Addendum to How not to prove your election outcome

The use of non-adaptive zero knowledge proofs in the Scytl-SwissPost Internet voting system, and its implications for cast-as-intended verification

Sarah Jamie Lewis[1], Olivier Pereira[2], and Vanessa Teague[3]

[1]Open Privacy Research Society, sarah@openprivacy.ca
[2]UCLouvain – ICTeam, B-1348 Louvain-la-Neuve, Belgium,
olivier.pereira@uclouvain.be
[3]The University of Melbourne, Parkville, Australia,
vjteague@unimelb.edu.au

March 29, 2019

This note extends our earlier analysis of the use of the weak Fiat-Shamir transform in the Scytl-SwissPost Internet voting system. Here we show that a cheating client can perform essentially the same attack we described in "How not to prove your election outcome." In the client case, this involves submitting a nonsense vote while successfully faking a proof that its partial return codes match the vote. The effect would be that the voter would receive exactly the right return codes as if their vote had been properly cast, but at decryption time the vote would be nonsense that would not be counted.

We notified SwissPost of these findings on Wednesday March 27th.

This note should be read as an added section of our prior report on this same weakness [LPT19].

1 Producing a false ballot validity proof

The same issue within the Fiat-Shamir transform can be exploited to subvert the individual verifiability of sVote: an honest voter could submit her/his vote intent to a corrupted voting client, which would encrypt a nonsense vote instead and submit that nonsense vote to the servers. Honest servers would accept it and send back to the voter the return codes he/she is expecting to see. The voter would then consider that her/his vote intent was correctly captured. However, when this vote was decrypted after being mixed, it would be invalid.

1.1 Ballot preparation and individual verifiability process

The discussion below focuses, for simplicity, on the case in which a voter wants to submit a single vote v encoded as a prime quadratic residue mod p. The extension to general ballots in which multiple (prime) votes would be used is immediate. We follow the notations of the sVote protocol specification [Scy18, Section 5.4], in a slightly simplified way.

The ballot preparation and individual verifiability processes rely on three keys.

1. An election public key EL. The corresponding private key is in the hands of the CCM's, in a distributed form. Votes are encrypted with this public key and, after validation, are mixed and decrypted by the CCM's.

2. A choice return code public key pk. The corresponding private key is in the hands of the CCR's, in a distributed form. Partial return codes are encrypted with this public key. They are then processed by the CCR's, leading to the decryption and sending of the choice return codes to the voting client, for verification on their vote verification card.

3. A verification card public key K, which is unique per verification card. The corresponding private key k is in the hands of the voting client. (We assume that its distribution is trusted.) The private key is used by the voting client to produce the partial return codes, which are then encrypted using pk.

The process of preparation and verification of a vote v is then as follows (we focus on the relevant steps):

1. The voting client computes the ciphertext $E_1 = (E_{10}, E_{11}) = (g^r, v \cdot EL^r)$.

2. The voting client computes a partial return code $pCC = v^k$, which is then encrypted with pk as $E_2 = (g^{r'}, pCC \cdot pk^{r'})$.

3. The Schnorr protocol is used to produce a proof of knowledge π_s of r used in E_{10}.

4. F_1 is computed as E_1^k, that is, $(F_{10}, F_{11}) = (E_{10}^k, E_{11}^k)$, and a proof of exponentiation π_e is computed to make sure that (K, F_1) is indeed equal to (g^k, E_1^k) for a secret k.

5. A plaintext equality proof π_p is used to show that F_1 and E_2 encrypt the same value (that is, pCC), w.r.t. EL in the first case and w.r.t. pk in the second case.

6. The vote defined as $E_1, E_2, \pi_s, F_1, \pi_e, \pi_p$ is submitted to the server.

7. The proofs are verified and, using E_2 only, the return codes are computed and sent back to the voter, who can then verify them on his verification card.

1.2 The proof of exponentiation

The proof of exponentiation used in sVote [Scy18, Sec. 9.1.5 and 9.1.14] is a slight generalisation of the Chaum-Pedersen proof implemented incorrectly for decryption. The exponentiation proof presents the same weakness in its implementation of the Fiat-Shamir transform.

In the context described above, it proceeds as follows:

1. Pick a random a and compute $(B_1, B_2, B_3) = (g^a, E_{10}^a, E_{11}^a)$.

2. **Compute $\mathbf{c = H(K, F_1, B_1, B_2, B_3)}$.**

3. Compute $z = a + ck$.

The proof consists of (c, z), and is verified by computing $B'_1 = g^z/K^c$, $B'_2 = E_{10}^z/F_{10}^c$, $B'_3 = E_{11}^z/F_{11}^c$, and checking that c equals $H(K, F_1, B'_1, B'_2, B'_3)$.

1.3 Lack of adaptive soundness of the proof of exponentiation

As for the Chaum-Pedersen proof discussed above, this proof lacks adaptive soundess. In particular, g and E_1 are not hashed, so there is no guarantee that they are chosen before the proof is computed.

In what follows, we assume that g was generated honestly, in a verifiable way (but this is not required by the sVote specification) and focus on E_1. A malicious adaptive prover could then proceed as follows.

Start by picking F_1 as a random encryption of pCC for the vote the voter intended, that is, as $(g^r, pCC \cdot EL^r)$. Then:

1. Pick random (a_1, a_2, a_3) and compute $(B_1, B_2, B_3) = (g^{a_1}, g^{a_2}, g^{a_3})$.

2. Compute $c = H(K, F_1, B_1, B_2, B_3)$.

3. Compute $z = a_1 + ck$.

The proof is (c, z). Then, set $E_1 = ((B_2 \cdot F_{10}^c)^{1/z}, (B_3 \cdot F_{11}^c)^{1/z})$, which guarantees that the verification equation passes, despite the fact that $F_1 \neq E_1^k$ with overwhelming probability, meaning that E_1 will not be an encryption of v except with negligible probability.

1.4 Why individual verifiability fails

Running the previous steps provides (E_1, F_1, π_e) that pass verification. F_1 is a valid encryption of the right partial return code, but E_1 is not an encryption of the right vote.

In order to complete the ballot, we can compute E_2 in a perfectly honest way, using whatever vote the voter asked for (which will not be cast) and the true pCC. We then compute π_s in a completely honest way by observing that $E_{10} = (B_2 \cdot F_{10}^c)^{1/z} = g^{(a_2+rc)/z}$, so $(a_2 + rc)/z$ is the random value used to produce E_{10}. Finally, we compute π_p in a perfectly honest way as well, since it corresponds to a true statement for which we have a witness: F_1 and E_2 do encrypt the same value, which is the correct function of the intended vote.

All the proofs are valid, so E_2 will be used to derive the return codes corresponding to the vote intent v, which will then be accepted by an honest voter, who will have his/her vote confirmed. However, when E_1 is decrypted (after being processed through the mixnet), it will be declared invalid.

References

[LPT19] Sarah Jamie Lewis, Olivier Pereira, and Vanessa Teague. How not to prove your election outcome: The use of non-adaptive zero knowledge proofs in the scytl-swisspost internet voting system, and its implications for decryption proof soundness, 2019. `https://people.eng.unimelb.edu.au/vjteague/HowNotToProveElectionOutcome.pdf`.

[Scy18] Scytl. Scytl sVote protocol specifications – software version 2.1 – document version 5.1, 2018.