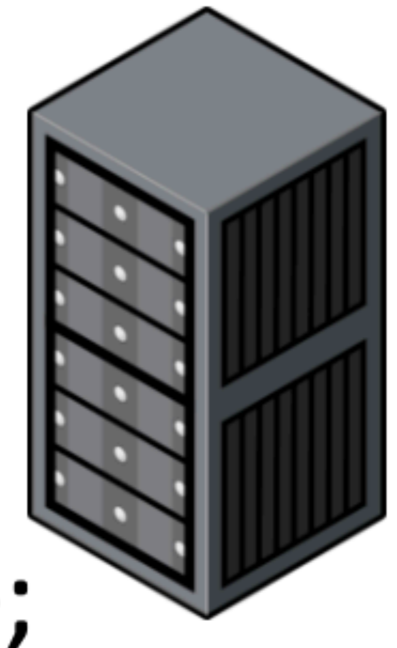
Browser: (1) load sub-resource, (2) compute hash of contents,
(3) compare value to the integrity attribute.

- if hash mismatch: script or stylesheet are not executed and an error is raised.

Enforce SRI with CSP

web browser

example.com



HTTP response from server:

HTTP/1.1 200 OK

...

Content-Security-Policy: **require-sri-for** script style;

...

Requires SRI for all scripts and style sheets on page

Securely Using Cookies

Cookies have no integrity

Users can change and delete cookie values

* Edit cookie database (FF: cookies.sqlite)
* Modify Cookie header (FF: TamperData extension)

Shopping cart software

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

User edits cookie file (cookie poisoning):

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

Similar problem with localStorage and hidden fields:

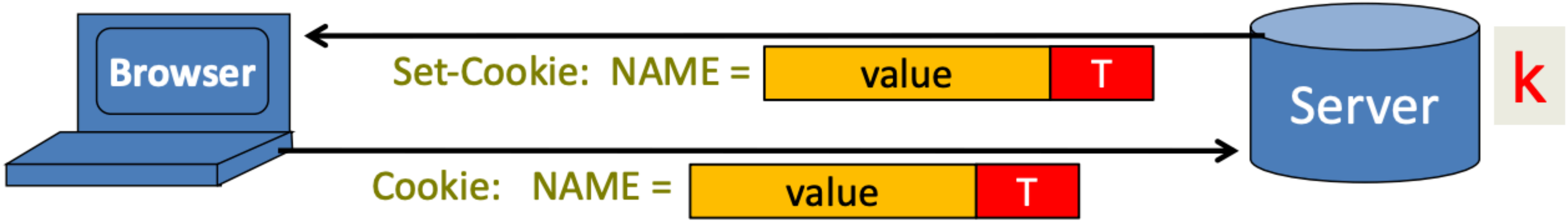
```
<INPUT TYPE="hidden" NAME=price VALUE="150">
```

Sign Cookies if Data

Goal: data integrity

Requires server-side secret key k unknown to browser

Generate tag: T ⟵ MACsign(k, (SID, name, value))



Verify tag: MACverify(k, (SID, name, value), T)

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

Protecting Cookies

Remember that you also need to limit the scope of when cookie can be used:

```
Set-Cookie: id=a3fWa;
            Expires=Wed, 21 Oct 2015 07:28:00 GMT;
            sameSite=Strict;
            Secure;
            HttpOnly
```

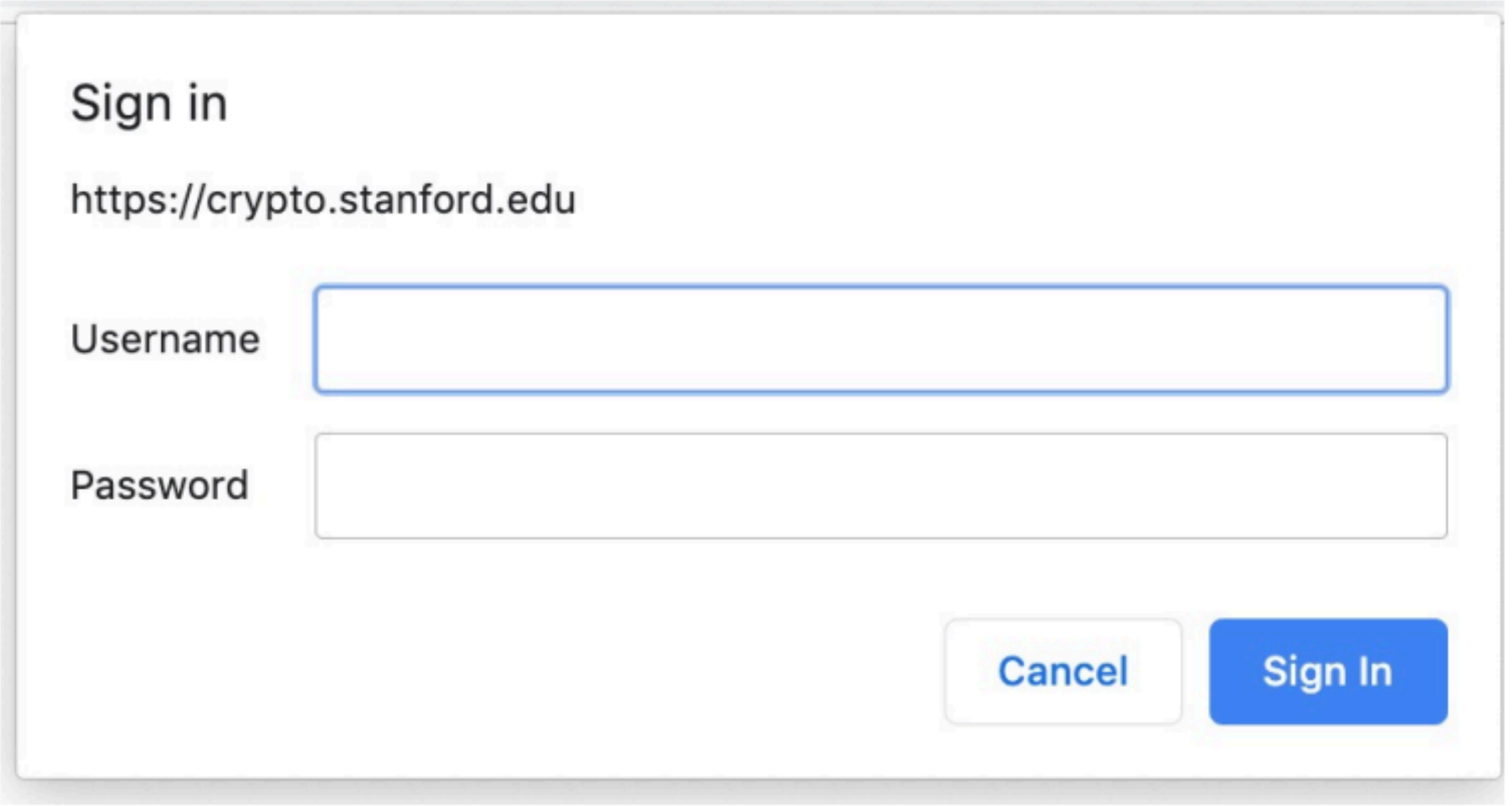
Authentication and Session Management

Pre-history: HTTP auth

HTTP request: GET   /index.html

HTTP response contains:

WWW-Authenticate: Basic realm="Password Required“



Browsers sends hashed password on all subsequent HTTP requests:

Authorization: Basic ZGFddfibzsdfgkjheczI1NXRleHQ=

HTTP auth problems

Hardly used in commercial sites:

- User cannot log out other than by closing browser
  - What if user has multiple accounts?
    multiple users on same machine?
- Site cannot customize password dialog
- Confusing dialog to users
- Easily spoofed

Do not use …

Session Management Today

GET / HTTP/1.1
cookies: []

HTTP/1.0 200 OK
cookies: [session: e82a7b92]
<html><h1>Welcome!</h1></html>

Create Anonymous Session ID

Session Management Today

```
GET  / HTTP/1.1
cookies: []
```

Create Anonymous Session ID

```
HTTP/1.0 200 OK
cookies: [session: e82a7b92]
<html><h1>Welcome!</h1></html>
```

```
GET  /loginform  HTTP/1.1
cookies: [session: e82a7b92]
```

```
HTTP/1.0 200 OK
cookies: [session: e82a7b92]
<html><form>…</form></html>
```

Session Management Today

GET / HTTP/1.1
cookies: []

HTTP/1.0 200 OK
cookies: [session: e82a7b92]
<html><h1>Welcome!</h1></html>

Create Anonymous Session ID

GET /loginform HTTP/1.1
cookies: [session: e82a7b92]

HTTP/1.0 200 OK
cookies: [session: e82a7b92]
<html><form>…</form></html>

POST /login HTTP/1.1
cookies: [session: e82a7b92]
username: zakir
password: stanford

Check Credentials + Upgrade Token

HTTP/1.0 200 OK
cookies: [session: e82a7b92]
<html><h1>Login Success</h1></html>

Session Management Today

GET / HTTP/1.1
cookies: []

HTTP/1.0 200 OK
cookies: [session: e82a7b92]
<html><h1>Welcome!</h1></html>

Create Anonymous Session ID

GET /loginform HTTP/1.1
cookies: [session: e82a7b92]

HTTP/1.0 200 OK
cookies: [session: e82a7b92]
<html><form>…</form></html>

POST /login HTTP/1.1
cookies: [session: e82a7b92]
username: zakir
password: stanford

HTTP/1.0 200 OK
cookies: [session: e82a7b92]
<html><h1>Login Success</h1></html>

Check Credentials + Upgrade Token

GET /account HTTP/1.1
cookies: [session: e82a7b92]

Session Tokens

Session Token Pitfalls

Example 1: counter

⇒ user logs in, gets counter value,
can view sessions of other users

Example 2: weak MAC. token = **{ userid, MAC_k(userid) }**

- Weak MAC exposes **k** from few cookies.

Session tokens must be unpredictable to attacker

To generate: use underlying framework (e.g. ASP, Tomcat, Rails)

Rails: token = SHA256(current time, <u>random nonce</u>)

Implementing Logout

Web sites must provide a logout function:

- Functionality: let user to login as different user
- Security: prevent others from abusing account

What happens during logout:

1. Delete SessionToken from client
2. Mark session token as expired on server

Problem: many web sites do (1) but not (2) !!

$\Rightarrow$ Especially risky in case of XSS vulnerability

How do you delete a cookie?

