Code Review of Voting Card Printing Service

CLIENT	**Federal Chancellery**	CLASSIFICATION	**Public**
DATE	**August 11, 2023**	AUTHORS	**Philippe Oechslin, Thomas Hofer**
VERSION	**1.6**	DISTRIBUTION	**Federal Chancellery**
GIT COMMIT	**f0a5633**	MODIFICATIONS	**Added appendix regarding version 2.17.0**
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 to participating Cantons. The review was mandated by the Federal Chancellery as part of the examination process according to OEV Article 10 paragraph 1.

The reviewed software (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 Execution of the work

Version 2.13.0: The review was carried out in the weeks 3 and 4 of 2023 on version 2.13.0 of the code. We were given the full set of source code and the resources used to build the code, as well as the packaged version of the code, that is delivered to the cantons.

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

Version 2.15.1: We reviewed the modification between version 2.13.0 and 2.15.1 on weeks 13 and 14 of 2023. The results are listed in appendix A.

Version 2.16.0: We reviewed the modification between version 2.15.1 and 2.16.0 in week 30 of 2023. The results are listed in appendix B.

Version 2.17.0: We reviewed the modification between version 2.16.1 and 2.17.0 in week 32 of 2023. The results are listed in appendix C.

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.

Unfortunate use of encryption: To prevent cross-canton leakage of voting card templates, the templates are packaged within the application after having been encrypted. This makes it much more tedious to verify that the templates do not include any malicious code. It would be safer and more aligned the transparency principle of the e-voting solution to only include the data pertaining to the target canton.

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 Analysis

The role of the software is to convert the voting cards from xml format to PDF, for printing. The security of the e-voting systems depends on the fact that the content of the voting card is not modified or revealed.

The software is installed on a secured standalone laptop, operated by at least two persons. Data is exchanged by USB keys.

A manual review of a sample of voting cards is already in place, to detect visible defects in the cards.

The software uses the following assets:

- **Identity of voters:** the voter register,
- **Definition of the ballots:** questions, answers, candidates,
- **Codes:** initialisation key, return codes, confirmation and finalisation code,
- **Signature key:** used for signing the produced PDF documents,
- **Encryption key:** used for protecting the document during their transfer to the printing office.

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

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

Attack 2: **Visible manipulation of voting cards to compromise individual verifiability:** inverting the return codes for voting options, inverting the text of questions, exchanging the names of candidates.
This is mitigated by the manual review of a sample of cards.

Attack 3: **Leaking information to the printing service by hiding it in the PDF files:**
This does not have an impact, as the printing service has access to all the data (identity of voters, codes, ballot, encryption key), except for the signing key. Since the printing office is in charge of verifying the signature, being able to create fake signatures would not be an advantage.

Attack 4: **Leaking information to an attacker by having it printed on a voting card:**
The most efficient attack would be to leak the encryption key on a voting card, potentially hiding it by some steganographic method. The attacker could obtain a copy of the encrypted cards while they are transmitted from the canton to the printing office and then decrypt them when the voting card is printed and delivered by mail.
A more straight-forward attack would be to leak codes by adding them to a card.
Adding the codes of a single other card would allow the attacker to break the secrecy of the vote cast with that card.
If the codes of a significant number of other cards can be added to one or few cards, this card could be used change the outcome of the vote (break the *vote correctness*).
The only effective mitigation against this class of attack is a review of the code. Publishing the code would allow for a much more thorough review of the code.

It is important to note that for these attacks to succeed, the attacker may have to compromise some elements of the untrusted part of the voting system. For example, to break vote secrecy, an attacker who has obtained the codes of a victim and breaks into the voting server can compare the codes that are returned to the victim with the leaked codes. While compromising the voting server could be difficult, the trust model mandates that the system be safe, even if all untrusted parts have been compromised.

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 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
- Axcrypt, encrypts the zip archive containing the cards
- Datamatrix postscript libraries, generates datamatrix codes
- xml-signature, to verify signatures, from Swiss Post[1]

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 file or printed on a card (part of Attack 4).

✓ We searched for all the code lines that operate on the signature key. The only ones we found were the lines used for executing the signature and for writing a log about the signature. We did not find any code that would include the key in a voting card. This seems to exclude any attack were the key would be leaked in the PDF file or printed on a card (part of Attack 4).

✓ We analysed the way the code generates the voting cards. The information about the ballot and the voters are stored in XML files. XSL style sheets are used to combine the data into an XML file containing all single cards. That file is then converted to PDF using a standard conversion engine (Apache FOP).
The XSL style sheets generate each card separately, with a *for-each* clause. We did not see any lines 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 (part of Attack 4).

