Addendum on the Swiss Post e-Voting System

Thomas Haines, Olivier Pereira, Vanessa Teague
thomas@hainest.com, olivier.pereira@uclouvain.be, vanessa.teague@anu.edu.au

November 21, 2022

Contents

1 Summary

In summing up our first round of review [1] we stated:

> The project and system are highly complex and, for the moment, the review process is adding to the list of open questions rather than reducing it. There are, at present, significant gaps in the protocol specification, verification specification, and proofs. We continue to find issues which we had not noticed in previous iterations. And, as several parts of the system documentation remain missing, our evaluation could not consider the system in full.

Since the first round of review concluded some major issues have been addressed satisfactorily, as we shall detail, which has increased the maturity of the system significantly; this is particularly true of the system specification. However, our summation from round one remains essentially adequate as a current description of the verifier specification, proofs, and architecture documents. For many of the outstanding issues, Post already has plans to address them in future releases but these plan have yet to be executed.

As we have started to look more carefully at the code, we have noticed areas where the code is incompatible with the specification as we shall detail below; while the lack of alignment between code and specification is the source of several severe problems it remains unclear how wide spread it is.

2 Update on Prior Issues in Scope 1

In the subsections below we will provide updates on issues we have raised in previous reports.

2.1 Alignment between the security model required by the Ordinance and the one used in the security proofs

We have expressed significant concerns about the alignment between the Ordinance and system. These issues are primarily concerned with the computational proofs document; this document has not been updated since our draft report in round two of the examination and so our concerns remains essentially unchanged. We highlight a few issues below.

2.1.1 Role of auditors

In our draft examination report in round 2, we expressed concerns about the roles of auditors, which we see as being more active and trusted than what the ordinance allows. More precisely:

- Article 2.2 of the Ordinance Appendix does not authorize any communication channel from the auditors of from their technical aids to any other

system participant. However, Section 4.4 of the Specification document and Section 11.4 of the Proof document indicate that the protocol requires the verifier to perform the VerifyConfigPhase algorithm at the end of the SetupVoting algorithm and to report an invalid execution to the Setup Component, which requires a communication channel.

- Art. 2.9.1.1 and 2.9.3.1 of the Ordinance appendix indicate that the auditors and their technical aids must be considered untrustworthy for individual verifiability and for privacy. However, Section 11.4 of the Proof document indicate that the auditors and their technical aids are considered to be trusted during the Setup phase, which is also reflected in the security definitions of the same document (e.g., VerifyConfigPhase is assumed to be performed honestly in the vote privacy experiment, even though it is executed by the auditor technical aid).

We feel that it is an important requirement from the Ordinance to require that the auditors should be treated as untrustworthy components (for individual verifiability and for privacy) and should perform their task essentially as external components, that is, without sending any message to protocol participants. An audit largely brings trust from the understanding that it is performed by independent people, external to the system, who won't collude or simply make the same honest mistakes as the system operators. But this independence also means that the system operator cannot rely, for running an election, on the auditors to be trustworthy: they are independent components, out of the control of the system operators. We then believe that it is sound to keep a clear separation of duties: the system operators need to be able to run a successful election and produce evidence of the success of the election independently of the auditors, and the auditors need to be able to verify the election independently of the system operators.

Of course, it may be convenient to feed the auditors with data during an election: it may lead to faster conclusions, and it may also be good to have an informal feedback from them before the election is complete. We understand that this is authorized by the Ordinance as well.

But the current description of the system goes much further than that: it states in Section 11.4 of the Proof document that

> The cantons run the setup component and the verifier [i.e., the auditor technical aid] in a similar, secure offline environment. Thus, we consider the verifier and the setup component equally secure and trustworthy in the configuration phase.

So, the auditor technical aid:

- *cannot* be running in an environment that is not controlled by the cantons, which brings concerns regarding its independence from the election process;

- *cannot* be considered untrustworthy during the setup/configuration phase, because the security proofs that are given assume that the verification is performed by a trusted component.

- *cannot* operate without a communication channel to other system participants, because an election won't complete unless they report that the VerifyConfigPhase algorithm completed successfully.

Our proposal to address these issues would be to merge the verifier and the trusted setup component: as far as we can see, this would remove all the non-conformities listed above, and would also be consistent with the reality of having the current verifier and setup component being operated by the cantons, in a similar secure offline environment.

We also feel that it would make sense to ask the control components to run the the VerifyConfigPhase algorithm on their side. It certainly is a time consuming operation, but a very important one, and one that can run before the beginning of the election, when time constraints appear to be limited.

The auditors and their technical aids should then be separate components.

