CE441: Data and Network Security **Fall 2020**

Project #2*

Due: Part 1: Monday, Aban 26 - 11:59pm,
Part 2: Saturday, Azar 8 - 11:59pm.

1 Introduction

In this project, you will construct several attacks against a web application (Part 1), and then update the application to defend against those attacks (Part 2). You will specifically be attacking Bitbar, a Node.js web app that lets users manage Bitbars, a new ultra-safe cryptocurrency. Each user is given 100 Bitbars when they register for the site. They can transfer Bitbars to other users using the web interface, as well as create and view other user profiles. You have been given the source code for the Bitbar application. Real attackers generally do not have access to the source of a target website, but the source may make finding the vulnerabilities a bit easier. Bitbar is powered by a collection of Node packages, including the Express.js web application framework, a SQLite database, and EJS for HTML templating. The list of resources in the next section includes links for more information on these packages as well as other information that you can use as a reference.

2 Setup Instructions

You will run the Bitbar application in a provided Docker container. When the server is running, the site is accessible at `http://localhost:3000`.

Browser

We will grade using the latest version of Mozilla Firefox, and we strongly recommend you test your attacks in Firefox. Chrome has introduced aggressive browser-side XSS guards, which may make some attacks unfeasible if you use Chrome. Other browsers may lack the protections Firefox has, making your defenses fail when we grade them.

* Acknowledgment: This project is obtained from CS155, Spring 2020, Stanford University.
Edited By H. Ahmadzadeh & A. Ehteshami.

Detailed setup instructions:

Your web server will run in a Docker container. The following instructions will walk you through installing Docker, and the container.

1. Install (and run) Docker Community Edition on your local machine [1]

   https://store.docker.com/search?type=edition&offering=community

2. Pull the Project 2 starter code from the CE441's tarasht repository.

3. Navigate to the starter code root directory and run `bash build image.sh`[2]. This builds your Docker image and installs all necessary packages. This may take a couple of minutes, depending on your internet speed. A successful build should end in the line Successfully `tagged ce441-proj2-image:latest`

4. To start the server, run bash start server.sh. Once you see
   `$ ./node modules/babel-cli/bin/babel-node.js ./bin/www`,
   the Bitbar application should be available in your browser at
   http://localhost:3000. [3]

   You can close the server by pressing Ctrl+C in the terminal. **The server will completely reset every time that you shut it down.** To restart the server with a clean database, just run `bash start server.sh` once again.

Docker tips:

You don't need to familiarize yourself with Docker in order to complete this assignment. However, a few tips that may prove useful:

- `docker ps -a` lists all of your containers.
- `docker images` lists your images.
- `docker system prune -a` deletes unused images and containers from your machine. (Do this when you're done with the assignment if you want to save space!)

[1] If you are running macOS, Linux, or Windows Enterprise, this should be very straightforward; just pick your operating system and download away. However, Docker for Windows does not support Windows Home. You may have to download Docker Toolbox (https://docs.docker.com/toolbox/toolboxinstallwindows/), which installs Docker in a VM.

[2] If you're running Docker Toolbox, you should run the shell scripts from the Docker Quickstart Terminal. When navigating to the directory in the terminal, make sure that your directories are capitalized correctly (except the "/c" at the beginning).

[3] If you're running Docker Toolbox, it may be available at your VM's broadcasted IP address instead.

- The scripts `build image` and `start` server are simply one-line Docker commands to build a Docker image and spin up a temporary container from that image.
- The only file that is mapped from your local machine to the running Docker container is `code/router.js`. So if you start modifying other files and the modifications aren't showing up, don't worry. You may have to restart your container after modifying `code/router.js` for changes to take effect. If you decide you must modify another file for some reason, you must rebuild the Docker image to copy your changes into the image.
- The docs: https://docs.docker.com/

3 Part 1: Attacks

For the first part of the project, you will develop a series of attacks against the Bitbar application. In each exercise, we describe what inputs you will need to provide to the grader, and what specific actions the grader will take using your input. The grader should obtain the result described in each exercise for you to receive credit. All of your attacks should assume that the site is accessible at the URL `http://localhost:3000`.

You may not use any external libraries nor may you edit the web app itself. In particular, this means you cannot use jQuery. You may use online resources, but please cite them in in your submission's `README.txt`.

Exploit Alpha: Cookie Theft

In the first attack, your goal is to steal the logged in user's Bitbar session cookie and send it to an attacker controlled URL. You need to create URL starting with: `http://localhost:3000/profile?username=` that sends the stolen cookie to `http://localhost:3000/steal_cookie?cookie=[stolen cookie here]` when visited. When the attack is successful, the server will log the stolen cookie to the terminal output.

