Hack-a-Vote: Demonstrating Security Issues with Electronic Voting Systems

Jonathan Bannet
jbannet@cs.rice.edu

David W. Price
dwp@alumni.rice.edu

Algis Rudys
arudys@cs.rice.edu

Justin Singer
jsinger@alumni.rice.edu

Dan S. Wallach
dwallach@cs.rice.edu

Department of Computer Science
Rice University
Houston, TX

Abstract

A representative democracy depends on a universally trusted voting system for the election of representatives; voters need to believe that their votes count, and all parties need to be convinced that the winner and loser of the election were declared legitimately. Direct recording electronic (DRE) voting systems are increasingly being deployed to fill this role. Unfortunately, doubts have been raised as to the trustworthiness of these systems. This article presents a research voting system and associated class project which was used to demonstrate several classes of bugs that might occur in such a voting system unbeknownst to voters, with the difficulty of detecting these bugs through auditing. The intent of this project is to justify the mistrust sometimes placed in DRE voting systems that lack a voter-verifiable audit trail.

1 Introduction

A fair, universally accessible voting system by which all citizens can easily and accurately cast a vote is central to the democratic process. The United States presidential election in 2000 provided a firsthand, very public look at what can happen when the voting system is flawed. Following that election, there was renewed public interest in reliable voting systems. People became enchanted by the computer's siren song, and voting systems in which computers play a significant role in recording and tabulating votes saw increased interest and adoption. Much of this attention has been focused on *direct recording electronic* (DRE) voting systems, in which paper ballots are eliminated from the voting process entirely.

DRE voting systems have some inherent advantages over paper-based voting schemes. Through a well-designed interface, DRE voting systems can help to eliminate voter error. For example, the voting system might seek confirmation if the voter

fails to cast a vote in a particular race, and the voting system can disallow multiple votes in the same race. In addition, voting systems can allow people with a number of disabilities to vote without needing human assistance.

Unfortunately, recent analyses of one popular vendor's DRE voting system have indicated numerous security oversights [17, 16]. Voters can cast multiple votes without leaving any trace. Voters, as well as voting staff or others with access to the machine, can perform actions that normally require administrative privileges, including viewing partial results and terminating the election early. Furthermore, communications between voting terminals and the central server are not properly encrypted, allowing a malicious *man in the middle* to alter the content of the communications. The authors found that an undisciplined approach to the development of this software, along with a process that did not encourage anticipating or fixing security holes, led to this flawed state. A closed-source approach to software development, in which the source code is not released to the public for review and comment, only served to delay the necessary scrutiny.

As daily headlines demonstrate, neither a commitment to software security nor an open source approach to software development prevents software security holes from arising. Microsoft's significant "Trustworthy Computing" initiative, announced at the beginning of 2002, has not stemmed the flow of security holes being found, including holes exploited by high-profile worms such as Sobig [10] and Blaster [6]. Likewise, open source projects have experienced their own share of vulnerabilities [3, 7, 8]. Software developers and auditors, following normal industry software engineering practices, have proven unable to ship truly bug-free software. In general, producing software free from *all* security holes is a significantly harder task than an attacker's goal: to find and exploit a single bug.

This article presents Hack-a-Vote, a simplified research DRE voting system implementation initially developed to demonstrate the ease with which a Trojan horse could be implemented in a voting system. We also discuss an associated course project, in which students implemented their own Trojan horses in this system and then audited the source code of their peers to find these Trojan horses. This article explores the potential damage that could be caused by malicious individuals targeting voting machines, the feasibility of finding weaknesses (deliberate or otherwise), and some solutions to mitigate the damage. Hack-a-Vote and the course assignment using it are freely available online.[1]

We describe the design and implementation of Hack-a-Vote in detail in Section 2 and the logistics for the associated course project in Section 3. Section 4 describes the malicious hacks that the students implemented and the auditing they did to detect these hacks. In Section 5, we discuss these results in terms of real-world DRE implementations like that of Diebold. Finally, Section 6 presents our conclusions.

