Security audit of the e-voting backend version 1.4.3

CLIENT	**Swiss Federal Chancellery**	CLASSIFICATION	**Public**
DATE	**July 26, 2024**	AUTHORS	**Philippe Oechslin, Delphine Peter**
VERSION	**1.0**	DISTRIBUTION	**Swiss Federal Chancellery**
GIT COMMIT	**ee62970**	MODIFICATIONS	
STATUS	**Final**		

OS 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

The goal of this audit was to examine the security of the server-side parts of the e-voting systems based on the publicly available end-to-end test system.

Swiss post publishes the end-to-end test system as a set of docker containers and configuration data[1]. This makes it possible to run a complete election event on a local machine.

Obviously, a local end-to-end system does not have all the security controls that exist in the real production environment. But since it uses the same code, it allows testing security aspects that do not depend on the production environment.

Two previous versions of the end-to-end system had already been audited by Objectif Sécurité. Additional tests were conducted on the latest released version (1.4.3).

1.2 Execution of the work

The security audit was conducted in July 2024 from Objectif Sécurité's premises.

All tests were performed on version 1.4.3 of the end-to-end system, which was the latest released version at the time of the audit. The three instances of Secure Data Managers (SDMs) as well as the Data Integration Service (DIS) and the Verifier were run on a Windows virtual machine (VM), while all docker containers were installed on a Linux VM.

Furthermore, our analysis of the end-to-end system was based on the following documentation as published on Swiss Post's GitLab repository[2]:

- Swiss Post Voting System, System Specification, v1.4.1
- Cryptographic Primitives of the Swiss Post Voting System, v1.4.1
- E-Voting Architecture Document, v.1.4.0

Previous versions[3] of the end-to-end system have been audited in 2022 and 2023. Results obtained on these versions are detailed in our previous reports. Note that all tests performed during these previous audits were run again on version 1.4.3 of the system.

[1] https://gitlab.com/swisspost-evoting/e-voting/evoting-e2e-dev

[2] https://gitlab.com/swisspost-evoting/

[3] versions 0.15.2.1 and 1.3.2.1

1.3 Executive summary

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

All sensitive data protected by signatures: We were able to verify that all messages required to be signed according to the specification are indeed signed in the implementation. Furthermore, we intercepted and modified several messages exchanged between the backend components in order to detect potential issues in the signature validation. All our tested resulted in an error raised by the recipient, which detected an invalid signature in the tampered messages.

Client software and parameter integrity validation: The voters can use a published hash to verify that they are using the correct software. This software verifies all parameters, including the public key used for encryption. The verification is done with a hash of all parameters that is stored in the authenticated data of the voter's keystore. Thus, when a voter types in the Start Voting Key that is printed on their material, that key is used both to decrypt a secret that will allow them to vote and to verify that the correct parameters will be used to vote.

Displayed texts not validated by the voting client: Although all voting parameters are properly validated by the voting client through authenticated encryption, the corresponding authenticated data do not include the texts that are actually displayed in the voter's browser. One could thus modify the questions or the voting options shown to the voters in order to influence their choices. Nevertheless, manipulations of the voting options' texts can be detected with the return codes, and the question texts can also be compared with those displayed on the printed material.

No technical flaws detected: We did not discover any vulnerabilities in the backend components that would allow an attacker to manipulate the end-to-end system operations. Some default or weak parameters were found in the containers' configuration, but they are most likely only applied in test environments.

2 Channel Security Signatures

The computational proofs rely on messages' authenticity which is not given in the communication channels. To achieve this property, the cryptographic protocol adds signatures to most messages. These signatures are detailed in section 7 of the System Specification.

We set up the system in a way that we could eavesdrop on all communications and verified the presence of the signatures listed in the specification. The signatures are given in tables 15, 16 and 17 of the System Specification.

