Avoiding Cross Site Request Forgery

- A comprehensive guide -

kiuwan

Contents

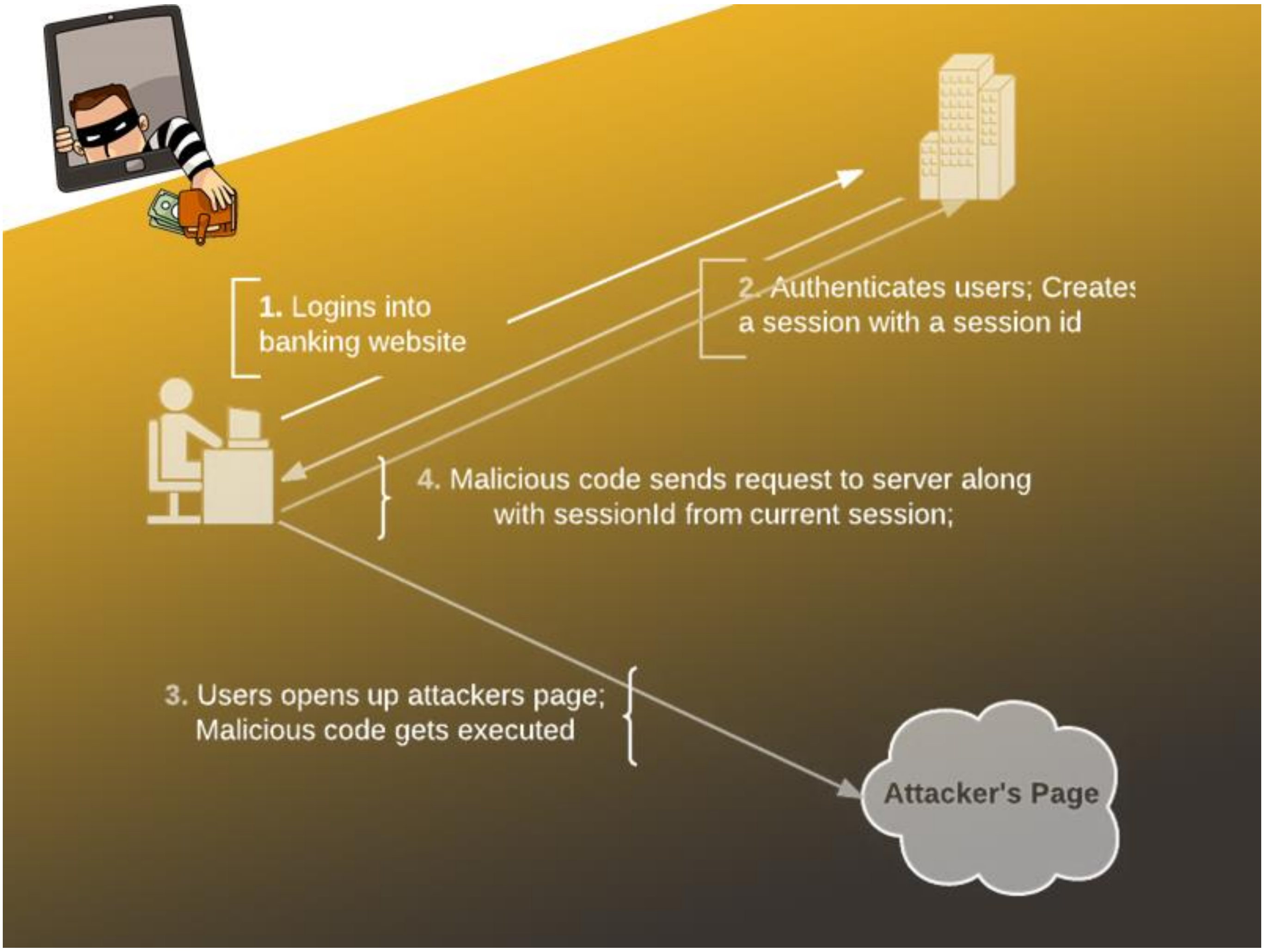
What is CSRF

An example will be helpful.

- Let's imagine that you are the end user, your online bank website is the vulnerable server, and the "unwanted" request is to transfer all your money to the attacker's account.

This is a very basic example of what an attacker could obtain by exploiting a CSRF-vulnerable server.

Now substitute your bank server by any other server you usually access, and only the imagination can limit what could be achieved by an ingenious hacker! Graphically depicted, the sequence would be like something like this:


1. Logins into banking website
2. Authenticates users; Create
a session with a session id
4. Malicious code sends request to server along with sessionId from current session;
3. Users opens up attackers page; Malicious code gets executed
Attacker's Page

CSRF attacks share these common characteristics:

CSRF - Cross Site Request Forgery	
Target	User with an account on a vulnerable server
Attack goal	To make request(s) to the vulnerable server through the user's browser that the server duly performs because it cannot distinguish them from "legitimate" requests
Attacker tools	Attacker's ability to get the user to click a link crafted by the attacker that makes the request to the vulnerable server

How does the attacker get the user to click a malicious link?

The most common way is when the user visits a website controlled by the attacker with a page containing something like:

```
<image src=http://bank.com/transfer.cgi?ammount=9999&to=attacker_account>
```

In this case, the user's browser will automatically visit the URL to obtain what it believes is an image. The request is done!

If the user has a valid session in the bank and the bank server is vulnerable to CSRF the transfer will be done.

This is not the only way (through <image src=...>), the link could also be clicked through a spam email or any other ingenious way.

Obviously, you should consider this example as a simplification, in real life it is much more complex, but what's important is to understand the underlying CSRF mechanism.

How Kiuwan helps you prevent CSRF (CWE-352)

Although CSRF is quite a dangerous security flaw, OWASP's Top 10 (2017) Most Critical Web Application Security Risks does not include CSRF.

Why? Because *most widely used frameworks already provide **built-in defenses** against CSRF attacks*.

> *Therefore, the strongest (and most widely used) CSRF protection mechanism is to rely on the underlying framework's CSRF protection mechanisms.*
>
> Most frameworks provide transparent and easy ways to implement CSRF protection.
>
> Kiuwan is aware of these built-in features, so Kiuwan scans your apps code to "discover" your application's underlying framework and checks if CSRF protection is properly enabled.

To perform such an inspection, Kiuwan scans the code and searches for CSRF vulnerabilities applying *language-specific "rules"*.

These rules are specifically targeted to every programming language and incorporate in-depth knowledge to detect CSRF vulnerabilities.

