Analysis of an Electronic Voting System

Tadayoshi Kohno
Information Security Institute
Johns Hopkins University
yoshi@cs.jhu.edu

Adam Stubblefield
Information Security Institute
Johns Hopkins University
astubble@cs.jhu.edu

Aviel D. Rubin
Information Security Institute
Johns Hopkins University
rubin@cs.jhu.edu

Dan S. Wallach
Department of Computer Science
Rice University
dwallach@cs.rice.edu

July 23, 2003

Abstract

Recent election problems have sparked great interest in managing the election process through the use of electronic voting systems. While computer scientists, for the most part, have been warning of the perils of such action, vendors have forged ahead with their products, claiming increased security and reliability. Many municipalities have adopted electronic systems, and the number of deployed systems is rising. For these new computerized voting systems, neither source code nor the results of any third-party certification analyses have been available for the general population to study, because vendors claim that secrecy is a necessary requirement to keep their systems secure. Recently, however, the source code purporting to be the software for a voting system from a major manufacturer appeared on the Internet. This manufacturer's systems were used in Georgia's state-wide elections in 2002, and the company just announced that the state of Maryland awarded them an order valued at up to $55.6 million to deliver touch screen voting systems.[1]

This unique opportunity for independent scientific analysis of voting system source code demonstrates the fallacy of the closed-source argument for such a critical system. Our analysis shows that this voting system is far below even the most minimal security standards applicable in other contexts. We highlight several issues including unauthorized privilege escalation, incorrect use of cryptography, vulnerabilities to network threats, and poor software development processes. For example, common voters, without any insider privileges, can cast unlimited votes without being detected by any mechanisms within the voting terminal. Furthermore, we show that even the most serious of our outsider attacks could have been discovered without the source code. In the face of such attacks, the usual worries about insider threats are not the only concerns; outsiders can do the damage. That said, we demonstrate that the insider threat is also quite considerable. We conclude that, as a society, we must carefully consider the risks inherent in electronic voting, as it places our very democracy at risk.

[1] `http://www.corporate-ir.net/ireye/ir_site.zhtml?ticker=DBD&script=410&layout=-6&item_id=433030`

1 Introduction

The essence of democracy is that everyone accepts the results of elections, even when they lose them. Elections allow the populace to choose their representatives and express their preferences for how they will be governed. Naturally, the integrity of the election process is fundamental to the integrity of democracy itself. And, unsurprisingly, history is littered with examples of elections being manipulated in order to influence their outcome.

The design of a "good" voting system, whether electronic or using traditional paper ballots or mechanical devices, must be robust against a wide variety of potentially fraudulent behavior. The *anonymity* of a voter's ballot must be preserved, both to guarantee the voter's safety when voting against a malevolent candidate, and to guarantee that voters have no evidence that proves which candidates received their votes. The existence of such evidence would allow votes to be purchased by a candidate. The voting system must also be *tamper-resistant* to thwart a wide range of attacks, including ballot stuffing by voters and incorrect tallying by insiders. Another important consideration, as shown by the so-called "butterfly ballots" in the Florida 2000 presidential election, is the importance of *human factors*. A voting system must be comprehensible to and usable by the *entire* voting population, regardless of age, infirmity, or disability. Providing accessibility to such a diverse population is an important engineering problem and one where, if other security is done well, electronic voting could be a great improvement over current paper systems. Flaws in any of these aspects of a voting system, however, can lead to indecisive or incorrect election results.

1.1 Electronic voting systems

There have been several studies on using computer technologies to improve elections [Cal01, Cal00, Mer00, Nat01, Rub02]. These studies caution about the risks of moving too quickly to adopt electronic voting machines because of the software engineering challenges, insider threats, network vulnerabilities, and the challenges of auditing.

As a result of the Florida 2000 presidential election, the inadequacies of widely-used punch card voting systems have become well understood by the general population. This has led to increasingly widespread adoption of "direct recording electronic" (DRE) voting systems. DRE systems, generally speaking, completely eliminate paper ballots from the voting process. As with traditional elections, voters go to their home precinct and prove that they are allowed to vote there, perhaps by presenting an ID card, although some states allow voters to cast votes without any identification at all. After this, the voter is typically given a PIN or a smartcard or some other token that allows them to approach a voting terminal, enter the PIN or smartcard, and then vote for their candidates of choice. When the voter's selection is complete, DRE systems will typically present a summary of the voter's selections, giving them a final chance to make changes. Subsequent to this, the ballot is "cast" and the voter is free to leave.

The most fundamental problem with such a voting system is that the entire election hinges on the correctness, robustness, and security of the software within the voting terminal. Should that code have security-relevant flaws, they might be exploitable either by unscrupulous voters or by malevolent insiders. Such insiders include election officials, the developers of the voting system, and the developers of the embedded operating system on which the voting system runs. If any party introduces flaws into the voting system software or takes advantage of pre-existing flaws, then the results of the election cannot be assured to accurately reflect the votes legally cast by the voters.

The only known solution to this problem is to introduce a "voter-verifiable audit trail." [DMNW03]. Most commonly, this is achieved by adding a printer to the voting terminal. When the voter finishes selecting candidates, a ballot is printed on paper and presented to the voter. If the printed ballot reflects the voter's intent, the ballot is saved for future reference. If not, the ballot is mechanically destroyed. Using this "Mercuri method," [Mer00] the tally of the paper ballots takes precedence over any electronic tallies. As a result, the correctness of the voting terminal software no longer matters; either a voting terminal prints correct ballots or it is taken out of service.

1.2 "Certified" DRE systems

Many government entities have adopted paperless DRE systems without appearing to have critically questioned the security claims made by the systems' vendors. Until recently, such systems have been dubiously "certified" for use without any public release of the analyses behind these certifications, much less any release of the source code that might allow independent third parties to perform their own analyses. Some vendors have claimed "security through obscurity" as a defense, despite the security community's universally held belief in the inadequacy of obscurity to provide meaningful protection.

"Security through obscurity" is a long-rejected theory that systems can be made more secure by simply hiding the security mechanisms from public view. While this theory has some validity in situations where the need for security is not great — hiding a spare key to a liquor cabinet just out of sight of small children — the theory has been soundly rejected as a means of serious security [Ker83]. This is because it has the twin faults of not providing serious security from real attackers, who can easily overcome minimal security measures, and of limiting public and general security oversight of the system, which has proven to be the best method for creating and maintaining a truly secure system [Sch00].

Indeed, source code that appears to correspond to a version of Diebold's voting system appeared recently on the Internet. This appearance, announced by Bev Harris and discussed in her book, *Black Box Voting* [Har03], gives us a unique opportunity to analyze a widely used, paperless DRE system and evaluate the manufacturer's security claims.

