INJECTION ATTACKS

The Complete Guide

SQL • LDAP • XML • XPATH • XXE • EL • OS COMMAND

kiuwan

INDEX

ACCORDING TO THE OWASP TOP 10 APPLICATION SECURITY RISKS (2017)

> ... ***Injection flaws are the most serious web application security risk.***
> (*https://www.owasp.org/index.php/Top_10-2017_Top_10*)

Injection attacks can be devastating to your business, both from a technical aspect and from the business side. Injection can result in data loss, corruption, or disclosure to unauthorized parties, loss of accountability, or denial of access. It can sometimes lead to complete host takeover. Once an injection attack takes place, you can no longer trust your data. It may be corrupted, or denial of access may occur.

> *Eliminating any opportunities for an attacker to take advantage of injection flaws should be a top concern for your business due to the potential impact of attack on critical business data.*

Do I need to be a security specialist to prevent injection attacks?

The short answer is No. You don't need to be an expert to protect yourself against injection attacks if you use a tool like **Kiuwan Code Security**.

Kiuwan Code Security assists you in answering these fundamental questions:

1. Does my app have injection vulnerabilities? Which ones and where?
2. How can I remediate them?
3. After fixing my code, have those vulnerabilities been fixed? Did I introduce new ones?

The aim of this guide is to provide some basic knowledge of injection attacks and how you can defend against them.

We will cover the basics of injection so that you can understand how injection attacks work and how to remediate those vulnerabilities.

But first of all, you should assess how vulnerable your application is. In other words, locate the "open doors" where an injection attack might succeed.

You could use a DAST approach, where you attack your application searching for "injection points," blocking any "malicious" request addressed to those vulnerable points. It's what we might call "a **symptomatic approach**": to detect the symptoms (the vulnerabilities) and provide a mechanism to avoid those consequences (basically, blocking requests with specific injection patterns).

Alternatively, a SAST approach (such as Kiuwan Code Security) is based on an "**etiological approach**", that is, in the study of its underlying causes. Symptoms are only the surface of the problem: root causes must be detected and fixed in their origin.

Know your enemy!! The more you know about how an injection attack works, the more you will be able to defend against it.

Once you clearly understand how injection works, it won't be complicated to prevent. You will also see that once a door is open (i.e. there exists an injection point) it's a matter of imagination on the part of an attacker to take advantage of it.

You can use Kiuwan to detect injection vulnerabilities in your source code, providing you with remediation action plans. But the more you know about how injection works, the more you will be able to assess the different remediation possibilities and how to define an action plan to defend against them, as well as to roll out prevention measures.

Consequently, you must first be aware of all the injection points (or injection vulnerabilities) of your app. Kiuwan will help you by scanning your source code and searching for all those injection points. It will show to you their root causes. It will provide remediation clues and will let you assess the effectiveness of your remediation.

WHAT IS INJECTION?

The Open Web Application Security Project (OWASP) defines injection vulnerabilities as follows:

A1:2017 – **Injection**

> *Injection flaws, such as SQL, NoSQL, 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.*

Injection is a broad concept that covers many different security risks. What's common to all of them is that "interpreters" running in the app's background can be intentionally cheated to run code that can be exploited by an attacker.

Depending on the underlying interpreter, injection flaws can occur on SQL engines, LDAP engines, OS command interpreters, XML interpreters, etc.

TAINTED DATA FLOW

A common injection root cause is to send ***untrusted*** (potentially tainted) data to an interpreter as part of a command or query.

Source locations are those places in the code from where data comes in, that can be potentially controlled by the user (or the environment) and must consequently be presumably considered as ***tainted*** (as they may be used to build injection attacks).

> *User input should always be considered as **untrusted** (you will have no way to know if a user is an attacker or a normal app user).*

Sink locations are those places where consumed data must be ***untainted***.

> *Data used by any interpreter (a sink) must always be **trusted** (must not be controlled by a threat), i.e. sensitive data sinks rely on trusted (**untainted**) data*

Your app will contain an ***injection point*** (an injection vulnerability) for each a data flow where any sink consumes input data which is not being properly neutralized. Kiuwan scans your source code to find any injection point:

> *For all possible sinks, prove that tainted data will never be used where untainted data is expected.*

INJECTION PREVENTION RULES

As is common in security issues, there's not one single, unique protection mechanism for injection attacks. The best approach is combining more than one.

In this sense, to prevent Injection attacks we can consider the following complementary approaches:

1. The first line of defense consists on ***using the interpreter though a safe API*** (i.e an API that avoids the use of the interpreter entirely or provides a parameterized interface). Even so, be careful of APIs such as stored procedures which are parameterized, but that can still introduce injection under the hood.
2. Whether or not you can use a safe API, you should ***always perform an adequate input validation***. By input validation, we mean accepting only what is known to be good (whitelist), rejecting what is known to be bad (backlist) and escaping special characters using the specific escape syntax or the interpreter.