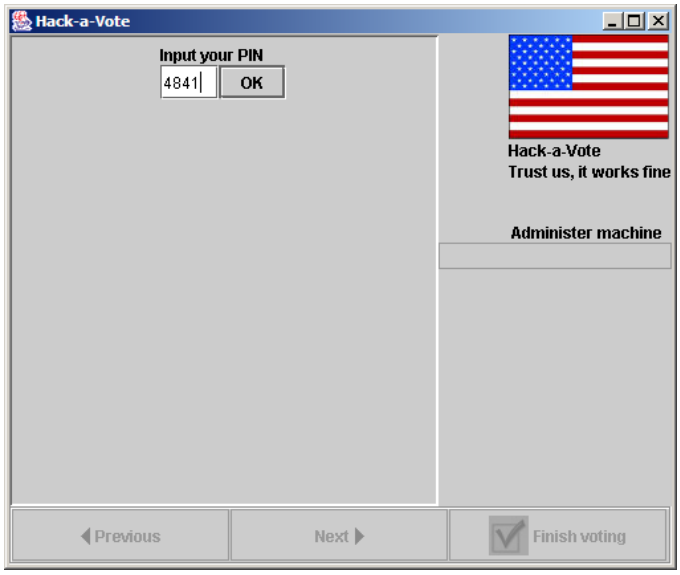

Figure 1: Hack-a-Vote PIN entry screen

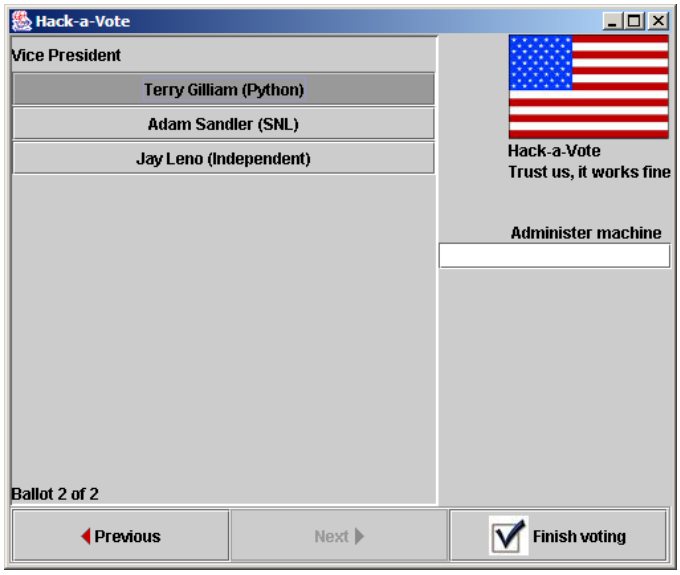

Figure 2: Hack-a-Vote selection screen

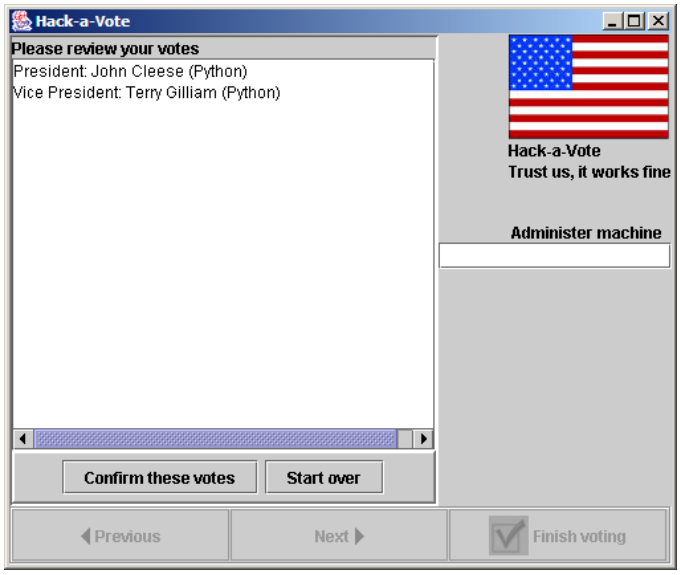

Figure 3: Hack-a-Vote confirmation screen

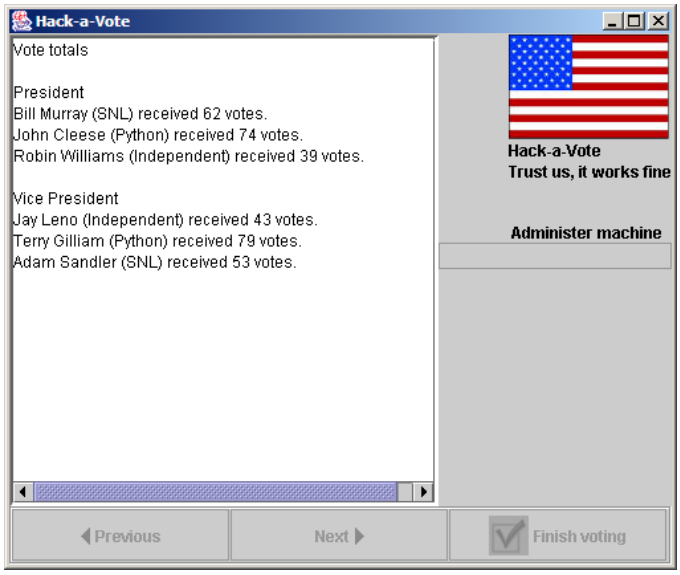

Figure 4: Hack-a-Vote final tally screen

2 The Hack-a-Vote System

Our demonstration voting machine is a relatively simple Java Swing application, weighing in at about 2,000 lines of code. It has a GUI front-end for interaction with the user, a back-end for reading in election configurations and writing out ballot images, and voting logic that drives the GUI and records votes. The GUI is shown in Figures 1 through 4.

When Hack-a-Vote is started, it first authenticates the user with a simple PIN authentication scheme inspired by Hart Intercivic's[2] eSlate, which is used for elections in Houston, Texas, among other places. With the eSlate system, voters first sign in to their local voting precinct. Every group of eSlate voting terminals is connected via a wired network to an election management console. The voter will approach the console and an election official will print a 4-digit PIN. The voter then goes to any available voting terminal and enters the PIN. The PIN is validated using the network and the voter can then make their voting selections. Similarly, in the Hack-a-Vote implementation, the server maintains a list of a few valid PINs which are displayed to the election administrator (we didn't bother to print the PIN, although it would be an easy extension to add). When a PIN is used, it is invalidated and a new PIN is randomly generated. The PIN login screen is shown in Figure 1.

After authentication, the user is presented with a series of races. The voting screen is shown in Figure 2. Once a vote has been cast in every race, there is an opportunity to view and confirm a summary of the votes cast (see Figure 3). If the voter confirms the selected candidates, the machine cycles back to the PIN entry screen for the next voter. Otherwise, the machine returns to the first ballot and allows the voter to change her selections. Once the election is closed, all of the votes are randomly shuffled, written out to disk, and the final tally is displayed (see Figure 4).

