Code review of the Data Integration Service

CLIENT	**Federal Chancellery**	CLASSIFICATION	**Public**
DATE	**March 1, 2023**	AUTHORS	**Philippe Oechslin, Thomas Hofer**
VERSION	**1.0**	DISTRIBUTION	**Federal Chancellery**
GIT COMMIT	**e588df1**	MODIFICATIONS	**classification**
STATUS	**Final**		

OBJECTIF SÉCURITÉ
Architecte de la sécurité informatique

Contents

OS

OS Objectif Sécurité SA
Route Cité-Ouest 19 - CH-1196 Gland
+41 22 364 85 70 - info@objectif-securite.ch

1 Introduction

1.1 Context

This report contains our review of a specific software used in the e-voting solution provided by Swiss Post. The review was mandated by the Federal Chancellery as part of the examination process according to OEV Article 10 paragraph 1.

The reviewed software (Data Information Service) is used to create the configuration file of an election event based on the electoral roll and the definition the election.

It is our understanding that the source code of the software will be published in the near future.

1.2 Execution of the work

The review was carried out in week 5 of 2023. We were given the full set of source code and the resources used to build the code, as well as input files necessary to run the application.

We were able to compile and debug the code as well as run it to generate an e-voting configuration.

1.3 Executive summary

The analysis of the results of the tests led us to the following conclusions:

No significant security issue: We found no evidence of code that would enable various attack scenarii that we imagined for the specific threat model of the environment in which the software is executed.

Vulnerable dependencies: Some of the application's dependencies have known vulnerabilities and should be updated or removed.

2 Analysis

The role of the software is to create a configuration file for an e-voting event. It processes the definition of an election and of the electoral roll and produces a single XML file containing all information necessary for setting up the the election event.

The software is installed on a secured standalone laptop, operated by at least two persons. Data is exchanged by USB keys. The laptop is used for configuring elections. It also contains the software (SDM) which implements a part of the cryptographic protocol of the setup phase.

During the setup phase, the SDM reads the configuration file created by the DIS and produces results which it signs with a key that is saved on the laptop.

The software uses the following assets:

- **Identity of voters:** the electoral roll,
- **Definition of the ballots:** questions, answers, candidates,
- **Signature key:** used for signing the produced xml documents.

The laptop also contains the following assets

- **SDM software:** The SDM uses the output of the DIS and implements a part of the cryptographic protocol during the setup phase.

- **Codes:** initialisation key, return codes, confirmation and finalisation code, produced by the SDM.

Malicious operation by the software could result in the following classes of attack:

Attack 1: **Adding or removing voters:**
This is mitigated by the manual verification of the number of voters and the reclamations of voter who do not receive a voting card.

Attack 2: **Manipulating the rights of the voters:** The software could manipulate the configuration of the election to allow certain voters to participate to more – or fewer – parts of an election (e.g cantonal or federal).
This is mitigated by the verification of the number of cast votes and the reclamations of voter who did not have are not able to participate to all expected parts of the election.

Attack 3: **Visible manipulation of voting cards to compromise individual verifiability:** The software could modify the phrasing of the questions or the names of candidates or parties to entice the voter to vote against their will. It could even manipulate the codes if it waits until they have been generated by other software on the laptop.
This is mitigated by the manual review of a sample of cards.

Attack 4: **Manipulating the trusted component:** The software could modify the behaviour of other software that runs on the same laptop or interact with their data. It could for example try to leak secret information through the data that is exported from the laptop or sign manipulated data with the keys that are available on the laptop.

We have identified two types of attackers:

- **Internal attackers:** An attacker who injects malicious code into the software, before it is delivered to the canton.
- **External attackers:** An attacker who injects code into the software through data that is given to the code. The code could for example be added to the address field of a voter, before the voter registry is imported.

Any malicious action by the operators or the software can be ignored in the analysis of the software because the operators already have the capability to manipulate the voting cards without the help of the software. Additionally, the operators are subject to strong security rules (e.g. 4 eyes principle) as mandated in Number 3 of the OEV Annex.

