The Cloudier Side of Cryptographic End-to-end Verifiable Voting: A Security Analysis of Helios

Nicholas Chang-Fong
Department of Electrical and Computer Engineering
Western University, Canada
nchangfo@uwo.ca

Aleksander Essex
Department of Electrical and Computer Engineering
Western University, Canada
aessex@uwo.ca

ABSTRACT

Helios is an open-audit internet voting system providing cryptographic protections to voter privacy, and election integrity. As part of these protections, Helios produces a cryptographic audit trail that can be used to verify ballots were correctly counted. Cryptographic end-to-end (E2E) election verification schemes of this kind are a promising step toward developing trustworthy electronic voting systems.

In this paper we approach the discussion from the flip-side by exploring the practical potential for threats to be introduced by the presence of a cryptographic audit trail. We conducted a security analysis of the Helios implementation and discovered a range of vulnerabilities and implemented exploits that would: allow a malicious election official to produce arbitrary election results with accepting proofs of correctness; allow a malicious voter to cast a malformed ballot to prevent the tally from being computed; and, allow an attacker to surreptitiously cast a ballot on a voter's behalf. We also examine privacy issues including a random-number generation bias affecting the indistinguishably of encrypted ballots. We reported the issues and worked with the Helios designers to fix them.

CCS Concepts

•**Security and privacy → Cryptanalysis and other attacks; Web application security;** *Privacy-preserving protocols;*

Keywords

Internet voting, cryptographic end-to-end verification, attacks

1. INTRODUCTION

Internet voting. Like flying cars it is standard fare in our collective vision of the future. The perceived advantages of internet voting typically center around otherwise reasonable goals like increasing voter turnout, reaching under-represented populations, and decreasing election costs. Although these advantages themselves have been widely debated, the real reason we don't all vote online already is because, simply put, internet voting is a *really* hard security challenge. As a simplification of rather a complex problem, the reason internet voting is harder than other security systems comes down to the tension between ballot secrecy and election integrity. Consider that when you bank online, both you and your bank have a record of your account totals and transactions. So when something goes wrong, at least there's a shared starting point with which to pursue the correction.

Internet voting by contrast does not typically have a straightforward way to answer questions about many of its critical functionalities such as "did my vote count?" In its most basic form, contemporary commercial internet voting systems consist of a standard web-application framework; Javascript is delivered to a client over TLS, and the ballot is returned to the server and stored in a database. Other systems offer some degree of verifiability,[1] but have variously been shown to have weak procedural security [29, 32], TLS configuration errors [30], command-line injection vulnerabilities [33], etc.

Cryptographic End-to-end Verifiable Voting.

One promising paradigm to tackle these challenges is cryptographic end-to-end (E2E) election verification [20], which produces a universally verifiable cryptographic proof, allowing (a) any voter to confirm the inclusion of their vote in the overall tally, and (b) anyone to confirm the tally was counted and decrypted correctly. Unlike conventional elections that focus on procedural controls, E2E verification focuses on providing evidence, exploiting the unique ability of cryptography to be able to prove statements without revealing any information beyond the truth of the statement itself. Applied in the voting context that means proving the correctness of the outcome without requiring direct access to the voting intention of individuals.

The E2E literature is extensive, so we will briefly mention a few related implementations. Chaum *et al.* [13] developed Scantegrity, a scheme for paper optical-scan ballots and notably ran the first E2E-verifiable governmental election in the City of Takoma Park, MD in 2009 [12]. They repeated the election 2011 with an internet voting add-on, Remotegrity [34]. Ryan *et al.* proposed another paper optical-scan scheme, Prêt à Voter [15]. Recently Burton *et al.* [10, 11] adapted and deployed a variant in the Australian state

ACSAC '16, December 05 - 09, 2016, Los Angeles, CA, USA
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4771-6/16/12…$15.00
DOI: http://dx.doi.org/10.1145/2991079.2991106

[1] http://www.cs.cornell.edu/~clarkson/papers/scytl-odbp.pdf
https://www.verifiedvoting.org/resources/internet-voting

election of Victoria. Bell *et al.* [7] proposed StarVote for Travis County, TX. Delis *et al.* [18] piloted an end-to-end code voting system during the 2014 European Elections in Greece.

Helios [3] is an E2E verifiable internet voting system, and the subject of this paper. We decided to focus on Helios for several reasons. It is one of the oldest E2E implementations, and at over 500,000 ballots cast, is by far the largest. It is among the only such implementations to have been actively maintained and still in continuous use. Finally, it is perhaps the only such implementation that could be used to run an election without any involvement of the software designers. This creates an interesting threat model for us to explore: an automated election-as-a-service in which the code is being executed as-is, underscoring the the idea that vulnerabilities are inherent to the implementation, and not simply the result of malicious alterations to the code.

Contributions. Our contributions are as follows:

- A security analysis of Helios uncovering a number of vulnerabilities to confidentiality, integrity and availability,
- An implementation of exploits demonstrating:
  - A malicious election official rigging an election by declaring an arbitrary election result, but issuing a cryptographic proof that the results were correctly tallied,
  - A malicious voter casting a maliciously formed ballot preventing the tally from being decrypted,
  - An XSS attack allow an attacker to cast a ballot on a voter's behalf.
- Discussion and lessons learned,
- Responsible disclosure and collaboration with the Helios designers to correct the issues presented in this paper.

The rest of the paper is organized as follows: Sections 2 and 3 provides background and preliminaries of the cryptography used by Helios. Section 4 presents cryptographic attack that (a) allow a voter to prevent a Helios tally from being computed and (b) allow an malicious election official to produce arbitrary tallies, with accepting proofs of correctness. Section 5 describes a RNG bias that undermining the semantic security of ballot encryptions. Section 6 presents the details of a ballot stealing cross-site scripting attack. Finally Section 7 concludes.

2. BACKGROUND AND RELATED WORK

2.1 Helios Open-Audit Internet Voting

As originally proposed by Adida [3], Helios used a mixnet-based proof for robustness. Later Adida *et al.* [4] moved to a homomorphic tally approach, in which encrypted votes are be summed under encryption (cf. Section 3). This scheme was used to run one of the first Helios elections, at the Université catholique de Louvain in 2008. Today, the Helios website claims to have been used to cast over half a million ballots, and some organizations have used it multiple times, including the Princeton Undergraduate Student Government (USG) since 2013[2] and the International Association for Cryptologic Research (IACR) since 2010.[3] The current actively maintained Helios code base[4] is an implementation of Helios protocol v3.[5] There have also been a number of proposals for Helios variants. Demirel *et al.* [19] propose a version offering everlasting privacy. Bluens *et al.* [9] revisit a mixnet-based approach to allow more expressive vote tallying schemes and explores the property of submission security. Tsoukalas *et al.* [31] also explore extending Helios to support other voting schemes, and provide an open-source implementation.

A number of papers have studied the security of Helios. Estehghari and Desmedt [21] propose an attack involving client-side malware, though the attack is admittedly outside Helios' stated threat model. Cortier and Smyth [16] identified an attack that would allow a voter to replay ballots and suggest a fix. Heiderich *et al.* [24] proposed a number of subtle client-side attacks to the web technology. Bernhard *et al.* [8] identify pitfalls in deciding what precisely to hash for the Fiat-Shamir heuristic, with implications to proof soundness. Küsters *et al.* [26] propose the notion of clash attacks in which a corrupt election authority issues multiple voters the same receipt toward the goal of undetectably modifying the tally. Finally, Karayumak *et al.* [25] and Acemyan *et al.* [2, 1] have examined the usability of Helios and found a variety of issues, and suggested a number of places for improvement in the voter and admin interfaces.