In the original implementation of Hack-a-Vote, a Trojan horse was added to the code that prepares votes for writing out to disk. We added a few extra command line options to allow the user to indicate a candidate or political party that should receive extra votes. With a configurable probability, votes for other candidates are modified before being written out to disk and counted. We removed the cheating code to produce the version that the students used — this involved removing only about 150 lines of code from one file.

3 The Hack-a-Vote Assignment

The Hack-a-Vote project served to introduce students to some of the many faces of security, from the malicious hacker's design of subtle attacks on a system, to the auditing required to uncover those attacks, to the higher level challenges of designing a secure system. The project also raises policy implications for how governments might regulate voting systems. The Hack-a-Vote project was inspired in part by Ross Anderson's

[1] `http://www.cs.rice.edu/~dwallach/courses/comp527_f2003/voteproject.html`

[2] See `http://www.hartintercivic.com/` for more information.

UK lottery experiment, in which students were asked to consider the security threats of running a national lottery [1].

Students in Comp527, a graduate level course in computer security, were split into groups of two for a three part assignment. For the first phase of the project, groups took on the role of developers at the Hack-a-Vote corporation, paid off by some nefarious organization to rig an election without getting caught. No requirements were placed on the kinds of malicious hacks requested by the organization, so long as the students could justify how such a hack could be used to ruin or bias an election without being detected. Solutions could run the gamut from denial of service attacks to arbitrary biases in the final election tally. The groups spent two weeks peppering their voting machines with Trojans horses, their stated goal being to hide discreet yet potent hacks.

Upon completion of the first phase, each group received modified voting systems from two other, randomly selected groups in the same class. They now took on the role of independent auditors with one week to find any security-critical flaws within the altered code. During the second phase, any direct comparison of the code under audit with the original code was prohibited. Students were required to rely on only their memory of the voting machine and its functionality, a requirement enforced by the university's honor code. (Otherwise, any changes could easily be located using the UNIX `diff` tool, giving the students greater powers than a genuine auditor might have when studying a voting system.)