✓ We analysed the style sheets used to position the texts and 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 just by reading them. However, the same stylesheets are used for all cards. A manual inspection of a sample of the cards thus seems to exclude any attack that would do a visible manipulation (Attacks 2 & 3).

✓ 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 that are used for review are the same as the ones that are transmitted to the printing office (Attacks 1 & 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`

✓ 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. While this does not guarantee that effective code injection is impossible, it makes it very improbable.

3.3 Generic analysis of the security of the code

3.3.1 Security issues

Problem 1 - Incorrect parameter handling could lead to command injection

While the incorrect handling of parameters passed to spawned processes could technically lead to command injection, the user-controlled inputs that present that type of risk are handled by the cantons staff who already have access to the secure, offline laptop this application runs on. This reduces the severity of this issue. The residual risk appears to be linked to robustness and maintainability of the source: if a canton were to configure an archive password containing the `"` symbol, the call to the `axcrypt` utility would fail.

The vulnerable code extracts are shown below:

```
// from Impression.Business\Utils\Process\ProcessManager.cs:30
using var process = new System.Diagnostics.Process();
process.StartInfo.Arguments = arguments ?? "";
process.StartInfo.FileName = exe;
...

    process.Start();

// from Impression.Business\Utils\AxCrypt.cs:25
string args = string.Format(_axCryptArgs,
    _axCryptPassword,
    fileOrDir);

// from Impression.Business\Configuration\ConfigurationFribourg.cs:1776
public string EncryptionAxCryptArgs
{
    get
    {
        return _util.StringOrDefault("Impression.Encryption.AxCrypt.Args", "-e -k \"{0}\" -z \"{1}\" -t");
    }
}
```

Recommendation

As per Microsoft's documentation, *if you are not sure how to properly escape your arguments, you should choose* ***ArgumentList*** *over* ***Arguments***.[2]

[2] https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.processstartinfo.argumentlist?view=net-6.0#remarks

Problem 2 - Some sensitive information might end up in logs

In several places, the logs might contain sensitive information. Some examples are provided below.

```
// from Impression.Engine\Commands\Electeur\ElecteurDatContextWriterCommand.cs:22
Logger.Info(correlation, "Add [Electeur: {0}]", electeur.NomPrenom);

// from Impression.Business\Utils\Process\ProcessManager.cs:61
// remember, the arguments provided to AxCrypt in problem 1
// include the password for the archive
Logger.Error(Guid.Empty, ErrorEnum.ProcessManagerRunException, "Arguments [{0}]", arguments);
```

The low severity of the problem takes into account the fact that this application is run on a dedicated laptop.

Recommendation

Sensitive information should be clearly identified, documented and systematically excluded from logs.[3]

Problem 3 - Some XML files are encrypted when packaged

According to the comment in `Impression.Business.Utils.XslTransformer.GetStream`, the `xml` files containing the cantons' templates for voting card and printing reports are encrypted in the package application. It is our understanding that originally the security goal of this measure was to prevent leakage of templates between cantons.

While the files are encrypted, it seems that since each canton runs their own instance of the application on dedicated hardware, the application never needs to have access to multiple templates at the same time.

Recommendation

It seems like a much more efficient and less error-prone approach would consist of having different build profiles, including only the templates of the targeted canton. While this adds some complexity to the build, it would certainly be offset by the removal of the need for canton-specific passwords in the build system, along with the processes required for the management of such secrets. Indeed the property `Impression.Canton.Password` seems to only be used for the decryption of the cantonal templates.

3.3.2 Anti-patterns and bad practices

While not immediately relevant to security, several anti-patterns are present in the code and represent an added hurdle for audit. They would also certainly lead to maintainability issues and broader community push back and criticism.

Problem 4 - Configuration overengineering

For instance, each canton has their own class for configuration, *eg.* `ConfigurationBaleVille`. Each of those configuration files consists of about two thousand lines of single line methods, most of which are either identical between cantons or `throw new System.NotImplementedException()`.

The methods that are implemented and repeated refer, more often than not, to the key of a property defined in a matching configuration file.

[3] https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html

```
return _util.StringOrDefault("Impression.Canton.Password", "");
```

This makes it extremely hard for reviewers and developers alike to follow the flow of the configuration and identify which parameters will actually be applied in which context.

Problem 5 – Complexity through granularity

Another example of such complexity is the excessively fine granularity of the commands, with several commands consisting of a single line. This renders the workflow of the application hard to keep track of.