2.1.2 Roles of tally control component and electoral board

In our draft examination report in round 2 we said

> There is also an important discrepancy between the threat model given in Table 1 and Table 5 of the Proof document and the one used in the privacy definition (Def. 15) of the same document. (A very similar point was already raised in our March 2022 report. Many changes were made since then, but this difficulty isn't solved, nor addressed in the responses we received.)
>
> Table 1 of the Proof document indicates that the Tally Control Component and the Electoral board are part of the "Untrustworthy System", and Table 5 of the same document confirms that the "Untrustworthy System" must be considered untrustworthy regarding voting secrecy, which is aligned with the Ordinance.

Our understanding is that Post will address this in the December release.

2.1.3 Authentication issues

Authentication issues are one of the more positive stories from this examination. Early versions of the system had weak authentication which allowed exploitable attacks, Post has already implemented authentication which seems suitable for the system. The one remaining concern which has not been addressed is that of messages being dropped without detection. The completeness checks in the auditor do not catch this.

We have had some discussion with Post about how to address the remaining concerns here and hope to see progress in the December release.

2.1.4 Lack of zero-knowledge proofs for key generation

We had complained in [1] that a lack of zero-knowledge key generation allowed attacks which were in scope of the ordinance but not the definitions in the computational proof document. This issue has since been addressed by adding the missing proofs.

2.2 Missing elements in the verifier specification

Missing elements in the verifier specification remains one of the most significant concerns for us. The checks in the verifier are critical for individual verifiability, universal verifiability and privacy. There has been some progress in this area but there are still critical deficiencies.

pTable Previously we stated that

> The pTable needs to generated or verified to ensure a consistent view with the information sent to the print office.

This has been substantially mitigated by changes in the create vote algorithm, and corresponding verifications, to include the pTable in the auxiliary information going into the (zk-)proofs. These changes ensure that if the auditors, CCRs and voter's device don't agree on the pTable then the proofs will not verify. We will review these changes in more details once the other issues in the verifier specification have been resolved.

Correct decoding Previously we stated that

> The correct decoding of the primes into voting options needs to be verified.

this step has now been added to the specification.

Auditors use of the tool Previously we stated that

> Moreover, all steps performed by the auditor with respect to the verifier UI needs to be checked. For example, verifying that the number of voter cards issued matched the number of eligible voters.

This remains the chief issue with the verification specification. The ordinance considers a human auditor with technical aids; the verifier specification deliberately removes this distinction

> we will no longer distinguish between auditors and their technical aid; we refer to the verifier as both the auditor and the software used by this auditor and assume that the auditor and the technical aid are trustworthy.

Attempting to remove this distinction is problematic because the auditor's (correct) use of the technical aid is critical to security. The verifier specification needs to describe how the auditor uses the technical aid otherwise there are potential attacks.

Data format and transmission Previously we stated that

> The verifier specification does not adequately detail the structure of the information logged and how that information is transmitted and received .

This has been increasingly resolved by the presence of example files in the verifier implementation. The strength of the authentication mechanisms seems to suffice to ensure that the exact transmission is no longer particularly relevant but we would like to check this further.

2.3 Technical issues in the security definitions and proofs

In our previous reports, we listed a number of open issues with the security definitions and proofs. There have been many important changes and improvements since then.

But these changes, while addressing some of the issues, also introduced new challenges. We elaborate on those of them that we mentioned in our draft examination report, and for which we feel that further discussions might be useful.

Handling inconsistent states In our March 2022 report, we wrote:

> The games in the formal security definitions restrict the adversary's action far more than the system does. This means attacks may be missed. [...] The central issue is that games do not model the ability of the adversary to control the execution flow of the protocol which in the real system it has significant ability to do through a corrupt voting server.

Section 12 of the Proof document makes an important step in order to address this issue, by introducing a new protocol for handling inconsistent views of the confirmed votes. The abstract description of this protocol is plausible. However, the protocol itself is missing in the Specification document, in the Verifier document, and it is absent from the Proof document as well. As such, it is impossible to assess whether the right votes would be included in the tally. We elaborated on this issue in our draft examination report. This issue is still open.

Unbounded extractors Our March 2022 report discussed challenges with the extraction of witnesses from proofs. The Proof document has been modified in order to address these questions and, in particular, Section 13.2 introduced

a strategy of using inefficient extractors that can decrypt ciphertext without knowledge of any decryption key. In essence, this is not a problem. But it may become one, for instance when challengers running an inefficient extractor are becoming part of a reduction to a problem that is only hard to solve by efficient adversaries. And the current security proofs do not offer any indication on why such problems do not occur.

Following interactions with Swiss Post based on our draft examination report, we would like to elaborate on this concern using one specific example, taken in Lemma 2 from the Proof document.

Figure 23 defines a security game between an adversary and a challenger. This challenger is not explicitly named in the proof, so let us call it $\mathcal{C}_0$.

Looking at the proof, we see that Game vc.1 defines a new challenger $\mathcal{C}_1$ that is identical to $\mathcal{C}_0$ except that it runs the Extract function that does not run in polynomial time (as explained in Sec. 13.2 of the same document).