2.1 Configuration Phase

	Message Name	Signer	Recipient(s)	Message Content	Context Data
1	ElectionEventContext	Setup Comp.	Online CC$_j$, Tally CC, Auditors, Voting Server	$(p, q, g, \mathsf{seed}, \mathbf{p}, n_{\max}, \psi_{\max}$ $\delta_{\max}, \{\mathtt{vcs}, \mathsf{pTable}, \mathtt{N_E}\}^{\forall \mathtt{vcs}})$	("election event context", ee)
2	CantonConfig	Canton	Setup Comp., Auditors, Tally CC	configuration XML	("configuration")
3	ControlComponentPublicKeys	Online CC$_j$	Setup Comp., Auditors	$(\mathbf{pk}_{\mathrm{CCR}_j}, \pi_{\mathrm{pkCCR},j}, \mathrm{EL}_{\mathrm{pk},j}, \pi_{\mathrm{ELpk},j})$	("OnlineCC keys", j, ee)
4	SetupComponentVerification-Data	Setup Comp.	Online CC$_j$, Auditors	$(\{\mathrm{vc}_{\mathrm{id}}, \mathrm{K}_{\mathrm{id}}, c_{\mathrm{pCC},\mathrm{id}}, c_{\mathrm{ck},\mathrm{id}}\}_{id=0}^{\mathrm{N_E}-1}, \mathrm{L}_{\mathrm{pCC}})$	("verification data", ee, vcs)
5	ControlComponentCodeShares	Online CC$_j$	Setup Comp., Auditors	$(\{\mathrm{vc}_{\mathrm{id}}, \mathrm{K}_{j,id}, \mathrm{Kc}_{j,id}, c_{\mathrm{expPCC},j,id}, c_{\mathrm{expCK},j,id}, \pi_{\mathrm{expPCC},j,id}, \pi_{\mathrm{expCK},j,id}\}_{id=0}^{\mathrm{N_E}-1})$	("encrypted code shares", j, ee, vcs)
6	SetupComponentLVCCAllowList	Setup Comp.	Online CC$_j$	$\mathrm{L}_{\mathrm{lVCC}}$	("lvcc allow list", ee, vcs)
7	SetupComponentEvotingPrint	Setup Comp.	Printing Comp.	evoting print XML	("evoting print")
8	SetupComponentCMTable	Setup Comp.	Voting Server	CMtable[1]	("cm table", ee, vcs)
9	SetupComponentVerification-CardKeyStores	Setup Comp.	Voting Server	**VCks**	("vc keystore", ee, vcs)
10	SetupComponentVoterAuthenticationData	Setup Comp.	Voting Server	$(\mathbf{credentialID}, \mathbf{hAuth})$	("voter authentication", ee, vcs)
11	SetupComponentPublicKeys	Setup Comp.	Online CC$_j$, Tally CC, Auditors, Voting Server	$(\{j, \mathbf{pk}_{\mathrm{CCR}_j}, \pi_{\mathrm{pkCCR},j}, \mathrm{EL}_{\mathrm{pk},j}, \pi_{\mathrm{ELpk},j}\}_{j=1}^{4}, \mathrm{EB}_{\mathrm{pk}}, \pi_{\mathrm{EB}}, \mathrm{EL}_{\mathrm{pk}}, \mathbf{pk}_{\mathrm{CCR}})$	("public keys", "setup", ee)
12	SetupComponentTallyData	Setup Comp.	Auditors, Tally CC	$(\mathbf{vc}, \mathbf{K})$	("tally data", ee, vcs)
13	SetupComponentElectoral-BoardHashes	Setup Comp.	Tally CC	$(\mathrm{hPW}_0, \ldots, \mathrm{hPW}_{k-1})$	("electoral board hashes", ee)

Figure 2.1 Table 15 of the document: messages of the configuration phase

We numbered the messages from 1 to 13. We successfully verified that messages 1 to 11 are signed. Messages 12 and 13 seem to correspond to the files transmitted from the Setup SDM to Verifier and Tally CC. Those files are encrypted, so we could not verify if they contain a signature.

We verified that the CMtable is correctly ordered alphabetically according to the base64 value of its first column.