The focus shifted for the final phase of the voting project. Students had two and a half weeks to fix a problem discovered by Kohno et al. in the smartcard authentication protocol used in Diebold voting machines [17]. Students first had to consider what security threats a voting system faces. They then designed an improved authentication protocol and used Cryptyc [15], a cryptographic protocol type checker, to model and verify this protocol. They also had to explain the protocol in terms of the security threats it was designed to thwart.

4 Results

Although the hacks submitted by the groups were quite varied, they generally fell into one of several categories. The most blatant were direct vote-manipulation hacks. A second class of attacks broke the authentication mechanism either for PINs or for administrative access. A third class of hacks were directed at defeating voter anonymity. Finally, a number of projects implemented denial-of-service bugs.

4.1 Result Manipulation

The most blatant hacks involved direct manipulation of vote records. These include changing previously-cast ballots to indicate a desired candidate and adding new ballots in support of a desired candidate. Although these hacks tended to be the most obvious, the groups employed numerous tricks to hide them. One blatant case of vote altering added a percentage of votes to candidates of a particular party. This, however, was only done on voting terminals with an ID greater than 2. The group presumed that black-

box testers would use at most two terminals for testing, thus allowing the hack to go undetected.

Another group also included a number of direct vote manipulations. In order to evade the black-box testers, this group only activated their hack after a pre-determined number of votes has been cast, presuming the black box tester would not cast so many votes in testing the system. The group also used the Java hash-code function to detect votes for favorable candidates. Java defines a `hashCode()` method on every object, typically used for storing objects in hashtables. This group determined when to bias a candidate based on whether the hash of the candidate's name was equal to a hard-coded integer, providing more stealth than hard-coding that candidate's name directly into the source code.

One of the submitted projects had a very subtle bug that manipulated the election system's vote counter. When tallying the results of an election, Hack-a-Vote uses a hashtable for each race, mapping a candidate to the number of votes that candidate has received. By cleverly rewriting the `hashCode()` and `equals()` methods of the `Candidate` class, this group made it appear to the hashtable that two candidates were the same if one's name was a prefix of the other. As a result, one of the candidates would get all of the other candidate's votes and the other candidate would not appear in the tally at all. Such subtle changes can significantly influence the outcome of an election yet be undetected by most forms of testing.

4.2 Broken Authentication

Groups also employed a number of techniques to break the authentication scheme built into Hack-a-Vote. Among the most common were back-doors to vote without knowing a PIN. One group implemented a hack that allowed any PIN to successfully authenticate after a non-numerical PIN had been entered. Other groups used numerical constants already in the code, such as 10, the number of PINs active at any one time, and 1776, the network port the console listens on, as back-door PINs. Yet another group accepted any PIN longer than 9 digits. In this last hack, although the administration console returned an error message to the voting terminal, the voting terminal only considered the numeric prefix of the error message. The numeric prefix, of course, indicated the PIN was valid.

Related hacks allowed multiple votes with just one PIN. In one hack, a `ClassCastException` could be triggered after the vote had been submitted but before the login screen was displayed, allowing the voter to vote multiple times. In another, clicking the *Start over* button, which is supposed to allow the voter to restart the voting process, actually submitted the voter's ballot in addition to allowing the voter to start over.

Another group weakened the random-number generator used to generate PINs. They seeded the random-number generator with the current second of the hour (ranging from 0 to 3599). Given knowledge of two PINs generated within a sufficiently narrow time range, a voter could fairly easily determine the initial seed and generate new PINs. In addition, 20 PINs were held valid at a time, despite the election console only displaying 10 PINs at a time. This change allowed a malicious voter to vote multiple times without PINs mysteriously disappear from the console. This group also used

the current PIN to seed a random-number generator used to generate the back-door administrator password, normally entered after a valid PIN to terminate the election. This allowed any voter "in the know" to end the election on that voting terminal.

4.3 Violating Anonymity

A third area that the groups attacked was the randomization of ballots. The original version of Hack-a-Vote shuffled the ballots before writing them to disk. Hacks ranged from simply omitting the shuffle to employing a function called "shuffle" that actually does nothing. One group tagged votes by the terminal ID of the voting terminal that generated them, and shuffled the votes using a random-number generator seeded by the terminal ID, allowing the original order of votes to be fairly easily recovered.