3 Analysis of the Code

3.1 Analysis specific to the identified attacks

The code creates the output file based on the inputs it reads. We did not find any code does not follow this principle and that would manipulate the resulting output file. This seems to exclude manipulations of the content of the cards or of the voting rights of the voters. (Attack 2 and 3)

We did not find any code that would read or modify other files on the laptop, other than the files needed for its operation. This seems to exclude any attack on the cryptographic protocol. (Attack 4).

We observed that the inputs of the software are XML files, which are handled by a standard XML parser, relying on well-defined XSD schemas. The use of a standard parser and a well-defined XSD schema should be sufficient in thwarting most injection risks.
We noted that there is a specific test for the proper escaping of some characters which is executed during compilation. We further confirmed that typical characters that could be used for injections (`` !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~ ``) are properly processed. While this does not guarantee that effective code injection is impossible, it makes it very improbable.

3.2 Generic analysis of the security of the code

In the `DataIntegrationService` class, line 46, a reference is made to the pipe `/pipes/cantonalTree/stdv1/cantonalTreeStdv1.groovy`. However, the path in the repository contains the `cantonaltree` directory, without the upper case `T`. This breaks the application on linux, where paths are case-sensitive.

Problem 1 - Vulnerable dependencies

Some of the dependencies needed by the project contain known vulnerabilities.

Dependency	Version
commons-jxpath	1.3 7
jdom	1.0
junit	4.12
svg-salamander	1.0
xercesImpl	2.12.2

Recommendation

The dependencies should be updated to versions without known vulnerabilities. Due to the controversy on the commons-jxpath vulnerabilities[1], a very careful analysis of the inputs given to the library should be conducted to ensure no input can be controlled by a malicious party.

[1] `https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=47133#c20`

Problem 2 - High cognitive complexity

While most of the code is structured in code units of reasonable sizes, some methods have a (very) high cognitive complexity. This complicates maintenance and review of the code. One such example is the `build` method of the `AuthorizationService` class, where the complexity is rated at 70.

Recommendation

Such complexities make it very hard to analyse those methods as a whole and require additional effort from maintainers and reviewers alike, thus increasing the risk of errors in logical flows and making such errors harder to spot. The code should be refactored to reduce the complexity.

Problem 3 - Generics usage

The `Stax2Wrapper` class raises several warnings related to generics usage, and could benefit from stronger type guarantees. Using two separate HashMaps to keep track of classes and consumers related to the same tag names seems like a misstep in encapsulation. So does keeping the type of the iterator and the value of the current node in the parent class rather than in the nested dedicated class.

Recommendation

Unchecked type conversions can lead to issues that tend to be very difficult to identify, try to avoid them altogether.

Problem 4 - Null value returned for a collection

In the `Ech015vx4Mapper` class, a `null` value is returned where a collection is expected. One would expect an empty collection to be returned instead.

Problem 5 - Inconsistent Java language level

While most of the code style is consistent, some places indicate a migration from an earlier version of Java that has not been fully completed. For instance, several places include the older `Stream.collect(Collectors.toList())` syntax, rather than the now recommended `Stream.toList()`.

Recommendation

While not necessarily an issue of itself, this indicates that the effort invested in the code is not consistent with other elements of the internet voting system.

4 Recommendations

Based on our analysis and tests, we can make the following recommendations:

4.1 Immediate recommendations:

We recommend to apply the following measure before the code is used in a voting operation:

- Have the software compiled at Swiss Post and record the hashes of the code, the executable and all dependencies in a trusted, observable way.
- Have the code reviewed after each modification, as long as the source code is not public.
- Verify that the dependencies are authentic and up to date.

4.2 Other recommendations

We understand that Swiss Post will soon publish the source code for general review and use a reproducible build.

- We recommend addressing the code quality issues pointed is this report as far as possible before publication.

5 Conclusions

The source code that we reviewed seems to faithfully create the configuration file from its inputs. However, if the software is not built during a trusted ceremony, there is no guarantee that the software is made from the analysed source code.

We identified 4 types of attacks that could be mounted in the specific setup in which the application is used. They can be easily excluded by reviewing the code.

If the recommendation above is applied we can conclude that we see no explicit danger in using the software.

Furthermore, the dependency management could be improved to identify components known to be vulnerable earlier in the process in order to update or remove them. Minor adjustments to code quality could also improve maintainability.