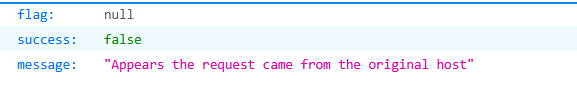
1. **Basic Get CSRF Exercise**

Mục tiêu của bài này là kích hoạt biểu mẫu bên từ nguồn bên ngoài khi đã đăng nhập. Phản hồi sẽ bao gồm một flag (giá trị số).

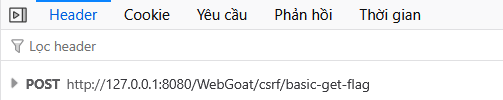
* 1. **Quá trình thực hiện**

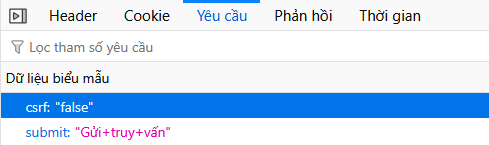
Ta tiến hành ấn submit, sau khi ấn webgoat sẽ đưa ta đến 1 tab <http://127.0.0.1:8080/WebGoat/csrf/basic-get-flag>, nơi đây chứa các thông tin



Lưu ý rằng chúng ta không thể lấy flag ở đây và thông báo success là false, nghĩa là chúng ta chưa giải được bài tập.

Khi ta ấn submit trên giao diện thì đồng thời webgoat sẽ gửi một request với method POST, ở đây khi ta vào devtool (F12), ta thấy được thông tin về URL của request: <http://127.0.0.1:8080/WebGoat/csrf/basic-get-flag>, và payload gồm 1 field csrf





Ở đây khi đọc lại đề bài, ta thấy yêu cầu là kích hoạt biểu mẫu bên từ nguồn bên ngoài, vậy ta sẽ tiến hành tạo 1 file html, và dùng file này để truy cập vào webgoat, sau đó phản hồi sau khi ấn submit sẽ bắt buộc phải bao gồm một flag.

Ta tiến hành tạo file attack.html với action bằng chính url khi nãy để khi truy cập file html này, nó sẽ chuyển hướng ta đến webgoat, bên cạnh đó method là post và field là csrf để đồng bộ với thông tin ta tìm hiểu ban nãy.

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <form action="http://127.0.0.1:8080/WebGoat/csrf/basic-get-flag" method="post">

        <input type="submit" value="submit">

        <input name="csrf" value="false">

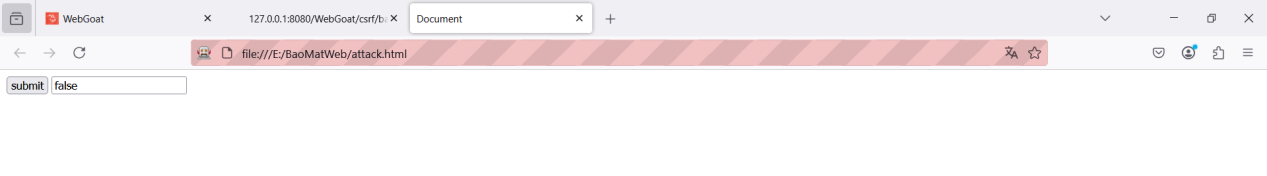
    </form>

</body>

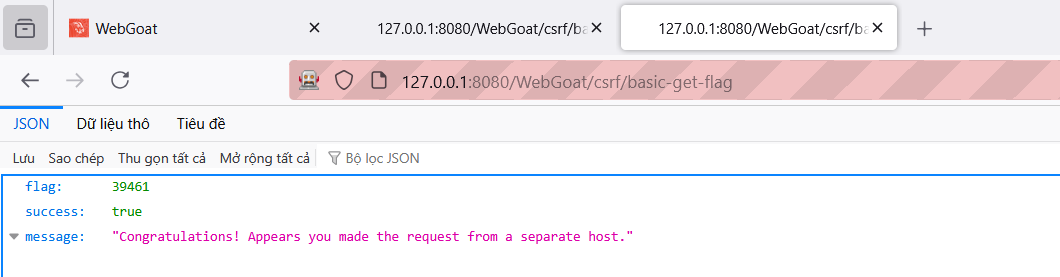
</html>

Đoạn mã dùng để kiểm tra lỗ hổng Cross-Site Request Forgery (CSRF). Khi người dùng mở trang web này và nhấn "Submit", trình duyệt sẽ gửi yêu cầu POST đến WebGoat để kiểm tra xem hệ thống có dễ bị tấn công CSRF hay không, nếu thành công, ta thu được giá trị của flag.