2.2 Protocol Overview

To motivate our discussion in the following sections provide a high-level overview of the phases of a Helios election. Other cryptographically end-to-end verifiable voting schemes proceed in similar high-level terms, albeit using varying cryptographic proof techniques.

Initial Setup. In the initial setup, the election officials set all the relevant election parameters, including the various ballot contests and candidates, voter lists, and election website, as well as setting generating the relevant cryptographic parameters including distributed- or threshold-shared public keys for ballot encryption/decryption.

Ballot Casting. The voter is directed to the election website and prompted to provide their login credentials. Within the browser the voter is presented with the various ballot contests and candidates. They mark their selections. Next, each of the voter's selections are separately encrypted using an additively homomorphic public key cryptosystem. For example, if a contest was between two candidates Alice, and Bob, and the voter selected Alice, the booth would produce two ciphertexts: the "Alice" ciphertext would be an encryption of 1, and the "Bob" ciphertext would be the encryption of 0. The booth then constructs a non-interactive cryptographic zero knowledge proof that the ciphertexts constitute a valid vote. Without loss of generality, a valid vote is one that casts either (a) a single vote for a single candidate, or (b) no vote for any candidate (i.e., an abstention).

Finally the voter casts their ballot by posting the ciphertexts and associated zero-knowledge proofs to the election website. The server verifies the proofs, and rejects the ballot if the proof is invalid. The voter retains a copy of their

[2]https://princeton.heliosvoting.org

[3]https://www.iacr.org/elections/2010

[4]https://github.com/benadida/helios-server

[5]http://documentation.heliosvoting.org

encrypted ballot as a privacy-preserving receipt, which they may refer to later during the cryptographic election verification phase.

Homomorphic Tally. When the election is complete, election officials produce the tally by homomorphically summing the ciphertexts cast for the respective candidates. For example, they would homomorphically sum all 'Alice' ciphertexts, decrypt the result, and issue a zero-knowledge proof that the decryption was correct. They would then repeat this for Bob. These vote totals, along with all encrypted ballots and all associated proofs are posted to a public bulletin board.

End-to-end Verification. The election can now be verified for correctness. The *end-to-end* nature of verification arises from the fact that each voter can:

1. Check their encrypted ballot was included in the collection of encrypted ballots,
2. Verify the zero-knowledge proofs of correctness in the collection of encrypted ballots,
3. Homomorphically re-compute the encrypted tally,
4. Verify the zero-knowledge proof of decryption and compare it to the reported outcome.

Recalling the goals of cryptographic end-to-end verification, the first check allows the voter to confirm the inclusion of their vote in the overall tally (Helios provides a verification step prior to ballot submission that allows the voter to verify their choices were correctly encrypted). Finally, the second, third and fourth checks allow anyone to confirm the tally was counted and decrypted correctly.

2.3 Threat Model and Assumptions

Helios does not attempt (nor claim) to protect against certain threats, such as over-the-shoulder coercion resistance, and client-side malware [3]. For the purposes of the attacks presented in this paper we attempt to provide a set of assumptions that fairly capture aspects of the threat model:

- **Semantic Security**. Bernhard *et al.* [8] demonstrated that Helios is non-malleable under chosen plaintext attack (NM-CPA), and by implication indistinguishable under chosen plaintext attack (IND-CPA). It should be computationally infeasible, therefore, for an adversary to guess how a voter voted with advantage based solely on the public audit trail.
- **Semi-Trusted Election Authority**. In this paper we do not consider attacks in which a malicious election authority attempts to recover voting preferences from encrypted ballots—this capability is assumed. Although Helios does in principle support multiple trustees with distributed decryption, the default and most common configuration is a single-trustee mode in which the Helios server has a copy of the election private key.
- **Completeness and Soundness**. Helios produces various non-interactive zero-knowledge proofs for the purposes of proving (a) a ballot is correctly formed, e.g., doesn't contain multiple votes, negative votes, etc., and (b) the election trustees decrypted the homomorphic tally correctly. We say a proof is *accepting* if a verification algorithm returns True, and *non-accepting* otherwise. We assume the Helios proofs are both *complete*, meaning (informally) that a verifier will accept valid proofs, and *sound* meaning a verifier will reject invalid proofs.
- **Election as a Service**. We analyze Helios from the perspective of an election-as-a-service meaning we don't consider exploits that could be achieved by altering the server-side code or by otherwise assuming the Helios service acts maliciously.

3. PRELIMINARIES

Let $\mathbb{G}_q$ denote a finite cyclic group of order q in which the discrete logarithm problem is assumed to be hard. Helios implements $\mathbb{G}_q$ over a finite field (as opposed to an elliptic curve), thus for the purposes of this paper let p, q be primes for which $q \mid (p-1)/2$, and $\mathbb{G}_q$ subgroup of $\mathbb{Z}_p^*$. Current NIST guidelines require $|p| \geq 2048$ bits and $|q| \geq 224$ bits, corresponding to the 112-bit security level [28]. Let $x \leftarrow_\$ \mathbb{Z}_q$ denote a value x sampled uniformly from the set of integers modulo q. Values $\langle a, b, c\rangle \in \mathbb{G}_q$ forms a Diffie-Hellman tuple if $\langle a, b, c\rangle = \langle g^x, g^y, g^{xy}\rangle$ for some integers $x, y \in \mathbb{Z}_q$.

Let $\langle \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}\rangle$ by a public-key encryption scheme. Helios uses the exponential variant of Elgamal:

$$\begin{aligned} \mathsf{Gen}(1^s) &\rightarrow g, y \in \mathbb{G}_q \text{ for } y = g^x,\ x \in_\$ \mathbb{Z}_q \\ \\ \mathsf{Enc}(m) &= \langle g^r, g^m y^r\rangle \text{ for } r \in_\$ \mathbb{Z}_q \\ &= \langle \alpha, \beta\rangle \\ \\ \mathsf{Dec}(\langle \alpha, \beta\rangle) &= (\alpha)^{-x}\beta \\ &= (g^r)^{-x} g^{m+xr} \\ &= g^m. \end{aligned}$$

In El Gamal's original description, a plaintext is encoded directly as an element in $m \in \mathbb{G}_q$, and $\beta = my^r$. Here instead we encode the message as an exponent of g, i.e., $\beta = g^m y^r$. As a downside, m must be recovered from g^m by computing the discrete logarithm, although this reasonably efficient when m is small. This provides us with the following additive homomorphism:

$$\begin{aligned} \mathsf{Enc}(a) \cdot \mathsf{Enc}(b) &= \langle g^{r_a} g^{r_b}, g^{a+xr_a} g^{b+xr_b}\rangle \\ &= \langle g^{r'}, g^{a+b+xr'}\rangle \\ &= \mathsf{Enc}(a+b). \end{aligned}$$

Helios uses this additive property to implement a homomorphic counter construction in which encrypted ballots can be homomorphically tallied by computing the product of their respective ciphertexts. Observe $\mathsf{Enc}(0) = \langle g^r, g^{rx}\rangle$, taken along with public key g^x form a DH tuple. Similarly if one were to claim $\langle \alpha, \beta\rangle = \mathsf{Enc}(m)$, then $\langle \alpha, \beta/g^m\rangle$ likewise forms a DH tuple. Helios makes extensive use of proofs of DH tuples due to Chaum and Pedersen [14], which use the standard three move commit-challenge-response flow. The proof is made non-interactive by the heuristic due to Fiat and Shamir [23]. To provide soundness, the Fiat-Shamir heuristic generates the challenge value by hashing a context, which typically would include the proof's inputs, and ideally information about the statement being proven. Bernhard *et al.* [8] suggest how best to select an appropriate context for Helios. Given two Boolean statements S_1, S_2 a logical disjunction of the form $S_1 \vee S_2$ is accomplished following the strategy of Cramer *et al.* [17] in which one proof is *real*, and the other is *simulated*, where the proof is executed out of the usual order, allowing the prover to pick the challenge before making the commitment, thereby allowing them fake the proof.

