Universitat Oberta de Catalunya

Final Masters Thesis

Masters Degree in Cybersecurity

Coercion-free electronic elections using Tor

Author
Pau Argelaguet

Supervisor
Silvia Puglisi

December 28, 2021

Contents

List of Figures

Listings

Part I

Introduction and planning

Chapter 1

Problem statement

Electronic voting is a hot topic of discussion, not only in academia but also in public opinion. The main question is whether this kind of system can have widespread use in public elections in substitution of classical processes with physical polling stations, ballots, ballot boxes, etcetera. Multiple electronic elections have been conducted in different contexts, ranging from small private organizations to large nationwide, official-electing polls [15][4][10]. However, trust in those elections is not always as high as it would be on an election conducted with physical ballots.

Trust is the main obstacle in electronic voting — people who participate in those elections not only need to get some outcome on the electoral process but they need convincing evidence that the provided results are correct and have not been tampered with. There are many points of failure in which an election can be tampered with: whenever the vote is cast, recorded, or tallied. Electronic elections that provide protections against those points of failure are called **end-to-end verifiable elections** [2]. Also importantly, a fair election must guarantee that voters are not coerced when they exercise their right to vote.

Classical paper-based elections have historically built that trust and they have a variety of failsafe mechanisms to prevent tampering at all those stages. At the very least, they have mechanisms to render attacks on a large scale impractical — a malicious actor can target a few polling stations, but that will not significantly affect the overall results of the election. That is not the case for electronic voting.

The motivation for altering elections is very high. It is easy to see how election results can highly influence politics, society, and economics, to name a few. Electronic voting, and especially voting systems that rely on the open Internet, make large-scale attacks that were not feasible on other systems feasible. Even though electronic voting provides several advantages over classical elections (faster tallying, voting is accessible for more communities, operation costs are cheaper, ...), the fact that they are more vulnerable to attacks turns them into a prime target for numerous groups of attackers.

1.1 The problem with coercion

End-to-end verifiable elections provide protections through the use of a myriad of cryptographic techniques to the casting, recording, and tallying steps of an election. That is, any voter can prove independently that their vote has not been tampered and has been correctly counted against the final result. As hinted in the previous section, coercion is a problem that is not directly solvable by the methods above.

Coercion in an election is defined as the intervention of an adversary, who might or might not be involved in the process, that influences a subset of voters with the purpose of altering their vote. When an attacker is performing coercion in an election, it is usually not compromising the voting system itself but forcing a human being to cast (or not cast) a vote in a specific way. Thus, it is possible that an end-to-end verifiable election is cryptographically correct but the results are not the ones the census would have intended to be.

A basic ingredient to coercion is for the attacker to know if someone is voting. That can be done by a simple packet inspection, which does not even need to go deep on the packet contents, but only needs to inspect the metadata that goes through the wire. It has been proven that by doing packet analysis, an attacker can identify an ongoing election[12].

1.1.1 Coercion types

There are many ways of performing coercion [7]. Those include the following.

- **Physical coercion**. The attacker has physical access to the user's device or can physically interact with the voter. Possibly through the use of force, it can prevent the voter from voting. It is very difficult to prevent but it also scales hardly.
- **Randomization attack**. The attacker coerces the voter by requiring them to submit randomly composed balloting material. In this attack, the attacker (and perhaps even the voter) is unable to learn what candidate the voter casts a ballot for. The effect of the attack is to nullify the choice of the voter with a large probability.
- **Simulation attack**. The attacker forces the voter to divulge their private keying material after the registration process but before the election process. These permit an attacker to divulge private keys or to buy private keys from voters and then simulate these voters at will, i.e., voting on their behalf.
- **Forced-abstention attack**. The attacker coerces a voter by demanding that they refrain from voting. If an attacker can see who has voted, they can use this information to threaten and effectively bar voters from participation.
- **Social engineering**. Instead of breaching a device through a vulnerability or watching its behavior in the network, social engineering consists of tricking a user to do something that goes against their will. In an election context, this might involve credential-stealing or voting on a fake website.

1.1.2 Forced-abstention attack

This thesis will focus on the forced-abstention attack. There are two straight forward ways of an attacker to know if a person has voted.

- Given that the census is public *and* the list of people who has voted is public too, its trivial to coerce a person about that. Most end-to-end verifiable voting systems solve this by publishing a list of pseudonyms instead of actual names on the list of voters.

- If an attacker has some degree of control over the network in which the voter it performing the vote and/or where the voting system is hosted, it can infer through traffic sniffing if somebody has voted.

  Most basic attacks can be prevented with basic on-transit encryption. Systems are still vulnerable to more advanced pattern-matching attacks, which will not actually tell the attacker *what* the person has voted, but the signal that they have voted at all might be a sufficient reason for coercion.

1.2 Preventing coercion through a desktop application

In modern web browsers, it is very common that HTTPS encryption is used in all connections to encrypt in-transit data. For that reason, man-in-the-middle attacks on (for instance) local networks are not a big issue, since they should not be able to observe the contents or modify the packets unless there is a compromised certificate.

To coerce a voter in an election, an attacker must observe the traffic and the packets going from the voter's device[1] to the server(s) that is ingesting the votes for the election. There are three kinds of packets that a traditional Internet-based voting system will use and that an attacker might pattern-match against.

- **Front-end assets**. This is the page that the user loads in their browser containing the front-end application[2] that lets them vote. That typically includes an HTML page plus some JavaScript, CSS, and image assets that are the same for all voters and elections.

- **Election data and metadata**. This is the data that populates the election itself. It ranges from the actual questions and answers options to some metadata including title, description, and cryptographic keys.

- **Vote casting**. The packages that the client sends to the server contain the ballots. They are usually already encrypted on the client.

[1] For simplicity, in this thesis it will be assumed that it is a desktop computer, but similar reasoning could be applied to other devices, like tablets or smartphones.

[2] Sometimes also referred as *voting booth*.

It is clear to see that the more packages circulate over the network (and the least variance they have), the easier it is for an attacker to find a pattern and then conclude that somebody is voting.

When using a desktop application, some of the above issues can be mitigated. No assets need to be downloaded, since they are already contained in the binary. An attacker might indeed detect or even block the download of the binary that contains the booth, but there are multiple ways from the voter side to circumvent that.

This thesis will focus on two other issues: metadata and vote casting. A desktop application will be developed in such a way that traffic cannot be detected hence an attacker cannot prove that a person has participated in an election.

1.3 Goals

The goal of this thesis is to propose a solution to prevent an attacker who has some degree of control over a network to impede voters which are using that network. That is a system that is resistant to forced-abstention coercion attacks.

Although the goal for the thesis is forced-abstention attacks, indeed, such a system will also help prevent (or at least, make them more difficult) the other described attacks. The fact that an attacker cannot know when a voter did vote or even if they voted at all adds a layer of complexity to the hypothetical attack.

Specifically, the goals of the thesis are the following.

- Provide a functional desktop application that can cast electronic votes.
- That application must provide a way to cast a vote that protects the voter of forced-abstention coercion from an adversary.
- Integrate the desktop client with a voting system that can perform an end-to-end verifiable election.

The following are *non-goals* for the thesis:

- Develop a novel voting system.
- Tackle end-to-end verifiable elections.
- Simulate a large scale network attack to pattern match an election.

Chapter 2

State of the art

2.1 Tor

Tor [16][3][5] is an overlay network (on top of the open Internet) that uses the concept of onion routing to provide privacy, anonymity, and communication security. Tor works by tunneling a user's traffic through several intermediate nodes on the network instead of directly connecting to the other end (usually, a web service). Each step adds its layer of encryption, thus the name *onion routing*.

With this system, a user's traffic is anonymous and harder to track by any network observers and by the receiving end of the transmission. This provides the user with a higher degree of privacy. Figures 2.1, 2.2 and 2.3, courtesy of the Electronic Frontier Foundation, illustrate how Tor works.

2.1.1 Bridges and pluggable transports

Although Tor provides a layer of privacy, the fact that somebody is using Tor might be suspicious by itself in some contexts. Attackers might know and block IP addresses of Tor nodes or even perform *Deep Packet Inspection* to identify and potentially block a Tor connection.

Tor addresses the first issue by using **bridges** [8]. Bridges are secret Tor nodes which IPs are unknown to an attacker and that change over time. Therefore, they cannot be blocked.

To prevent traffic analysis, pluggable transports disguise Tor traffic by making it look like something else (like a WebRTC connection from a videochat application).

Figure 2.4, courtesy of Ramzi A. Haraty, shows how Bridges work.

2.1.2 Snowflake

Snowflake is another tool to disguise Tor traffic in highly restrictive networks, such as Internet users in the People's Republic of China. Snowflake works by routing the traffic through volunteer-run computers using a WebRTC connection.

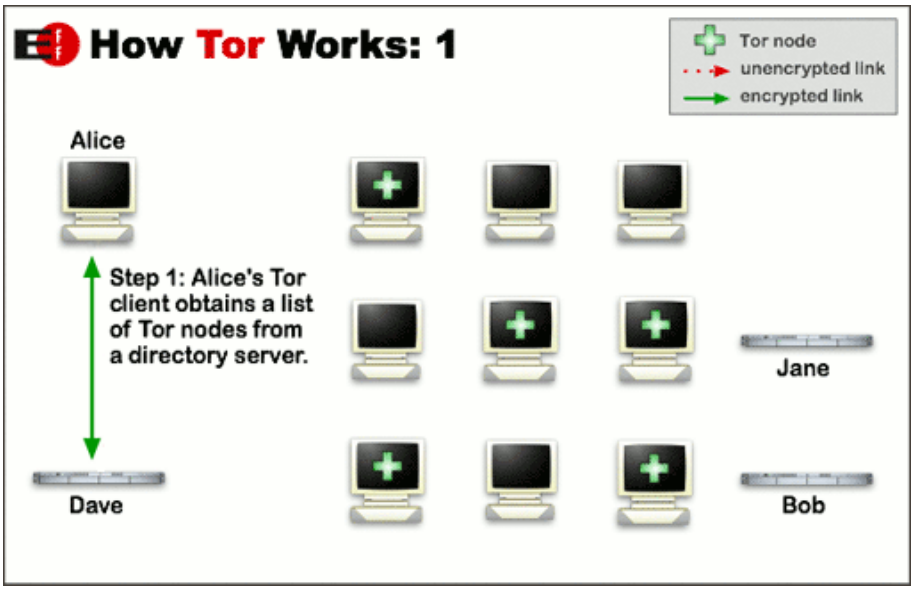

Figure 2.1: How Tor works (1)