This is a general approach to prevent injection flaws. But, depending on the interpreter, there can be further prevention possibilities.

When you use Kiuwan Code Security to scan your source code, all the vulnerabilities of the same type are grouped under a Kiuwan rule, indicating how many files are "affected" (and where), how many vulnerabilities were found, and providing specific remediation clues based on the specifics of the programming language or interpreter being used.

What follows is a description of the main types of injection attacks, providing references to further detailed documentation and the detection coverage provided by Kiuwan.

INJECTION TYPES

Kiuwan provides out-of-the-box rules to detect Injection points in the following programming languages:

ABAP, ASP.NET, C/C++, C#, COBOL, Java, JavaScript, JSP, Objective-C, Oracle Forms, PHP, PL/SQL, Python, RPG IV, Swift, and Transact-SQL.

This list is continuously growing. If you need a particular programming language that is not currently included, please contact Kiuwan Support.

Below is a summary of the ***most common injection types***. Each type is referenced by its ID in the Common Weakness Enumeration (CWE) list of common vulnerabilities.

Visit **https://cwe.mitre.org/** for further information on each injection type.

- CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('***SQL Injection***')
- CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('***LDAP Injection***')
- CWE-91: XML Injection (aka ***Blind XPath Injection***)
- CWE-643: Improper Neutralization of Data within XPath Expressions ('***XPath Injection***')
- CWE-611: Improper Restriction of XML External Entity Reference ('***XXE***')
- CWE-917: Improper Neutralization of Special Elements used in an Expression Language Statement ('***Expression Language Injection***')
- CWE-78: Improper Neutralization of Special Elements used in an OS Command ('***OS Command Injection***')

Although this guide focuses on the injection types listed above, there are **many other types of injection attacks**. Below you can find a complete list of injection attacks not described in this paper but **covered by Kiuwan Code Security**. If you are a current Kiuwan Code Security user, you can find these in your Kiuwan model by searching for CWE ID#:

- CWE-15: External Control of System or Configuration Setting
- CWE-20: Improper Input Validation
- CWE-88: Argument Injection or Modification
- CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- CWE-564: SQL Injection: Hibernate
- CWE-345: Insufficient Verification of Data Authenticity
- CWE-93: Improper Neutralization of CRLF Sequences ('CRLF Injection')
- CWE-94: Improper Control of Generation of Code ('Code Injection')
- CWE-95: Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection')

- CWE-114: Process Control
- CWE-917: Improper Neutralization of Special Elements used in an Expression Language Statement ('Expression Language Injection')
- CWE-99: Improper Control of Resource Identifiers ('Resource Injection')
- CWE-113: Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Response Splitting')
- CWE-494: Download of Code Without Integrity Check
- CWE-117: Improper Output
- Neutralization for Logs
- CWE-134: Use of Externally-Controlled Format String
- CWE-159: Failure to Sanitize Special Element
- CWE-180: Incorrect Behavior Order: Validate Before Canonicalize
- CWE-183: Permissive Whitelist
- CWE-185: Incorrect Regular Expression
- CWE-235: Improper Handling of Extra Parameters
- CWE-346: Origin Validation Error
- CWE-352: Cross-Site Request Forgery (CSRF)
- CWE-470: Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection')
- CWE-472: External Control of Assumed-Immutable Web Parameter
- CWE-473: PHP External Variable Modification
- CWE-501: Trust Boundary Violation
- CWE-502: Deserialization of Untrusted Data
- CWE-601: URL Redirection to Untrusted Site ('Open Redirect')
- CWE-776: Improper Restriction of Recursive Entity References in DTDs ('XML Entity Expansion')
- CWE-652: Improper Neutralization of Data within XQuery Expressions ('XQuery Injection')
- CWE-915: Improperly Controlled Modification of Dynamically-Determined Object Attributes
- CWE-918: Server-Side Request Forgery (SSRF)

SQL INJECTION (CWE-89)

CWE-89 describes SQL injection as follows:

> *"The software constructs all or part of an **SQL command** using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component."*

Any SQL injection attack basically consists of insertion (or "injection") of malicious code within the SQL command executed by the app.

Effects of such malicious code injections can be unpredictable, depending on the attacker's intelligence and SQL-interpreter's characteristics. Some of the most common effects are:

- Read/modify sensitive data
- Execute administrative operations
- Execute commands on underlying OS

The most basic SQL-injection attack is based on exploiting a dynamically constructed SQL query based on input data.

Let's suppose an app that display user's data based on user's name as typed from the application user in a web form.

Dynamically constructed SQL in app code might be something as

```
"select * from users where name = '" + userName + "'";
```

userName is user-supplied data that is directly inserted into the query and it will be sent to SQL-engine to be executed.