Cookies can have expiration dates

```
Set-Cookie: sessionID=XYZ; Expires=<Date>
```

To delete a cookie, set expiration to the past:

```
Set-Cookie: sessionID=;
   Expires=Thu, 01 Jan 1970 00:00:00 GMT
```

Authenticating Users

Plain Text Passwords (Terrible)

- Store the password and check match against user input
- Don’t trust anything that can provide you your password

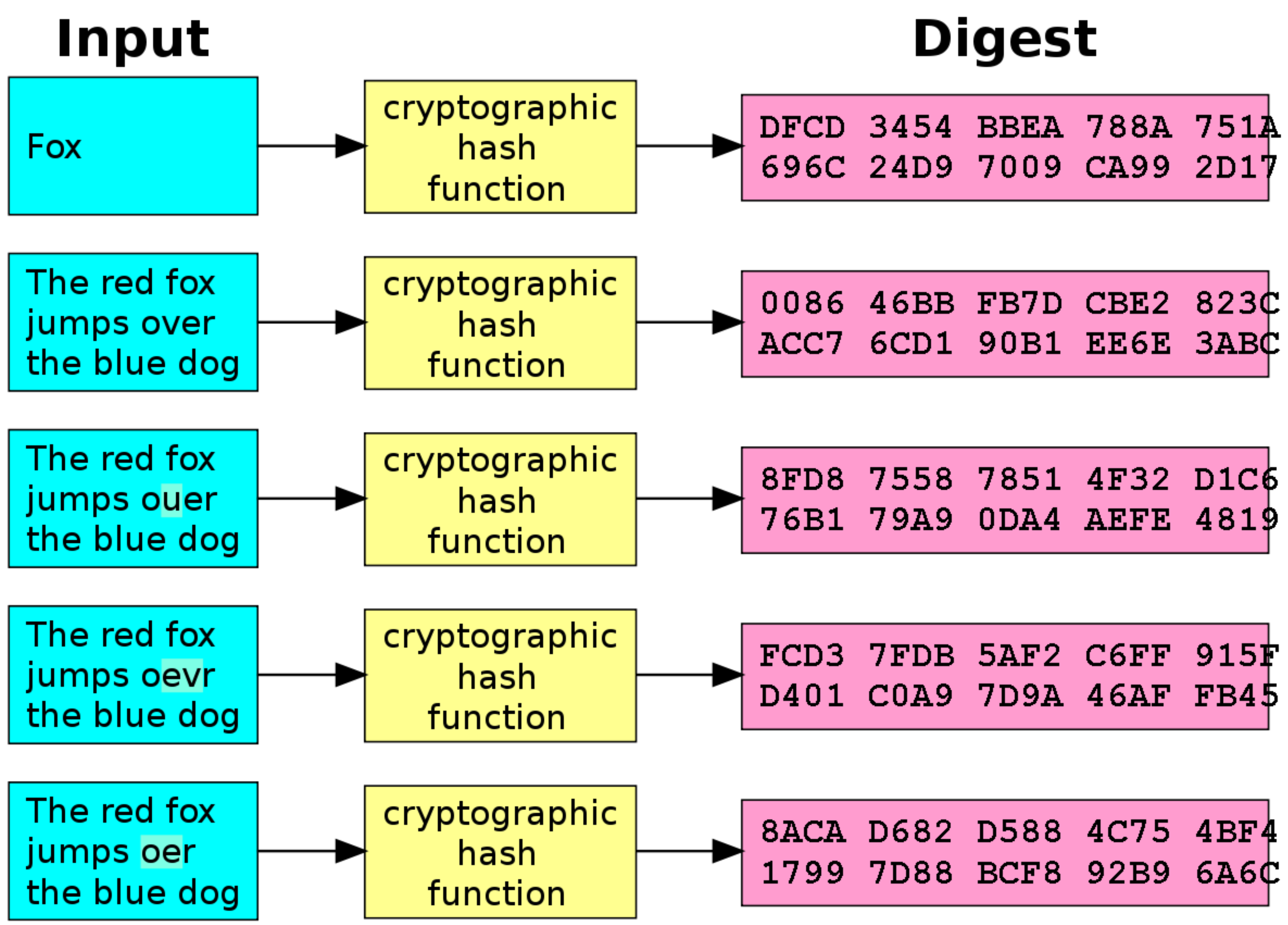
Authenticating Users

Plain Text Passwords (Terrible)

- Store the password and check match against user input
- Don't trust anything that can provide you your password

Store Password Hash (Bad)

- Store SHA-1(pw) and check match against SHA-1(input)
- Weak against attacker who has hashed common passwords



Authenticating Users

Plain Text Passwords (Terrible)

- Store the password and check match against user input
- Don't trust anything that can provide you your password

Store Password Hash (Bad)

- Store SHA-1(pw) and check match against SHA-1(input)
- Weak against attacker who has hashed common passwords

Store Salted Hash (Better)

- Store **`(r, Hash(pw||r))`** and check against **`Hash(input||r)`**
- Prevents attackers from pre-computing password hashes

Authenticating Users

Store Salted Hash (Best)

- Store (r, **H**(pw || r)) and check match against **H**(input || r)
- Prevents attackers from pre-computing password hashes

Making sure to choose an **H** that's expensive to compute:

SHA-512: 3,235 MH/s
SHA-3 (Keccak): 2,500 MH/s
BCrypt: 43,551 H/s

Use **bcrypt and salt passwords** if you're storing passwords!

Password Requirement Downfalls

Complexity (e.g., as measured by entropy) isn't necessarily strong — users add complexity in predictable ways

Requiring users to regularly change passwords leads to weak passwords

Length is the most important factor for a secure password

Modern Password Recommendations

- Minimum password length should be at least 8 characters
- Maximum password length should be at least 64 characters
    - Do not allow unlimited length, to prevent denial-of-service
        - Common gotcha: bcrypt has a max length of 72 ASCII characters
- Check passwords against known breach datasets
- Rate-limit authentication attempts
- Encourage/require use of a second factor

Designing Login Workflows

- **Helpful error messages can leak information to attackers**
  - “Invalid User ID”
  - “Invalid password for User X”
  - “Login failed; account disabled”
- **Correct response:**
  - “Login failed; invalid User ID or Password”
- Not only login — think about User Registration and Password Reset

Designing Login Workflows

- **Helpful error messages can leak information to attackers**
  - “Invalid User ID”
  - “Invalid password for User X”
  - “Login failed; account disabled”
- **Correct response:**
  - “Login failed; invalid User ID or Password”
- Not only login — think about User Registration and Password Reset

In general, error messages should not leak any information about the state of a system (in the web or beyond)

Preventing Guessing

- It's your responsibility to also prevent attackers from guessing passwords of your users:
    - Limit the rate at which an attacker can make authentication attempts, or delay incorrect attempts
    - Track of IP addresses and limit the number of unsuccessful attempts
    - Temporarily lock user account after too many unsuccessful attempts

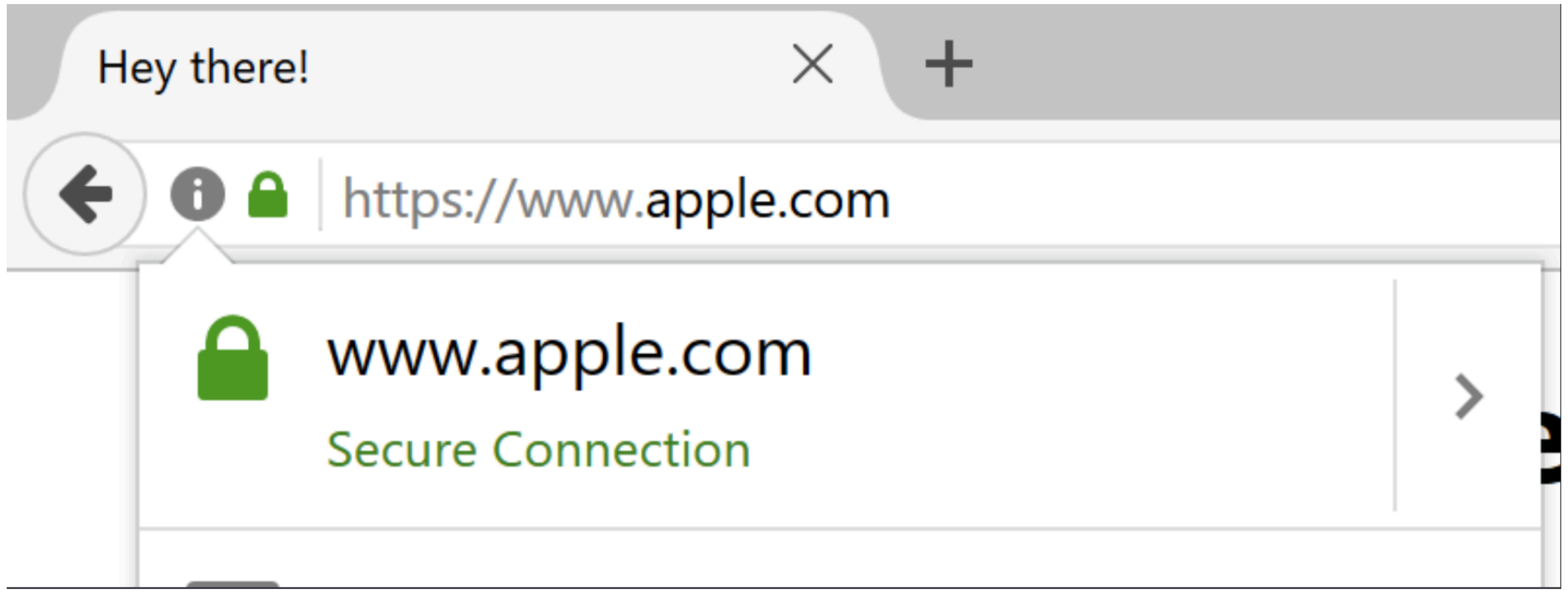
Phishing

What do Passwords Protect Against?

- A strong password can protect against:
  - **Password spray:** Testing a weak password against large number of accounts
  - **Brute force:** Testing multiple passwords from dictionary or other source against a single account
- But do not protect against:
  - **Credential stuffing:** Replaying passwords from a breach
  - **Phishing:** Man-in-the-middle, credential interception
  - **Keystroke logging:** Malware, sniffing
  - **Extortion:** Blackmail, insider threat

Phishing

- Acting like a reputable entity to trick the user into divulging sensitive information such as login credentials or account information
- Often easier than attacking the security of a system directly
  - Just get the user to tell you their password



Internationalized Domain Names (IDN)

- Domain names consist of ASCII characters
- Hostnames containing Unicode characters are transcoded to subset of ASCII consisting of letters, digits, and hyphens called punycode
- Allows registering domains with foreign characters!
    - münchen.example.com → xn--mnchen-3ya.example.com

IDN homograph attack

- Many Unicode characters are difficult to distinguish from common ASCII characters
  - apple.com vs. apple.com

xn--pple-43d.com

apple.com

Did you mean apple.com?

