Apollo – End-to-end Verifiable Internet Voting with Recovery from Vote Manipulation

Dawid Gaweł[2], Maciej Kosarzecki[2], Poorvi L. Vora[1], Hua Wu[1], and Filip Zagórski[2]

[1] Department of Computer Science*,
The George Washington University
[2] Department of Computer Science**,
Wroclaw University of Science and Technology

Abstract. We present security vulnerabilities in the remote voting system Helios. We propose Apollo, a modified version of Helios, which addresses these vulnerabilities and could improve the feasibility of internet voting.
In particular, we note that Apollo does not possess Helios' major known vulnerability, where a dishonest voting terminal can change the vote after it obtains the voter's credential. With Apollo-lite, votes not authorized by the voter are detected by the public and prevented from being included in the tally.
The full version of Apollo enables a voter to prove that her vote was changed. We also describe a very simple protocol for the voter to interact with any devices she employs to check on the voting system, to enable frequent and easy auditing of encryptions and checking of the bulletin board.

1 Introduction

With the perceived security of internet banking and electronic commerce, there has been a lot of interest in voting on the internet. The internet voting system Helios is a prominent end-to-end verifiable (E2E-V) system that has been used for multiple non-governmental elections. In this paper we present attacks to the Helios voting system and propose voting protocol Apollo to address these.

Attempts at voting on the internet in governmental elections have been demonstrated to be vulnerable to client-and/or-server-side adversaries [15,23,13,25]. An E2E-V system would allow the detection of such attacks. However, the E2E-V property, while necessary, is not sufficient for secure elections. For example, a voting terminal may behave honestly throughout the E2E-V voting protocol, until the voter enters her credential. The terminal could then cast a vote of its choice.

* This material is based upon work supported in part by the Maryland Procurement Office under contract H98230-14-C-0127 and NSF Award CNS 1421373
** Authors were partially supported by Polish National Science Centre contract number DEC-2013/09/D/ST6/03927.

Or the election server could replace the vote with another one. An alert voter will notice that there is a problem and may complain; however, she has no evidence to back her complaint. It is well-known that Helios possesses this vulnerability. The inability to resolve multiple such three-way disputes among the voter, her terminal and the election server could result in undesirable uncertainty about an election outcome. Additionally, while voters can audit encryptions and check the bulletin board for the correct vote encryption, it is well-known that they rarely do so. In the 2009 elections of the City of Takoma Park, MD, fewer than 4% of cast ballots were subject to the voter verification [7]. A recent study [20] examined the frequency and conditions under which voters check their receipts, reporting that only about 7.5% of voters performed receipt checks (and just 0.5% filed a dispute when shown an incorrect receipt).

Benaloh's SVE Benaloh's *Simple Verifiable Elections (SVE)* protocol [3] for in-person voting enables the voter to detect a dishonest terminal (voting machine). After the voter tells the machine her choice, the machine prints an encryption of the choice on a piece of paper. The voter can either take the printout and cast it as her ballot or she can challenge the printed encryption. In the second case, the machine reveals (prints) the randomness used for the encryption; the voter can use another computer, or many computers, she trusts to check that the printed string is indeed an encryption of her vote. In this way, the voter is able check if the voting machine cheats while encrypting votes. One implementation of this protocol is the STAR-Vote system [4].

Helios The Helios [1] protocol is an online voting protocol inspired by *SVE*. The role of the machine in *SVE* is played by the voter's web browser in Helios. After the voter communicates her choices, the browser encrypts it and displays a commitment to the ballot encryption (called a *ballot tracker*), which plays the role of the printed encryption in *SVE*. The voter chooses whether to audit or cast the encrypted votes. If she audits, the randomness used for encryption is displayed. Else she authenticates herself and the browser sends the encrypted ballot to the server, which performs a verifiable tally of all encrypted ballots sent in with valid credentials.

1.1 Our contributions

Our contributions are as follow: we present a set of vulnerabilities we discovered in the Helios code (Cross-Site Scripting, Cross-Site Request Forgery and other attacks); we have informed Helios developers about our findings and the currently available version is patched. The main contribution is a voting protocol Apollo which addresses some of the problems with Helios. In addition Apollo explicitly describes an auditing protocol to be used by the voter's computational voting assistant(s), allowing the voter to focus only on checking what the voting assistant says and whether multiple voting assistants agree.

Apollo as an extension of Helios Apollo uses the same approach for verification as *SVE*. In contrast with Helios, a machine commits to the ballot encryption on the public bulletin board instead of on the machine's screen. This change has positive security consequences. The posting of the encryption on the bulletin board does not imply that all information necessary to check an audited ballot is also on the bulletin board. We describe a protocol for auditing the vote and checking the bulletin board which allows the voter to choose who obtains this information. This allows the voter to protect not only her true vote, but also the audited vote, which is not displayed on the bulletin board.

The voter is encouraged to use *voting assistants* (*e.g.,* tablet, smart watch, phone) that enable her to check if the voting terminal is behaving honestly. If a voter chooses not to use any voting assistants, her voting experience is exactly the same as in the original Helios system, but she is still better protected than in the original Helios. Additionally, if a voter chooses to use one or more voting assistants, we present a real-time protocol for auditing and checking. We have attempted to keep the voter experience as simple as appears possible for these tasks. If the voter uses a single voting assistant, she needs to only check what the voting assistant says. If she uses multiple assistants, she needs to additionally check if they agree. The insertion of all voter tasks into the voting process, in a minimal fashion will, we hope, increase the frequency and ease of the audits and checks, improving the overall confidence in the election outcome. An experimental study of the usability of the protocol is outside the scope of this paper.