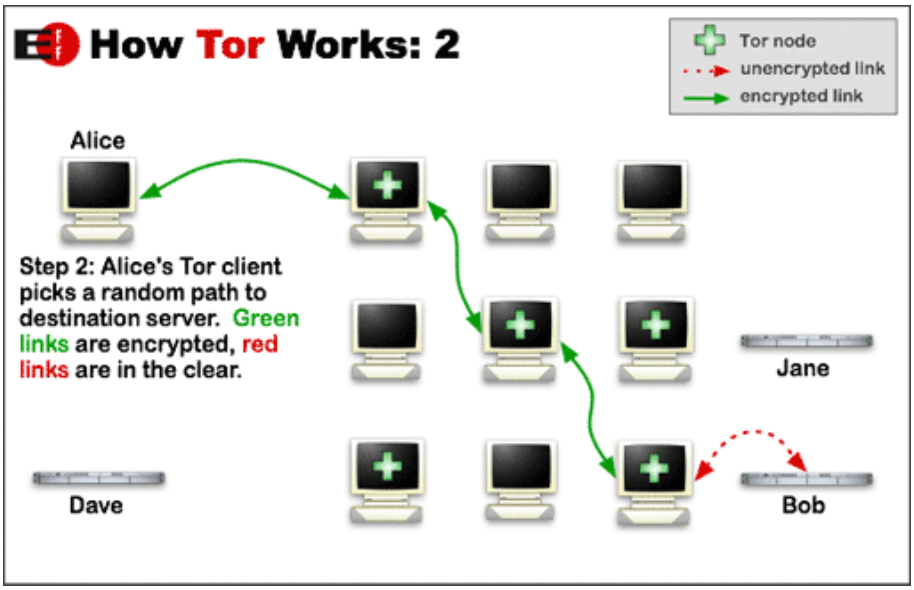

Figure 2.2: How Tor works (2)

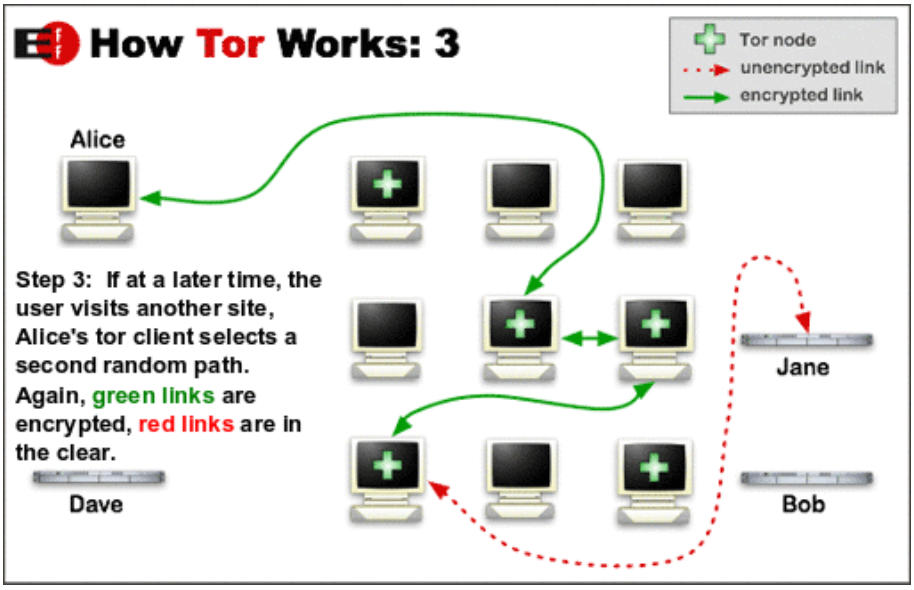

Figure 2.3: How Tor works (3)

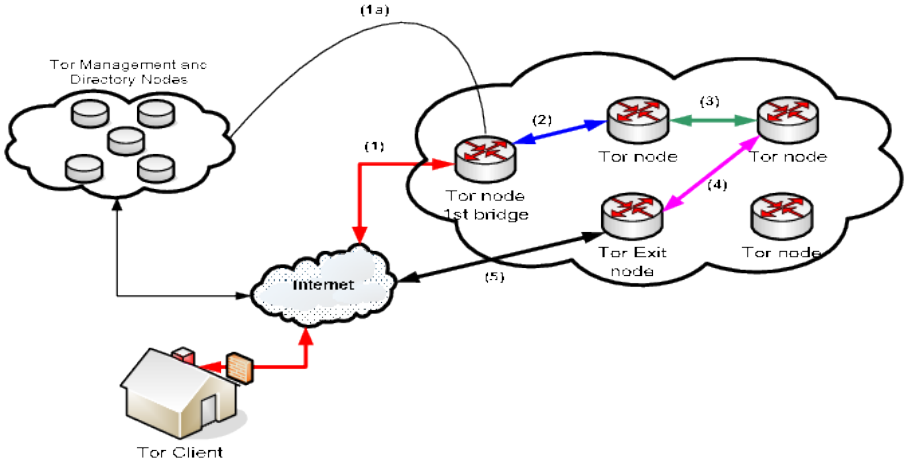

Figure 2.4: Tor Bridges schema

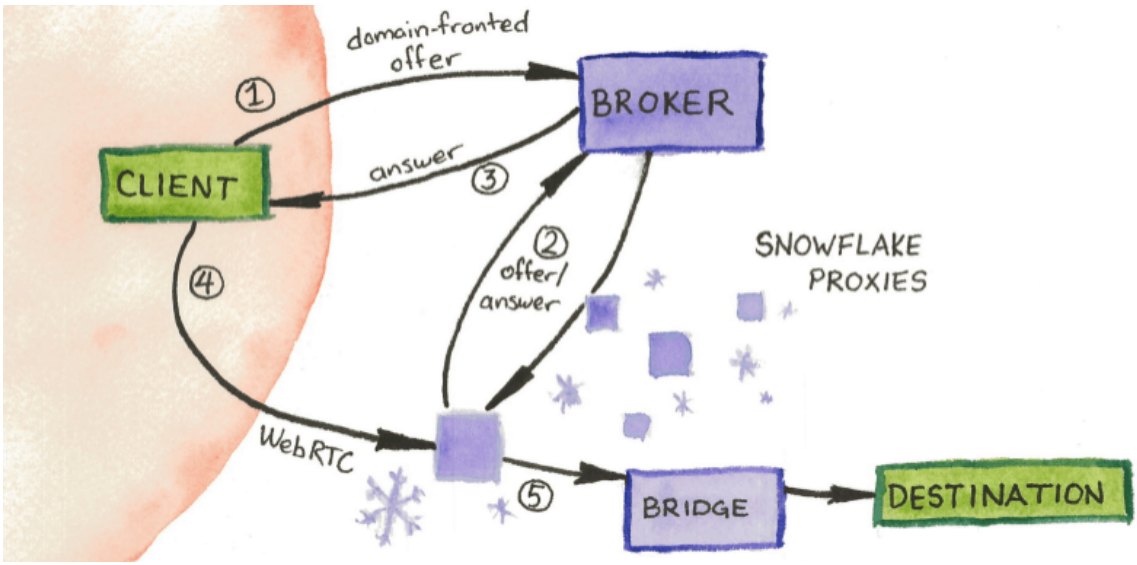

Figure 2.5: Snowflake schematic

A schema of how Snowflake works, courtesy of the Tor project, can be seen in figure 2.5.

2.2 Helios Voting

Helios is an electronic voting system [1] that provides end-to-end verifiable elections. It is a Django[1] based web application that can be run in a self-hosted environment or their public instance[2]. In a Helios instance, any registered user can set up an election and anyone can cast a vote on that election (if the vote is counted or not will depend on whether the election is public or private, i.e. it requires the voter to be on a census).

Helios provides a voting booth that displays the election questions, collects the answers from the voter and creates an encrypted ballot for that election. The user can be convinced that a vote is properly encrypted by decrypting it and discarding it afterward using the tool provided by the system or something done on their own, since the algorithms used to do so are open source. Once it is encrypted and cast, Helios stores the vote and the user receives a receipt. Once the election is closed, the Helios server shuffles the votes using a mixnet and produces a tally.

Helios publishes two boards. One board with all the encrypted votes, that cannot be traced to who emitted that vote. The other with all people who voted with their receipts (optionally, they can be hidden with pseudonyms).

Helios makes extensive use of email to communicate with the voters. In a normal voting process, it sends at least an email with the credentials (username and randomly

[1]Very popular Python framework for web development.