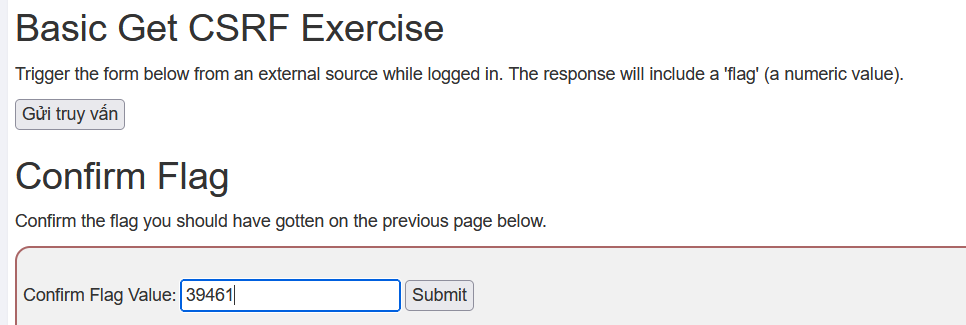
Khi truy cập vào file attack.html, ta sẽ mở ra giao diện webgoat với button submit chính là button ta đặt trong file attack.html

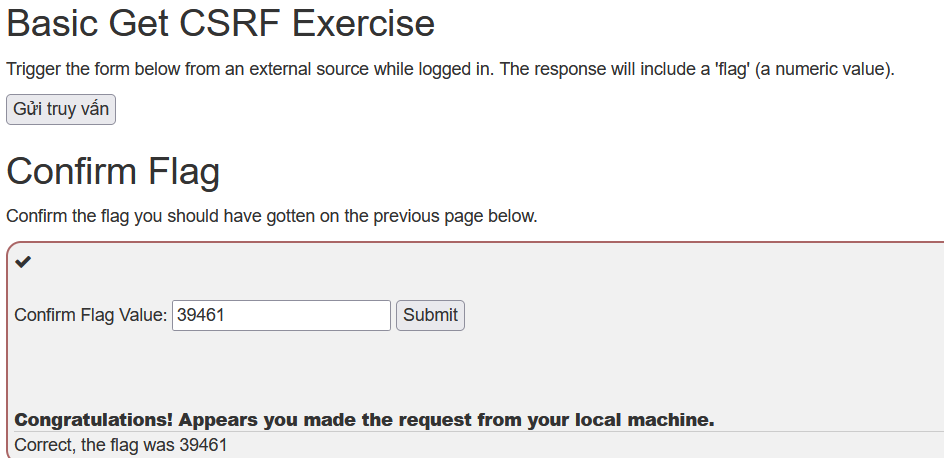


Khi ấn vào button submit, ta thấy được nó trả về giá trị của flag



Tiến hành copy giá trị flag và nhập vào ô trống





* 1. **Nguyên nhân**

Vậy nguyên nhân gây ra lỗi như vậy ở bài này là do đâu? Để tìm ra câu trả lời, ta phải tiên hành vào source code nơi chứa code của bài này để tìm hiểu, ta truy cập url: <https://github.com/WebGoat/WebGoat/blob/main/src/main/java/org/owasp/webgoat/lessons/csrf/CSRFGetFlag.java>

Tại đây ta thấy được đây là một REST API trong WebGoat xử lý yêu cầu POST đến /csrf/basic-get-flag. Ta thấy được 2 tham số referer và host, vậy 2 tham số này có ý nghĩa gì?

Đầu tiên là referer, ta thấy nó được lấy từ

String referer = (req.getHeader("referer") == null) ? "NULL" : req.getHeader("referer");

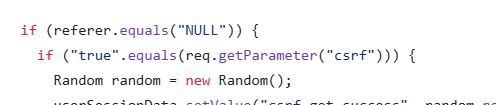
Ta hiểu được referer ở đây là một *HTTP header* chứa URL của trang web gửi yêu cầu, nghĩa là khi ta nhấp vào một liên kết hoặc thực hiện một hành động dẫn đến một yêu cầu HTTP (truy cập tab với url khác), trình duyệt tự động thêm header Referer vào yêu cầu để cho biết nó xuất phát từ đâu. Vậy tóm lại có thể hiểu nó cho biết được nguồn gốc web này được mở từ web nào.