Important! The attack should not be visibly obvious to the user. This means there should no changes to the site's appearance and no extraneous text should be visible. Except for the browser location bar (which can be different), the grader should see a page that looks normal when the grader visits their profile. Avoiding the blue warning text stating that a user is not found is an important part of the attack. It is fine if the number of Bitbars displayed or the contents of the profile are not correct (so long as they look "normal"). It's also fine if the page looks weird briefly before correcting itself.

Deliverable and Grading. You need to submit a file named a.txt that contains only your malicious URL. The grader will be logged in to Bitbar as user1 and will

be on the Profile tab. From here, the grader will copy your URL on the address bar and navigate to it. You can verify that your solution works by looking for the stolen cookie in your terminal output.

Hint: Try adding random text to the end of the URL. How does this change the HTML of the page?

Exploit Bravo: Cross-Site Request Forgery

In the second attack, you will construct a Cross Site Request Forgery (CSRF) attack that steals Bitbar from another user. You will specifically build a malicious website, which, when visited by the victim, steals 10 Bitbars from their account and deposit's them into your attacker account.

Your submitted attack is a self-contained HTML page (`b.html`) that transfers 10 Bitbars from the grader's logged in account to the user `attacker`. As soon as the transfer is complete, your attack site should immediately redirect the user to http://sharif.edu/~kharrazi/courses/40441-991/. This should happen fast enough that normal users won't notice.

Important! The location bar of the browser should never contain localhost:3000 at any point, as this might tip off the victim to the attack.

Deliverable and Grading. You need to submit a single self-contained HTML file `b.html` that contains your exploit. The grader will be logged in to Bitbar before loading `b.html` on a web browser. The grader will check that (1) 10 Bitbars are transferred out of their account to the attacker, (2) the attacker site immediately redirects to the CE441 website, and (3) the web browser never directly visits localhost:3000.

Exploit Charlie: Session Hijacking with Cookies

In the third attack, you need to trick the Bitbar application into thinking you're logged in as a different user by hijacking the victim's session cookie. At the start of the attack, you will be logged in as `attacker` and you need to convince Bitbar into believing that you are `user1` instead of `attacker` so that you can transfer user1's Bitbar into your attack.

Your solution will be a Javscript file (`c.txt`) that can be copy/pasted into your browser's Javscript console. After executing your Javscript in the Console, Bitbar should show that `user1` is logged in, instead of the attacker account, and you should be able to transfer 10 Bitbar from `user1` to `attacker` in the web UI.

The password for the user `attacker` is `evil`. User1's ID is 1.

Deliverable and Grading. You must submit a file c.txt containing the JavaScript to be executed in the Javascript console. You can assume that the grader will run your

attack while the database has the original user1 1 with 200 Bitbars. After running this JavaScript and refreshing the page, the application must be logged in as user1 and must allow the grader to transfer Bitbar into the attacker account.

Hint: How does the site store its sessions?

Exploit Delta: Cooking the Books with Cookies

In this attack, you need to forge 1 million new Bitbar rather than stealing them from other users. You specifically need to develop a Javascript exploit that results in your account balance being bumped to 1 million Bitbar after completing any small transaction (e.g., sending 1 Bitbar to `user1`).

Begin this attack by creating a new user. You have big plans for this account, so a starting balance of 100 Bitbars is not sufficient for your intentions.

Similar to Attack Charlie, your solution is Javscript code (`d.txt`) that can be copied and pasted into your browser's JavaScript console when logged into your new account. After pasting this code into the console, conducting a small transaction should bump your account balance to 1 million Bitbars.

Important! The new balance must persist between sessions. After logging out and back into the account, the balance should be 1 million Bitbars. The goal here is to forge 1 million Bitbars out of thin air without affecting any other users. The transaction should look completely valid to the innocent recipient.

Deliverable and Grading. You will submit a text file `d.txt` containing your exploit code. The grader will create a new account, paste this code into the browser console, send 1 Bitbar to another user, and verify that the new account contains 1 million Bitbars.

Exploit Echo: SQL Injection

In this attack, you need to develop a malicious username that executes malicious SQL against the backend database that powers the Bitbar application.

The grader will create a new user account with your provided username, click on "Close" and then confirm that they want 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 may assume that all other usernames in the database besides the one you create are "non-malicious". More specifically, you can assume they will not contain spaces.

Deliverable and Grading. A text file `e.txt` containing your malicious username. The grader will Close the account, then verify that the malicious account and `user3` have been removed from the database.