2.2 Voting Phase

Message Name	Signer	Recipient(s)	Message Content	Context Data
VotingServerEncryptedVote	Voting server	Online CC_j	$(\texttt{E1}, \texttt{E2}, \widetilde{\texttt{E1}}, \pi_{\text{Exp}}, \pi_{\text{EqEnc}})$	("encrypted vote", ee, vcs, $\texttt{vc}_{\texttt{id}}$)
ControlComponentPartialDecrypt	Online CC_j	Online $CC_{j'}$	$(\mathbf{d_j}, \pi_{\text{decPCC},j})$	("partial decrypt", j, ee, vcs, $\texttt{vc}_{\texttt{id}}$)
ControlComponentLCCShare	Online CC_j	Voting Server	$\texttt{lCC}_{j,\texttt{id}}$	("lcc share", j, ee, vcs, $\texttt{vc}_{\texttt{id}}$)
VotingServerConfirm	Voting server	Online CC_j	$\texttt{CK}_{\texttt{id}}$	("confirmation key", ee, vcs, $\texttt{vc}_{\texttt{id}}$)
ControlComponenthlVCC	Online CC_j	Online $CC_{j'}$	$\texttt{hlVCC}_{\texttt{id},j}$	("hlvcc", j, ee, vcs, $\texttt{vc}_{\texttt{id}}$)
ControlComponentlVCCShare	Online CC_j	Voting Server	$\texttt{lVCC}_{\texttt{id},j}$	("lvcc share", j, ee, vcs, $\texttt{vc}_{\texttt{id}}$)

Figure 2.2 Table 16 of the document: messages of the voting phase

We observed all messages with their respective signatures.

2.3 Tally Phase

	Message Name	Signer	Recipient(s)	Message Content	Context Data
1	ControlComponentVotesHash	Online CC_j	Online $CC_{j'}$	$\texttt{hvc}_j$	("voteshash", j, ee, bb)
2	ControlComponentBallotBox	Online CC_j	Tally CC, Auditors	$(\{\texttt{vc}_{j,i}, \texttt{E1}_{j,i}, \widetilde{\texttt{E1}}_{j,i}, \texttt{E2}_{j,i}, \pi_{\text{Exp},j,i}, \pi_{\text{EqEnc},j,i}\}_{i=0}^{N_E-1})$	("ballotbox", j, ee, bb)
3	ControlComponentShuffle	Online CC_j	Online $CC_{j'}$, Tally CC, Auditors	$(\mathbf{c}_{\text{mix},j}, \pi_{\text{mix},j}, \mathbf{c}_{\text{dec},j}, \pi_{\text{dec},j})$	("shuffle", j, ee, bb)
4	TallyComponentShuffle	Tally CC	Auditors	$(\mathbf{c}_{\text{mix},5}, \pi_{\text{mix},5}, \mathbf{m}, \pi_{\text{dec},5})$	("shuffle", "offline", ee, bb)
5	TallyComponentVotes	Tally CC	Auditors	$(L_{\text{votes}}, L_{\text{decodedVotes}}, L_{\text{writeIns}})$	("decoded votes", ee, bb)
6	TallyComponentDecrypt	Tally CC	Auditors	evoting decrypt XML	("evoting decrypt")
7	TallyComponentEch0222	Tally CC	Auditors	eCH 0222 XML	("eCH 0222")
8	TallyComponentEch0110	Tally CC	Auditors	eCH 0110 XML	("eCH 0110")

Figure 2.3 Table 17 of the document: messages of the tally phase

We observed the signatures of the first three messages, as well as the signatures included in the three files generated by the Tally CC (messages 6 to 8 in Table 17).
Again, messages 4 and 5 are transmitted to the Verifier through an encrypted file generated by the Tally CC. We were thus unable to verify if a signature is present in this file.

3 Creating error situations

We created several error scenarios to verify the correct reaction of the system.

These tests required us to intercept Artemis messages exchanged between the control components and the voting server through the message broker. Since these packets are not encrypted in the test end-to-end configuration, we could modify the intercepted messages in order to trigger the errors detailed below.

3.1 Casting a vote with an invalid partial choice code

Before calculating return codes for a submitted vote, the control components first validate that the received partial choice codes are in a defined *allow-list* enumerating all possible valid inputs. This guarantees that the control components do not perform calculations with manipulated parameters.

We used our own voting client to submit a vote with an invalid partial choice code. As a result, we did not receive any answer from the voting server. The latter server did not reply either when we subsequently tried to submit a correct vote with the same voting card.

The behaviour of the system is correct in the sense that a vote with invalid pCCs will be detected when the control components run the CreateLCCShare algorithm, during the calculation of the return code. Subsequent submissions of a vote are blocked because the voting card has been recorded in the list L_{decPCC} of cards for which a pCC has already been decrypted.

We checked the logs of the end-to-end system to verify whether the lack of answer was due to a correctly intercepted error or to a crash of the program.