Problem 6 – Blank inheritance instead of simple check

The fact that several controllers have two different implementations, one suffixed with `Blank` and another with a cantonal suffix also increases the complexity significantly, since it requires verifying which implementation is used in which case, and the main logic code seems to apply in the same manner for each canton, until one digs into the details of each implementation.

Recommendation

It would seem easier to control those features with a boolean setting (*e.g.* DatamatrixEnabled) and have that logic incorporated in the main workflow.

3.4 Deployment

A build ceremony is carried out to compile the code in an observable way. At the end of the ceremony a protocol containing the hashes of the application is signed by all participants.

When the cantons obtain the software, they can verify the hashes against the values on the protocol to make sure that they have the authentic software.

4 Recommendations

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

Deployment:

- Compare the hashes of the software with the hashes obtained during the build ceremony.
- Have the changes in the code reviewed before each deployment. This includes the template files which may be updated for each election.

Operations:

- Review a sample of voting cards before sending them to the printing office. Verify that the texts and codes are identical to a known good source.

We understand that the recommendations regarding deployment and operations are already implemented.

Development: We recommend the following measures in the long run:

- Have the source code published, for general review, ideally with a reproducible build.
- Simplify the code, for easier analysis.
- Do not encrypt the stylesheets, for the same reason.
- Require that the developers of the software use a secure coding standard that is similar to the one required from the developers of the e-voting software at Swiss Post.

5 Conclusions

The source code that we reviewed seems to faithfully translate the received XML data into PDF files of voting cards.

We identified 4 types of attacks that could be mounted in the specific setup in which the application is used. Most of them can be easily excluded by reviewing the code. However, a manual review of a sample of the voting cards is recommended as a complement.

The software is built in a trusted ceremony, which guarantees that it is obtained from the original source code and from a controlled set of dependencies. Any changes in the code should be reviewed and the dependencies must regularly be checked, to verify that they are authentic and up to date.

Any changes in the code should be reviewed and the dependencies must regularly be checked, to verify that they are authentic and up to date.

If these recommendations are applied, we can conclude that we see no explicit danger in using the software. We note however that regarding security and auditability, the code quality is below the one of the Swiss Post evoting system. Moreover, as long as the code is not published, it does not conform the OVE.

In the long run, the source code should be published. Before doing so, the code should be simplified and the unnecessary encryption of the stylesheets removed, to make it easier to analyse. The general security of the code should be increased by applying coding standards similar to the ones used by the developers of the e-voting software.

A Review of version 2.15.1

We received the source code of versions 2.15.0 and 2.15.1 on March 28th and April 3rd. The SHA-256 hash of the file containing the code was the following:

```
b5ffb183e840756c0459b0d2f7de73709ee5bd353e1ef86335721f89ae0e12d1 vcps-source-code-2.15.1.zip
```

We analysed all differences with the code of version 2.13.0. Out of 651 source files we found:

- 40 removed files
- 68 new files
- 126 modified files

The changes are all imputable to the evolution of the software, namely

- A new method for verifying the hashes of third party libraries
- Changes in the layout and text of voting cards
- Changes in names of files and variables
- The splitting of the report of the generation of the voting cards into two separate documents
- Cleaning of the code
- Changes in version strings of the software or its libraries
- Changes in the debugging function

We identified no changes that would impact our specific or generic analysis of the security of the code presented in sections 3.2 and 3.3.

B Review of version 2.16.0

We received access to the source code of version 2.16.0 on July 21st. The SHA-256 hash of the file containing the code was the following:

```
c324c4dbeedf7a7bc7593ef01e4f7608667455a84837527bddf24913f7937e44 vcps-source-code-2.16.0.zip
```

We analysed all differences with the code of version 2.15.1.

Out of 679 source files we found:

- 25 removed files
- 4 new files
- 121 modified files

We reviewed all modified files and confirmed that the changes matched the changes specified in the change log (new evote-config format, layout for Swiss Post, watermark).

We identified no changes that would impact our specific or generic analysis of the security of the code presented in sections 3.2 and 3.3.

C Review of version 2.17.0

We received access to the source code of version 2.17.0 on August 11. The SHA-256 hash of the file containing the code was the following:

```
2df6ea9b5650eb223928e0e6c3355f32099dd7fa434a84f2f03a8cff5ea2e101 vcps-source-code-2.17.0.zip
```

We analysed all differences with the code of version 2.16.1.

Out of 658 source files we found:

- 0 removed files
- 0 new files
- 66 modified files

We reviewed all modified files and confirmed that the changes matched the changes specified in the change log (adapted layout for elections, configuration of memory allocation).

We identified no changes that would impact our specific or generic analysis of the security of the code presented in sections 3.2 and 3.3.