How Does CVE-2025-5878 Impact ESAPI?

Kevin W. Wall <kevin.w.wall@gmail.com>

Summary

Description:	This vulnerability enables the bypass of SQL injection defenses in ESAPI but is not a SQL injection vulnerability in ESAPI itself. According to the vendor, misleading Java class documentation for **org.owasp.esapi.codecs.OracleCodec** may have led to unsafe use of the **Encoder.encodeForSQL** interface. This interface is inherently unsafe and requires additional data validation. Similar risks exist for **MySQLCodec** and **DB2Codec**. As a mitigation, the project has disabled **Encoder.encodeForSQL** by default and any attempt to use it will now trigger a warning. **[Auto-generated; update this with the actual CVE description once the CVE is officially created.]**
Module:	Encoder.encodeForSQL, OracleCodec; likely DB2Codec and MySQLCodec are affected as well.
Publicly Announced:	As part of the ESAPI 2.7.0.0 release, in this security bulletin and CVE-2025-5878 itself.
Credits:	Longlong Gong; VulDB CNA team; ESAPI team
Affects:	All versions of ESAPI 2.x up through and including 2.6.2.0; fixed in 2.7.0.0.
Impact:	**May be exploitable (as SQL injection), depending on if and how you use the affected methods and classes.** **If you still wish to use the Encoder.encodeForSQL interface though (or more specifically,its implementation, DefaultEncoder.encodeForSQL), you will first have to explicitly allow it. If you enable it, its use will get logged as a warning in ESAPI. (If you are already using encodeForSQL, it temporarily will "break" your runtime code with an unchecked exception until you explicitly allow it again. See Appendix B for details.) We also have deprecated the Encoder.encodeForSQL interface and the OracleCodec, MySQLCodec, and DB2Codec classes.**
GitHub Issue #:	None.
Related:	https://github.com/uglory-gll/javasec/blob/main/ESAPI.md (Vulnerability explanation and PoC exploit)
CWE:	Primary – CWE-138: Improper Neutralization of Special Elements

	Secondary – CWE-20: Improper Input Validation
CVE Identifier:	CVE-2025-5878
CVSS scores:	CNA score: (based on https://vuldb.com/?kb.cvss) • CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:L/VI:L/VA:L/SC:N/SI:N/SA:N [6.9] • CVSSv3: AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:L/A:L [7.3] • CVSSv2: AV:N/AC:L/Au:N/C:P/I:P/A:P [7.5] NVD score: TBD

Background

OWASP ESAPI (the 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 for Java library is designed to make it easier for programmers to retrofit security into existing applications. ESAPI for Java also serves as a solid foundation for new development.

One of the security controls provided by ESAPI for Java is its "Encoders", which provide an interface for an assortment of methods for encoding output and decoding input of various types. However, its primary use is to provide *output* encoding as a defense against XSS . The individual encoding and decoding methods generally are implemented in association with particular Codec classes.

Summary Description

A vulnerability has been found in all ESAPI 2.x versions up through 2.6.2.0. The vulnerability allows the bypass of intended SQL Injection defenses. [Note this vulnerability is not a SQL Injection in itself, as the ESAPI library contains no SQL statements of any kind.] The major issue is a combination of absent, incorrect, and misleading documentation where the dangers of the methods are not clearly specified. In a nutshell, the Java documentation over promises, while the code in question, under delivers.

From the perspective of the ESAPI development team, the class Javadoc for the org.owasp.esapi.codecs.OracleCodec class was incorrect and in conflict with warnings provided in the Encoder.encodeForSQL interface. Specifically, it gave an example of a SQL statement that was claimed to be safe to encode with that Codec class without any additional input validation. Since the OracleCodec class was originally written, \ (backslash) was added as a default escape character for Oracle Database, but the escape character itself was not being handled correctly.

The Details

One of the Encoder methods, Encoder.encodeForSQL, had the following documentation associated with it. (This version is associated with ESAPI 2.6.2.0.) [NOTE: Red text is not in the original; it is used here for emphasis.]

> Encode input for use in a SQL query, according to the selected codec (appropriate codecs include the MySQLCodec and OracleCodec). This method is not recommended.

> The use of the `PreparedStatement` interface is the preferred approach. However, if for some reason this is impossible, then this method is provided as a weaker alternative. The best approach is to make sure any single-quotes are double-quoted. Another possible approach is to use the {escape} syntax described in the JDBC specification in section 1.5.6. However, this syntax does not work with all drivers, and requires modification of all queries.