We indeed observed an error in all control components' logs, which was properly caught:

```
 $ docker logs control-component-1
Caused by: java.lang.IllegalStateException: Failed to execute exactly once command
        at ch.post.it.evoting.controlcomponent.ExactlyOnceCommandExecutor.process(ExactlyOnceCommandExecutor.java:94)
        ~[!/:1.4.3.0]
        [...]
        ... 10 common frames omitted
Caused by: java.lang.IllegalStateException: The partial Choice Return Codes allow list does not contain the partial
Choice Return Code.
       at ch.post.it.evoting.controlcomponent.protocol.voting.sendvote.CreateLCCShareAlgorithm.lambda$createLCCShare$0(Cre-
ateLCCShareAlgorithm.java:119) ~[!/:1.4.3.0]
```

3.2 Modifying signed parameters

As mentioned in chapter 2, all messages exchanged between the control components, the Online SDM and the voting server are signed. We tried to intercept and modify several of these messages in order to verify if their signatures are properly validated by the recipients. The following scenarios were tested:

1. **Request CC Keys :** In the configuration phase, the control components send their public keys to the Online SDM over HTTP. These public keys are used during the voting phase to encrypt the partial Choice Return Codes. We intercepted the HTTP response containing those keys and modified the first control component's key.

This manipulation is detected by the Setup SDM when the Electoral Board is constituted. The following error is displayed in the Setup SDM's logs:

```
java.util.concurrent.CompletionException: ch.post.it.evoting.domain.InvalidPayloadSignatureException: Signature of
payload ControlComponentPublicKeysPayload is invalid. [electionEventId: F5654BC3A3D4345FAADB32A5F5217346, nodeId: 2]
[...]
Caused by: ch.post.it.evoting.domain.InvalidPayloadSignatureException: Signature of payload
ControlComponentPublicKeysPayload is invalid. [electionEventId: F5654BC3A3D4345FAADB32A5F5217346, nodeId: 2]
        at ch.post.it.evoting.securedatamanager.setup.process.constituteelectoralboard.
        ControlComponentPublicKeysConfigService.validateSignature(ControlComponentPublicKeysConfigService.java:112)
        [...]
```

Note that the signatures of the public keys are validated by the Setup SDM, and not by the Online SDM which requests them. Thus, the Online SDM indicates that the keys have been successfully retrieved (see figure 3.1). The error is only displayed when one tries to subsequently constitute the Electoral Board with the Setup SDM (see figure 3.2).
Even though the message displayed by the Online SDM might be misleading, we did not find a way to complete the configuration phase with a tampered control component encryption key.

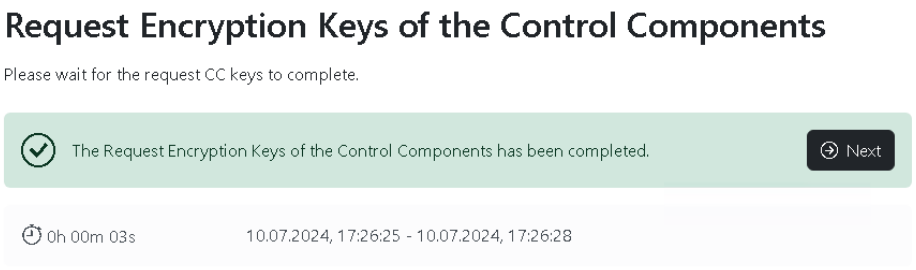

Figure 3.1 Success message displayed by the Online SDM when a control component's public key has been tampered with

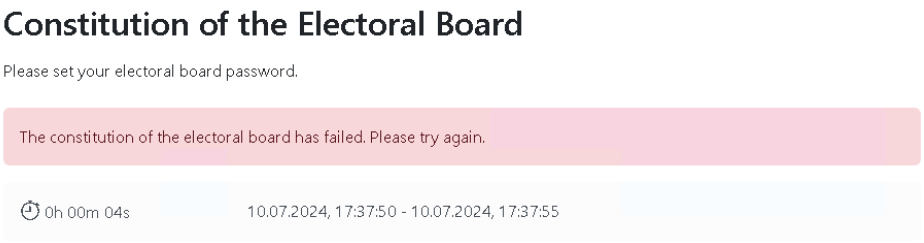

Figure 3.2 Error message displayed by the Setup SDM upon Electoral Board constitution

Ideally, the Online SDM could verify the signatures and prevent the tampered data to be delivered to the Setup SDM.

