Voting system security architecture

Last updated: 2nd November 2009

Overview

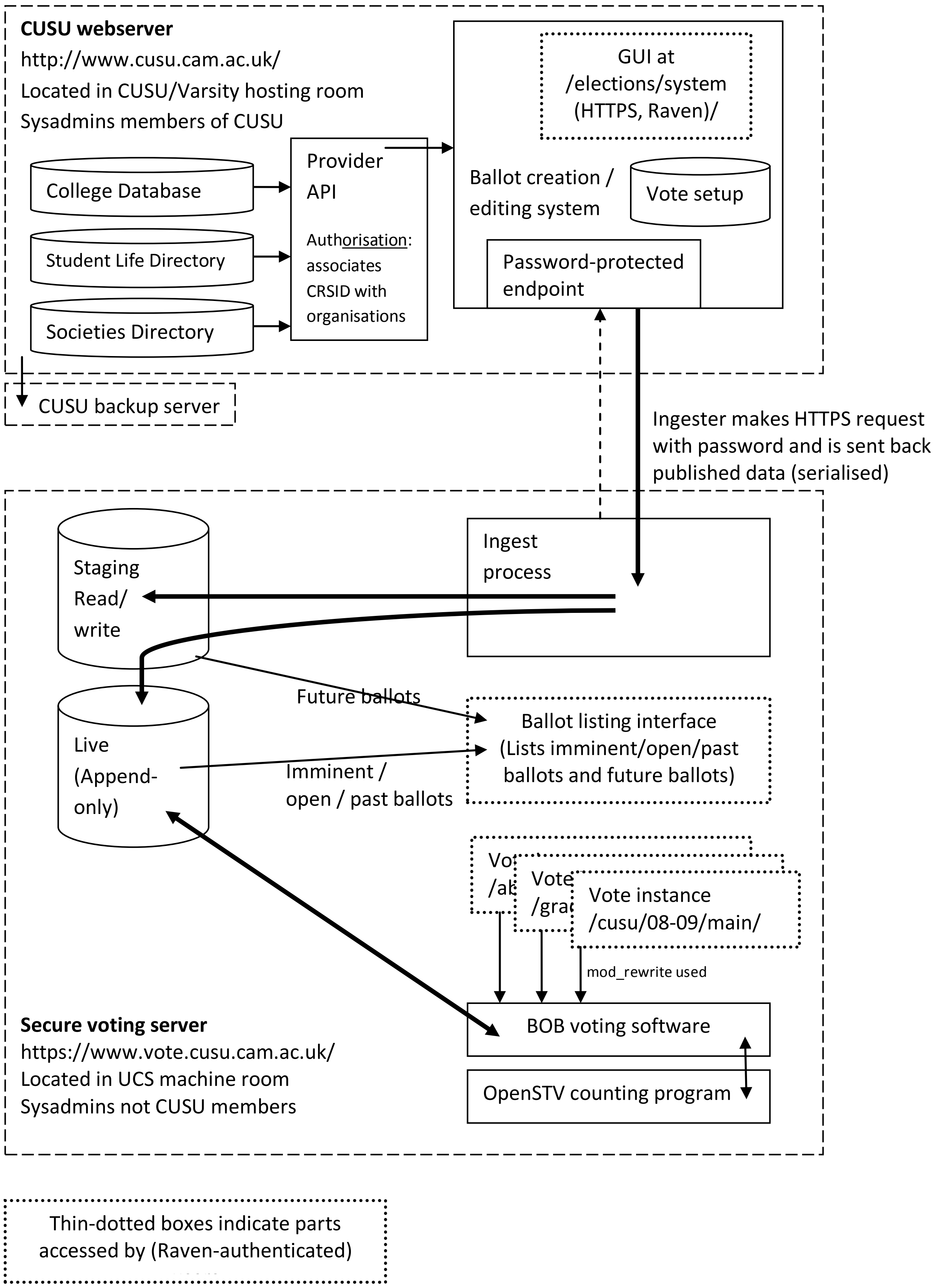
The CUSU voting system consists of four main components, each of which has its own code that is unshared with other components. These are spread across two machines, one in the CUSU offices and one hosted in the University Computing Service machine room. Further details of these two machines are given later in this documentation.

