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

In a quest for election legitimacy, officials are increasingly deploying direct recording electronic (DRE) voting systems. A project to assess their trustworthiness revealed both the ease of introducing bugs into such systems and the difficulty of detecting them during audits.

Jonathan Bannet, David W. Price, Algis Rudys, Justin Singer, and Dan S. Wallach
Rice University

The democratic process rests on a fair, universally accessible voting system through which all citizens can easily and accurately cast a vote. With the 2000 US presidential election, however, the country got a firsthand look at the results of a flawed voting system, which fueled renewed public interest in voting system reliability. People became especially enchanted by the computer's siren song, so election officials have increasingly examined and adopted voting systems that rely primarily on computers to record and tabulate votes. Much of their attention has centered on direct recording electronic (DRE) voting systems, which completely eliminate paper ballots.

DRE voting systems have some inherent advantages over paper-based voting schemes, including a decrease in voter error. If the system's interface has been well designed, for example, it seeks confirmation if the voter fails to cast a vote in a particular race and disallows multiple votes in the same race. DREs also make it possible to accommodate people with different disabilities, helping them vote without human assistance. Unfortunately, recent analyses of one popular vendor's DRE voting system indicated numerous security oversights,[1,2] including

- Voters can cast multiple votes without leaving a trace.
- Anyone with access to a voting machine can perform administrative actions, including viewing partial results and terminating an election early.
- Communications between voting terminals and the central server are not properly encrypted, making it possible for a malicious "man in the middle" to alter communication content.

According to analysts, this flawed state is the result of undisciplined software development and a process that failed to encourage developers to anticipate or fix security holes. The closed-source approach to software development, which shielded the source code from public review and comment, only served to delay the necessary scrutiny. Of course, as daily headlines demonstrate, neither a commitment to software security nor an open-source approach to software development prevents software security holes. (For example, see www.cert.org/advisories, which lists security holes that have been discovered in the past year.) Software developers and auditors who follow standard software engineering practices have proven unable to ship bug-free software. In general, producing software free from *all* security holes is significantly harder than an attacker's goal: to find and exploit a single bug.

We recently conducted a project to demonstrate that electronic voting software is not immune from these security concerns. Here, we describe Hack-a-Vote, a simplified DRE voting system that we initially developed to demonstrate how easy it might be to insert a Trojan horse into a voting system. Having accomplished this, we used Hack-a-Vote in an associated course project, in which student teams implemented their own Trojan horses, then searched the source code for their classmates' malicious code. The Hack-a-Vote project revealed the potential damage individuals can cause with electronic voting systems, the feasibility of finding system weaknesses (deliberate or otherwise), and some solutions to mitigate the damage. Hack-a-Vote and our course assignment are freely available online at www.cs.rice.edu/~dwallach/courses/comp527_f2003/voteproject.html.

The Hack-a-Vote system

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

Hack-a-Vote first authenticates the user with a simple PIN authentication scheme inspired by Hart Intercivic's eSlate (www.hartintercivic.com/solutions/eslate.html), which is used for elections in Houston, Texas, among other places. With eSlate, voters first sign in at their local voting precinct. Every group of eSlate voting terminals is connected via a wired network to an election management console. The user approaches the console, and an election official prints him or her a four-digit PIN; the user then goes to any available voting terminal and enters this PIN. Once eSlate validates the PIN, the user can vote. Similarly, in the Hack-a-Vote implementation, the server maintains a list of a few valid PINs that are displayed to the election administrator (we didn't bother to print the PINs, although it would be an easy extension to add). Once a voter uses a PIN, the system invalidates it and randomly generates a new one. Figure 1a shows the PIN login screen.

Following authentication, users are presented with voting screens containing a series of races (see Figure 1b). Once they've cast a vote in every race, they can view and confirm a summary of their votes (see Figure 2a). When the user 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 lets users change their selections. Once the election closes, the system randomly shuffles and tallies all votes and writes them out to disk. The machine then displays the final tally, as Figure 2b shows.

In Hack-a-Vote's original implementation, we added a Trojan horse to the code that prepares votes for writing out to disk. By adding a few extra command-line options, we made it possible for the user to indicate a candidate or political party that should receive extra votes. With a configurable probability, the system would modify votes for other candidates before writing them to disk and counting them. In the student version, we removed the cheating code (about 150 lines in one file).