Let's imagine the result when the attacker supplies next text

```
' or '1'='1
```

In this case, sql-engine will return all users' data because 1=1 will always be TRUE.

This would allow the attacker to access user data (involving a privacy breach). The consequences would be even more serious as a result of combining query chaining with administrative commands such as

> *Smith';drop table users; truncate audit_log;—*

In this case, the attacker would be able to delete the "Users" table or truncate system tables such as the audit log. The exact effect depends on the concrete case, but, with SQL injection, "the door is open." As you can imagine, the attacker's imagination is the limit. How does the attacker know the app database tables? Depending on the error messages the application produces when a SQL-injection attack happens, a smart attacker might be "inferring" database structure information from the error page. It's just a matter of having the patience to discover useful information.

To prevent this, you might adopt a common app error management approach consisting of a generic error page that doesn't display any exploitable information about the app's internals.

But even in this case, the app is still vulnerable to SQL injection. Without useful error information to exploit, an attacker simply uses a technique known as "Blind SQL Injection." This hacking technique is based on asking the database questions and determining the answer based on the application's response. This attack is often used when the web application is configured to show generic error messages but has not mitigated the code that is vulnerable to SQL injection.

Some variants of SQL injection apply to specific frameworks or conditions:

- CWE-564: SQL Injection: Hibernate
- CWE-566: Authorization Bypass Through User-Controlled SQL Primary Key

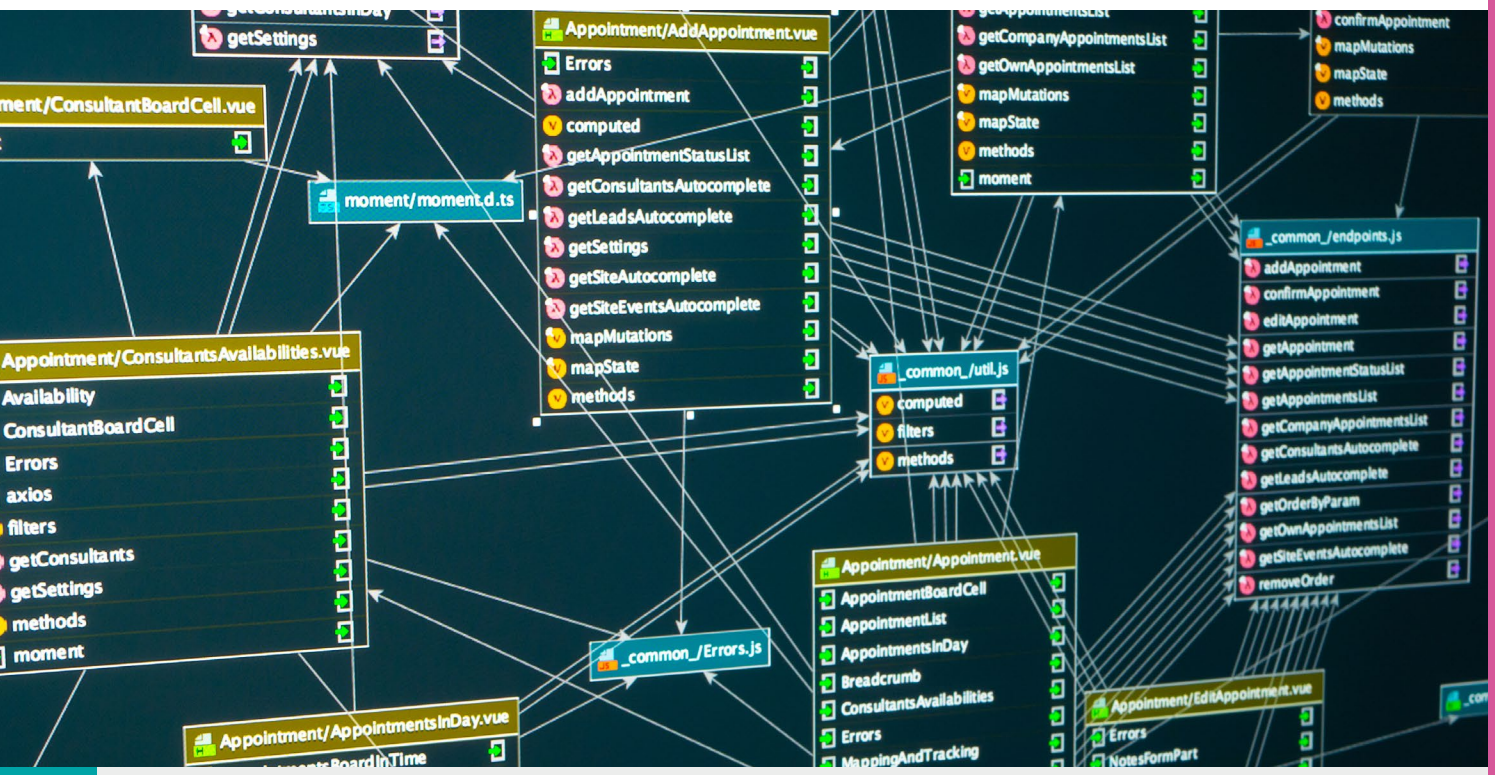
You can find further information on SQL Injection at
https://www.kiuwan.com/blog/sql- injection-avoid/