To the best of our knowledge, the code (hereafter referred to as the "Diebold code") was discovered by others on a publicly available Diebold ftp site in January, 2003. It has since been copied to other sites around the world and its release has been the subject of numerous press reports. To the authors' knowledge, Diebold has raised no objection to the broad publication and republication of the code to date. Jones discusses the origins of this code in extensive detail [Jon03].

The security of Diebold's voting system is of growing concern as on July 21, 2003, the company finalized an agreement for up to $55.6 million to deliver touch-screen voting technology to the state of Maryland. The contract includes about 11,000 Diebold touch-screen voting systems. Diebold voting systems were also used in state-wide elections in Georgia in 2002.

1.3 Summary of results

We only inspected unencrypted source code that we believe was used in Diebold's AccuVote-TS voting terminal [Die03] (the "AVTSCE" tree in the CVS archive). We have not independently verified the current or past use of the code by Diebold or that the code we analyzed is actually Diebold code, although as explained further in Section 6.1, the copyright notices and code legacy information in the code itself are consistent with publicly available systems offered by Diebold and a company it acquired in 2001, Global Election Systems. Also, the code itself built and worked as an election system consistent with Diebold's public descriptions of its system. We concluded that even if it turned out that the code was not part of a current or past Diebold voting system, analysis of it would be useful to the broader public debate around electronic voting systems security and assist election officials and members of the public in their consideration of not only Diebold systems, but other electronic voting systems currently being marketed nationwide and around the world. We did not have source code to Diebold's GEMS back-end election management system.

Furthermore, we only analyzed source code that could be directly observed and copied from the CVS software archive without further effort. A large amount of the other data made publicly available was protected by very weak compression/encryption software known as PKZip, which requires a password for access to the underlying work. PKZip passwords are relatively easy to avoid, and programs for locating passwords for PKZip files are readily available online. Moreover, passwords that others have located for these files have been freely available online for some time. Nonetheless, we decided to limit our research to only the files that were publicly available without any further effort, in part due to concerns about possible liability under the anti-circumvention provisions of the Digital Millennium Copyright Act.

Even with this restricted view of the source code, we discovered significant and wide-reaching security vulnerabilities in the AccuVote-TS voting terminal. Most notably, voters can easily program their own smartcards to simulate the behavior of valid smartcards used in the election. With such homebrew cards, a voter can cast multiple ballots without leaving any trace. A voter can also perform actions that normally require administrative privileges, including viewing partial results and terminating the election early. Similar undesirable modifications could be made by malevolent poll workers (or even maintenance staff) with access to the voting terminals before the start of an election. Furthermore, the protocols used when the voting terminals communicate with their home base, both to fetch election configuration information and to report final election results, do not use cryptographic techniques to authenticate the remote end of the connection nor do they check the integrity of the data in transit. Given that these voting terminals could communicate over insecure phone lines or even wireless Internet connections, even unsophisticated attackers can perform untraceable "man-in-the-middle" attacks.

As part of our analysis, we considered both the specific ways that the code uses cryptographic techniques and the general software engineering quality of its construction. Neither provides us with any confidence of the system's correctness. Cryptography, when used at all, is used incorrectly. In many places where cryptography would seem obvious and necessary, none is used. More generally, we see no evidence of rigorous software engineering discipline. Comments in the code and the revision change logs indicate the engineers were aware of areas in the system that needed improvement, though these comments only address specific problems with the code and not with the design itself. We also saw no evidence of any change-control process that might restrict a developer's ability to insert arbitrary patches to the code. Absent such processes, a malevolent developer could easily make changes to the code that would create vulnerabilities to be later exploited on Election Day. We also note that the software is written entirely in C++. When programming in an unsafe language like C++, programmers must exercise tight discipline to prevent their programs from being vulnerable to buffer overflow attacks and other weaknesses. Indeed, buffer overflows caused real problems for AccuVote-TS systems in real elections.[2]

2 System overview

Although the Diebold code is designed to run on a DRE device (an example of which is shown in Figure 1), one can run it on a regular Microsoft Windows computer (during our experiments we compiled and ran the code on a Windows 2000 PC).

In the following we describe the process for setting up and running an election using the Diebold system. Although we know exactly how the code works from our analysis, we must still make some assumptions about the external processes at election sites. In all such cases, our assumptions are based on the way the Diebold code works, and we believe that our assumptions are reasonable. There may, however, be additional administrative procedures in place that are not indicated by the source code. We first describe the architecture at a very high level, and then, in Section 2.1 we present an overview of the code. Since the Diebold code can be run both on DRE devices and PCs, we shall refer to a device running the vote collection software as a *voting terminal*.

SETTING UP. Before an election takes place, one of the first things the election officials must do is specify the

[2] `http://www.sccgov.org/scc/assets/docs/209815KeyboardAttachment-200440211.pdf` (page 48)

	Voter (with forged smartcard)	Poll Worker (with access to storage media)	Poll Worker (with access to network traffic)	Internet Provider (with access to to network traffic)	OS Developer	Voting Device Developer	Section
Vote multiple times using forged smartcard	●	●	●				3.1.1
Access administrative functions or close polling station	●	●			●	●	3.1.2
Modify system configuration		●			●	●	4.2
Impersonate legitimate voting machine to tallying authority		●	●	●	●	●	4.3
Modify ballot definition (e.g., party affiliation)		●	●	●	●	●	4.3 and 5.1
Cause votes to be miscounted by tampering with configuration		●	●	●	●	●	4.3 and 5.1
Tamper with audit logs		●			●	●	4.4
Create, delete, and modify votes on device		●			●	●	4.4
Link votes to voters		●			●	●	4.4
Delay the start of an election		●	●	●	●	●	5.2
Tamper with election results		●	●	●	●	●	5.2
Insert backdoors into code					●	●	6.3

Table 1: This table summarizes some of the more important attacks on the system.