1. **The control panel (ballot setup GUI)**: A web application, hosted on the CUSU website, which provides groups such as JCRs/MCRs, Societies and CUSU itself with a set of webforms with which to set up ballot instances. This is used to set up the parameters of the ballot, such as the contact details for the Returning Officer, the opening and closing times, list of candidates and list of voters.
2. **An ingesting mechanism**, hosted on the high security voting server, which acts as a go-between that pulls ballot instance configurations (including voter lists) from the control panel (component 1) via an encrypted connection and inserts these into the secure voting server (components 3 and 4). It contains logic that prevents running ballots, or ballots about to open, being altered in any way, with the live database rights being append-only, effectively acting as a ratchet system (as explained later in this documentation).
3. **A ballot listing interface**, hosted on the high security voting server, which lists the currently open, forthcoming, and previous ballot instances.
4. **The voting system itself, called BOB**, which contains the voting workflow. It reads the configuration of a vote from the same database as used by the listing interface (component 3). It receives a username from the server (as authenticated by Raven), checks the ability of a user to vote, and allows the user to submit a vote once only, giving them a unique token as confirmation. It also incorporates a means to check that the voter is on the roll in advance of opening of the ballot. At the end of the ballot, the list of votes cast and each associated token is displayed.
5. Additionally, an **STV counting implementation**, using the separate OpenSTV program. This is run by BOB if a subsequent paper vote is not being run.

These components are each described in more detail below.

Diagram of voting system architecture

(Full explanations of the data flows through the system represented by the arrows are explained in the text on the following pages. The bold arrows indicate the most important data flows through the system.)


CUSU webserver
http://www.cusu.cam.ac.uk/
Located in CUSU/Varsity hosting room
Sysadmins members of CUSU
College Database
Student Life Directory
Societies Directory
Provider API
Authorisation: associates CRSID with organisations
GUI at /elections/system (HTTPS, Raven)/
Ballot creation / editing system
Vote setup
Password-protected endpoint
CUSU backup server
Ingester makes HTTPS request with password and is sent back published data (serialised)
Staging Read/ write
Ingest process
Live (Append-only)
Future ballots
Ballot listing interface (Lists imminent/open/past ballots and future ballots)
Imminent / open / past ballots
Vote instance /cusu/08-09/main/
mod_rewrite used
BOB voting software
OpenSTV counting program
Secure voting server
https://www.vote.cusu.cam.ac.uk/
Located in UCS machine room
Sysadmins not CUSU members
Thin-dotted boxes indicate parts accessed by (Raven-authenticated)

Principal risks

The most serious risks which the system is built to guard against are identified as:

1) **Disruption of ballots currently taking place**, principally through deletion or amendment of the ballot configuration or its lists of voters and/or votes. This is primarily enforced through the database-level privileges and then in the application code, as well as through timing checks. Furthermore, the use of two separate machines dealing with the setup and voting sides minimises the ability for any potential insecurity in the setup side (which naturally involves more complex user interactions) to affect the live voting side.
2) **Malicious voting**, which is enforced by logic in the voting workflow code itself, and which is relatively mature code which has previously been used in a standalone context. A series of checks and balances exist to detect the tables containing votes and associated tokens becoming inconsistent (e.g. more/fewer votes cast than people voted).

These risks and prevention strategies are explained in more depth in the descriptions of each component.

1. The control panel (ballot setup GUI)

This is hosted on the CUSU webserver, which is located in the CUSU office machine room. It is run in a lower-security context than the other components, as it is not running or creating the live ballots themselves. Because the later ingesting mechanism acts as a pull from this server and implements logic to deny the taking down of existing votes, any potential insecurity relating to the fundamental aspects of the vote configuration at this initial control panel level will be denied later in the chain, and such an error condition notified to the System Administrator.