The Fiat-Shamir heuristic is used to generate an overall challenge $\mathsf{c_{overall}}$, and prover can be forced to produce at least one real challenge $\mathsf{c_{real}}$ by the verifier enforcing:

$$\mathsf{c_{real}} + \mathsf{c_{sim}} = \mathsf{c_{overall}}.$$

The Helios Ballot.

Without loss of generality consider a contest between two candidates, Alice and Bob. The voter is allowed to vote for *up to* one candidate *i.e.,* may vote for either Alice, Bob, or neither (i.e., abstain). Helios uses a homomorphic counter approach in which the voting preference, $v \in \{0, 1\}$, is encrypted separately for each candidate. To indicate a vote *for* Alice, the voter set $\mathsf{Enc_{Alice}} = \mathsf{Enc}(1)$, otherwise the voter would set $\mathsf{Enc_{Alice}} = \mathsf{Enc}(0)$. Let $\mathsf{Enc_{Alice}}$ and $\mathsf{Enc_{Bob}}$ be encryptions of votes for Alice and Bob respectively. The voter then issues three disjunctive non-interactive proofs that the encryptions are well formed, i.e.,

- $\pi_1 = \big(\mathsf{Enc_{Alice}} = \mathsf{Enc}(0)\big) \vee \big(\mathsf{Enc_{Alice}} = \mathsf{Enc}(1)\big)$
- $\pi_2 = \big(\mathsf{Enc_{Bob}} = \mathsf{Enc}(0)\big) \vee \big(\mathsf{Enc_{Bob}} = \mathsf{Enc}(1)\big)$
- $\pi_3 = \big(\mathsf{Enc_{Alice}} \cdot \mathsf{Enc_{Bob}} = \mathsf{Enc}(0)\big) \vee$ $\big(\mathsf{Enc_{Alice}} \cdot \mathsf{Enc_{Bob}} = \mathsf{Enc}(1)\big)$

The Helios ballot is the tuple $\langle \mathsf{Enc_{Alice}}, \mathsf{Enc_{Bob}}, \pi_1, \pi_2, \pi_3 \rangle$.

To homomorphically tally the votes for each candidate, the election officials multiply the respective ciphertests. Suppose $\mathsf{Enc^{i}_{Alice}} = \mathsf{Enc}(v_1), \ldots, \mathsf{Enc^{n}_{Alice}} = \mathsf{Enc}(v_n)$ are all the encrypted Alice votes received during the election. Alice's vote total is homomorphically tallied as:

$$\prod_{i=1}^{n} \mathsf{Enc^{i}_{Alice}} \quad = \quad \mathsf{Enc}\Big(\sum_{i=1}^{n} v_i\Big).$$

The same process is used to homomorphcally tally Bob's votes. Each counter is decrypted and, of course, the candidate with the greatest number of votes wins.

4. CRYPTOGRAPHIC ATTACKS

Recall that Helios works in the cyclic group $\mathbb{G}_q$ and assumes the discrete logarithm problem is hard. This assumption not only typically requires q to be large and prime, but that any group elements are in $\mathbb{G}_q$. Unlike a safe prime group in which almost exactly half the values in the range $2 \ldots p - 1$ are in a group of order q, Helios uses a 256-bit subgroup of a 2048-bit group $\mathbb{Z}_p^*$. The probability a random element is in $\mathbb{G}_q$, therefore, is:

$$P(x \in \mathbb{G}_q | x \in_{\$} \mathbb{Z}_p^*) = \frac{1}{2^{1792}}$$

This would seem to provide a lot of opportunity for a kleptographic channel, such as a voter trying to encode additional information into another subgroup of their ballot ciphertext. This is where the correctness proofs come into play. All things considered, they do a good job of detecting problems within the confines of $\mathbb{G}_q$.

As we discovered, however, Helios does *not* check the underlying assumptions about the group, or element membership in the group, which allowed us to construct the following attacks.

4.1 Poison Ballot Attack

In this attack a disgruntled voter wishes to disrupt the election by preventing the tally from being computed by submitting a maliciously formed "poison" ballot. Normally a voter encrypts their preference for each candidate and constructs the associated proofs. Suppose the voter wishes to vote for Alice. An honest voting client would compute the following encryption:

$$\mathsf{Enc_{Alice}}(1) = \langle g^r, g^{1+rx} \rangle \tag{1}$$

Suppose the homomorphic tally of votes for Alice is a ciphertext $\langle \alpha_s, \beta_s \rangle$. As the first step of decryption, the election official would compute $\alpha_s^x = (g^{r_s})^x$, and prove the correctness by proving $\langle g, \alpha_s, y, \alpha_s^x \rangle$ forms a DH tuple. An attacker, however, could cause this proof to fail by intentionally forcing the tuple to *not* be a DH tuple. The only degree of freedom the voter has is their encrypted ballot, which is protected by the proof of correctness.

Exploit. The idea here is to select a generator h of a subgroup of order k such that $k | p - 1$. The Helios prime p provides us with with a number of subgroups to choose from, but for efficiency we selected h to have order $k = 2$. The malicious voter then computes the encryption:

$$\mathsf{Enc_{Alice}}'(1) = \langle hg^r, g^{1+rx} \rangle \tag{2}$$

The next problem to malicious voter must overcome is the proof of correctness. As shown in Figure 1 it must be the case that

$$g^{\mathsf{r_{real}}} \overset{?}{=} (g^r)^{c_{real}} A$$

but instead we have

$$g^{\mathsf{r_{real}}} \overset{?}{=} (hg^r)^{c_{real}} A$$

But if we were to have a situation in which $c_{real} \equiv 0 \bmod k$, then $h^{c_{real}} \equiv 1 \bmod p$, meaning the h term effectively disappears. In the simulated proof the attacker can directly cause c_{sim} to have this property. The attacker then checks if $c_{real} \equiv 0 \bmod k$. If so they proceed, otherwise they rewind the proof and try again. Following the same strategy they must also ensure that the h term disappears in the summation proof. With a valid-looking proof complete, the voter submits the poisoned ballot and waits. Once again let the homomorphic sum be $\langle \alpha_s, \beta_s \rangle$. The election official now computes $\alpha_s^x = (hg^{r_s})^x$. If $x \not\equiv 0 \bmod k$ then it is easy to see

$$\langle g, \alpha_s, y, \alpha_s^x \rangle = \langle g, hg^{r_s}, hg^{xr_s} g^x \rangle,$$

which is clearly not a DH tuple, meaning the verification of the decryption proof will fail. If $x \equiv 0 \bmod k$, however, then the h term will disappear as it did in the proofs, and the decryption will verify. This happens with probability $1/k$, and thus k can be adjusted to make the desired outcome as likely as possible depending on the subgroup options of p.