> In summary, Kiuwan's CSRF-detection rules are based on the following principles:
>
> First, detection of global anti-CSRF protection
> If no global protection is found, Kiuwan checks that state-changing server methods are specifically CSRF-protected

Besides specific CSRF-protection mechanisms, there are *programming practices* that, although they are not CSRF-specific, help to prevent CSRF-attacks.

A sample of good programming practices that can avoid CSRF attacks would be:

- Be sure you app does not contain Cross-Site Scripting (XSS) vulnerabilities Use proper HTTP verbs
- Use secretized links Very same-origin
- Use synchronizer tokens

Kiuwan can also help you in this sense by identifying these kinds of situations, reporting vulnerabilities on risky programming practices.

The examples contained throughout this document are a sample of how Kiuwan checks for CSRF, not a reflection of how it actually performs these checks. The list is partial and is continuously updated.

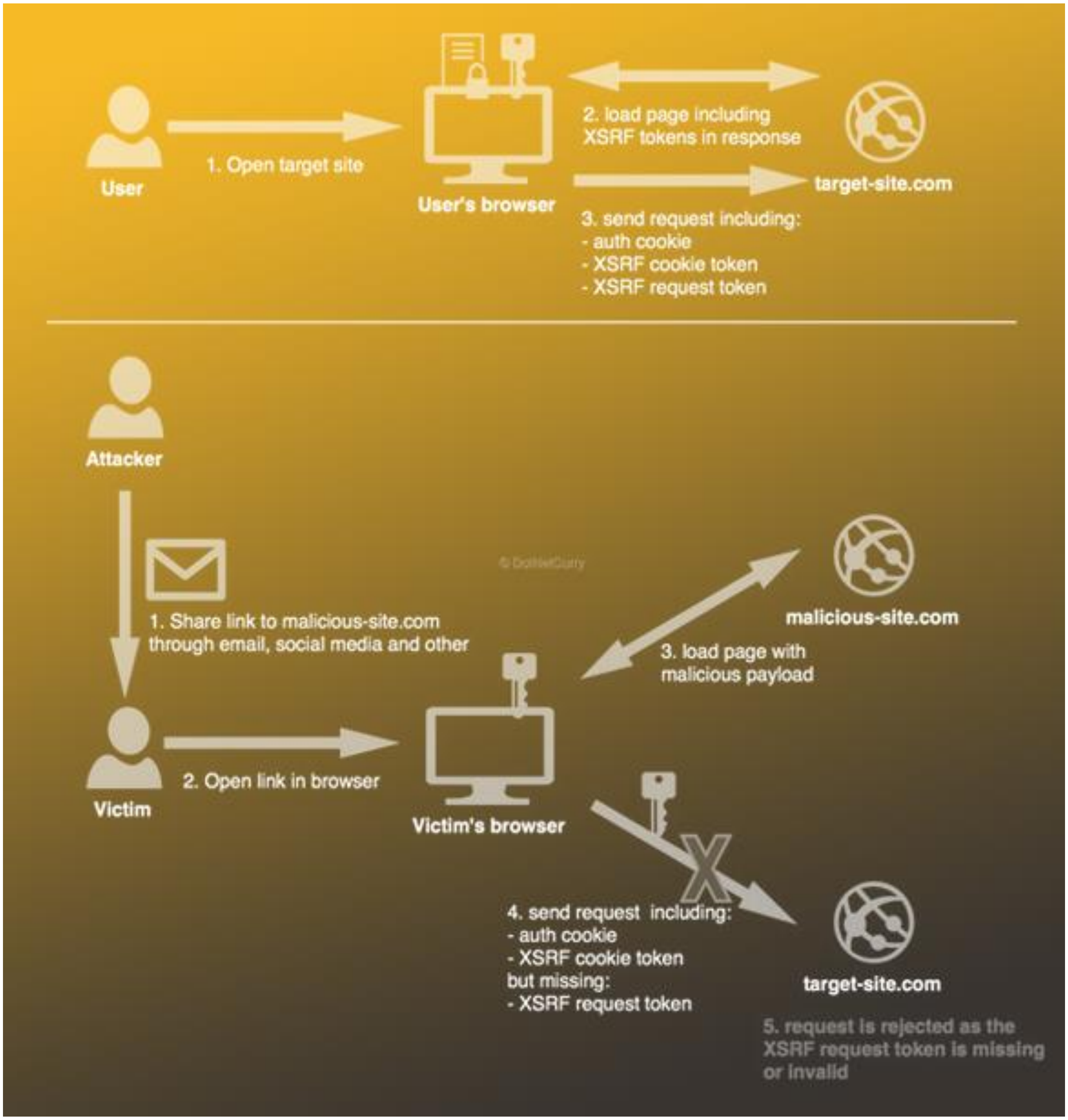
Synchronizer Tokens

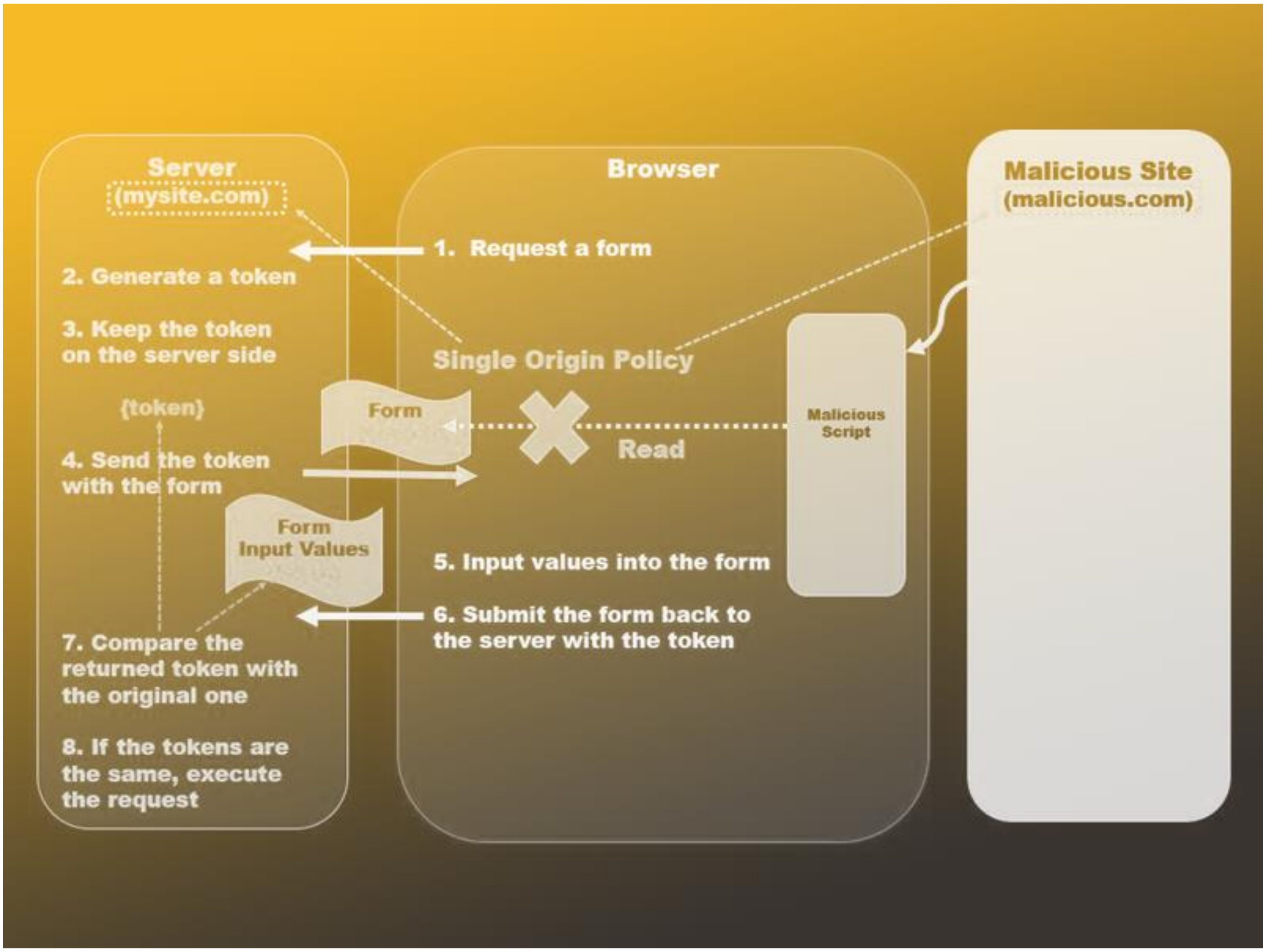
Many frameworks provide built-in CSRF protection mechanisms that help you defend against CSRF attacks quite transparently.

> Frameworks that provide CSRF defense are based on **Synchronizer Token Pattern** (http://www.corej2eepatterns.com/Design/PresoD esign.htm).

- Basically, the approach is *to include a secure random token (CSRF token) within all the state-changing operations* (i.e. those that make modifications), until the session expires.
    - Remember that the CSRF attacker cannot see the results of the request, so a CSRF attack goal is always to change something in the server.
- This CSRF token is added to the request (through a hidden field, or in any other equivalent mechanism) and the server checks that token against the token stored in the session.
    - If both tokens match, go ahead. Otherwise, reject the request and log it.
- When the request is issued by the end-user (through a hidden field, or in any other equivalent mechanism), the server verifies the existence and validity of the token in the request as compared to the token stored in the session.
    - If the token is not found within the request or the value does not match the value stored in the session, then the request should be aborted.

Graphically depicted, the sequence would be like something like this:


User
1. Open target site
User's browser
2. load page including
XSRF tokens in response
target-site.com
3. send request including:
- auth cookie
- XSRF cookie token
- XSRF request token
Attacker
1. Share link to malicious-site.com
through email, social media and other
© DotNetCurry
malicious-site.com
3. load page with
malicious payload
Victim
2. Open link in browser
Victim's browser
X
4. send request including:
- auth cookie
- XSRF cookie token
but missing:
- XSRF request token
target-site.com
5. request is rejected as the
XSRF request token is missing
or invalid


Server
(mysite.com)
2. Generate a token
3. Keep the token on the server side
{token}
4. Send the token with the form
Form
Form
Input Values
7. Compare the returned token with the original one
8. If the tokens are the same, execute the request