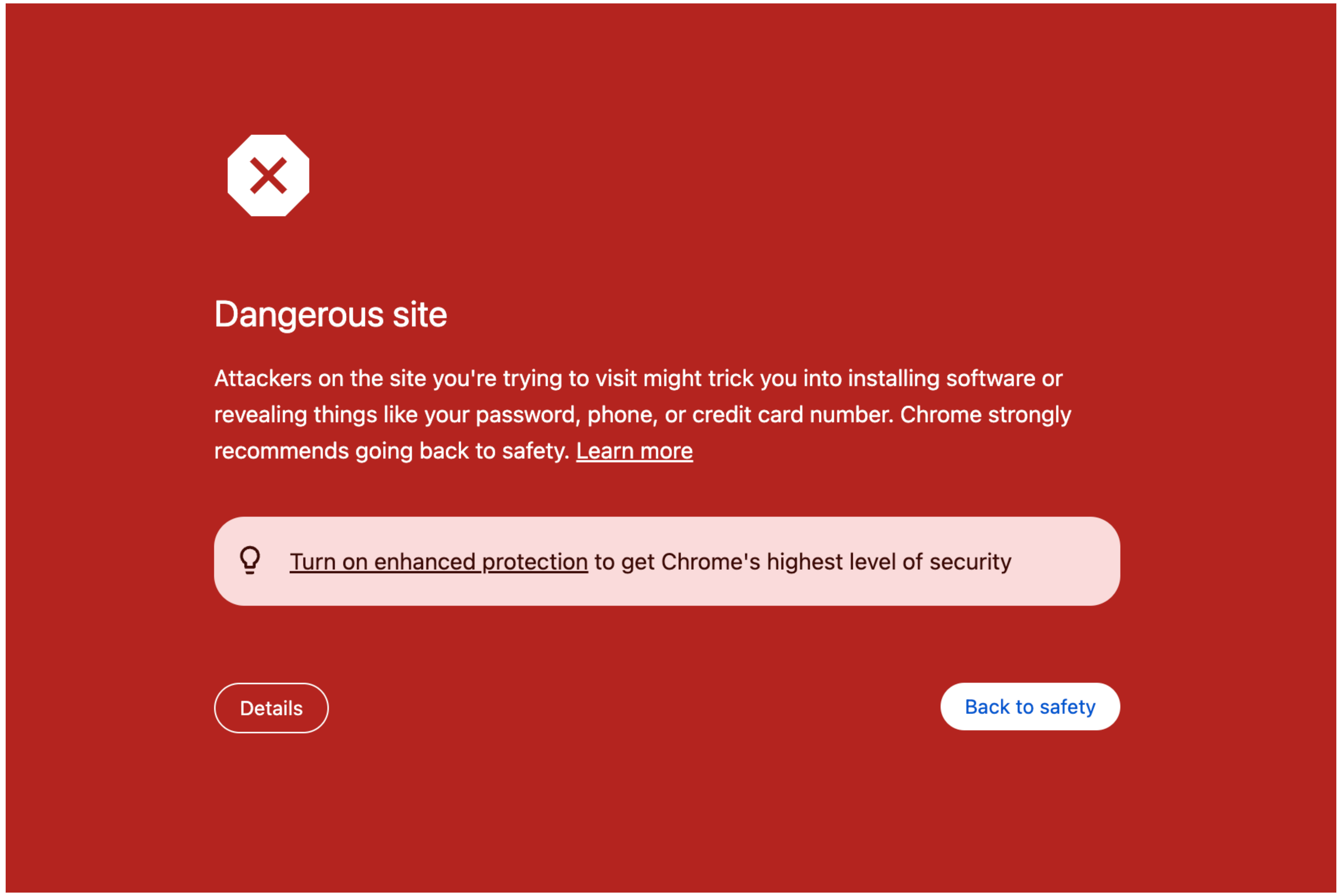
The site you just tried to visit looks fake. Attackers sometimes mimic sites by making small, hard-to-see changes to the URL.

Ignore

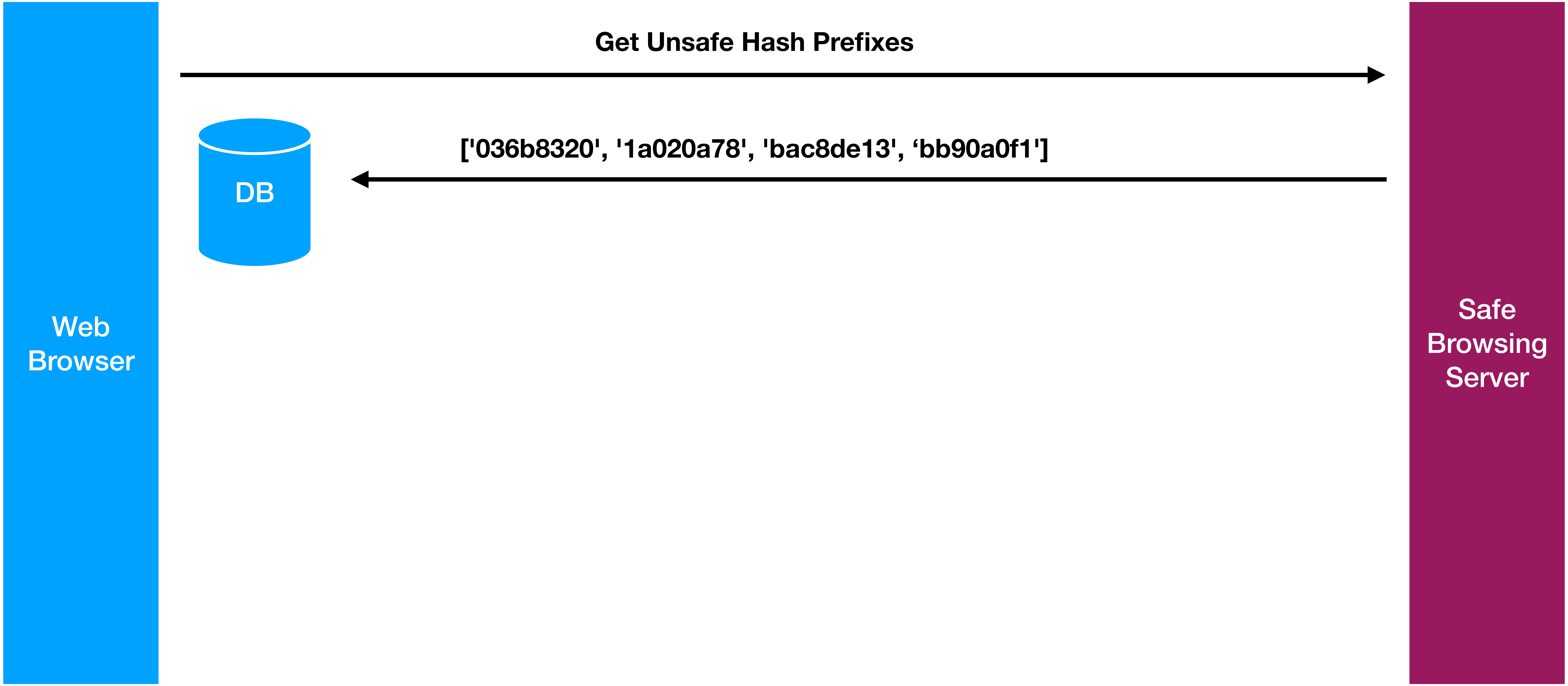
Go to apple.com

Google Safe Browsing

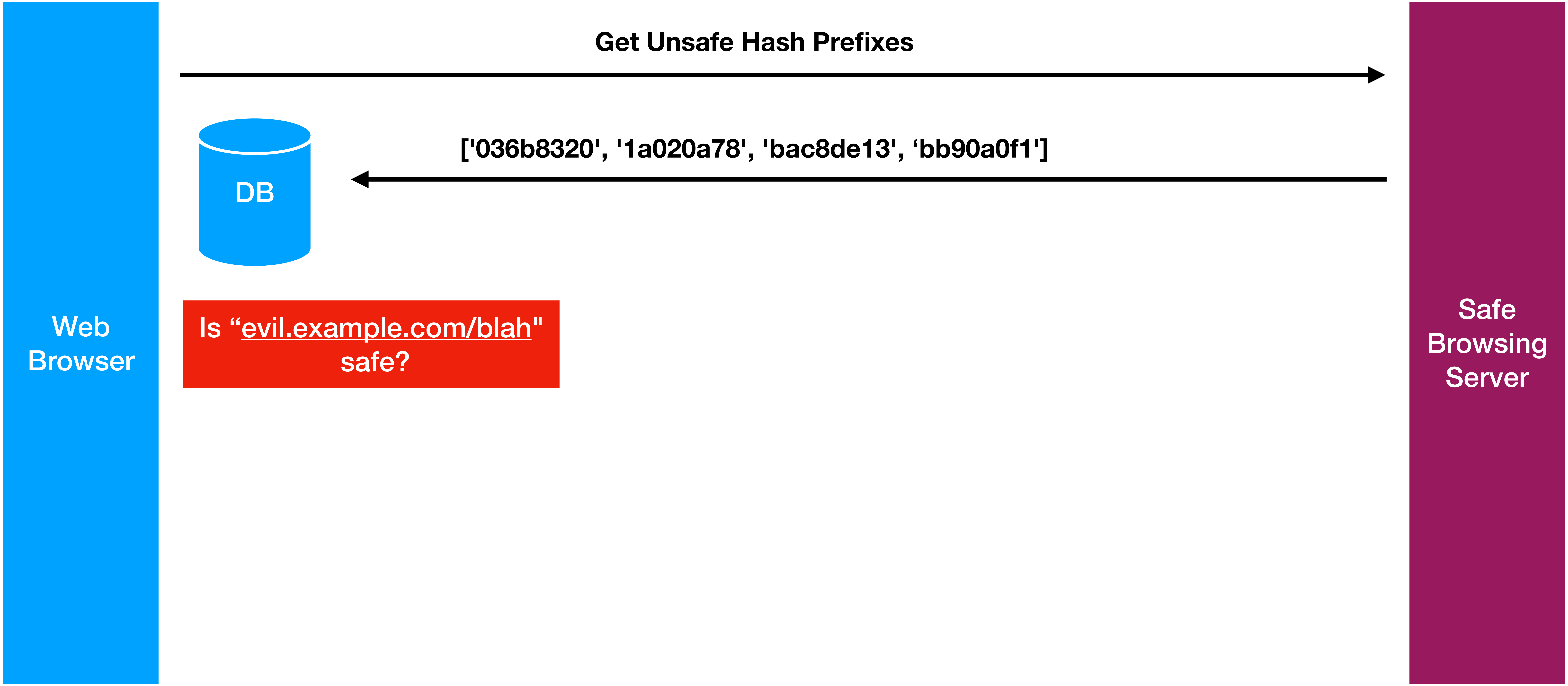
- Google maintains a list of known malware and phishing URLs — tries to protect user
- But, how do you let users look up dangerous sites without leaking all traffic to Google?