KIUWAN CODE SECURITY SQL INJECTION (CWE-89) COVERAGE

In Kiuwan Code Security, you can search rules covering SQL-Injection (CWE-89) filtering by Vulnerability Type ("Injection") and/or by CWE tag ("CWE:89").

Kiuwan incorporates next rules for SQL-Injection (CWE-89) for the following languages. Please, visit the documentation page for every rule to obtain detailed information on functionality, coverage, parameterization, remediation, example codes, etc.

Language	Rule Code
ABAP	OPT.ABAP.SEC.SqlInjection
C#	OPT.CSHARP.SqlInjection
COBOL	OPT.COBOL.SEC.SqlInjection
Java	OPT.HIBERNATE.BindParametersInQueries OPT.JAVA.ANDROID.ContentProviderUriInjection OPT.JAVA.SEC_JAVA.IBatisSqlInjectionRule OPT.JAVA.SEC_JAVA.SqlInjectionRule
JavaScript	OPT.JAVASCRIPT.SqlInjection
Objective-C	OPT.OBJECTIVEC.AvoidSqlInjection
Oracle Forms	OPT.ORACLEFORMS.SqlInjection
PHP	OPT.PHP.SqlInjection
Python	OPT.PYTHON.SECURITY.SqlInjection
RPG IV	OPT.RPG4.SEC.SqlInjection
Swift	OPT.SWIFT.SECURITY.SqlInjection
Transact-SQL	OPT.TRANSACTSQL.AvoidDynamicSql



LDAP INJECTION (CWE-90)

CWE-90 describes LDAP Injection as follows:

> *"The software constructs all or part of an **LDAP query** using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended LDAP query when it is sent to a downstream component."*

LDAP directory services are software applications that store and organize sensitive information. Apps typically use LDAP services for the following purposes:

- Access control (user-password verification, etc.)
- Privilege management
- Resource management

The key to exploiting LDAP through injection consists of manipulating the filters used to search into the directory services. The goal of an LDAP injection attack is to inject LDAP search filter metacharacters in a query which will be executed by the application.

A successful exploitation of LDAP injection vulnerability could allow the hacker to:

- Access unauthorized content
- Bypass application restrictions
- Add or modify objects within LDAP structure

LDAP injection attacks are based on the same techniques and principles of SQL injection attacks, i.e. the attacker takes advantage of dynamically constructed queries based on non-neutralized user input data. If the app does not properly filter input data, the attacker can inject malicious code.

To determine if an application is vulnerable to LDAP injection, simply try to send some data that should produce invalid input, such as an asterisk (*). If the app returns user data, it executed the query with your input. Therefore, the app is vulnerable. (Remember, that you can replace this trial and error approach with analysis by Kiuwan Code Security, which will identify all LDAP injection points.)

As a very basic example, let's suppose you face an app that asks for username and password in a web form. If the app is vulnerable to LDAP-injection, it reads username and password from the HTTP request and (without any filtering) builds the LDAP filter:

```
String filter = "(&(USER=" + username + ")(PASSWORD=" + password + "))";
```

resulting in a LDAP filter like:

(&(USER=username)(PASSWORD=password))

If the user would know a valid username of other user (for example, "user2"), an entry like

- username="user2)(&)"
- password="dddddd"

will produce a filter like

(&(USER= user2)(&))(PASSWORD=password))

This constitutes two filters, but only the first will be executed. As (&) is always true, just entering a valid username, without having a valid password, will grant access to any user's data.

KIUWAN CODE SECURITY LDAP INJECTION (CWE-90) COVERAGE

In Kiuwan Code Security, you can search rules covering LDAP-Injection (CWE-90) filtering by Vulnerability Type ("Injection") and/or by CWE tag ("CWE:90").

Kiuwan incorporates next rules for LDAP -Injection (CWE-90) for the following languages. Visit the documentation page for each rule to obtain detailed information on functionality, coverage, parameterization, remediation, example codes, etc.