In contrast with the single casting credential used by each Helios voter, an Apollo voter is issued multiple credentials: multiple *casting codes* to change a vote if an incorrect one is posted, and a *lock-in code* allowing the voter to communicate to the public that she believes her vote is correctly represented on the bulletin board (similarly to Remotegrity [26]).

Apollo: Assumptions and Properties We present two versions of Apollo that address the problems of credential stealing and the attacks described above. Like Helios and all other E2E-V systems, both versions assume a secure bulletin board with authenticated append-only write access and public read access. Both versions explicitly address the audit process as carried out by one or more voting assistants, making it part of the main protocol.

- Making the same assumptions as Helios—of an honest credential authority and a second channel for electronic delivery of credentials—Apollo-lite prevents the inclusion of votes not authorized by the voter by enabling public detection of the problem.
- When an honest registrar may not be assumed, the full version of Apollo allows an incorrect vote to be counted only if the registrar has been dishonest. It enables the voter to prove that she did not cast it. The full version requires that the voter have the ability to provide a final irrepudiable instruction; this can be achieved through the use of scratch-off authentication cards as with

Remotegrity [26], or a special computational device trusted only to digitally sign a single instruction, such as described in [14].

While a rigorous demonstration of the above properties is outside the scope of this paper, we provide a non-rigorous security analysis with respect to common attacks in the paper.

We assume that the voter has access to at least one honest terminal and that there are at most $k-1$ dishonest terminals. When the assumption regarding terminals is not met, the voter encounters a denial of service attack; unlike in Helios, when her vote may be replaced. A denial of service attack may be targeted towards a particular vote or type of voter, preventing the casting of a particular type of vote. However, the voter can prove that her vote is not among those being counted. She can then obtain the opportunity to cast a vote using another channel, such as the postal mail system or in-person voting. Note that any system which receives the plaintext vote is capable of launching a targeted DoS attack of this sort. While coded voting can make targeted DoS harder, coded voting protocols pose usability challenges. Further, a voting terminal, especially one the voter uses for other purposes as well, might be able to profile a voter and guess her vote with considerable accuracy without seeing it.

We assume that at least one of the voting assistants is honest. The assumption of a less powerful adversary (*e.g.*, a majority of the assistants is honest) results in a small modification of the audit protocol. Note that any E2E system used by human voters will need to make an assumption about the computer(s) used to check the audits and/or the bulletin board.

1.2 Organization of this paper

Section 2 presents related work in remote voting systems, section 3 presents the Apollo protocol, section 4 its security properties, section 5 the vulnerabilities in Helios code and section 6 our conclusions.

2 Related work

The Helios voting system [1] has been used in several binding elections, including those for office in the ACM and IACR. Main attacks on the system include those that exploit client-side vulnerabilities [11,16] and those where two voters are issued the same receipt ("clash attacks") [19].

To protect against the attacks described in [11,16], a modification of Helios [21] presents to the voter a QR-code with which a mobile application can check whether the ballot is correctly encrypted. But the app does not checke if a ballot is correctly posted.

The idea behind clash attacks [19] on end-to-end verifiable schemes is that an attacker provides two distinct voters with the same cryptographic receipt and casts an additional vote. As described in [19], the original version of Helios—where the name of the voter is published next to her ballot—is immune to the clash attack. However, the variant of Helios proposed in [2] (and used in, for example, IACR elections)—where voters obtain aliases from the election authority in a registration phase—is vulnerable. The browser (Helios client), the bulletin board and the authority in charge of issuing aliases to voters need to collude to carry out the attack.

Online voting using the Smartmatic voting system in the state of Utah to choose the Republican nominee for the Presidential election in the US drew considerable attention recently (the website providing information on the voting process is no longer available). From the information provided, and in the absence of any ability to audit the tally, the system is vulnerable to client and server side attacks.

New South Wales, Australia, used iVote in 2015. iVote was demonstrably vulnerable to attacks on the server side, and to clientside attacks when the voter either did not verify her vote, or was misdirected about where to verify her vote [15].

The Estonian internet voting system is vulnerable to several attacks [23], including client-side attacks that change the ballot without being noticed during the voting phase. The voter will notice the malfunction or cheating if she decides to verify the ballot, but she is not able to prove there is a problem. The system also possesses several server-side vulnerabilities.

The internet voting pilot in Washington, DC, did not provide any means for the voter to verify any aspect of the election, and was demonstrated to be vulnerable to server-side attacks [25].

The Norwegian internet voting system used in 2011 [13] has the voter using a computer to encrypt the vote, and receiving a receipt from the receipt generator. Voter verification requires trusting the receipt generator, and there is no evidence released to enable the public verification of tally correctness.

3 Apollo

In this section we present Apollo, which provides evidence of vote manipulation that can be verified by a third party.

3.1 Participants and Threat Model

We first explain the Apollo contribution in the context of the Helios threat model, which is also standard for other E2E-V voting protocols and systems. We term this the threat model for Apollo-lite, or the *lite threat model.* All except the last assumption below are also assumptions made by Helios.

- The voter, V, is a human and is able to:
  - read and compare short strings;
  - choose a candidate to vote for;
  - choose at random whether to cast or audit an encryption (Benaloh's challenge);

  - choose a random short string (this is required to secure the protocol against clash-attacks, but low-entropy strings are sufficient—selected strings need to be unique only across voting sessions active at that time).