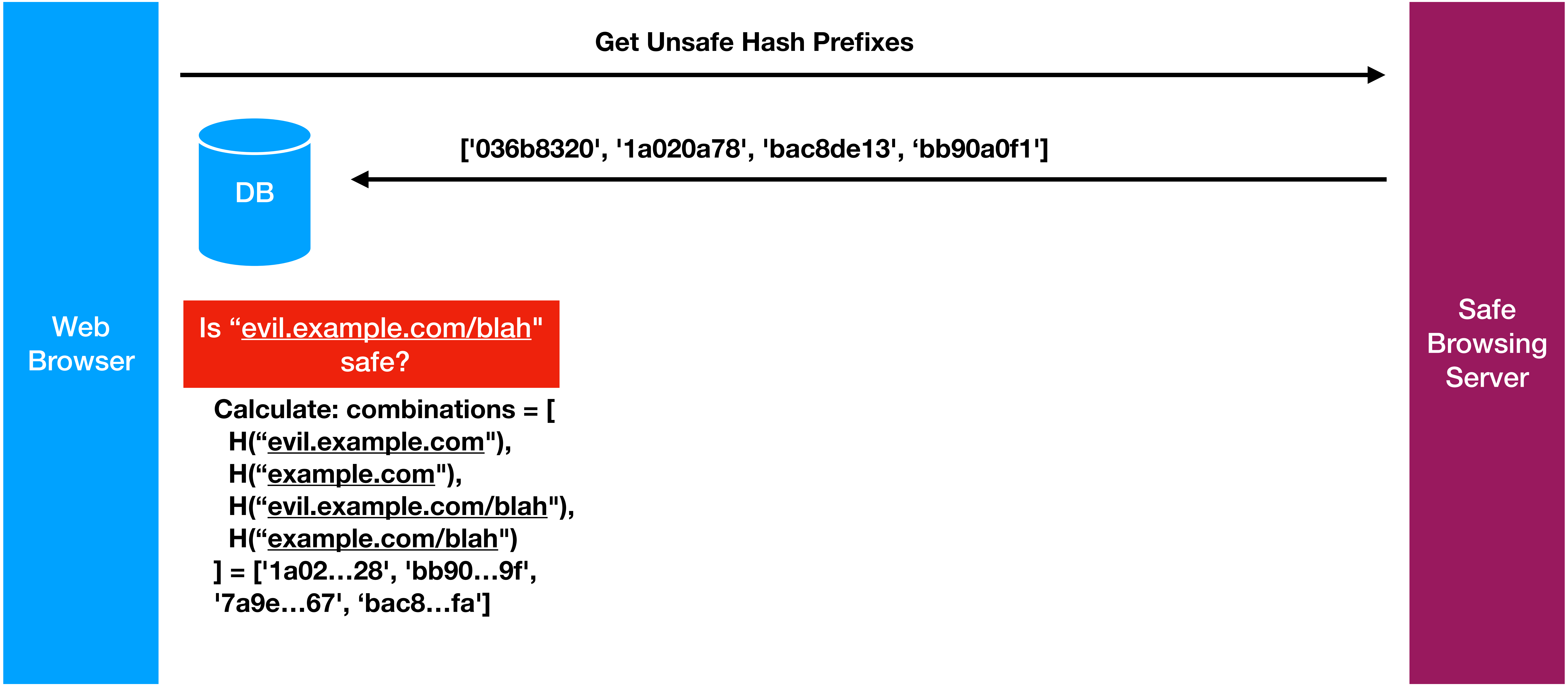
Safe Browsing Approach



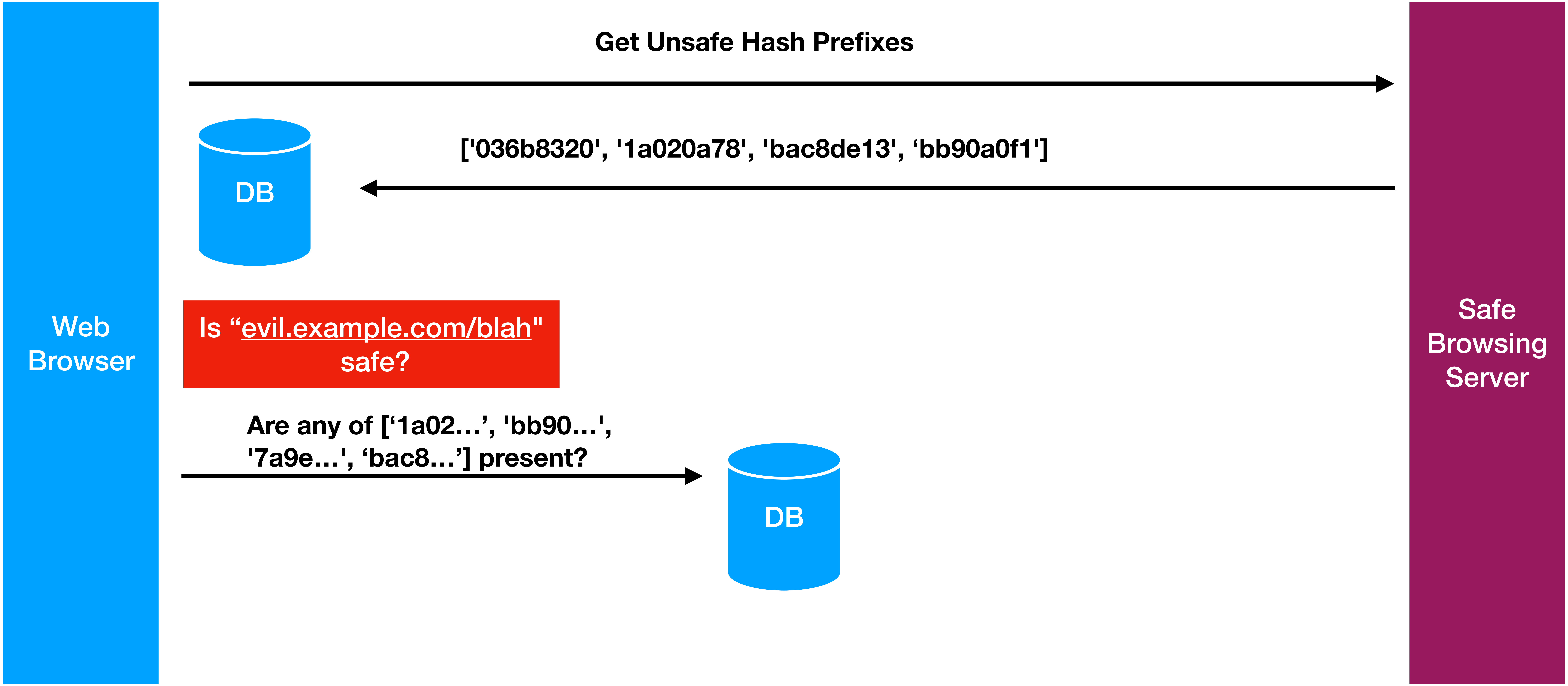
Safe Browsing Approach



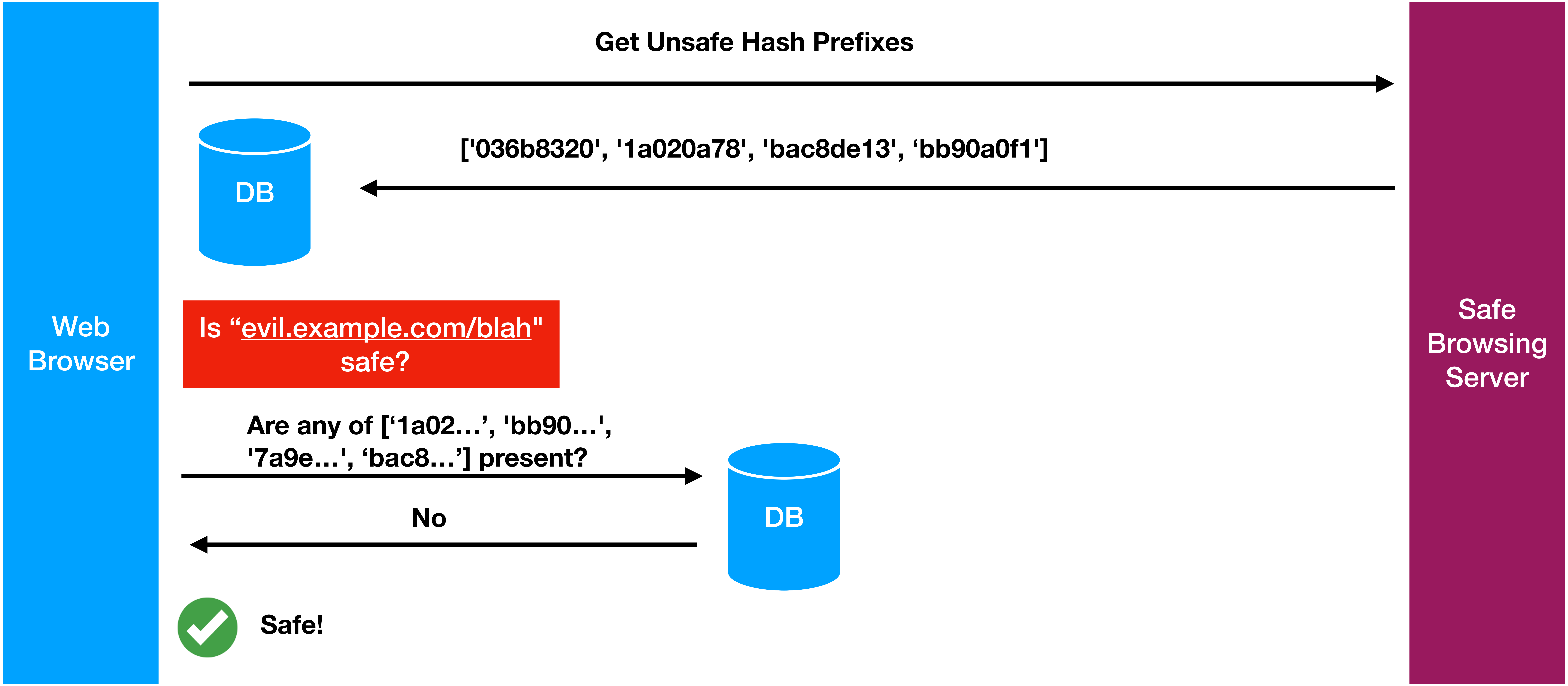
Safe Browsing Approach



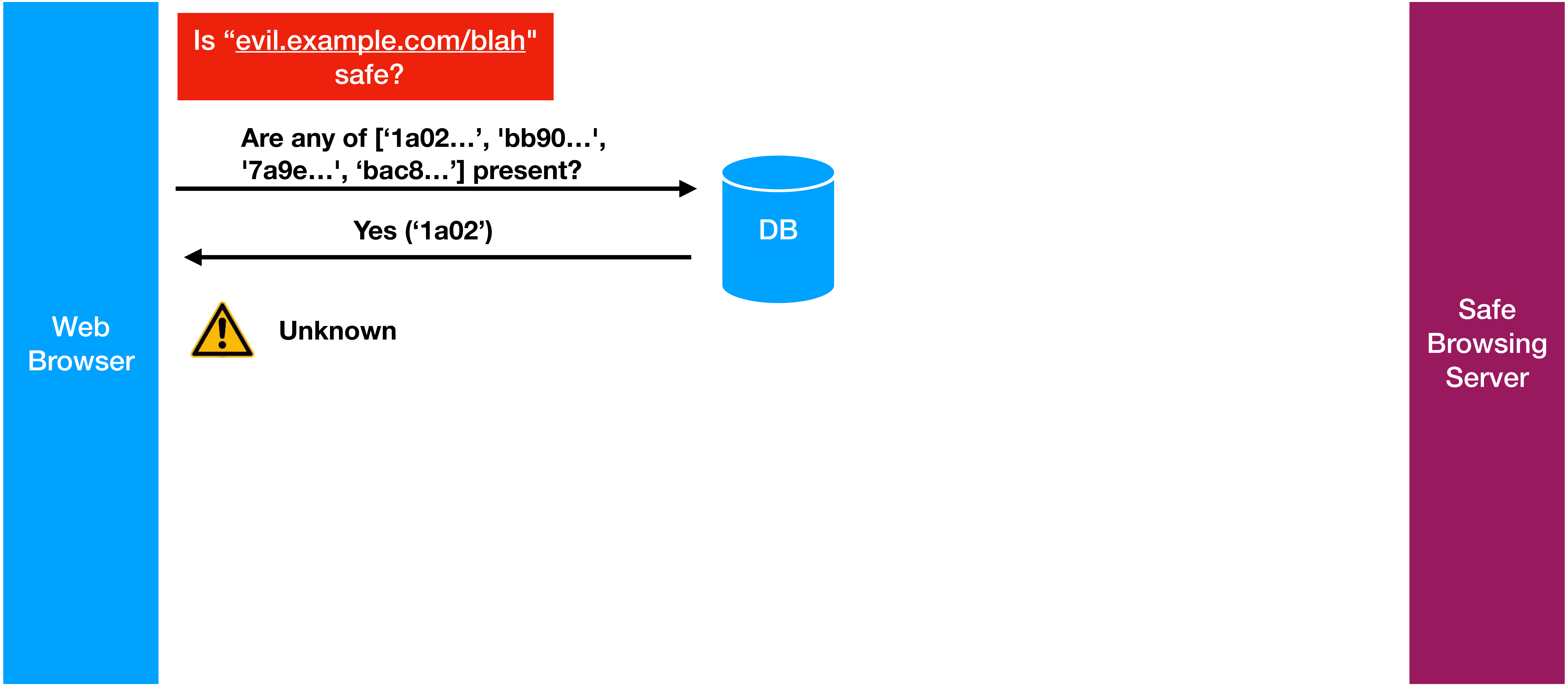
Safe Browsing Approach



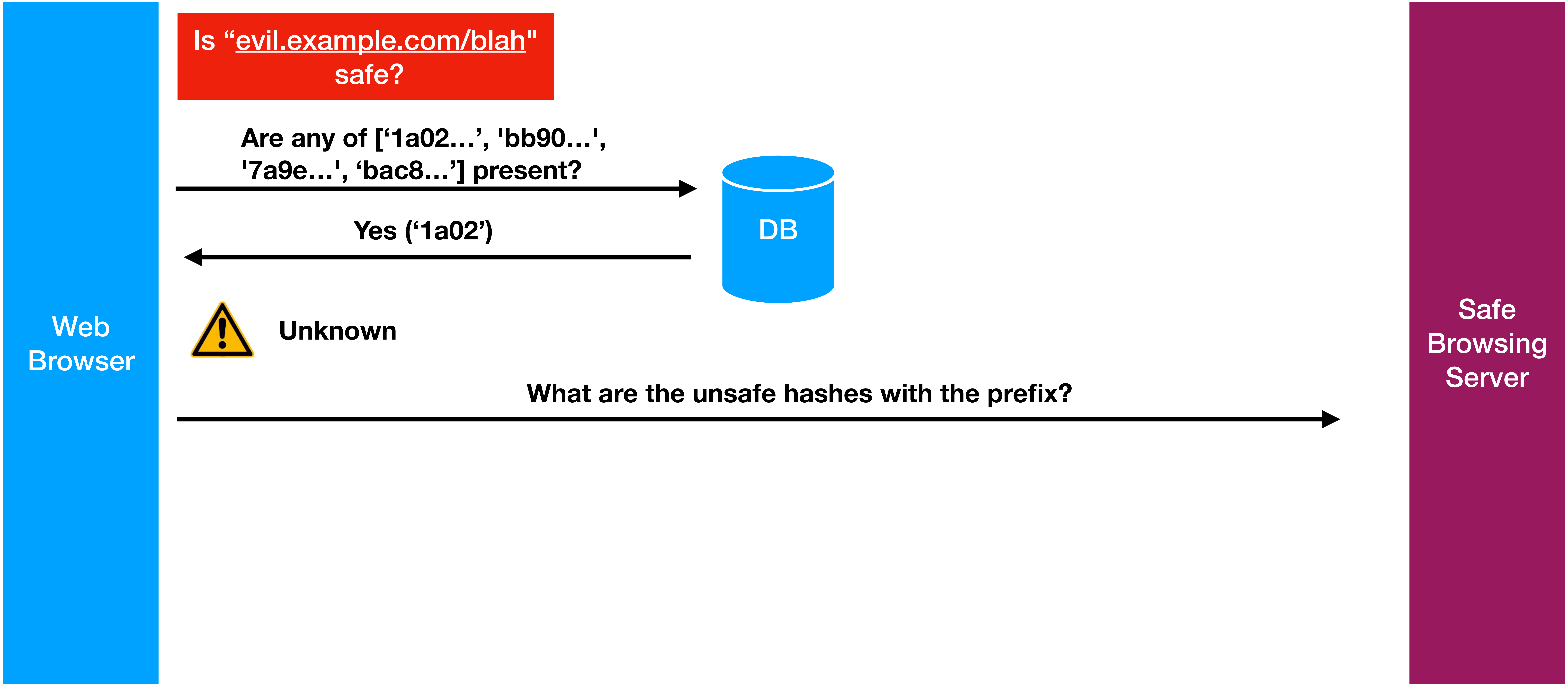
Safe Browsing Approach



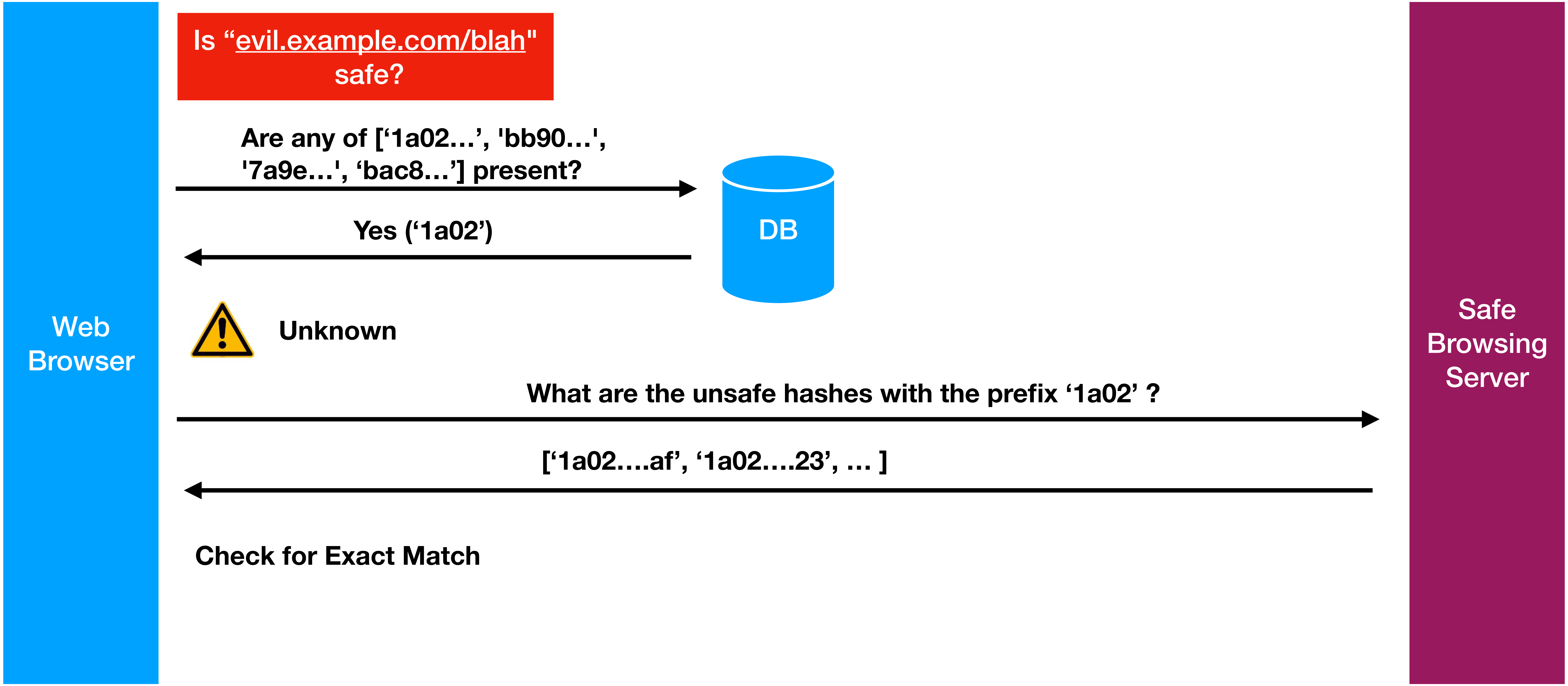
Safe Browsing Approach



Safe Browsing Approach



Safe Browsing Approach



Beyond Passwords

What do Passwords Protect Against?

- A strong password can protect against:
  - **Password spray:** Testing a weak password against large number of accounts
  - **Brute force:** Testing multiple passwords from dictionary or other source against a single account
- But do not protect against:
  - **Credential stuffing:** Replaying passwords from a breach
  - **Phishing:** Man-in-the-middle, credential interception
  - **Keystroke logging:** Malware, sniffing
  - **Extortion:** Blackmail, insider threat

';--have i been pwned?

Check if your email address is in a data breach

email address pwned?