Language	Rule Code
C#	OPT.CSHARP.LdapInjection
Java	OPT.JAVA.SEC_JAVA.LdapInjectionRule
Javascript	OPT.JAVASCRIPT.LdapInjection
PHP	OPT.PHP.LdapInjection

XML INJECTION (CWE-91)

CWE-91 describes XML Injection as follows:

> *"The software does not properly neutralize special elements that are used in XML, allowing attackers to modify the syntax, content, or commands of the XML before it is processed by an end system."*

XML Injection (CWE-91) attacks can be successful if the app does not properly neutralize special elements that are used in XML, allowing attackers to modify the syntax, content, or commands of the XML before it is processed by an end system.

By using special metacharacters, an attacker might be able to discover information about the XML structure, and then it will be able to try to inject XML data and tags (Tag Injection).

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 with 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.

Another case is when the application deserializes XML documents from untrusted sources (e.g. in a REST framework). 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.

KIUWAN CODE SECURITY XML INJECTION (CWE-91) COVERAGE

In Kiuwan Code Security, you can search rules covering XML-Injection (CWE-90) filtering by Vulnerability Type ("Injection") and/or by CWE tag ("CWE:91").

Kiuwan Code Security incorporates next rules for XML-Injection (CWE-91) for the languages listed below. Visit the documentation page for every rule to obtain detailed information on functionality, coverage, parameterization, remediation, example codes, etc.

Language	Rule Code	CWE
C#	OPT.CSHARP.JSONInjection OPT.CSHARP.XMLInjection	91 91
Java	OPT.JAVA.SEC_JAVA.XsltInjection	91
Objective-C	OPT.OBJECTIVEC.JSONInjection	91,345
PHP	OPT.PHP.SEC.XsltInjection	91
Python	OPT.PYTHON.SECURITY.XmlInjection	91
Swift	OPT.SWIFT.SECURITY.XMLInjection	91

XPATH INJECTION (CWE-643)

CWE-643 describes XPath Injection as follows:

> *"The software uses external input to dynamically construct an **XPath expression** used to retrieve data from an XML database, but it does not neutralize or incorrectly neutralizes that input. This allows an attacker to control the structure of the query."*

Similar to SQL injection, XPath injection (CWE-643) attacks occur when a web site relies on user-supplied information to construct an XPath query for XML data. By sending intentionally malformed information into the web site, attackers find out how the XML data is structured, or access data that they may not normally have access to. They may even be able to elevate their privileges on the web site if the XML data is being used for authentication (such as an XML based user file).

The net effect is that attackers will have control over the information selected from the XML database and may use that ability to control application flow, modify logic, retrieve unauthorized data, or bypass important checks (e.g. authentication).

Querying XML is done with XPath, a type of simple descriptive statement that allows the XML query to locate a piece of information. Like SQL, you can specify certain attributes to find, and patterns to match. When using XML for a web site it is common to accept some form of input on the query string to identify the content to locate and display on the page. This input must be sanitized to verify that it doesn't mess up the XPath query and return the wrong data.

KIUWAN CODE SECURITY XPATH INJECTION (CWE-643) COVERAGE

In Kiuwan Code Security, you can search rules covering XPath injection (CWE-643) filtering by Vulnerability Type ("Injection") and/or by CWE tag ("CWE:643").

Kiuwan Code Security incorporates next rules for XPath-Injection (CWE-643) for the following languages. Visit the documentation page for each rule to obtain detailed information on functionality, coverage, parameterization, remediation, example codes, etc.

Language	Rule Code
C#	OPT.CSHARP.XPathInjection
Java	OPT.JAVA.SEC_JAVA.XPathInjectionRule
Javascript	OPT.JAVASCRIPT.XPathInjection
Objective-C	OPT.OBJECTIVEC.XPathInjection
PHP	OPT.PHP.XPathInjection
Python	OPT.PYTHON.SECURITY.XpathInjection
Swift	OPT.SWIFT.SECURITY.XpathInjection

XML EXTERNAL ENTITY REFERENCE (XXE) (CWE-611)

CWE-611 describes XXE injection as follows:

> *"The software processes an XML document that can contain **XML entities** with URIs that resolves to documents outside of the intended sphere of control, causing the product to embed incorrect documents into its output."*

An XML External Entity attack is a type of attack against an application that parses XML input. This attack occurs when XML input containing a reference to an external entity is processed by a weakly configured XML parser. This attack may lead to the disclosure of confidential data, denial of service, server-side request forgery and other system impacts.

The XML standard defines the structure of an XML document. The standard defines a concept called an entity, which is a storage unit of some type.

There are a few different types of entities (external entity), that can access local or remote content via a declared system identifier. The system identifier is assumed to be a URI that can be dereferenced (accessed) by the XML processor when processing the entity. The XML processor then replaces occurrences of the named external entity with the contents dereferenced by the system identifier. If the system identifier contains tainted data and the XML processor dereferences this tainted data, the XML processor may disclose confidential information normally not accessible by the application. Similar attack vectors apply the usage of external DTDs, external stylesheets, external schemas, etc. which, when included, allow similar external resource inclusion style attacks.