Browser
1. Request a form
Single Origin Policy
Read
5. Input values into the form
6. Submit the form back to the server with the token
Malicious Script
Malicious Site
(malicious.com)

Other helpful defense

No XSS

> *Any Cross-Site Scripting (XSS) vulnerability can be used to overcome anti-CSRF defenses.*

This is because an attacker's XSS payload acts as if it is coming from the same origin as the application content, and may read any page content or use XMLHttpRequest to read the CSRF token from the "form" page, and then forge the request to the action URL with that token.

If you need to prevent CSRF vulnerabilities, please be sure there are no XSS vulnerabilities.

Kiuwan provides XSS detection rules for all major languages (Java, JavaScript, C#, PHP, Swift, Python, Objective-C, among others).

Use proper HTTP verbs

Another recommendation to protect against CSRF attacks is not including any sensitive information in an HTTP GET request

HTTP/1.1 RFC 2616 Section 15.1.3 (Encoding Sensitive Information in URI's) states:

> *Authors of services which use the HTTP protocol SHOULD NOT use GET based forms for the submission of sensitive data, because this will cause this data to be encoded in the Request-URI. Many existing servers, proxies, and user agents will log the request URI in some place where it might be visible to third parties. Servers can use POST-based form submission instead.*

The main reason is that including private information in an HTTP GET can cause the information to be leaked.

This recommendation ("not using GET") can be generalized to "use PATCH, POST, PUT and/or DELETE for anything that modifies server state".

As we will see later on this document, Kiuwan also provides checks to detect these types of situations.

Secretized Links

Once you are authenticated in a website, you most probably will receive a unique session ID (typically in a cookie). If the cookie is not expired, the browser will send it automatically along with the request to the server!

> CSRF attacks work because certain types of requests have the same structure minus the session information. But the session information is automatically sent by the browser. This produces a request with a valid session that the server trusts because it cannot recognize that it was not produced by the legitimate user.

Therefore, *a solution would be to include a "secret" in every link/form that the attacker cannot guess* (or takes a huge time to guess): The "secret" can be within a hidden form field, in a custom HTTP header or even can be encoded directly in the URL

-
- Of course, the secret must not be a value that can be guessed
- It could be even the same as the session id that is sent in the cookie (double submit cookie)

Bear in mind that the attacker does not have access to the cookies, so a double submit of the ID (in the cookie) and in the URL is a common way to add a CSRF prevention mechanism.

Verify Same Origin with standard headers

> This CSRF-prevention technique relies on obtaining information about source and target origin of the request, and checking that source and target origins matches. If not, your app might be under attack by CSRF.

