How can I detect vulnerabilities in my application?

An application can be vulnerable to attack through any component in a production system, such as servers, networks, endpoints, access to related systems, etc. However, most vulnerabilities can be prevented by writing source code which is secure and protected against potential threats. Kiuwan Code Security helps to detect insecure code that could be corrected to prevent a security problem in the future. Let’s review the OWASP TOP TEN and see how Kiuwan rules can identify these vulnerabilities in the coding phase:

**A1 Injection**

Injection flaws, such as SQL, OS, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

In addition, there are several Kiuwan rules to verify that your code is not vulnerable to the most common security problem:

* **Avoid non-neutralized user-controlled input in the LDAP search filters**(OPT.JAVA.SEC\_JAVA.LdapInjectionRule):  Software constructs all or part of an LDAP query using externally-influenced input, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended LDAP query when sent to the LDAP server.
* **Improper Neutralization of Special Elements used in an SQL Command ‘SQL Injection’**(OPT.JAVA.SEC\_JAVA.SqlInjectionRule): The software constructs all or part of an SQL command using externally-controlled input, but it does not neutralize properly that input that could modify the intended SQL command when it is sent to a database interaction method. [Read more here.](https://www.kiuwan.com/blog/what-is-sql-injection/)
* **Improper Neutralization of Special Elements used in an OS Command ‘OS Command Injection’**(OPT.JAVA.SEC\_JAVA.CommandInjectionRule): The software constructs all or part of an operating system command using externally-controlled input, but it does not neutralize properly that input that could modify the intended command when it is sent to a method that executes the command.
* **Improper neutralization of special elements in data query logic ‘NoSQL injection’**(OPT.JAVA.SEC\_JAVA.NoSQLInjection): The application generates a query intended to access or manipulate data in a data store such as a NoSQL database, but it does not neutralize or incorrectly neutralizes special elements that can modify the intended logic of the query.
* **XML Injection (aka Blind XPath Injection)**(OPT.JAVA.SEC\_JAVA.XsltInjection): If the software allows untrusted inputs to control part or all of an XSLT stylesheet, an attacker may change the structure and content of resulting XML. If the resulting XML ends in a browser, the attacker may choose contents to launch cross-site scripting attacks, or execute operations at server level with the victim’s identity allowed by the browser’s same-origin policy (a variant of the cross-site request forgery attack). The attacker may also use this flaw to launch attacks targeted at the server, like fetching content from arbitrary files, running arbitrary Java code, or executing OS commands, when certain XSLT functions are not disabled.

To learn more about the #1 security issue, watch this tutorial:

**A2 Broken authentication**

Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys or session tokens, or to exploit other implementation flaws to assume other users’ identities.

However, Kiuwan provides several rules to verify the correct use of Java sessions and authentication:

* **Trust boundary violation**(OPT.JAVA.SEC\_JAVA.TrustBoundaryViolationRule): The application mixes trusted and untrusted data in the same data structure or structured message. A trust boundary violation occurs when a program blurs the line between what is trusted and what is untrusted. By combining trusted and untrusted data in the same data structure, it becomes easier for programmers to mistakenly trust unvalidated data. This violation may open attack avenues, like injection attacks, when the application takes untrusted data from the trusted domain without validation. (e.g. when untrusted external input, like input HTTP message contents, is set as session or web application attributes, without proper validation / neutralization).
* **Checks that session expiration interval is positive and does not exceed a limit** (OPT.JAVA.SEC\_JAVA.InsufficientSessionExpirationRule): Check that when setting in code a session expiration interval, the interval is not negative or zero (no expiration) and does not exceed a maximum interval. If a negative interval is set, the session never expires.
* **Avoid non-serializable objects stored in session in J2EE applications**(OPT.JAVA.SEC\_JAVA.AvoidJ2EENonSerializableObjectsStored): In order to make the multiple JVMs appear as a single application to the end user, the J2EE container can replicate an HttpSession object across multiple JVMs so that if one JVM becomes unavailable another can step in and take its place without disrupting the flow of the application. For session replication to work, the values the application stores as attributes in the session must implement the serializable interface.
* **Avoid misconfiguring security properties in web.xml descriptor** (OPT.JAVA.SEC\_JAVA.WebXmlSecurityMisconfigurationsRule): Many websites use SSL for authentication, but then either revert back to non-SSL for subsequent communication or have parts of the site that can still be accessed via non-SSL. This leaves the session cookie (i.e. JSESSIONID) vulnerable to session hijacking attacks. To prevent this, session cookies should be created with the “Secure” flag, which ensures that the browser will never transmit the specified cookie over non-SSL.
* **Acegi Misconfiguration – Run-As Authentication Replacement** (OPT.JAVA.SEC\_JAVA.AcegiRunAsAuthenticationReplacementRule): Using the Run-As authentication replacement feature in Acegi can lead to a privilege escalation vulnerability.
* **Avoid using HTTP instead of HTTPS**(OPT.JAVA.JAX.UseSecuredTransportLayer): Communications sent over HTTP (Hypertext Transfer Protocol) protocol are in plain text, so they are fully exposed to attacks such as man in the middle. This situation is not acceptable when sensitive information is being communicated.