[2]`https://vote.heliosvoting.org/`

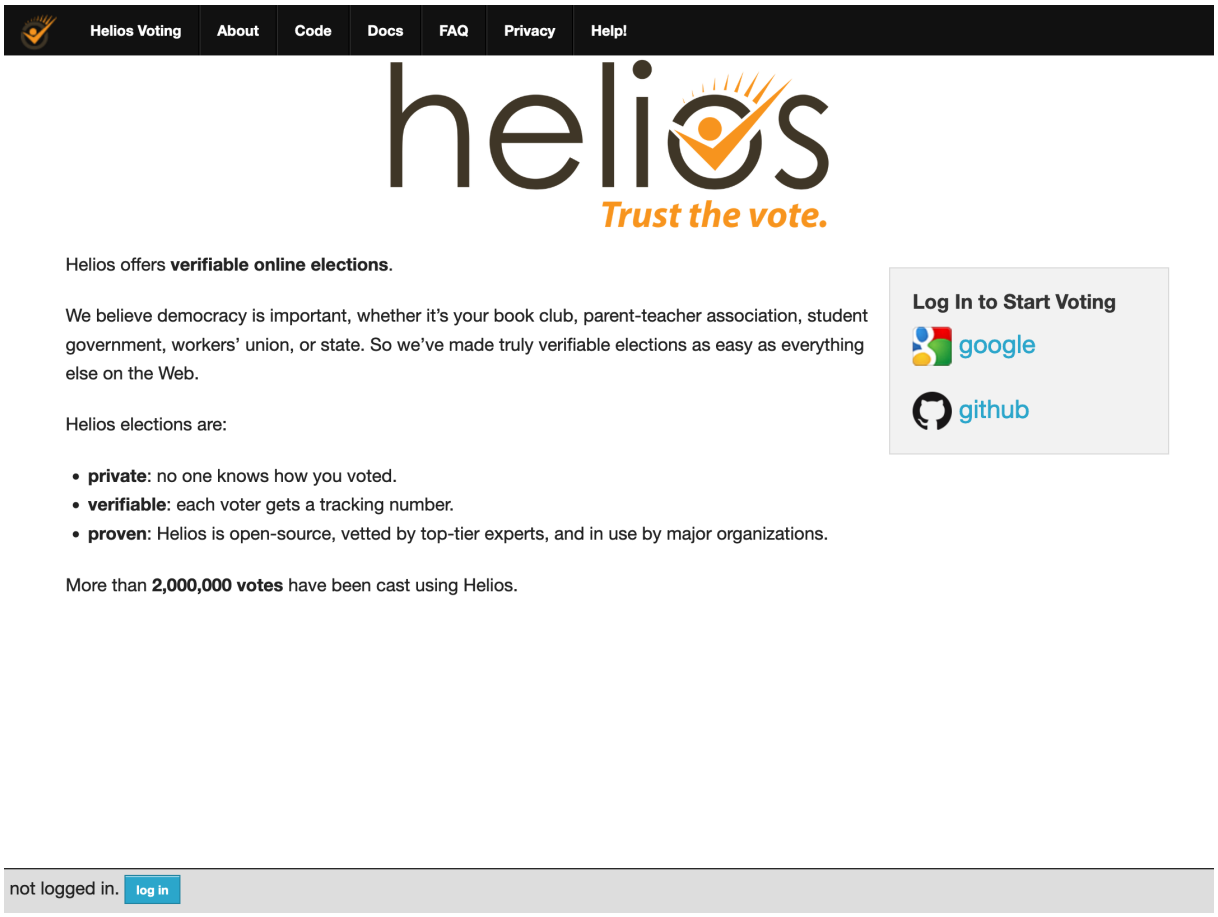

Figure 2.6: Helios Welcome Page

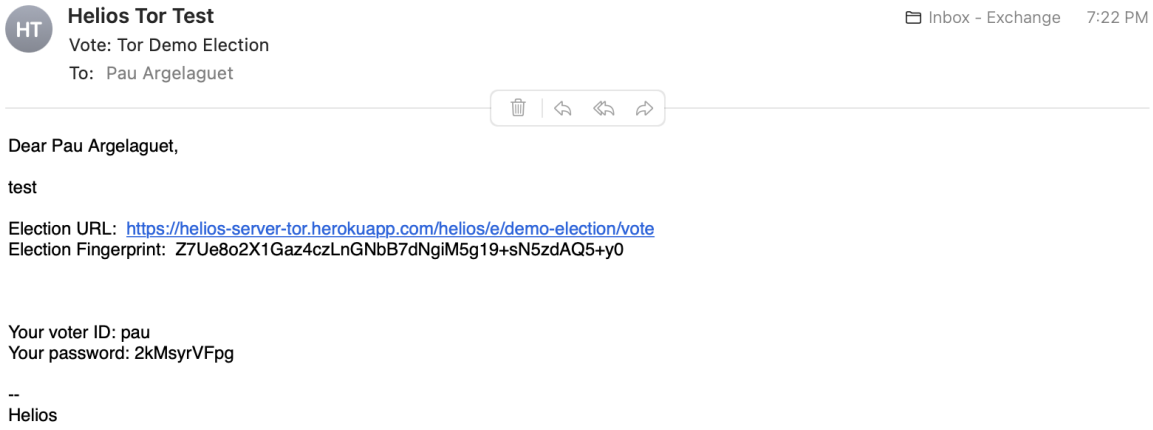

Figure 2.7: Default Helios email credentials

generated, election unique password) as seen in figure 2.7 and another one to confirm a cast vote. In terms of censorship, as long as an attacker does not have access to the voter's email server, those emails are a secure way of communicating election data.

2.3 NodeJS

NodeJS is an open-source JavaScript runtime based on Chromium's V8 engine. Its main function is to execute JavaScript code outside a web browser, that is, as a server-side language, in a desktop application, or even inside a browser through a transpilation step. Node also includes a variety of libraries that allow native access to system resources that are not accessible through the browser, such as the filesystem.

This technology is usually tied to NPM or *Node Package Manager*, a package manager to install and use libraries and third-party code in a project. The ecosystem is well known to have an especially active community and provides a wide range of prebuilt code that implements commonly used features. This is especially useful to speed up development.

NodeJS supports ES6. ES6, also known as *ECMAScript 6* or *ECMAScript 2015*, is a new JavaScript standard that supports many syntactic sugar improvements (such as classes, arrow functions, or constants) that make the developer experience easier and less bug-prone. ES6 is now compatible with all modern browsers and it is used as an industry standard.

2.3.1 Electron

Electron is an open-source NodeJS development framework that enables the development of cross-platform[3] desktop applications using web technologies[4]. It packages a Chromium engine together with the application's code inside a binary, that can be executed without any additional requirements on a desktop system. It also provides many additional APIs, on top of Node's, to interact with the host OS.

Electron is very popular to develop cross-platform applications because its compiler can generate binaries for different platforms without having to change the code.

[3]Including macOS, Linux and Windows.

[4]HTML, JavaScript, CSS and derived technologies.

Chapter 3

Methodology, tasks and, planning

In this section, we will describe an overview of the implementation and how the work is going to be laid out to meet the goals stated in the first section. It is worth noting that four milestones plus a video presentation are required by UOC, therefore, planning is done around those.

3.1 Implementation of the solution

The complete voting solution that will be implemented consists of two parts: a **desktop application** and a **voting system**. The goal of the thesis is to develop the first one, but at the same time, a complete solution must be provided, so an open-source voting system will be used[1].

3.1.1 Voting System

The **Helios Voting** system will be used as a back-end for holding elections. Elections will be set up on that side and the standard Helios' procedures will be used to encrypt the ballots and tally the votes.

Some small changes will be required from standard Helios to enhance privacy. The changes are related to how Helios sends data to the client, including when it sends the election metadata or when it sends a response to acknowledge a correct vote. Those communications have the problem of being quite similar over time. The required changes might involve dropping connections, not responding to some others, inserting randomness to the packets, or sending noise.

The changes on Helios will be implemented in a fork of the Helios repository and they will be deployed in a web server.

[1]The possibility of using a mock voting system was considered but discarded. A meaningful voting experience should be provided, and that was not the case with a mock back-end.

3.1.2 Desktop client

The Helios Voting system is already designed with a heavy reliance on the client. For that reason, it ships with a voting booth, the *Helios Booth*, that is already a relatively standalone JavaScript application. That booth is responsible for fetching the election metadata, rendering the UI for the votes, encrypting the votes, and casting the votes (sending the encrypted packets to Helios). It even contains a utility to validate votes.

The drawback of that JavaScript application is that is fairly complex, written in "old" JavaScript (not modern ES6) and the UI is not usable on mobile devices.

The goal of this project will be to package that JavaScript app into an Electron-based desktop application. Deep changes will have to be made in the application itself since it makes a lot of assumptions revolving around the fact that the application is served from the Helios Django back-end and session cookies are available.

The approach of reusing the booth and not writing a new one from scratch is justified by the fact that most of the crypto stack, in charge of encrypting the votes, can be reused. It is not the goal of this thesis to get deep into the cryptography that enables end-to-end verifiable elections.

A middleware layer will be placed between the Electron app and the Internet, which will handle network connectivity from the app to the Helios server. That middleware will be responsible for injecting and handling randomness to camouflage the packets sent. It will also interact with the Tor network, making use of bridges and pluggable transports.

When a user wants to vote, they will have to select a Helios server by providing a server location (a URL) and an election in which to vote. The front-end client will also handle authentication in case the Helios election is set up in such a way.

3.2 Milestones

This is a high-level overview of the milestones with the official deadlines. Aside from those, a weekly meeting between the author and the thesis supervisor will be held to assess progress and gather feedback on smaller iterations.

3.2.1 First milestone (work plan)

Deadline: Setpember the 28th, 2021.

The first milestone consists of initial the work plan, which is this document. Research has been made to evaluate the problem and come up with a proposed solution. The descriptions in this document have some room for iteration and change, given that technical challenges during the implementation phase will occur.

The goal of the first milestone is to deliver this document.

3.2.2 Second milestone

Deadline: October 26th, 2021.

The second milestone is the first one where coding starts to happen. At this stage, the goal is to have a working Electron application that can cast votes to a Helios instance. No Tor implementation is needed yet at this point.

The deliverable at this stage will be a binary (and its source code).

3.2.3 Third milestone

Deadline: November 23rd, 2021.

The third milestone will take the previous desktop app and add the networking layer, responsible for implementing Tor plus any noise and randomness necessary to camouflage the packets.

The deliverable at this stage will be a binary (and its source code), but containing the changes at this iteration.

3.2.4 Fourth milestone (memoir and source code)

Deadline: December 28th, 2021.

The final milestone will have two sides. On the coding side, it will wrap up the desktop app and fix any outstanding issues, and will implement the changes mentioned in the previous sections on the Helios/Django part. On the writing side, in this milestone, the final document (memoir) will be written.

This final deliverable will consist of the document plus the binary/source code.

Part II

Implementation

Chapter 4

Decoupled Voting Booth

The goal of this section is to describe the design and implementation of the Helios Booth that will be used in the project. Tor is not yet introduced, therefore, a tool is being built such that a voter can participate in an election in a completely decoupled setup, that is, without having to interact with pages rendered by the Helios Server (Django).

At the end of this section, a desktop application is available that casts votes into a *helios-server* modified instance running in a local machine.

4.1 Electron Wrapper

The Booth (from now on, *helios-desktop* interchangeably) is a multi-platform desktop application[1]. It has been built using the Electron framework plus Electron Forge[2] as a packaging tool. The codebase contains a combination of TypeScript and JavaScript as the programming languages of choice plus HTML and CSS for building the User Interface.

Other languages (such as C++ and Rust through WebAssembly) can be used for this purpose. TypeScript has been chosen as the main language for the wrapper code because it is a superset of JavaScript that introduces type checking and thus, type safety. This is an important feature to have when it comes to develop reliable and maintainable software projects.

Another advantage of this setup is that it allows for TypeScript and JavaScript interoperability. Most of the Booth's code has been reused from what is available in the open-source Helios project, which is written in legacy JavaScript. That codebase has not been rewritten in TypeScript but can interact with the modern, ES6-based, typed TypeScript wrapper.

ElectronForge is being used as a packaging tool for packaging the app (creating the platform-specific binary bundles). It is a CLI tool that abstracts some of the logic that ties together the different tools used in the development and build process of the desktop app. More specifically, it orchestrates the transpilation of TypeScript code to JavaScript

[1] Can run on macOS, Linux and Windows

[2] `https://www.electronforge.io/`

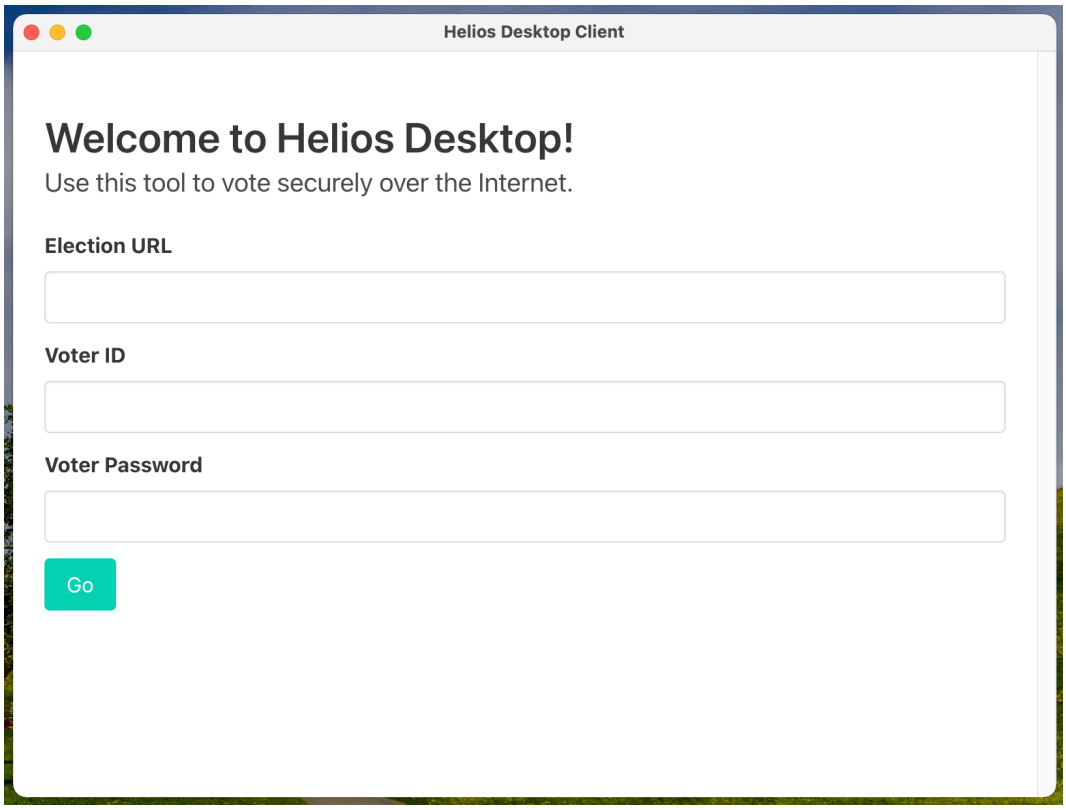

Figure 4.1: Helios Desktop welcome screen

and the creation of executable binaries using those assets. To start up the project, the TypeScript template from ElectronForge was used[3].