Game vc.2 goes the same way: it defines a new challenger $\mathcal{C}_2$ that is identical to $\mathcal{C}_1$, except for some extra checks. More precisely, $\mathcal{C}_2$ takes two ciphertexts that have been provided by the adversary together with a proof that they encrypt the same message, uses the Extract function to compute these messages, and aborts if they are different even though the proof checks.

It is claimed that "the Adversary can only notice the extra check if it can generate a plaintext equality proof for an invalid statement without the honest control component noticing" and a reduction $\mathcal{B}_2$ is outlined, that submits the unsound ZK proof to a proof soundness challenger. This argument follows the pattern of a "transition based on failure events" in Shoup's terminology.[1]

Let us try to make it more explicit. We call S_i the event that there is a $\mathsf{bad}_{\mathsf{vc}}$ event occurring as a result of an interaction between $\mathcal{A}$ and $\mathcal{C}_i$, and F the event that the plaintext equality proof provided by the adversary passes verification for two ciphertexts that encrypt different messages.

We observe that $S_1 \wedge \neg F \Leftrightarrow S_2 \wedge \neg F$: indeed, the behaviors of $\mathcal{C}_1$ and $\mathcal{C}_2$ only differ when F happens, and we exclude F. As a result, Shoup's difference lemma shows that $|\Pr[S_1] - \Pr[S_2]| \leq \Pr[F]$. Until now, we just tried to make more explicit what we think is written in the proof.

The challenging part comes when the proof concludes by bounding $\Pr[F]$ by claiming that it can be bounded thanks to a reduction $\mathcal{B}_2$ that uses the plaintext equality proof provided by $\mathcal{A}$ and submits it as a fresh proof that breaks the simulation soundness security.

As far as we can see, the obvious way of defining $\mathcal{B}_2$ would be to let it run an interaction between $\mathcal{A}$ and $\mathcal{C}_1$ and submit the proof that comes out of this interaction to the simulation soundness adversary (our guess is that it is what the proof suggests to do, and this is also the strategy that Shoup suggests as the "usual" one). However, as such, this strategy does not work here because $\mathcal{B}_2$ would not run in PPT (because of $\mathcal{C}_1$), which would make it useless in the context of the simulation soundness security game that is only defined against PPT adversaries.

[1] We follow p.2 of `https://www.shoup.net/papers/games.pdf`.

So, we believe that this proof calls for a more detailed explanation of how $\mathcal{B}_2$ is built in such a way that it runs in PPT and can win the simulation soundness game when F occurs.

One possible way (to be confirmed, we did not check the details) of going around such difficulties would be to observe that $\mathcal{A}$ must submit his proofs *before* any extraction step starts. If this is the case, then one can define a reduction $\mathcal{B}_2$ that runs the interaction between $\mathcal{A}$ and $\mathcal{C}_1$ only to the point where $\mathcal{A}$ produces the plaintext equality proof, and that this part is still PPT because it does not require any plaintext extraction. $\mathcal{B}_2$ can then submit the proof it got from $\mathcal{A}$ to the simulation soundness adversary.

This approach would apparently not work for the reduction $\mathcal{B}_3$ in the vc.3 game, though: there, the decryption proofs are generated *after* the extractions added in $\mathcal{C}_1$ and $\mathcal{C}_2$ happened. A solution in that case might be to change the order of the game hops, starting with the modifications of the vc.3 game, then bringing those of the vc.1 and vc.2 games: that may make it possible to write reductions that would not need to run the Extract algorithm.

There certainly are other ways of making these reductions and sequences of game hops.

To conclude: our point is that introducing inefficient challengers requires additional caution when writing security reductions: reductions typically make use of the challengers and may then become inefficient, as illustrated above in one specific case. This may or may not be a problem depending on the context. We think that the proofs that are proposed should be more explicit on how reductions are defined and, in particular, explain why they remain efficient when it is needed.

2.4 Context Verification: Input vs. Context

We have highlighted on several occasions the issues with input versus context in the pseudocode algorithms and related input validation issues in the code. We comment here mainly about specification issues with some mentions of the code; we will discuss these issues further in (Sec. 3.2) for the code side. This investigation has unearthed several possible attack avenues:

Attack on Universal Verifiability In our draft report in round two we mentioned an exploitable attack on universal verifiability which exploits several vulnerabilities in the system but is best addressed by increasing input validations; we include our previous explanation with some elaboration in appendix A. We suggested that this attack was partly to the numerous complicated options in ElectionDataExtractionService which has seemingly only grown more convoluted since. It does not appear that the input validations which prevent this problem have been implemented yet; we assume it will be in the December release.

Possible attack on Individual Verifiability In our draft report for round two we commented

> The setup component (re)learns the verification ids from the CCRs and checks consistency, this seems fine as long as at least one CCR is honest.

we have since had a chance to increase our examination and the problem is worse than we initially thought; we have discussed the issue with Post and agreed on a solution which we expect to see in the December release.

