Code Review of Voting Card Printing Service V3.0

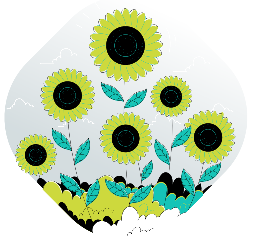
CLIENT	**Federal Chancellery**	CLASSIFICATION	**Public**
DATE	**June 5, 2025**	AUTHORS	**Philippe Oechslin, Thomas Hofer**
VERSION	**1.1**	DISTRIBUTION	**Federal Chancellery**
GIT COMMIT	**a6e47b3**	MODIFICATIONS	**Clarified attack 4**
STATUS	**Final**		

OBJECTIF SÉCURITÉ
Architecte de la sécurité informatique

Contents

OS

OS Objectif Sécurité SA
Chemin du Vernay 14a - CH-1196 Gland
+41 22 364 85 70 - info@objectif-securite.ch



1 Introduction

1.1 Context

This report contains our review of the Voting Card Printing Service (VCPS), a software used in Swiss Post's e-voting solution. The review was mandated by the Federal Chancellery as part of the examination process according to OEV Article 10 paragraph 1.

The VCPS is used after the content of the voting cards has been created. It transforms the raw data (names, codes, texts in XML format) to PDF files that can be sent to the printing office. It is a third-party software, which is not developed by Swiss Post.

1.2 Conditions of work

The audited software is the VCPS v3.0. This is the first version of the software that will have its source code published.

The review was carried out in the weeks 49 and 50 of 2024. We were given access to the source code on Gitlab and the instructions, scripts and dependencies to build the code. A set of input files was also provided such that voting cards could be generated with the software.

We were able to compile and debug the code as well as run it to generate cards for a test election.

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.

Overall design and code quality issues: While not an immediate threat to security, some code quality and architectural flaws complicate maintainability of the software and might lead to issues in the future.

2 Risk Analysis

2.1 Assets

The role of the software is to convert the voting cards from XML format to PDF, for printing.

The software uses the following assets:

In the `configuration.xml` file:
- **Identity of voters:** the voter register, names, addresses, voter identification numbers,
- **Definition of the ballots:** questions, answers, candidates.

In the `evoting-print_<name-of-election>.xml` file:
- **Codes:** for each voter identification number, initialisation key, return codes, confirmation and finalisation code.

In the configuration file of VCPS:
- **Signature key:** used for signing the produced PDF documents,
- **Encryption key:** used for encrypting the documents before their transfer to the printing office.

2.2 Attacks and their Impacts

The security of the e-voting system hinges on the fact that the content of the voting cards is not modified nor revealed.

Following classes of attacks are possible if the content of the cards is revealed or modified:

Attack 1: Removing cards:
Cards could be removed from the output to prevent some people from voting.

Attack 2: Inverting return codes: If the return codes of a question are inverted and the voting client is manipulated to invert the votes that are cast, the voters will see return codes that seem to correspond their intention. This completely compromises individual verifiability.

Attack 3: Inverting the meaning of the questions: If a question is modified to say 'Do you reject …' instead of 'Do you accept …', the voter will cast the opposite of their intentions and still receive the expected return code.

Attack 4: Leaking the initialisation and confirmation key:
An attacker having access to initialisation confirmation keys of voters, could vote in the name of the voters who do not make use of their voting rights.

Attack 5: Leaking the return or finalisation codes:
An attacker having access to return codes could set up a fake voting server that would return the expected return codes and cast a different ballot into the real voting server.
Similarly, the fake server could return the correct finalisation code without casting a ballot.

Attack 6: Leaking the encryption key of the PDF documents:
An attacker having access to this key could intercept the transmission of the PDF files to the printing office.

If the files can only be read by the attacker, this would result in the two leaking attacks described above. If the attacker can modify the files on the fly, the three first attacks would also be possible.

2.3 Mitigations in place

- The software is installed on a secured standalone laptop, operated by at least two trustworthy operators. Data is exchanged by USB keys.
- In the trust model it is thus assumed that the operators follow the instructions. The only operations that happen during the usage of the VCPS are the import of the configuration files, running of the VCPS, possibly viewing of the output files for verification and exporting the encrypted PDF files for transmission to the printing service.
- The VCPS is built from the published source code using a reproducible build procedure.
- The input files of the VCPS are signed and their signature verified.
- A sample of voting cards is reviewed manually to verify that their content is correct.

2.4 Potential Vulnerabilities

We can imagine the following vulnerabilities that would allow carrying out some of the attacks in spite of the mitigations that are in place.

Vulnerability 1: Malicious source code in the VCPS: An attacker could include some malicious code into the source of the VCPS to carry out the attacks.

Vulnerability 2: Malicious code injected through the input files: The software that parses the input files could be vulnerable to injection attacks. An attacker could thus include an injection vector into the input data. For example, an external party that provides the names of the voters, could add an injection vector into the name of a voter. The VCPS would receive a signed input file containing the injection vector and execute malicious code while reading the name of the voter.

Vulnerability 3: Leaking secrets through voting cards: Malicious code could hide return codes or initialisation keys in voting cards. The cards would then be printed and sent to the attacker by post, provided that the attacker is a voter.
Note that one of the two first vulnerabilities would be necessary, in order to have the malicious code that exploits this vulnerability.

Vulnerability 4: Intercepting the transmission to the printing office: An attacker could obtain a copy of the encrypted output files when they are transmitted to the printing office through an untrusted channel. By combining this vulnerability with the previous vulnerability that allows to leak the encryption key, the attacker would only have to wait for this voting card to arrive by mail to be able to decrypt the files.

All the vulnerabilities hinge on the fact that malicious code is executed by the VCPS either because it is part of it source code, or because it was injected by one of the input files.