SOP (Same origin policy) mechanism restricts access to content loaded from a web site other than current document origin. It is a powerful security feature when preventing attacks through malicious scripts, such as Cross Site Scripting attacks.

In HTML5, SOP feature can be disabled specifying a CORS (Cross-origin resource sharing) policy using the new HTTP header Access-Control-Al low-Origin.

This new header provides flexibility but must be used carefully. *With a overly broad CORS policy, exploitability of security flaws increases notably.*

Kiuwan provides rules for several languages that report vulnerabilities when CORS is excessively broad. You can visit http://lenxwei.blogspot.com.es/2014/08/how-to-mitigate-csrf-using-origin.html for further info.

CSRF-Protection for Java

CSRF-Protection for Java

You can see in the table below existing CSRF-related Kiuwan rules.

Rule code	Name
OPT.JAVA.SEC_JAVA.CrossSiteRequestForgeryRule	Cross-Site Request Forgery (CSRF)
OPT.JAVA.SEC_JAVA.PlaySecurityMisconfiguration	Security misconfiguration in Play framework.
OPT.JAVA.SEC_JAVA.InsecureRandomnessRule	Standard pseudo-random number generators cannot withstand cryptographic attacks
OPT.JAVA.SEC_JAVA.UnsafeCookieRule	Generate server-side cookies with adequate security properties
OPT.JAVA.SEC_JAVA.WebXmlSecurityMisconfigurationsRule	Avoid misconfiguration of security properties in web.xml descriptor

While *CrossSiteRequestForgeryRule* and *PlaySecurityMisconfiguration* are specifically aimed to detect CSRF vulnerabilities, the others are quite helpful to detect conditions that could be used to exploit a CSRF attack.

OPT.JAVA.SEC_JAVA.CrossSiteRequestForgeryRule

The main Java CSRF rule in Kiuwan is named "**Cross-site request forgery (CSRF)**" (code: OPT.JAVA.SEC_JAVA.CrossSiteRequestFor geryRule).

This rule basically works by inspecting your code and checking you are using some defense against CSRF attacks.

As CSRF protection is widely based on mechanisms provided by underlying frameworks, this rule checks that some CSRF protection mechanism is being used AND is properly configured.

This rule detects if you are not using any CSRF protection mechanism, or it's disabled, or it's wrongly configured. In any of these cases, the rule reports such a vulnerability.

Configuration

This rule allows to be configured according to the following parameters:

- checkStateChange
- checkers patterns

checkStateChange

You may set *checkStateChange=true* for reporting as *vulnerable only through those actions that perform state-modification operations*, like file/database writes or changes in session.

Kiuwan contains an internal repository of most-common state-change operations.

When *true*, Kiuwan will only report CSRF vulnerabilities on those operations that call state-change operations, so potentially vulnerable actions not performing any state-change operation are not reported.

When *false*, Kiuwan will check all the operations.

Default value is true, which is adequate for most of occasions (unless you want Kiuwan performs a full scan of all actions), so avoiding false positives.

checkers

OPT.JAVA.SEC_JAVA.CrossSiteRequestForgeryRule checks if your code is using some of the most commonly used CSRF Protection mechanisms..

Below is a partial list of *checked CRSF Protection mechanisms* that Kiuwan performs.

CSRF Protection	References
Spring	https://docs.spring.io/spring-security/site/docs/current/reference/html/csrf.html http://www.codejava.net/frameworks/spring/spring-web-mvc-security-basic-example-part-1-with-xml-configuration http://www.codejava.net/frameworks/spring/spring-web-mvc-security-basic-example-part-2-with-java-based-configuration
OWASP CSRFGuard 3	https://www.owasp.org/index.php/Category:OWASP_CSRFGuard_Project
Tomcat CSRF Prevention Filter	https://tomcat.apache.org/tomcat-7.0-doc/config/filter.html#CSRF_Prevention_Filter https://tomcat.apache.org/tomcat-7.0-doc/api/org/apache/catalina/filters/CsrfPreventionFilter.html https://tomcat.apache.org/tomcat-7.0-doc/config/filter.html#CSRF_Prevention_Filter https://help.sap.com/viewer/65de2977205c403bbc107264b8eccf4b/Cloud/en-US/e5be9994bb571014b575a785961062db.html
OWASP ESAPI	https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API http://www.jtmelton.com/2010/05/16/the-owasp-top-ten-and-esapi-part-6-cross-site-request-forgery-csrf/ https://www.owasp.org/index.php/ESAPI_Secure_Coding_Guideline https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API https://www.owasp.org/index.php/ESAPI_Secure_Coding_Guideline https://static.javadoc.io/org.owasp.esapi/esapi/2.1.0/org/owasp/esapi/HTTPUtilities.html#addCSRFToken(java.lang.String)
Struts 1	https://stackoverflow.com/questions/4303635/cross-site-request-forgery-prevention-using-struts-token
Struts 2	https://stackoverflow.com/questions/22802225/how-to-implement-csr-forgery-prevention-code-on-struts2
Hdiv	https://hdivsecurity.com/docs/csrf/ https://github.com/hdiv/hdiv
JavaServer Faces (JSF) 2	https://stackoverflow.com/questions/26969415/should-protected-views-be-used-for-jsf-2-2-csrf-protection http://arjan-tijms.omnifaces.org/p/jsf-22.html#869

Spring Security

> **Spring Security** helps you to protect applications against CSRF attacks through its built-in Spring Security CRSF Protection mechanism.

Kiuwan checks that you are using Spring by checking in */WEB-INF/web.xml* the existence of a filter such as:

```
<filter>
   <filter-name>springSecurityFilterChain</filter-name>
   <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
```

Spring Security version is extracted from descriptor files according to the version of *spring-security-x.y.xsd* schema definition file.

> By default, *CSRF Protection is enabled since Spring Security 4.0*. But, *for older versions you must explicitly activate it*. Kiuwan checks the Spring version your application is using and apply proper checks in XML configuration files to ensure that CSRF Protection is active, raising CSRF vulnerabilities wherever it founds it's not activated.