The wrapper code contains some basic user interface outside the Helios Booth. Concretely, the welcome screen 4.1 where voters introduce the election URL and their credentials and the success screen 4.2, where users see a message letting them know that they have completed the voting procedure.

The User Interfaces's styling on the screens outside the booth has been done using the Bulma[4] framework. It has been chosen because it is a lightweight open-source package that consists of a CSS file only. By using it, the interface can be styled, have good looks, and be more usable and familiar from a user experience point of view by trivially adding some CSS classes to the HTML elements.

The code has been formatted using the standard tool in NodeJS/TypeScript codebases, ESlint[5] with the Automattic's WordPress configuration[6]. This has been chosen to enforce a code style and automatically format files, which produces better readable and maintainable code. To keep parity with the Open Source Helios project, the Helios Booth files are not being linted.

[3] `https://www.electronforge.io/templates/typescript-template`
[4] `https://bulma.io/`
[5] `https://eslint.org/`
[6] `https://www.npmjs.com/package/@wordpress/eslint-plugin`

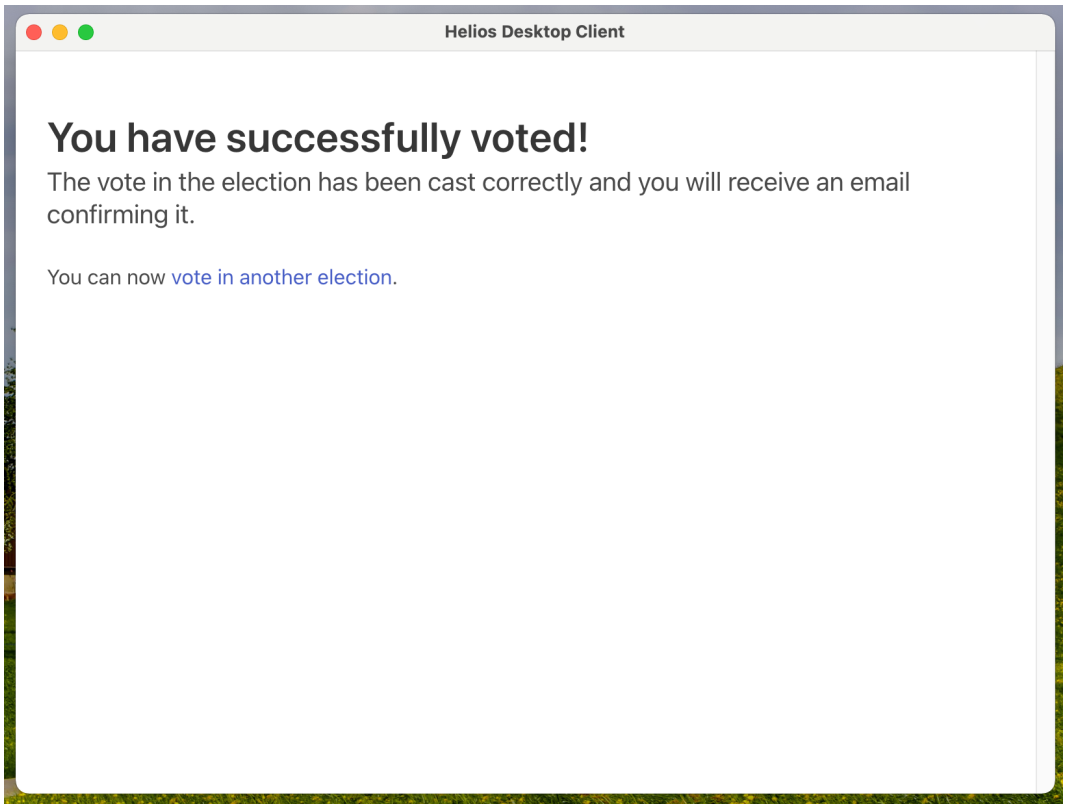

Figure 4.2: Helios Desktop success screen

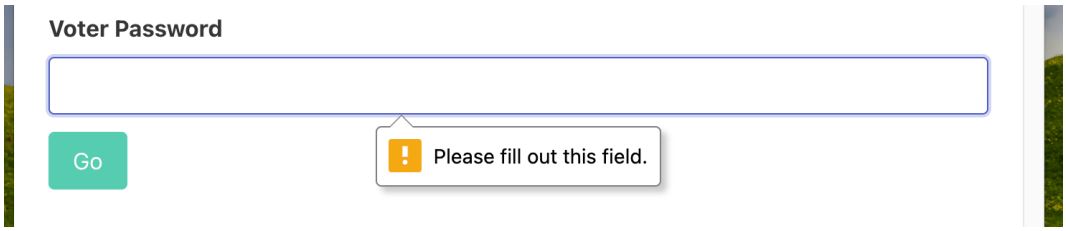

Figure 4.3: Empty field validation error

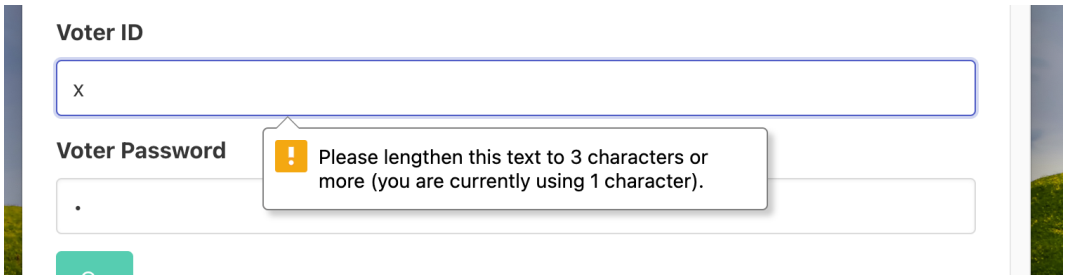

Figure 4.4: Invalid field validation error

NodeJS 16, LTS[7] at the time of writing, and Electron 16 are the versions used to develop this project.

4.1.1 Error Handling

Some basic validation has been added in the front-end to prevent users from inputting invalid parameters. Since these validations are HTML5-based, they can be bypassed relatively easily by a motivated attacker, but they are good enough for preventing involuntary bad usages from normal users.

Validation consists on adding correct types and attributes to the HTML `<input>` tags. The attributes can be seen in the following snippet:

```
1 <input class="input" id="election_url" name="election_url" type="url"
     required />
2 <input class="input" id="voter_id" name="voter_id" type="text" minlength
     ="3" maxlength="64" required />
3 <input class="input" id="voter_password" name="voter_password" type="
     password" minlength="8" maxlength="64" required />
```

Listing 4.1: Input field validation on welcome page

Note that the full use of those attributes is restricted to HTML5-compatible browsers, which is the case with the modern Electron engine that is being used (Chromium).

Figures 4.3, 4.4 and 4.5 show how errors are rendered in the UI.

[7]Long Term Support.

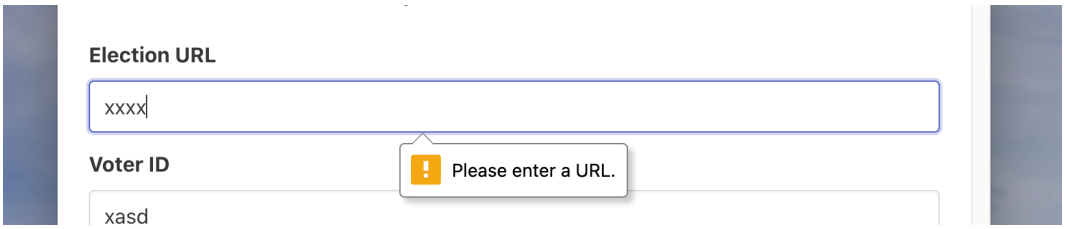

Figure 4.5: Invalid URL validation error

4.1.2 User-provided election URL

The Helios server expects the URLs to **not** end with a backslash. That is, if a user navigates to `http://www.example.com/page/` it will get an error 404 whereas `http://www.example.com/page` will render correctly.

It is not expected that a voter is aware of this limitation. A simple parsing function is added in the wrapper, on top of the validation explained in the previous section, to make sure the requested URL is valid.

```
1 const parseElectionURL = (rawUrl: string): string => {
2   if (rawUrl.endsWith('/')) {
3     return rawUrl.slice(0, -1);
4   }
5   return rawUrl;
6 };
```

Listing 4.2: Election URL parser method

4.2 Changes to the Helios Booth

Although the goal is to not modify the Booth that comes with Helios, some minor changes have been done to it to make it compatible with this project. Some of the changes are not Helios Desktop specific, therefore can be ported to the upstream Helios project.

In this section, those changes will be discussed.

4.2.1 Responsive booth

As is, the Helios Booth is not responsive and it does not adapt to different screen sizes. This is especially problematic on mobile devices, such as smartphones and tablets. This is also important because desktop applications are usually resized and moved through multiple monitors, often with varying sizes and resolutions. What is more, the default Booth is using an old HTML version, HTML 4.

While maintaining the overall structure and appearance, the booth has been migrated to HTML 5 and some layout changes have been made to support a responsive viewport. The changes include repositioning buttons, tweaking CSS so `<divs>` and containers are

centered and have a `max-width`[8] instead of a fixed `width`[9].

This update has been contributed to Ben Adida's upstream Helios repository.

4.2.2 jQuery upgrade

Helios, as is, uses an old jQuery version, 1.2.6 in conjunction with jQuery JSON (an add-on package that was used in jQuery to work with JSON). 1.2.6 was released in 2008 and has multiple security and performance issues. In terms of feature set, it cannot send headers on AJAX requests (a feature that is needed in this project).

For those reasons, a jQuery update was needed. In version 1.5, jQuery introduced built-in support for JSON and the possibility to send headers on requests. Given that the current version of jQuery, 3.x, contains multiple breaking changes that would affect the Helios Booth codebase, it has been decided that jQuery will be upgraded to the latest version of the 1.x branch, 1.12.5 (released in 2016). Although it is not the latest version, it is a fair compromise between new features and backward compatibility.