Impact and Mitigation. We implemented the attack and confirmed it would prevent the tally from being decrypted. The impact is high since any eligible voter can perform this

Simulated proof of DH tuple for $\langle g, g^r, g^x, g^{rx} \rangle$

Prover		**Verifier**
$\mathsf{c}_{\mathsf{sim}}, \mathsf{r}_{\mathsf{sim}}, w \leftarrow\!\!\$\, \mathbb{Z}_q$		
$A = g^{\mathsf{r}_{\mathsf{sim}}}(g^r)^{-w}$		
$B = (g^x)^{\mathsf{r}_{\mathsf{sim}}}(g^{rx})^{-w}$		
	$\xrightarrow{\mathsf{c}_{\mathsf{sim}}, \mathsf{r}_{\mathsf{sim}}, A, B}$	
		$g^{\mathsf{r}_{\mathsf{sim}}} \overset{?}{=} (g^r)^{c_{sim}} A$
		$(g^x)^{\mathsf{r}_{\mathsf{sim}}} \overset{?}{=} (g^{rx})^{c_{sim}} B$

Real proof of DH tuple for $\langle g, g^r, g^x, g^{rx} \rangle$

Prover		**Verifier**
$w \leftarrow\!\!\$\, \mathbb{Z}_q$		
$A = g^w$		
$B = (g^r)^w$		
$\mathsf{c}_{\mathsf{real}} = \mathsf{c}_{\mathsf{overall}} - \mathsf{c}_{\mathsf{sim}} \bmod q$		
$\mathsf{r}_{\mathsf{real}} = \mathsf{c}_{\mathsf{real}} r + w \bmod q$		
	$\xrightarrow{\mathsf{c}_{\mathsf{real}}, \mathsf{r}_{\mathsf{real}}, A, B}$	
		$g^{\mathsf{r}_{\mathsf{real}}} \overset{?}{=} (g^r)^{c_{real}} A$
		$(g^x)^{\mathsf{r}_{\mathsf{real}}} \overset{?}{=} (g^{rx})^{c_{real}} B$

Figure 1: Disjunctive non-interactive proofs of DH tuple.

attack from the voting client. We worked with the Helios designers to ensure that the server checks all relevant parameters are in $\mathbb{G}_q$ before checking the proof.

4.2 Rigging an Election & Proving You Didn't

In this attack a malicious election authority seeks to rig an election, i.e., alter the vote totals to an arbitrary result.Nominally Helios prevents this by requiring (a) all ballots included in the homomorphic tally have accepting proofs of correctness, and (b) the decryption of the homomorphic tally has an accepting proof of correctness. Universal verifiability arises from the fact that anyone in the public can run the proof verification and re-compute the homomorphic tally. Although we assume the election trustee has the ability to decrypt any individual ballot, they nominally do not have the ability to break the soundness of the proofs.

This attack is stronger than the conventional notion of ballot stuffing since in the case it is possible not only to add spurious votes, but *subtract* them as well!

Exploit. Initially we considered attacking the decryption proof, but in our single-trustee model, decryption occurs on the Helios server itself. As an alternative to maliciously modifying server code (which is outside our threat model), we considered the possibility of the election official submitting a maliciously constructed ballot instead. We believe this is a plausible scenario since often election officials are themselves voters in an election.

First the malicious trustee begins by creating a set of custom domain parameters $\langle p, q, g, y, x \rangle$ in which the expected properties still apply $|p| = 2048$, $q|p-1$, $g, y \in \mathbb{G}_q$ and $y = g^x$. The only exception is we select q, and hence $|\mathbb{G}_q|$ to be as small as possible while still being able to accommodate the ballots of all potential voters. The default choice of p could be used to create a homomorphic counters that could accommodate up to around 16-million votes since,

$$p - 1 = 2 \cdot 3^2 \cdot 5 \cdot 13 \cdot 23 \cdot 647 \cdot \big(\text{256-bit factor}\big) \cdot \ldots$$

This would be large enough to conduct an election in all but the largest cities on Earth, and even then we could reasonably expect homomorphic counters would be divided into smaller regions. The malicious election trustee constructs these purposefully weak parameters and uploads in a JSON file to the Helios server using built-in parameter upload page.

Similar to the poison ballot attack, the trustee will attempt to cherry-pick challenge values to achieve their goal of submitting an arbitrary ballot with a valid proof. As an added bonus, because we're working in a small group, the trustee can decrypt the intermediate homomorphic sum of the other ballots in order to know what to encrypt to achieve the desired election result. Suppose we have an election with two voters: a honest voter, and the malicious trustee. Suppose the honest voter casts a vote for Alice:

$$\langle \mathsf{Enc}_{\mathsf{Alice}} = \mathsf{Enc}(1), \mathsf{Enc}_{\mathsf{Bob}} = \mathsf{Enc}(0) \rangle.$$

But suppose the trustee wants Bob to win. If the trustee simply casts a vote for Bob, then the result will be a tie. Instead the trustee will cast *two* votes for Bob, and *minus one* vote for Alice:

$$\langle \mathsf{Enc}_{\mathsf{Alice}} = \mathsf{Enc}(-1), \mathsf{Enc}_{\mathsf{Bob}} = \mathsf{Enc}(2) \rangle$$

such that the homomorphic tally will have the desired outcome of a landslide victory for Bob:

$$\langle \mathsf{Enc}_{\mathsf{Alice}} = \mathsf{Enc}(1-1=0), \mathsf{Enc}_{\mathsf{Bob}} = \mathsf{Enc}(0+2=2)\rangle.$$

But the trustee is on the hook now to produce accepting proofs of correctness. Recall for each candidate, as well as the combined sum, this involves proving a ciphertext is either $\mathsf{Enc}(0)$ or $\mathsf{Enc}(1)$. If $\mathsf{Enc}(m)$ for $m \neq 0$ and $m \neq 1$ the proof will nominally fail. Recall from Figure 1 that the verifier confirms a DH tuple by checking

$$(g^x)^{\mathsf{r}} \stackrel{?}{=} (g^{rx})^c B.$$

Instead we have

$$(g^x)^{\mathsf{r}} \stackrel{?}{=} (g^{m+rx})^c B.$$

and the equality does not hold. If, however, the trustee could select a challenge $c \equiv 0 \bmod q$ then we have

$$\begin{aligned} resp &= cr + w \bmod q \\ &= w \bmod q \end{aligned}$$

and therefore

$$\begin{aligned} (g^x)^{resp} &\stackrel{?}{=} (g^{m+rx})^c B \\ (g^x)^w &\stackrel{?}{=} (g^{cm+crx})(g^x)^w \\ g^{xw} &= g^{xw}. \end{aligned}$$

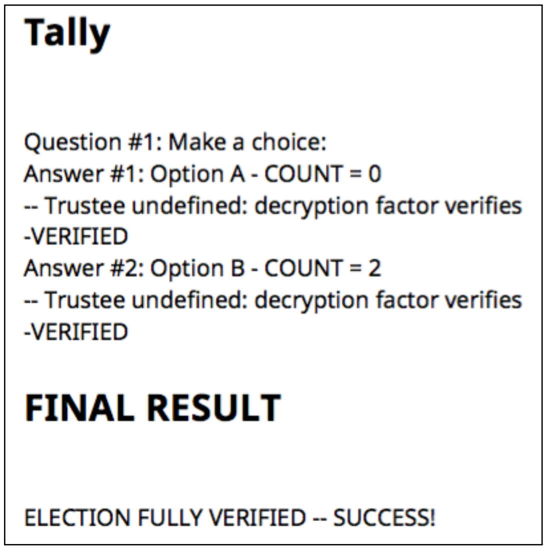
Thus verification success, independent of the value of the homomorphic counter $\mathsf{Enc}(m)$. The trustee must then ensure that all 6 real and simulated challenges are 0 mod q. Once again the simulated challenges can be directly chosen to have this property. The real challenges, being essentially a random value in $\mathbb{Z}_q$, will have this property with probability $P = \frac{1}{q}$. All 3 real challenges (Alice, Bob, Sum) will simultaneously have this property with probability $P = \frac{1}{q^3}$. Once again the trustee attempts to generate such challenges, rewinding if any real challenge does not meet the criteria and trying again until successful. Figure 2 shows a screenshot of our ballot stealing attack.