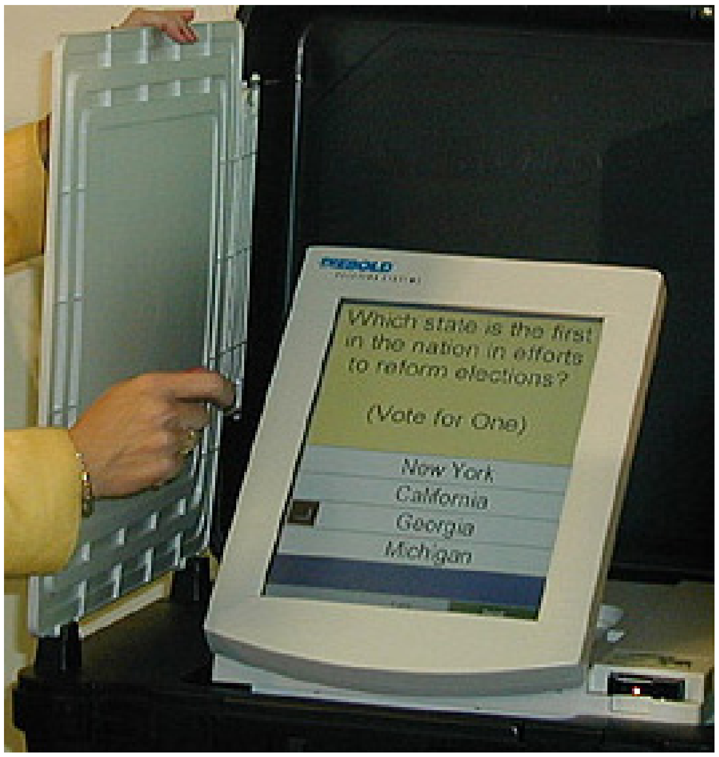
DIEBOLD
Which state is the first in the nation in efforts to reform elections?
(Vote for One)
New York
California
Georgia
Michigan

Figure 1: A Diebold DRE Voting Machine (photo from `http://www.sos.state.ga.us/`). Note the smartcard reader in the lower-right hand corner.

political offices and issues to be decided by the voters along with the candidates and their party affiliations. Variations on the ballot can be presented to voters based on their party affiliations. We call this data a *ballot definition*. In the Diebold system, a ballot definition is encoded as the file `election.edb` and stored on a back-end server.

Shortly prior to the election, the voting terminals must be installed at each voting location. In common usage, we believe the voting terminals will be distributed without a ballot definition pre-installed. Instead, a governmental entity using Diebold voting terminals has a variety of choices in how to distribute the ballot definitions. They may be distributed using removable media, such as floppy disks or storage cards. They may also be transmitted over the Internet or a dial-up connection. This provides additional flexibility to the election administrator in the event of last-minute changes to the ballot.

THE ELECTION. Once the voting terminal is initialized with the ballot definitions, and the election begins, voters are allowed to cast their votes. To get started, however, the voter must have a *voter card*. The voter card is a memory card or smartcard; i.e., it is a credit-card sized plastic card with a computer chip on it that can store data and, in the case of the smartcard, perform computation. We do not know exactly how the voter gets his voter card. It could be sent in the mail with information about where to vote, or it could be given out at the voting site on the day of the election. To understand the voting software itself, however, we do not need to know what process is used to distribute the cards to voters.

The voter takes the voter card and inserts it into a smartcard reader attached to the voting terminal. The terminal checks that the smartcard in its reader is a voter card and, if it is, presents a ballot to the voter on the terminal screen. The actual ballot the voter sees may depend on the voter's political party, which is encoded on the voter card. If a ballot cannot be found for the voter's party, the voter is given a nonpartisan ballot.

At this point, the voter interacts with the voting terminal, touching the appropriate boxes on the screen for his or her desired candidates. Headphones are available for visually-impaired voters to privately interact with the terminal. Before the ballots are committed to storage in the terminal, the voter is given a final chance to review his or her selections. If the voter confirms this, the vote is recorded on the voting terminal and the voter card is "canceled." This latter step is intended to prevent the voter from voting again with the same card. After the voter finishes voting, the terminal is ready for another voter to use.

REPORTING THE RESULTS. A poll worker ends the election process by inserting an *administrator card* or an *ender card* (a special card that can only be used to end the election) into the voting terminal. Upon detecting the presence of such a card (and, in the case of the administrator card, checking a PIN entered by the card user), the poll worker is asked to confirm that the election is finished. If the poll worker agrees, then the voting terminal enters the post-election stage and can transmit its results to the back-end server.

As we have only analyzed the code for the Diebold voting terminal, we do not know exactly how the back-end server tabulates the final results it gathers from the individual terminals. Obviously, it collects all the votes from the various voting terminals. We are unable to verify that there are checks to ensure, for example, that there are no more votes collected than people who are registered at or have entered any given polling location.

2.1 Details

We now describe the Diebold system in more detail, making explicit references to the relevant portions of the code. The voting terminal is implemented in the directory `BallotStation/`, but uses libraries in the supporting directories `Ballot/`, `DES/`, `DiagMode/`, `Shared/`, `TSElection/`, `Utilities/`, and `VoterCard/`.

The method `CBallotStationApp::DoRun()` is the main loop for the voting terminal software. The `DoRun()` method begins by invoking `CBallotStationApp::LoadRegistry()`, which loads information about the voting terminal from the registry (the registry keys are stored under `HKEY_LOCAL_`

MACHINE\Software\GlobalElectionSystems\AccuVote-TS4). If the program fails to load the registry information, it believes that it is uninitialized and therefore creates a new instance of the CTSRegistryDlg class that asks the administrator to set up the machine for the first time. The administrator chooses, among other things, the COM port to use with the smartcard reader, the directory locations to store files, and the polling location identifier. The CBallotStationApp::DoRun() method then checks for the presence of a smartcard reader and, if none found, gives the administrator the option to interact with the CTSRegistryDlg again.

The DoRun() method then enters a while loop that iterates until the software is shut down. The first thing DoRun() does in this loop is check for the presence of some removable media on which to store election results and ballot configurations (a floppy under Windows or a removable storage card on a Windows CE device). It then tries to open the election configuration file election.edb. If it fails to open the configuration file, the program enters the CTSElectionDoc::ES_NOELECTION state and invokes CBallotStationApp::Download(), which creates an instance of CTransferElecDlg to download the configuration file. To do the download, the terminal connects to a back-end server using either the Internet or a dial-up connection. The program then enters the CTSElectionDoc::ES_PREELECT state, invokes the CBallotStationApp::PreElect() method, which in turn creates an instance of CPreElectDlg. The administrator can then decide to start the election, in which case the method CPreElectDlg::OnSetForElection() sets the state of the terminal to CTSElectionDoc::ES_ELECTION.

Returning to the while loop in CBallotStationApp::DoRun(), now that the machine is in the state CTSElectionDoc::ES_ELECTION, the DoRun() method invokes CBallotStationApp::Election(), which creates an instance of CVoteDlg. When a card in inserted into the reader, the reader checks to see if the card is a voter card, administrator card, or ender card. If it is an ender card, or if it is an administrator card and if the user enters the correct PIN, the CVoteDlg ends and the user is asked whether he wishes to terminate the election and, if so, the state of the terminal is set to CTSElectionDoc::ES_POSTELECT. If the user entered a voter card, then DoVote() is invoked (here DoVote() is an actual function; it does not belong to any class). The DoVote() function finds the appropriate ballot for the user's voter group or, if none exists, opens the nonpartisan ballot (recall that the system is designed to allow voters to vote by group or party). It then creates an instance of CBallotDlg to display the ballot and collect the votes.