To perform the migration, jQuery JSON has been removed and all the code that was using it has been replaced by the built-in jQuery/vanilla JS methods for JSON manipulation. These were the only breaking changes of the upgrade.

This update has been contributed to Ben Adida's upstream Helios repository.

4.2.3 Underscore update

The NodeJS proxy that will be introduced in further sections requires some advanced NodeJS features plus the ability to move data through the main and the UI threads in the Electron application.

This behavior is disabled by default in Electron, and it has to be enabled by adding the following configuration:

```
1 const mainWindow = new BrowserWindow({
2   width: 1280,
3   height: 800,
4   webPreferences: {
5     nodeIntegration: true,
6     contextIsolation: false,
7   },
8 });
```

Listing 4.3: Electron configuration that breaks old Underscore

Those changes are acceptable for the configuration of this project, but they break compatibility with the included Underscore[10] version, which is in a similar place to jQuery (the Booth includes version 1.1.6, released in 2009). This is a library that provides general utility functions for JS.

[8] https://developer.mozilla.org/en-US/docs/Web/CSS/max-width

[9] https://developer.mozilla.org/en-US/docs/Web/CSS/width

[10] https://underscorejs.org/

Similar to jQuery, Underscore has been upgraded to a recent version, 1.13.1, which fixes the compatibility errors that would not let the page render.

4.2.4 Custom JS minification script

The Helios booth uses a minified version of its own JS files plus the libraries, which boosts performance. This includes the two upgraded libraries.

Helios provides a text file with a script to generate that file, but the script relies on an old version of a minifier software plus has some broken paths. To compile the changes from both previous sections, a custom (executable) bash script has been added:

```
1 #!/bin/bash
2
3 uglifyjs js/jquery-1.12.4.min.js js/jscrypto/class.js js/jscrypto/bigint
    .dummy.js js/jscrypto/jsbn.js js/jscrypto/jsbn2.js js/jscrypto/sjcl.
    js js/underscore-min.js js/jquery.query-2.1.5.js js/jquery-jtemplates
    .js js/jscrypto/bigint.js js/jscrypto/random.js js/jscrypto/elgamal.
    js js/jscrypto/sha1.js js/jscrypto/sha2.js js/jscrypto/helios.js -o
    js/20211010-helios-booth-compressed.js
```

Listing 4.4: Custom script to minify JavaScript code

4.2.5 Election exit button

The Booth contains an exit button that, in the web version, closes the booth and returns the user to the election homepage (described as *election url*). If unchanged, this link would bring the user back to a public URL hence performing network traffic that is susceptible to being pattern-matched and thus giving away that a user might be participating in an election.

In helios-desktop, the exit button has been changed so it redirects to the initial page 4.1 of the booth. The confirm dialog has been kept as is 4.6.

The email button has not been changed because it does not perform a request, it simply opens an email window.

4.2.6 Error handling in the booth

The majority of the booth's code regarding error handling has been left unchanged because overall the booth itself is good enough communicating to the user that something went wrong. However, there is one part of the code that no longer works with the current setup – that is when a user requests to vote in an election with incorrect parameters, that is, incorrect URL or credentials.

Standard Helios relies on redirections to the Django server to handle those cases. If the user is not authenticated or requests an incorrect URL, it is redirected to a login page within Django. In the decoupled setup, no such page exists.

In this implementation, the error cases are handled through HTTP status codes. The Helios Booth fires two requests when loading an election, and a third one if they both

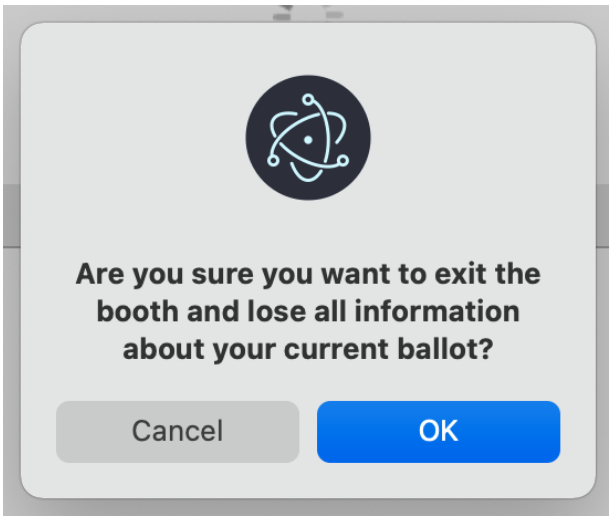

Figure 4.6: Helios Desktop exit confirmation dialog

succeed. To prevent leaving the user in a dead state, the HTTP status code is checked. Unless it is 200, which is the expected one, the user is redirected to the initial page with an error message.

```
1  BOOTH.load_and_setup_election = function(election_url) {
2      $.ajax({
3          url: election_url,
4          method: "GET",
5          complete: function(response) {
6              if (response.status === 200) {
7                  // let's also get the metadata
8                  ...
9              } else {
10                 alert('Could not load the election.');
11                 window.location = "../index.html";
12             }
13         }
14     });
15     ...
```

Listing 4.5: Loading election data in the booth

4.2.7 Authenticating requests through headers and parameters

Helios works by having a decoupled booth that displays the election questions and lets the user create an encrypted payload containing the vote. However, this vote is sent in a form (as a POST request) to a Django-rendered page (*cast* page). Then, Django is responsible for displaying an authentication form and submitting the vote in the database by redirecting the user to the *cast confirm* page. The issue with that is that Django relies on a cookie-based session, which is a thing that is aimed to be avoided in this project.

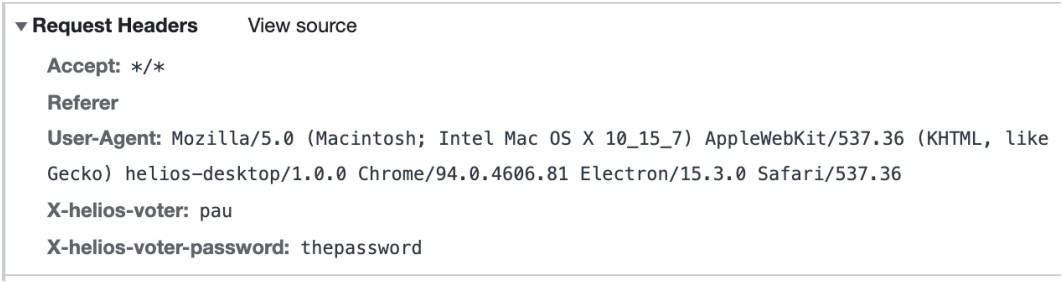

Figure 4.7: Helios Desktop custom authentication headers

```
▼<form method="POST" action="http://localhost:8000/helios/elections/de28f03a-24e3-11ec-
a544-acde48001122/cast_confirm" id="send_ballot_form" class="prettyform">
    <input type="hidden" name="election_uuid" value="de28f03a-24e3-11ec-a544-acde4800112
    2">
    <input type="hidden" name="election_hash" value>
    <input type="hidden" name="helios_voter" value="pau">
    <input type="hidden" name="helios_voter_password" value="KdVYU7BsAg">
  ▶<textarea name="encrypted_vote" style="display: none;">…</textarea>
  </form>
```

Figure 4.8: Additional form parameters on the booth

Ideally, the booth should do a single request to Django containing the vote **and** the user credentials.

Fetching the election metadata is also an issue. In public elections, the client can hit the user-provided URL and Django will return the data (being the election data or the randomness). Whenever a user hits a private election, Django will rely on a cookie-based session to determine if the data can be returned or not.

In this case, a way of providing an authentication other than the cookie that is not present in the Electron app to the request is needed.

In helios-desktop, the user has been asked for some credentials at the beginning of the process, in conjunction with the election URL. Those credentials are being stored in the local storage of the application[11]. Having them stored in *localstorage* means that the data is available in all the views of the app. The local storage is cleared every time the main application window is rendered.

The situation is fixed as follows:

- For the REST-like requests the client passes the authentication parameters through HTTP headers, *X-helios-voter* and *X-helios-voter-password* (as seen in figure 4.7). The affected requests are /election/id (metadata) and /election/id/get_randomness, both of them being GET requests.

- For the POST form, two additional hidden fields have been added (figure 4.8).

In both cases, it is safe to transmit the credentials because the requests go through an encrypted connection (HTTPS).

[11] `https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage`

The changes required in the Django side to support this newly-introduced authentication are described in the following sections.

Chapter 5

Adapting Helios Django Server

The goal of this project is to change the open-source Helios project the least possible. That has been the case for Helios Booth, as described in the previous section. The changes in the back-end side, a Python/Django-based server are quite minimal and have to do mainly with authentication.

None of these changes are contributed back to Ben Adida's main project because they are helios-desktop specific. It is worth noting that none of the changes break the normal functionality of Helios, so elections without helios-desktop can still be held in a modified system.

5.1 Reading authentication headers

Due to the decoupled nature of the client, the server is not able to keep track of sessions. Therefore, the client sends non-standard authentication headers `X-helios-voter` and `X-helios-voter-password`. They both contain a string.

The helios-server code includes a `security.py` file with security-related helper functions. One of those functions is `get_voter` which is responsible to get a `Voter` object from the database and it is commonly used across the views that return election data and cast votes.

In lines 55 to 64 in figure 5.1 it can be seen how the headers are retrieved and then used to search in the database. If a voter is found with those parameters, it is injected into the current session.

5.2 Merging two-step cast confirmation

Helios casts votes using a two-step process, that is, the user goes through two endpoints. When a normal booth finishes, a `POST` request is sent to `elections/<election_id>/cast`. That view gets the user from the request and **saves the encrypted vote in the session**. Then, the user is redirected on the second screen, `elections/<election_id>/cast_confifm`. Crucially, this first view redirects the user to a login page in case they are not logged in.

Listing 5.1 shows the Helios logic of the view.