The current pseudocode algorithms often occur in a context (within the protocol) in which multiple parties could perhaps have provided the input; it is then unclear which party did. Often this issue is handled appropriately at the code level by checking that all parties agree on the contentious element of the input. There are, however, exceptions and we think that this could be profitable addressed by specifying the origin of the input in the system specification and any consistency checks required.

For an example of the issue mentioned above, consider the vector of verification card IDs received as input by the algorithm CombineEncLongCodeShares. Looking at Fig. 6 of the system specification (Overview of the SetupVoting algorithm) it seems like that this input must come from the Setup Component because the Setup Component generates the value and never receives it. This is not the case and the actual code takes this as input from an untrusted CCR; and in combination with other issues this may result in a breach of individual verifiability. We will discuss this issue in more depth in appendix B.

Overall we think Post now has a reasonable understanding of these issues as evidenced by the changes to system specification in the most recent release where they write

> The context variables should stem from a trusted source (for instance an internal view or a trusted component) and the implementation must check the input variables against the context.

We would still encourage making clear what the source of the input is supposed to be and any corresponding consistency checks but the main step forward requires is to ensure the code actually takes does ensure the context comes from a trusted source.

3 Update on Prior Issues in Scope 2

3.1 Undocumented architecture decisions

There are numerous architecture decisions which have been made which have various advantages and disadvantages. This is to be expected but we think many of decisions should be better described in the documents to ensure a clear view, particularly of the disadvantages. In our previous report we highlighted micro services (which describes both the online system and the verifier)

The micro services, though they implement various checks, rarely check the data for consistency with their view of the election state, which makes checking the possible execution flows hard. This problem seems to be particularly important because the protocol specification document assumes a very coordinated execution flow.

We view this issue principally as a documentation issue but one that belongs more to scope 2 than scope 1. This has not been addressed.

3.2 Unvalidated inputs

In this section we continue our discussion of the issues of input validation we started discussing in Sec. 2.4. While we have not had to time to exhaustively look through input validation issues we will include a short summary of our current thoughts on the validation of input into key algorithms below.

Alg. 4.1 GenKeysCCR The algorithm in the specification has no input only context. However, in the code all "context"[2] comes from the untrusted voting server; this does not seem compatible contrast between input and context in the system specification.

If the voting server gives different groups to different CCRs this will be caught no later than ElectoralBoardConstitutionService by the control component and actually slight earlier by the CCRs as we detail below.

Alg. 4.4 GenEncLongCodeShares The data going into this algorithm comes from the GenEncLongCodeSharesProcessor which checks that the setup component signed the data; the one exception to this is the encryption parameters which comes from the local state. However, the encryption parameters were placed in local state after coming from an untrusted party and hence should not really be considered trusted.

If the local version of encryption parameters does not match that of the setup component this will be detected by a consistency check in implementation which is not explicit in the pseudocode.

Alg. 4.5 CombineEncLongCodeShares The implementation of this algorithm has the intended split between context (coming from it's internal view) and input (coming from external sources). However, the verification card ids should be part of context not the input, or at least checked as consistent within the verification card set id. As we mentioned in Sec. 2.4, this leads to a possible attack which we briefly sketch in appendix B.

Alg. 4.6 GenCMTable Like Alg. 4.5 the implementation reflects the specifications split between context and input; however, it also has the same issue where verification card ids are part of input.

[2] with the exception of the maximum number of selectable voting options

Alg. 4.7 GenVerCardSetKeys It is not clear to us that GenVerCardSetKeys actually ensures that $pk_{CCR_2}, pk_{CCR_3}, pk_{CCR_4}$ are actually ϕ long and not longer; the underlying issues is that GroupVector reports element size as the size of its first element without checking that all its elements have the same size. In this specific instance this does not appear to matter.

Alg. 4.8 GenCredDat Unlike Alg. 4.5 and 4.6, GenCredData does retrieve the verification card IDs from a trusted source.

Alg. 4.9 SetupTallyCCM Similar to Alg. 4.1 it seems that the context is again coming from an untrusted source, with the exception of the node id and maximun number of support write-ins.

Alg. 4.10 SetupTallyEB It seems the implementation of this algorithm takes the encryption group from the untrusted CCRs; this is contradictory with it being part of the context in the specification. We assume that due to the flow of the implementation that if the encryption parameters where incorrect then another of the (earlier) checks would catch this.

The specs lists the maximum number of write-ins as part of the input where similar parameters were considered context in other algorithms, this seems to be an a minor error in the specification.

Alg. 5.3 VerifyBallotCCR and Alg. 5.4 PartialDecryptPCC It seems much of the context including encryption parameters, election event id, verification card id comes from an untrusted source. The verification of the payload does check the consistency of the election event id which the local view of the corresponding encryption parameters. There are numerous checks which would make sense to add such as ensuring the verification card set exists for the specified election event and the verification card exists within the verification card set.