Tip: If you mess up the user database while working on the problem, simply kill (`ctrl-C`) the Docker container and restart the server to reset the database.

Exploit Foxtrot: Profile Worm

In the next attack, you need to develop a Worm, similar to the Samy Worm, which steals Bitbar and spreads to other accounts. You specifically need to construct a profile that when visited transfers 1 Bitbar from the logged-in user to `attacker` and replaces the profile of the current user with itself.

If the `attacker` user changes their profile to whatever you provide in your solution, the following should happen:

1. When `user1` views `attacker`'s profile, 1 Bitbar will be transferred from `user1` to `attacker`, and `user1`'s profile will be replaced with your solution profile.
2. Later, if `user2` views `user1`'s profile, 1 Bitbar will be transferred from `user2` to `attacker`, and `user2`'s profile will be replaced as well, and so on.

When viewing an infected profile, the number of Bitbars should appear to be 10, regardless of the corresponding user's true bitbar balance. This also applies to the attacker. Some tips as to how to display the Bitbars for infected profiles and for building your attack:

- There is no problem if infected profiles display 10 Bitbars immediately instead of counting up to 10.
- There is no problem if the number of Bitbars the worm displays for infected users counts up to 10.
- There IS a problem if, for example, the count is first set to 100, and then set to 10.
- There is no problem if a newly-infected user only sees the exploit text in their profile after they have logged out and logged back in again.
- There is no problem if the exploit is triggered when the attacker sees their own infected profile.

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.

Deliverable and Grading. A file named `f.txt` containing your malicious profile. We will copy and paste your profile text into attacker's profile and view that profile using the grader's victim account. We will then view the victim's profile profile with more accounts, checking for the transfer and replication. You will not be graded on the corner case where the user has no Bitbar in their account.

Exploit Gamma: Password Extraction via Timing Attack

A timing attack is a side-channel attack where an attacker attempts to extract data by analyzing the time that a system takes to execute an action. For example, a web server may take longer to respond to a login request that contains a valid password, compared to a request using an invalid password. Even if the Same Origin Policy prevents the attacker from directly viewing the HTML response to a login request, the amount of time the server takes to respond may leak whether the provided password was correct or incorrect.

In the last exploit, you will develop an attack that determines the password of another user by exploiting such a timing side-channel. You will specifically find the victim password by analyzing the amount of time it takes for the Bitbar login page to respond to a correct password versus incorrect passwords. This enables an attacking web site to mount a dictionary attack on a victim web site, by leveraging web browsers that visit the attacking site. The attacking site never connects directly to the victim site; it makes visiting browsers do all the work.

You need to construct a malicious username, which consists of a script that guesses the password of userx by testing the passwords in a provided dictionary and measuring the server response time for each provided password. Your script needs to analyze server's response times to all the passwords in the provided list, determine the correct password and send it to:

http://localhost:3000/steal_password?password=[password]&timeElapsed=[time elapsed]

You can use the code snippet `proj2/code/gamma_starter.html` as a starting point for your attack. This snippet includes the dictionary of passwords to attempt.

Deliverable and Grading. You should submit a file named `g.txt` containing malicious username script. To grade your attack, we will log in as `attacker`, go to transfer page, enter the malicious username script you specify in your solution into the `username` field, and transfer 10 Bitbars to it.

There is no problem if the grader is logged in as `userx` after performing the attack, and the attack may take a few seconds to fully execute. The grader will not click anywhere or leave the transfer webpage while the attack is in progress. There should not be any visible changes to the website and the blue error message on the transfer page should say "The user does not exist".

Hint: Make sure you use backticks instead of quotes for this attack. Timing side

channels can be subtle.

Race conditions

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.

Part 1 Submission

- Put the deliverable file for each exploit inside a directory called `attacks`.
- Create a `README.txt` to cite the online references you used and to give any special notes to the grader, and place that in the `attacks` directory as well.
- For each part, you'll need to provide a screen recording. In your screen recording, explain your solution step by step and record your voice with the screen. Your screen recording should contain all the tools you have used to solve the problems. Submit only one screen recording for each part. Try to minimize the size of each one. Screen capture's length should be less than 5 minutes. **Do Not push** your screen recordings to your repository in the tarasht. Upload them to the internet and write links in `proj2/attacks/links.txt`. Push `proj2/attacks/links.txt` to your repository.
- Put your files in your private repositoty at tarasht.
- When we pull your repository, we expect to see the following seven files inside '`proj2/attacks`':
  - `README.txt`
  - `links.txt`
  - `a.txt`
  - `b.html`
  - `c.txt`
  - `d.txt`
  - `e.txt`
  - `f.txt`
  - `g.txt`

4 Part 2: Defenses

Now that you understand how insecure the Bitbar web application really is, you will modify the application to defend against the attacks from Part 1 (and you will never make these mistakes in your own web apps!). There might be more than one way of implementing each attack, so think about other possible attack methods – you will need to defend against all of them.

