CS155 (Spring 2015) Project 2

Part 1 due May 12 11:59pm || Part 2 due May 19 11:59pm

Sections for this project will be on May 1 (Part 1) and May 8 (Part 2)

In this project, you will gain experience implementing and defending against web attacks. You have been given the source code for a banking web application written in Ruby on Rails. In Part 1 of this project, you will implement a variety of attacks. In Part 2, you will modify the Rails application to defend against the attacks from Part 1.

The web application lets users manage bitbars, a new form of crypto currency. Each user is given 200 bitbars when they register for the site. They can transfer bitbars to other users using an intuitive web interface, as well as create and view user profiles.

You have been given the source code for the bitbar application. Real web attackers generally would not have access to the source code of the target website, but having source might make finding the vulnerabilities a bit easier. You should not change any of the Rails source code until Part 2.

Resources

CS142, Stanford's web programming class, has a variety of useful materials posted online. You may find the following handy:

- Notes/Slides: http://www.stanford.edu/class/cs142/cgi-bin/lectures.php
- Commands: http://www.stanford.edu/class/cs142/cgi-bin/railsCommands.php
- Installation: http://www.stanford.edu/class/cs142/cgi-bin/railsInstall.php

Note that CS142 covers these technologies in far greater depth than is needed here.

In addition, here are some other resources you may find helpful:

- HTML, CSS, JavaScript
  - http://www.w3schools.com/
- Ruby and Ruby on Rails
  - http://guides.rubyonrails.org/index.html
  - http://api.rubyonrails.org/
  - http://ruby-doc.org/
- AJAX
  - http://openjs.com/articles/ajax_xmlhttp_using_post.php
- XSS
  - http://crypto.stanford.edu/cs155/papers/CSS.pdf

  - https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
- SQL
  - http://www.w3schools.com/sql/
  - http://rails-sqli.org/
- Clickjacking Attacks and Defenses
  - http://seclab.stanford.edu/websec/framebusting/framebust.pdf
  - http://media.blackhat.com/bh-eu-10/presentations/Stone/BlackHat-EU-2010-Stone-Next-Generation-pdf
- No Explanation Needed
  - http://stackoverflow.com/

Setup Instructions

The Bitbar application should be run locally on your machine (using Ruby 2.0 and Rails 4.0). When the server is running, the site should be accessible at the URL http://localhost:3000. This is the environment the grader will use during grading, and therefore all of your attacks should assume the site is located at the URL http://localhost:3000.

On the client side, we will grade using the latest version of Chrome. The specific browser should not make a difference for most of the attacks, but we recommend you test your attacks in Chrome in order to be safe.

Detailed setup instructions:

1. Install Ruby 2.0 and Rails 4.0. See the following page for instructions:
   http://www.stanford.edu/class/cs142/cgi-bin/railsInstall.php
2. Download and extract the starter code from the CS155 website.
3. Navigate to the "bitbar" directory of the starter code. This is the Rails root directory.
4. Run the following command to install the Rails plugins you will need:

   ```
   bundle install
   ```

5. To start a server, run the following command on a terminal in your bitbar directory:

   ```
   rails server
   ```

   This will make the bitbar application available in your browser at the URL http://localhost:3000

   You can close the server by pressing Ctrl + C in the terminal. Note that you do NOT need to restart the server every time you make a change to the Rails source; the running Rails server will automatically update the website when you make a change to the source code.

Part 1: Attacks

For the first part of the project, you will write a series of attacks against the Bitbar application. All of your attacks should assume the site is accessible at the url http://localhost:3000.

You may not use any external libraries. In particular, this means no jQuery.

You may use online resources, but please cite them in a file called `README.txt` inside your attacks directory.

Warm-Up Exercise: Cookie Theft

- Your solution is a URL starting with

  `http://localhost:3000/profile?username=`

- The grader will already be logged in to Bitbar before loading your URL.
- Your goal is to steal the user's *bitbar*session cookie and send it to

  `http://localhost:3000/steal_cookie?cookie=...cookie data here...`

- You can view the most recently stolen cookie using this page:
  http://localhost:3000/view_stolen_cookie
- The attack should not be visibly obvious to the user. Except for the browser location bar (which can be different), the grader should see a page that looks as it normally does when the grader visits their profile page. No changes to the site's appearance or extraneous text should be visible. Avoiding the blue warning text stating that a user is not found is an important part of the attack. It is ok if the number of bitbars displayed or the contents of the profile are not correct (so long as they look "normal"). It's also ok if the page looks weird briefly before correcting itself.
- Put your final answer in a file named warmup.txt
- **Hint**: try adding things random text to the end of the URL. How does this change the HTML source code loaded by the browser?