Alg. 5.5 DecryptPCC and Alg. 5.6 CreateLCCShare These algorithms share much of the same issues discussed in the previous point. This is concerning here because the "process once" defenses in LCCShareProcessor could be bypassed using the same verification card id with different verification card set id or election event id.

3.3 Update on our proposed future work

In our draft report in scope two, we envisioned completing significant investigation of the code with respect with requirements 2.5, 2.6, and 2.7 in time for this addendum; we have not succeed in doing this partially due to tighter time constraints then we had hoped for and partly because in the process of investigating the code we discovered new errors we were not expecting. We include in appendix B an example of a possible attack based on the kind of vulnerabilities we were finding; the issues of odd input flows and set equalities when order is important, which appear in the above example, also occur elsewhere in the code

and we have not exhaustively checked what else might be affected by this. We also include in appendix C an example (and unpolished) line of the reasoning we went through to think about individual verifiability at the code level; we include it because in following it we discovered significant issues in the code which suggests that Post is not going through a similar exercise since they had not already identified the issues.

4 New issues

In this section we will comment very briefly on some new issues that have arisen.

4.1 Write-ins

In general, it seems likes the process for handling write-ins is sensible and avoids several errors that we worried would be made. It doesn't seem like the write-in can break anything for predetermined choices, unless the system for predetermined choices is already broken.

We do have a few specific comments, in the Specification document:

- **Algorithm 3.9**: It would be important to clarify that the square root that is returned must be the one that is less than q, since Algorithm 3.10 assumes that x is in $\mathbb{Z}_q$. For instance, the first operation of Algorithm 3.9 could become: $x \leftarrow y^{(p+1)/4} \bmod p$; if $x > q$ then $x = p - x$.
- **Section 3.6.4**: we are a bit confused about the numbering: positions are counted from 1, while write-in fields are counted from 0. For instance, shouldn't it be the case that $w_{id,1}$ gets the value 1, not $w_{id,2}$?

4.2 Authentication

We comment here on the security of the authentication which occurs before CreateVote, under the assumption that Start Voting Key (SVK) has sufficient entropy; this is not specified in the documentation. Since the voting server is untrusted for all security properties there are a fairly limited number of attacks which it couldn't launch regardless of the security of this step. We have not examined the vote portal carefully, nevertheless at present our only concern is that the software might allow a malicious voting portal to prompt the voter for the ballot casting key outside of the proper sequence. The control flow of the voting portal is not particularly transparent and the related architecture decisions should be better documented.

4.3 Missing Elements

We observe that some elements are missing in the Specification that are important to assess the compliance with the Ordinance.

For instance, Article 2.7.3 of the Appendix indicates that:

> It must be ensured that no attacker can take control of user devices unnoticed by manipulating the user device software on the server. The person voting must be able to verify that the server has provided his or her user device with the correct software with the correct parameters, in particular the public key for encrypting the vote.

But we did not find anything in the Specification suggestion how this kind of attack would be prevented. Indications about this are given in the Architecture document (p. 11). We understand that the countermeasures against such attacks are different in nature from those in the protocol, and that describing them in the Specification would hamper the readability of that document. However, it also appears that the Specification should be sufficient to assess whether such attacks are properly prevented. As such, we would suggest having a short section in the Specification that explains what is taken away from that document, with pointers to the places where such elements are described.

References

[1] Thomas Haines, Olivier Pereira, and Vanessa Teague. Report on the swiss post e-voting system. `https://www.newsd.admin.ch/newsd/message/attachments/71147.pdf`, March 2022.

A Attack on Universal Verifiability Update

The universal verifiability of the system relies (largely) upon a chain of zero-knowledge proofs. These proofs demonstrate that the announced result, for a given ballot box, is the correct decryption (and permutation) of that ballot box (which is here used to refer to an agreed upon collection of ciphertexts).

The vulnerability underlying the attack below is as follows: the verifier is inconsistent in how it extracts data from the disk; this inconsistency breaks the chain of zero-knowledge proofs. Specifically, it sometimes extracts data based on the filename/location of the data and sometimes based on the content of the data. The inconsistent data could plausibly have been detected by the following parts of the verification specification to varying degrees but the implementation does not prevent the exploitation of the vulnerability:

- The authentication checks could plausibly have helped. However, they take the context data as input from the signer, in this case the adversary, rather than check based on their own view of the context.
- No consistency check is performed that each online control component has contributed exactly one shuffle payload per ballot box.

Attack: We will assume below that the auditor and control component 1 are honest but all other control components are dishonest.

The attack works by altering the order and names of the shuffle payloads going to the verifier; in addition, the dishonest control components divate from the protocol during phase in which key generation occurs and the tally control component deviates during tallying phase.