2. **PartialDecryptPCC :** When a vote is submitted, each control component runs the PartialDecryptPCC algorithm on the received encrypted vote and sends the partially decrypted PCCs along with exponentiation proofs to the other CCs in order to complete decryption. We modified the result of the PartialDecryptPCC algorithm (`exponentiatedGammas`) sent from CC1 to the other CCs through the voting server.

   The following error was observed in the receiving CCs' logs:

```
org.springframework.jms.listener.adapter.ListenerExecutionFailedException: Listener method 'public <T,U> void
ch.post.it.evoting.controlcomponent.MessageHandler.onMessage(jakarta.jms.Message) throws jakarta.jms.JMSException'
threw exception
[...]
Caused by: ch.post.it.evoting.domain.InvalidPayloadSignatureException: Signature of payload
ControlComponentPartialDecryptPayload is invalid. [contextIds:
ContextIds[electionEventId=F6B099CAC1C51075E70B780A4BA1DB27, verificationCardSetId=02117EB17A4A01E86FDD0DB9F3B7E39D,
verificationCardId=6CFE22CC190776263D3DED6BAF2D39FD]]
[...]
```

3. **CreateLVCCShare :** When calculating the finalisation code, the control components exchange their partial results (from CreateLVCCShare) and check that the combination of the results is in an allow-list (VerifyLVCCHash). Only then do they reveal to the voting server the information needed to calculate the finalisation code.

   We intercepted the traffic between the voting server and a control component and modified the message containing the confirmation key CK_{id}, which is used as input for the CreateLVCCShare algorithm. Note that this message is signed by the voting server.

   The modification is detected, and no finalisation code is generated. There is a timeout. The control component that received the manipulated value has the following log entry:

```
org.springframework.jms.listener.adapter.ListenerExecutionFailedException: Listener method 'public <T,U>
void ch.post.it.evoting.controlcomponent.MessageHandler.onMessage(jakarta.jms.Message) throws
jakarta.jms.JMSException' threw exception
[...]
 Caused by: java.lang.IllegalStateException: The signature is not valid. [requestMessageType:
 ch.post.it.evoting.domain.voting.confirmvote.VotingServerConfirmPayload, correlationId:
 462684ea-ea39-4722-8762-a6672299ba4b, nodeId: 1]
 [...]
```

   After tallying, we can confirm that the vote is not part of the votes that have been tallied.

4. **VerifyLVCCHash:**

   As mentioned in the previous paragraph, the control components verify that the long Vote Cast Return Codes calculated by all four control components (in the CreateLVCCShare algorithm) are valid, according to a pre-defined allow-list. This is the VerifyLVCCHash algorithm.

   We intercepted the VerifyLVCCHash request sent to CC4 and modified the hash computed by CC1.

This modification is detected, and no finalisation code is generated. There is a timeout. CC4, which received the manipulated request, has the following log entry:

```
org.springframework.jms.listener.adapter.ListenerExecutionFailedException: Listener method 'public <T,U> void
ch.post.it.evoting.controlcomponent.MessageHandler.onMessage(jakarta.jms.Message) throws jakarta.jms.JMSException'
threw exception
[...]
Caused by: java.lang.IllegalStateException: The signature is not valid. [requestMessageType:
ch.post.it.evoting.domain.voting.confirmvote.ControlComponenthlVCCRequestPayload, correlationId:
88dceae1-2dcd-4935-8526-e1490ac0b275, nodeId: 4]
[...]
```

5. **MixDecOnline:** In the MixDecOnline algorithm run in the tally phase, the control components successively compute partially decrypted and shuffled votes, as well as the corresponding proofs. We intercepted the message containing the MixDecOnline result computed by CC1 and transmitted to CC2, and modified the decryption proofs.

This modification is detected and the Mixing phase fails. CC2, which received the manipulated message, has the following log entry:

```
Unable to consume message: Signature of payload ControlComponentShufflePayload is invalid. [nodeId: 1,
electionEventId: 20B4AE0C3D63046E6CA1DAAF1D59EFFF, ballotBoxId: D87E813E98501A755C434E7F163CDF27]
```

3.3 Removing signatures

Since all our attempts to modify signed parameters resulted in errors indicating that the message's signature is invalid (see section 3.2), we tried to remove signatures in several messages in order to verify if it is indeed required for the recipient to proceed with the expected operation.