Attack A: Session Hijacking with Cookies

- In this attack, you will have the login credentials of the user `attacker`, and try to impersonate `user1`, who has a user id of 1. The password for the user `attacker` is also `attacker`.
- Your solution is a script that when executed will output to the console a line of Javascript that can be copied and pasted into the JavaScript console to trick the web application into thinking you are logged in as `user1`.
- Put your final answer in a script of the form `a.sh`. This script can call one or more other scripts in the submission directory as needed.
- **Hint**: How does the site store its sessions? How does the site verify which user is currently logged in? How does the site verify the integrity of its cookies?

Attack B: Cross-Site Request Forgery

- Your solution is a short HTML document named b.html that the grader will open using the web browser.
- The grader will already be logged in to Bitbar before loading your page.
- Transfer 10 bitbars from the grader's account to the "attacker" account. As soon as the transfer is complete, the browser should redirect to http://crypto.stanford.edu/cs155/ (so fast the user might not notice).
- The location bar of the browser should not contain localhost:3000 at any point.

Attack B+: Cross-Site Request Forgery With User Assistance

- Your solution is one or two HTML documents named bp.html and (optionally) bp2.html. The grader will open bp.html using the web browser.
- The grader will already be logged in to Bitbar before loading your page.
- The grader will interact with the web page in a way that is reasonable. This means that if there is a button on the web page for the user to click, the grader will click it. The same goes for any other interaction on the page.
- After interacting with the page, 10 bitbars should be transferred from the grader's account to the "attacker" account. As soon as the transfer is complete the browser should be redirected to `http://crypto.stanford.edu/cs155/`.
- Your attack must work by involving user interaction (i.e. do NOT just do another CSRF attack against `post_transfer`). In particular, you attack must target the pages `http://localhost:3000/super_secure_transfer` and/or `http://localhost:3000/super_secure_post_transfer`, which have a weak CSRF protection implemented. You may NOT directly call `http://localhost:3000/transfer` or `http://localhost:3000/post_transfer` in any way for this attack.
- It should not be obvious that your page is loading content from `http://localhost:3000`.
- There is some basic framebusting code that protects the application from this sort of attack. You must come up with a way of bypassing this defense. You may NOT used the disable_fb=yes parameter.
- Make sure to test your page layout in the virtual machine with default settings.

Attack C: Little Bobby Tables (aka SQL Injection)

- Your solution is a malicious username that allows you to close an account that you do not have login access to.
- The grader will create a new user account with your provided username, click on "Close" and then confirm that he/she wants to close the current account.
- As a result user3 should be deleted from the database. The new user account should also be deleted to leave no trace of the attack behind. All other accounts should remain.
- You can view the list of all active accounts using this page:

  `http://localhost:3000/view_users`
- If you mess up your user database while working on the problem, stop Rails and run `rake db:reset` to reset your database.
- Put your final answer in a file named `c.txt`
- **Hint**: read up on SQL injections and the WHERE clause.

Attack D: Profile Worm

- Your solution is a profile that, when viewed, transfers 1 bitbar from the current user to a user called "attacker" and replaces the profile of the current user with itself.
- Submit a file named d.txt containing your malicious profile.
- Your malicious profile should include a witty message to the grader (to make grading a bit easier).
- To grade your attack, we will cut and paste the submitted profile into the profile of the "attacker" user and view that profile using the grader's account. We will then view the copied profile with more accounts, checking for the transfer and replication.

- The transfer and application should be reasonably quick (under 15 seconds). During that time, the grader will not click anywhere.
- During the transfer and replication process, the browser's location bar should remain at:

  `http://localhost:3000/profile?username=x`

  where `x` is the user whose profile is being viewed. The visitor should not see any extra graphical user interface elements (e.g. frames), and the user whose profile is being viewed should appear to have 10 bitbars.
- You will not be graded on the corner case where the user viewing the profile has no bitbars to send.
- **Hint**: The site allows a sanitized subset of HTML in profiles, but you can get around it. The MySpace vulnerability may provide some inspiration.

Attack E (Extra Credit): Password Theft

- Your solution is an HTML document named e.html that the grader will open using the web browser.
- The grader will not be logged in to Bitbar before loading your page.
- Upon loading your document, the grader should be immediately redirected to `http://localhost:3000/login`. The grader will enter a username and password and press the "Log in" button.
- When the "Log in" button is pressed, steal the username and password by making a HTTP POST request to http://localhost:3000/steal_login?username=...&password=... (insert the correct username and password).
- You can view the most recently stolen username and password by going to:

  `http://localhost:3000/view_stolen_login`
- The login form should appear perfectly normal to the user. No extraneous text (e.g. warnings) should be visible, and assuming the username and password are correct the login should proceed the same way it always does.
- Hint: the site uses Rail's built-in escaping mechanism to escape the username, but something isn't quite right...

