

## **An emitted-code model of configuration control and practical audit of e-voting software**

Everyone Counts Inc  
www.everyonecounts.com  
contact@everyonecounts.com

19 July 2006

### **Abstract**

We report on the ongoing use of a evoting architecture which employs a Java translator to produce a free-standing balloting application for single use. The system accepts a formal description of election parameters (candidates, formality rules, instructions, etc) and creates Java source codes and comments of about 4000 lines. The codes can be exported and made public, but are intended to be code-audited with the auditor compiling and signing the applet on a clean system. The evoting application uses a PKI approach to protect collected votes so that supporting systems do not require code auditing. The reported design has formed the basis of a binding public election and several private elections. Our ongoing work involves reducing the code size of the ballot application and deploying a parallel testing framework.

### **Introduction**

Evoting machines deployed in Ireland and the United States have received damning criticism primarily because they automate the voting process in a manner that makes traditional observation impossible. That is, traditional election transparency is not possible. Indeed, all forms of automation rendering a physical process in to an electronic one suffer this kind of compromise and the rise of legal action over the non-deniability of software flaws can be seen in other public systems law suits such as those concerning motorway speed camera software [1].

Open Source software has been proposed as a new basis from which to provide more transparent voting systems. However, while this does by virtue of the GPL and similar licenses also provide software sources for all to see, there are numerous other problems which reduce our confidence that publicly observable software sources are the basis of the voting application we want to trust. They include :

1. Is the live voting application built from the published software sources?
2. Were the compiler or development tools used to build from the sources free from errors or malicious software - 'malware' ?
3. Can we in fact audit the software to the degree of detail necessary to be sure that it is significantly free of errors or malware?
4. Does live voting application obtain any election parameters from external files or external sources? If so, can we be sure those parameters or outside sources will not affect the voting process?
5. Can the evoting application software be modified in transit or in situ so that a cleanly compiled

binary can be compromised after an in situ audit?

6. How can the software be practically maintained and improved and yet retain a software certification?

The above are the confounding questions which make the certification of voting systems (both hardware and software) a very difficult task. Additionally, for closed-source systems, exposing proprietary sources is a competitive risk for suppliers. Due to these significant hurdles, and considering that the United States Voting Systems Standards 2002[2] required updating, the current US standards are voluntary[3] and the Electoral Assistance Commission has now requested voting systems providers to surrender software for source control by NIST[4]. While this is certainly a step in the right direction, it perhaps only addresses point 1 of our questions above, and only if both sources and production applications are submitted to NIST and sources can be compiled to the production applications.

The subject of this report is a more incremental approach to providing an evoting application which has been designed to address at least some of the above questions and so attempt to increase voter confidence in the correct handling of their votes. It is important to assert here that while there has also been progress in cryptographic protocols for voting, the above problems are simple and profound : source control, independent audit, practical audit, configuration control and open inspection are the foundation of a transparent system. In addition, incremental approaches to dealing with security are becoming more common as the complexity of software increase and prevalence of zero-day threats grows.

The system and software auditing process in this report formed the basis of a binding UK election [5,6] and has served a number of private elections. The software has also been tested as part of a distributed (P2P) voting system [7].

### **An (abridged) threat model for electronic voting**

There are a wide range of attacks which may seek to take down an evoting service, replace it, expose voter identity, discover voting results early, or affect official results. We will focus on attacks that seek to transparently alter votes and so achieve a pre-determined outcome, but with no trace of such an effect. We determine this to be the more pernicious attack because it breaks the democratic process and may never be detected. There are several stages in the software life cycle where evoting software can be changed, damaged or completely replaced with a fraudulent application. They include :

1. An attacker among the developers introduces malware in to the voting software as it is written
2. An attacker modifies the software compiler the developers use so that while the source may be clean, the compiled application is not
3. An attacker modifies or replaces the software once it is installed on the voting machine
4. An attacker modifies files that the software needs so that the software is clean but the parameters it is given cause it to malfunction, for example, causing it to misrepresent candidates
5. The software is modified or replaced by an attacker in the maintenance cycle

For now we ignore the supporting platform the evoting application runs on.

We first introduce our system and then attempt to show how it defends the voting application against the above attacks, then what measures are needed to strengthen it against several of the six “hard” questions at the top of this article. We introduce a planned parallel testing method which will help us assess the overall health of the evoting platforms (including supporting software and hardware) and we make some suggestions about how the evoting platform may be simplified to remove opportunities for technical fraud.