Attacks can include disclosing local files, which may contain sensitive data such as passwords or private user data. Since the attack occurs relative to the application processing the XML document, an attacker may use this trusted application to pivot to other internal systems, possibly disclosing other internal content via http(s) requests or launching a Cross-site request forgery (CSRF) attack to any unprotected internal services. In some situations, an XML processor library that is vulnerable to client-side memory corruption issues may be exploited by dereferencing a malicious URI, possibly allowing arbitrary code execution under the application account. Other attacks can access local resources that may not stop returning data, possibly impacting application availability if too many threads or processes are not released.

KIUWAN CODE SECURITY XXE (CWE-611) COVERAGE

In Kiuwan Code Security, you can search rules covering XXE (CWE-611) filtering by Vulnerability Type ("Injection") and/or by CWE tag ("CWE:611").

Kiuwan Code Security incorporates rules for XXE (CWE-611) for the following languages. Visit the documentation page for each rule to obtain detailed information on functionality, coverage, parameterization, remediation, example codes, etc.

Language	Rule Code
C#	OPT.CSHARP.SEC.XMLEntityInjection
Java	OPT.JAVA.SEC_JAVA.XmlEntityInjectionRule
Javascript	OPT.JAVASCRIPT.XmlEntityInjection
Objective-C	OPT.OBJECTIVEC.XMLEntityInjection
PHP	CUS.PHP.XmlEntityInjection OPT.PHP.XmlEntityInjection
Python	OPT.PYTHON.SECURITY.XmlEntityInjection
Swift	OPT.SWIFT.SECURITY.XMLEntityInjection

EXPRESSION LANGUAGE (EL) INJECTION (CWE-917)

CWE-917 describes Expression Language (EL) injection as follows:

> *"The software constructs all or part of an **expression language (EL) statement in a Java Server Page (JSP)** using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended EL statement before it is executed."*

Another interpreter suitable to be attacked by injection is Expression Language (EL) in JSPs. Expression Language (EL) Injection happens when attacker-controlled data enters an EL interpreter.

In frameworks like Spring MVC, EL tags are evaluated twice (one by the application server and the result is evaluated as EL expression again by the Spring tag implementation), which allows an attacker to pass in the HTTP request message a value (header, cookie, message parameter) containing EL expression that could be executed.

Depending on the context, this may allow execution of arbitrary code, modification of unintended session or application attributes, or even downloading remote malicious Java classes with custom classloaders.

Other frameworks, like Struts, use a similar expression language (OGNL) that in certain cases allow double execution of OGNL.

KIUWAN CODE SECURITY EL INJECTION (CWE-917) COVERAGE

In Kiuwan Code Security, you can search rules covering EL Injection (CWE-917) filtering by Vulnerability Type ("Injection") and/or by CWE tag ("CWE:917").

Kiuwan Code Security incorporates rules for EL Injection (CWE-917) for the following language. Visit the documentation page for each rule to obtain detailed information on functionality, coverage, parameterization, remediation, example codes, etc.

Language	Rule Code
JSP	OPT.JSP.SEC_JSP.ExpressionLanguageInjection

OS COMMAND INJECTION (CWE- 78)

CWE-78 describes OS Command injection as follows:

> *"The software constructs all or part of an **OS command** using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended OS command when it is sent to a downstream component."*

OS Command injection is therefore an attack in which the goal is execution of arbitrary commands on the host operating system. These attacks are possible when an application passes unsafe user supplied data (forms, cookies, HTTP headers etc.) to a system shell, being usually executed with the privileges of the vulnerable application.

The impact of command injection attacks ranges from loss of data confidentiality and integrity – such as accessing resources without proper privileges – to unauthorized remote access to the system that hosts the vulnerable application (being able to perform malicious actions such as delete files, add new users, etc.).

Unlike other injection attacks based on specific languages, command injection attacks can occur in any OS (Windows and Unix-based) and affect any programming language that might call OS commands (C/C++, Java, PHP, etc.).

Obviously, first remediation should go in the direction of using API calls instead of external commands (if possible) or to ensure that the application runs under a non-privileged account with rights for the intended commands.

Anyway, the main reason that an application is vulnerable to command injection attacks is due to incorrect or insufficient input data validation by the application. Therefore, ***sanitization of user input should always be done.***

In case of a web app, the URL and form data need to be sanitized for invalid characters. A blacklist of characters is an option, but it may be difficult to think of all the characters to validate against. A better approach would be based on creating a whitelist containing only allowable characters, or a command list to validate the user input.

Let's have a look at this very basic example. As you can see, user data is collected through program arguments and directly used to construct an OS command.