The Hack-a-Vote assignment

The Hack-a-Vote project introduces students to some of the many faces of security, from the malicious hacker's design of subtle system attacks, to the auditing required to uncover those attacks, to the higher level challenges of secure system design. 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 UK lottery experiment, in which he asked students to consider the security threats of running a national lottery.[3]

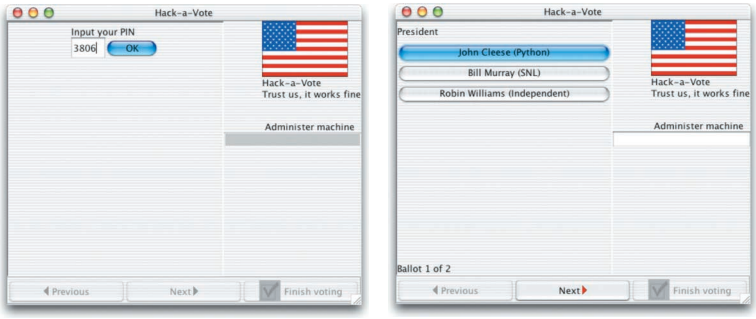

Figure 1. The initial Hack-a-Vote screens. (a) The PIN login screen. Once a voter enters a PIN, the system invalidates it for future use. (b) Following authentication, the system displays selection screens containing a series of races.

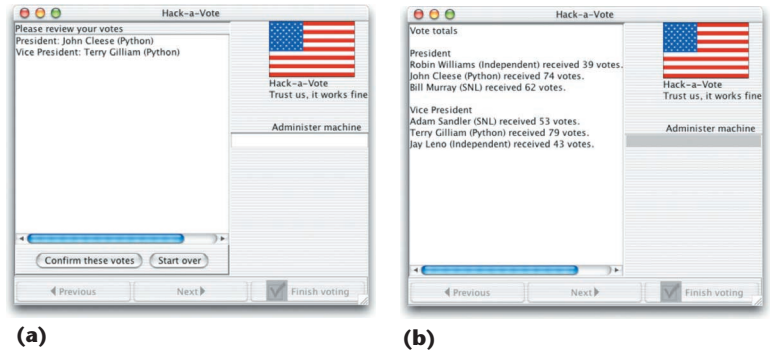

Figure 2. Additional screens. (a) The Hack-a-Vote confirmation screen offers a summary after users have voted. (b) The final tally screen.

For Hack-a-Vote, we split students in Comp527, a graduate-level computer security course, into groups of two for a three-part assignment. For the project's first phase, the groups assumed the role of Hack-a-Vote developers who were paid off by a nefarious organization to rig an election without getting caught. We placed no requirements on the type of malicious hack, so long as students could justify how it could be used to ruin or bias an election without being detected. Solutions could thus 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.

In the second phase, each group received modified voting systems from two other randomly selected groups in their class. They now assumed the role of independent auditors, with one week to find any security-critical flaws in the altered code. In this phase, we prohibited any direct comparison with the original code, requiring students to

rely solely on their memory of the voting machine and its functionality—a requirement enforced by the university's honor code. (Otherwise, students could easily locate changes using the UNIX `diff` tool, giving them greater powers than a genuine auditor might have when studying a voting system.)

In the final phase, the focus shifted: We gave students two and a half weeks to fix a problem discovered in the Diebold voting machine's smartcard authentication protocol.[2] Students first had to consider what security threats a voting system faces. They then designed an improved authentication protocol and used Cryptyc, a cryptographic protocol type checker,[4] to model and verify the protocol. They also had to explain the protocol in terms of the security threats it was designed to thwart.

Results

Although the groups designed various hacks, they generally fell into one of four categories:

- direct vote-manipulation hacks,
- attacks aimed at breaking the authentication mechanism for PINs or administrative access,
- hacks directed at defeating voter anonymity, and
- denial-of-service bugs.

The most blatant hacks fell into the first category.