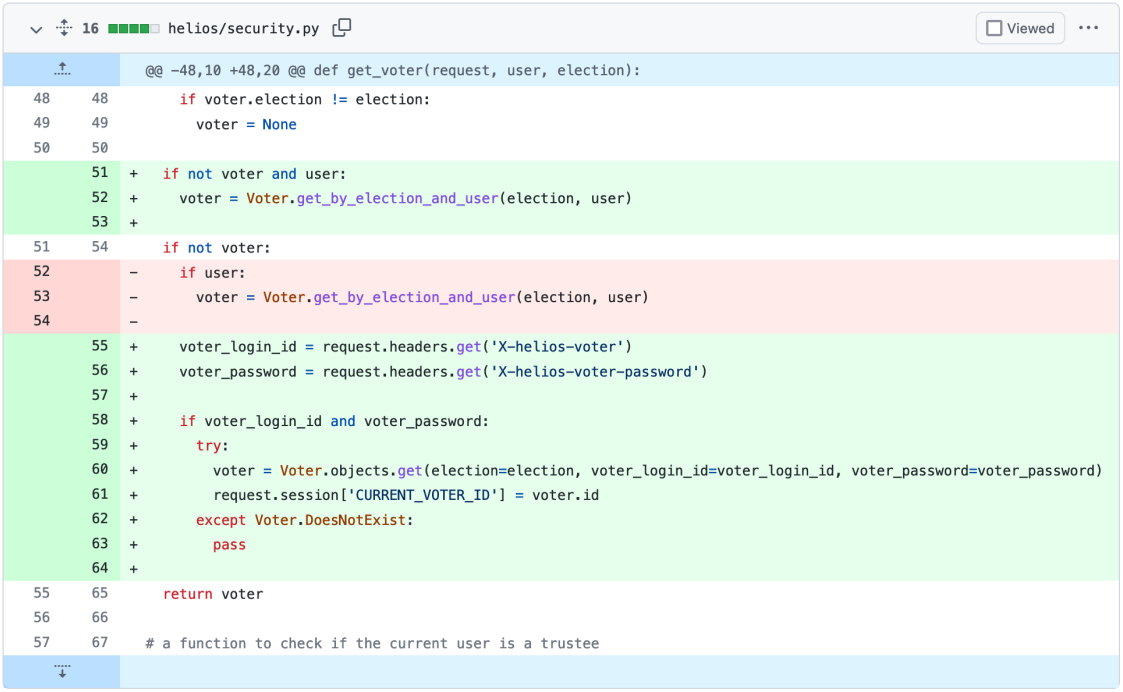

Figure 5.1: Changes in the security.py file

```python
@election_view()
def one_election_cast(request, election):
  """
  on a GET, this is a cancellation, on a POST it's a cast
  """
  if request.method == "GET":
    return HttpResponseRedirect(settings.SECURE_URL_HOST + reverse(
        url_names.election.ELECTION_VIEW, args = [election.uuid]))

  user = get_user(request)
  encrypted_vote = request.POST['encrypted_vote']

  save_in_session_across_logouts(request, 'encrypted_vote',
      encrypted_vote)

  return HttpResponseRedirect(settings.SECURE_URL_HOST + reverse(
      one_election_cast_confirm, args=[election.uuid]))
```

Listing 5.1: One election cast method in views.py

Casting the vote and saving it to the database happens in the second view. That view expects to have the `encrypted_vote` in the session object, but that is only the case if the first view got called.

Given that one of the goals of this thesis is to reduce the amount of traffic so it is harder to trace for third parties, it has been decided that the first call is going to be skipped and the booth will call `cast_confirm` directly.

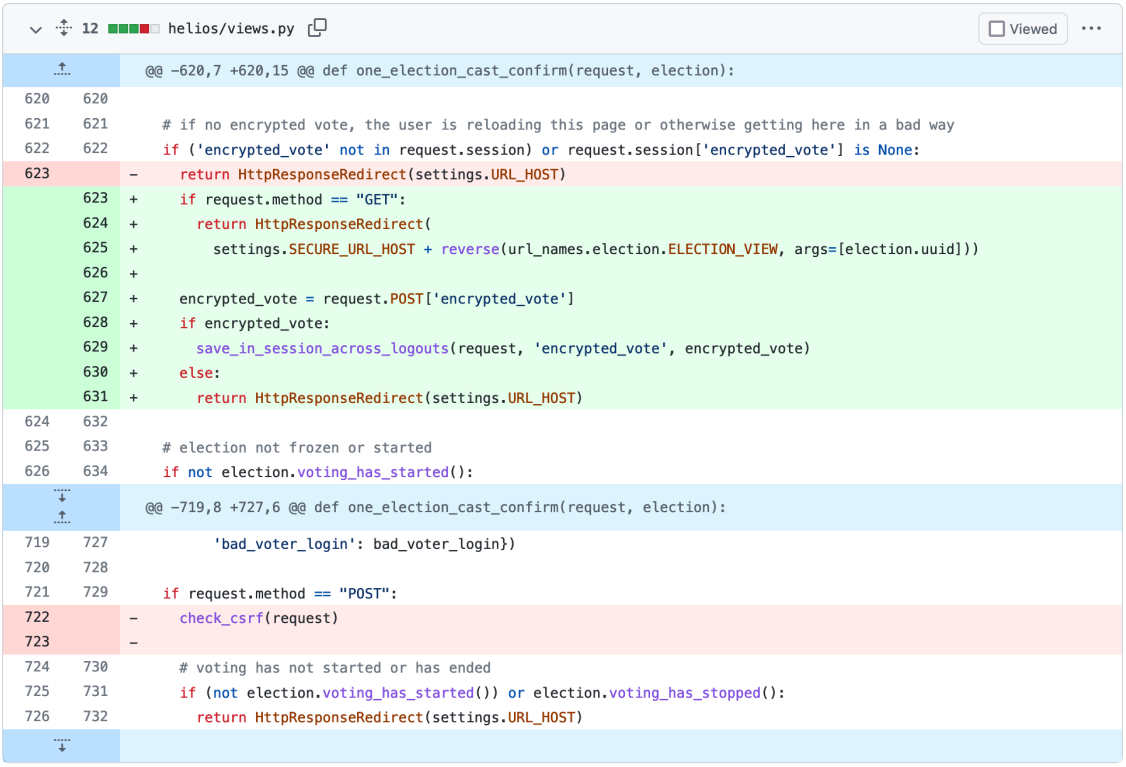

Figure 5.2: Changes in the views.py file

In figure 5.2 the changes performed to the view can be seen. Instead of redirecting the user out when there is no encrypted vote in the session, the view tries to fetch it from the POST request that the booth sent.

Since the booth lives in a different environment than the server, the csrf check has been disabled.

Chapter 6

Deploying Django to Heroku Cloud

Helios is a Django-based Python application that requires a web server, a database, and a Celery[1] worker system (a process plus storage to store the queue). This is a pretty standard configuration that can be deployed in a myriad of cloud services and on-premise servers.

To create a realistic election environment in this thesis (aside from a local instance for development), the (modified) Django project has been deployed to the Heroku platform[2]. A server on the Internet is needed to have the full Tor circuit[3].

One of the main reasons to use Heroku is the fact that Helios is prepared to be deployed there out-of-the-box and it has community support for being deployed there [11]. The other reason is that Heroku has a very generous free plan[4] that allows low-traffic services, like this one for demo purposes, to run.

More specifically, this project has been deployed in the free tier using two dynos[5] in the *Europe* region. Deploys have been set up such every commit on a defined branch in the project's repository triggers a new build. The chosen database has been Postgres (known to be the one that works best with Django) and Redis to power the Celery queue (all those add-ons are also in the free tier).

In figure 6.1 the two Dynos that are being used in the demo configuration for this project can be seen.

Helios manages authentication for administrators (not to be confused with *voter* authentication) with third-party providers, such as Google, Facebook, or GitHub. For this setup, Google has been chosen and a demo key pair for their OAuth system has been created.

Heroku does not provide a built-in way to send emails. Helios makes intensive use of emails to send different communications to the administrators (a voter file has been processed, an election tally has been computed, etc.) and voters (voter credentials, vote verification).

[1] `https://docs.celeryproject.org/en/stable/`

[2] `https://www.heroku.com`

[3] A local onion server could have been used, but it is not a scenario likely to happen in a real election.

[4] `https://www.heroku.com/pricing`

[5] Dynos are what Heroku calls their instances.

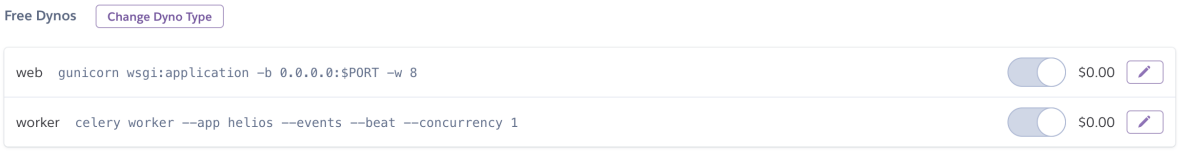

Figure 6.1: Heroku Dyno set up

Amazon SES[6], part of AWS has been used to send emails. Sending a low volume of emails is free of charge, so the setup is good enough for the demo.

[6]Simple Email Server, `https://aws.amazon.com/ses/`

Chapter 7

Sending network traffic through a proxy

The key to this project is to disguise the traffic that the Helios Booth produces so it is not easily identifiable by an attacker who might potentially have control over the network. A myriad of ways of doing so are possible, but the constraint that has been running all over this project comes up again – the Helios Booth code is old and it is difficult to modify.

As seen in previous sections, the Helios Booth performs HTTP over TLS (HTTPS) AJAX calls, using the jQuery framework. A standard solution in the industry to introduce modifications in a request is to run a proxy server. This way, the legacy app sends and receives data as if it were communicating to a *normal* server while allowing a modern codebase to treat that request.

Figure 7.1 illustrates how the network traffic flows in the application, from the local Helios Booth to the public Helios Server living in the cloud. The diagram also introduces Tor, which will be discussed in the next chapter.

7.1 Proxy architecture

The proxy used in this thesis is a NodeJS-based proxy, built around the ExpressJS framework[1] and the http-proxy-middleware library[2]. The proxy has been written in TypeScript.

The reason to use a NodeJS-based proxy is that the application container is Electron, a NodeJS-based platform. Although a CLI process could be called from Electron, the fact of calling other JS/TS code makes interoperability easier. This is crucial because the Proxy and the UI need to exchange data during the life cycle of execution. This also provides out-of-the-box logging in the application's console, which helps a lot while to debug.