**A3 Sensitive data exposure**

Many web applications do not properly protect sensitive data, such as credit cards, tax IDs and authentication credentials. Therefore, attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser.

Consequently, to protect sensitive data exposure, you must use secure protocols, like HTTPS. Kiuwan can discover misconfigurations with rules like:

* **Use SOAP messages authentication** **and encrypted SOAP messages** (OPT.JAVA.JAX.UseAuthenticatedSOAPMessages / OPT.JAVA.JAX.UseEncryptedSOAPMessages): SOAP messages are a way to exchange messages between two endpoints. They contain important data that should not be accessed or modified by a malicious attacker.
* **A hardcoded salt can compromise system security** (OPT.JAVA.SEC\_JAVA.HardcodedSaltRule): It is never a good idea to hardcode a salt. Not only does a hardcoded salt allow all of the project’s developers to view the salt, it also makes fixing the problem extremely difficult. Once the code is in production, the salt cannot be easily changed. If attackers know the salt value, they can precompute rainbow tables for the application, and more easily reverse hashed values.
* **Inadequate padding** (OPT.JAVA.SEC\_JAVA.InadequatePaddingRule): When used in practice, encryption algorithms are generally combined with some padding scheme. The goal of the padding scheme is to prevent a number of attacks that potentially work against the encryption algorithm without padding.
* **Weak cryptography, insufficient key length**(OPT.JAVA.SEC\_JAVA.InsufficientKeySizeRule): A robust cryptographic algorithm (like RSA or AES) with a small key length could be vulnerable to brute-force type attacks.
* **Weak cryptographic hash**(OPT.JAVA.SEC\_JAVA.WeakCryptographicHashRule): Weak cryptographic hashes cannot guarantee data integrity and should not be used in security-critical contexts. MD5 and SHA-1 are popular cryptographic hash algorithms often used to verify the integrity of messages and other data. Recent advances in cryptanalysis have discovered weaknesses in both algorithms. Consequently, MD5 and SHA-1 should no longer be used to verify data integrity in security-critical contexts.
* **Avoid hard-coded or in-comment passwords in code or configuration files**(OPT.JAVA.SEC\_JAVA.PasswordInCommentRule – OPT.JAVA.SEC\_JAVA.PasswordInConfigurationFile – OPT.JAVA.SEC\_JAVA.HardcodedUsernamePassword): Hardcoded passwords can compromise system security in a way that cannot be easily remedied. It is never a good idea to hardcode a password. Not only does hardcoding a password allow all of the project’s developers to view the password, it also makes fixing the problem extremely difficult. Once the code is in production, the password cannot be changed without patching the software.
* **Weak symmetric encryption algorithm**(OPT.JAVA.SEC\_JAVA.WeakEncryptionRule): Old encryption algorithms such as DES no longer provide sufficient protection for use with sensitive data. Encryption algorithms rely on key size as one of the primary mechanisms to ensure cryptographic strength. Cryptographic strength is often measured by the time and computational power needed to generate a valid key. Advances in computing power have made it possible to obtain small encryption keys in a reasonable amount of time.