Result manipulation

Hacks aimed at directly manipulating vote records skewed results in favor of a desired candidate by changing previously cast ballots or adding new ballots accordingly. Although these hacks were typically the most obvious, the groups used numerous tricks to hide them. One overt case of vote altering added a percentage of votes to a particular party's candidates. However, the students did this only on voting terminals with an ID greater than two, presuming that black-box testers would use at most two terminals for testing.

Another group attempted to evade black-box testers by activating their hack only after a predetermined number of votes had been cast, presuming that testers would cast only a limited number of votes. This group also used the Java hash-code function to detect votes for favorable candidates. For every object, Java defines a `hashCode()` method, which is typically used to store objects in hash tables. This group determined when to bias a candidate based on whether the candidate name's hash was equal to a hard-coded integer, providing more stealth than hard coding the name directly into the source code.

Another project had a very subtle bug that manipulated the election system's vote counter. When tallying election results, Hack-a-Vote uses a hash table for each race, mapping each candidate to the number of votes he or she received. By cleverly rewriting the `Candidate` class's `hashCode()` and `equals()` methods, the group made it appear to the hash table 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, which can significantly influence an election's outcome, would go undetected by most forms of testing.

Broken authentication

Groups used several techniques to break Hack-a-Vote's authentication scheme. Among the most common were using back doors to vote without a PIN. One group implemented a hack that let any PIN successfully authenticate after a non-numerical PIN was 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 nine digits. In this hack, the administration console returned an error message to the voting terminal, but the voting terminal only considered the error message's numeric prefix, which indicated that the PIN was valid.

Related hacks let users vote multiple times with a single PIN. In one hack, a `ClassCastException` was triggered after a vote had been submitted but before the login screen was displayed, letting the user vote multiple times. In another, clicking the "Start over" button—which is supposed to let the user restart the voting process from scratch—submitted the existing ballot and let the voter start over.

Another group weakened the random-number generator that generates PINs by seeding it with the hour's current second (ranging from 0 to 3,599). 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, the hack made 20 PINs valid at a time, despite the election console displaying only 10 PINs at a time. A malicious user could thus vote multiple times without PINs mysteriously disappearing from the console. This group also used the current PIN to seed a random-number generator that generated a back-door administrator password, which is usually entered after a valid PIN to terminate the election. A malicious user could thus end the election on that voting terminal.

Violating anonymity

A third area of attack was ballot randomization. The original Hack-a-Vote version shuffled ballots before writing them to disk. Hacks here ranged from simply omitting the shuffle to employing a "shuffle" function that actually did 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, which made the original vote order

fairly easy to recover.

Finally, instead of reordering the votes, one group simply renumbered the ballots, outputting them in their original order but with new numbers. Although a casual examination would not reveal the lack of anonymity, the votes could be easily tied to the original voters if the order in which they departed the polls was observed.

Denial of service

Several groups implemented denial-of-service (DoS) 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, which the authenticated administrator typically uses 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.

Hiding hacks in Java

Groups made proficient use of Java features to hide their hacks. Several of them 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 user's votes have been submitted, letting the user vote multiple times. Another group introduced an `UnknownHostException` in the communication between the voting terminal and the authentication console, which was triggered when an invalid PIN entry was detected. With this exception, the code skipped the authentication check on the subsequent PIN, making any entry acceptable. Finally, a group introduced a `NumberFormatException` in the authentication console, which crashed the election console—and thus disabled the voting terminals—if a user entered a nonnumerical PIN.

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. Groups used GUI event handlers to trigger both the "Election over" denial of service and the `ClassCastException` bug that we described earlier.

Finally, groups also used Java's polymorphism to obscure their hacks. One group added a variable called `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 invoke an interloper class instance method. As we described earlier, other groups exploited the semantics of Java's standard `equals()` and `hashCode()` methods to effect their hacks.

Hack detection

Different groups used different methodologies to analyze the voting systems. Some groups began with the largest file, an implementation of Steve DeGroof's Windows-style initialization file handling (see http://degroof.home.mindspring.com/java/ for more information). Others began by examining unusual or unexpected code, including the use of GUI event handlers and Java reflection.