Spring Security 4.x

Spring Security 4.x allows to configure CSRF

- protection at two levels: through XML configuration
- through Java configuration

Spring Security 4.x XML CSRF Protection is enabled by default and can be disabled through:

```
<http>
 <!-- ... -->
 <csrf disabled="true"/>
</http>
```

Therefore, Kiuwan checks the existence of explicit disablement of CSRF Protection in the following XML configuration files: the top-level application context configuration file: WEB-INF/spring-security.xml, and servlets configuration files defined at /WEB-INF/web.xml :
servlet configuration file as defined in contextConfigLocation param-name of <init-param>, or
WEB-INF/<servlet-name>-servlet.xml

Any occurrence of a explicit disablement of CSRF will be reported by Kiuwan as a CSRF vulnerability. Spring Security 4.x Java CSRF Protection is enabled by default and can be disabled through:

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

 @Override
 protected void configure(HttpSecurity http) throws Exception { http
   .csrf().disable();
 }
}
```

Spring Security 3.x (or older)

Spring Security 3.x CSRF Protection is NOT enabled by default and can be enabled through:

```
<http>
 <!-- ... -->
 <csrf/>
</http>
```

Therefore, in this case, *Kiuwan checks the existence of explicit enable of CSRF Protection in* *the same XML configuration files*. Any non-existence of explicit CSRF protection will be reported by Kiuwan as a CSRF vulnerability.

OWASP CSRFGuard

> **OWASP CSRFGuard** is a library that implements a variant of the synchronizer token pattern to mitigate the risk of Cross-Site Request Forgery (CSRF) attacks.

The OWASP CSRFGuard library is integrated through the use of a JavaEE Filter and exposes various automated and manual ways to integrate per-session or pseudo-per-request tokens into HTML.

At https://www.owasp.org/index.php/Category:OWASP_CSRFGuard_Project you can get details of configuration and use of OWASP CSRFGuard.

Kiuwan will inspect *web.xml* file to check if CSRFGuard protection is properly configured.

```
<filter>
 <!-- ... -->
 <filter-name>CSRFGuard</filter-name>
 <filter-class>org.owasp.csrfguard.CsrfGuardFilter</filter-class>
</filter>
<filter-mapping>
 <!-- ... -->
 <filter-name>CSRFGuard</filter-name>
 <url-pattern>/*</url-pattern>
</filter-mapping>
```

Tomcat CSRF Prevention Filter

> **Tomcat** provides a **CSRF Prevention Filter** that performs CSRF protection for web applications (https://tomcat.apache.org/tomcat-7.0- doc/config/filter.html#CSRF_Prevention_Filter).

The filter assumes that it is mapped to /* and that all URLs returned to the client are encoded via a call to *HttpServletResponse#encodeRedirectU RL(String)* or *HttpServletResponse#encodeURL(String).*

This filter prevents CSRF by generating a nonce and storing it in the session. URLs are also encoded with the same nonce. When the next request is received the nonce in the request is compared to the nonce in the session and only if they are the same is the request allowed to continue.

The filter class name for the CSRF Prevention Filter is *org.apache.catalina.filters.CsrfPreventionFilter*. Also, Tomcat provides a equivalent filter for REST APIs (*org.apache.catalina.filters.RestCsrfPreventionFilter*).

Kiuwan will inspect *web.xml* file to check if Tomcat protection is configured for both filters.

OWASP Enterprise Security API (ESAPI)

> ESAPI (OWASP Enterprise Security API) is a free, open source, web application security control library that makes it easier for programmers to write lower-risk applications.

The ESAPI libraries are designed to make it easier for programmers to retrofit security into existing applications.

At https://www.owasp.org/index.php/ESAPI_Secure_Coding_Guideline you can see the guidelines to protect against CSRF using ESAPI calls.

AU009	Link and form URLs for all transactions shall be updated with the *HTTPUtilities.addCSRFToken()* method to add a CSRF token.
AU010	All HTTP requests for transactions shall be verified using the *HTTPUtilities.verifyCSRFToken()* method to check that the request is not forged.

Kiuwan scans your code (*web forms* and *web URL preparing controller methods*) to check if the above calls are used and are CSRF-protected.

Struts 1

> The Struts 1 Action token methods will add a token to the session and check it on form submission.

The basic workflow is:

- The user gets to the form through a *Struts Action* (not directly to the JSP). The Struts Action will call *saveToken(request)* before forwarding onto the JSP that contains the form.
- The form on the JSP must use the *<html:form>* tag.
- Your Action that the form submits to will first call *isTokenValid(request, true)*, and you should redirect back to the first Action with an error message if it returns false. This also resets the token for the next request.

Doing this will not only prevent duplicate form submissions but any script will have to hit the first Struts Action and get a session before it can submit to the second Struts Action to submit the form. Since a site can't set a session for another site, this should prevent CSRF.

Kiuwan scans your code to check if *saveToken(req) / isTokenValid(req)* is called in the body of an Action execute method, thus preventing CSRF.

Struts 2

Struts 2 CSRF protection uses token interceptor, which will protect all actions including that interceptor BEFORE the defaultStack.

The "security" stack could be registered as default interceptor, or protect EVERY sensitive action. The interceptor should go before the defaultStack/completeStack or workflow interceptors.

```
<interceptors>
                <interceptor-stack name="defaultSecurityStack">
                  <interceptor-ref name="defaultStack"/>
                  <interceptor-ref    name="tokenSession">
                          <param name="excludeMethods">*</param>
                  </interceptor-ref>
             </interceptor-stack>
  </interceptors>
```

Kiuwan scans your code to check token interceptor is properly configured, thus preventing CSRF.

HDIV

HDIV is a security framework for J2EE, that works with Spring MVC, Thymyleaf, Grails, JSTL, Struts 1 and 2, and JSF.

HDIV prevents CSRF attacks by inserting a random token in links and forms. It uses a filter that provides "transparent security", including anti-CSRF synchronized token.

HDIV adds a random token to each link or form existing within the application. This makes it extremely difficult to implement a CSRF attack as the attacker does not know the token value.