However, what is shown for the Javadoc summary sentence associated with method signatures for Encoder.encodeForSQL in the Summary section is not so enlightening, as it only stated this (the first sentence of the method's Javadoc):

> Encode input for use in a SQL query, according to the selected codec (appropriate codecs include the MySQLCodec and OracleCodec).

What's shown in red in the full method's Javadoc is there for emphasis, but is not there in the original. Between these two, it is easy to see how the dangers could be missed, especially if IDE code-completion only shows the Summary portion. Still, had that been the only documentation problem, the ESAPI team was prepared to tell the CNA to reject the issue. (In fact, I initially did so, but then changed my mind because of this problem in the OracleCodec Javadoc.)

The signature for this encodeForSQL method is:

```
encodeForSQL(Codec codec, String input)
```

It turns out there are three acceptable Codec subclasses that refer to database query encoding:

- DB2Codec: Implementation of the Codec interface for DB2 strings. This function will only protect you from SQLi in limited situations.
- MySQLCodec: Codec implementation which can be used to escape string literals in MySQL.
  Implementation accepts 2 Modes as identified by the OWASP Recommended escaping strategies:
- OracleCodec: Implementation of the Codec interface for Oracle strings. This function will only protect you from SQLi in the case of user data bring placed within an Oracle quoted string such as: select * from table where user_name=' USERDATA ';

But it was the OracleCodec class Javadoc show here in red for emphasis, that resulted in me contacting the CNA a second time and accepting the CVE. Without a significantly severe CVE, there was concern that many in the ESAPI user community would remain ignorant of this unsafe method, because many users do not follow our release notes, security bulletins, and announcements in GitHub discussions and ESAPI news groups.

Remediation for this CVE

We created a new ESAPI release (2.7.0.0) that does the following:

- Rewrote most of the associated Javadoc to improve and clarify the warnings and then include this Security Bulletin with similar warnings as well so that developers

will be more apt to notice the inherent dangers of the Encoder.encodeForSQL interface.

- Marks the Encoder.encodeForSQL interface, and the DB2Codec, OracleCodec, and MySQLCodec as *deprecated* and noted them for *candidates* for removal 1 year after the 2.7.0.0 release.
- Disabled by default the DefaultEncoder.encodeForSQL method, which is the implementation method for the Encoder.encodeForSQL interface. We did that because:
    1. Deprecation warnings during builds often go unnoticed.
    2. There would be some applications that would just drop the new ESAPI jar into deployment and not recompile at all.
- If you do enable it for use, it logs a warning that can be used as an alert in your SIEM that can be monitored to tell which application is using it.

Starting in ESAPI 2.7.0.0, in order to use the DefaultEncoder.encodeForSQL (the default reference implementation) with any of the Codecs requires a developer to first explicitly enable it by adding the fully qualified method name as the value to a new property name (**ESAPI.dangerouslyAllowUnsafeMethods.methodName**) in the **ESAPI.properties** file. That is, for ESAPI 2.7.0.0 and later, ESAPI will have the encodeForSQL method in the reference implementation disabled by default. If you try to use it without explicitly enabling it, ESAPI will throw NotConfiguredByDefaultException, which is an unchecked exception. See Appendix B for additional details about the use of this property, the exception, and messages that are logged.

Explanation of the Exploit

Security researcher Longlong Gong has prepared an excellent explanation along with a Proof of Concept working exploit at https://github.com/uglory-gll/javasec/blob/main/ESAPI.md. What makes this proof of concept exploit so interesting (and a reason we felt we could not reject it) is because the SQL statement that it is bypassing is essentially the same pattern that the OracleCodec Javadoc for ESAPI 2.6.2.0 (the latest version at the time and the one tested against) claims that ESAPI would safely encode. So kudos to Longlong Gong for that.

The reason that this exploit works is that it passes in the value for the “id” query parameter string (when decoded) of:

> “1\’ and if(1=1,sleep(5),1)--”

to the Oracle Database. If you recall the original SQL statement was:

> “select * from sqli where id = ‘” + ESAPI.encoder().encodeForSQL(oracleCodec, id) + “’”;

So after processing, the SQL query ends up getting processed as:

> “select * from sqli where id = ‘1’ and if(1=1,sleep(5),1)--”

Because \ is the default escape character for Oracle Database, the \' sequence results in the single quote being escaped, so it terminates the first term in the WHERE clause. The 'and' introduces a second term, which tells the database to sleep for 5 seconds when the 'if' clause is true.