Tiếp theo là host nó tượng trưng cho tên miền của máy chủ nhận yêu cầu.

String host = (req.getHeader("host") == null) ? "NULL" : req.getHeader("host");

Tiếp theo ta tìm hiểu về cách đoạn code kiểm tra Referer và Host:

Trong file CSRFGetFlag.java, code xử lý yêu cầu POST đến /csrf/basic-get-flag, nó kiểm tra header Referer để quyết định có cấp flag hay không.



* **Nếu Referer là "NULL", hệ thống vẫn cấp flag.**
* **Nếu Referer khác Host, hệ thống cũng cấp flag.**
* **Chỉ khi nào Referer chứa cùng một host, hệ thống từ chối yêu cầu.**

Vậy vì sao lỗi CSRF xảy ra?

Lỗ hổng CSRF xuất hiện vì kiểm tra Referer là không đủ để bảo vệ khỏi tấn công. Những điểm yếu trong cách kiểm tra Referer:

* *Trình duyệt có thể không gửi Referer*

→ Một số trình duyệt hoặc cài đặt bảo mật sẽ không gửi Referer, khi đó referer.equals("NULL") và flag vẫn được cấp.

* *Kẻ tấn công có thể kiểm soát Referer*

→ Một số kỹ thuật như Open Redirect hoặc Meta Refresh có thể được sử dụng để giả mạo Referer.

* *Không kiểm tra CSRF Token*

→ Đoạn code này không yêu cầu CSRF Token, nghĩa là bất kỳ ai cũng có thể gửi yêu cầu POST đến /csrf/basic-get-flag và lấy được flag.

* *Chặn sai trường hợp*

→ Nếu Referer khớp với Host, hệ thống từ chối yêu cầu. Nhưng nếu Referer khác Host, hệ thống lại cấp flag, điều này hoàn toàn ngược với nguyên tắc bảo vệ CSRF.

Quay lại bài tập, vậy lỗ hổng CSRF trong bài này đã được ta khai thác như thế nào?

- Đầu tiên ta đã tạo 1 file acctack.html

- Tiếp theo khi ta mở trang web này, trình duyệt tự động gửi yêu cầu POST đến <http://webgoat-instance/csrf/basic-get-flag.>

- Ở đây chắc chắn Referer sẽ khác host do ta mở file html này từ máy local của họ. Kết cục là nhận được flag.

* 1. **Cách khắc phục**

Sử dụng CSRF Token:

* Server phải sinh ra một token duy nhất và gửi nó về client.
* Mọi yêu cầu POST phải đính kèm CSRF Token hợp lệ.
* Server phải kiểm tra token trước khi xử lý request.

Sửa mã nguồn (CSRFGetFlag.java):

@PostMapping(

path = "/csrf/basic-get-flag",

produces = {"application/json"})

@ResponseBody

public Map<String, Object> invoke(HttpServletRequest req) {

Map<String, Object> response = new HashMap<>();

// Lấy CSRF Token từ session của người dùng

String sessionToken = (String) userSessionData.getValue("csrf-token");

String requestToken = req.getParameter("csrf-token");

if (sessionToken == null || requestToken == null || !sessionToken.equals(requestToken)) {

response.put("success", false);

response.put("message", "CSRF token missing or invalid");

return response;

}

// Ngược lại nghĩa là thông tin về token hợp lệ

Random random = new Random();

userSessionData.setValue("csrf-get-success", random.nextInt(65536));

response.put("success", true);

response.put("message", "Request processed successfully with valid CSRF token.");

response.put("flag", userSessionData.getValue("csrf-get-success"));

return response;