We recall that if, during the election process, someone inserted an administrator or ender card into the terminal and choses to end the election, the system would enter the CTSElectionDoc::ES_POSTELECT state. At this point the voting terminal would offer the ability to upload the election results to some back-end server for final tabulation. The actual transfer of results is handled by the CTransferResultsDlg::OnTransfer() method.

We will present more details about the Diebold system in the following sections.

3 Smartcards

While it is true that one can design secure systems around the use of smartcards, simply the use of smartcards in a system does *not* imply that the system is secure. The system must use the smartcards in an intelligent and security-conscious way. Unfortunately, the Diebold system's use of smartcards provides very little (if any) additional security and, in fact, opens the system to several attacks.

3.1 Homebrew smartcards

Upon reviewing the Diebold code, we observed that the smartcards do not perform any cryptographic operations.[3] For example, authentication of the terminal to the smartcard is done "the old-fashioned way:" the terminal sends a cleartext (i.e., unencrypted) 8-byte password to the card and, if the password is correct, the card believes that it is talking to a legitimate voting terminal. Unfortunately, this method of authentication is insecure: an attacker can easily learn the 8-byte password used to authenticate the terminal to the card (see Section 3.3), and thereby communicate with a legitimate smartcard using his own smartcard reader.

Furthermore, there is no authentication of the smartcard to the device. This means that nothing prevents an attacker from using his own homebrew smartcard in a voting terminal. One might naturally wonder how easy it would be for an attacker to make such a homebrew smartcard. First, we note that many smartcard vendors sell cards that can be programmed by the user. An attacker who knows the protocol spoken by the voting terminal to the legitimate smartcard could easily implement a homebrew card that speaks the same protocol. Even if the attacker does not *a priori* know the protocol, an attacker could easily learn enough about the protocol to create new voter cards by attaching a "wiretap" device between the voting terminal and a legitimate smartcard and observing the communicated messages. The parts for building such a device are readily available and, given the privacy of voting booths, might be unlikely to be noticed by poll workers. An attacker might not even need to use a wiretap to see the protocol in use. Smartcards generally use the ISO 7816 standard smartcard message formats. Likewise, the important data on the legitimate voting card is stored as a file (named `0x3D40` — smartcard files have numbers instead of textual file name) that can be easily read by a portable smartcard reader. Again, given the privacy of voting booths, an attacker using such a card reader would be unlikely to be noticed. Given the ease with which an attacker can interact with legitimate smartcards, plus the weak password-based authentication scheme (see Section 3.3), an attacker could quickly gain enough insight to create homebrew voting cards, perhaps quickly enough to be able to use such homebrew cards during the same election day.

The only impediment to the mass production of homebrew smartcards is that each voting terminal will make sure that the smartcard has encoded in it the correct `m_ElectionKey`, `m_VCenter`, and `m_DLVersion` (see `DoVote()` in `BallotStation/Vote.cpp`). The `m_ElectionKey` and `m_DLVersion` are likely the same for all locations and, furthermore, for backward-compatibility purposes it is possible to use a card with `m_ElectionKey` and `m_DLVersion` undefined. The `m_VCenter` value could be learned on a per-location-basis by interacting with legitimate smartcards, from an insider, or from inferences based on the `m_VCenter` values observed at other polling locations.

It is worth pointing out that both the smartcards and readers supported by the code are available commercially over the Internet in small quantities. The smartcards, model number CLXSU004KC0, are available from CardLogix[4], where $33.50 buys ten cards. General purpose CyberFlex JavaCards from Schlumberger cost $110 for five cards[5]. Smartcard reader/writers are available in a variety of models for under $100 from many vendors.

In the following two subsections we illustrate what an attacker could accomplish using homebrew smartcards in voting terminals running the Diebold code.

[3]This, in an of itself, is an immediate red-flag. One of the biggest advantages of smartcards over classic magnetic-stripe cards is the smartcard's ability to perform cryptographic operations internally with physically protected keys. Since the smartcards in the Diebold system do not perform any cryptographic operations, they effectively provide no more security than traditional magnetic-stripe cards.

[4]`http://cardlogix.com/product_smart_select_most.asp`

[5]`http://www.scmegastore.com`

3.1.1 Multiple voting

In the Diebold system, a voter begins the voting process by inserting a smartcard into the voting terminal. Upon checking that the card is "active," the voting terminal collects the user's vote and then deactivates the user's card (the deactivation actually occurs by changing the card's type, which is stored as an 8-bit byte on the card, from `VOTER_CARD` (`0x01`) to `CANCELED_CARD` (`0x08`)). Since an adversary can make perfectly valid smartcards, the adversary could bring a stack of active cards to the voting booth. Doing so gives the adversary the ability to vote multiple times. More simply, instead of bringing multiple cards to the voting booth, the adversary could program a smartcard to ignore the voting terminal's deactivation command. Such an adversary could use one card to vote multiple times.

Will the adversary's multiple-votes be detected by the voting system? To answer this question, we must first consider what information is encoded on the voter cards on a per-voter basis. The only per-voter information is a "voter serial number" (`m_VoterSN` in the `CVoterInfo` class). Because of the way the Diebold system works, `m_VoterSN` is only recorded by the voting terminal if the voter decides *not* to place a vote (as noted in the comments in `TSElection/Results.cpp`, this field is recorded for uncounted votes for backward compatibility reasons). It is important to note that if a voter decides to cancel his or her vote, the voter will have the opportunity to vote again using that same card (and, after the vote has been cast, `m_VoterSN` will not be recorded).

Can the back-end tabulation system detect multiple-vote casting? If we assume the number of collected votes becomes greater than the number of people who showed up to vote, and if the polling locations keep accurate counts of the number of people who show up to vote, then the back-end system, if designed properly, should be able to detect the existence of counterfeit votes. However, because `m_VoterSN` is only stored for those who did not vote, there will be no way for the tabulating system to count the true number of voters or distinguish the real votes from the counterfeit votes. This would cast serious doubt on the validity of the election results. We point out, however, that we only analyzed the voting terminal's code; we do not know whether such checks are performed in the actual back-end tabulating system.

3.1.2 Administrator and ender cards

As noted in Section 2, in addition to the voter cards that users use when they vote, there are also administrator cards and ender cards. These cards have special purposes in this system. Namely, the administrator cards give the possessor the ability to access administrative functionality (namely, the administrative dialog `BallotStation/AdminDlg.cpp`), and both types of cards allow the possessor to end the election (hence the term "ender card").