```
<listener>
    <listener-class>org.hdiv.listener.InitListener</listener-class>
</listener>

<!-- Hdiv Validator Filter -->
<filter>
    <filter-name>ValidatorFilter</filter-name>
    <filter-class>org.hdiv.filter.ValidatorFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>ValidatorFilter</filter-name>
    <!-- Spring MVC Servlet name-->
    <servlet-name>SampleMvc</servlet-name>
</filter-mapping>
```

In order to offer an additional security level HDIV does not use a random token per session and creates a new token for each requested page. Even the token used by links and forms within the same page are different avoiding the reuse of link tokens to exploit a web form.

Kiuwan scans your code to check HDIV token filter is properly configured, thus preventing CSRF.

JavaServer Faces (JSF) 2

JSF 2 has some implicit protection against this when state is saved on the server and no stateless views are used, since a post-back must then contain a valid javax.faces.ViewState hidden parameter.

Contrary to earlier versions, this value seems sufficiently random in modern JSF implementations. Do note that stateless views and saving state on the client does not have this implicit protection.

JSF 2.2 introduced an additional/stricter explicit protection against CSRF attacks.

Among others, client state encryption is now on by default and there's a token parameter for protection of non-postback views, stateless views and views with client side state.

Kiuwan looks for *<csrf />* element in *faces-config.xml* (if JSF 2.2+ this is enabled by default), so all postback *<h:form>* pages are safe (they encode a CSRF token automatically), while non-postback pages (i.e. reachable with GET) with sensitive operations should be listed in the *<prote cted-view>* list of patterns.

OPT.JAVA.SEC_JAVA.PlaySecurityMisConfiguration

Play Framework (https://www.playframework.com) is becoming a widely used framework to build web applications with Java.

Play provides several filters that could block/mitigate common attacks in a web application. These filters are configured in the application configuration files (like *application.conf*).
In recent versions, most of those filters are enabled and properly configured by default. In particular, three filters are enabled by default:

- *CSRFFilter*, which provide protection against CSRF (cross-site request forgery) attacks, using a SRF token that enables any sensitive action to execute only if the token is passed from another site resource, which avoids CSRF.
- *SecurityHeadersFilter*, which sets security-related HTTP headers for protection/mitigation of cross-site scripting, clickjacking, MIME sniffing, Adobe Flash cross-domain policy hijacking, Referrer header leakage, and other security issues.
- *AllowedHostsFilter*, which sets a white list of hosts that can access the application. This is useful to prevent cache poisoning attacks. In addition, a *RedirectHttpsFilter* (not enabled by default) could be enabled for redirecting all HTTP requests to HTTPS automatically.

Disabling these filters, or misconfiguring them with incorrect or too-permissive values, might open the avenue for security vulnerabilities. The rule reports any suspicious configuration element.

Specifically for CSRF, this rule looks in the *application.conf* file for a global deactivation of the default anti-CSRF filter, and for each route in routes file, the *nocsrf* modifier is not applied to a state-modification method (POST, PUT, DELETE, PATCH).

OPT.JAVA.SEC_JAVA.InsecureRandomnessRule

Insecure randomness errors occur when a function that can produce predictable values is used as a source of randomness in a security-sensitive context:

- security tokens (like anti-CSRF or password-reset tokens)
- values used in cryptographic operations (session key material, initialization vector in block or stream ciphers) password seeds.

The rule checks that your code is using *cryptographically secure practices to generate tokens*, a necessary condition to any CSRF-protection based on synchronizer tokens.

OPT.JAVA.SEC_JAVA.UnsafeCookieRule

This rule checks that cookies generated in server side are generated with adequate security properties:

- Non persistent - Cookie is not stored persistently by the browser
- HttpOnly - Cookie is not accessible by client-side scripts
- Secure - Cookie is sent in HTTP/SSL communications only
- Path - Path should not match a certain patterns that allow transmission to unintended web applications
- Domain - Domain should not be too wide so the cookie is sent to unintended servers

Although all of them contribute to a more secure cookie management, HttpOnly is specially suited to mitigate some CSRF issues.

HttpOnly makes more difficult for the attacker to capture sensitive cookies from client-side code generated by exploiting an XSS vulnerability in a web application.

OPT.JAVA.SEC_JAVA.WebXmlSecurityMisconfigurationsRule

This rule, although it is not CSRF-specific, implements *several checks for common security misconfigurations* in *web.xml* descriptors.

- No default error pages for 404/500 error codes and for uncaught exceptions. No methods in security constraints.
- Configure SSL for protected areas. Send session ID under SSL only.
- Send session cookies with the HttpOnly flag set.
- Use cookies, not URL rewriting, to exchange session IDs with browsers. Expire sessions with no too large timeouts.

If present, these misconfigurations facilitate CSRF-attacks. Pay attention of this rule to detect any misconfiguration.

CSRF-Protection for JavaScript

CSRF-Protection for JavaScript

You can see in the table below existing CSRF-related Kiuwan rules.

Rule code	Name
OPT.JAVASCRIPT.CrossSiteRequestForgery	Execution of an action on user behalf in a previously authenticated web site (cross-site request forgery, CSRF)
OPT.JAVASCRIPT.UnsafeCookie	Generate server-side cookies with adequate security properties

While *CrossSiteRequestForgeryRule* is specifically aimed to detect CSRF vulnerabilities, *UnsafeCookie* is quite helpful to detect conditions that could be used to exploit a CSRF attack.

> In web sites developed with **JavaScript server-side** frameworks (like ***Node.js***, ***Express*** or ***Koa***), anti-forgery tokens, also known as request verification tokens, should be utilized to prevent CSRF attacks.

OPT.JAVASCRIPT.CrossSiteRequestForgery

As explained before, anti-forgery tokens are random values generated in the server when a form is requested, and they are included in every request, so the server can verify not only that user is authenticated, but that the request was originated from the application.

This rule checks:

- If a code fragment should be protected against CSRF attacks (for example, handling a POST request).
- If such code is protected with one of the recommended anti-CSRF protection schemes.
- If no protection is found for the candidate code, a violation is reported.

Use an anti-CSRF protection module for your framework.

Table below lists a partial list of supported JavaScript engines, frameworks and protection mechanisms that Kiuwan checks for CSRF-protection.