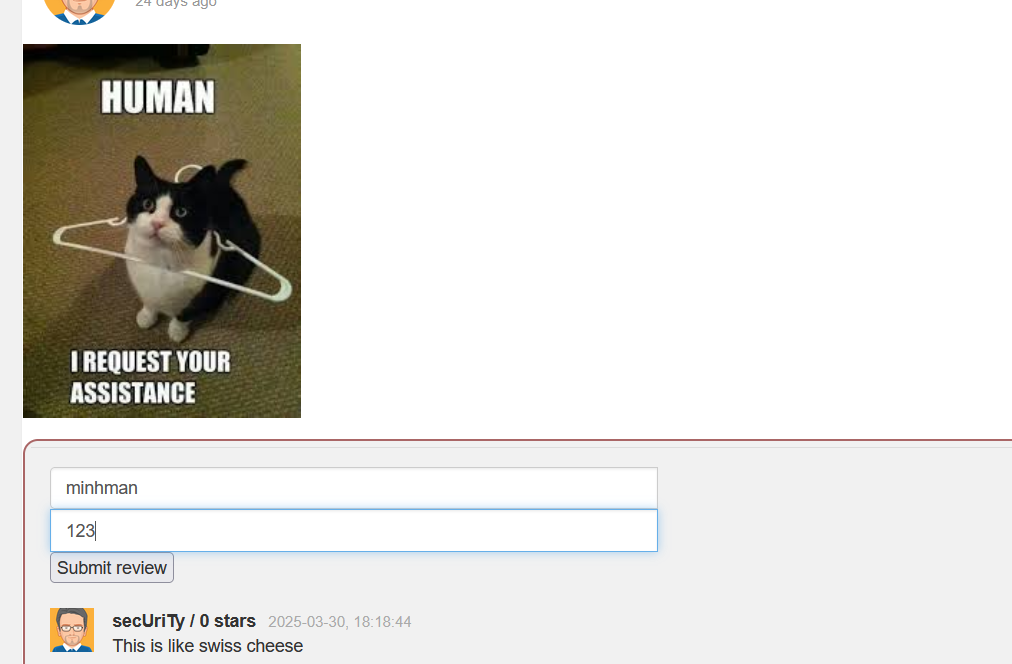
}

1. **Post a review on someone else’s behalf**

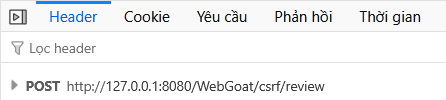
Mục tiêu của bài này là lợi dụng lỗ hổng CSRF để gửi đánh giá (review) mạo danh người dùng hiện tại, cụ thể:

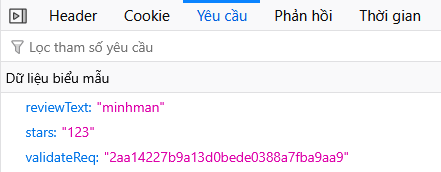
* Gửi một đánh giá (review) mà không cần sự đồng ý của nạn nhân.
* Làm điều này từ một trang web hoặc nguồn khác, chứ không phải trực tiếp từ giao diện hợp pháp.
* Nạn nhân phải đăng nhập vào ứng dụng khi cuộc tấn công diễn ra.
  1. **Quá trình thực hiện**

Ta tiến hành truy cập vào trang bài tập, trang này thường mô phỏng đánh giá và xếp hạng của người dùng trên một bài đăng. Nhiệm vụ của chúng ta là gửi đánh giá và xếp hạng này từ một máy chủ khác giống như chúng ta đã làm trong bài tập trước. Trước hết ta nhập bừa 1 thông tin ngẫu nhiên vào 2 trường nhập liệu trên màn hình sau đó ấn submit.



Khi ta ấn submit review trên giao diện thì đồng thời webgoat sẽ gửi một request với method POST, ở đây khi ta vào devtool (F12), ta thấy được thông tin về URL của request: <http://127.0.0.1:8080/WebGoat/csrf/review>, và payload gồm field reviewText, stars và validateReq.





Tương tự như bài trước, ta cũng tiến hành tạo file attack2.html với action bằng url <http://127.0.0.1:8080/WebGoat/csrf/review> để khi truy cập file html này, nó sẽ chuyển hướng ta đến webgoat nơi ta đang làm bài, bên cạnh đó method là post và field là reviewText, stars và validateReq để đồng bộ với thông tin ta vừa thấy được trong devtool.

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <form action="http://127.0.0.1:8080/WebGoat/csrf/review" method="post">

        <input type="submit" value="submit">

        <input name="reviewText" value="minhman">

        <input name="stars" value="123">

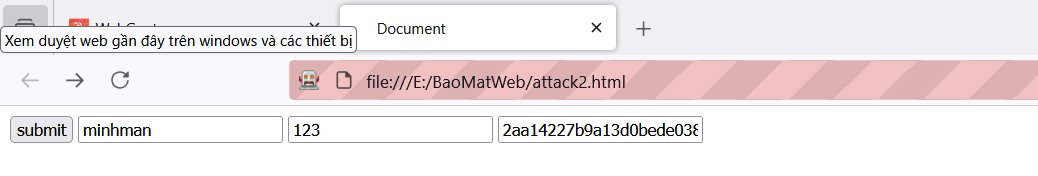
        <input name="validateReq" value="2aa14227b9a13d0bede0388a7fba9aa9">