Impact and Mitigation. We implemented the attack and confirmed we could produce arbitrary election tallies with accepting proofs. The impact is severe since a malicious election official can not only (a) completely bypass the cryptographic protections of the cryptographic audit to produce whatever result they wish, but can also (b) produce an accepting proof that the tally was correct. It might rightly be pointed that this could be mitigated by independent parties writing their own implementations of the verification protocol. We are presently aware of only one such independent implementation for Helios verification, and did not find evidence that it would catch this attack.[6] Again we worked with the Helios designers to ensure that the domain parameters p, q implement a cyclic group $\mathbb{G}_q$ of large prime order, and that $g, y \in \mathbb{G}_q$.

5. ATTACKS ON BALLOT SECRECY

We demonstrate the random number generator (RNG) used in the Helios client-side voting booth exhibits a bias, allowing an attacker to distinguish between real and simulated votes with non-negligible advantage. This breaks the formal security notion of ciphertext indistinguishably in Helios, and appears to affect all past elections. Based on the particular group parameters chosen by the Helios designers, an attacker observing only the public cryptographic audit trail can correctly guess how a voter voted approximately 53% of the time in a two-candidate race. Depending on the group parameters used, we show the attacker can be successful up to 67% of the time. We show the attacker has negligible advantage when safe-prime groups are used. Interestingly, however, we discovered that if safe-prime groups *were* used in Helios, a separate implementation flaw in the RNG would reveal ballot selections with overwhelming probability.

[6]https://github.com/google/pyrios



Tally

Question #1: Make a choice:
Answer #1: Option A - COUNT = 0
-- Trustee undefined: decryption factor verifies
-VERIFIED
Answer #2: Option B - COUNT = 2
-- Trustee undefined: decryption factor verifies
-VERIFIED

FINAL RESULT

ELECTION FULLY VERIFIED -- SUCCESS!

Figure 2: Screen capture of a rigged Helios election. One ballot contained a vote for Option A. Another ballot contained *two* votes for Option B, and -1 votes for Option A.

5.1 Helios RNG Bias

Like other cryptographic implementations (such as TLS), random-number generators (RNGs) and pseudo-random number generators (PRNGs) are critical components not only for privacy, but also integrity and authenticity. Although RNGs and PRNGs are fundamentally different beasts, in practice RNGs are often implemented as a hybrid: entropy is collected, extracted, and run through a PRNG to boost the output length, and clean up any unanticipated deviations from the intended output distribution. For simplicity we simply refer to the hybrid case as an *RNG*. Helios uses random number generation for a variety of cryptographic purposes: random factors for Elgamal encryption, trustee private keys, commitment exponents, and as a challenge value in the proof of DH tuple.

Let R be an RNG that accepts a value q and returns a random value in the range $[0, q-1]$. We define the bias $0 \leq b < \frac{1}{2}$ of R as follows:

$$P\left[R(q) \leq \left\lfloor \frac{q}{2} \right\rfloor\right] = \frac{1}{2} + b.$$

Suppose $x \leftarrow R(q)$, and $y = g^x \bmod p$. A small bias in $R(q)$ (e.g., $b = 0.01$) may be acceptable in this setting; the effective search space is still on the order of 2^q, and the element y is still statistically uniform in $\mathbb{Z}_p^*$.

The raw output of R, however, shows up in the cryptographic audit trail of Helios. Of particular interest here are the *challenge* values of the disjunctive proofs. The simulated challenge $\mathsf{c}_{\mathsf{sim}}$ is directly sampled from R. By contrast the real challenge $\mathsf{c}_{\mathsf{real}}$ is computed as:

$$\mathsf{c}_{\mathsf{real}} = \mathsf{c}_{\mathsf{overall}} - \mathsf{c}_{\mathsf{sim}} \bmod q.$$

The Helios client-side RNG is written in Javascript (See Listing 1). It accepts a max value max and returns a random value in the range $[0, max)$. First it computes the bit-length the of max: $\ell = |max|$. It then calls a function to receive $\frac{\ell}{32}$ random 32-bit words. Specifically it calls `sjcl.random.randomWords()`, part of the Stanford Javascript Crypto Library (SJCL).[7] SJCL implements a variant of the Fortuna pseudo-random number generator [22]. Briefly, it collects entropy from a number of sources in the client (*e.g.,* mouse position), hashes them, and uses the result as the key in `AES-CTR`.

`sjcl.random.randomWords()` returns a value in the range $[0, 2^{|max|})$. To coerce this to a value in the range $[0, max)$, Helios returns the value modulo max, thus introducing a modulo bias.

```
Random.getRandomInteger = function(max) {
    var bit_length = max.bitLength();
    Random.setupGenerator();
    var random =
        sjcl.random.randomWords(bit_length / 32, 0);
    var rand_bi =
        new BigInt(sjcl.codec.hex.fromBits(random), 16);
    return rand_bi.mod(max);
};
```

Listing 1: Helios Voting Booth PRNG

5.2 Modulo Bias

We now analyze the bias of the Helios RNG. The RNG is idealized as follows:

```
$R_{mb}(q)$
$\ell = \lceil \log_2(q) \rceil$
$a \leftarrow\!\$\ [0, 2^\ell)$
return $a \bmod q$
```

Define event e to be the outcome when $R_{mb}(q)$ returns a value that is less than $\frac{q}{2}$. This outcome arises from two possible events. Define event e_1 to be the event when $a \in [0, \frac{q}{2})$. R_{mb} will return a directly. Let let us define another event, e_2 to be the case where $a \in [q, 2^\ell)$. Here R_{mb} will apply the modulo reduction, producing a result that is strictly less than $\frac{q}{2}$ (for any ℓ-bit q). As e_1 and e_2 are mutually exclusive, we have $P(e) = e_1 + e_2$. We expect to see outputs less than $\frac{q}{2}$ *more* than half the time. Let us now compute the probability of receiving a value from R_{mb} that is less than $\frac{q}{2}$.

THEOREM 1. *The probability that $R_{mb}(q)$ produces a value less than $\frac{q}{2}$ is:*

$$P\left[R_{mb}(q) < \frac{q}{2}\right] = \begin{cases} \frac{q}{2^\ell} & q \leq \frac{2}{3} \cdot 2^\ell \\ 1 - \frac{q}{2^{\ell+1}} & \textit{otherwise} \end{cases} \qquad (3)$$

[7]https://crypto.stanford.edu/sjcl/

PROOF. Assuming that a is sampled uniformly,

$$P(e_1) = \frac{q}{2} \cdot \frac{1}{2^q} = \frac{q}{2^{\ell+1}},$$

$$P(e_2) = (2^\ell - q) \cdot \frac{1}{2^\ell} = 1 - \frac{q}{2^\ell}.$$

There are now three cases to consider.
Case 1. $P(e_1) = P(e_2)$. Then we have

$$\begin{aligned} P(e) &= 2 \cdot \frac{q}{2^{\ell+1}} \\ &= \frac{q}{2^\ell}. \end{aligned}$$

Case 2. $P(e_1) > P(e_2)$. Then we have

$$\begin{aligned} P(e) &= P(e_1) + P(e_2) \\ &= \frac{q}{2^{\ell+1}} + 1 - \frac{q}{2^\ell} \\ &= 1 - \frac{q}{2^{\ell+1}}. \end{aligned}$$

Case 3. $P(e_1) < P(e_2)$. This implies $P(e_2) > \frac{q}{2^{\ell+1}}$ which implies $P(e) > \frac{q}{2^\ell}$, which implies $q > 2^\ell$, which is false by definition. □

Helios uses a custom generated algebraic group. The prime modulus p is 2048 bits, with a generator of a 256-bit group order q where

$$\log_2(q) = 255.0831..$$

which is fortuitously is fairly far off the optimal biased value of $|q| = 255.4150$. Applying q to Equation (3), however, we find:

$$P\left[R_{mb}(q) < \frac{q}{2}\right] \approx 0.53.$$

In other words, the Helios RNG will return a value less than $\frac{q}{2}$ approximately 53% of the time.