In this experiment, we tampered with the following messages:

- **Configuration phase:** we removed the signature of the control components' public keys sent to the Online SDM (message 3 in figure 2.1)
- **Voting phase:** we removed the signatures of all messages listed in figure 2.2
- **Tally phase:** we removed the signature of the MixDecOnline result computed by a control component (message 3 in figure 2.3)

In each case, the modification is detected and the receiving party refuses to process the tampered message. A `NullPointerException` is shown in the recipient's logs when the message is parsed:

```
org.springframework.jms.listener.adapter.ListenerExecutionFailedException: Listener method 'public <T,U> void
ch.post.it.evoting.controlcomponent.MessageHandler.onMessage(jakarta.jms.Message) throws jakarta.jms.JMSException' threw
exception
[...]
Caused by: com.fasterxml.jackson.databind.exc.ValueInstantiationException: Cannot construct instance of
`ch.post.it.evoting.evotinglibraries.domain.signature.CryptoPrimitivesSignature`, problem:java.lang.NullPointerException
[...]
Caused by: java.lang.NullPointerException: null
       at com.google.common.base.Preconditions.checkNotNull(Preconditions.java:906) ~[guava-33.2.1-jre.jar!/:na]
       at ch.post.it.evoting.evotinglibraries.domain.signature.CryptoPrimitivesSignature.<init>
       (CryptoPrimitivesSignature.java:30) ~[e-voting-libraries-domain-1.4.3.0.jar!/:1.4.3.0]
```

3.4 Creating inconsistency in the tally phase

As detailed in the 4th experiment described in section 3.2, we triggered an error in the `VerifyLVCCHash` algorithm by modifying one of the hash shares that are exchanged between the control components ($hlVCC_{id}$) for validation.

The control component that receives the modified hash will not register the vote as confirmed in the ballot box, whereas the three other components will.

In our previous audit, which was conducted on version 1.3.2.1 of the end-to-end system, this inconsistency led to the following error in the second control component's logs:

```
The initial ciphertexts vector and verifiable shuffles ciphertexts vector must have the same size.
```

In order to avoid the size differences in the ciphertext vectors handled by the control components during mixing, we conducted a new test on the latest version of the system (1.4.3): the same manipulation of the `VerifyLVCCHash` request was performed on four different votes, once for each control component. In this way, they would all have the same number of votes, since each control component would have discarded a different one. This situation would not be detected by the consistency check that compares the number of initial votes with the number of shuffled votes.

We indeed did not obtain the above error message in the control component's logs upon mixing. However, the voting server still detects the manipulation and outputs the following error, which makes the tally phase fail:

```
Unable to consume message: All Control Component Votes Hash Payloads must have the same encrypted confirmed votes hash.
[...]
Caused by: java.lang.IllegalArgumentException: All Control Component Votes Hash Payloads must have the same encrypted confirmed votes hash.
        at com.google.common.base.Preconditions.checkArgument(Preconditions.java:145)
```

This is due to a new validation added to the MixOnline algorithm since our previous audit: each control components now sends the hash generated from the list of encrypted confirmed votes it registered (GetMixnetInitialCiphertexts) to the voting server before starting mixing. The voting server then returns an error if those hashes differ, and the mixing fails. Thus, we could not reproduce the scenario observed in our previous audit, because the tally phase fails before comparing the ciphertext vectors.

It should be noted that the voting server is not considered as trusted, so the other parties involved in the tally phase should perform a similar check on the hashes to detect such manipulations, and not solely rely on the voting server. We were not able to bypass the voting server's verification in order to confirm that the CCMs actually perform a similar check, but the system documentation states that they do.

4 Voting client security and Authenticated Data

4.1 Security of the JavaScript code

When the voter accesses the home page of the voting server, four JavaScript files are loaded:

- runtime.js
- polyfills.js
- crypto.ov-api.js
- main.js

They contain the code of the voting client. For each JavaScript file, an integrity tag containing a hash of the file is provided, e.g.:

```
<script src="runtime.js"
  integrity="sha384-YwZU+M0RWirxGUXpR82bV38PfKIXCa1y7oI8xk3Xo3I+rHG5znVHx9+02CX0YUS6">
  [..]
</script>
```

Voters have the possibility to compare this hash to known good hashes on the web sites of the cantons. They can trust their browsers to verify the hash of the files or download the files and verify the hash with a tool they trust.