Two groups analyzed each system, and—with only one exception—at least one of these groups found all the hacks; both groups found about two out of three hacks. The one hack that went entirely undetected was the denial-of-service command-line option. In other cases, groups detected the hacks, but did not accurately divine their implications. Among these was the hack that used the `equals()` and `hashCode()` methods to remove a candidate from the final tally (although a bug in this hack might have impeded its analysis).

Discussion

One of our primary goals for 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 systems, the deviousness of the Trojan horses the students created and the effectiveness of their auditing are indicative of what might be accomplished with real-world voting systems.

Hack implications

If the hacks our students devised were implemented in a real-world voting system, the results would be devastating. The impact of the vote-manipulation bugs is probably the clearest: these bugs could result in a candidate who did not actually receive the most votes being declared the winner of an election. Similarly, the PIN-authentication bugs could let a voter cast multiple votes, likewise affecting the election's outcome.

The DoS hacks and the administrator-authentication bug would create 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 highly partisan neighborhood, for example—targeted users might never get the chance to cast their ballots. Finally, anonymity hacks could let malicious users with access to an election's raw results determine how an individual voted, opening the door to bribery and voter coercion.

Auditing

By design, the Comp527 project gave auditing groups many advantages that real-world auditors do not have:

- Auditors were familiar with the unaltered code base, allowing for limited "mental diffs."
- The code base consisted of only 2,000 lines of well-commented, clean code that was easy to inspect.
- We did not permit drastic changes to the voting machines, such as reimplementing major subsystems.

On the other hand, the students were not full-time auditors. We gave each group one week to analyze two different voting systems, and the students didn't have eight or more hours each day to devote to the task. This deliberate lack of attention is an imperfect simulation of the time that real auditors would devote to genuine voting systems.

One question our results raise is this: How many auditors are required to analyze a system? In many cases, two student auditors did not successfully find all the bugs. In one case, both groups (a total of four auditors) missed a bug. In a real-world system, it's difficult to predict how many auditor-hours would be required to ensure the discovery of all security holes. The Diebold voting machine code, for example, has over 50,000 lines of code (compared to Hack-a-Vote's 2,000 lines). According to an employee of Science Applications International Corporation, which independently reviewed the Diebold code, it would be "easy" to hide malicious code in such a big code base, with a 99.9 percent chance that the hack would go undetected.[5]

In fact, security holes have been found in many real-world systems that millions of people use. In many of these cases, curious individuals without any particular time constraints found the bugs. Security researchers studying Java as it is used in millions of Web browsers, for example, discovered that many basic security checks were never enforced.[6] Despite Microsoft's genuine and intense effort to find and remove security bugs in its code, Windows has been hit by numerous security holes (see www.cert.org/advisories/CA-2003-16.html and www.cert.org/advisories/CA-2003-27.html for an overview of some recent security vulnerabilities).

Finally, open-source systems also have had their difficulties. Consider Sendmail, for example, which is one of the most popular Internet email transfer agents. Although Sendmail has been around for nearly 20 years and its source code is visible to anyone who's interested, it's been hit by several recent security holes (see www.cert.org/advisories/CA-2003-25.html for more information).

Even when vulnerabilities are discovered and patched, those patches are not always universally applied. Many of the high-profile and devastating Windows worms, including Blaster and Slammer, were based on security holes that Microsoft patched (see www.cert.org/advisories/CA-2003-20.html and www.cert.org/advisories/CA-2003-04 for more information). With voting machines, such patches might not be so easily applied. Many jurisdictions would require that patched systems be recertified—a slow and costly process. In fact, in a recent election, many voting machines reportedly ran uncertified software.[7]

Auditing is not the only way to discover and patch security holes. Techniques such as proof-carrying code[8] and system-specific static analysis[9] can uncover specific vulnerabilities. A rigorous software engineering process also can help prevent 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 failed to pass through the appropriate channels.[10] 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 a program's security flaws. Although such practices should remain an important part of the software development process, in an election, too much is at stake 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 in America's voting machines, the repercussions could be catastrophic.

Many people view electronic voting as a solution to traditional paper-based voting problems, such as those in the 2000 US presidential election. But, even though DRE voting systems have some usability advantages over traditional systems, they raise serious security concerns. As we've shown with our "toy" system, it's easy to compromise a purely electronic voting system and difficult for auditors to identify and correct hacks that might otherwise completely compromise election results.