The result of this reordering is that the online control components' shuffles will verify as expected BUT when the verifier attempts to verify the tally control component's payload it will NOT do this with respect to the output of the CC4 as the verifier intends, but some other output. In the example we provided to Post, the verifier checks the tally control component shuffle of the ballot box 750a359fc3bd48aca4a1156666846267 with respect to the (decrypted and permuted) output of CC1 for the ballot box 99208915ab634a4293e36fcf4efadf54. This means that the validity of the tally proofs no longer link the results for 750a359fc3bd48aca4a1156666846267 to its own ballot box but to that of 99208915ab634a4293e36fcf4efadf54.

We initially thought the attack would not work because, for technical reasons, the mismatched input must be from control component 1, 2, or 3. In other words, the dishonest tally control component's proof would be verified with the wrong input ciphertexts but would ultimately result in garbled group elements which would be detected by the VerifyProcessPlaintextsAlgorithm. We realised later that the adversary could choose the secret keys of the dishonest control components such that they cancel each other out and the decryption of the ballots using the secret keys of the first control component and election board would have the same result as decrypting using all keys. This means that this verification should then pass.

Let NXBBY denote the contribution of the shuffle payload contribution of the Xth CC to the Yth ballot box. The expected shuffle order is as follows where the file name is implicit in the order of the payloads

Folder BB1: N1BB1 N2BB1 N3BB1 N4BB1
Folder BB2: N1BB2 N2BB2 N3BB2 N4BB2
Folder BB3: N1BB3 N2BB3 N3BB3 N4BB3
Folder BB4: N1BB4 N2BB4 N3BB4 N4BB4

For this attack the payloads are reordered as follows:

Folder BB1: N1BB1 N1BB2 N1BB3 N1BB4
Folder BB2: N2BB1 N2BB2 N3BB2 N4BB2
Folder BB3: N3BB1 N2BB3 N3BB3 N4BB3
Folder BB4: N4BB1 N2BB4 N3BB4 N4BB4

The line in VerifyOnlineControlComponents evidence which pulls the data is as follows

```
final Map<String, List<ControlComponentShufflePayload>> controlComponentShufflesByBallotBoxId =
electionDataExtractionService.getAllControlComponentShufflePayloads(inputDirectoryPath).stream()
      .collect(Collectors.groupingByConcurrent(ControlComponentShufflePayload::getBallotBoxId));
```

It is immediate that when grouping by ballot box ids, the payloads in the example above come out exactly the same in the original and reordering. In other words the value of controlComponentsShufflesByBallotBoxId is unaffected by the reordering attack. This ensures that checks performed by VerifyOnlineControlComponents will not detect this attack.

The dishonest tally control component deviates from the protocol by producing, for ballot box BB1, a proof of shuffle and decryption for the contents of N1BB4 instead of N4BB1.

Discussion of why this attack isn't caught The above analysis considers how to produce a shuffle payload which passes VerifyTallyControlComponent; however, such a payload would also need to pass the following consistency checks:

CheckSignatureOfflineShuffle Since the TallyControlComponent is malicious it is straightforward for it to sign the tampered payload.

VerifyCiphertextConsistency Checks that the ciphertexts are of the expected size; this check pulls the files by folder so the tampered ballot boxes likely need to have the same number of write-in options.

VerifyPlaintextConsistency Checks that the plaintext has the required size; this follows in this case from the constraints already applied.

VerifyNumberConfirmedEncryptedVoteConsistency As the name indicates this test checks that the number of confirmed votes is equal; this ensures that any attack of this kind must occur between two ballot boxes which have the same number of confirmed ballots.

Impact The attack allows the adversary to change the election result without detection by the voter or system. The limitation on the attack is that the result the adversary claims, with respect to a given ballot box, must have a relationship to the ballots cast in a different ballot box. The impact of the attack depends significantly on the election parameters but it seems likely it could be exploited in practice within the threat mode; fortunately the vulnerability is readily fixed.

Resolution Our understanding from talking with Post is that they will address this by using consistency checks to ensure the file names and contents have the expected correspondence. This approach would seem to work but we encourage any such requirement to be documented in the verifier specification or architecture document.

B Possible Attack on Individual Verifiability

What's the problem: The system is missing a couple of checks that ensure that the verification card ids and related payloads are processed as expected. We have not diagnosed exactly how this can be exploited but there seems to be a moderate probability that this breaks individual verifiability. The solutions seems fairly cheap so it seems easier to resolve than properly understand the full implications of the current state.

The essence of the current vulnerability is the association between verification card ids and key material can be permuted halfway through the setup phase; this causes the various allow lists and tables of encrypted to get into a state well outside of the limits within which the system is expected to operate.