771	13,080,233,673	115,769	228,884,627
pwned websites	pwned accounts	pastes	paste accounts

Largest breaches

772,904,991 Collection #1 accounts

763,117,241 Verifications.io accounts

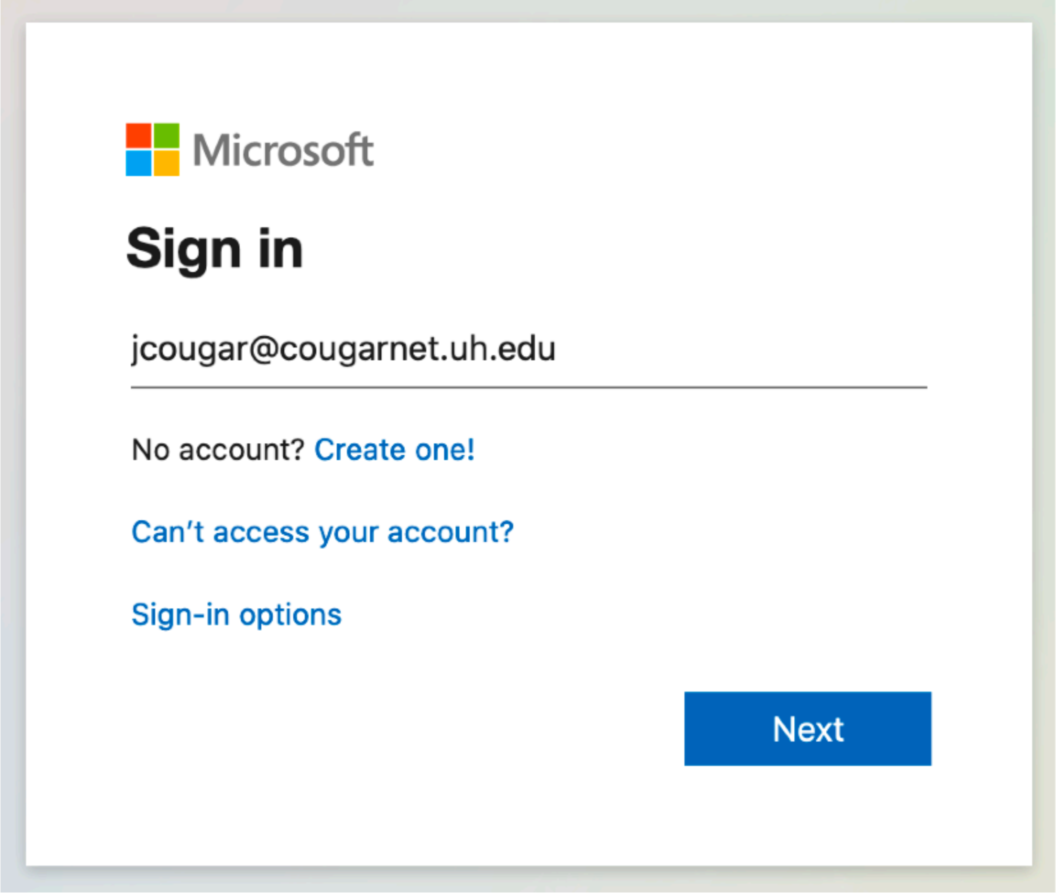
Recently added breaches

6,009,014 MovieBoxPro accounts

2,103,100 Piping Rock accounts

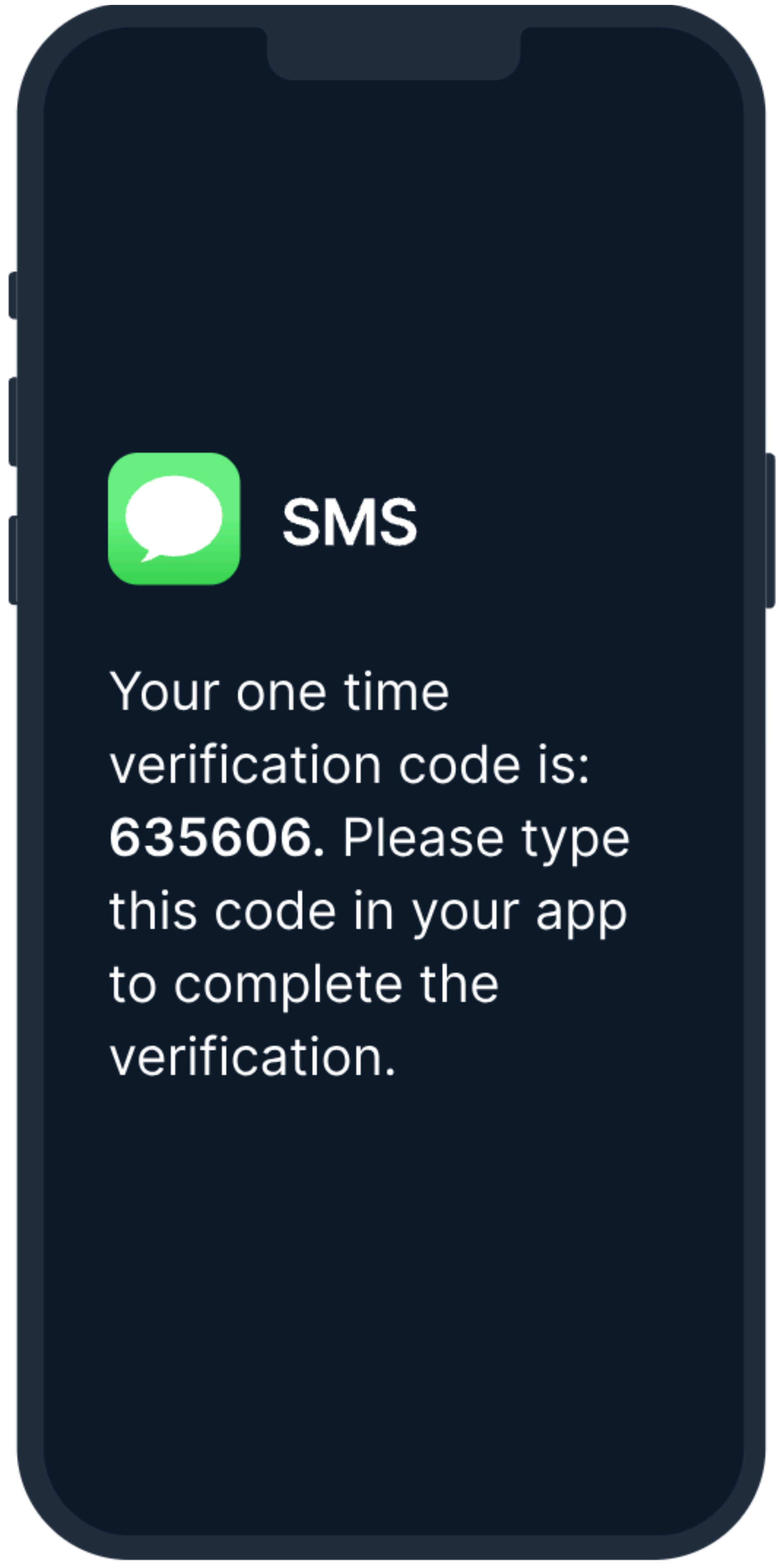
Multi-Factor Authentication

- Microsoft: “Based on our studies, your account is more than 99.9% less likely to be compromised if you use MFA”
- How are accounts compromised in practice?
  - Credential Stuffing — attackers try to log in using purchased lists of usernames and passwords
  - Phishing — users are deceived into entering their password

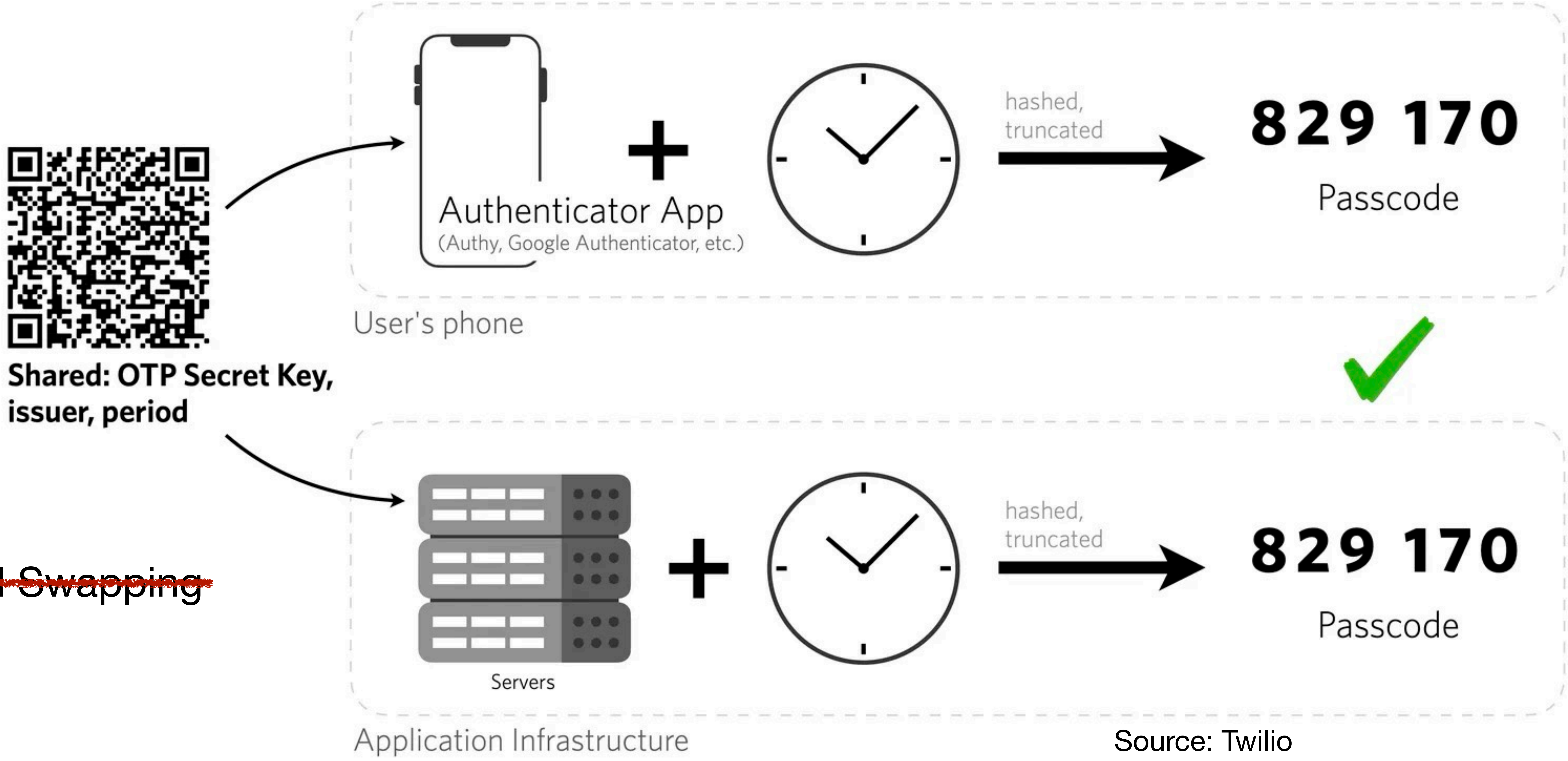


SMS-Based Two Factor

- Prevents attackers from logging in using stolen credential by sending *One Time Code (OTC)* to user
- Now considered obsolete. Fails against:
  - Phishing sites
  - SIM Swapping
  - Social Engineering Attacks