The URL of the control panel is: https://www.cusu.cam.ac.uk/elections/system/ . Accessing this via unencrypted HTTP means results in a redirection to the HTTPS side. Mixed content warnings on pages are avoided, by ensuring that no http:// files are loaded. Raven is used for authentication, and is not restricted to specific usernames.

Once authenticated, an authorisation check is then done to see which organisations (e.g. a JCR or Society) they have rights to act on behalf of, if any. The authorisation system uses the Provider API, an internal class on the CUSU website, which acts as code-level bridge between the various directory services, namely the Societies Directory, Student Life Directory and the College Database. Through the Provider API, the control panel receives a list of organisations (together with associated details such as the URL of a logo, name, etc.) to which the authenticated username is registered against. Thus the ballot GUI itself has no internal registry of user rights of its own.

Creation of a ballot is then done via the GUI. This involves a second stage of adding a list of users. In both cases, a series of sanity checks are done to ensure a complete ballot configuration. The ballot configuration sets the user creating the instance as the first Returning Officer (and others can be added, e.g. a College Fellow as used by some JCR elections).

The Returning Officer does not get to see the vote recipients, because this might enable her/him to associate with a voter whom s/he knows is voting at a particular time. (University-wide CUSU elections have previously had such receipts sent to the Returning Officer but this may change for future votes.)

Once a ballot has been created, authorisation to make changes to that ballot is checked against the ballot configuration, and not the Provider API. In other words, by this point changes to the ballot are fixed to the person who created it, and not whoever may now be in charge of the JCR/MCR/Society, etc.

Logic is implemented in the SQL queries retrieving the ballot data when editing to prevent a ballot that is currently in progress being deleted (which in any case would be prevented further up the chain). Furthermore, ballots cannot be deleted or their configuration changed within 2 hours of the opening time. This is to say that, from 2 hours before opening time, the ballot will go ahead and cannot be stopped.

The control panel (ballot setup GUI) publishes its data via a URL end point that the ingesting mechanism (component 2) can read. The URL is also HTTPS and is password-protected. A check is done within the logic of this endpoint to ensure that HTTPS and a REMOTE_USER are supplied, i.e. that the container webserver configuration is correct.

The published array includes a timestamp of seconds since the Unix Epoch. The ingest process uses this and makes a comparison with its own clock (see below).

It publishes all the data about each ballot (both its configuration and voter list) as a serialised array for safe transmission. If there are no ballots an empty serialised array (which is a non-empty string) will be transmitted, i.e. the published data is never an empty string (which could be confused with a 'data not transmitted' state when retrieving).

If the webserver were compromised, the worst that could be done by a malicious user is to delete a vote that has not yet begun, or to set up a new vote under an organisation to which they are not entitled. This is because the ingesting mechanism acts as a ratchet which implements its own logic as to what to accept. Under non-compromised conditions, the GUI will not publish invalid data, but if it did, the ingest mechanism will ignore it after retrieval.

2. The ingesting mechanism

This is hosted on the high security voting server, to which only the two System Administrators have shell-level access. Two databases, a staging database and a live database exist here. The live database has append-only rights for the ingest user, and the staging database has wider rights. (These rights are enumerated as shown towards the end of this document.)

The ingest is launched as a shell (i.e. non-HTTP context) process, running under its own username, at around 30 minutes past each hour (a time checked by the script). The ingest process writes a lock file at the start of its process. The ingest process checks that a lock file is not present before beginning work.

Each stage of the ingest mechanism is logged with a checkpoint marker in a log file for auditing purposes.

If an error occurs, the System Administrator is notified by e-mail (with a customised error case written for each part of the script), and the lock file is left in place.

The ingest process opens a connection to the staging database and raises an error condition if that fails.

The ingest then makes an HTTPS request including a password to the endpoint published by the setup GUI. It checks for receipt of a non-empty string and so will raise an error condition if that does not happen.

Having unpacked the string, the timestamp is compared and, allowing for a small discrepancy (e.g. due to delays in the HTTPS transfer), an error is raised if the timestamps of the two machines are out-of-step. Each check involving something time-related is immediately preceded by a check that the current time is not 5 minutes greater than the start time of the whole script.

It then loops through each supplied instance and ascertains that the data structure is as expected and non-empty.