Finally, instead of reordering the votes, one group simply renumbered the ballots and output the ballots in the order originally cast but with the new numbers. A casual examination would not reveal the lack of anonymity, but the votes could easily be tied back to the original voters, assuming the order in which they departed the polls was observed.

4.4 Denial of Service

A number of groups implemented denial-of-service hacks in an attempt to selectively disable the voting system. One group introduced a bug whereby a non-numeric PIN would crash the authentication console, preventing others from authenticating their PINs. Another group implemented a GUI trigger to disable the *Election over* button that would normally be used by an authenticated administrator to end the election, write the votes to disk, and compute a tally. Finally a group added a command-line option to the voting terminal that disrupted communications with the authentication console, effectively disabling the voting terminal.

4.5 Hiding Hacks in Java

Groups made proficient use of features of the Java language in hiding their hacks. A number of groups used Java's exception handling features to craft unusual control flows and skip security checks. One group introduced a `ClassCastException` in the code that clears the ballot and generates the PIN login screen after a voter's votes have been submitted. The result was that the voter could vote multiple times. Another group introduced an `UnknownHostException` in the communication between the voting terminal and the authentication console, triggered upon the detection of an invalid PIN being entered. As a result, all subsequent PINs would also not work. Finally, a group introduced a `NumberFormatException` in the authentication console. Attempting to authenticate using a PIN that is not a number would cause the election console to crash, effectively disabling every voting terminal.

Another common technique for hiding hacks was to integrate them into Java's GUI event handler. In Java, a handler can be registered for any GUI item. On GUI events, such as a mouse click, the handler runs in a separate thread. GUI event handlers were

used to trigger the *Election over* denial of service discussed in Section 4.4 as well as the `ClassCastException` bug described in Section 4.2 and in the previous paragraph.

Finally, groups used Java's polymorphism to obscure their hacks. One group added a variable named `BallotControl` to the `BallotControl` class which contains several important static methods. A call to `BallotControl.saveVotes()`, would appear to invoke a static method of class `BallotControl`, yet actually invoked an instance method of the interloper class. As described in Section 4.1, other groups exploited the semantics of Java's standard `equals()` and `hashCode()` methods to effect their hacks.

4.6 Detection of Hacks

In analyzing the voting systems, different groups used different methodologies. Some groups began with the largest file, an implementation of Windows-style initialization file handling by Steve DeGroof.[3] Others began by examining unusual or unexpected code, including GUI event handlers or usage of Java reflection.

All but one of the hacks were found by at least one group analyzing a system. Approximately two out of every three hacks were caught by both groups analyzing a system. The one hack that went entirely undetected was the denial-of-service command-line option described in Section 4.4. Some of the hacks were detected, but the implications of these hacks were not accurately divined. This included the hack described in Section 4.1 that used the `equals()` and `hashCode()` methods to remove a candidate from the final tally (although there was a bug in this hack which might have impeded its analysis).

5 Discussion

A primary goal of the Hack-a-Vote project was to teach students (and others) about security threats to real-world voting systems. Although the Hack-a-Vote system has only a fraction of the size and complexity of real-world DRE voting systems, we can consider the deviousness of the Trojan horses created by our students and the effectiveness of their auditing to be indicative of what might be accomplished with real-world voting systems.

5.1 Implications of Hacks

Section 4 describes a variety of hacks. If these hacks were to be implemented in a real-world voting system, the results would be devastating. The impact of the vote manipulation bugs, discussed in Section 4.1, is probably the clearest; these bugs could result in a candidate being declared to be the winner even though that candidate didn't actually get the most votes. Similarly, the PIN authentication bugs discussed in Section 4.2 could allow a voter to cast multiple votes, likewise affecting the outcome of the election.

[3] See `http://www.mindspring.com/~degroof/` for more information.

The denial of service hacks discussed in Section 4.4, as well as the administrator authentication bug mentioned in Section 4.2, would result in a great deal of confusion and require some time and expense to fix. If these bugs are exploited in a targeted fashion, perhaps disabling voting machines in a neighborhood known to be highly partisan, then those voters might never get the chance to cast their ballots, with the resulting effect on the election's outcome. Finally, the anonymity hacks discussed in Section 4.3 could allow someone with access to the raw results of an election to determine how an individual voted, opening the door to bribery and coercion of voters.