Just as an adversary can manufacture his or her own voter cards, an adversary can manufacture his or her own administrator and ender cards (administrator cards have an easily-circumventable PIN, which we will discuss in Section 3.2). This attack is easiest if the attacker has knowledge of the Diebold code or can interact with a legitimate administrator or ender card. However, an attacker without knowledge of the inner-workings of the Diebold system and without the ability to interact directly with a legitimate administrator or ender card may have a difficult time producing a homebrew administrator or ender card since the attacker would not know what distinguishes an administrator or ender card from a normal voter card. (The distinction is that, for a voter card `m_CardType` is set to `0x01`, for an ender card `m_CardType` is set to `0x02`, and for an administrator card `m_CardType` is set to `0x04`.)

As one might expect, an adversary in possession of such illicit cards has further attack options against the Diebold system. Using a homebrew administrator card, a poll worker, who might not otherwise have access to the administrator functions of the Diebold system but who does have access to the voting machines before and after the elections, could gain access to the administrator controls. If a malicious voter entered an administrator or ender card into the voting device instead of the normal voter card, then the voter would be able to terminate the election and, if the card is an administrator card, gain access to additional administrative controls.

The use of administrator or ender cards prior to the completion of the actual election represents an interesting denial-of-service attack. Once “ended,” the voting terminal will no longer accept new voters (see `CVoteDlg::OnCardIn()`) until the terminal is somehow reset. Such an attack, if mounted simultaneously by multiple people, could shut down a polling place. If a polling place is in a precinct considered to favor one candidate over another, attacking that specific polling place could benefit the less-favored candidate. Even if the poll workers were later able to resurrect the systems, the attack might succeed in deterring a large number of potential voters from voting (e.g., if the attack was performed over the lunch hour). If such an attack was mounted, one might think the attackers would be identified and caught. We note that many governmental entities do not require identification to be presented by a voter, instead allowing for “provisional” ballots to be cast. By the time the poll workers realize that one of their voting terminals has been disabled, the perpetrator may have long-since left the scene.

3.2 Unprotected PINs

When a user enters an administrator card into a voting terminal, the voting terminal asks the user to enter a 4-digit PIN. If the correct PIN is entered, the user is given access to the administrative controls.

Upon looking more closely at this administrator authentication process, however, we see that there is a flaw with the way the PINs are verified. When the terminal and the smartcard first begin communicating, the PIN value stored on the card is sent in cleartext from the card to the voting terminal. Then, when the user enters the PIN into the terminal, it is compared with the PIN that the smartcard sent (`CPinDlg::OnOK()`). If these values are equal, the system accepts the PIN. Herein lies the flaw with this design: any person with a smartcard reader can easily extract the PIN from an administrator card. The adversary doesn’t even need to fully understand the protocol between the terminal and the device: if the response from the card is n bytes long, the attacker who correctly guesses that the PIN is sent in the clear would only have to try $n-3$ possible PINs, rather than 10,000. This means that the PINs are easily circumventable. Of course, if the adversary knows the protocol between the card and the device, an adversary could just make his own administrator card, using any desired PIN (Section 3.1.2).

3.3 Terminal-to-card authentication

Let us now consider how the terminal authenticates to the card in more detail (note that this is different from the Administrator PIN; here the terminal is being authenticated rather than the user). The relevant code from `VoterCard/CLXSmartCard.cpp` is listed below. The code is slightly modified for illustrative purposes, but the developers’ comments have been left intact.

```
SMC_ERROR CCLXSmartCard::Open(CCardReader* pReader)
{
  ... [removed code] ...

  // Now initiate access to the card
  // If failed to access the file then have unknown card
  if (SelectFile(0x3d40) != SMC_OK)
    st = SMC_UNKNOWNCARD;
  // Else if our password works then all done
  else if (Verify(0x80, 8, {0xed, 0x0a, 0xed, 0x0a, 0xed, 0x0a, 0xed, 0x0a})
    == SMC_OK)
    st = SMC_OK;
  // Else if manufactures password works then try to change password
  else if(Verify(0x80, 8, {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08})
    == SMC_OK) {
    st = ChangeCode(8, {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
                     8, {0xed, 0x0a, 0xed, 0x0a, 0xed, 0x0a, 0xed, 0x0a});
    // Else have a bad card
  } else
    st = SMC_BADCARD;
  return st;
}
```

In the above code, the `SelectFile(0x3d40)` command sends the bytes `0x00`, `0xA4`, `0x00`, `0x00`, `0x02`, `0x3D`, `0x40`, `0x00` to the smartcard in cleartext; this is the ISO 7816 smartcard command for selecting a file on the card (`0x3D40` in this case). If `passwd` is a sequence of 8 bytes, the `Verify(0x80, 8,passwd)` command sends the bytes `0x00`, `0x20` `0x00`, `0x80`, `0x08`, `passwd[0]`, `passwd[1]`, ..., `passwd[7]`, `0x00` to the smartcard in cleartext.

There are several issues with the above code. First, hard-coding passwords in C++ files is generally a poor design choice. We will discuss coding practices in more detail in Section 6, but we summarize some issues here. Hard-coding passwords into C++ files suggests a lack of key and password management. Furthermore, even if the developers assumed that the passwords would be manually changed and the software recompiled on a per-election basis, it would be very easy for someone to forget to change the constants in `VoterCard/CLXSmartCard.cpp`. (Recompiling on a per-election basis may also be a concern, since good software engineering practices would dictate additional testing and certification if the code were to be recompiled for each election.)

The above issues would only be a concern if the authentication method were otherwise secure. Unfortunately, it is not. Since the password is sent in the clear from the terminal to the card, an attacker who puts a fake card into the terminal and records the command from the terminal will be able to learn the password (and file name) and then re-use that password with real cards. An adversary with knowledge of this password could then create counterfeit voting cards. As we have already discussed (see Section 3.1.1), this can allow the adversary to cast multiple votes, among other attacks. Hence, the authentication of the voting terminal to the smartcards is insecure. We find this particularly surprising because modern smartcard designs allow cryptographic operations to be performed directly on the smartcard, making it possible to create systems that are not as easily vulnerable to security breaches.

Furthermore, note the control flow in the above code-snippet. If the password chosen by the designers of the system ("`\xED\x0A\xED\x0A\xED\x0A\xED\x0A`") does not work, then `CCLXSmartCard::Open()` uses the smartcard manufacturer's default password[6] of "`\x00\x01\x02\x03\x04\x05\x06\x07`." One issue with this is that it implies that sometimes the system is used with un-initialized smartcards. This means that an attacker might not even need to figure out the system's password in order to be able to authenticate to the cards. As we noted in Section 3.1, some smartcards allow a user to get a listing of all the files on a card. If the system uses such a card and also uses the manufacturer's default password of `\x00\x01\x02\x03\x04\x05\x06\x07`, then an attacker, even without any knowledge of the source code and without the ability to intercept the connection between a legitimate card and a voting terminal, but with access to a legitimate voter card, will still be able to learn enough about the smartcards to be able to create counterfeit voter cards.[7]

[6]Many smartcards are shipped with default passwords.