Grading

Beware of Race Conditions: Depending on how you write your code, all of these attacks could potentially have race conditions that affect the success of your attacks. Attacks that fail on the grader's browser during grading will receive less than full credit. To ensure that you receive full credit, you should wait after making an outbound network request rather than assuming that the request will be sent immediately.

Submission

- Create files named `warmup.txt`, `a.sh` (along with any relevant scripts), `b.html`, `bp.html` (and optionally `bp2.html`), `c.txt`, `d.txt`, and `e.html` (if you did the extra credit) containing your attacks. Please do not submit an `e.html` if you did not implement the attack.
- Create a file called `README.txt` inside the attacks directory which cites the online references you used, as well as any special notes for the grader.

- Create a file named `ID.csv` in the attacks/ directory with a comma-separated line for each group member with your SUID number, Leland username, last name, first name (order matters).
- Run the following command from the project root directory to generate the submission tarball submission_part1.tar.gz:

  ```
  make submission1
  ```

- Upload the submission tarball to a Stanford machine (i.e. Myth or Corn)
- To submit, run the following command from the directory containing the tarball:

  ```
  /usr/class/cs155/bin/submit proj2_part1
  ```

- Follow the on-screen instructions
- You can check whether we have received your submission by going into /usr/class/cs155/submissions/proj2_part1/

Part 2: Defenses

For this part, you should modify the Bitbar application to defend against the six attacks from Part 1. You must defend against Attack E even if you did not do the extra credit in the first part. There might be more than one way of implementing the attacks, so think about other possible attack methods - you must defend against all of them.

Do not change the site's appearance or behavior on normal inputs. A non-malicious user should not notice anything different about your modified site.

When presented with bad inputs, your site should fail in a user-friendly way. You can sanitize the inputs or display an error message, though sanitizing is probably the more user-friendly option in most cases.

Rails comes with a number of built-in defenses that were disabled in Part 1. These built-in defenses must be disabled. Although in the real world it's better to use standard, vetted defense code instead of implementing your own, we want you to get practice implementing them. In particular, that means you cannot use the following, except as specified in the special notes.

- Use Rail's built-in protect_from_forgery function for CSRF
- Modify any HTTP headers (including cookies)

You are allowed to use Rail's sanitization functions, but make sure you don't over-sanitize (for example, the profile should still allow the same set of sanitized HTML tags minus any you deem unsafe).

Special Notes

- For Attack A, it suffices to describe the defense without actually implementing it.
- For the Warm-Up Exercise and Attack D, you must use a Content Security Policy header for the defense, and then modify any code necessary to maintain the functionality of the site.

Submission

- For part 2, you will submit your modified BitBar source code and a write up that briefly describes the changes you made. Use the file, README.txt, located in the bitbar/ directory for your writeup. For each change you make, provide a brief description of the change (and where in the code you made it), what vulnerability(ies) it fixes, and any changes in functionality that might be observed.
- Fill in the ID.csv file in the bitbar/ directory with a comma-separated line for each group member with your SUID number, Leland username, last name, first name (order matters).
- Run the following command from the project root directory to generate the submission tarball submission_part2.tar.gz:

  ```
  make submission2
  ```

- Upload the submission tarball to a Stanford machine (i.e. Myth or Corn)
- To submit, run the following command from the directory containing the tarball:

  ```
  /usr/class/cs155/bin/submit proj2_part2
  ```

- Follow the on-screen instructions
- You can check whether we have received your submission by going into /usr/class/cs155/submissions/proj2_part2/

Useful Rails Tricks

From the bitbar/ directory, run "rails console" to open a Ruby console with access to all of the classes and objects in the Rails project.

To "printf debug" a Rails project, use the "puts" command. Output will appear in the terminal window running your server. For example:

```
puts "Hello, world!"
```

To reset your database and start fresh, run the following commands from the bitbar/ directory:

```
rake db:reset
rake db:seed
```

To see how URLs are mapped to controllers and methods, look at the file /config/routes.rb.

Rails uses a lot of files and directories, but most of the interesting ones are under the app/ subdirectory. Here's what each subdirectory contains:

- assets: static files used by the site: images, stylesheets, etc.
- controllers: contain the code that runs to handle a request to a particular URL. Each controller method performs whatever processing is needed, then makes a call to the render function to generate a view (the HTML returned in the HTTP response). Variables prefaced with an @ in the controller are automatically available in the view.
- helpers: helper functions used by views to generate HTML.
- models: define objects used to interact with the database.

- views: generate the HTML that is returned. Each file is a ".html.erb" file (HTML with embedded Ruby). Normal HTML is output as-is. The tags <% %> and <%= %> interpret their contents as Ruby code (the later tag outputs the result into the HTML being generated). There is one .html.erb file for each controller method, and the Ruby code in each file has access to any variables prefaced with a @ in the corresponding controller method. The view in layouts/application.html.erb is special - it is automatically output as part of every HTML page generated.