Where is the problem in the online system: The setup component generates the verification card ids in GenVerDatAlgorithm which is ultimately called through the VotingCardSetSetupController. These are then persisted in various services. If we jump to CombineEncLongCodeSharesAlgorithm which is called by VotingCardSetDataGenerationService, the verification card ids put into this algorithm ultimately come from EncryptedNodeLongReturnCodeSharesService.load which chooses the verification ids from encryptedSingleNodeLongReturnCodesGenerationValues.get(0).getVerificationCardIds() which is ultimately based on the first CCR's payload.

There does not appear to be any check which ensures that the verification card ids from the first CCR match the expected verification card ids.

Solution: We encourage a check that all verification cards ids received from the CCRs by the setup component are verified to match what the setup component expects both in content and order. The check consists of equality of lists which I assume is fairly cheap in comparison to other operations performed by the setup component.

Where is the problem in the verifier: VerifyVerificationCardIdsConsistency checks that the verification card ids from the CCRs are as as expected but

using set equality which does not check that the order is as expected. These ids then seem to be ignored by ExponentiationProofsVerificationExtractionService which just passes the expected ids from the setup component. In combination this means that a changing of order of verification card ids will not be noticed by the verification.

Solution: Check the verification card ids match in both content and order.

C Reasoning about individual verifiability

In this section we (extremely informally) analyse one component of individual verifiability with respect to the code. This section is intended to be indicative of the kind of sanity checking we expect Post to be doing internally but which we have not seen evidence of. At the level of polished shown here, this does not constitute evidence of the security of the code but is useful in finding vulnerabilities. We do not claim the analysis is complete but it does serve to check for certain application vulnerabilities. Specifically, we think this kind of analysis is better suited to capture data validation and authentication issues than any analysis of the system we have seen to date.

Essentially the line of reasoning works by trying to figure out the required preconditions for security at the code level and chasing these backwards through the execution flow.

C.1 Definitions

- An honest control component considers an encrypted vote to confirmed with respect to a given verificationCardSetId if it is returned when calling EncryptedVerifiableVoteService.getConfirmedVotes(verificationCardSetId).

C.2 Assumptions

Cryptographic:

- We will be imprecise and say certain events cannot happen when we rather mean that will not happen with non-negligible probability for a polynomial time adversary under the assumption that certain problems are hard.

About the code:

- encryptedVerifiableVoteService.save does not save if the claimed verification card does not exist or does exist but already has a corresponding vote.
- we assume certain properties of the setup component which hold for the spec but we have not validated for the code.
- we assume that the setup component checks the verification ids revived from the CCRs match those it sent in both order and content. (This does not occur at present but Post has agreed to add it.)

- we assume that the verifier checks the verification ids revived from the CCRs match the setup components sent in both order and content. (This does not occur at present but Post has agreed to add it.)

C.3 No confirmed votes without submission of confirmation code from the voter

An encrypted vote is confirmed according to our definition if it is:

- marked as confirmed and
- belongs to the relevant verification card set

(since this is what EncryptedVerifiableVoteService.getConfirmedVotes does). For this aspect of individual verifiability we will argue that the honest component component will only have satisfied these two conditions if it received the ballot confirmation code associated with the verification card that the vote purports to come from; we most also argue that there is no way (within the threat model) to learn the ballot confirmation code without the voter submitting it.

C.3.1 The secrecy of the ballot confirmation code

The setup component generates the ballot confirmation code and puts out an encryption which is derived from it. Very informally, all the decryptions that occur with the corresponding decryption key are associated are for ciphertexts with associated zk-proofs (which are checked by verify config) which prevent an unintentional decryption oracle. Both the allow list and CMtable have values derived from the ballot confirmation key through the random oracle which prevents leakage beyond guessing which suffices if the value is high entropy; the systems use various measures to boost the entropy of these values.

C.3.2 That the honest component saw the conformation code

Any vote can be saved The only method which saves encrypted votes is generatePartiallyDecryptedEncryptedPayload in the PartialDecryptProcessor. This processor accepts input from the untrusted voting server which it first verifies using verifyPayload before saving. This verification checks the signature from the untrusted voting server and the group matches the election event.

All verification cards in the honest component were placed there by GenEncLongCodeSharesProcessor which checks that it's input did come for the trusted setup component. We therefore conclude that all encrypted votes are attached to a valid verification card id and verification card set.

Steps required for confirmation We will now argue that the checks performed by honest CCR before marking the encrypted vote as confirmed require that it has seen the ballot confirmation code.

Votes are set as confirmed in VerifyLVCCHashAlgorithm used by the LongVoteCastReturnCodesShareVerifyProcessor. For a vote to be considered confirmed input must be provided such that the hash of that input exists within the the LongVoteCastReturnCodesAllowList. This allow list is retrieved according to the verificationCardSetId from the VerificationCardSetEntity key by the component.

The LongVoteCastReturnCodesAllowList is saved by the LongVoteCastReturnCodesAllowListProcessor which checks that the payload came for the trusted setup component.