  V need not be honest. In particular, *V* may make false complaints.
- An honest registrar, *R*, issues valid credentials, which are securely delivered to the voter through a channel that is not accessible to the voting terminal. The registrar does not share a voter's credentials with anyone other than the voter, and correctly identifies all purported credentials as being valid or not during and after the election, as necessary.
- A secure bulletin board—with append-only-authenticated-write and public-read access—is available to all participants.
- The voting terminal (including any software on it, referred to as *Voting Booth* (*VB*) in Helios) and the election authority (*EA*) (including servers and election officials, any software deployed by the election authority) are not assumed honest for the integrity properties, and may collude. This assumption takes into account the possibility of implementation vulnerabilities (like those described in Section 5).
- The protocol is not expected to provide privacy of the vote with respect to *VB* or *EA*, but the *EA* may be split to provide some privacy.
- The voter may have access to one computational device other than the voting terminal (we refer to such a device as a voting assistant, *VA*) which helps her check on *VB* and *EA*. This device should not learn the vote.
- The voter may have access to n such devices, denoted VA_1, VA_2, ..., VA_n, which she uses to make the checks required by the protocol. The probability with which she makes an incorrect estimate of the correctness of a check using these devices is small. We explicitly include multiple devices here to allow for the possibility of dishonest devices, though our protocol works for $n = 1$.

The full version of the Apollo protocol assumes a threat model exactly like the above, except *R* may share valid credentials with an adversary, or try to use them to cast a vote. We term this the *full threat model.*

3.2 Voter Experience

In this section we present the voter experience.
Credentials: *V* receives her credentials from *R*: a set of k *casting codes* and a *lock-in code.*

Pre-Voting Phase: Before beginning the voting session, *V* chooses n voting assistants VA_1, VA_2, ... VA_n. She chooses n based on the maximum acceptable probability of not detecting a cheating *EA* or *VB*. If she chooses $n = 0$, her ability to detect cheating will be limited (just as in the case of Helios)[3].

[3] Apollo is designed so that the terminal cannot tell whether $n = 0$ or $n > 0$.

Role of Voting Assistants: After each protocol step, each *VA* checks *BB* and provides feedback to *V*. If *V* is satisfied with the outcome of the check, she moves to the next step. *V* may choose to require that a majority of the *VA* present the same information, or she may require that they all do, or she may choose another rule to determine whether the check demonstrates a problem. If she determines that there is a problem, she should immediately abort the protocol, change the computer running *VB* and try to vote again. She should always (reuse) an old credential unless she hears back from the *EA* that it has been used.

Voting Phase:

1. *V* opens the voting application on *VB*, which asks her to provide a short string for the session title. She enters the string. *VB* displays the (voting) session ID and a *QR*-code. *BB* displays the (voting) session ID, see Figure 1 and Step 5 on Figure 2.
2. *V* scans the *QR*-code into all the other voting assistants, and checks that they display the session ID and Title is displayed on *VB* (step 8 on Figure 2).

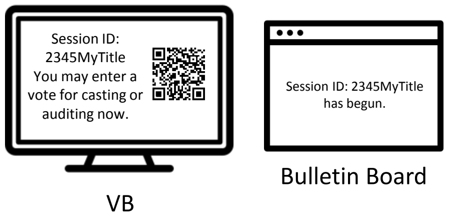
Session ID:
2345MyTitle
You may enter a vote for casting or auditing now.
Session ID: 2345MyTitle has begun.
VB
Bulletin Board

Fig. 1. Voter initializes session

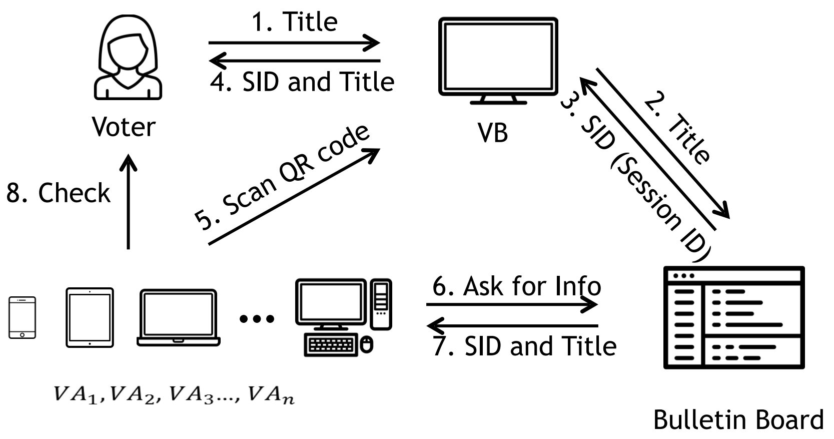

Fig. 2. Voting Assistants check Bulletin Board and inform a voter about the SID and the Title.

3. V enters a vote for candidate X. BB displays the encrypted vote and VB and each VA inform her that the encrypted vote is displayed, and she should now enter an audit or cast request (see Figure 3.

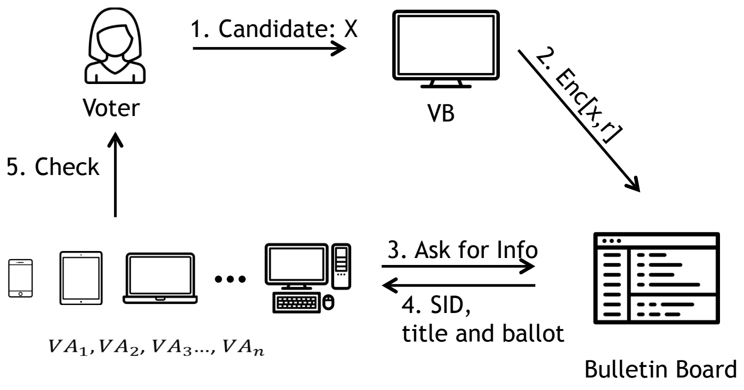

Fig. 3. Encryption is Posted

4. If the voter enters a cast code, each VA displays the code she entered and informs her that her vote is ready for locking.