[7]Making homebrew cards this way is somewhat risky, as the attacker must make assumptions about the system. In particular, the attacker is assuming that his or her counterfeit cards would not be detected by the voting terminals. Without access to the source code, the attacker would never know, without testing, whether it was truly safe to attack a voting terminal.

4 Data storage

While the data stored internally on each voting terminal is not as accessible to an attacker as the voting system's smartcards, exploiting such information presents a powerful attack vector, especially for an election insider. In this section we outline the data storage available to each terminal and then detail how each distinct type of data is stored.

4.1 Data storage overview

Each voting terminal has two distinct types of internal data storage. A main (or system) storage area contains the terminal's operating system, program executables, static data files such as fonts, and system configuration information, as well as backup copies of dynamic data files such as the voting records and audit logs. Each terminal also contains a removable storage device that is used to store the primary copies of these dynamic data files. When the terminal is running a standard copy of Windows (e.g., Windows 2000) the removable storage area is the first floppy drive; when the terminal is running Windows CE the removable storage area is a removable storage card. Storing the dynamic data on two distinct devices is advantageous for both reliability and security: if either of the two storage mediums fails, data can still be recovered from the copy.

Unfortunately, under Windows CE, which we believe is used in commercial Diebold voting terminals, the existence of the removable storage device is not enforced properly. Unlike other versions of Windows, removable storage cards are mounted as subdirectories under CE. When the voting software wants to know if a storage card is inserted, it simply checks to see if the `Storage Card` subdirectory exists in the filesystem's root directory. While this is the default name for a mounted storage device, it is also a perfectly legitimate directory name for a directory in the main storage area. Thus, if such a directory exists, the terminal can be fooled into using the same storage device for all of the data.[8] This would reduce the amount of redundancy in the voting system and would increase the chances that a hardware fault could cause recorded votes to be lost.

4.2 System configuration

The majority of the system configuration information for each terminal is stored in the Windows registry under `HKEY_LOCAL_MACHINE\Software\GlobalElectionSystems\AccuVote-TS4`. This includes both identification information such as the terminal's serial number and more traditional configuration information such as the COM port that the smartcard reader is attached to. All of the configuration information is stored in the clear, without any form of integrity protection. Thus, all an adversary must do is modify the system registry to trick a given voting terminal into effectively impersonating any other voting terminal. It is unclear how the tallying authority would deal with results from two different voting terminals with the same voting ID — at the very least human intervention to resolve the conflict would probably be required.

The Federal Election Commission draft standard[9] requires each terminal to keep track of the total number of votes that have ever been cast on it — the "Protective Counter." This counter is used to provide yet another method for ensuring that the number of votes cast on each terminal is correct. However, as the following code from `Utilities/machine.cpp` shows, the counter is simply stored as an integer in the file `system.bin` in the terminal's system directory (error handling code has been removed for clarity):

[8]This situation can be easily corrected by checking for the `FILE_ATTRIBUTE_TEMPORARY` attribute on the directory as described in `http://msdn.microsoft.com/library/en-us/wcefiles/htm/_wcesdk_Accessing_Files_on_Other_Storage_Media.asp`

[9]`http://fecweb1.fec.gov/pages/vss/vss.html`

```
long GetProtectedCounter()
{
  DWORD protectedCounter = 0;
  CString filename = ::GetSysDir();
  filename += _T("system.bin");
  CFile file;
  file.Open(filename, CFile::modeRead | CFile::modeCreate |
                         CFile::modeNoTruncate);
  file.Read(&protectedCounter, sizeof(protectedCounter));
  file.Close();
  return protectedCounter;
}
```

By modifying this counter, an adversary could cast doubt on an election by creating a discrepancy between the number of votes cast on a given terminal and the number of votes that are tallied in the election. While the current method of implementing the counter is totally insecure, even a cryptographic checksum would not be enough to protect the counter; an adversary with the ability to modify and view the counter would still be able to roll it back to a previous state. In fact, the only solution that would work would be to implement the protective counter in a tamper-resistant hardware token, requiring modifications to the physical voting terminal hardware.

4.3 Ballot definition

The "ballot definition" for each election contains everything from the background color of the screen to the PPP username and password to use when reporting the results. This data is not encrypted or checksummed (cryptographically or otherwise) and so can be easily modified by any attacker with physical access to the file. While many attacks, such as changing the party affiliation of a candidate, would be noticed by some voters, many more subtle attacks are possible. By simply changing the order of the candidates as they appear in the ballot definition, the results file will change accordingly. However, the candidate information itself is not stored in the results file. The file merely tracks that candidate 1 got so many votes and candidate 2 got so many other votes. If an attacker reordered the candidates on the ballot definition, voters would unwittingly cast their ballots for the wrong candidate. As with denial-of-service attacks (see Section 3.1.2), ballot reordering attacks would be particularly effective in polling locations known to be heavily partisan.

Even without modifying the ballot definition, an attacker can gain almost enough information to impersonate the voting terminal to the back-end server. The terminal's voting center ID, PPP dial-in number, username, password and the IP address of the back-end server are all available in the clear (these are parsed into a `CElectionHeaderItem` in `TSElection\TSElectionObj.cpp`). Assuming an attacker is able to guess or create a voting terminal ID, he would be able to transmit fraudulent vote reports to the back-end server by dialing in from his own computer. While both the paper trail and data stored on legitimate terminals could be used to compensate for this attack after the fact, it could, at the very least, delay the election results. (The PPP number, username, password, and IP address of the back-end server are also stored in the registry `HKEY_LOCAL_MACHINE\Software\GlobalElectionSystems\AccuVote-TS4\TransferParams`. Since the ballot definition may be transported on portable memory cards or floppy disks, the ballot definition may perhaps be easier to obtain from this distribution media rather than from the voting terminal's internal data storage.)

We will return to some of these points in Section 5.1, where we show that modifying and viewing ballot definition files does not always require physical access to the terminals on which they are stored.

4.4 Votes and audit logs

Unlike the other data stored on the voting terminal, both the vote records and the audit logs are encrypted and checksummed before being written to the storage device. Unfortunately, neither the encrypting nor the checksumming is done securely.