5.3 Distinguishing Ballots in Helios

Here we show the modulo bias can be used to guess how a voter voted from the cryptographic audit trail with advantage (i.e., better than random). For space we examine the 2-candidate race scenario. First recall that an encrypted ballot with 3 disjunctive proofs, i.e., each candidate was voted for 0 or 1 times, and the total number of votes was 0 or 1. This consists of $2 \cdot 3 = 6$ Chaum-Pedersen proofs of DH tuple. Let the challenge values for each of the proofs respectively be: $\mathsf{c}_{\mathsf{Alice}=0}, \mathsf{c}_{\mathsf{Alice}=1}, \mathsf{c}_{\mathsf{Bob}=0}, \mathsf{c}_{\mathsf{Bob}=1}, \mathsf{c}_{\mathsf{Sum}=0}, \mathsf{c}_{\mathsf{Sum}=1}$. We now give a strategy for distinguishing ballots:

Ballot Distinguisher Strategy

```
in Challenges: $c_{Alice=0}, c_{Alice=1}, c_{Bob=0}, c_{Bob=1}$
if $\Big((c_{Alice=0} < c_{Alice=1})$ AND $(c_{Bob=0} > c_{Bob=1})\Big)$
    return Guess vote for Alice
elseif $\Big((c_{Alice=0} > c_{Alice=1})$ AND $(c_{Bob=0} < c_{Bob=1})\Big)$
    return Guess vote for Bob
else
    return Random guess
```

THEOREM 2. *($\mathcal{A}$'s advantage) Using the strategy given above, for an RNG with bias b, an adversary can correctly guess the encrypted ballot with probability $\frac{1}{2} + b$.*

PROOF. First notice the simulated challenges are generated by the RNG, $P(c_{sim} < \frac{q}{2}) = \frac{1}{2} + b$, and recall $c_{sim} + c_{real} = c_{overall} \mod q$. Helios uses `SHA1` to produce $c_{overall}$, which has a 160-bit output. Given the size of q relative to this length (256-bits), we have $c_{sim} + c_{real} = s \mod q$ for a small s. This means that when $c_{sim} < \frac{q}{2}$, the real challenge will be $c_{real} > \frac{q}{2}$, and vice versa.

In the presence of a bias we expect $c_{sim} < c_{real}$. So if we examine the challenges in the Alice proof and we find $c_{Alice=0} < c_{Alice=1}$, we would expect $c_{Alice=0}$ is simulated and $c_{Alice=1}$, thereby indicating a vote for Alice. If our guess is correct, then in the Bob proof we would expect the opposite, $c_{Bob=0} > c_{Bob=1}$. Let us define two outcomes: a *consistent* outcome is when the challenges are consistent with our expectation that the vote implied by the challenges in the Alice proof will similarly imply the opposite outcome in the Bob proof. We call an outcome *inconsistent* when the challenges do not meet this expectation. Let p be the probability that the simulated challenge is less than the real challenge. There are now three cases:

1. **Case 1. Consistent challenges, and expectation is correct**: If the result is consistent, then the probability that both simulated challenges are less than their corresponding real challenges is p^2.
2. **Case 2. Consistent challenges, and expectation is incorrect**: If the result is consistent, then the probability that both simulated challenges are greater than their corresponding real challenges is $(1-p)^2$
3. **Case 3. Inconsistent challenges**: If the result is inconsistent, then one simulated challenge was less than its real challenge, and one was greater. The probability of this is $2p(p-1)$.

The overall probability of being correct is

$$\begin{aligned} P(\mathcal{A}\text{ guesses correctly}) &= p^2 + p(1-p) \\ &= p. \end{aligned}$$

Using this strategy in Helios where $p = 0.53$, an adversary can correctly guess how a voter voted with probability $P = 0.53$. If q had been chosen to be optimally 'bad', i.e., $\frac{2}{3}2^{257}$, this probability would grow as high as $P = 0.67$. Finally, if q were chosen optimally close to a power of 2, *e.g.,* if prime modulus $|p| = 2048$ bits and $q = \frac{p-1}{2}$, the attackers advantage would be negligible. □

5.4 Attacks on Custom DL Parameters

Helios uses its own custom DL parameters. Recently attacks like Logjam [5] have suggested that election officials may wish to use their own parameters. One common choice is to use a safe-prime group, i.e., one for which $p = 2q + 1$. Safe prime groups, for example, account for the majority of DHE parameters used in TLS. Another more severe bias could manifest if the default group parameters were changed. If election officials were to use a safe prime group, we show that ballot secrecy would be compromised in the public audit.

Recalling that the RNG computes `bit_length` of q. When the call gets made to the SJCL's `randomWords` function, it passes in an integer representing how many 32-bit words should be returned:

```
random = sjcl.random.randomWords(bit_length/32,0);
```

A problem arises when the bit-length of q is not a multiple of 32. The number of words that get returned is:

$$\#\text{words to generate} = \left\lfloor \frac{\texttt{bit\_length}}{32} \right\rfloor$$

but the intended result should be

$$\#\text{words to generate} = \left\lceil \frac{\texttt{bit\_length}}{32} \right\rceil.$$

What this means is if the bit length of q is not a multiple of 32 such as in a safe prime group where $|q| = |p| - 1$, then we have

$$P\left[R_{mb}(q) < \frac{q}{2}\right] = 1, \tag{4}$$

thus,

$$P[c_{sim} < c_{real}] = 1 \tag{5}$$

and finally,

$$P(\text{Attacker guesses correctly}) = 1. \tag{6}$$

In the case of a safe prime group with a 2048-bit prime modulus p, the bit length of q is 2047, Instead of the RNG returning 64 words (as expected), it would return 63 words. Because the real and simulated challenges sum to a small value modulo q, the simulated challenge will be appear significantly smaller than the real challenge (see Figure 3) allowing real and simulated votes to be distinguished with certainty.

```
c_real = 2457959957311721763991064225088283650242666
         5024266050571479050957694347055489460681O2

c_sim  = 22134278396622071142450162853498951503873 6
         959263058396679729826957 7
```

Figure 3: Illustration of length difference between a real challenge (top) and simulated challenge (bottom) when using Helios with a safe prime group.

Impact and Mitigation. To the best of our knowledge all past Helios elections (totally approximately 500,000 cast ballots) used the biased client-side RNG with the default parameters. In practical terms the threat to voters in past elections is low. The significance of the vulnerability is twofold: (1) it breaks the formal security guarantees of Helios as proven in [8]. Although the Helios protocol is non-malleable under chosen-plaintext attack, we have shown that in the Helios implementation, ballots are distinguishable under eavesdropping, thus breaking the most basic formal notion of privacy. (2) The other more serious risk is to future election officials who use their own custom parameters and either experience a much larger bias, potentially and inadvertently revealing the voting preferences of all voters on a public website.

The textbook fix is to not use a modulo reduction to bring large values inside the desired range, but rather loop the RNG until it produces a value less than q. But this approach consumes more entropy than it strictly needs, so instead we used the "simple modular method" for converting random bits to a random integer in the range $[0..max - 1]$ as described by NIST[6]. In this approach, given a max, and a security parameter s, the random bit generator generates $|max| + s$ (or more) bits. The resulting bits are then returned modulo max. Because the bit value is likely much larger than max, the modulo bias is similarly negligible. We corrected the RNG code using the NIST recommended value of $s = 64$.