    </form>

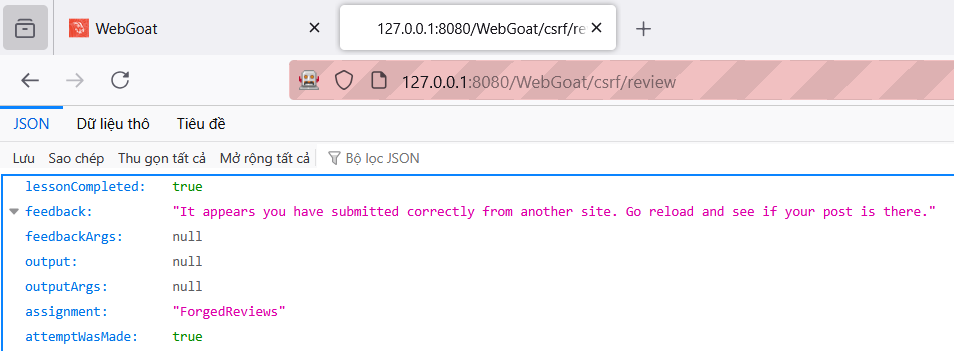
</body>

</html>

Khi truy cập vào file attack.html, ta sẽ mở ra giao diện webgoat với button submit chính là button ta đặt trong file attack.html



Khi ấn vào button submit, ta thấy được nó trả về kết quả



* 1. **Nguyên nhân**

Vậy nguyên nhân gây ra lỗi như vậy ở bài này là do đâu? Để tìm ra câu trả lời, ta phải tiên hành vào source code nơi chứa code của bài này để tìm hiểu, ta truy cập url: <https://github.com/WebGoat/WebGoat/blob/main/src/main/java/org/owasp/webgoat/lessons/csrf/ForgedReviews.java>, ta tiến hành xem xét đoạn code xử lý yêu cầu POST đến /csrf/review trong class ForgedReviews.java.

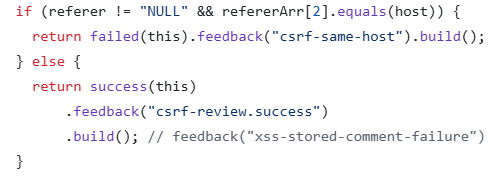
Tại đây, ta thấy có hai tham số referer và host được lấy từ request:

final String host = (request.getHeader("host") == null) ? "NULL" : request.getHeader("host");

final String referer = (request.getHeader("referer") == null) ? "NULL" : request.getHeader("referer");

Tiếp theo ta tìm hiểu về cách đoạn code kiểm tra Referer và Host:

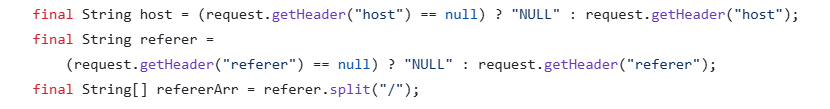
Trong file CSRFGetFlag.java, code xử lý yêu cầu POST đến /csrf/review, nó kiểm tra nếu Referer có tồn tại và trùng với Host thì từ chối yêu cầu. Ngược lại, nếu Referer không trùng với Host hoặc bằng "NULL", hệ thống chấp nhận yêu cầu và đánh dấu thành công.



Vậy vì sao lỗi CSRF xảy ra?

Lỗ hổng CSRF xuất hiện do cách kiểm tra yêu cầu không đủ chặt chẽ. Những điểm yếu trong đoạn code này:

* *Kiểm tra Referer không đáng tin cậy*



Referer có thể bị chặn bởi trình duyệt hoặc bị thay đổi bằng các kỹ thuật như Open Redirect, Meta Refresh.

* *Hệ thống sử dụng một "weakAntiCSRF" cố định*



Giá trị này cố định và không thay đổi theo từng session người dùng.

Kẻ tấn công có thể dễ dàng lấy được giá trị này từ mã nguồn hoặc request trước đó và sử dụng nó để giả mạo yêu cầu.

* *Cho phép request nếu Referer là "NULL"*

→ Một số trình duyệt hoặc cài đặt bảo mật không gửi Referer, khi đó referer == "NULL" và yêu cầu vẫn được chấp nhận.