It then reads a list of live instances on the second, 'live' database. Each instance has a unique key. A check is then done to ensure that the supplied new data does not contain the key of an existing live instance, because the published data for ingesting should only be future data. (However, a later attempt to re-insert such an entry would be denied anyway due to the database privileges.)

The staging database is then emptied, and the received data is then put into the emptied staging database.

Having now ingested the published data from the GUI, the second stage of the ingest process begins. This is to move the data in the staging database that represents a future ballot into the live database. The live database rights are append-only, so existing data cannot be amended or deleted.

The placing of an instance on the live side is done just after 90 minutes (i.e. the 2nd on-the-half-hour process) before a vote opens and is then fixed from that point. The use of two on-the-half-hour runs before the vote opening rather than a single run avoids the potential for timing issues to arise at the change to/from Daylight Saving Time twice a year.

In summary, the ingest mechanism has thus obtained the data from the GUI side, put this into a staging database then implemented a ratchet-like mechanism to add imminent votes to the live database. Those votes are then available for the following two components to read (and in the case of the voting component itself, append to).

3. Ballot listing interface

This is also run on the high-security side. It makes a database connection which has only the rights to read the live and staging databases and nothing else.

It is implemented as a class which has a straightforward set of logic that reads the live database's list of instances and lists them in various ways. It then provides links through to the URL of where each vote instance (component 4) is expected to be.

It also reads the staging database to merge into the listings in-future votes that have not yet started. This is because a key part of voting system as a whole is for voters to be able to check in advance they are on the electoral roll.

As part of the usability efforts made on the system, the ballot listing page URLs (e.g. /forthcoming.html) use mod_rewrite to direct all requests to a stub launching file that then runs the listing class code itself. The class implements the Front Controller pattern.

4. The voting system itself, called BOB

The voting workflow and GUI itself is implemented in a program called BOB. This is a separate assembly to other parts of the system, i.e. the voting workflow is not linked to instance listing/importing.

This is implemented as a class with a launching stub file. The stub file contains just enough information to bootstrap itself, e.g. making the database connection. It is installed with a configuration file containing the name of the table where the instance configurations are listed.

The BOB class also uses the Front Controller pattern, and the code is visible to a voter should they wish to inspect it via a self-referencing mechanism which displays the source when requested via a suitable URL. It is also downloadable freely from http://www.cl.cam.ac.uk/~dme26/proj/BOB/ .

The vote URLs are in the form of /organisation/year-year/moniker/, e.g. https://www.vote.cusu.cam.ac.uk/cusu/08-09/main/ , and the action is set via a query string component, e.g. https://www.vote.cusu.cam.ac.uk/cusu/08-09/main/?results which is sanitised through the Front Controller mechanism.

There is only one installation of BOB on the server. Although it might be considered better to have separate files for each instance, this infers the need for the ingesting process to have file write access to the live htdocs area, which would increase the attack surface. To work around this, therefore, each vote instance URL is run from the central installation via a mod_rewrite process that results in the configuration file being given an ID, e.g. cusu-08-09-main to take the above example. This is then checked within BOB when reading the instances table. BOB itself echoes the configuration of the vote in its comments. BOB instances do not interact with each other.

The head of the front controller within BOB introduces a large set of sanity-checks (which were added mainly to flag misconfigurations when run in a standalone context). This includes:

- That the requested instance exists;
- That the supplied configuration is valid;
- That HTTPS is in use;
- That a username is being supplied by the webserver (using container authentication)
- That the PHP environment is secured, in particular that deprecated configuration such as register_globals, safe_mode, etc. are not in use;
- That the auto_prepend_file PHP feature is not in use, to ensure that additional code is not injected prior to the loading of the class;
- That errors are set to be logged and sensitive details such as file paths are not leaked on-screen;
- That the PHP environment provides sufficient memory;
- That a database connection can be made;
- That the runtime database privileges are neither too few or too many;
- That there are voters registered;

- Checks on the time to set the current status of the vote;
- That no-one has been marked as having voted and that the number of votes cast is zero if the time is before the scheduled opening of the ballot;
- That the number of people who have been marked as having voted is the same as the number of votes cast;
- That the requested URL action is a registered page within the Front Controller.