Furthermore, they can compare the content of the HTML file that loads the hashes with a known good copy on the web sites of the cantons.

If they trust the known good sources, the voters can thus be sure that they are being served the correct voting client from the voting server.

4.2 Authenticated data

Once the voting client is loaded, and after the voter has authenticated, the client receives all the necessary parameters and data from the voting server.

One key parameter is the private key that the voter needs to carry out the cryptographic protocol. It is received in a keystore that is encrypted with a key derived from the Start Voting Key (SVK) which is printed on the voting material. The voter can thus only vote if they type in the correct SVK.

Additionally to the private key, the keystore also carries a hash of all cryptographic parameters and of the semantic of the ballot. The voting client verifies that the hash matches a hash calculated with the parameters that were actually received by the client. The list of parameters that are fed into the hash are given in the GetHashContext algorithm of the protocol specification as seen in figure 4.1.

If the hashes do not match, the voting client refuses to vote. We verified this behaviour in the source code of the client and during our online test.

If the voter thus verifies the integrity of the voting client downloaded from the home page, they will know that the voting client will only proceed if none of the parameters received by the server have been tampered. The more inclined voters may even calculate the hash of the parameters and verify it themselves.

Algorithm 3.11 GetHashContext

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
Election event ID $\mathtt{ee} \in (\mathbb{A}_{Base16})^{l_{\mathtt{ID}}}$
Verification card set ID $\mathtt{vcs} \in (\mathbb{A}_{Base16})^{l_{\mathtt{ID}}}$
Primes mapping table $\mathsf{pTable} \in (\mathcal{T}_1^{50} \times ((\mathbb{G}_q \cap \mathbb{P}) \setminus g) \times {\mathbb{A}_{UCS}}^* \times \mathcal{T}_1^{50})^n$ ▷ pTable is of the form $((\mathrm{v}_0, \tilde{\mathrm{p}}_0, \sigma_0, \tau_0), \ldots, (\mathrm{v}_{n-1}, \tilde{\mathrm{p}}_{n-1}, \sigma_{n-1}, \tau_{n-1}))$
Election public key $\mathbf{EL_{pk}} = (\mathrm{EL}_{\mathrm{pk},0}, \ldots, \mathrm{EL}_{\mathrm{pk},\delta_{\max}-1}) \in \mathbb{G}_q^{\delta_{\max}}$
Choice Return Codes encryption public key $\mathbf{pk}_{\mathrm{CCR}} = (\mathrm{pk}_{\mathrm{CCR},0}, \ldots, \mathrm{pk}_{\mathrm{CCR},\psi_{\max}-1}) \in \mathbb{G}_q^{\psi_{\max}}$

Operation: ▷ As indicated in section 1.4 we flatten the lists and avoid nested structures

1: $h \leftarrow ()$
2: $h \leftarrow (h, \text{“}\mathtt{EncryptionParameters}\text{”}, p, q, g)$
3: $h \leftarrow (h, \text{“}\mathtt{ElectionEventContext}\text{”}, \mathtt{ee}, \mathtt{vcs})$
4: $h \leftarrow (h, \text{“}\mathtt{ActualVotingOptions}\text{”}, \mathsf{GetActualVotingOptions}(()))$ ▷ See 3.4
5: $h \leftarrow (h, \text{“}\mathtt{EncodedVotingOptions}\text{”}, \mathsf{GetEncodedVotingOptions}(()))$ ▷ See 3.3
6: $h \leftarrow (h, \text{“}\mathtt{SemanticInformation}\text{”}, \mathsf{GetSemanticInformation}())$ ▷ See 3.5
7: $h \leftarrow (h, \text{“}\mathtt{CorrectnessInformation}\text{”}, \mathsf{GetCorrectnessInformation}(()))$ ▷ See 3.6
8: $h \leftarrow (h, \text{“}\mathtt{ELpk}\text{”}, \mathrm{EL}_{\mathrm{pk},0}, \ldots, \mathrm{EL}_{\mathrm{pk},\delta_{\max}-1})$
9: $h \leftarrow (h, \text{“}\mathtt{pkCCR}\text{”}, \mathrm{pk}_{\mathrm{CCR},0}, \ldots, \mathrm{pk}_{\mathrm{CCR},\psi_{\max}-1})$
10: $d \leftarrow \mathsf{Base64Encode}(\mathsf{RecursiveHash}(h))$ ▷ See crypto primitives specification