Generation of allow list The generation of the allow list is distributed on a fairly large stack of classes

VotingCardSetGenerationController Given an electionEventId and a votingCardSetId the controller uses the votingCardSetGenerateService to construct the codes.

VotingCardSetGenerateService This service is responsible for many things but importantly for us it calls returnCodesPayloadsGeneratedService.

ReturnCodesPayloadsGeneratedService This service uses the votingCardSetDataGenerationService to generate the allow list which is signs and sends.

VotingCardSetDataGenerationService This service takes the contributions of the CCRs from encryptedNodeLongReturnCodeSharesService and uses combineEncLongCodeSharesAlgorithm to combine them.

EncryptedNodeLongReturnCodeSharesService This service retrieves the payloads it has received for the relevant verificationCardSetId and electionEventId. We comment on the adequacy of these checks in C.3.2 but for now we will state simply the suffice to ensure the full payload is received for each honest CCR. *The code takes the verificationCardId for the first CCR without checking if this is correct.*

CombineEncLongCodeSharesAlgorithm The return code list is constructed by hashing (“VerifyLVCCHash”,electionEventID,verificationCardSetID,verificationCardID, Hash(“CreateLVCCShare”,electionEventID,verificationCardSetID,verificationCardID,1,$\texttt{lVCC}_{id_1}$), Hash(“CreateLVCCShare”,electionEventID,verificationCardSetID,verificationCardID,2,$\texttt{lVCC}_{id_2}$), Hash(“CreateLVCCShare”,electionEventID,verificationCardSetID,verificationCardID,3,$\texttt{lVCC}_{id_3}$), Hash(“CreateLVCCShare”,electionEventID,verificationCardSetID,verificationCardID,4,$\texttt{lVCC}_{id_4}$))

Checking of allow list VerifyLVCCHashAlgorithm is called LongVoteCastReturnCodesShareVerifyProcessor. This processor revives input from the untrusted voting server.

- The signatures on the payloads are checked with respect to the key of the claimed NodeId. *The context check is weird because it is based on the first input not on a global truth.*

- The list of payloads is checked to be of the expected length
- The expected NodeIds are expected to be present (this ensures in combination with the previous two checks that we have exactly one payload from each NodeId)
- All confirmation keys and node ids are in agreement
- These verified payloads are passed on to generateControlComponentlVCCSharePayload

After several more steps VerifyLVCCHashAlgorithm checks that (("VerifyLVCCHash", ee, vcs, vc_id), hlVCC_id_1, hlVCC_id_2, hlVCC_id_3, hlVCC_id_4) is in the allow list.

By the domain separation of recursiveHash the only way for for this to be in the allow list is if hlVCC_id_i, where i the honest CCRs index, to match Hash("CreateLVCCShare",electionEventID,verificationCardSetID,verificationCardID,i,lVCC_id_i). This was generated in the CreateLVCCShareAlgorithm so we need to jump back to there.

CreateLVCCShareAlgorithm which is handled by LongVoteCastReturnCodesShareHashProcessor as follows:

- lVCC_id_i = $(H(CK)^2)^{kc_{id}}$

This will only be true if $CK = BCK^{k_{id}}$ barring a collision on the hash function.

The adequacy of the checks performed by EncryptedNodeLongReturnCodeSharesService among others The underlying repositories looks in a particular file line location and loads all the files' contents into memory throwing an error if it does not type check. Each file is expected to contain the payloads of each CCR with respect to a particular chunk. The repository returns all these as a list of list of payloads order by the chunkId of the first element of the inner list.

- NodeContributionsResponsesService checks that each payload received claims to belong to the correct election event and verification card set.
- EncryptedNodeLongReturnCodeSharesService now checks the following:
  - that the list received is non-empty
  - that each payload us signed by the claimed node with respect to the claimed data
  - Every payload has the chunck id we would expect based on it's location in the data structure and for each chunk has the correct number of payloads

The system then goes through all response retrieving all payloads matching a particular node id and retrieved all the contained data.

- Checks that set of verification card ids received from each component are duplicate free
- Checks that there are the expected number of cards

These values are further checked by prepareCombineEncLongCodeSharesInput within the VotingCardSetDataGenerationService.

- There are no duplicates in the verification card ids
- The number of rows and columns in the data is as expected
- Various other checks whose adequacy we don't discuss.

We now wish to conclude that information in encryptedSingleNodeLongReturnCodesGenerationValues.get(i-1) really came from the ith CCR without tampering if i is honest.

- By line 102 we know the data claimed to come from i and by the signature check it did.
- We know by line 130 that we received the correct number of cards *so by assuming that the honest CCR only produced the correct amount data* we know the only issue is duplicates
- By the check in EncryptedSingleNodeLongReturnCodeShares.Builder we know that their are no duplicates.

Knowing that this information matches what the honest CCRs sent is a good start.