The voting workflow is then run.

The header/footer files which prettify the ballot listing (as well as ensuring a reasonably large font is used to lessen voter error) are loaded as static strings (i.e. are not parsed for PHP) and added in for usability purposes. Note there is no connection to the listings GUI itself (which also loads these same header/footer files for consistency).

Three counts exist which can be used to confirm a non-tampered vote. Firstly, the list of voters is displayed after the vote, and the total of these can be summed. Secondly, the tokens associated with each vote are listed, which also can be summed. Lastly, an e-mail receipt for each ballot cast is sent to the user and blind-carbon-copied to a special Hermes mailbox; again the total of those e-mails can be summed. The use of a Hermes mailbox has the additional benefit that the mail to it could potentially be audited by requesting the Hermes administrators to examine the log files. The three totals should match, as has been the case with all votes run by CUSU using the BOB program in previous years.

5. STV count implementation, using OpenSTV

A counting facility has been integrated into BOB. This runs OpenSTV (written in Python) which is installed on the server. This is launched using the proc_* family of commands in PHP rather than using exec() or similar commands (which require much more error-prone escaping code).

External security aspects

The GUI on the CUSU webserver is considered to need a lower security context. Accordingly it is considered appropriate to run this component within the CUSU website (which also runs a myriad of other services), both for usability reasons and to enable direct access to the Provider API directory service mentioned above. The only potential class of vulnerability would be the creation of ballot instances for users who do not have authorisation within the organisation concerned to create them. Disruption of in-play ballots is guarded against, twice, further up the chain.

The hosting of the GUI, which involves more substantial amounts of code (including resulting of writing of data by ordinary users) than the listing side, on a separate machine to the voting server, means that the potential for leakage across systems is eliminated.

This CUSU webserver is hosted within the offices in a locked area and is rack-mounted, with a UPS. This room is shared by CUSU and Varsity. Backups are made periodically and the SysAdmin team patch the server regularly.

The voting server clearly requires a much higher security context, which therefore includes physical security. It is hosted in the University Computing Service machine room, which is the location for

UCS services such as Hermes, the PWF, etc. It is considered the highest physical security environment available within the University for hosting of the machine.

The IP of the machine is marked in the IP Register database as requiring special clearance to make changes. That is to say, that it cannot be changed by the CUSU Sysadmins, which are likely to be members of CUSU.

There is not currently a routine backup made of the voting server. This is an aspect that needs to be addressed, and funding would need to be found.

The voting server system software is kept patched by one of the System Administrators.

Personnel

The voting server is currently run by two Computer Officers within the University. Both are salaried members of the Regent House and neither are students/members of CUSU and so are considered to be disinterested in the outcome of University-wide student votes.

The main system administrator is Bob Dowling, Head of Unix Systems Division, who has kindly allowed the machine to be hosted by UCS and has donated his efforts in the provision of system administration. The other Administrator is Martin Lucas-Smith, who is also a Computer Officer at the Department of Geography (his work for CUSU is unrelated to that position).

If the system were to be extended to involve Regent House voting, the issue of personnel would need consideration (alongside other issues) as the System Administrators perform a trust role analogous to that of a Returning Officer in a traditional paper vote, although there are additional safeguards in place that increase the likelihood of a compromise in the vote being recognised.

Required privileges on the high-security voting server

Ingest user (ingests votes from www.cusu website into staging database then pushes live):

staging:	SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, ALTER
votes:	SELECT, INSERT, CREATE

Listing user (lists current, past and future votes):

staging:	SELECT
votes:	SELECT

Vote setup user (used by the BOB voting system; privileges are lower than would be used in a standalone context because the ingest process is doing the setup in this installation):

staging:	SELECT, INSERT, UPDATE
votes:	SELECT, INSERT, UPDATE

Vote runtime user (used by the BOB voting system):

staging:	SELECT, INSERT, UPDATE
votes:	SELECT, INSERT, UPDATE