**A4 XML external entities (XXE)**

Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.

Kiuwan helps with this rule:

* **XML entity injection**  (OPT.JAVA.SEC\_JAVA.XmlEntityInjectionRule): In XML parsers, when XML document to parse could be altered by untrusted input, and the untrusted input may alter the Document Type Definition (DTD, embedded or external), the parser should be configured to avoid two kinds of potential attacks:
  + XML entity expansion DoS (XML bomb or “billion laughs attack”), CWE-776, where well-formed DTD with internal entities are expanded exponentially until they exhaust memory.
  + XML external entity injection (XXE), CWE-611, where an external entity controlled by the attacker refers to the URL of a resource that gives unauthorized access to sensitive files on the server machine, or to a resource like /dev/random in Unix systems that, when entity is expanded, leads to a denial-of-service condition. A variant of this attack is the server-side request forgery (SSRF), where the URL for external DTD or XML-Schema is processed by the URL handler used by the XML parser, allowing the attacker access to an unexpected network resource (for cross-site request forgery attacks or port-scanning enumeration).

Please note that Java parsers are particularly vulnerable to XML entity injection attacks because their default settings have both entity expansion and external entity resolution enabled. Disabling validations does not help, as entity expansion / external entity loading is done even when validation is disabled.

DTD is not the unique element that an attacker could use for injection: XML-Schema attributes could be specified on any tag, so SSRF injection may happen at any place in the XML document, not only at the start where DTD should be placed, but fortunately common Java XML parsers do not seem to fetch schema URLs by default.

XML parsing could be hidden in other frameworks and libraries. For example, with REST, when a resource has an XML representation, the REST framework typically use an underlying XML parser for processing the request message, which obviously is fully under control of a potential attacker.

**A5 Broken access control**

Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as viewing sensitive files, modifying other users’ data, changing access rights, etc.

Application code should protect unauthorized data access. Kiuwan helps with these rules:

* **Java access restriction subverted (Reflection)**(OPT.JAVA.SEC\_JAVA.AccessibilitySubversionRule): Java imposes access restrictions for code in one class to access fields, methods and constructors in objects of different classes (the access level specifiers with public, protected and private keywords enforce in code such restrictions). Using reflection, Java allows a programmer to get around the access control checks provided by these specifiers.
* **Access Control – Anonymous LDAP Bind** (OPT.JAVA.SEC\_JAVA.AnonymousLdapBindRule): Without proper access control, executing an LDAP statement that contains a user-controlled value can allow an attacker to access unauthorized records. Executing LDAP queries under an anonymous bind, effectively without authentication, can allow an attacker to abuse a poorly configured LDAP environment.
* **Dynamic method invocation in Struts 2**(OPT.JAVA.SEC\_JAVA.DynamicMethodInvocation): Struts 2 added a feature (Dynamic Method Invocation, DMI) allowing an Action to expose service methods other than execute() (or other @Action annotated methods). When DMI is enabled, the ! character could be used in the action URL to invoke any public method in the action class. Developers that are not aware of DMI could unexpectedly expose internal business logic to attackers.
* **File disclosure in server-side J2EE forward/include**(OPT.JAVA.SEC\_JAVA.J2eeFileDisclosureRule): Constructing a server-side forward/include with user-controlled input could allow an attacker to view arbitrary files and configuration files, or download application binaries (including application classes or jar files), within protected directories (e.g. /WEB-INF).
* **Avoid non-neutralized user-controlled input composed in a pathname to a resource** (OPT.JAVA.SEC\_JAVA.PathTraversalRule): Software uses external input to construct a pathname that is intended to identify a file or directory that is located underneath a restricted parent directory, but the software does not properly neutralize special elements within the pathname that can cause the pathname to resolve to a location outside the restricted directory.
* **Library loaded from untrusted source**(OPT.JAVA.SEC\_JAVA.ProcessControlRule): Under Java, load() and loadLibrary() methods, in java.lang.System or java.lang.Runtime, load a native library. For load(), the path to the library is specified, while for loadLibrary() the library name is specified and a system-dependent search path is used (for example, java.library.path property, or LD\_LIBRARY\_PATH environment variable). If the argument depends on untrusted input (including in this category system environment or Java system properties), an attacker could place a malicious library with the same name as the loaded library upper in the search path, which may run with elevated privileges.
* **Improper control of resource identifiers (“Resource Injection”)**(OPT.JAVA.SEC\_JAVA.ResourceInjection): Software takes from an untrusted source the identifier for a system resource (like a port number), and allows access to a non-expected resource. Attackers may use the flaw for malicious behavior, for example, a port scan, change software configuration, transmit sensitive information to a third-party server, etc.
* **Server-Side Request Forgery (SSRF)**(OPT.JAVA.SEC\_JAVA.ServerSideRequestForgeryRule): The software builds a URL using untrusted input, and retrieves the contents of this URL (e.g. a webservice), without ensuring that the target (scheme, host, port …) is the intended one.  This is an instance of the ‘confused deputy’ weakness, where the vulnerable application acts as a proxy or relay for connections totally or partially controlled by an attacker. As the connection originates from the host of the vulnerable application, that may hide the attacker’s origin, and bypass network controls.
* **Use of Externally-Controlled Input to Select Classes or Code (‘Unsafe Reflection’)** (OPT.JAVA.SEC\_JAVA.UnsafeReflection): The software uses external input with reflection to select which classes or code to use, but it does not prevent the input from selecting improper classes or code. If the application uses external inputs to determine which class to instantiate or which method to invoke, then an attacker could supply values to select unexpected classes or methods. If this occurs, then the attacker could execute code that was not intended by the developer. It may bypass authentication or access control checks, or otherwise cause unexpected behavior. It the attacker is able to upload files in a location that appears in the classpath, or add new entries to the classpath, this flaw could lead to the execution of arbitrary code, chosen by the attacker.
* **Avoid data submissions to non-editable fields** (OPT.JAVA.SPRING.AvoidDataSubmissionToNonEditableField): Usually, some model fields should not be edited, or at least its modification is not expected, i.e. username, id… Spring MVC module automatically auto binds the HTML form data into the model by using all the available setters, even a naive user doesn’t expect they are going to be used. Thus, a malicious attacker can send an infected request containing a value for some of the non-editable fields. The controller would receive this data and bind it to the model, allowing the attacker to perform malicious, unexpected, activities.

**A6 Security misconfiguration**

Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, and platform. So, secure settings should be defined, implemented and maintained, as defaults are often insecure. Additionally, software should be kept up to date.

Preventing this vulnerability requires paying attention to the source code, but you can also keep in mind:

* **Security misconfiguration in the Play framework.**(OPT.JAVA.SEC\_JAVA.PlaySecurityMisconfiguration): the Play framework 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, many filters are enabled and properly configured by default. Disabling these filters, or misconfiguring them with incorrect or too-permissive values, might open the avenue for security vulnerabilities.
* **CORS policy (Cross-origin resource sharing) too broad**(OPT.JAVA.SEC\_JAVA.TooMuchOriginsAllowedRule): Traditionally, web browsers enforced the Same Origin Policy (SOP) to avoid cross-domain JavaScript to access certain contents in the web page. This policy prevents a malicious script on one page from obtaining access to sensitive data on another web page (“victim”) with a different origin. A strict separation between content from unrelated sites must be maintained on the browser to prevent loss of data confidentiality or integrity. Cross-Origin Resource Sharing (CORS) relaxes the SOP. To do this, essentially two HTTP headers (Origin, emitted by the client, and Access-Control-Allow-Origin, emitted by the server). This allows, for example, to Ajax requests (XMLHttpRequest) to access content from a different origin that should be forbidden by the SOP. With Access-Control-Allow-Origin header, the server defines which other domains are allowed to access content in its domain using cross-origin requests. A too much permissive value (like ‘\*’) will allow a malicious application to communicate with the vulnerable application in unexpected ways, leading to different security problems.