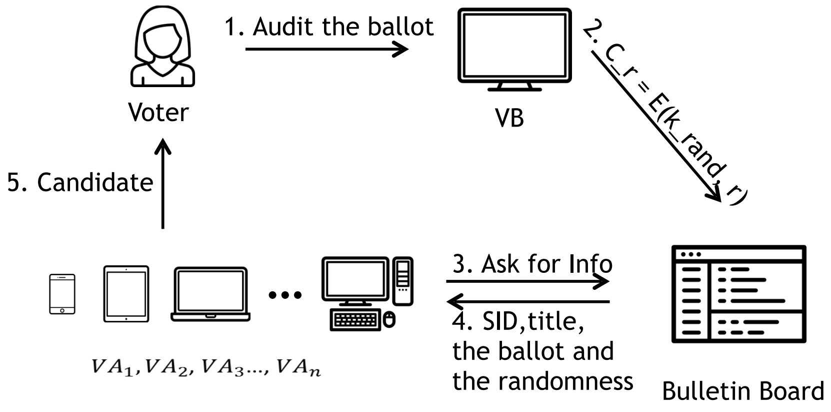

Fig. 4. Voter Chooses to Audit the Vote

5. If the voter enters an audit request, each VA informs her that the encrypted string has been audited and shows a vote for candidate X (see Figure 4). The voter may repeat the audit step as many times as she wishes.

Lock-in Phase

The voter may return at any time to lock-in her vote, and she may do so from any computer by identifying her session ID and adding her lock-in code (see Figure 5). She may check that the code has been posted, again, from any (other) computer.

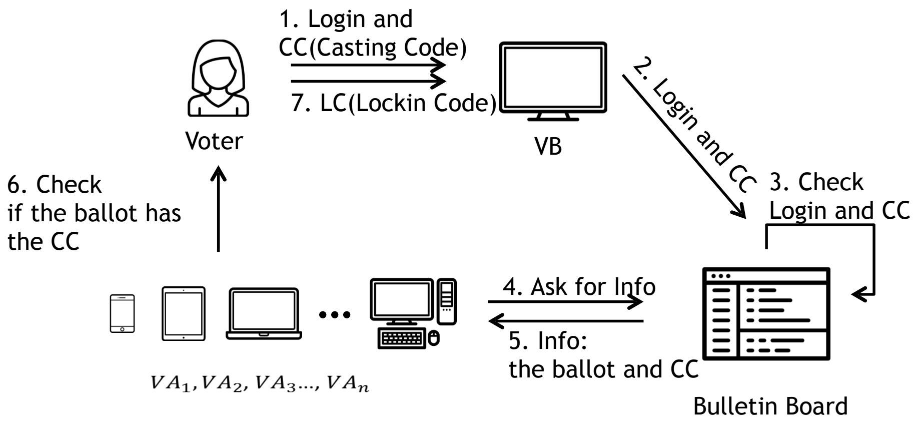
1. Login and CC(Casting Code)
7. LC(Lockin Code)
Voter
VB
2. Login and CC
3. Check Login and CC
6. Check if the ballot has the CC
4. Ask for Info
5. Info: the ballot and CC
$VA_1, VA_2, VA_3..., VA_n$
Bulletin Board

Fig. 5. Voter Chooses to Cast the Vote

3.3 Informal Protocol Description

All interactions among voting assistants and the voting system are digitally signed and posted on the BB. The voter may only post instructions on the BB through a voting assistant. The protocol proceeds as follows.

V interacts with VB to generate an encrypted ballot; this ballot is posted on BB. VB displays a QR-code containing a session ID and a session symmetric key, and a human-readable version of session ID. The voter scans the QR-code onto all n VAs, which each display the session ID. The voter compares it with the one on VB.

Each VA checks BB and indicates to V whether a string is posted for the session. Once V is satisfied that it is, she enters a cast code or audit instruction into VB, which is posted on the BB.

If the code is a cast code, the registrar signs the encrypted ballot with the signing key for cast ballots and posts it on the BB. Each VA checks BB and displays the cast code posted for the session, as well as the fact that a signed encrypted ballot has been posted against the cast code which has been accepted as valid by the registrar. The voting session ends. When a voting session ends with the submission of a cast code accepted as valid, a confirmation email containing: sessionID, session title, cast code and a list of identifiers of audited and cast ballots (together with time stamps of arrival) is sent to V.

If the code is an audit code, VB opens the encryption by posting the randomness encrypted with the session key. Each VA checks BB and displays the plaintext value. V may repeat this audit process as often as she wishes.

After casting her vote and receiving the conformation email, if V is satisfied, she supplies the lock code from any computer by using the session ID. She should then check that it has been correctly posted, from any (other) computer. If not, she attempts to lock-in again.

All locked votes are tallied in a verifiable manner.

The Apollo casting and lock-in procedures are described in detail in Protocol 1.1.

4 Security analysis

In this section, we analyze the security properties of Apollo with respect to common attacks.

4.1 Privacy

In Apollo, voters may lose ballot privacy through information that is (a) posted to the bulletin board, (b) provided to the voting terminal, (c) obtained by the voting assistants.