Time-based One-Time Passwords (TOTP)

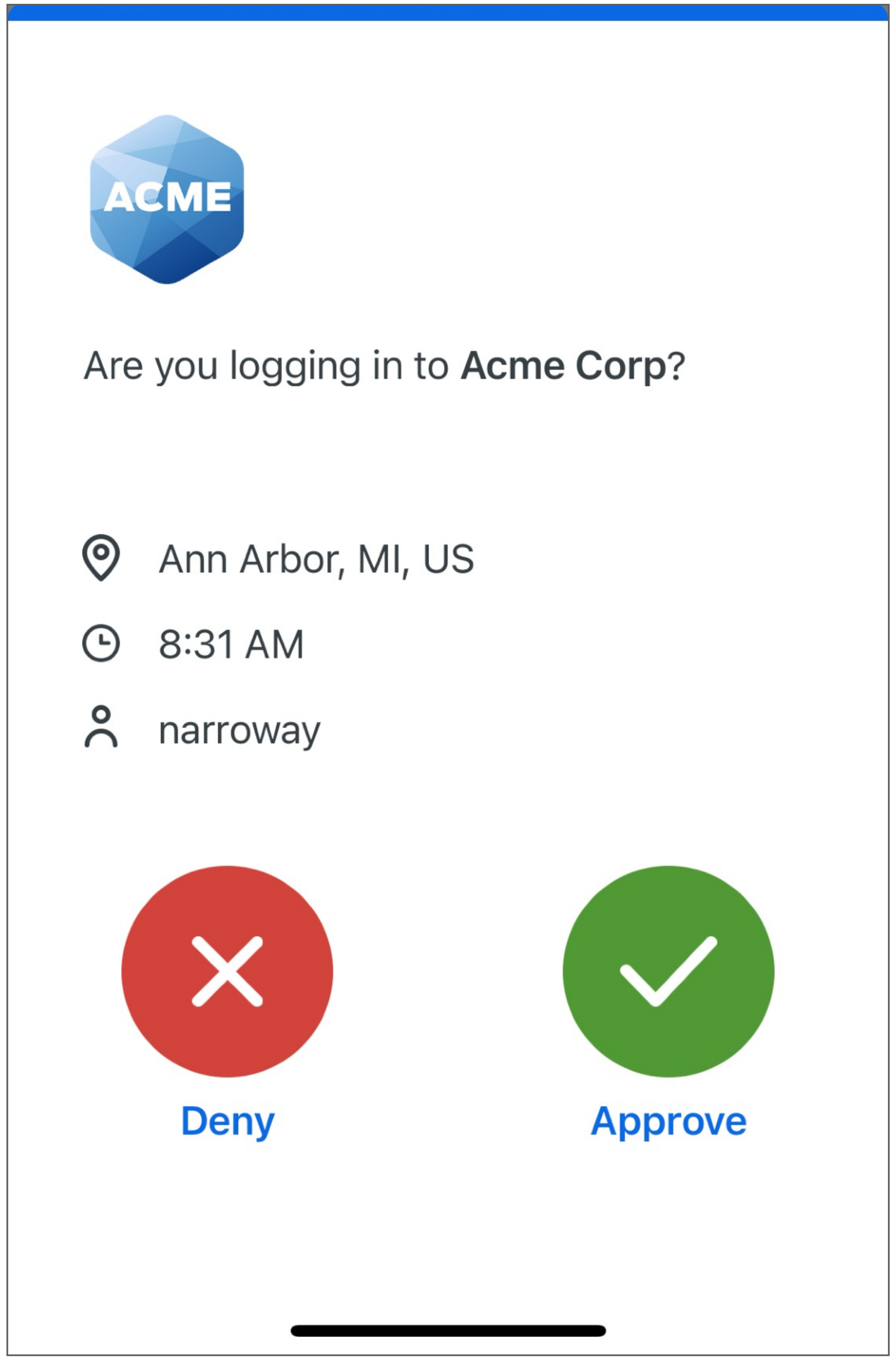


~~SIM Swapping~~

Source: Twilio

Duo Push Notifications

- Duo (or similar) Push Notifications prevent doesn't show a code — can't be stolen by an attacker
- Doesn't provide full-proof defense against "push phishing":
  - User clicking Approve out of habit
  - Real-Time Phishing Site attacks



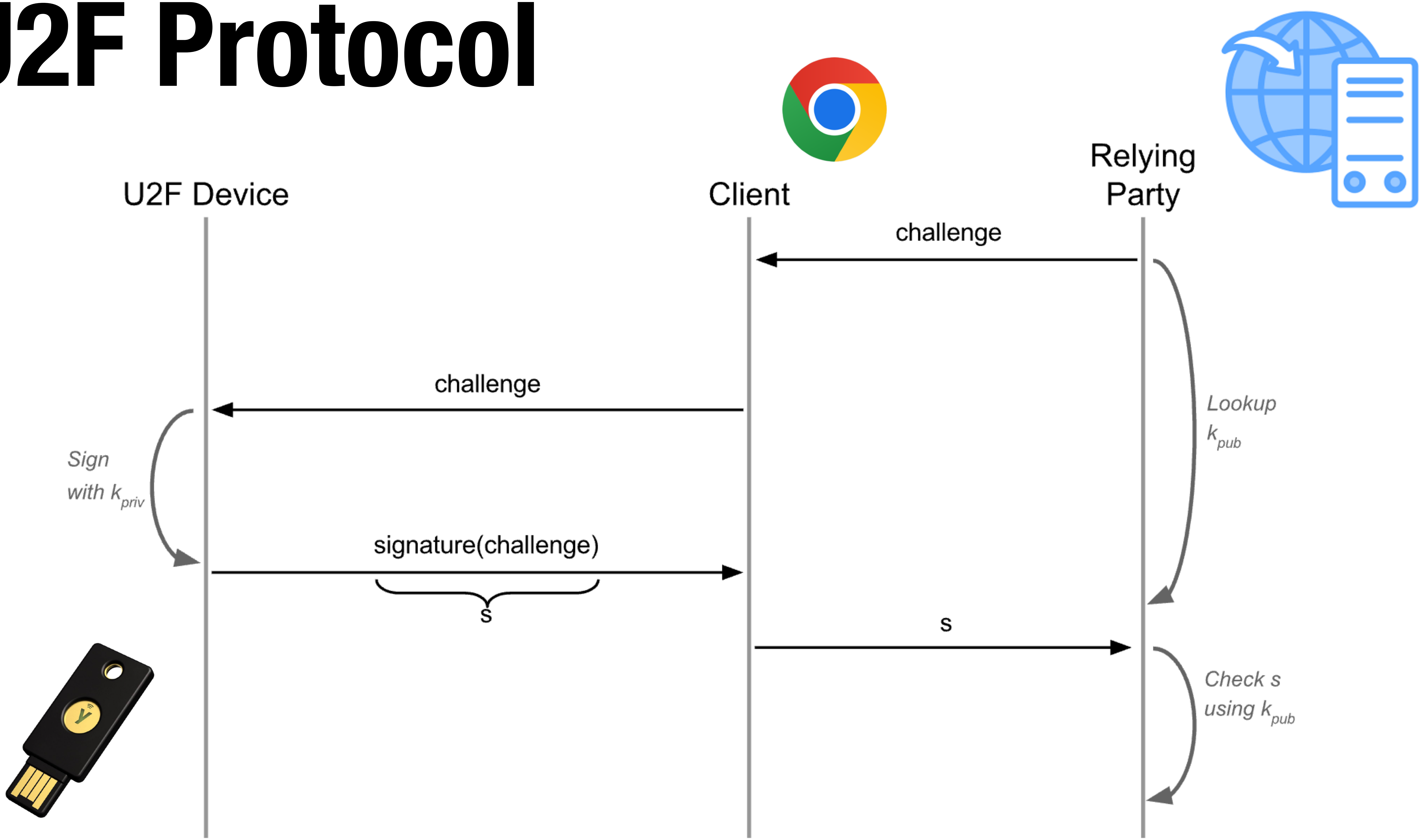
How to provide foolproof 2FA?

- Most secure solutions rely on cryptographic operation that's tied to the *website* being visited by the user
- We have fool-proof solutions today: physical security tokens and Passkeys