5.2 Auditing

By design, the Comp527 project gave many advantages to the auditing groups that real world auditors do not have. The auditors had the advantage of familiarity with the unaltered code base, allowing for limited "mental diffs." The code base was only $2,000$ lines of well commented, clean code which easily lent itself to inspection. In addition, students were instructed that drastic changes to the voting machines, such as reimplementing major subsystems, was outside the spirit of the project.

On the other hand, the students are not full-time auditors. They were not devoting eight or more hours each day for the week they were given to perform their analyses. They also were responsible for analyzing two voting systems each, effectively halving the time they could devote to analyzing each system. This deliberate lack of attention is an imperfect simulation of the time available to real auditors of genuine voting systems.

One question these results raise is how many auditors are required to analyze a system. In many cases, two student auditors did not successfully find all of the bugs. In one case, even both groups (a total of four auditors) missed a bug. In a real world system it is dificult to predict how many auditor-hours would be necessary before one could be sure all security holes have been discovered.. The Diebold voting machine code analyzed by Kohno et al. [17], for example, has over $50,000$ lines of code, compared to the $2,000$ lines of code for Hack-a-Vote. According to an employee of SAIC, which was contracted to independently review the Diebold voting machine's code, it would be "easy" to hide malicious code in such a big code base, and the chance of the hack going undetected was 99.9% [13].

In fact, security holes have been found in many real-world systems used in production by millions of people. In many of these cases, the bugs were found by curious individuals without any particular time constraints. For example, security researchers studying Java, as used in millions of web browsers, discovered that many security checks that should have been enforced never were [11]. Despite Microsoft's genuine and intense efforts to find and remove security bugs in its code, Microsoft Windows has also been hit by numerous security holes [5, 9]. Finally, note that open source systems have not been spared. For example, consider Sendmail, one of the most popular Internet e-mail transfer agents. Although Sendmail has been around for nearly twenty years with its source code visible to anybody, it has been hit by several recent security holes [8].

Furthermore, even when vulnerabilities have been discovered and patched, those patches are not always universally applied. Many of the high-profile and devastating Windows worms, including Blaster [6] and Slammer [4], were based on security holes

that had been patched by Microsoft. With voting machines, such patches might not be so easily applied. Many jurisdictions would require that the patched systems be recertified, which is a costly and slow process. In fact, there have been reports of voting machines in a recent election running uncertified software [2].

Auditing is not the only way for security holes to be discovered and patched. Techniques such as proof-carrying code [20] and system-specific static analysis [14] can discover particular types of vulnerabilities. A rigorous software engineering process can also help avoid the malicious introduction of security vulnerabilities. In a recent incident, an attempt to introduce a hack into the Linux kernel was discovered because the modification did not pass through the appropriate channels [18]. Unfortunately, though, there are no known, general-purpose techniques for proving that code is perfectly secure, much less rigorous definitions of what "perfectly secure" might actually mean.

Auditing, security-directed automated code analysis, and rigorous software engineering practices are powerful techniques for reducing the number and severity of security flaws in a program. Although they should remain an important part of the software development process, too much is at stake in an election to rely on these techniques alone. While security flaws discovered in operating systems, servers, or user level applications can have serious repercussions, if security flaws are found within America's voting machines, the repercussions can be catastrophic.

6 Conclusion

Electronic voting has been seen by many as a solution to the problems of traditional paper-based voting schemes, as prominently seen in the 2000 United States presidential election. While direct recording electronic (DRE) voting systems have some usability advantages over traditional systems, they raise serious security concerns. We have shown, using a "toy" voting system called Hack-a-Vote, how easily a purely electronic voting system can be compromised and how difficult it can be for auditors to identify and correct any "hacks" in the voting system, any of which could otherwise completely compromise the results of an election.

The best solution is not to abandon the trend toward computerized voting but rather to augment voting machines with a paper audit trail. With such a voter-verifiable audit trail [19, 12], voting machines generate a paper printout of a voter's vote which the machine cannot then alter; that paper becomes the canonical representation of the voters' intent. Voters can verify that their votes are correctly indicated, and election officials can count the printouts either mechanically or by hand to achieve a high degree of accuracy. Such systems remove the software in the voting machines from the trusted computing base of the election. With the inevitable bugs and potential for manipulation, our elections need to generate the most accurate tallies possible. This can only be achieved with election systems that are, by design, not dependent on the correctness of their software.