6. WEB ATTACKS

6.1 Cross-site Scripting (XSS) Attacks

Cross-site scripting attacks (XSS) are a type of code injection attack where malicious code is injected into a website and then executed on a victim's machine. The malicious code often takes the form of a JavaScript element embedded in the Document Object Model (DOM). They can be difficult to completely eliminate, and may be leveraged to perform a wide range of actions on the user's behalf.

Prevention of XSS attacks generally involve sanitizing any user input by encoding scripting characters to be displayed as plain text. There is not, however, a universal solution to the problem. How user data is escaped largely depends on the context in which it will eventually be used which varies between applications. For example, user input that will be displayed as a paragraph element has different escape requirements than user input that will be assigned to a JavaScript variable. We encountered a small oversight in the Helios code that would allow an XSS attack to be performed on a voter's device. In this section we describe an attack that would allow a remote attacker to cast a ballot on a voter's behalf, and further display the ballot had been cast as intended. Note this differs from our threat model allowing client-side malware, exploiting instead the trust the voter's browser places in Helios.

6.2 XSS Vulnerability

As previously mentioned, how input will eventually be displayed affects how it should be escaped. The vulnerability in Helios is caused by the use of HTML escaping on data that is used in a JavaScript context. This happens on the "Questions" page of an election where any user, registered or not, can view the questions and candidates of an election. These questions and candidates are specified by the administrator(s) of the election, and elections can be created by anyone with a Google or Facebook account.

Creating an Election. The steps for creating a Helios election are relatively simple:

1. Log in with Facebook or Google account,
2. Provide an election name and description,
3. Add ballot questions and answers,
4. Provide a list of approved voters, or allow any registered user to vote,
5. Freeze the ballot (i.e., prevent future changes) thereby opening voting phase.

After questions are added, they are serialized into JSON and stored in the Helios database. The only apparent character that is escaped is the double quote marks ($\ddot{()}$). When it comes time to serve this content to users viewing the questions page, this JSON object is retrieved from the database and parsed by Django to build the requested pages. In `election_questions.html` the JSON object is assigned to the `QUESTIONS` variable as

```
QUESTIONS = {{questions_json|safe}} .
```

The double-brace notation informs Django that a variable is contained within and will eventually replace the variable name with its value. The pipe operator informs Django that the variable is to be passed through a filter. Filters are functions that modify variables before displaying them. In the previously mentioned code, the variable, `questions_json`, is passed through the safe filter. The safe filter informs Django that the variable passed into it requires no further HTML escaping and that it is safe for display.

The consequence is whatever an election administrator provides as a question is not escaped when displayed to other users. This is likely because HTML escaping a JSON structure results in a structure that cannot be used in JavaScript, but this also creates a perfect opportunity to perform an XSS attack.

Exploit. We were able to cast ballots in an election on behalf of registered voters by taking advantage of Helios's recast feature where Helios allows for the re-casting of ballots and only counts the most recently cast ballot.

To perform the attack, assume we have three parties: Alice, an honest election administrator; Bob, an honest voter; and Charlie, a remote attacker. Alice creates an election with a single question and two candidates, "Kang" and "Kodos" with open registration (i.e., any Helios user can vote) and promotes the election through the relevant channels. Alice and Bob cast their ballots through the normal process and coincidentally both vote for Kang, while Charlie casts his ballot for Kodos and wants to ensure that Bob does the same.

To do this, Charlie creates two elections: a malicious election with a ballot question containing a `<script>` tag linking to an externally hosted attack script, and another decoy election for "Favorite Soda". The malicious question exploits the XSS vulnerability can then take the form:

```
<script src=example.com/vote-stealer.js></script>
```

Charlie then sends Bob and others an email containing a link to the malicious election asking them to view his "Favorite Soda" election. While this would execute the script, the user wouldn't necessarily be logged for the cast ballot to be accepted. To get around this, Charlie sends a link to the Google OAuth endpoint with the return parameter set to the malicious election. The victim doesn't necessarily detect anything unusual since the URL still points to `heliosvoting.org`. An example link would be of the form:

```
https://vote.heliosvoting.org/auth/?return_url=
/helios/elections/c0e6fec8 ... d9/questions
```

Users clicking on the link will first be presented with the Google login screen for Helios. After entering their credentials, the user will be redirected to the page containing the XSS payload.

We constructed an XSS payload and hosted it on a remote server. It begins by extracting a user-specific CSRF token from the page. Because the CSRF token is the same across all elections for the authenticated user, it is sufficient to take the CSRF token from the malicious election page and use it in other elections. In the attack, the ballot variable is hardcoded but it could be arranged such that code from the helios-booth is used to generate ballots at runtime, although, in the interest of time, it would make more sense for the attacker to pre-generate ballots and include a new one each time the script is called. The script then builds a POST request containing the ballot and CSRF token and sends it to the Helios endpoint for Alice's election. At this point, the script has successfully cast a ballot in Alice's election from Bob's browser via the malicious election page. Upon successful completion of the POST request, the success callback is executed and the user is redirected to the "Favorite Soda" election.

From the user's point of view, they started by having to log in to Helios and were then redirected to the "Favorite Soda" election. The only evidence that a malicious action has taken place is the email confirmation for casting a ballot in Helios. Although this may alert the voter of unusual activity, Mohr *et al.* [27] recently suggested that even if voters suspect something went wrong, they may more likely attribute it to a fault or misunderstanding in their own action. From Helios's point of view, the ballot was legitimately cast and since the most recent ballot is the one tallied, Bob's initial vote is overwritten by the ballot specified by Charlie.

Impact and Mitigation. We implemented the attack and were able to steal ballots from voters who clicked our malicious heliosvoting.org URL. The impact of this exploit is high, since anyone (not just eligible voters) can create a dummy election with the vote stealing XSS. We informed the Helios developers and they released a fix of the XSS vulnerability.

Related Attacks. A similar XSS vulnerability in the Helios "Questions" page was discovered in a 2011 paper by Heiderich *et al.* [24] caused by a lack of context-sensitive filtering. Heiderich is also credited on the Helios page with the disclosure[8] of a different XSS vulnerability from the paper resulting in a fix. Our work extends theirs by successfully demonstrating a complete exploit of this vulnerability and identifying the responsible code finally resulting in its fix.

7. DISCUSSION

Criticism of the E2E verification paradigm has often focused on practical issues such as poor usability design [2, 1], cognitive dissonance associated with encountering a verification error [27], or a high pedagogical bar to understanding the cryptography.[9] We believe the results of this paper point to another avenue for consideration: The risk introduced by the cryptographic audit trail itself. The benefit of E2E over conventional voting systems is that it ultimately focuses on verifying elections, not the software or voting machines themselves. In some sense, however, this pushes back the problem: who verifies the verifiers?

1. **Privacy Risk**. The public cryptographic audit trail creates a tension between the ability to detect fraud and preservation of voter privacy: if you take an attacked transcript down early, it limits exposure of future privacy vulnerability discoveries (e.g., the Helios RNG), and the expense of the public record, not to mention election integrity (e.g., weak group parameters).

2. **Integrity Risks**. Ultimately E2E elections are making an assertion about the election results and ostensibly supporting it with evidence. But we also must consider the potential threat of a rigged election with a valid-looking proof. Such a situation might make matters doubly bad: not only must the election results be disavowed, but the proof must as well.