JS Server	Framework	Protection	Reference
Node.js	Express	csurf	https://github.com/expressjs/csurf
		alt-xsrf	https://www.npmjs.com/package/alt-xsrf
	Koa	koa-csrf	https://github.com/koajs/csrf
		stateless-csrf	https://github.com/koajs/stateless-csrf
		koa-atomic-session	https://github.com/koajs/atomic-session
SAP	Hana XS	prevent_xsrf	http://hanaperspective.blogspot.com.es/2017/02/sap-hana-xs-application-access-file.html

Additional references:

- Express : https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs Koa : http://koajs.com/
- Hana XS :
    - http://saphanatutorial.com/sap-hana-xs-sap-hana-extended-application-services/ https://help.sap.com/viewer/400066065a1b46cf91df0ab436404ddc/2.0.02/en-US/a9fc5c220d744180850996e2f5d34d6c.html#loioa9fc5c220d744180850996e2f5d34d6c__section_N101F7_N10016_N10001

OPT.JAVASCRIPT.UnsafeCookie

This rule checks that *cookies generated at server-side (for example, in Node.js) have adequate security properties*.

This rule is analogous to Java rule (see OPT.JAVA.SEC_JAVA.UnsafeCookieRule for details).

CSRF-Protection for JSP

CSRF-Protection for JSP

OPT.JSP.SEC_JSP.FileInclusionVulnerability

JSP technology provides the *<jsp:include>* action (or JSTL *<c:import>*) for including content in the current page, either from a local (web application) resource or from an URL, respectively.

When the page path or URL is formed using untrusted input, an attacker may provide the input in the HTTP request to force the J2EE application server to include an unintended resource, which opens the way to either sensitive local file disclosure (like a /WEB-INF configuration file), or catastrophic remote file injection (e.g. remote attacker-controlled content with embedded malicious JavaScript code for CSRF attacks, or Java code in scriptlet to execute unexpected server-side operations, including operating system commands).

> Kiuwan provides the rule *OPT.JSP.SEC_JSP.FileInclusionVulnerability* that avoids unintended leakage of sensitive local/remote files, or remote file include attacks, in JSP dynamic include actions.

Use the "compile-time" *<%@ include %>* directive, if the included page is local and non dynamic.

If included page should be dynamic, never let untrusted input to directly form part of the page path (for *<jsp:include>*) or page URL (for *<c:import>*
). Better use a request attribute, set in the request processing server-side controller class, where the dynamic page is selected (but untrusted input should not be part of the page path/url anyway).

A "*white-list*" validation scheme (untrusted input may be used only to select from a known list of allowed pages) could be used as well.

CSRF-Protection for Csharp

CSRF-Protection for Csharp

You can see in the table below existing CSRF-related Kiuwan rules.

Rule code	Name
OPT.CSHARP.CrossSiteRequestForgery	Cross-Site Request Forgery (CSRF)
OPT.CSHARP.CrossSiteScripting	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
OPT.CSHARP.StoredCrossSiteScripting	Web content generation from improper sanitized database data and escaped output (Stored Cross-site Scripting, XSS)
OPT.CSHARP.MVCPostInControllers	Restrict allowed HTTP verbs for state-change operations in MVC controllers
OPT.CSHARP.SEC.CrossSiteHistoryManipulation	Cross-Site History Manipulation (XSHM)
OPT.CSHARP.SEC.UnsafeCookieRule	Generate server-side cookies with adequate security properties
OPT.CSHARP.TooMuchOriginsAllowed	Too much allowed origins in HTML5 Access-Control-Allow-Origin header

While *CrossSiteRequestForgeryRule* is specifically aimed to detect CSRF vulnerabilities, the others are quite helpful to detect conditions that could be used to exploit a CSRF attack.

OPT.CSHARP.CrossSiteRequestForgery

> In web sites developed with ASP.Net framework, anti-forgery tokens, also known as request verification tokens, should be utilized to prevent CSRF attacks.

Anti-forgery tokens are random values generated in the server when a form is requested, and they are included in every request, so the server can verify not only that user is authenticated, but that the request was originated from the application.

This rule checks:

1. If a code fragment (an *MVC / Web API controller* or state-changing method, or a *Web Forms page*) should be protected against CSRF attacks.
2. If such code is protected with one of the recommended anti-CSRF protection schemes:
   a. For a *WebForms page*, checks whether *ViewStateUserKey* is set (without disabling *EnableViewStateMac*). Parent pages and master pages are taken into account.
   b. For a *MVC controller* action method, checks for *[ValidateAntiForgeryToken]* attribute (or a call to *AntiForgery.Validate()*).
   c. Alternatives, like common Captcha controls like Google's Recaptcha, are checked for.
3. If no protection is found for the candidate code, a CSRF vulnerability is reported.

Implementation in Web Forms

In *Web forms*, the use of anti-forgery tokens is based on having enabled *EnableViewStateMac* attribute and using *ViewStateUserKey* field to store a unique identifier per session.

EnableViewStateMac attribute is compulsory enabled since version 4.5.2 of .Net framework, released in Sept-2014, and there are security patches (whose installation is highly recommended) for versions from 1.1 to 4.5.2.

If you have a version older than 4.5.2 and no security patched have been applied, it's essential to review that this attribute is not disabled (in the application configuration files or any of its pages).

As to *ViewStateUserKey*, its value can be filled in *Page_Init* method of pages or in your web application's master page, or in your pages' *OnInit* m ethod.

Implementation in ASP .NET MVC and Web API

In *ASP .NET MVC* and *Web API* applications, .NET framework facilitates the creation and validation of anti-forgery tokens.

- To create anti-forgery tokens, you can use *@AntiForgery.GetHtml()* method in *Razor* page or *@Html.AntiForgeryToken()* method in *MVC Views*.
- For validation, you can use *@AntiForgery.Validate()* method or include a *ValidateAntiForgeryToken* attribute in your action or MVC Controller.

If you want to extend the built-in functionality provided by ASP .NET, you can use *IAntiForgeryAdditionalDataProvider* to add additional information to the generated tokens and, subsequently, make the needed validation.

Additional Kiuwan rules and notes on CSRF in Csharp

Generally speaking, *you should pay attention to the following considerations when protecting from CSRF attacks*:

- Any Cross-Site Scripting (XSS) vulnerability can be used to defeat anti-CSRF defenses. This is because an attacker's XSS payload acts as if coming from the same origin as the application content, and may read any page content or use XMLHttpRequest to read the CSRF token from the "form" page, and then forge the request to the action URL with that token. XSS cannot defeat challenge-response defenses, such as Captcha, re-authentication, or one-time passwords.
- Any check on Referrer header as mitigation against CSRF is not practical, as it may be spoofed by attackers under certain circumstances, and browsers may disable it for privacy reasons (link type "noreferrer"). Verifying the Referer header is NOT considered a secure approach for preventing CSRF attacks.