Acknowledgements

The authors would like to thank the students who took Comp527 this fall and participated in this project: David Anderson, Jonathan Bannet, Ryan Bergauer, Anwis Das, Eliot Flannery, Brian Greinke, Andreas Haeberlen, Feng He, James Hsia, Dayong Huang, Brian Teague, and Ping Yuan. Thanks also to David Dill for the original idea leading to Hack-a-Vote. This work is supported, in part, by NSF Grant CCR-9985332, Texas ATP grant 003604-0053-2001, and gifts from Microsoft and Schlumberger.

References

[1] R. Anderson. How to cheat at the lottery (or, massively parallel requirements engineering). In *Annual Computer Security Applications Conference Proceedings*, Phoenix, AZ, Dec. 1999.

[2] California Secretary of State. Secretary of State Kevin Shelley launches independent audit of California voting systems. News release KS03:100, Nov. 12, 2003. `http://www.ss.ca.gov/executive/press_releases/2003/03_100.pdf`.

[3] CERT Coordination Center. CERT advisory CA-2002-17: Apache web server chunk handling vulnerability, June 2002. `http://www.cert.org/advisories/CA-2002-17.html`.

[4] CERT Coordination Center. CERT advisory CA-2003-04: MS-SQL Server worm, Jan. 2003. `http://www.cert.org/advisories/CA-2003-04.html`.

[5] CERT Coordination Center. CERT advisory CA-2003-16: Buffer overflow in Microsoft RPC, July 2003. `http://www.cert.org/advisories/CA-2003-16.html`.

[6] CERT Coordination Center. CERT advisory CA-2003-20: W32/Blaster worm, Aug. 2003. `http://www.cert.org/advisories/CA-2003-20.html`.

[7] CERT Coordination Center. CERT advisory CA-2003-24: Buffer management vulnerability in OpenSSH, Sept. 2003. `http://www.cert.org/advisories/CA-2003-24.html`.

[8] CERT Coordination Center. CERT advisory CA-2003-25: Buffer overflow in Sendmail, Sept. 2003. `http://www.cert.org/advisories/CA-2003-25.html`.

[9] CERT Coordination Center. CERT advisory CA-2003-27: Multiple vulnerabilities in Microsoft Windows and Exchange, Oct. 2003. `http://www.cert.org/advisories/CA-2003-27.html`.

[10] CERT Coordination Center. CERT incident note IN-2003-03: W32/Sobig.F worm, Aug. 2003. http://www.cert.org/incident_notes/IN-2003-03.html.

[11] D. Dean, E. W. Felten, D. S. Wallach, and D. Balfanz. Java security: Web browsers and beyond. In D. E. Denning and P. J. Denning, editors, *Internet Besieged: Countering Cyberspace Scofflaws*, pages 241–269. ACM Press, New York, NY, Oct. 1997.

[12] D. L. Dill, R. Mercuri, P. G. Neumann, and D. S. Wallach. Frequently asked questions about DRE voting systems, Feb. 2003. http://www.verifiedvoting.org/drefaq.asp.

[13] M. Dresser. Legislators are warned by voting system critic: Expert who found flaws fears they weren't fixed. *The Baltimore Sun*, Nov. 14, 2003. http://www.sunspot.net/news/local/bal-md.diebold14nov14,0,3307906.story.

[14] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating System Principles*, Banff, Alberta, Canada, Oct. 2001.

[15] A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. In *14th IEEE Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, Canada, June 2001.

[16] B. Harris and D. Allen. *Black Box Voting: Vote Tampering in the 21st Century*. Plan Nine Publishing, High Point, NC, 2003.

[17] T. Kohno, A. Stubblefield, A. D. Rubin, and D. S. Wallach. Analysis of an electronic voting system. Technical Report TR-2003-19, Johns Hopkins Information Security Institute, Baltimore, MD, July 2003.

[18] L. McVoy. BK2CVS problem. *Linux Kernel Mailing List*, Nov. 5, 2003. http://www.ussg.iu.edu/hypermail/linux/kernel/0311.0/0621.html.

[19] R. Mercuri. *Electronic Vote Tabulation: Checks and Balances*. PhD thesis, University of Pennsylvania, Philadelphia, PA, Oct. 2000.

[20] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *2nd Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Oct. 1996.