OracleCodec failed here because it was not doing any special processing of the escape character itself. Normally, codecs will want to also escape the escape character itself (thus rewriting a single backslash as \\, resulting in it being interpreted as a harmless \ rather than it being interpreted as instructions to escape the following character. However, it wasn't coded this way because back in 2007, the author of OracleCodec researched Oracle Database and did not find any evidence of any escape character. Times change, and in this case, it resulted in a vulnerability that resulted in allowing a SQL injection.

Note that while for Oracle Database the backslash character is the *default* escape character, it can be defined dynamically by the developer. This is the short reason why we decided not to try to fix this. (E.g., see the OracleCodec class Javadoc for ESAPI 2.7.0..0 or later for details.)

Impact

If your application is using the Encoder.encodeForSQL interface—either directly or via a direct or transitive dependency—and there an exploitable path to the SQL statement that encodeForSQL is attempting to escape, then SQL injection vulnerabilities are possible unless Encoder.encodeForSQL has been used ***with extreme caution***. In general, this approach was never advised and is now tagged as deprecated and is a *candidate* for removal in the near future (as soon as a year). (Note we do NOT intend to simply delete these deprecated methods without first engaging in a discussion with the ESAPI community to allow your voice to be heard. Watch the ESAPI GitHub's Discussion space for a post called "Should ESAPI Remote the Deprecated Encoder.encodeForSQL Interface?".)

How Can I Tell If My Application Is Impacted?

The simple short answer is, "recursively grep your code to look for a method named 'encodeForSQL'". If you find one and it's the one for ESAPI, you probably are affected unless you provide sufficiently strong data validation to prevent SQL injection.

The longer answer is more complicated, as there is always a possibility that you are using ESAPI's DefaultEncoder.encodeForSQL or one of its 3 database Codecs in a vulnerable manner via a dependency that exposes you to an exploitable path. That is much more difficult to determine. You can individually thoroughly research each of the dependencies that are using ESAPI to see if they are using encodeForSQL (or OracleCodec; possibly DB2Codec and MySQLCodec) and see if that code is reachable from your code so that there is an exploitable path. Some SCA tools have "reachability analysis" that may be helpful in such a context. IAST tools such as Contrast Security may also be helpful if you have good regression tests or you have a web or API interface that you can use in combination with a tool for penetration testing.

Workarounds

Stop using Encoder.encodeForSQL!!! If you must use it for some reason (see the Appendix), then make sure that you apply additional security-in-depth such as canonicalizing the input followed by rigorous input validation, ideally using an allow-list approach. Note that the ESAPI Validator interface may be a good place to start. At the very least, you will want to ensure that your validation of tainted input either properly escapes whatever escape character your SQL database is using for that particular SQL statement, or it removes the escape character entirely. However, perhaps a more robust option is to rewrite these SQL statements where you were relying on the Encoder.encodeForSQL interface as stored procedures.

Note are unable to provide specific examples, because developers often over generalize the provided solutions. Details need to try to be specific. If you post questions about this to the ESAPI Users Google Group "esapi-project-users" (note: you must first subscribe to the Google Group before posting) or the ESAPI GitHub Discussion page. We will try to answer them. We will NOT answer questions on the GitHub issues, so please do not post questions there as they will be closed without a response.

Solution

This section describes the high level changes made to ESAPI in version 2.7.0.0 to address this issue.

We debated whether to just flat-out remove the code or replace it with throwing an unchecked RuntimeException that referred to the GitHub security advisory. However, since we didn't want to be responsible for breaking people's production code, we decided the better course of action was to deprecate the the relevant methods and classes and then ***consider removing*** after giving the ESAPI user community some time for a healthy debate regarding whether they should go or stay. However, depending on the ensuing discussion, we may remove these methods in as short as 1 year's time, so beware.

Lessons Learned

I think there are two related lessons here, plus one last one that stands on its own:

1. You should not assume that technology and standards stand still.

Aside from the previously mentioned Javadoc issues which we believe are now addressed in ESAPI 2.7.0.0, the current Oracle JDBC Driver that was used in the proof-of-concept, now allows "\" (i.e., backslash) as an escape character. ESAPI founder and original implementer of encodeForSQL and OracleCodec, Jeff Williams, states that back in 2007(when these were originally written), only "'" (i.e., a single quote) was the only permissible escape character. [I trust Jeff when he says this, so I have not bothered to confirm its truthfulness. Call me lazy, but given this is water under the bridge, it hardly seems to matter.]