The proxy is spawned as an HTTP server running in localhost, port 9051. That port number has been chosen because it is one more than the default Tor proxy port. Since

[1] https://expressjs.com/

[2] https://github.com/chimurai/http-proxy-middleware

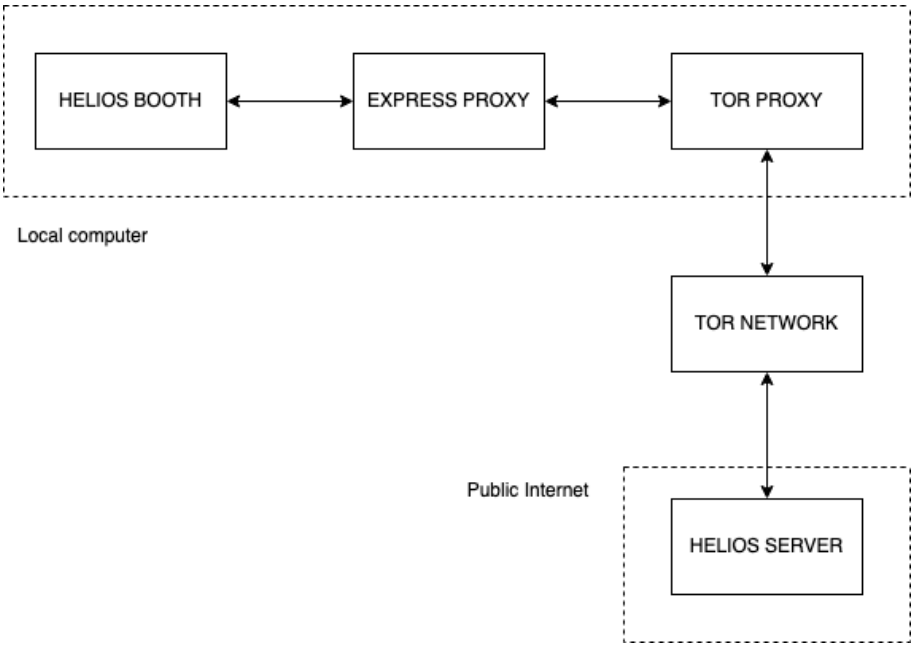

Figure 7.1: Proxy network traffic

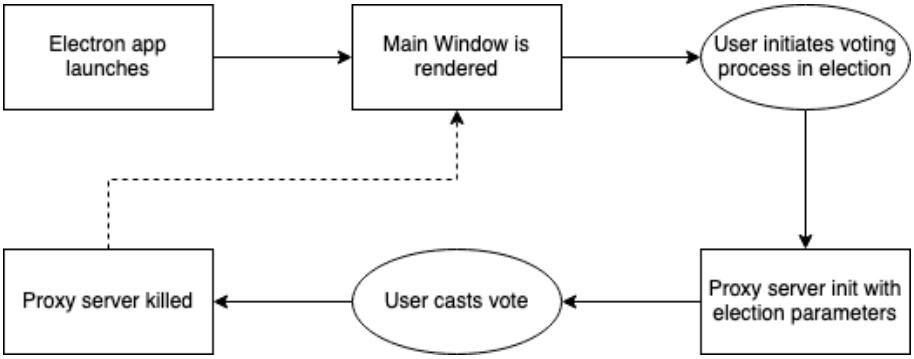

Figure 7.2: Proxy app life cycle

it is an HTTP server, that means is an HTTP proxy that all applications that can send and receive HTTP traffic, like the Helios Booth, can talk to.

Figure 7.2 contains a diagram illustrating the life cycle of the proxy component of the application. It can also be followed in listing 7.1.

It is important to point out that the proxy is spawned up for a specific election and is shut down when it finishes. This is due to the proxy not knowing the `election_url` in advance (before the user inputs it), so it does not know where to proxy to. Also, by not actively running the proxy unless it is strictly needed, system resources are being saved. For a similar reason, the user's credentials are passed in this step.

Only when the server is up and ready, a signal is sent back to the front-end to render the Helios Booth. This has to be done to prevent requests from failing because the forked version of the booth triggers a request to `localhost` immediately after rendering.

```
function createWindow() {
  ...

  const mainWindow = new BrowserWindow({...});

  mainWindow.loadFile(path.join(__dirname, '../src/views/index.html'));

  ipcMain.on(EVENT_STORE_DATA, function (event, store) {
    ...
    proxy = runProxy(target, voterId, voterPassword, () => {
      mainWindow.webContents.send('redirect');
    });
  });

  ipcMain.on(EVENT_PROXY_KILL, () => {
    ...
      proxy.close();
    ...
  });
}
```

Listing 7.1: Main function of the Electron application

7.2 Communicating processes

In Electron, the UI and the back-end run in separate system processes. That is, even though it is all JavaScript/TypeScript code, the memory space is different for both components and functions and variables can't be directly accessed. What is more, some functions are not available in all places (for instance, the `window` object is only available in the front-end.

To overcome that limitation Electron exposes to the developer a publisher-subscriber model, based in NodeJS's `EventEmitter`[3]. Two objects are used, `ipcRenderer` [4] in the front-end and `ipcMain` [5] in the back-end.

The two objects expose a simple API to subscribe to a channel, which provides a callback function that is executed whenever an event occurs to the channel. That event may also carry data. The same API also has the `send` function, which creates an event and attaches optional data in a channel.

Listings 7.2 and 7.3 exemplify the process of sending and receiving data using the model. This procedure is used to send the events in figure 7.2.

```
document.addEventListener('DOMContentLoaded', function () {
  ...

  const form = document.getElementById('election_initiator');
  form.onsubmit = (event: Event) => {
```

[3] https://nodejs.org/api/events.html#events_class_eventemitter
[4] https://www.electronjs.org/docs/v14-x-y/api/ipc-renderer
[5] https://www.electronjs.org/docs/latest/api/ipc-main

```
    event.preventDefault();

    const electionUrl: string = (
      document.getElementById('election_url') as HTMLInputElement
    ).value;
    const target: string = parseElectionURL(electionUrl);

    const voterId: string = (
      document.getElementById('voter_id') as HTMLInputElement
    ).value;
    const voterPassword: string = (
      document.getElementById('voter_password') as HTMLInputElement
    ).value;

    ipcRenderer.send(EVENT_STORE_DATA, {
      target,
      voterId,
      voterPassword,
    });
  };
});
```

Listing 7.2: Send data from the UI

```
document.addEventListener('DOMContentLoaded', function () {
  ...

  const form = document.getElementById('election_initiator');
  form.onsubmit = (event: Event) => {
    event.preventDefault();

    const electionUrl: string = (
      document.getElementById('election_url') as HTMLInputElement
    ).value;
    const target: string = parseElectionURL(electionUrl);

    const voterId: string = (
      document.getElementById('voter_id') as HTMLInputElement
    ).value;
    const voterPassword: string = (
      document.getElementById('voter_password') as HTMLInputElement
    ).value;

    ipcRenderer.send(EVENT_STORE_DATA, {
      target,
      voterId,
      voterPassword,
    });
  };
});
```

Listing 7.3: Receive data in the main process

7.3 Injecting headers to proxied requests

Departing from an initial implementation where the Helios Booth code was modified to inject the authentication headers, with a proxy in place that change is no longer necessary.

Given that the user's credentials are available from the front-end signal, at the proxy's startup time, they can be inserted automatically on every request. Therefore, the booth front-end does not need to know the credentials nor it does need to insert additional fields in the submission form.

Another header is inserted aside from authentication. That header is noise. This header is ignored by the Helios server, so it does not affect the result of the request. However, it does change the size of the request randomly, therefore making it more difficult to identify a pattern in a packet – two voters voting the same options in the same elections would have packets of different sizes.

Listing 7.4 shows how the additional data is injected into every request.

```
1 headers: {
2   'X-helios-voter': voterId,
3   'X-helios-voter-password': voterPassword,
4   'X-noise': crypto
5     .randomBytes(Math.floor(Math.random() * 1024))
6     .toString('hex'),
7 },
```

Listing 7.4: Header modification in the proxy

7.4 Randomly delaying requests

To set up an election, the already mention three requests are fired from Helios. Two are fired in parallel whereas the third one happens once the first one is done. This is a pattern an attacker might match against.

As shown with the header injection, a good way of making pattern matching more difficult is to add noise. For this reason, instead of a deterministic time between requests, in this project, random delays are being inserted between requests.

This is accomplished by adding a simple middleware to the Express (proxy) server. A middleware, in the Express framework, is a function that is put in the request/response pipeline and takes three arguments: the request, the response object, and a `next` function, which is the following function to be called in the pipeline. A middleware works by being called by its previous function, altering the request and/or response objects, and then calling the next function in the pipeline. The proxy library that is being used in the project is essentially a middleware.

For randomly delaying requests, the middleware in listing 7.5 is inserted. This function does not touch the request or response object, but it simply adds a random delay by using a JavaScript timeout. The value of that timeout, as per `setTimeout`[6] definition is in milliseconds, which take a random value from 0 to 4096 (around 4 seconds).

[6] `https://developer.mozilla.org/en-US/docs/Web/API/setTimeout`

This value has been chosen because introducing bigger delays could result in a subpar user experience – there are (at least) four requests in an election, so it would mean at worst 16 seconds of pure wait, plus the actual time that the request takes.

```
app.use((req, res, next) => {
  setTimeout(next, Math.floor(Math.random() * 4096));
});
```

Listing 7.5: Randomly waiting to respond to requests

Chapter 8

Integrating Tor in the client

Tor was intentionally left out from the previous chapter, even though it is technically a proxy. Tor is designed to be a transparent layer for applications as long as they can talk to a SOCKS proxy.

Since the Helios Booth does not support such a connection, the NodeJS-based proxy is used to route the traffic through Tor. After it has done the header modification tasks, instead of calling the webserver directly, it makes use of the `tor` CLI to route the traffic.

Tor is usually run using the Tor Browser, a modified version of Mozilla Firefox with the Tor Proxy integrated. The Helios Booth does not need the Firefox rendering engine, since that is being provided by Electron (Chromium). The other mainstream way of running Tor is by using its command-line interface (CLI), `tor`. This tool is available on the main desktop platforms and it has a straightforward way of running.

Running Tor as a containerized in Docker has also been evaluated, but discarded because it would not provide any benefits, especially given the wide availability of installers. Docker Desktop also has a restrictive license in some platforms and adds a significant weight on execution, that can be avoided.

The `tor` CLI has been chosen to run on the background. By default, when launched this tool establishes a circuit and acts as a SOCKS proxy in port 9050 for all local traffic that is thrown at it.

In contrast with the purpose-built HTTP proxy, this one is a compiled binary so it cannot be interacted with easily from Node. This is not an issue, and this first version of the desktop app assumes that as long as the Tor proxy is up, it will forward the traffic correctly. Also, it assumes that the application is installed in the system. In the future, Tor could be installed with the app's installer if it were not detected.

Since Tor is designed to be a general-purpose proxy, its lifecycle is different from the HTTP proxy. Whenever the Electron is launched, the Tor proxy is launched and when the application quits (not to be confused with when the user finishes an election), the Tor proxy quits.

Aside from simplicity, this approach has been chosen because Tor startup might take some time. It depends on the current user network setup, but establishing a working circuit is not instant. For this reason, when the application is launched, Tor starts this process. While it happens in the background, the user is typically filling in the election

details, which gives Tor enough time to be ready when it is needed (the time when the first call to the election server happens).

Tor is run as a separate process spawned from the main Node process, using `child_process`. Code can be seen in listing 8.1.

```
1 // main.ts
2 import * as child from 'child_process';
3 ...
4 child.exec(TOR_COMMAND);
5
6 // constants.ts
7 export const TOR_COMMAND = 'tor -f bridge.1';
```

Listing 8.1: Tor execution within Node

8.1 Using bridges

By default Tor already proves a very good degree of anonymity. However, in hostile environments bridges may give the user one step further in terms of anonymity.

Given the transparent nature of the Tor proxy, the rest of the booth is independent of whether Tor is using bridges or not. As long as the Tor proxy functions correctly, bridges are just a configuration detail of that proxy that happens to be more secure. The only user-facing impact *might be* the connection speed.

Bridges are not a public, permanent list, but they keep rotating. That is the reason why they must be set up and rotated in short periods by the user of the application. The `tor` CLI accepts an argument, `f`, followed by the location of a bridges file (a file that holds the addresses of the servers that the computer will connect to). That is a user-modifiable text file and will be automatically picked up by the application when it starts.