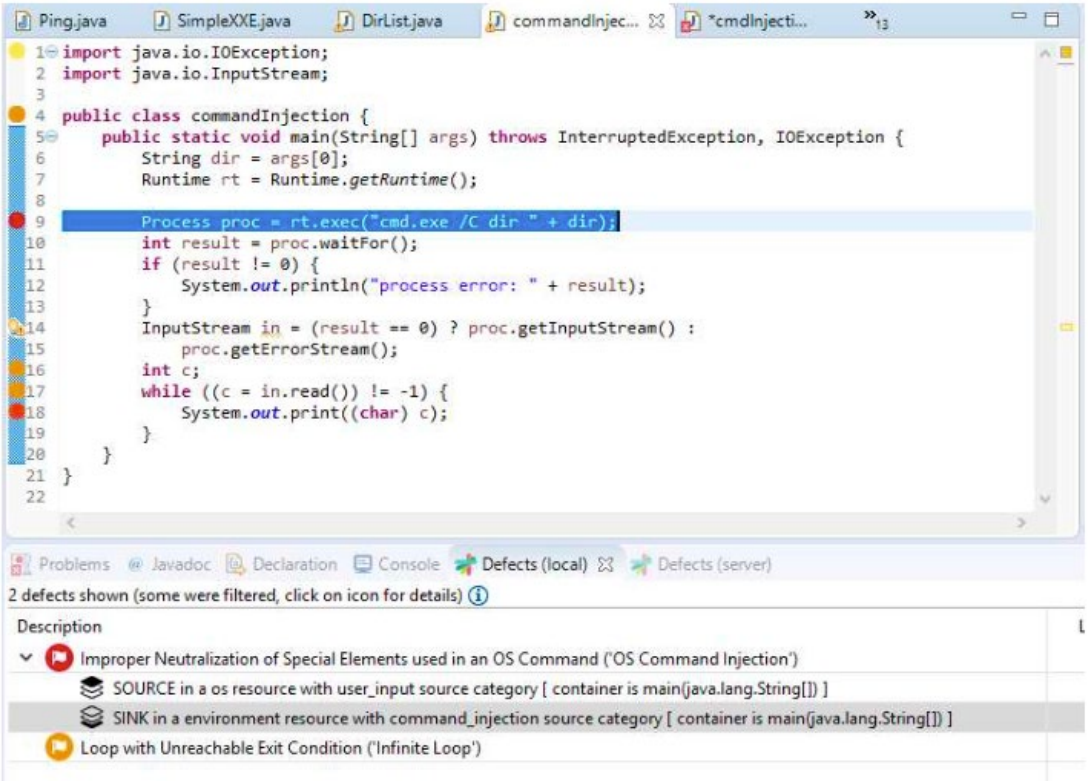
```
public class commandInjection {
    public static void main(String[] args) throws
InterruptedException, IOException {
            String dir = args[0];
            Runtime rt = Runtime.getRuntime();
            Process proc = rt.exec("cmd.exe /C dir " + dir);
            int result = proc.waitFor();
            if (result != 0) {
                System.out.println("process error: " + result);
            }
            InputStream in = (result == 0) ? proc.getInputStream() :
                proc.getErrorStream();
            int c;
            while ((c = in.read()) != -1) {
                System.out.print((char) c);
            }
    }
}
```

It's easy to imagine the result of running this program with the arguments below:

```
"c:\tmp > dir.txt & type c:\Windows\system.ini"
```

In this example, the program will display system.ini configuration file, but the most important thing is that the attacker gains full control over the attacked system. It's an open door – and a smart hacker will take full advantage of it.

Developers using the Kiuwan Code Security plug-in for their IDE will be automatically alerted of the vulnerability, indicating the sink and the source of the injection, as shown in the image below.



As said above, a check of user data against a whitelist containing only allowable characters (or command list) will remediate the vulnerability.

```
if (Pattern.matches("[0-9A-Za-z@.]+", dir)) {
        Process proc = rt.exec("cmd.exe /C dir " + dir);
}
```