## Auditing

A software audit is an exercise which seeks to exhaustively confirm that a software specimen conforms to its specification. Additionally, specified security mechanisms within the software can be tested for their conformance to information security principles and standards such as the Common Criteria. This is in contrast to external tests of the software when it is running. External tests can measure conformance of the application to its functional specification, but only in as much as the application meets the functional specification for a set of specific tests. Code-level auditing is preferable to external testing because all possible test cases can effectively be tried via methods of program proving, for example.

The greatest obstacle to any kind of audit is software complexity. External testing is confounded by software which has many different internal states or different combinations of input. These states require impractically large numbers of tests, possibly tens of thousands. Exhaustive code level software auditing is confounded by the volume of an application's source codes and the manner in which the codes have been written.

It becomes prohibitively difficult to fully audit software exceeding a few thousand lines in length. Indeed there are popular competitions to demonstrate how software can be written to obscure additional unexpected actions it will take when executed. However, when software is clearly written, self-contained and not large, it becomes difficult to “hide” additional routines from a human assessor. If in doubt over the purpose of a section of the software, the assessor can fail the audit and request that code be rewritten.

The current report introduces an evoting software design where the evoting software is small and concise enough to be practically audited. This means that the audit task can be carried out exhaustively by one- or few- people in a few days. So far, the level of auditing possible at this stage is NATA[8] standard software testing against a specification (for example, in Australia, CSC.com has proposed our work can be tested in this way as it qualifies as a NATA test facility[9]). We plan to simplify the software further and refine the specification so that we can seek a CC Assurance Level. Prior to this, auditing has been provided by an open source software consultancy.

Originally started as a code optimization solution for high-volume server transactions, the authors developed a Java translator which produces an evoting application that is a voting *client* intended to be run on evoting or PC hardware. All privacy and security actions the evoting software takes are encapsulated in this Java application so that supporting software does not have to transport or store sensitive information in any vulnerable format. Instead, the emitted Java application is the only part of the system which can make or modify a vote. The remainder of the system can effectively be shut down, opened to inspection or be replaced with a third-party ballot collection service.

An *emitted* Java application is thus created for each and every election. The Java source codes embed the election candidates or referendum question, candidate images and names and formality rules for determining if the ballot is correctly completed for the legislation it enacts. The source codes are 4000 lines long including all cryptographic routines, any number of ballot page layouts, comments and navigation. The software is kept short in length because it does not need to provide generic routines for all election styles. These are built in as needed.

The application takes no external configuration parameters. It is provided as a single-use piece of software and is intended for archive with the details of the specific election it is created for.

The Java application takes part in an election as follows:

1. The Electoral Returning Officer (ERO) uses a server-based web application to enter the details

of an election : times, candidates ballot styles, graphics and so on. This facility intended for a non-programmer.

2. The web application provides a downloadable tool intended for a “clean” PC (as opposed to the web application which is mounted on a shared web server). This downloadable tool facilitates the creation of PKI key pairs and random, unique voter access credentials.
3. The web system creates a formal ballot description as a text document. This document can also be manually edited. This document defines exactly what the ballot will do, based on what the ERO has provided. The web system can produce drafts of the ballots for logic and accuracy testing and for sign off.
4. A translator service builds Java sources from the formal ballot description. The ERO downloads the Java sources.
5. The ERO sends the Java sources to a software auditor.
6. The Auditor takes about 4 days to audit the software. The ERO or the auditor may provide the Java sources for public or academic examination.
7. The Auditor compiles the sources on a “clean” compiler on a trusted system. The Auditor signs the compiled Java application with a digital certificate.
8. The Auditor returns the compiled, signed application to the ERO.
9. The ERO distributes the application for execution of voting machines or for access to remote voters via the web. The ERO distributes voter access credentials to voters or to polling places.
10. The Java application renders the correct voter-jurisdiction ballots based on voter credentials, encrypts the vote with a 3DES symmetric key, then encrypts the 3DES key with a common RSA public key embedded in the applet. An HMAC is created to sign the encrypted vote. The applet returns this encrypted file to a central web server or the encrypted file is recorded elsewhere for manual transport to a counting facility.
11. The ERO and observers who have access to the private key for the election decrypt votes and count them.
12. A optional receipting system allows the voters to determine if a vote from them was received and decrypted correctly. The receipting system does not expose a voter's vote not their identity.
13. The Java application, sources and other information are archived indefinitely.