Bulletin Board (*BB*) Apollo uses two different encryption schemes for posting vote-related information on the bulletin board: an asymmetric-key encryption scheme for encrypting ballots (*e.g.,* the same scheme as in Helios) and a symmetric-key encryption scheme for encrypting randomness. We follow a series of works [2,5,6,9] suggesting the correct choice of ballot encryption and ZKP-proofs, so that these do not leak the vote to the public; the symmetric-key encryption proposed for use is the authenticated mode of operation of AES.

The privacy of data on the bulletin board thus depends on the security of the symmetric and asymmetric-key encryption schemes used, which depends on the splitting of the EA into trustees (there is no privacy with respect to the combined EA), on the secrecy of the keys of trustees and on whether the collusion among trustees is within the limits of the secret-sharing scheme used.

Note here that the public does not learn the audited vote as the encryption randomness is not posted in the clear when the vote is audited. Through the qr-code, the voter controls the VAs with access to the symmetric-key used to encrypt the encryption randomness.

Voting Booth (*VB*) VB is the only party of the system that directly learns the voter's choice. It also knows the randomness that is used to encrypt the ballot. VB may reveal the vote to anyone; with the presented version of Apollo, as with Helios, this is inevitable.

Apollo: casting

1. *VB* generates a key pair, publishes this on *BB* before the voting session begins.
2. *V* initiates the voting session on *VB*, and is asked to enter a short string, $MyTitle$.
3. *VB* displays:
   (a) A qr-code which contains: k_{rand} (a secret key for symmetric encryption), $sessionID$ (a string with $MyTitle$ appended), signed with its key. This qr-code is intended as communication between *VB* and any *VA*s the voter chooses; it may be stored and/or printed.
   (b) Human-readable $sessionID$
4. *V* checks that $MyString$ forms the last part of $sessionID$. She scans the qr-code with multiple *VA*s.
5. *VA*s check the *BB* and look for the $sessionID$, obtain the public key of *VB*, display $sessionID$.
6. *V* verifies whether $sessionID$ presented by *VB* and *VA*s is the same.
7. *V* sends vote choices to *VB*: $V \xrightarrow{x} VB$
8. *VB*does the following:
   (a) computes the encryption of the ballot: $c \leftarrow \mathsf{Enc}(x, r)$, where r is the randomness used during encryption,
   (b) sends the encrypted vote to *BB*: $VB \xrightarrow{c} BB$
9. *VA*s inform the voter that c is posted on *BB* in the transcript of her $sessionID$
10. *V* makes a decision about cast/audit:
    Audit is selected:
    (a) *VB* sends randomness $c_r = E(k_{rand}, r)$ used for encrypting c to *BB*
    (b) The *VA*s decrypt c_r and present the vote x' to *V*
    (c) *V* accepts or not based on what the other VAs say the vote decrypted to:
        $x = x'$ Prepares new encryption; goto step (7).
        $x \neq x'$ Begins again with new *VB* and, if necessary, *VA*s
    Cast is selected:
    (a) *V* is asked to enter: *Login* and *CastCode* (these can be combined to be a single long string)
    (b) *VA*s display the *Login*/*CastCode* pair; *V* checks if they are as expected.

Apollo: lock-in

1. *V* chooses a terminal and accesses the election website.
2. *V* enters her $sessionID$ and lock-in code.
3. *V* checks *BB* from another terminal. If *V* does not see the lock-in code, she attempts to lock-in again.

Protocol 1.1. The casting and lock-in procedures for Apollo.

Voting Assistant (*VA*) If we assume that the cast and audited votes are independent, any *VA* used by the voter learns nothing about the cast vote, because it gets all its information about it from the *BB*. It learns only the audited votes.

4.2 Integrity

We define three levels of security with respect to different attacks.

Level 1 E2E-V – the voter is able to detect an attack (but cannot prove it to a third party),

Level 2 Evidence of an attack – the voter is able to detect an attack and prove that the attack took place.

Level 3 Recovery: the voter is able to prevent or recover from the attack.

Level 1 corresponds to the end-to-end verifiability approach – the voter can detect that some of her directions were not followed but is unable to transfer this knowledge to a third party.

Level 2 lets the voter detect an attack and provide evidence to a third party that the protocol was not followed. We would like to say that this level corresponds to dispute-resolution [17,22] or accountability [18] but in the Internet voting setting it is almost impossible to assign blame. For many attacks, it may not be possible to determine whether they result from a dishonest election server or a malicious terminal, which is malicious because of a flaw in the lower-level library (like TLS/SSL allowing an attacker to subvert a terminal's code).

With Apollo, an adversary attempting to change a vote would have to do so before it was locked-in, in which case the voter would not lock it. If a dishonest voting system attempts to count a vote that is not locked-in, this will be detected by the public, and there is evidence (a non-locked-in-vote that is tallied) that the protocol was not followed. There is no other way to include a vote in the tally that is not authorized by the voter. Any errors in the vote tallying process also result in evidence through the tally-correctness proof.

There is always the question of what to do when one discovers that a voting system was the subject of a successful attack during the election (rerunning the election may be difficult, costly or impossible). When a system allows voters not only to detect that the protocol was not followed but also to recover from the "error" we obtain a robust, Level-3 solution. In the case of Apollo, a non-locked-in vote is not final, and can be replaced by the voter using another channel, perhaps by voting in person. Errors in the tally process can only be recovered from if the tallying server(s) cooperate.

