|  |  |
| --- | --- |
| **SCR-1.2: Cross-Site Request Forgery** | **No. of Findings: 3** |
| **Description**  The form post at homepage.html line 3 must contain a user-specific secret in order to prevent an attacker from making unauthorized requests.   |  |  |  |  | | --- | --- | --- | --- | | **Source location and affected line number (parent)** | **Line number variable/method affected (child)** | **Risk Rating** | **Variable/method affected** | | Templates/index.html/index.html | 3 | Low | <link rel="stylesheet" href="/static/style.css" type="text/css"> <form action="/login" method="POST"> <div class="login"> <div class="login-screen"> | | Templates/sl.html/sl.html | 3 | Low | <link rel="stylesheet" href="/static/style.css" type="text/css"> <form action="/sl" method="POST"> <div class="login"> <div class="login-screen"> | | Templates/homepage.html/homepage.html | 3 | Low | <link rel="stylesheet" href="/static/style.css" type="text/css"> <form action="/login" method="GET"> <div class="login"> <div class="login-screen"> | | |
| **Implications**  A cross-site request forgery (CSRF) vulnerability occurs when: 1. A Web application uses session cookies. 2. The application acts on an HTTP request without verifying that the request was made with the user's consent. A nonce is a cryptographic random value that is sent with a message to prevent replay attacks. If the request does not contain a nonce that proves its provenance, the code that handles the request is vulnerable to a CSRF attack (unless it does not change the state of the application). This means a Web application that uses session cookies has to take special precautions in order to ensure that an attacker can't trick users into submitting bogus requests. Imagine a Web application that allows administrators to create new accounts by submitting this form: <form method="POST" action="/new\_user" > Name of new user: <input type="text" name="username"> Password for new user: <input type="password" name="user\_passwd"> <input type="submit" name="action" value="Create User"> </form> An attacker might set up a Web site with the following: <form method="POST" action="http://www.example.com/new\_user"> <input type="hidden" name="username" value="hacker"> <input type="hidden" name="user\_passwd" value="hacked"> </form> <script> document.usr\_form.submit(); </script> If an administrator for example.com visits the malicious page while she has an active session on the site, she will unwittingly create an account for the attacker. This is a CSRF attack. It is possible because the application does not have a way to determine the provenance of the request. Any request could be a legitimate action chosen by the user or a faked action set up by an attacker. The attacker does not get to see the Web page that the bogus request generates, so the attack technique is only useful for requests that alter the state of the application. Applications that pass the session identifier in the URL rather than as a cookie do not have CSRF problems because there is no way for the attacker to access the session identifier and include it as part of the bogus request. CSRF is entry number five on the 2007 OWASP Top 10 list. | |
| **Recommendations**  Applications that use session cookies must include some piece of information in every form post that the back-end code can use to validate the provenance of the request. One way to do that is to include a random request identifier or nonce, as follows: RequestBuilder rb = new RequestBuilder(RequestBuilder.POST, "/new\_user"); body = addToPost(body, new\_username); body = addToPost(body, new\_passwd); body = addToPost(body, request\_id); rb.sendRequest(body, new NewAccountCallback(callback)); Then the back-end logic can validate the request identifier before processing the rest of the form data. When possible, the request identifier should be unique to each server request rather than shared across every request for a particular session. As with session identifiers, the harder it is for an attacker to guess the request identifier, the harder it is to conduct a successful CSRF attack. The token should not be easily guessed and it should be protected in the same way that session tokens are protected, such as using SSLv3. Additional mitigation techniques include: Framework protection: Most modern web application frameworks embed CSRF protection and they will automatically include and verify CSRF tokens. Use a Challenge-Response control: Forcing the customer to respond to a challenge sent by the server is a strong defense against CSRF. Some of the challenges that can be used for this purpose are: CAPTCHAs, password re-authentication and one-time tokens. Check HTTP Referer/Origin headers: An attacker won't be able to spoof these headers while performing a CSRF attack. This makes these headers a useful method to prevent CSRF attacks. Double-submit Session Cookie: Sending the session ID Cookie as a hidden form value in addition to the actual session ID Cookie is a good protection against CSRF attacks. The server will check both values and make sure they are identical before processing the rest of the form data. If an attacker submits a form in behalf of a user, he won't be able to modify the session ID cookie value as per the same-origin-policy. Limit Session Lifetime: When accessing protected resources using a CSRF attack, the attack will only be valid as long as the session ID sent as part of the attack is still valid on the server. Limiting the Session lifetime will reduce the probability of a successful attack. The techniques described here can be defeated with XSS attacks. Effective CSRF mitigation includes XSS mitigation techniques. | |
| **Management Comments** | |
| **Follow-up** | |

|  |  |
| --- | --- |
| **SCR-1.2: Privacy Violation: Autocomplete** | **No. of Findings: 1** |
| **Description**  The form in index.html uses autocompletion on line 14, which allows some browsers to retain sensitive information in their history.   |  |  |  |  | | --- | --- | --- | --- | | **Source location and affected line number (parent)** | **Line number variable/method affected (child)** | **Risk Rating** | **Variable/method affected** | | Templates/index.html/index.html | 14 | High | <label class="login-field-icon fui-user" for="login-name"></label></div> <div class="control-group"> <input type="password" class="login-field" value="" placeholder="password" name="password"> <label class="login-field-icon fui-lock" for="login-pass"></label></div> <input type="submit" value="Log in" class="btn btn-primary btn-large btn-block"> | | |
| **Implications**  With autocompletion enabled, some browsers retain user input across sessions, which could allow someone using the computer after the initial user to see information previously submitted. | |
| **Recommendations**  Explicitly disable autocompletion on forms or sensitive inputs. By disabling autocompletion, information previously entered will not be presented back to the user as they type. It will also disable the "remember my password" functionality of most major browsers. Example 1: In an HTML form, disable autocompletion for all input fields by explicitly setting the value of the autocomplete attribute to off on the form tag. <form method="post" autocomplete="off"> Address: <input name="address" /> Password: <input name="password" type="password" /> </form> Example 2: Alternatively, disable autocompletion for specific input fields by explicitly setting the value of the autocomplete attribute to off on the corresponding tags. <form method="post"> Address: <input name="address" /> Password: <input name="password" type="password" autocomplete="off"/> </form> Note that the default value of the autocomplete attributed is on. Therefore do not omit the attribute when dealing with sensitive inputs. | |
| **Management Comments** | |
| **Follow-up** | |