In the rest of this audit, we will thus try to obtain assurance that those two vulnerabilities are not present in the software.

3 Analysis of the code

3.1 Third party software

The software makes use of following third party software:

- Apache FOP, generates postscript or PDF from XML
- ghostscript, generates PDF from postscript
- xml-signature-1.4.4.4.jar, to verify signatures, from Swiss Post evoting tools[1]
- file-cryptor-1.4.4.4-runnable.jar, from Swiss Post evoting tools[2]

3.2 Analysis specific to the identified attacks

✓ We searched for all the code lines that operate on the encryption key. The only ones we found were the lines used for executing the encrypting and for writing a log about the encryption. We did not find any code that would include the key in a voting card. This seems to exclude any attack where the key would be leaked in the PDF and then printed on a card (attack 6 through vulnerability 1).

✓ We analysed the way the voting cards are generated by the code. The information about the ballot and the voters is stored in one file. The codes are stored in a second XML file. The program first combines the two files to create on large XML file that with the sequence of all voting cards with their codes. Then, XSL style sheets are used to generate the different blocks that make up a voting card (e.g. the coverpage, the address, the return codes, etc.). Another XSL style sheet is finally used to add the physical layout (e.g. alignments and distances in millimeters) of all the elements. The resulting file is then given to Apache FOP to generate a PDF file.
In each step the cards are written sequentially, with the data of one voter at a time. We did not see any line of code that would insert data from a different card into the card being generated. This seems to exclude any attack where codes would be leaked by having them printed on the card (attacks 4 and 5 through vulnerability 1).

✓ We analysed how the PDF documents are zipped and encrypted. The PDF files are first written to disk, then added to a zip archive on disk. Then a third party software is called to create an encrypted copy of the zip file. This seems to guarantee that the PDF files which are used for review are the same as the ones that are transmitted to the printing office (attacks 2 and 3 through vulnerability 1).

✓ We analysed the style sheets used to position the texts on the voting cards, to verify that the codes are printed beside the correct answers or candidates. The style sheets are too complex to exclude any error in our analysis. However, the same stylesheets are used for all cards. A manual inspection of a sample of cards thus seems to exclude any attack that would do a visible manipulation (attacks 2 and 3 through vulnerability 1 or 2).

[1] `https://gitlab.com/swisspost-evoting/e-voting/e-voting/-/tree/master/tools/xml-signature/src/main/java/ch/post/it/evoting/tools/xmlsignature`
[2] `https://gitlab.com/swisspost-evoting/e-voting/e-voting/-/tree/master/tools/file-cryptor`

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 further confirmed that typical characters that could be used for injections (`` !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~ ``) are properly processed. They are either properly encoded or prevent the generation of the cards. We were not able to generate any unexpected behaviour of the code. While this does not guarantee that ode injection is impossible, it makes it very improbable. This prevents using vulnerability 2 to carry out any of the attacks.

3.3 Generic analysis of the security of the code

3.3.1 Security issues

All potential security issues that we reported in earlier versions have been fixed.

We noticed that this version of the software contained one vulnerable library.

Dependency	CVEs	Severity
System.Text.Json 9.0.0	CVE-2024-43485	High

This vulnerability is fixed in version 8.0.404 of .NET, dated November 12, 2024. The VCPS software is delivered with version 8.0.400 from August 2024.

According to Microsoft, unpatched systems are vulnerable to denial of service attacks, which have no impact to the attack scenarios described in this report.

3.3.2 Codes quality and structure

Problem 1 - Missing documentation

The code contains very few comments. For example, the code contains 17 commands that are executed to generate the output from the input. There is no explanation of what each command does.

For the audit of previous versions, we were given an internal report that explained the structure of the software.

Problem 2 - Configuration overengineering

Compared to previous versions, the configuration files have been simplified a little bit.

The cantonal configurations are stored in program files (.cs) which seem to be generated automatically from an XML source that is not published. Each cantonal configuration contains about 150 entries out of which less than 40 are different.

The code would be easier to understand if there was a set of generic default parameters, and each cantonal configuration would only contain the parameters which are different.

Moreover, it would be more readable if the parameters were written as XML configuration files, in the same way as the general configuration file VotingCardPrintService.config.

Problem 3 - Complexity of the code

Another source of complexity is due to the fact that the program is made of services that each consume and produce data and that are orchestrated by an engine that instantiates and connects the services together.

Some progress has been made by reducing the number of services from over 80 to 17.

The framework can be used to select and organise the services arbitrarily on the fly, depending on the choices that are made on the UI of the program.

It turns out that the user interface has only one button and that the same sequence of commands is carried out every time.

It should be possible to use a much simpler framework, with or without an UI, to carry out a hardwired sequence of operations. This would make the code much easier to audit as it would contain less code not related to the purpose of the program and the flow of execution would be easier to follow.

3.4 Deployment

Instructions for a reproducible build are given together with the hash of the program. This can be used to independently build the program and verify that the resulting program is really built from the sources that have be published.

Moreover, the program verifies the hashes of all of 3rd party dependencies on startup.

4 Discussion and conclusion

Compared to the previous versions, important progress has been made. Except for a recent patch that is missing, we have found no security issue.

The remaining issues we have are the lack of documentation and the unnecessary complexity of the code. While they have no direct impact on security, they make the code audit more difficult.

We have refined our attack model and are still of the opinion that for an attack to be possible, malicious code would either have to be included in the source code or injected in the input files. The publication of the code and the reproducible build give a strong guarantee that no malicious code is built into the program. Using XML formatted input files and up-to-date XML parsers protects against code that would be injected through the inputs.

Additionally, the review of a sample voting card against the input file allows verifying that the texts are correct and that the answers have not been swapped.

In the given trust model, we see no explicit danger in using the software.