4.3 Terminal misbehaviour

Changing the vote Benaloh's challenge protects the voter from *VB*'s attempts to change the vote before she submits her credentials. By itself, as implemented in Helios, it provides Level-1 security against *VB* stealing her credentials to cast another vote.

Stealing credentials In Apollo, too, *VB* may attempt to steal the credential (cast code) and post it against a new encryption of its own, either within the same voting session, or in a new session it begins for this purpose.

In the first case, if the voter is using a *VA*, it will inform her of a new encryption posted in her session, and of it being cast. If the voter does not use any *VA*s, she can detect that more encryptions were posted within her session by checking the bulletin board or by checking the confirmation email.

In the second case, if she is using a *VA*, it will not report the correct posting of the cast code. Additionally, the voter will not receive a confirmation email, and the *BB* will not display a successful cast vote, both of which can be detected without the use of a *VA*.

Thus, in either case, whether she uses a *VA* or not, she will notice that the cast session is not successful. She will then use a new terminal and new *VA*s if so indicated (maybe if they don't agree on the outcomes of the checks) to start the voting process again. She should use the same cast code, in general, (in case it was not used by the terminal). If it is rejected because it was used by the malicious previous terminal, she should then use a new cast code.

The voter's ability to successfully complete the cast session is limited by the number of cast codes issued. However, unlike Helios, the lack of access to an honest terminal results in a denial of service and not a change of vote.

4.4 Clash-Attack resistance

Because voters choose part of the session ID of their own sessions and it is displayed by the *VA*s, each voter is able to detect the situation when two terminals attempt to generate the same receipt for her and another voter. While the quality of randomness used by voters to generate a session-title can be poor, this should be sufficient to protect against clash attacks that need to happen at about the same time (during the active voting session) when voters are using *VA*s. This helps protect those voters who do not use *VA*s as well, because *VB* does not know if a voter is using a *VA* or not.

A clash attack can be successful only when: (a) (at least) two voters, who begin their voting sessions at about the same time, pick the same session title (while their terminals collude) and (b)the voter who enters her *cast code* later does not notice that it was not correctly displayed on her *VA*.

From the birthday paradox the probability of such an event is $\geq 1/2$ when at least $\sqrt{2^l}$ voters start their sessions "at the same time" and l is min-entropy for their session-titles. It hence depends on the size of the alphabet and the length of the session-title (and the ability of voters to compare strings).

Even voters who do not use voting assistants are able to detect the attack by checking session titles and cast codes, and/or by verifying if the(signed) confirmation email contains the correct information.

4.5 Credential Distribution

Apollo does not restrict the format of credentials. Here we describe the security benefits of using ways of distributing credentials other than by email (which is the default in Helios).

Credentials in the form of printed codes hidden under a scratch-off layer provide security against a dishonest Registrar, who might post a vote against a voter's credential. In such a case, the voter has evidence of vote manipulation because she can display an unscratched surface over her lock-in or cast codes.

If one may assume the ability of the voter to sign commands (in a manner similar to [8]) then digital signatures under commands "cast" and "lock in" can be used instead of codes generated by the authority.

5 Evaluation of Helios implementation

In this section we describe our findings of security-related problems in the Helios implementation (*i.e.,* in `helios-server/heliosbooth`, source code which we refer to was used between May 1, 2014 and December 21, 2015). A description of our findings together with proposed solutions was sent to the Helios team who patched the code in January 2016 (pull requests #111 and #112) and May 2016 (pull request #110).

5.1 Cross-Site Scripting

Description Helios Booth takes a parameter named `election_url` whose value is a link to a micro-service that sends data in JSON format for the election given an identifier. Based on that data, it builds a form.

Let us take a look at the code responsible for initialization, see listing 1.1.

/heliosbooth/vote.html

```
BOOTH.so_lets_go = function () {
    BOOTH.hide_progress();
    BOOTH.setup_templates();
    // election URL
    var election_url = $.query.get('election_url');
    BOOTH.load_and_setup_election(election_url);
};
```

Listing 1.1: A fragment of Helios Booth responsible for initialization of app modules.

Function `so_lets_go` is executed just after the HTML is loaded. After templates are initialized the GET variable `election_url` is passed to a function `load_and_setup_election`.

To obtain the GET a jQuery method `$.query.get` was used. At this step the obtained parameter is not checked/verified, but is treated as a trusted one – this opens up the possibility for an XSS attack. The parameter is is not checked in any further step, see listing 1.2.

```
/heliosbooth/vote.html
368 BOOTH.load_and_setup_election = function(election_url) {
369     // the hash will be computed within the setup function call now
370     $.get(election_url, function(raw_json) {
371         // let's also get the metadata
372         $.getJSON(election_url + "/meta", {}, function(election_metadata) {
373           BOOTH.election_metadata = election_metadata;
374           BOOTH.setup_election(raw_json, election_metadata);
375           BOOTH.show_election();
376           BOOTH.election_url = election_url;
377         });
378     });
```

Listing 1.2: A code of Helios Booth responsible for retrieving election information data.

The `election_url` variable is treated as an election URL (see lines 370 and 372). In these lines AJAX queries are sent to the URL defined in `election_url`. All data received is in JSON format and contains: keys, election questions, etc. The problem is that `election_url` may point to a service which is under the control of an attacker.

If this is the case then this malicious service has full control over the data that is passed to the Helios Booth. It, for instance, can play the role of a proxy.