### **What do the emitted code sources expose?**

The Java evoting application created for one specific election exposes the Java commands which are intended to offer ballots, collect votes, warn the voter of incorrectly filled ballots, issue a receipt and encrypt the vote. It is important to note that this does not expose a means by which to easily decrypt voter's votes. The strength of the encryption is derived from the secrecy of the RSA secret key and the correct management of voter access credentials. Exposing the evoting software is equivalent to exposing the mechanism of a common door lock, knowledge of its function does not make it easier to open other similar locks without a key.

Exposing the sources brings the following benefits:

1. The exposed sources can be compiled by any person who downloads them and the correct version of the Java compiler to confirm that the election applet was exactly derived from those sources
2. The Auditor can digitally sign the sources as well to provide greater trust that public accessible sources are also those that were audited
3. The exposed sources can be examined by experts other than the Auditor to affirm or deny the Auditor's assurances
4. The single-use Java application and sources can be kept as evidence with the election results for future examination.

We admit that exposing the sources of the voting application does provide more information for an attacker. We argue, like open source itself, that this disclosure brings benefits which outweigh the

risks. While the exposure of the mechanism of the applet gives an attacker a “lead” to affect a wider range of attacks on the voting platform itself, an attacker must still collect the voter access credentials and other secret information the applet collects in order to create a valid vote. The attacker must replace the voter's vote with the valid, fraudulent vote.

These attacks may fall in to three classes:

1. The attacker replaces the Java execution environment with a facsimile. This new environment interprets the commands the evoting application provides in pre-determined way so that voter secrets can be harvested and the final submission of the legitimate vote can be intercepted.
2. The attacker installs a trojan that acts on the evoting application and substitutes parts of it for other compiled Java routines. When the evoting application runs on the default Java execution environment, it performs new actions to affect the vote. This is a trojan attack which replaces code in the applet before execution. We call this a software modification attack.
3. The attacker turns off the cryptographic function of the applet so that the resulting vote can be intercepted, modified, encrypted and recorded as intended.

The signed applet cannot be modified in transit as its digital signature can be checked on installation to confirm it is the applet the Auditor signed and that it has not been modified in any way. The published sources vary from election to election as maintenance requires and they are published after audit, which takes place after close of withdrawal of nominations. This leaves a narrow window for an attacker to engineer a specific attack and reduces the likelihood that a software modification attack could be crafted that worked without failing visibly at least once.

Any attack on the evoting system has to not be detected, even once. This in itself is a tall order for a hacker given that the attack is just another form of software development. Unfortunately, it is easy to evade detection if any testing is carried out which differs in some way to live voting. During live voting, voters may not notice or report some kinds of malfunction.

Outside the live election period malware can perform differently based on the current date (and so remain dormant until a live election), based on features of tests which identify them only as tests (for example, voter access credentials which are shorter than live credentials or contain a marker) or even detect the volume and rate of voting being enacted (a test machine might enjoy a short battery of quickly-executed tests while a live system processes an irregular stream of voters). Malware may also only act on every 10<sup>th</sup> or 50<sup>th</sup> ballot and still change the outcome of the election. These considerations require that evoting systems be tested in parallel with live voting machines during the live election.

### **A parallel testing model**

A design for parallel testing which is being developed by the authors for remote voting allows the remote voter to telephone an automated service and then request their vote be canceled. The voter can then voluntarily provide some or all of their voting choices to the telephone system. The canceled vote is retrieved based on the voter's correct citation of a sequence of bytes from the encrypted body of the vote before submission along with part of their voter access credentials. Since remote voters will use their computers for only a small number of votes than a polling place machine, any trojan would be programmed to act on more votes than an evoting machine trojan and so we expect the number of purposefully false reports not to exceed the number of cancellations which in turn expose the intervention of trojans.

### **How does the system answer the “hard” questions?**

*1. Is the live voting application built from the published software sources?*

This can be confirmed at least to installation by compiling the published sources with the right

compiler to render the same binary application. Secondly, checking the source and binary signatures confirms they are authentic and unchanged.

*2. Were the compiler or development tools used to build from the sources free from errors or malicious software - 'malware' ?*

The chosen software auditor (or in fact, auditors) can assert this by defining how they maintain a clean compiler on a clean system. This can be also be tested by the general public compiling the sources an several instances of the compiler and then reporting any deviations in the output binary.