* *Chấp nhận request nếu Referer khác Host*

→ Nếu yêu cầu đến từ một trang web bên ngoài (Referer khác Host), hệ thống vẫn xử lý và cho phép tạo review, hoàn toàn đi ngược lại nguyên tắc bảo vệ CSRF.

* *Không xác thực danh tính người gửi*

Hệ thống không xác minh xem yêu cầu có thực sự đến từ người dùng hợp lệ hay không.

Chỉ cần biết URL /csrf/review và giá trị weakAntiCSRF, bất kỳ trang web nào cũng có thể gửi yêu cầu POST đến endpoint này mà không cần sự đồng ý của người dùng.

Quay lại bài tập, vậy lỗ hổng CSRF trong bài này đã được khai thác như thế nào?

- Đầu tiên, ta đã tạo một file attack2.html.

- Khi người dùng mở trang này, trình duyệt tự động gửi yêu cầu POST đến <http://webgoat-instance/csrf/review>.

- Trong trường hợp này, Referer sẽ là URL của trang attack2.html, khác với Host của WebGoat. Do cách kiểm tra sai logic, hệ thống chấp nhận yêu cầu giả mạo và lưu review vào database.

Như vậy, ta đã khai thác thành công lỗ hổng CSRF để gửi một review giả mạo mà không cần sự đồng ý của người dùng.

* 1. **Cách khắc phục**

Để khắc phục lỗ hổng CSRF trong bài này, ta cần thay đổi cách kiểm tra yêu cầu POST đến /csrf/review. Dưới đây là các biện pháp cần thực hiện:

* *Loại bỏ "weakAntiCSRF" cố định*

Trong đoạn code hiện tại, giá trị weakAntiCSRF được đặt cố định là "2aa14227b9a13d0bede0388a7fba9aa9". Điều này làm cho nó vô nghĩa, vì kẻ tấn công có thể dễ dàng lấy được giá trị này từ mã nguồn hoặc từ các request trước đó. Một khi giá trị weakAntiCSRF bị lộ, kẻ tấn công có thể sử dụng nó để thực hiện các yêu cầu CSRF giả mạo mà hệ thống vẫn chấp nhận.

Tiến hành thay thế weakAntiCSRF bằng CSRF Token sinh ra ngẫu nhiên, lưu CSRF Token vào session của từng người dùng và yêu cầu mỗi request POST phải có CSRF Token hợp lệ.

@GetMapping("/csrf/get-token")

@ResponseBody

public Map<String, String> getCsrfToken(HttpServletRequest request) {

// Tạo một CSRF Token ngẫu nhiên

String csrfToken = UUID.randomUUID().toString();

// Lưu token vào session của người dùng

request.getSession().setAttribute("csrf-token", csrfToken);

// Trả về token cho client

Map<String, String> response = new HashMap<>();

response.put("csrfToken", csrfToken);

return response;

}

* *Kiểm tra CSRF Token khi xử lý request POST*

Mọi request POST đến /csrf/review đều phải kèm theo CSRF Token và server sẽ kiểm tra nó.

@PostMapping("/csrf/review")

@ResponseBody

public AttackResult createNewReview(

String reviewText,

Integer stars,

String csrfToken, // Nhận CSRF Token từ request

HttpServletRequest request,

@CurrentUsername String username) {

// Lấy CSRF Token từ session của user

String sessionToken = (String) request.getSession().getAttribute("csrf-token");

// Kiểm tra token hợp lệ

if (sessionToken == null || csrfToken == null || !sessionToken.equals(csrfToken)) {

return failed(this).feedback("csrf-token-invalid").build();

}

// Nếu hợp lệ, tiến hành lưu review

Review review = new Review();

review.setText(reviewText);

review.setDateTime(LocalDateTime.now().format(fmt));

review.setUser(username);

review.setStars(stars);

var reviews = userReviews.getOrDefault(username, new ArrayList<>());

reviews.add(review);

userReviews.put(username, reviews);

return success(this).feedback("csrf-review.success").build();

}

* *Cách client gửi CSRF Token*

Lấy CSRF Token trước khi gửi yêu cầu:

fetch('/csrf/get-token')

.then(response => response.json())

.then(data => {

const csrfToken = data.csrfToken;

sendReview(csrfToken);

});

function sendReview(csrfToken) {

fetch('/csrf/review', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify({

reviewText: "This is a secure review",

stars: 5,

csrfToken: csrfToken // Đính kèm CSRF Token

})

});

}

**1. Hacker làm sao biết form gửi comment có những gì?**

Vì form được render trực tiếp trong HTML (frontend), nên hacker chỉ cần mở DevTools > tab Elements/Network là thấy cấu trúc form, endpoint, input fields.