> In fact, the actual situation seems worse than that. Jeff Williams referred me to a PL/SQL reference for "SET ESCAPE" that allows for setting the escape character to some arbitrary character for Oracle SQL*Plus. (The default is the \ character.) It also turns out that by using the "ESCAPE" keyword, one can also change the Oracle database escape character, although it's not been confirmed one can do this via Oracle's JDBC driver. (See "How to Escape Characters in Oracle PL/SQL Queries" for

details and an example.) It is because of the possibility of changing the escape character that we believe escaping is indefensible and are not attempting any code fixes in the OracleCodec class. (Partially fixing deprecated, broken code known to be unsafe would only encourage developers to use it more.) Similar things may be possible in the DB2Codec and MySQLCodec classes if their respective databases allow similar constructs to change the escape character, so those are not being changed either. Instead, we have deprecated all 3 of those Codecs and the Encoder.encodeForSQL interface as well.

2. The static unit tests that we have are not sufficient to detect this sort of change in the JDBC drivers, in part because they are mostly mock tests based on 2007 JDBC tests.

   A better approach, but one much more complex (and potentially, expensive) to write and execute, would be to write automated dynamic tests similar in nature to the manual pen testing that security researcher, Longlong Gong, did.

   Conceptually this could work by configuring some test infrastructure by setting up a small web UI or API example to test against, built using the latest JDBC drivers and various and sundry SQL statements escaped using encodeForSQL and then connecting them to a real (but small) Oracle / IBM DB2 / MySQL database. Then regularly run some dynamic scanner(s) to test for SQL injections against the web UI or API endpoints.

   In all likelihood, we will not do this but instead, we will continue to rely on the abilities and good graces of offensive security white-hat researchers to responsibly report such findings to us. We hope that along with the new extensive warnings that we have added to the 2.7.0.0 release will make this a moot point in the future.

3. Documentation is important. If it is incomplete, obtuse, or contradictory, it can mislead developers to do dangerous things. The core team needs to review at least all the public Javadoc.

Acknowledgments

Security Researcher who brought this to our attention: Longlong Gong (uglory-gll)
Reviewers: Matt Seil, Jeremiah Stacey, Erika von Kampen, Jeff Williams, Ken Pyle, Bill Sempf

References

Longlong Gong's original vulnerability report:
https://github.com/uglory-gll/javasec/blob/main/ESAPI.md

Appendix A – Commonly Cited Use Cases for encodeForSQL

While reading though this section, it's important to keep in mind the following primary defenses, ranked in order of overall effectiveness as referenced in the OWASP SQL Injection Prevention Cheat Sheet, Primary Defenses section:

- Option 1: Use of Prepared Statements (with Parameterized Queries)

- Option 2: Use of Properly Constructed Stored Procedures
- Option 3: Allow-list Input Validation
- Option 4: STRONGLY DISCOURAGED: Escaping All User Supplied Input

Jeff Williams, ESAPI's founder, claims that sometimes Encoder.encodeForSQL (which corresponds to Option 4, above) is needed as a last resort for things that cannot be done by one of the other preferred options. Note that the current core ESAPI development team doesn't necessarily agree with all of these points, but one does what one feels they need to do in an expedient manner to protect their application, so we are not question any specific individual choices that have been made by well-informed development teams. However, we do believe that some of the previous existing ESAPI Javadoc was incomplete, possibly misleading and even partially contradictory in parts. We have striven to address these documentation issues in the ESAPI 2.7.0.0 release.

[Source: My summary, based on the content of lengthy SMS messages between myself and Jeff Williams on June 7, 2025. I have done my best to capture Jeff's thoughts in a succinct manner, but any mistakes here are of my own doing.]

- You sometimes need Option 4, but escaping is fragile protection.
  - It is necessary in some situations where you can't parameterize.
  - In fact, encodeForSQL is basically for when developers are doing crazy stuff, and that should be part of the risk calculation.
- In a parameterized SQL query, you cannot parameterize structural elements such as:
  - **Identifiers**: This includes things like:
    - T**able names**: You cannot use a parameter to specify which table the query should operate on, like SELECT * FROM @tablename.
    - **Column names**: Similarly, you cannot parameterize which columns to select, like SELECT @columnname FROM mytable.
    - **Schema names, database names**: These are also considered part of the query's structure and cannot be parameterized.
  - **SQL Keywords:** Keywords like SELECT, FROM, WHERE, ORDER BY, etc., cannot be parameterized.
  - **Operators**: You cannot parameterize operators, like using a parameter for the comparison operator in a WHERE clause (e.g., WHERE column @operator value).
  - **Function names**: Similar to operators, you cannot parameterize function names within the query.
  - **Expressions**: Complex expressions cannot be directly parameterized.
  - Lists of values for IN predicates: While you can use multiple parameter placeholders, you cannot use a single placeholder to represent a whole list of values within an IN() predicate.
- It has also been used for dynamic query building for performance that gets too complex to parameterize.
- Developers have also use escaping to fix complex concatenated queries because it's faster than changing everything to a parameterized query.