KIUWAN CODE SECURITY OS COMMAND INJECTION (CWE-78) COVERAGE

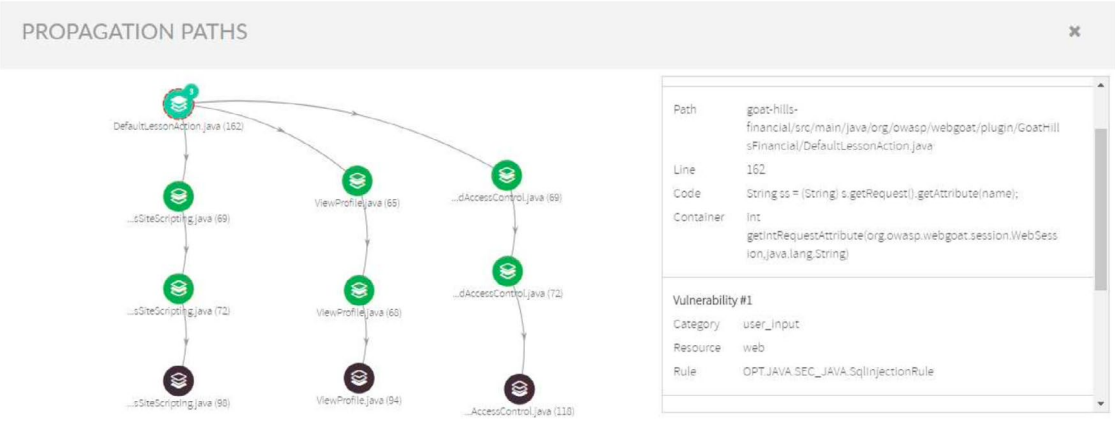
In Kiuwan Code Security, you can search rules covering OS Command Injection (CWE-78) filtering by Vulnerability Type ("Injection") and/or by CWE tag ("CWE:78").

Kiuwan Code Security incorporates rules for OS Command Injection (CWE-78) for the languages listed below. Visit the documentation page for each rule to obtain detailed information on functionality, coverage, parameterization, remediation, example codes, etc.

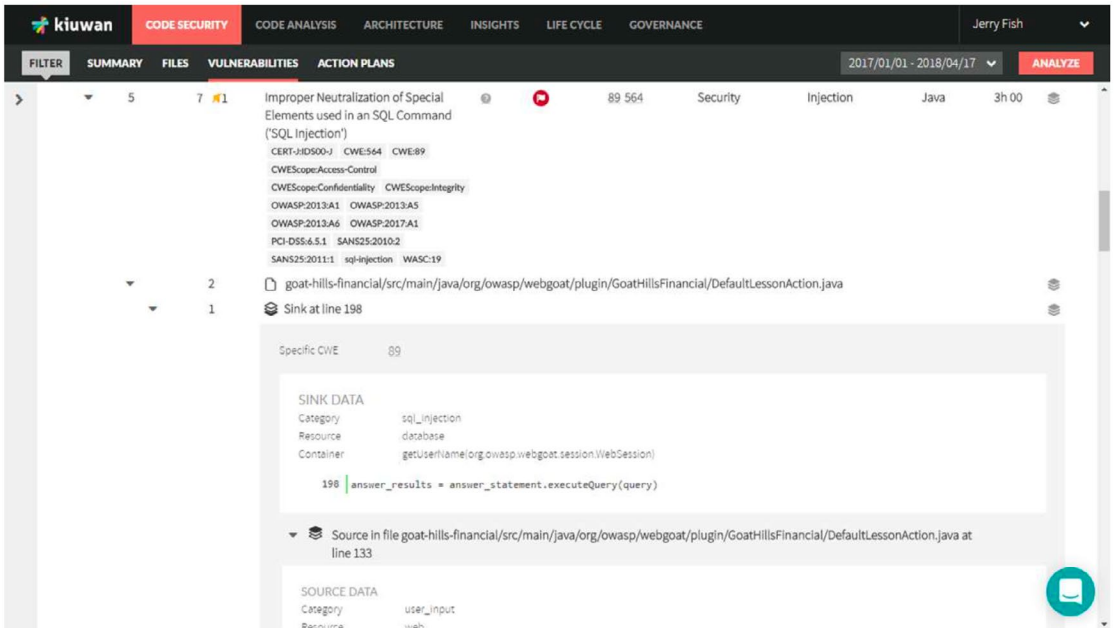
Language	Rule Code
Abap	OPT.ABAP.SEC.CommandInjection
C	OPT.C.CERTC.ENV04 OPT.C.CERTC.STR02
C#	OPT.CSHARP.CommandInjection
C++	OPT.CPP.CERTC.ENV04 OPT.CPP.CERTC.STR02
Cobol	OPT.COBOL.SEC.OSCommandInjection
Java	OPT.JAVA.SEC_JAVA.CommandInjectionRule
Javascript	OPT.JAVASCRIPT.CommandInjection
Objective-C	OPT.OBJECTIVEC.DoNotUseSystem
PHP	OPT.PHP.CommandInjection
Python	OPT.PYTHON.SECURITY.CommandInjection
RPG IV	OPT.RPG4.SEC.OSCommandInjection
Swift	OPT.SWIFT.SECURITY.CommandInjection

SOME KIUWAN SCREENSHOTS

Propagation Paths for an Injection vulnerability



Detailed vulnerability info, including labels for CWE and other security standards, involved lines of code, remediation info, and more



TRY KIUWAN CODE SECURITY FOR FREE

KIUWAN.COM/FREE

CONTACT US

CONTACT@KIUWAN.COM
LIVE CHAT: KIUWAN.COM

BECOME A PARTNER

PARTNERS@KIUWAN.COM

kiuwan