Ngoài ra, hacker còn có thể dùng công cụ như Burp Suite/ZAP để bắt gói tin khi user thật gửi review và sao chép lại.

**2. Nếu không có source code, hacker tạo form từ đâu, bắt thông tin bằng gì?**

✔ Bổ sung tình huống HTTPS/ẩn nội dung:

Nếu website chạy HTTPS, và thông tin không để trong HTML (VD: sinh động bằng JavaScript), hacker vẫn có thể bắt request bằng: cách quan sát request gửi từ browser thật (DevTools > Network tab > chọn request POST review > copy as cURL / Raw request).

→ Dù frontend cố ẩn, hacker vẫn bắt được nếu là người dùng thật (hoặc qua reverse engineering frontend code).

* Nhìn được URL, method, params.
* Xem cookie đang dùng.
* Tái tạo lại request giả mạo.

→ Đây là kiểu “black-box CSRF attack”, không cần biết code nhưng vẫn thành công.

**3. Hacker tấn công WebGoat bằng cách nào?**

Hacker tấn công bằng cách dụ người dùng truy cập một trang web giả mẹo chứa form tấn công. Link đó có thể:

* Đặt trên cùng server như WebGoat (nếu chạy local).
* Gửi qua email, tin nhắn, mạng xã hội… (social engineering).
* Nhúng trong iframe trên trang khác.

Tuy nhiên, CSRF chỉ thành công nếu người dùng đã đăng nhập trước đó, và trình duyệt tự động gửi session cookie.

**4. Làm sao phân biệt victim và hacker trong lab này?**

Victim: người đang login vào WebGoat, nạn nhân có nhu cầu thực hiện các nhu cầu khác bên ngoài mà vô tình ấn vào trang web giả mà hacker chuẩn bị.

Hacker: không cần login, chỉ cần viết file HTML có form trỏ về endpoint của WebGoat.

📍 Nếu dùng Chrome:

Mở DevTools > Application > Storage > Cookies → xem cookie domain WebGoat (thường là JSESSIONID hoặc webgoat.session.id)

Username hiện tại có thể thấy ở giao diện sau khi submit comment, hoặc nhìn vào payload nếu có trường username.

**Câu 5. WebGoat xác thực người dùng như thế nào?**

WebGoat sử dụng Spring Security với cơ chế xác thực dựa trên session (JSESSIONID).

Khi người dùng đăng nhập thành công, Spring Security:

<https://github.com/spring-projects/spring-security/blob/main/web/src/main/java/org/springframework/security/web/authentication/UsernamePasswordAuthenticationFilter.java>

Theo file trên, dòng 74: **Tạo một đối tượng Authentication chứa thông tin người dùng.**

<https://github.com/spring-projects/spring-security/blob/main/web/src/main/java/org/springframework/security/web/authentication/AbstractAuthenticationProcessingFilter.java>

Theo file trên, dòng 342: successfulAuthentication(HttpServletRequest req, HttpServletResponse res, ...): **Gán Authentication vào SecurityContext. Ở dòng 345: context.setAuthentication(authResult);**

<https://github.com/spring-projects/spring-security/blob/main/web/src/main/java/org/springframework/security/web/context/SecurityContextPersistenceFilter.java>

Theo file trên, dòng 120 lấy context đã được gán trước đó (bởi successfulAuthentication()), dòng 122 xóa context khỏi ThreadLocal (dọn dẹp) và 123 lưu context này vào nơi được chỉ định – mặc định là HttpSessionSecurityContextRepository. Có thể xem cụ thể hàm saveContext tại <https://github.com/spring-projects/spring-security/blob/main/web/src/main/java/org/springframework/security/web/context/HttpSessionSecurityContextRepository.java>

Dòng 148 nơi hàm saveContext hoạt động: Lưu SecurityContext vào session (gắn với JSESSIONID), nó gọi hàm **saveContextInHttpSession** và hàm **saveContextInHttpSession** thực thi gọi hàm s**etContextInSession**

Chính đoạn **session.setAttribute("SPRING\_SECURITY\_CONTEXT", context);** dòng 170này tạo session và gán SecurityContext vào session.

