Code Review of Voting Stimmunterlagen Offline

CLIENT	Federal Chancellery	CLASSIFICATION	Public
DATE	March 1, 2023	AUTHORS	Philippe Oechslin, Thomas Hofer
VERSION	1.1	DISTRIBUTION	Federal Chancellery
GIT COMMIT	e0c4968	MODIFICATIONS	Typos, 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 Abraxax to the Canton of St-Gallen. The review was mandated by the Federal Chancellery as part of the examination process according to OEV Article 10 paragraph 1.

The reviewed software (Voting Stimmunterlagen Offline) 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.

It is our understanding that the source code of the software will be published in Spring 2023.

1.2 Execution of the work

The review was carried out in the weeks 4 and 5 of 2023. 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.

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 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 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

The software makes use of following third party software:

- SDelete (signed by Microsoft), securely deletes files on Windows
- Razor library, uses templates to transforms JSON data to HTML

3.1 Analysis specific to the identified attacks

✓ We searched for the code lines that operate on the encryption and signature keys in all application components. The only components that use them are included in the dedicated `CryptoTool` package and executable. We did not find any uses of the keys beyond the expected decryption and signature features. (part of Attack 4).

✓ We analysed the way the code generates the voting cards. The input data is spread over several XML files, which are first aggregated into a single JSON model. The JSON model is then processed and broken down into individual voting cards in HTML format, which is then further transformed into PDF files. The HTML to PDF transformation is very straight-forward and does not perform any change on data. The input to JSON transformation relies on parsers for the dedicated files, and we did not find any evidence of improper mapping of data. The JSON to HTML transformation is driven by templates, relying on a third-party library (RazorLight). The templates are not part of the code, but we were provided with sample templates. Those templates we were given never mix data between voting cards (part of Attack 4).

✓ The templates we were given do not contain any logic to invert mappings in conditional scenarii, and therefore a randomized manual verification on known good datasets would be sufficient to detect errors, as long as the templates themselves have not been tampered with (Attacks 2 & 3).

✓ We analysed how the PDF documents are zipped and encrypted. The application has a preview feature for the generated PDFs. The files can then be downloaded, after having been encrypted. The code of the application uses the same data for preview and as input to the encryption tool. 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).

✓ 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. While the application relies on external libraries for which we did not get access to the source for the intermediary object representation of the data, 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.

✓ The process as a whole is streamlined by the client interface, which ensures continuity of the data handling. The front-end passes the expected elements to the expected command-line utilities.

Currently, Abraxas delivers the software to the Canton of St-Gallen as a package and publishes the hash that can be used to check the integrity of the package. However, there is no way for the Canton to know which source code was used to compile the software.
The only way to be sure that the software is authentic is either to have a trusted build ceremony or to publish the code and have a reproducible build process. We understand the this is the intention of the developers.

3.2 Generic analysis of the security of the code

3.2.1 Security issues

Problem 1 - Outdated framework

The version of the .NET framework targeted for the build (`netcoreapp2.0`) is vulnerable and should be upgraded.

Recommendation

It is the auditors understanding that a migration is in progress. In the shorter term, setting the target version to `netcoreapp2.2` would already be an improvment since version 2.2, while out of support, has no known critical vulnerabilities.

Problem 2 - Frontend dependencies vulnerable

Several front-end dependencies are vulnerable and should be updated or replaced. Overall, 96 vulnerabilities were identified across 54 vulnerable dependencies. Out of those, 28 had vulnerabilities with a `HIGH` risk rating, as summarized in the table below.

Dependency	Version
ansi-html	0.0.7
axios	0.15.3
decode-uri-component	0.2.0
ejs	2.7.4
electron	1.8.8
engine.io	3.1.5
eventsource	0.1.6
glob-parent	2.0.0
hawk	3.1.3
hoek	2.16.3
https-proxy-agent	1.0.0
is-svg	2.1.0
json5	2.2.1
loader-utils	0.2.17
loader-utils	1.1.0

Dependency	Version
netmask	1.0.6
node-forge	0.10.0
postcss	5.2.18
redis	2.8.0
requestretry	1.13.0
scss-tokenizer	0.2.3
socket.io-parser	3.1.3
ssri	5.3.0
tar	2.2.2
trim-newlines	1.0.0
underscore	1.7.0
url-parse	1.4.7
webpack-dev-server	2.11.5
xmlhttprequest-ssl	1.5.5

Recommendation

Dependencies should be kept up to date and regularly scanned for known vulnerabilities (*e.g* using dependency-check[1]).

3.2.2 Code quality review

✓ Coding style and good practices are generally consistant within the backend code.

The problems noted below are some examples of generic software quality issues found throughout the code.

Problem 3 - Variable shadowing

In the frontend, name shadowing (reuse of names within nested scopes) is frequent, making code harder to parse and follow. The list of instances can be found on Sonar, using the rule identifier `javascript:S1117`.

Recommendation

Enforcing consistent coding style contributes to limiting name shadowing.

Problem 4 - Assignments should not occur within more complex expressions

In several places, variable are assigned as part of a more complex expression, and happen as a side effect of the statement. This makes the code harder to maintain and analyse and is generally considered a bad practice.

Recommendation

The assignments should be extracted into their own expressions.

Problem 5 - Ternary operators should not occur within more complex expressions

In several places, ternary operators(`condition ? expression-if-true : expression-if-false`)are used as part of more complete statements. This makes the code harder to maintain and audit, especially when multiple ternary operators are nested.

Recommendation

Expressions should be simplified.

Problem 6 - Variables used outside of declaration scope

While the syntax allows for out-of-scope use of variables declared with the `var` syntax, it is considered a bad practice and should be avoided.

Recommendation

Variables should be declared within the outermost scope that uses them.

[1] `https://owasp.org/www-project-dependency-check/`

4 Recommendations

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

4.1 Immediate recommendations:

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

Deployment:

- Have the software compiled at Abraxas 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. This includes the template files which may be updated for each election.
- Verify that the dependencies are authentic and up to date.

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.

4.2 Other recommendations

We recommend the following measures in the long run:

- Have the source code published, for general review, ideally with a reproducible build.
- 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. In particular, the issues mentioned in subsection 3.2.2, while not an exhaustive list, should be fixed.

5 Conclusions

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

Finally, the software makes use of 3rd party libraries. These must also 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.

Abraxas plans to publish the source code in the near future. Before doing so, the code should be simplified to make it more easy 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.