The security vulnerability is caused by the method `$.getJSON` (line 372) – which is a part of jQuery library and is similar to `$.get` method: it performs asynchronous HTTP GET but unlike `$.get` it treats the response as data in JSON or JSONP format (default: JSON) and on receiving it parses it into a JavaScript object. In jQuery library before the version 1.2.3 there was a bug which had the following result: upon querying non-relative URL each response was treated as JSONP (executable JavaScript). Helios Booth was using version 1.2.2 which was vulnerable to this.

The parameter `election_url` was supposed to contain a relative URL but if an attacker used a modified URL leading to the attacker's proxy it would result in the attacker's ability to execute any arbitrary JavaScript code in the voter's browser. It was enough that proxy would answer to a query of `/meta` resource with a JavaScript code.

So the vulnerability can be treated as non-persistent Cross-Site Scripting (A3 from OWASP Top 10).

Exploiting vulnerability In order to take advantage of non-persistent Cross-Site Scripting, an attacker needs to make a victim start a voting app with a modified URL.

Then one possibility would be to correctly encrypt every voter choice (to pass each of the Cast/Audit steps) but when the voter decides to submit a ballot, the attacker prepares a new ballot and casts it instead of voter's ballot.

This vector of the attack is impossible to be detected from the server's side. It can still be detected by the voter but only in the situation when the voter: (1) remembers the tracker of the cast ballot and (2) checks the bulletin board later. Various experiences and studies suggest that the (2) check is not performed often enough [20,7], and what is even worse the fraction of voters who discover the discrepancies and report them can be as low as 0.5%.

Remedy We suggested to (1) replace jQuery library with a newer version and (2) to introduce filtering the `election_url` not to allow non-relative URLs.

Another, more general, suggestion to make the system immune against Cross-Site Scripting we suggest is to introduce Content Security Policy [24] in the most rigorous form `default: self-src`. This would require changes in HTML, CSS and JavaScript.

5.2 Cross-Site Request Forgery

We found that some of the key functions of the system are not secured against the CSRF. This could easily lead to the situation when an election admin (logged in) can be tricked to perform an action that was not intended.

Vulnerability description We found a few methods which are executed (both GET and POST) without necessary checks. Actions not immune to CSRF attacks are listed in the table 1 (This type of attack is at position 8 in OWASP Top 10).

Action	**Query type**	**Relative url of the method**
Election creation	POST	/helios/elections/new
Election edition	POST	/helios/elections/:election_id/edit
Archiving elections	GET	/helios/elections/:election_id/archive?archive_p=1
Canceling archiving elections	GET	/helios/elections/:election_id/archive?archive_p=0
Featuring elections	GET	/helios/elections/:election_id/set_featured?featured_p=1
Canceling featuring elections	GET	/helios/elections/:election_id/set_featured?featured_p=0
Adding a trustee	POST	/helios/elections/:election_id/trustees/new

Table 1. List of methods in Helios vulnerable to Cross-Site Request Forgery attacks.

Exploiting CSRF To exploit a vulnerability, an attacker would need to (1) create a website with self-sending POST or GET query to one of the unsecured methods (2) make a user with admin privileges visit the site.

Lifetime of Helios cookies are set to 14 days so the attack would have been successful if a victim was logged into an admin console within this period of time.

Most of the vulnerable methods cannot do much more than a denial of service. Methods that allow the addition of trustees to given elections, however, can lead to loss of ballot privacy.

5.3 Framework exploits

Framework exploits is the vector of attacks that lets one attempt to use a vulnerability of the method of the underlying library to attack a given system. Helios relies on the Django framework, so any vulnerable Django method used in Helios can also create a vulnerability.

Description Helios used Django 1.6 till October 4, 2015 while the support for this branch ended on April 1, 2015. Thus, for about 186 days Helios was not protected by the patches applied to Django. Beginning October 4, 2015, Helios has been using Django 1.7.10 but this version has not been supported since December 1, 2015. Just in 2015 there were 14 vulnerabilities discovered in Django [10].

Exploiting At the time of our audit no publicly open vulnerability of Django was known. But taking into account the types of security weaknesses, about one third of the discovered issues allowed for the performance of a denial of service attack. An attacker could have selectively disallowed voters to cast their ballots by blocking the server.

5.4 Clickjacking

Clickjacking is an attack that takes advantage of a user who thinks she clicks on an element (*e.g.,* button, link) of an app, while, thanks to the use of invisible layers, the action is linked with an element provided by an attacker.

Description Every page of the Helios app can be placed in `<iframe>` which can lead to clickjacking attacks.

Exploiting As with other attacks, one needs to use socio-engineering techniques to convince a voter to visit the site prepared by the attacker. This can be used, for instance, for early-finishing of the elections (if an attacked person has admin privileges).

Remedy In order to exclude the possibility of clickjacking attacks on Helios we suggested to use HTTP Header `X-Frame-Options: SAMEORIGIN` which disallows the embedding of an app within iframes that are hosted on a different server. Django has a built in middleware `XFrameOptionsMiddleware` that takes care of sending the correct header [12].

6 Conclusions

We presented possible consequences of attacks on Helios. We also proposed an end-to-end verifiable Internet voting scheme Apollo which enables the voter to detect and correct problems in the representation of her vote. Apollo can also be used to provide evidence of vote manipulation. Additionally, Apollo offers a higher level of protection against a number of attacks (*e.g.,* clash-attacks, credentials stealing) than does, for example, Helios. We proposed an easier way to integrate the use of voting assistants, requiring the scanning of a single QR-code. Other proposals require the scanning of $2k$ codes for k audited ballots (a scan each for reading the commitment and checking encryption-correctness).

Interesting future directions include usability testing of the protocol, and an open problem is whether the credential stealing problem can be addressed with simpler protocols.