All of the data on a storage device is encrypted using a single, hardcoded DES [NBS77] key:

```
#define DESKEY ((des_key*)"F2654hD4")
```

Note that this value is not a hex representation of a key. Instead, the bytes in the string "`F2654hD4`" are fed directly into the DES key scheduler. If the same binary is used on every voting terminal, an attacker with access to the source code, or even to a single binary image, could learn the key, and thus read and modify voting and auditing records. Even if proper key management were to be implemented, many problems would still remain. First, DES keys can be recovered by brute force in a very short time period [Gil98]. DES should be replaced with either triple-DES [Sch96] or, preferably, AES [DJ02]. Second, DES is being used in CBC mode which requires an initialization vector to ensure its security. The implementation here always uses zero for its IV. This is illustrated by the call to `DesCBCEncrypt` in `TSElection/RecordFile.cpp`; since the second to last argument is NULL, `DesCBCEncrypt` will use the all-zero IV.

```
DesCBCEncrypt((des_c_block*)tmp, (des_c_block*)record.m_Data, totalSize,
               DESKEY, NULL, DES_ENCRYPT);
```

This allows an attacker to mount a variety of cryptanalytic attacks on the data. To correctly implement CBC mode, a source of "strong" random numbers must be used to generate a fresh IV for each encryption [BDJR97]. Suitably strong random numbers can be derived from many different sources, ranging from custom hardware through observation of user behavior. (Jones reports that the vendor may have been aware of this design flaw in their code for several years [Jon01, Jon03]. We see no evidence that this design flaw was ever addressed.)

Before being encrypted, a 16-bit cyclic redundancy check (CRC) of the plaintext data is computed. This CRC is then stored along with the ciphertext in the file and verified whenever the data is decrypted and read. This process in handled by the `ReadRecord` and `WriteRecord` functions in `TSElection/RecordFile.cpp`. Since the CRC is an unkeyed, public function, it does not provide any real integrity for the data. In fact, by storing it in an unencrypted form, the purpose of encrypting the data in the first place (leaking no information about the contents of the plaintext) is undermined. A much more secure design would be to first encrypt the data to be stored and then to compute a keyed cryptographic checksum (such as HMAC-SHA1 [BCK96]) of the ciphertext. This cryptographic checksum could then be used to detect any tampering with the plaintexts. Note also that each entry has a timestamp, which will prevent the re-ordering, though not deletion, of records.

Each entry in a plaintext audit log is simply a timestamped, informational text string. At the time that the logging occurs, the log can also be printed to an attached printer. If the printer is unplugged, off, or malfunctioning, however, no record will be stored elsewhere to indicate that the failure occurred. The following code from `TSElection/Audit.cpp` demonstrates that the designers failed to consider these issues:

```
if (m_Print && print) {
CPrinter printer;
// If failed to open printer then just return.
CString name = ::GetPrinterPort();
if (name.Find(_T("\\")) != -1)
  name = GetParentDir(name) + _T("audit.log");
if (!printer.Open(name, ::GetPrintReverse(), FALSE))
  ::TSMessageBox(_T("Failed to open printer for logging"));
} else {
```

Do the printing…

```
}
```

If the cable attaching the printer to the terminal is exposed, an attacker could create discrepancies between the printed log and the log stored on the terminal by unplugging the printer (or, by simply cutting the cable).

An attacker's most likely target will be the voting records, themselves. Each voter's votes are stored as a bit array based on the ordering in the ballot definition file along with other information such as the precinct the voter was in, although no information that can be linked to a voter's identity is included. If the voter has chosen a write-in candidate, this information is also included as an ASCII string. An attacker given access to this file would be able to generate as many fake votes as he or she pleased, and such votes would be indistinguishable from the true votes cast on the terminal.

While the voter's identity is not stored with the votes, each vote is given a serial number. These serial numbers are generated by a linear congruential random number generator (LCG), seeded with static information about the election and voting terminal. No dynamic information, such as the current time, is used.

```
// LCG - Linear Conguential Generator - used to generate ballot serial numbers
// A psuedo-random-sequence generator
// (per Applied Cryptography, by Bruce Schneier, Wiley, 1996)
#define LCG_MULTIPLIER 1366
#define LCG_INCREMENTOR 150889
#define LCG_PERIOD 714025

static inline int lcgGenerator(int lastSN)
{
  return ::mod(((lastSN * LCG_MULTIPLIER) + LCG_INCREMENTOR), LCG_PERIOD);
}
```

While the code's authors apparently decided to use an LCG because it appeared in *Applied Cryptography* [Sch96], LCG's are far from secure. However, attacking this random number generator is unnecessary for determining the order in which votes were cast: each vote is written to the file sequentially. Thus, if an attacker is able to determine the order in which voters cast their ballots, the results file has a nice list, in the order in which voters used the terminal. A malevolent poll worker, for example, could surreptitiously track the order in which voters use the voting terminals. Later, in collaboration with other attackers who might intercept the poorly encrypted voting records, the exact voting record of each voter could be reconstructed. Physical access to the voting results may not even be necessary to acquire the voting records, if they are transmitted across the Internet.

5 Communicating with the outside world

The Diebold voting machines cannot work in isolation. They must be able to both receive a ballot definition file as input and report voting results as output. As described in Section 2, there are essentially two ways to load a voting terminal with an initial election configuration: via some removable media, such as a floppy disk, or over the Internet. In the latter case, the voting terminal could either be plugged directly into the Internet or could use a dial-up connection (the dial-up connection could be to a local ISP, or directly to the election authority's modem banks). After the election is over, election results are sent to a back-end post-processing server over the network (again, possibly through a dial-up connection).

Unfortunately, there are a number of attacks against this system that exploit the system's reliance on and communication with the outside world.

5.1 Tampering with ballot definitions

We first note that it is possible for an adversary to tamper with the voting terminals' ballot definition file (`election.edb`). If the voting terminals load the ballot definition from a floppy or removable storage card, then an adversary, such as a poll worker, could tamper with the contents of the floppy before inserting it into the voting terminal. On a potentially much larger scale, if the voting terminals download the ballot definition from the Internet, then an adversary could tamper with the ballot definition file en-route from the back-end server to the voting terminal. With respect to the latter, we point out that the adversary need not be an election insider; the adversary could, for example, be someone working at the local ISP. If a wireless network is used, anybody within radio range becomes a potential adversary. With high-gain antennas, the adversary can be sufficiently distant to have little risk of detection. If the adversary knows the structure of the ballot definition, then the adversary can intercept and modify the ballot definition while it is being transmitted. Even if the adversary does not know the precise structure of the ballot definition, many of the fields inside are easy to identify and change, including the candidates' names, which appear as plain ASCII text.[10]

Let us now consider some example attacks that make use of modifying the ballot definition file. Because no cryptographic techniques are in place to guard the integrity of the ballot definition file, an attacker could add, remove, or change issues on the ballot, and thereby confuse the result of the election.