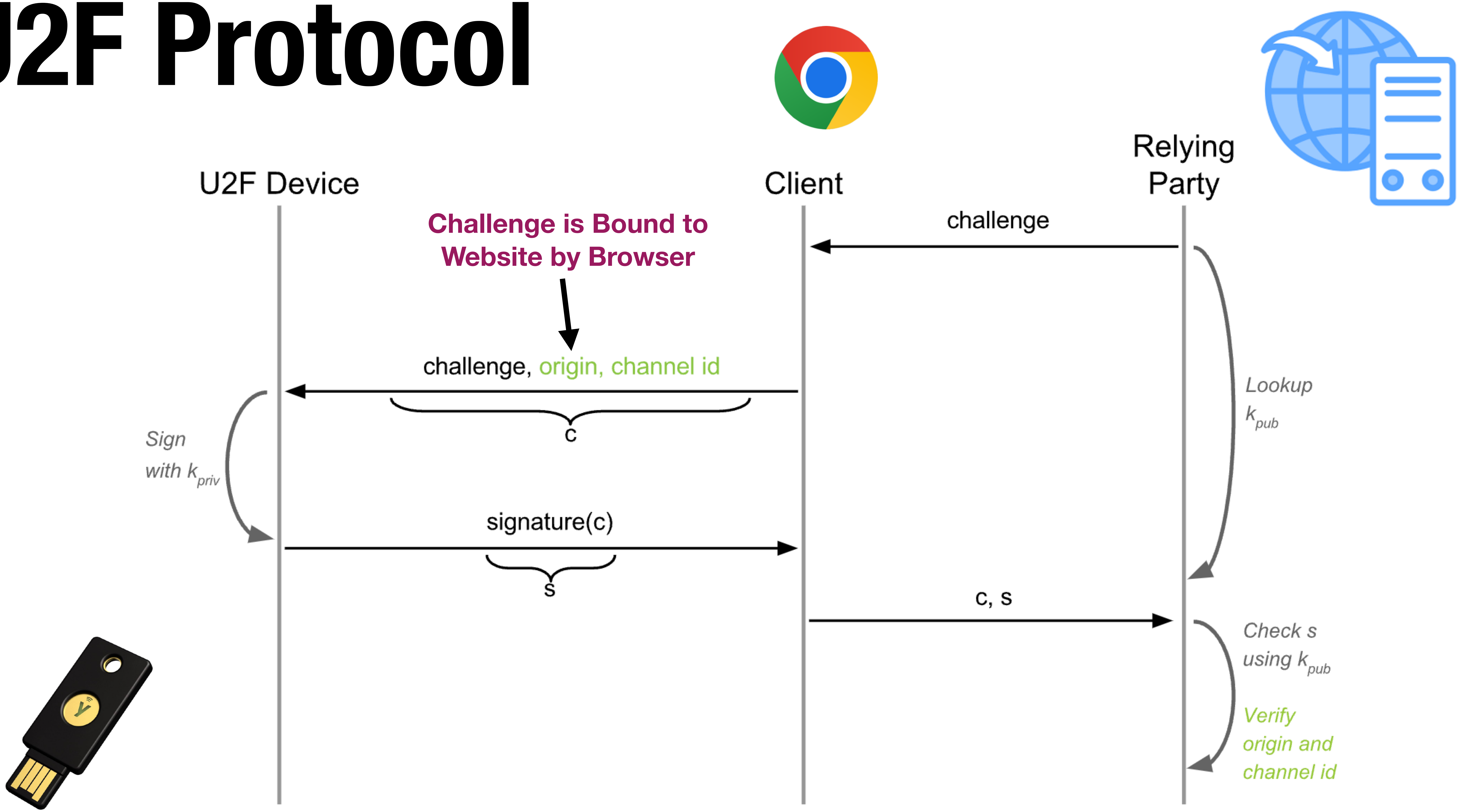
Physical Tokens

- Each token has a public and private key pair
- Private key cannot be extracted from the device
- Pushing button signs a challenge presented to the device

U2F Protocol

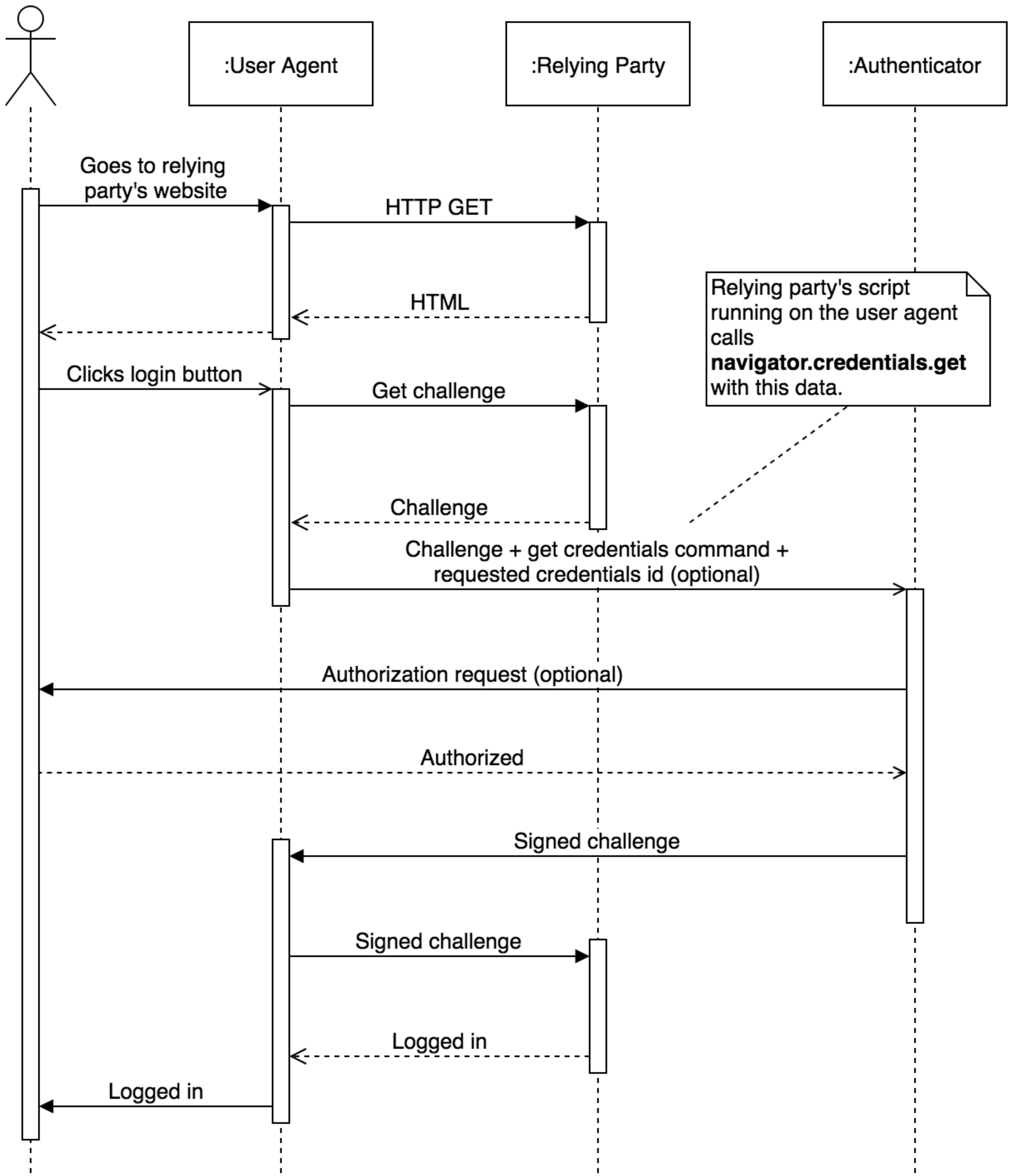


U2F Protocol



FIDO2/WebAuthN

- U2F Protocol only allowed hardware tokens to be used as a second factor
- FIDO2 allows them to be used as primary authentication mechanism
- Allows authenticators beyond hardware token (e.g., TouchID)



Pass Keys

- Technical Name: "Multi-Device FIDO Credentials"
- Public/Private key pair that is synchronized across devices (e.g., by Google or Apple) and can be used through WebAuthN API

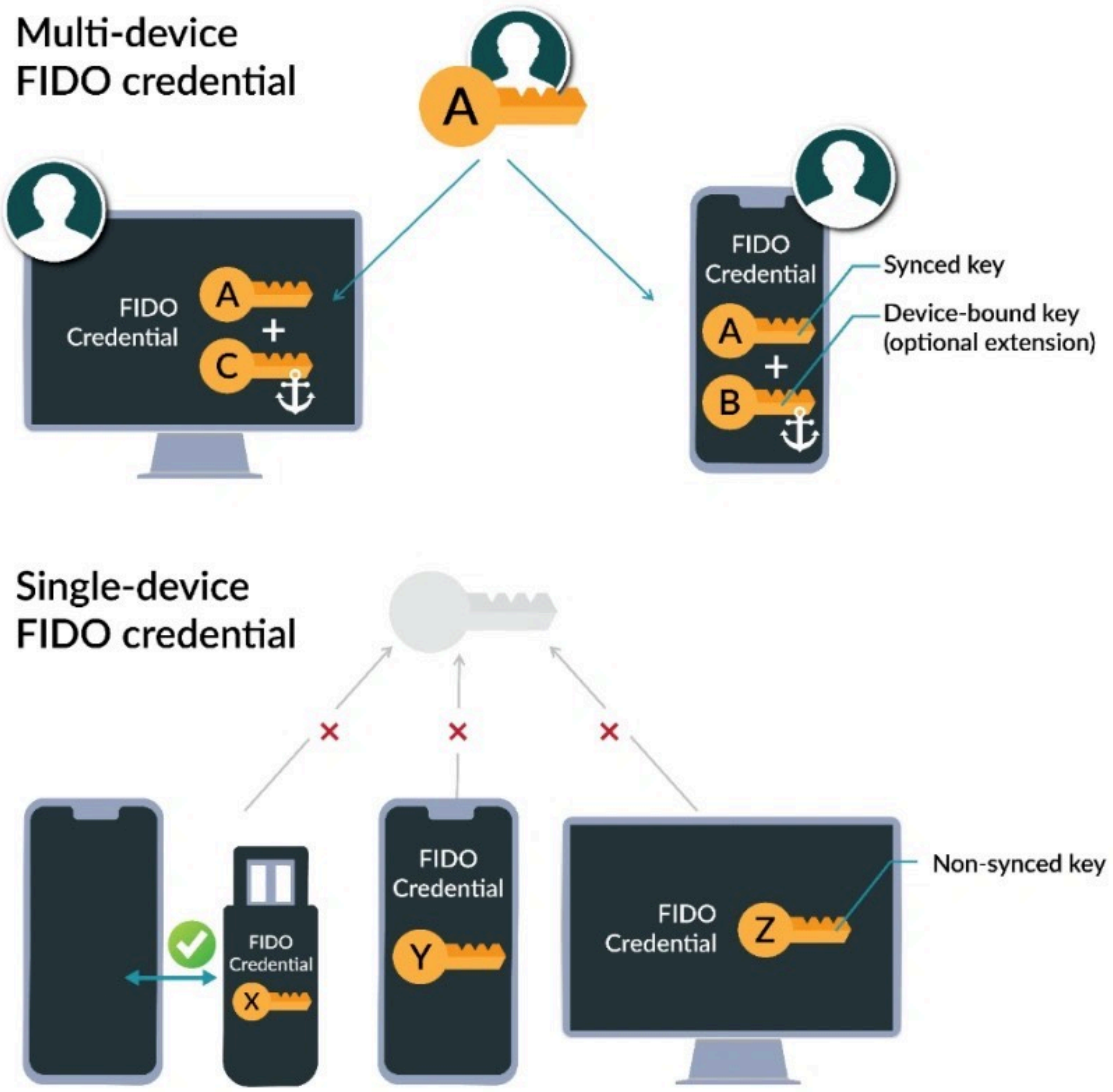


Figure 1: Multi-device vs. single-device credentials

Building a Secure Web Application

Many Steps Involved

Best Advice: Use a modern web framework — many security precautions are built in today — but don't assume!

Protect Against CSRF: Never depend on cookies to signal user intent! Use CORS Pre-Flight or CSRF Tokens.
Set cookies as **sameSite** and **secure**.

Protect Against XSS: Set a **Content Security Policy** and do not use any inline scripts. Use **httpOnly** cookies.

Protect Against SQL Injection: Use **Parameterized SQL** or Object Relational Mapper (ORM)

Many More Steps Involved

Protect Against Data Breach: Use modern hashing algorithm like BCRYPT and salt passwords

Protect Against Clickjacking: Set **Content Security Policy** that prevents you from being shown in an IFRAME

Protect Against Malicious Third Parties: Use Iframes, CSP, and HTML5 Sandboxes

Protect Against Compromised Third Parties: Use Sub-Resource Integrity Headers

Protect Against Credential Compromise and Phishing: Use U2F