References

1. B. Adida. Helios: web-based open-audit voting. In *USENIX Security Symposium*, pages 335–348, 2008. 1, 2
2. B. Adida, O. De Marneffe, O. Pereira, J.-J. Quisquater, et al. Electing a university president using open-audit voting: Analysis of real-world use of helios. *EVT/-WOTE*, 9:10–10, 2009. 2, 4.1
3. J. Benaloh. Simple verifiable elections. In *EVT*, 2006. 1
4. J. Benaloh, M. Byrne, P. T. Kortum, N. McBurnett, O. Pereira, P. B. Stark, and D. S. Wallach. STAR-Vote: A secure, transparent, auditable, and reliable voting system. *CoRR*, abs/1211.1904, 2012. 1
5. D. Bernhard, V. Cortier, O. Pereira, B. Smyth, and B. Warinschi. Adapting Helios for provable ballot privacy. In *Computer Security–ESORICS 2011*, pages 335–354. Springer, 2011. 4.1
6. D. Bernhard, O. Pereira, and B. Warinschi. How not to prove yourself: Pitfalls of the fiat-shamir heuristic and applications to helios. In *Advances in Cryptology–ASIACRYPT 2012*, pages 626–643. Springer, 2012. 4.1
7. R. T. Carback, D. Chaum, J. Clark, J. Conway, A. Essex, P. S. Hernson, T. Mayberry, S. Popoveniuc, R. L. Rivest, E. Shen, A. T. Sherman, and P. L. Vora. Scantegrity II Municipal Election at Takoma Park: The First E2E Binding Governmental Election with Ballot Privacy. In *USENIX Security Symposium*, 2010. 1, 5.1
8. V. Cortier, D. Galindo, S. Glondu, and M. Izabachne. Election verifiability for helios under weaker trust assumptions. In M. Kutyowski and J. Vaidya, editors, *Computer Security - ESORICS 2014*, volume 8713 of *Lecture Notes in Computer Science*, pages 327–344. Springer International Publishing, 2014. 4.5
9. V. Cortier and B. Smyth. Attacking and fixing Helios: An analysis of ballot secrecy. *Journal of Computer Security*, 21(1):89–148, 2013. 4.1
10. C. Details. Django: List of security vulnerabilities. Technical report, MITRE's CVE web site, 2015. 5.3
11. S. Estehghari and Y. Desmedt. Exploiting the client vulnerabilities in internet e-voting systems: Hacking Helios 2.0 as an example. In *EVT/WOTE*, 2010. 2

12. D. Foundation. Clickjacking protection in django. Technical report, Django Software Foundation, 2015. 5.4
13. K. Gjosteen. Analysis of an internet voting protocol. Technical report, IACR Eprint Report 2010/380, 2010. 1, 2
14. G. S. Grewal, M. D. Ryan, L. Chen, and M. R. Clarkson. Du-vote: Remote electronic voting with untrusted computers. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, pages 155–169, 2015. 1.1
15. J. A. Halderman and V. Teague. The new south wales ivote system: Security failures and verification flaws in a live online election. In *E-Voting and Identity - 5th International Conference, VoteID 2015, Bern, Switzerland, September 2-4, 2015, Proceedings*, pages 35–53, 2015. 1, 2
16. M. Heiderich, T. Frosch, M. Niemietz, and J. Schwenk. The bug that made me president a browser-and web-security case study on helios voting. In *E-voting and identity*, pages 89–103. Springer, 2012. 2
17. A. Kiayias and M. Yung. The vector-ballot e-voting approach. In *Financial Cryptography*, 2004. 4.2
18. R. Kusters, T. Truderung, and A. Vogt. Accountability: Definition and relationship to verifiability. In *CCS*, 2010. 4.2
19. R. Kusters, T. Truderung, and A. Vogt. Clash attacks on the verifiability of e-voting systems. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 395–409. IEEE, 2012. 2
20. E. Moher, J. Clark, and A. Essex. Diffusion of voter responsibility: Potential failings in e2e voter receipt checking. *USENIX Journal of Election Technology and Systems (JETS)*, 1:1–17, 2014. 1, 5.1
21. S. Neumann, M. M. Olembo, K. Renaud, and M. Volkamer. Helios verification: To alleviate, or to nominate: Is that the question, or shall we have both? In *Electronic Government and the Information Systems Perspective*, pages 246–260. Springer, 2014. 2
22. S. Popoveniuc, J. Kelsey, A. Regenscheid, and P. Vora. Performance requirements for end-to-end verifiable elections. In *Proceedings of the 2010 international conference on Electronic voting technology/workshop on trustworthy elections*, pages 1–16. USENIX Association, 2010. 4.2
23. D. Springall, T. Finkenauer, Z. Durumeric, J. Kitcat, H. Hursti, M. MacAlpine, and J. A. Halderman. Security analysis of the estonian internet voting system. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 703–715, New York, NY, USA, 2014. ACM. 1, 2
24. M. West, A. Barth, and D. Veditz. Content security policy level 2. Last call WD, W3C, July 2014. 5.1
25. S. Wolchok, E. Wustrow, D. Isabel, and J. A. Halderman. Attacking the Washington, D.C. internet voting system. In *Financial Cryptography*, 2012. 1, 2
26. F. Zagórski, R. T. Carback, D. Chaum, J. Clark, A. Essex, and P. L. Vora. Remotegrity: Design and use of an end-to-end verifiable remote voting system. In *Applied Cryptography and Network Security*, volume 7954. Springer, 2013. 1.1, 1.1