Implementation Tips

General

- Forcing a logout if CSRF or cookie tampering is detected is acceptable, but for other bad inputs, err on the side of displaying an error message.
- It is acceptable to have secret key(s) in the server's process memory as JavaScript variable(s).
- You are not required to defend against cookie replay attacks.
- As we do not require you to bring in TLS, you can assume a cookie will not be stolen by a network attacker.

Alpha Defense

- If you have detected an injected username that does not conform to what you have defined as a "valid" username, you do not need to display that invalid username in the error message.

Beta Defense

- You do not need to defend against the case where an attacker makes a simple GET request to recover a CSRF token intended for a victim user.
- The more short-lived a CSRF token is, the better it is.

Foxtrot Defense

- You should use a CSP-related defense to this attack. In order to do this, you may add lines to `app.js` (but you don't have to).
- We will test your defense with 4-5 test case profiles that exercise some of the most common methods of XSS.

Restrictions:

- **You are only allowed to implement your defenses in `router.js` with the two exceptions: you may also change any of the files in `views/` to add CSRF secret token defenses and modify attributes of `<script>` tags (see below for special restrictions on changes in `views/`), and you may change `app.js` for the foxtrot defense. All other files must remain unchanged.**
- 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.
- Do *not enable the built-in defenses in Express.js.* Express.js comes with a number of built-in defenses that were disabled in Part 1. These built-in defenses must remain 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 these defenses. In particular, that means you cannot:

  1. Use Express.js to change the CORS policies that have been set in `app.js`,
  2. Enact stricter EJS escaping policies in any files inside `views/`,
  3. Add any additional Node packages beyond those provided to you in the starter code.

     Note: You can add CSRF secret tokens to files inside `views/`. You can also add `nonce` attribute to `<script>` tags in `views/`. However, do not modify the `<%-` tags within these files to enact stricter EJS escaping functionality.
- Do *not over-sanitize inputs.* You are allowed to sanitize inputs using default JavaScript functions, but make sure you don't over-sanitize (for example, the profile should still allow the same set of sanitized HTML tags).

We highly encourage use of the functions imported from '`./utils/crypto`' in your defenses. Specifically, consider how the `generateRandomness` and `HMAC` functions can be used to prevent CSRF and cookie tampering, respectively.

Finally, please describe your defenses in a `README.txt` file. A couple sentences for each exploit from Part 1 is sufficient. We will read your `README.txt` and source code on top of testing that the attacks from part 1 are not possible. This means that it is possible for you to receive no credit for a defense even if the relevant attack from part 1 is blocked if your defense is very ad hoc or unsound.

Part 2 Submission

1. Ensure that `router.js, app.js` and the files inside `views/` are the only files you have changed to implement your defenses.

2. In your `README.txt` file you should describe your defenses, cite the online references you used, and leave special notes for the grader.

3. For each part, you'll need to provide a screen recording. In your screen recording, explain your solution step by step and record your voice with the screen. Your screen recording should contain all the tools you have used to solve the problems. Submit only one screen recording for each part. Try to minimize the size of each one. Screen capture's length should be less than 5 minutes. **Do Not push** your screen recordings to your repository in the tarasht. Upload them to the internet and write links in `proj2/code/links.txt`. Push `proj2/code/links.txt` to your repository.

4. Put your files in your private repositoty at tarasht.

5 Resources

- HTML, CSS, JavaScript: http://www.w3schools.com/
- Node.js (Bitbar uses version 9.11.1): https://nodejs.org/dist/latest-v9.x/docs/api/
- Express JS (The web framework that powers Bitbar in app.js): https://expressjs.com/en/4x/api.html
- EJS for HTML Templating (See .ejs files within views/): http://ejs.co/#docs
- XSS: https://cs155.stanford.edu/papers/CSS.pdf, https://www.owasp.org/index.php/XSS Filter Evasion Cheat Sheet
- SQL: http://www.w3schools.com/sql/, https://github.com/kriasoft/node-sqlite (package Bitbar uses)
- Clickjacking Attacks and Defense: http://media.blackhat.com/bh-eu-10/presentations/Stone/BlackHat-EU-2010-Stone-Next-Generation-Clickjacking-slides.pdf
- Timing Attacks: https://crypto.stanford.edu/~dabo/pubs/papers/webtiming.pdf