Opinion: I am *not* a SQL expert; I haven't even played one on TV or stayed at a Holiday Inn Express last night. But I have been in the AppSec field for more than 25 years and during that time have worked with dozens of DBAs who were SQL experts. And I believe

most of the DBAs would tell developers “if you can’t use Option 1 for the use cases presented here, Option 2 generally would be a better workable alternative than Option 4”. And I would add, that if you are going to resort to Option 4, then you should always use it with an additional security-in-depth layer, such as canonicalization, allow-list validation, and strong typing.

Appendix B – Enabling the Encoder.encodeForSQL Interface in ESAPI 2.7.0.0 and Later

So, you’ve decided after reading all this and all the warnings in the updated related Javadoc, that you still need to use Encoder.encodeForSQL and want to know how to enable “foot gun” mode and risk a SQL injection? Well, since ESAPI is open source, you could have just created your own version of ESAPI’s DefaultEncoder class and bypassed our attempts to protect you from yourself. Since we couldn’t stop you anyhow, we will show you how to do it without all that mess. (Because, chances are, if you copy DefaultEncoder, you likely wouldn’t keep up with any patches to it and that would be even worse.) So here’s what what we recommend that you do:

1. First, have your AppSec team analyze all your uses of the Encoder.encodeForSQL interface to determine if you are not leaving yourself vulnerable to SQL injection vulnerabilities.

2. Next, locate your application’s **ESAPI.properties** file and add the value “org.owasp.esapi.reference.DefaultEncoder.encodeForSQL” to the property named “ESAPI.dangerouslyAllowUnsafeMethods.methodNames”. If that property name is not present, either uncomment it or add it as appropriate. If you fail to do this step and DefaultEncoder.encodeForSQL is called, it will throw an unchecked exception (NotConfiguredByDefaultException) causing the call to fail. The exception message will look something like:

   > org.owasp.esapi.errors.NotConfiguredByDefaultException: Method not explicitly enabled in property ESAPI.dangerouslyAllowUnsafeMethods.methodNames; SIEM ALERT: Method 'org.owasp.esapi.reference.DefaultEncoder.encodeForSQL' has been invoked despite having credible security concerns; see CVE-2025-5878 and ESAPI Security Bulletin #13 for details.

   That is why we urge you to keep your AppSec team in the loop.

3. If you are going to do this, we strongly recommend that you also set a value for the new property “ESAPI.dangerouslyAllowUnsafeMethods.justification” to give a brief justification for this. An example might be:

   > ESAPI.dangerouslyAllowUnsafeMethods.justification=Approved by AppSec via security exception ticket InfoSecEx4763

   This justification will appear in the log file. If you do not have a justification “None” will be printed instead for the justification. If you were to use the above value for this

property, when ESAPI logs the event as a warning of an Logger.SECURITY_FAILURE, the logged warning message would look something like this:

> WARNING: [SECURITY FAILURE *username*:@64.14.103.52 -> 10.1.43.6:80/MyApplicationName/Encoder] SIEM ALERT: Method 'org.owasp.esapi.reference.DefaultEncoder.encodeForSQL' has been invoked despite having credible security concerns; for additional details, see see CVE-2025-5878 and ESAPI Security Bulletin #13 for details. Provided justification: Approved by AppSec via security exception ticket InfoSecEx4763

If the justification is left unset, the warning would instead be logged as:

> WARNING: [SECURITY FAILURE *username*:@64.14.103.52 -> 10.1.43.6:80/MyApplicationName/Encoder] SIEM ALERT: Method 'org.owasp.esapi.reference.DefaultEncoder.encodeForSQL' has been invoked despite having credible security concerns; for additional details, see see CVE-2025-5878 and ESAPI Security Bulletin #13 for details. Provided justification: None

The former, with some reasonable explanation given, is less likely to result in a phone call to one to your developers in the wee morning hours from someone monitoring SIEM alerts. You have been warned.

Note: In the above *username* is only replaced with the actual authenticated username if ESAPI is being used for authentication. (ESAPI.authenticator().getCurrentUser() is being used to obtain that.) Otherwise “Anonymous” will be used for the username.