The best solution is not to abandon the trend toward computerized voting, but rather to augment voting machines with a paper audit trail. With a voter-verifiable audit trail,[11,12] voting machines generate a paper printout of a user's vote that the machine cannot alter; that paper becomes the canonical representation of the voters' intent. Voters can thus verify that their votes are correctly indicated, and election officials can count the printouts—either mechanically or by hand—to assure accuracy. Such systems remove the voting machine software from the election's trusted computing base. Given the inevitable bugs and potential for manipulation, our elections must generate the most accurate tallies possible. This can only be achieved with election systems that are, by design, independent of software correctness. □

Acknowledgments

We thank the students in Comp527 who 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 US NSF grant CCR-9985332, Texas ATP grant 003604-0053-2001, and gifts from Microsoft and Schlumberger.

References

1. B. Harris and D. Allen, *Black Box Voting: Vote Tampering in the 21st Century*, Plan Nine Publishing, 2003.
2. T. Kohno et al., *Analysis of an Electronic Voting System*, tech. report TR-2003-19, Johns Hopkins Information Security Inst., 2003.
3. R. Anderson, "How to Cheat at the Lottery (or, Massively Parallel Requirements Engineering)," *Ann. Computer Security App. Conf. Proc.*, IEEE CS Press, 1999, pp. 1–2.
4. A.D. Gordon and A. Jeffrey, "Authenticity by Typing for Security Protocols," *14th IEEE Computer Security Foundations Workshop*, IEEE CS Press, 2001, pp. 145–159.
5. M. Dresser, "Legislators Are Warned by Voting System Critic: Expert Who Found Flaws Fears They Weren't Fixed," *The Baltimore Sun*, 14 Nov. 2003.
6. D. Dean et al., "Java Security: Web Browsers and Beyond," *Internet Besieged: Countering Cyberspace Scofflaws*, D.E. Denning and P.J. Denning, eds., ACM Press, 1997, pp. 241–269.
7. California Secretary of State, "Secretary of State Kevin Shelley Launches Independent Audit of California Voting Systems," news release KS03:100, 12 Nov. 2003, www.ss.ca.gov/executive/press_releases/2003/03_100.pdf.
8. G.C. Necula and P. Lee, "Safe Kernel Extensions without Run-Time Checking," *2nd Symp. Operating Systems Design and Implementation* (OSDI), Usenix Assoc., 1996, pp. 229–243.
9. D. Engler et al., "Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code," *Proc. 18th ACM Symp. Operating System Principles*, ACM Press, 2001, pp. 57–72.
10. L. McVoy, "BK2CVS Problem," *Linux Kernel Mailing List*, 5 Nov. 2003, www.ussg.iu.edu/hypermail/linux/kernel/0311.0/0621.html.
11. R. Mercuri, *Electronic Vote Tabulation: Checks and Balances*, PhD thesis, Dept. of Computer and Information Systems, Univ. of Pennsylvania, Philadelphia, Oct. 2000.
12. D.L. Dill et al., "Frequently Asked Questions about DRE Voting Systems," Feb. 2003, www.verifiedvoting.org/drefaq.asp.

Jonathan Bannet *is a graduate student in computer science at Rice University. His research interests include language-based security. He received a BS in computer science from Rice University. Contact him at jbannet@cs.rice.edu.*

David W. Price *is a law student at the University of Texas Law School. His research interests include technology policy. He received a BS in computer science from Rice University. Contact him at dwp@alumni.rice.edu.*

Algis Rudys *is a graduate student in computer science at Rice University. His research interests include mobile code security. He received an MS in computer science from Rice University. He is a member of the Usenix Association. Contact him at arudys@cs.rice.edu.*

Justin Singer *is president of SMBology, a Houston-based IT consulting firm. He received a BS in computer science from Rice University. Contact him at jsinger@smbology.com.*

Dan S. Wallach *is an assistant professor of computer science at Rice University. His research interests include a variety of security topics. He received a PhD in computer science from Princeton University. He is a member of the ACM, Usenix, and the IEEE. Contact him at dwallach@cs.rice.edu.*