**A7 Cross-site scripting (XSS)**

XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation or escaping. XSS allows attackers to execute scripts in the victim’s browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.

So, most rules of the previous point prevent cross-site scripting problems. But Kiuwan provides more:

* **Improper neutralization of input during web content generation (Cross-site Scripting, XSS)** (OPT.JAVA.SEC\_JAVA.CrossSiteScriptingRule): Software places user-controlled input in page content. An attacker could inject browser script code that is executed in the client browser. The end-user is the attacked subject, and the software is the vehicle for the attack. There are two main kinds of XSS:
  + *Reflected XSS*: The attacker may cause the victim to supply malicious content to a vulnerable web application, which renders HTML content embedding a malicious script executed in the victim’s browser. A variation of this is called DOM-based XSS, where the vulnerable software does not generate content depending on user input but includes script code that uses user-controlled input.
  + *Persisted XSS*: The attacker provides malicious content to a vulnerable application. When other users access the vulnerable pages that embed without proper neutralization, the attacker’s script code is executed in the victim’s browser.

The script executed in the victim’s browser could perform malicious activities. Many browsers could limit the damage via security restrictions (e.g. ‘same origin policy’), but user browsers generally allow scripting languages (e.g. JavaScript) in their browsers (disabling JavaScript severely limits a web site).

* **Same Origin Method Execution (SOME)**(OPT.JAVA.SEC\_JAVA.SameOriginMethodExecution): SOME is a web application attack that abuses callback endpoints by forcing a victim into executing arbitrary scripting methods of any page on the endpoint’s domain.
* **Use *defaultHtmlEscape=true* with SpringMVC for better cross-site scripting prevention**(OPT.JAVA.SEC\_JAVA.SpringNoAntiXssConfiguration): Please note that *defaultHtmlEscape* only transforms potential XSS attack payloads during HTTP output construction, for Spring tags in JSP pages. The default behavior, when no *defaultHtmlEscape* context parameter is defined, is to apply HTML escaping to all Spring tags in the form tag library (that render values), but not to the other tags that merely expose values but don’t render the values themselves.

**A8 Insecure deserialization.**

Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.

Kiuwan can detect misconfigurations, as:

* **Dynamic code injection during XML deserialization**(OPT.JAVA.SEC\_JAVA.CodeInjectionWithDeserializationRule): Certain Java serialization frameworks (like JDKs XmlEncoder/XmlDecoder or XStream library) provide a framework for arbitrary object persistence without mapping information), serializing objects to XML documents. Such frameworks allow to serialize constructor and method calls, to reconstruct object instances properly (to call the object constructors or setter methods) during deserialization. If the application deserializes XML documents from untrusted sources (e.g. in a REST framework), and if an attacker can provide the XML document to be deserialized, he/she may be able to execute arbitrary code on the server, including opening a reverse shell to launch commands.

**A9 Using components with known vulnerabilities**

Components, such as libraries, frameworks and other software modules, almost always run with full privileges. When a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. To clarify, applications using components with known vulnerabilities may undermine application defenses and enable a range of possible attacks and impacts.

[Kiuwan Insights](https://www.kiuwan.com/insights-open-source/) comes to answer all these questions by providing a complete Components Inventory of 3rd party software used by your applications, and detailed information on Security, Obsolescence and Licensing Risks of those components.

**A10 Insufficient logging & monitoring**

Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show that the time needed to detect a breach is on average over 200 days, typically detected by external parties rather than internal processes or monitoring.

The exploitation of insufficient logging and monitoring is the bedrock of nearly every major incident. Attackers rely on the lack of monitoring and timely response to achieve their goals without being detected. One strategy for determining if you have sufficient monitoring is to examine the logs following penetration testing. The testers’ actions should be recorded sufficiently to understand what damages they may have inflicted.