Output:

The digest $d \in {\mathbb{A}_{Base64}}^{l_{\mathtt{HB64}}}$ ▷ $l_{\mathtt{HB64}}$ is the character length of the Base64 encoded hash output

Figure 4.1 Hash Context: List of parameters that are included in the hash which is verified when decrypting the private key of the voter

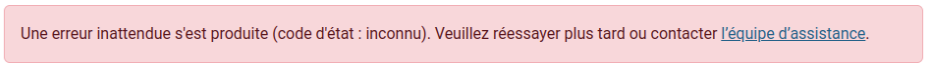

Figure 4.2 Error displayed when one of the parameters of the Hash Context has been tampered with

Although the authenticated parameters include the actual voting options (i.e. yes, no, list and candidate names) and the semantic information (text of the questions) it does not include the text that is actually displayed to the voters. This text, in all four national languages, is indeed part of a structure called `ballotTexts`, which is not authenticated.
The voters can detect manipulation of the text of voting options by comparing the return codes with the codes printed on their voting material. They can detect the manipulation of the questions (e.g. 'do you accept' vs 'do you reject') by reading the questions on their printed material. Still, the voting client would be more secure if the hash of the ballot texts was also included in the information that is authenticated along with the other voting parameters.

5 Technical security tests

5.1 Analysis of Open Ports

The voting server and the four control components have an open port for debugging. The port gives access to the Java Debug Wire Protocol (JDWP). This protocol can be abused to execute arbitrary command on the machine.

The debugging port is activated from the command line when starting the application with the parameter:

```
-agentlib:jdwp=transport=dt_socket,address=*:<port>,server=y,suspend=n
```

This parameter is set in the common configuration file (`docker-compose.common.yml`) for the five docker containers mentioned above.

We assume that this parameter is not set in production environments.

Other than JDWP, which is valid in a test environment, we did not discover any unnecessarily open ports.

5.2 Analysis of exposed REST endpoints

The voting server offers a set of REST endpoints that are used to interact with the end-to-end system's backend. For example, the following endpoint allows to retrieve all used voting cards for a given election event ID:

```
/vs-ws-rest/api/v1/voting-card-manager/used-voting-cards/election-event/{electionEventId}
```

Unlike the previously tested version of the end-to-end system, the backend API is no longer split over multiple microservices representing the different phases of the process and running in separate containers. Now, the entire backend API is handled by the `voting-server` container.

Although the API endpoints are no longer listed in the application logs on startup, we could retrieve them from the source code.

We analysed the lists of endpoints available to check if there was a service that would give access to protected information or functionality, maybe for debugging purposes. We did not identify any endpoint that did not seem to be legitimate.

Note that the control components do not expose any REST endpoint. They connect to an ActiveMQ message broker they can exchange messages with.

5.3 Other configurations

We noticed some default or weak configurations applied to the end-to-end system's docker containers:

- Default credentials are configured for the Artemis ActiveMQ administration console
- Weak credentials are configured for the database used by the voting server and the control components
- Artemis messages are exchanged in cleartext between the voting server and the control components (through the message broker)
- HTTP traffic is also sent in cleartext between the Online SDM and the voter portal, as well as between the voter portal and the voting server

These configurations are acceptable in a test environment but should not be used in production. We assume that production systems use more secure parameters.

5.4 Scanning for vulnerabilities

The docker containers only contain the minimal set of programs needed for e-voting. We scanned the containers with a vulnerability scanner and also checked the version of some of the software manually.

All software seems to be up-to-date with no known vulnerabilities.

6 Conclusions

We conducted various tests on the backend components of the end-to-end e-voting system. This audit did not reveal any vulnerability in the implementation of the backend systems.

In particular, all sensitive data exchanged between the various backend components in the three phases (configuration, vote, tally) are signed by the sender and the latter signature is properly validated by the recipient. This greatly reduces the possibilities for a third party to tamper with exchanged information.

Moreover, the voting client allows voters to verify that they are using the correct software and parameters. However, we noticed that the texts that are actually displayed to the voters are not validated by the voting client. Voters still have the possibility to manually validate the integrity of those texts based on their printed material.