*3. Can we in fact audit the software to the degree of detail necessary to be sure that it is significantly free of errors or malware?*

This can be determined by the scope of the task at hand, the budget for audit and the skills of chosen auditors. The emitted evoting software is short, concise and documented. It is difficult to infect the sources with a process to affect voting. This is difficult because a human auditor seeks conformance of all software routines within the application and the human auditor may well have some expertise in the motivations of a hacker and so further constrain how the evoting software could be changed.

*4. Does live voting application obtain any election parameters from external files or external sources? If so, can we be sure those parameters or outside sources will not affect the voting process?*

The evoting software is self contained. The supporting systems are however, relied upon for correct execution of the evoting software and correct carriage and storage of encrypted votes. We propose a parallel testing technique and a receipting technique to at least allow a random sample of voting integrity.

*5. Can the evoting application software be modified in transit or in situ so that a cleanly compiled binary can be compromised after an in situ audit?*

Again, the system is protected by its digital signature. Of course it has to also be practical to confirm that the voting software signature is valid. In addition, the single-use nature of this software model means that software will not be maintained on evoting machines, but instead will be entirely replaced for each election. This reduces the scope for attacks against the evoting software while evoting machines may not be in use. Finally, because the evoting application can be entirely replaced, we would advocate a clean install of all supporting software from controlled and signed sources as well.

*6. How can the software be practically maintained and improved and yet retain a software certification?*

The software is audited for each and every election. It can and should change to remain current and an extent, to remain as a moving target rather than a static one.

## **Conclusion**

We have discussed how the task of auditing an evoting system may be made more practical by the provision of single-use software that is created for each specific election and so does not include large, complex, generic components. We have listed what we believe to be difficult questions for software control in evoting systems and we have listed a number of attacks which we anticipate in the software life cycle. We introduce a voting application and a parallel testing development to strengthen the integrity of the system, at least for remote voting elections.

We have argued that short (4000 lines) software which is written to execute a single election will be within the scope of a practical (few-person, few-days) exhaustive code-level audit. This then would augment or replace a traditional entire-system audit as defined by the updated US VSS standard and other standards for electronic voting systems. The approach is an incremental audit of reduced

scope as compared to a “one-off” audit intended to cover an entire, generic election system. This incremental approach falls within a larger trend in security systems to provide small, incremental updates to better cope with the speed and sophistication of attacks.

We contend that exposing the emitted software source codes to public scrutiny does give more information to attackers but that the benefits outweigh the risks. Those benefits being that correct auditing, compilation and analysis can be confirmed by outside experts.

Our new work will see the further reduction and simplification of the emitted voting software and the use of the parallel testing system in a binding election.

*The authors welcome comments and feedback.*

## References

- [1] The Age (2004), “\$26m speed payout” cited at 2 June 2006  
<http://www.theage.com.au/articles/2004/05/14/1084289885456.html>
- [2] “2002 Voting Systems Standards (as developed by the Federal Election Commission)” cited at 2 June 2006 [http://www.eac.gov/election\\_resources/vss.html](http://www.eac.gov/election_resources/vss.html)
- [3] “Voluntary Voting Systems Guideline Version 1 (VVSG 1) - Draft Materials & Versions” cited at 2 June [http://vote.nist.gov/vvsg1\\_drafts.htm](http://vote.nist.gov/vvsg1_drafts.htm)
- [4] Zetter, K (2004) “E-Vote Vendors Hand Over Software”, Wired Magazine, October 26, cited at 2 June <http://www.wired.com/news/evote/0,2645,65490,00.html>
- [5] The United Kingdom Electoral Commission “Pilot scheme evaluation Stratford on Avon District Council 1 May 2003 Part A” cited at 2 June 2006  
<http://www.electoralcommission.org.uk/templates/search/document.cfm/8267>
- [6] Burton, C.A (2004) “A Virtual Private Network for Internet Voting”, available permanently from <http://www.everyonecounts.com/downloads/Virtual-Private-Network.PDF>
- [7] Burton, C.A, Karunasekera, S., Harwood, A.J., Stanley, D., Ioannou, I. (2005) A distributed network architecture for robust Internet Voting Systems, in Wimmer et al eds., Electronic Government, Proceedings of the 4th International Conference, EGOV 2005, Copenhagen Denmark 21-23 August, Springer LNCS 3591, ISSN 0302-9743
- [8] National Association of Testing Authorities <http://www.nata.asn.au/>
- [9] CIO Government (2004) “Software accreditation to benefit govt departments” CIO Government Magazine 26 October  
<http://www.cio.com.au/index.php/id;1706391845;fp;4;fpid;21>