The U.S Vote Foundation report recommends that "any public elections conducted over the Internet must be end-to-end verifiable" [20]. For the most part we would agree. But as we feel this case study has demonstrated, it may not be appropriate for all situations. At very least a conversation needs to happen regarding the potential risks to privacy and integrity that an E2E scheme potentially *introduces* relative to a conventional scheme, and it be weighed against the benefits. Surely E2E is worse than a conventional "less-verifiable" election if it is used as a tool to convince people of the veracity of a malicious election result using an unsound proof.

Conclusion

In this paper we conducted a security analysis of Helios, an E2E-verifiable internet voting system. We discovered a range of serious vulnerabilities attacking confidentiality, integrity and availability. We presented the technical details of the vulnerabilities and worked with the Helios designers to fix them. Unlike conventional elections, the public nature of the cryptographic audit introduces new risks to ballot secrecy and election integrity that have still not been fully explored, but hopefully will become a point for further debate over the future role of E2E-verification.

Acknowledgments

Special thanks to Ben Adida and the Helios project. Additional thanks to Jeremy Clark, Olivier Pereira, Poorvi Vora, and Filip Zagórski. This work was funded in part by the National Science and Engineering Research Council of Canada (NSERC) Discovery Grants Program.

8. REFERENCES

[1] C. Z. Acemyan, P. Kortum, M. D. Byrne, and D. S. Wallach. From Error to Error: Why Voters Could not

[8] http://documentation.heliosvoting.org/attacks-and-defenses

[9] http://csrc.nist.gov/groups/ST/e2evoting/

Cast a Ballot and Verify Their Vote With Helios, Prêt à Voter, and Scantegrity II. In *USENIX Journal of Election Technology and Systems*, 2015.

[2] C. Z. Acemyan, P. Kortum, M. D. Byrne, and D. S. Wallach. Usability of voter verifiable, end-to-end voting systems: Baseline data for helios, prêt à voter, and scantegrity ii. In *USENIX EVT/WOTE*, 2015.

[3] B. Adida. Helios: web-based open-audit voting. In *USENIX Security Symposium*, pages 335–348, 2008.

[4] B. Adida, O. d. Marneffe, O. Pereira, and J.-J. Quisquater. Electing a university president using open-audit voting: Analysis of real-world use of Helios. In *EVT/WOTE*, 2009.

[5] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, et al. Imperfect forward secrecy: How diffie-hellman fails in practice. In *CCS*, 2015.

[6] E. Barker and J. Kelsey. Recommendation for random number generation using deterministic random bit generators. special publication 800-90a. In *National Institute of Standards and Technology*, 2012.

[7] S. Bell, J. Benaloh, M. D. Byrne, D. Debeauvoir, B. Eakin, P. Kortum, N. McBurnett, O. Pereira, P. B. Stark, D. S. Wallach, G. Fisher, J. Montoya, M. Parker, and M. Winn. Star-vote: A secure, transparent, auditable, and reliable voting system. In *Electronic Voting Technology Workshop/Workshop on Trustworthy Elections (EVT/WOTE 13)*, 2013.

[8] D. Bernhard, O. Pereira, and B. Warinschi. How not to prove yourself: Pitfalls of the fiat-shamir heuristic and applications to helios. In *ASIACRYPT*, volume 7658 of *LNCS*, pages 626–643. 2012.

[9] P. Bulens, D. Giry, and O. Pereira. Running mixnet-based elections with helios. In *USENIX EVT/WOTE*, 2011.

[10] C. Burton, C. Culnane, J. Heather, T. Peacock, P. Ryan, S. Schneider, S. Srinivasan, V. Teague, R. Wen, and Z. Xia. Using Pret a Voter in Victoria State elections. In *EVT/WOTE*, 2012.

[11] C. Burton, C. Culnane, and S. Schneider. Secure and verifiable electronic voting in practice: the use of vvote in the victorian state election. 2015.

[12] 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 election at Takoma Park. In *USENIX Security Symposium*, 2010.

[13] D. Chaum, R. Carback, J. Clark, A. Essex, S. Popoveniuc, R. L. Rivest, P. Y. A. Ryan, E. Shen, and A. T. Sherman. Scantegrity II: end-to-end verifiability for optical scan election systems using invisible ink confirmation codes. In *EVT*, 2008.

[14] D. Chaum and T. P. Pedersen. Wallet databases with observers. In *CRYPTO*, 1992.

[15] D. Chaum, P. Y. A. Ryan, and S. Schneider. A practical voter-verifiable election scheme. In *ESORICS*, 2005.

[16] V. Cortier and B. Smyth. Attacking and fixing helios: An analysis of ballot secrecy. In *Computer Security Foundations Symposium (CSF), 2011 IEEE 24th*, pages 297–311, 2011.

[17] R. Cramer, I. Damgård, and B. Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *CRYPTO*, 1994.

[18] A. Delis, K. Gavatha, A. Kiayias, C. Koutalakis, E. Nikolakopoulos, L. Paschos, et al. Pressing the button for european elections: verifiable e-voting and public attitudes toward internet voting in greece. In *Electronic Voting: Verifying the Vote (EVOTE), 2014 6th International Conference on*, 2014.

[19] D. Demirel, J. van de Graaf, and R. S. dos Santos Araújo. Improving helios with everlasting privacy towards the public. In *USENIX EVT/WOTE*, 2012.

[20] S. Dzieduszycka-Suinat, J. Murray, J. Kiniry, D. Zimmerman, D. Wagner, P. Robinson, A. Foltzer, and S. Morina. The Future of Voting: End-to-End Verifiable Internet Voting - Specification and Feasibility Study. https://www.usvotefoundation.org/E2E-VIV, 2015.

[21] S. Estehghari and Y. Desmedt. Exploiting the client vulnerabilities in internet e-voting systems: Hacking helios 2.0 as an example. In *USENIX EVT/WOTE*, 2010.

[22] N. Ferguson and B. Schneier. *Practical Cryptography.* John Wiley & Sons, 2003.

[23] A. Fiat and A. Shamir. How to prove yourself: practical solutions to identification and signature problems. In *CRYPTO*, pages 186–194, 1986.

[24] 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 *VoteID*, 2011.

[25] F. Karayumak, M. M. Olembo, M. Kauer, and M. Volkamer. Usability analysis of helios - an open source verifiable remote electronic voting system. In *USENIX EVT/WOTE*, 2011.

[26] R. Küsters, 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, 2012.

[27] E. Moher, J. Clark, and A. Essex. Diffusion of voter responsibility: Potential failings in e2e voter receipt checking. In *USENIX Journal of Election Systems and Technology*, 2015.

[28] National Institute of Standards and Technology (NIST). *NIST Special Publication 800-57, Part 1, Revision 4. Recommendation for Key Management. Part 1: General.* 2016.

[29] 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 21st ACM Conference on Computer and Communications Security.* ACM, Nov. 2014.

[30] V. Teague and J. A. Halderman. The new south wales ivote system: Security failures and verification flaws in a live online election. In *VoteID*, 2015.

[31] G. Tsoukalas, K. Papadimitriou, P. Louridas, and P. Tsanakas. From helios to zeus. In *USENIX EVT/WOTE*, 2013.

[32] S. Wolchok, E. Wustrow, J. A. Halderman, H. K. Prasad, A. Kankipati, S. K. Sakhamuri, V. Yagati,

and R. Gonggrijp. Security analysis of india's electronic voting machines. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, 2010.

[33] S. Wolchok, E. Wustrow, D. Isabel, and J. A. Halderman. *Financial Cryptography*, chapter Attacking the Washington, D.C. Internet Voting System, pages 114–128. 2012.

[34] 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.* 2013.