Cuối cùng, dòng 158 cũng trong file **HttpSessionSecurityContextRepository.java,** phương thức**: saveContextInHttpSession(...)** với dòng **HttpSession session = request.getSession(createSession);** là nguyên nhân gây ra Set-Cookie: JSESSIONID=… về browser.

-> Spring Security không tự tạo Set-Cookie, nhưng khi gọi request.getSession(true) trong HttpSessionSecurityContextRepository, nếu chưa có session, thì Servlet Container sẽ sinh JSESSIONID và tự động gửi Set-Cookie về browser.

Trình duyệt lưu JSESSIONID:

* Browser giữ JSESSIONID trong cookie.
* Ở mỗi request tiếp theo (kể cả form giả mạo từ attacker), trình duyệt tự động đính kèm JSESSIONID.

Quan sát được ở đâu?

Mở Chrome DevTools → tab Application → Cookies → xem giá trị JSESSIONID.

**6. Làm cách nào WebGoat biết comment là do nạn nhân gửi?**

Phân tích từ source code:

@CurrentUsername String username

Đây là custom annotation của WebGoat, lấy username hiện tại từ session đang đăng nhập,nằm tại: <https://github.com/WebGoat/WebGoat/blob/main/src/main/java/org/owasp/webgoat/container/CurrentUsername.java> nơi đoạn code CurrentUsername hoạt động.

|  |
| --- |
| @AuthenticationPrincipal(expression = "#this.getUsername()")  public @interface CurrentUsername {} |

Annotation này yêu cầu Spring lấy:

|  |
| --- |
| SecurityContextHolder.getContext().getAuthentication().getPrincipal().getUsername() |

Lấy username từ principal hiện tại được Spring Security khôi phục từ session (dựa vào JSESSIONID)

Tức là: trình duyệt tự động đính kèm session cookie, backend sẽ tự lấy username tương ứng từ SecurityContextHolder. Không cần input username từ form, mà sẽ tự động lấy ra từ SecurityContext, nhờ cookie JSESSIONID.

Kết luận:

WebGoat xác định “người gửi review là ai” dựa vào session cookie được gửi kèm request, và ánh xạ cookie này đến @CurrentUsername.

Vì vậy, khi user (nạn nhân) mở trang giả chứa form CSRF, trình duyệt gửi cookie session hợp lệ, dẫn đến review được gán cho user đó.

Điểm yếu: Không có kiểm tra CSRF token thật sự, chỉ check validateReq.equals(weakAntiCSRF) (hardcoded!).➡️ Tóm lại: WebGoat không cần token hay input username – chỉ cần cookie session là biết ai đang login. Đây chính là lý do CSRF thành công.

1. **Sau khi gửi comment, dữ liệu được lưu ở đâu?**

🔎 Phân tích từ code:

private static final Map<String, List<Review>> userReviews = new HashMap<>();

Comment được lưu vào userReviews, là HashMap static trên memory, key là username, value là danh sách Review.

Khi gửi comment:

var reviews = userReviews.getOrDefault(username, new ArrayList<>());

reviews.add(review);

userReviews.put(username, reviews);

→ Dữ liệu được lưu vào RAM của server (JVM), không lưu vào database hay file.

Quan sát comment sau khi gửi:

Bạn có thể bấm F12 > tab Network > XHR khi gửi comment, sẽ thấy response JSON chứa feedback csrf-review.success.

Sau đó WebGoat gọi lại endpoint /csrf/review để hiển thị danh sách comment bao gồm comment mới.

🔎 Tóm lại:

Comment lưu ở: userReviews (trong RAM, dạng static Map<String, List<Review>>)

Comment được gán cho ai: Lấy từ @CurrentUsername → chính là user hiện tại đang đăng nhập dựa theo session cookie.

**8. Cách phòng chống CSRF – bật csrf thôi chưa đủ?**

Đúng, bật CSRF trong SecurityConfig thôi là chưa đủ. Để CSRF protection thực sự hoạt động, ta cần:

Form phải có input hidden field:

<input type="hidden" name="\_csrf" value="...">

Spring phải sinh ra token và gắn vào mỗi form.

Nếu vẫn dùng hardcoded value như "weakanticsrf" thì token là giả tạo, vô dụng -> Dùng CSRF sinh token tự động