- A previous "generic" verification on the Origin header, when properly implemented, provides a stronger prevention mechanism. But the devil is in the details: old browsers do not send it, and for requests originated from the same site, the browser may not send it. If you trust the browser (blocking old browsers), you may either accept requests with no Origin header as coming from same site, and otherwise check if Origin header is an allowed origin (e.g. the application site, or against Host or X-Forwarded-Host headers).
- Never ever accept HTTP verbs different from POST (or PUT/PATCH/DELETE) for sensitive actions. GET (and HEAD/OPTIONS) should not be allowed.

Besides the specific *OPT.CSHARP.CrossSiteRequestForgery* rule, Kiuwan provides additional rules that can check the above conditions. Pay attention to vulnerabilities found for rules in the following table.

Rule code	Name
OPT.CSHARP.CrossSiteRequestForgery	Cross-Site Request Forgery (CSRF)
OPT.CSHARP.CrossSiteScripting	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
OPT.CSHARP.StoredCrossSiteScripting	Web content generation from improper sanitized database data and escaped output (Stored Cross-site Scripting, XSS)
OPT.CSHARP.MVCPostInControllers	Restrict allowed HTTP verbs for state-change operations in MVC controllers
OPT.CSHARP.SEC.CrossSiteHistoryManipulation	Cross-Site History Manipulation (XSHM)
OPT.CSHARP.SEC.UnsafeCookieRule	Generate server-side cookies with adequate security properties
OPT.CSHARP.TooMuchOriginsAllowed	Too much allowed origins in HTML5 Access-Control-Allow-Origin header

For reference on these rules, please consult Java version of those rules as the mechanism is analogous.

CSRF-Protection for PHP

CSRF-Protection for PHP

You can see in the table below existing CSRF-related Kiuwan rules.

Rule code	Name
OPT.PHP.CrossSiteRequestForgery	Cross-Site Request Forgery (CSRF)
OPT.PHP.AvoidUseDefaultSecret	Avoid using secret value default symfony: ThisTokenIsNotSoSecretChangeIt
OPT.PHP.CrossSiteScripting	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
OPT.PHP.StoredCrossSiteScripting	Improper neutralization of stored data during web content generation (Cross-site Scripting, XSS)
OPT.PHP.SEC.CrossSiteHistoryManipulation	Cross-Site History Manipulation (XSHM)

While *CrossSiteRequestForgery* is specifically aimed to detect CSRF vulnerabilities, the others are quite helpful to detect conditions that could be used to exploit a CSRF attack.

OPT.PHP.CrossSiteRequestForgery

> In web sites developed with PHP, anti-forgery tokens, also known as request verification tokens, should be utilized to prevent CSRF attacks.

Anti-forgery tokens are random values generated in the server when a form is requested, and they are included in every request, so the server can verify not only that user is authenticated, but that the request was originated from the application.

This rule checks:

1. If a code fragment should be protected against CSRF attacks (that is, performs state-change operations, like database changes or file writes).
2. If such code is protected with one of the recommended anti-CSRF protection schemes.
3. If no protection is found for the candidate code, a violation is reported.

Table below is a partial list of supported PHP libraries and protection mechanisms that Kiuwan checks for CSRF-protection.

PHP Library	Protection	Reference
OWASP PHP CSRF Guard		https://www.owasp.org/index.php/PHP_CSRF_Guard
CSRF Magic	csrf-magic.php	https://github.com/ezyang/csrf-magic
CSRF Protector	csrfprotector.php	https://github.com/mebjas/CSRF-Protector-PHP/wiki
CSRF4PHP	CsrfToken	https://github.com/foxbunny/CSRF4PHP/
NoCSRF		https://github.com/BKcore/NoCSRF
Csrf (skookum)		https://github.com/Skookum/csrf/blob/master/classes/csrf.php
Anticsurf		https://code.google.com/archive/p/anticsurf
CSRF Protection (XCMer)		https://github.com/XCMer/csrfprotect
Paragonie Anti-CSRF		https://github.com/paragonie/anti-csrf
EasyCSRF		https://github.com/gilbitron/EasyCSRF
PHP RFC		https://wiki.php.net/rfc/automatic_csrf_protection

Additional Kiuwan rules and notes on CSRF in PHP

Beside the specific OPT.PHP.CrossSiteRequestForgery rule, Kiuwan provides additional rules that can check CSRF-related conditions. Pay attention to vulnerabilities found for rules in the following table.

Rule code	Name
OPT.PHP.AvoidUseDefaultSecret	Avoid using secret value default symfony: ThisTokenIsNotSoSecretChangeIt
OPT.PHP.CrossSiteScripting	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

OPT.PHP.StoredCrossSiteScripting	Improper neutralization of stored data during web content generation (Cross-site Scripting, XSS)
OPT.PHP.SEC.CrossSiteHistoryManipulation	Cross-Site History Manipulation (XSHM)

CSRF-Protection for Python

CSRF-Protection for PHP

You can see in the table below existing CSRF-related Kiuwan rules.

Rule code	Name
OPT.PYTHON.SECURITY.CrossSiteRequestForgery	Cross-Site Request Forgery (CSRF)
OPT.PYTHON.SECURITY.CrossSiteScripting	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
OPT.PYTHON.SECURITY.StoredCrossSiteScripting	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
OPT.PYTHON.SECURITY.UnsafeCookie	Generate server-side cookies with adequate security properties

While *CrossSiteRequestForgery* is specifically aimed to detect CSRF vulnerabilities, the others are quite helpful to detect conditions that could be used to exploit a CSRF attack.

OPT.Python.Security.CrossSiteRequestForgery

In web sites developed in Python with *Django* framework, anti CSRF protection can be enabled for the whole application by including *django.middleware.csrf.CsrfViewMiddleware* module in the *MIDDLEWARE_CLASSES* array within the *settings.py* config file.

The *@csrf_exempt* excludes a certain part from the web application from the CSRF validation, creating a potential security hole into the web application. This is considered a bad programming practice and must be avoided.

This rule checks:

1. If Django is used
    a. The middleware django.middleware.csrf.CsrfViewMiddleware (which is enabled by default) must be kept enabled.
    b. Controllers shouldn't be decorated with @csrf_exempt, as it disabled the csrf
2. If no protection is found (not using Django CSRF-protection or it's disabled), a CSRF vulnerability is reported.

Additional Kiuwan rules and notes on CSRF in Python

Beside the specific OPT.Python.Security.CrossSiteRequestForgery rule, Kiuwan provides additional rules that can check CSRF-related conditions. Pay attention to vulnerabilities found for rules in the following table.

Rule code	Na
OPT.PYTHON.SECURITY.CrossSiteScripting	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
OPT.PYTHON.SECURITY.StoredCrossSiteScripting	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
OPT.PYTHON.SECURITY.UnsafeCookie	Generate server-side cookies with adequate security properties

Thank you!

Try Kiuwan for free
kiuwan.com/free

Contact us:
contact@kiuwan.com
+1 9045123050
Live chat: kiuwan.com

Become a Partner:
partners@kiuwan.com

kiuwan