Listing 8.2 holds a sample bridges file. Bridges can be obtained from `https://bridges.torproject.org/`.

```
1 Bridge obfs4 188.126.94.109:8080
     D734D62C13012A9B8E49F4BDCA98F355B71214DE cert=
     cD8xzwtH9DWgXb21kMRulQIEUUE5pUyJqRUmB17r/zPnemkcJuHK/7
     vMLAgCo0DaXlfsHg iat-mode=0
2 Bridge obfs4 85.2.43.214:3480 7DD0C82D0D2586673809338BFEC612951E4DC1AB
     cert=+WKZ74k2ARc876puYn49KRxFqN83REVwWcb8VT3T2Il44XOcIq4KlU+
     cyygcqLkEjhotOg iat-mode=0
3 Bridge obfs4 81.174.147.57:53677 3
     AF500006A1637D21B1F5D3E64F63882A9623717 cert=4MKPza4o09lrOiAB+/xcDqUx
     /b51YbDj3mbnRA8eKlmqFIK/jCyh4PesG3kBvjN4UukrRg iat-mode=0
```

Listing 8.2: Sample bridges file

For this demonstration, Tor's Snowflake has finally not been used. Following the same reasoning made with bridges though, it could be added transparently to the rest of the application as long as the `tor` CLI supports it.

Part III

Conclusion

Chapter 9

Revisiting coercion issues

According to the problem statement of this thesis, the goal of this thesis is to prevent coercion in electronic elections. Coercion can be done by a third party if that third party knows that somebody is voting in an election. Therefore, the focus during this implementation has been to prevent a third party from obtaining that information.

In this chapter, the ways an attacker can detect that a person is voting will be evaluated against the protections that this thesis has introduced.

9.1 Coercion issues

9.1.1 Application asset load

Internet-based electronic voting systems work by downloading a web page through a web browser where the voting occurs. That includes *classic* Helios. The issue in this procedure is that the static files (HTML, CSS, and JS) are the same or quite similar for all users, so it is easy for an attacker to pattern-match a user that is connecting to a page.

The issue is also related to timing. When a user goes into those systems, the packets that will probably follow after the application is loaded are the ones containing the voting material.

Helios Desktop fixes this by using a desktop application. There is no need to connect to the Internet to fetch the application because that is already in the computer of the voter. If the voter downloads the binaries in a moment different from the one they are voting, it is more difficult to correlate by the attacker. In a more sensitive scenario, the voter does not even need to download the binaries – they can be transmitted by offline means, for example, a USB stick.

9.1.2 Packet interception

Once a user has the voting application ready, it starts the voting process. A package exchange occurs, which includes the election metadata from the server and the cast votes from the user. An attacker might perform a *man-in-the-middle* attack and might be able to intercept that exchange partially or completely.

In the worst-case scenario, the attacker can read the contents of the package. That is mitigated in Helios Desktop by forcing using HTTPS connections, so as long as the server where the user is connecting to vote is compliant with TLS and certificates are correctly issued, the contents of the connection should not be sniffed. On top of that, since Helios Desktop uses Tor, this system also adds three layers of encryption to the packets.

At this point, the attacker might not *know for sure* what the user is voting, but the sole fact that the user is voting might let the attacker infer what the user's choices are. This might be contextual data, for instance, a user lives in a neighborhood that typically votes for a determined party. It is still important to prevent the attacker from knowing that the user has voted at all.

Assuming an attacker has (at least) a partial control of the network in which the user is interacting with the voting system, they can see that a device is sending packets towards a server known to host an election. Hence, it can be inferred that somebody is voting. This is the classical use case for Tor – to provide anonymity by routing the traffic through several hops. Helios Desktop routes **all** traffic through the Tor network.

9.1.3 Traffic pattern matching

With the protections described so far, an attacker might suspect about a user, but the only thing that they will know for sure is that a user is sending and receiving data using the Tor network. That by itself might raise some suspicion if the user does not use Tor frequently or if they are the single user in a network or geographical area using Tor.

The key concept is pattern matching [9]. Without knowing the actual contents of the data, an attacker will attempt to learn more about the user by analyzing information that is leaked by the connection. The goal of Helios Desktop is to minimize the amount of information that is leaked and making the bits that are leaked meaningless or difficult to analyze.

Knowing a user is using Tor is the simplest way of pattern matching some traffic – a user talking to a well-known set of Tor nodes. There is plenty of literature [14] [17] and work from the Tor Project about circumventing this issue, which is indeed not unique to this thesis. The most usual way of going around it is by using bridges, which Helios Desktop does use.

By using Tor bridges, the only thing that an attacker might analyze is the number of packets, their size, or their frequency. This is why Helios Desktop injects randomness on the headers of the requests, so size is not constant, even if two voters are casting the same options in the same election using similar setups. Helios Desktop also introduces random delays between requests, so time-based pattern matching is more difficult.

Some further improvements could be done on this front, as well. For instance, the proxy could send or drop packets randomly, to further introduce noise on the wire. The issue with this kind of behavior is that they go against a good user experience and require additional error handling from the booth's side.

Finally, the proxy could also cache duplicated requests, so in case a user asks for the same piece of data twice, that does not emit a network request. Implementing this feature is not trivial not only on the actual cache side but on the application lifecycle side because

if the proxy gets killed every time a user starts an election, chances of repeating a request during that period are low.

9.2 Threat modeling the attacks

A fundamental concept in cybersecurity is threat modeling – assessing a system's security taking into consideration the resources and willingness of a potential attacker. Simply put, if it is more expensive for the attacker to perform an attack than the potential gain when succeeding, the attacker has little to no incentive to carry it through.

This chapter has been laid out as a list of concerns that have been addressed. More interestingly, those concerns can be seen as layers on top of each other, making an attack increasingly more difficult. It is intuitive to see that a less resourceful attacker might not be able to coerce an election even without any of the protections of this thesis, whereas a very powerful one might render all the protections laid out noneffective.

The coercion attacks this thesis is fighting against are network-control related. That means that an attacker that might want to coerce a voter needs to be able to do some level of packet inspection. There are some scenarios where that might be possible:

- User voting from a compromised WiFi network. The server is somewhere else, thus the attacker only sees some traffic going away. This is the lowest level of resources and traffic will be disguised and encrypted with the current setup.
- A user is voting in an election held in a public institution, such as a university. That voter might be connected to the University's network and the Helios voting server might be also located in that same network. Therefore, the user is effectively performing all the voting procedures in a local network.
- Similar to the university example, a nation-state might control the traffic on all Internet Service Providers. Thus, if the user and the voting server are located in the same country, a correlation attack is also possible.

Timing or correlation attacks [13][6] are only possible if an attacker has full control of the network. That **if** word is very important in these past examples because only if that happens an attack has real chances of succeeding. It is important because achieving that if is very often difficult and impractical.

This work introduces an enhanced version of Helios that helps prevent coercion. It does improve the default Helios model to prevent coercion and it will help users in certain scenarios. Like all software products, given enough time and resources, it will not be enough.

The conclusion, therefore, is that in any electronic voting scenario, it is important to assess what the threat model for each case is. Following that, the election officials and the voters have to make an informed decision and decide if the protections provided by the system are sufficient to prevent a potential attacker. It is only with open and transparent systems that voters can make that assessment correctly and make sure they participate in open and free elections.

Chapter 10

Code availability

The code for this thesis is available as an open-source project on GitHub.

- **Electron app**. `https://github.com/pauarge/helios-desktop`
- **Server**. `https://github.com/pauarge/helios-server/tree/helios-desktop`

The testing server is hosted in Heroku and can be accessed (for a limited time) in:

- `https://helios-server-tor.herokuapp.com`

Bibliography

[1] Ben Adida. "Helios: Web-based Open-Audit Voting". In: (2008). DOI: https://www.usenix.org/legacy/events/sec08/tech/full_papers/adida/adida.pdf.

[2] Josh Benaloh et al. *End-to-end verifiability*. 2014. URL: https://escholarship.org/content/qt7c9994dg/qt7c9994dg_noSplash_97d64dc5a809c552701079250f47b4cb.pdf.

[3] Roger Dingledine, Nick Mathewson, and Paul Syverson. "Tor: The Second-Generation Onion Router". In: (2004). DOI: https://apps.dtic.mil/sti/pdfs/ADA465464.pdf.

[4] Jan Gerlach and Urs Gasser. "Three Case Studies from Switzerland: E-Voting". In: (2009). DOI: https://subs.emis.de/LNI/Proceedings/Proceedings205/21.pdf.

[5] David M. Goldschlag, Michael G. Reed, and Paul F. Syverson. "Hiding Routing Information". In: (1996). DOI: https://www.onion-router.net/Publications/IH-1996.pdf.

[6] Aaron Johnson et al. "Users Get Routed: Traffic Correlation on Tor by Realistic Adversaries". In: (2013). DOI: https://www.ohmygodel.com/publications/usersrouted-ccs13.pdf.

[7] Ari Juels, Dario Catalano, and Markus Jakobsson. "Coercion-Resistant Electronic Elections". In: (2010). DOI: https://eprint.iacr.org/2002/165.pdf.

[8] Sheharbano Khattak, Laurent Simon, and Steven J. Murdoch. "Systemization of Pluggable Transports for Censorship Resistance". In: (2016). DOI: https://arxiv.org/pdf/1412.7448.pdf.

[9] Shuai Li, Huajun Guo, and Nicholas Hopper. "Measuring Information Leakage in Website Fingerprinting Aacks and Defenses". In: (2019). DOI: https://arxiv.org/pdf/1710.06080.pdf.

[10] Ülle Madise and Travi Martens. "E-voting in Estonia 2005. The first practice of country-wide binding internet voting in the world". In: (2006). DOI: https://dl.gi.de/bitstream/handle/20.500.12116/29155/GI-Proceedings-86-1.pdf.

[11] Warwick McNaughton. *Heroku walk-through*. 2021. URL: https://wrmack.github.io/helios-server-docs/site/deploy_heroku_walkthrough/.

[12] Christian Meter et al. "Tor is not enough: Coercion in Remote Electronic Voting Systems". In: (2017). DOI: https://arxiv.org/pdf/1702.02816.pdf.

[13] Milad Nasr, Alireza Bahramali, and Amir Houmansadr. "DeepCorr: Strong Flow Correlation Attacks on Tor Using Deep Learning". In: (2018). DOI: https://arxiv.org/pdf/1808.07285.pdf.

[14] Andriy Panchenko et al. "Website Fingerprinting in Onion Routing Based Anonymization Networks". In: (2011). DOI: https://www.freehaven.net/anonbib/cache/wpes11-panchenko.pdf.

[15] Ida Sofie, Gebhardt Stenerud, and Christian Bull. "When Reality Comes Knocking. Norwegian Experiences with Verifiable Electronic Voting". In: (2012). DOI: https://subs.emis.de/LNI/Proceedings/Proceedings205/21.pdf.

[16] Paul F. Syverson, David M. Goldschlag, and Michael G. Reed. "Anonymous Connections and Onion Routing". In: (1997). DOI: https://apps.dtic.mil/sti/pdfs/ADA465126.pdf.

[17] Tao Wang and Ian Goldberg. "Improved Website Fingerprinting on Tor". In: (2013). DOI: https://www.cypherpunks.ca/~iang/pubs/webfingerprint-wpes.pdf.