In the system, different voters can be presented with different ballots depending on their party affiliations (see `CBallotRelSet::Open()`, which adds different issues to the ballot depending on the voter's `m_VGroup1` and `m_VGroup2` `CVoterInfo` fields). If an attacker changes the party affiliations of the candidates, then he may succeed in forcing the voters to view and vote on erroneous ballots. Even in municipalities that use the same ballot for all voters, a common option in voting systems is to allow a voter to select all the candidates from a given political party. Voters may not notice if candidates have incorrect party affiliations listed next to their names, and by choosing to vote a straight ticket, would end up casting ballots for undesirable candidates.

As an example of what might happen if the party affiliations were listed incorrectly, we note that, according to a news story[11], in the 2000 New Mexico presidential election, over 65,000 votes were incorrectly counted because a worker accidentally had the party affiliations wrong. (We are not claiming, however, that those affiliations were maliciously assigned incorrectly. Nor are we implying that this had an effect on the results of the election.)

Likewise, an attacker who can change the ballot definition could also change the ordering of the candidates running for a particular office. Since, at the end of the election, the results are uploaded to the server in the order that they appear in the the ballot definition file, and since the server will believe that the results appear in their original order, this attack could also succeed in swapping the votes between parties in a predominantly partisan precinct. This ballot reordering attack is also discussed in more detail in Section 4.3.

5.2 Preventing the start of an election

Suppose that the election officials are planning to download the configuration files over the Internet and that they are running late and do not have much time before the election starts to distribute ballot definitions manually (i.e., they might not have enough time to distribute physical media with the ballot definition files from central office to every voting precinct). In such a situation, an adversary could mount a traditional

[10]The relevant sections of code are as follows: `CTransferElecDlg::OnTransfer()` invokes `CTSElectionDoc::CreateEDB()`, which reads the data from a `CDL2Archive`. The way the code works, `CDL2Archive` reads and writes to a `CBufferedSocketFile`. Returning to `CTSElectionDoc::CreateEDB()`, we see that it invokes `CTSElectionDB::Create()`, which subsequently invokes `CTSElectionFile::Save()`. The functions called in `CTSElectionFile::Save()` read data, such as strings, from the `CDL2Archive`.

[11]`http://www.gcn.com/vol19_no33/news/3307-1.html`

Internet denial-of-service attack against the election management's server and thereby prevent the voting terminals from acquiring their ballot definitions before the start of the election. To mount such an attack effectively, the adversary would ideally need to know the topology of the system's network, and the name of the server(s) supplying the ballot definition file.[12] If a fair number of people from a certain demographic plan to vote early in the morning, then this could impact the results of the election.

Of course, we acknowledge that there are other ways to postpone the start of an election at a voting location that do not depend on the use of this system (e.g., flat tires for all poll workers for a given precinct). Unlike such traditional attacks, however, the network-based attack (1) is relatively easy for anyone with knowledge of the election system's network topology to accomplish; (2) this attack can be performed on a very large scale, as the central distribution point(s) for ballot definitions becomes an effective single point of failure; and (3) the attacker can be physically located anywhere in the Internet-connected world, complicating efforts to apprehend the attacker. Such attacks could prevent or delay the start of an election at all voting locations in a state. We note that this attack is not restricted to the system we analyzed; it is applicable to any system that downloads its ballot definition files using the Internet.

5.3 Tampering with election results

Just as it is possible for an adversary to tamper with the downloading of the ballot definition file (Section 5.1), it is also possible for an adversary to tamper with the uploading of the election results. To make this task even easier for the adversary, we note that although the election results are stored "encrypted" on the voting devices (Section 4.4), the results are sent from the voting devices to the back-end server over an unauthenticated and unencrypted channel. In particular, `CTransferResultsDlg::OnTransfer()` writes ballot results to an instance of `CDL2Archive`, which then writes the votes in cleartext to a socket without any cryptographic checksum.

Sending election results in this way over the Internet is a bad idea. Nothing prevents an attacker with access to the network traffic, such as workers at a local ISP, from modifying the data in transit. Such an attacker could, for example, decrease one candidates vote count by n and increase the another candidate's count by n. Of course, to introduce controlled changes to the votes, the attacker would require some knowledge of the structure of the messages sent from the voting terminals to the back-end server. If the voting terminals use a modem connection directly to the tabulating authority's network, rather than the Internet, then the risk of such an attack is less, although still not inconsequential. A sophisticated adversary (or employee of the local phone company) could tap the phone line and intercept the communication. All of these adversaries could be easily defeated by properly using standard encryption suites like SSL/TLS, used throughout the World Wide Web for e-commerce security. We are puzzled why such a widely accepted and studied technology is not used by the voting terminals to safely communicate across potentially hostile networks.

5.4 Attacking the voting terminals directly

In some configurations, where the voting terminals are directly connected to the Internet, it may be possible for an adversary to attack them directly, perhaps using an operating system exploit or buffer overflow attack of some kind. Ideally the voting devices and their associated firewalls would be configured to accept no incoming connections [CBR03]. This concern would apply to any voting terminal, from any vendor, with a direct Internet connection.

[12] Knowledge of the specific servers is unnecessary for a large scale distributed denial of service attack. An attacker simply needs to correctly guess that the ballot definition's file server is on a specific network (e.g., at the voting system's vendor or in the municipality's government).

6 Software engineering

When creating a secure system, getting the design right is only part of the battle. The design must then be securely implemented. We now examine the coding practices and implementation style used to create the voting device. This type of analysis can offer insights into future versions of the code. For example, if a current implementation has followed good implementation practices but is simply incomplete, one would be more inclined to believe that future, more complete versions of the code would be of a similar high quality. Of course, the opposite is also true, perhaps even more so: it is very difficult to produce a secure system by building on an insecure foundation.

Of course, reading the source code to a product gives only an incomplete view into the actions and intentions of the developers who created that code. Regardless, we can see the overall software design, we can read the comments in the code, and thanks to the CVS repository, we can even look at earlier versions of the code and read the developers' commentary as they committed their changes to the archive.

6.1 Code Legacy

Inside `cvs.tar` we found multiple CVS archives. Two of the archives, `AccuTouch` and `AVTSCE` implement full voting terminals. The `AccuTouch` code dates to around 2000 and is copyrighted by "Global Election Systems, Inc." while the `AVTSCE` code dates to mid-2002 and is copyrighted by "Diebold Election Systems, Inc." (The CVS logs show that the copyright notice was updated on February 26, 2002.) Many files are nearly identical between the two systems and the overall design appears very similar. Indeed, Diebold acquired Global Election Systems in September, 2001.[13] Some of the code, such as the functions to compute CRCs and DES, dates back to 1996, when Global Election Systems was called "I-Mark Systems."

This legacy is apparent in the code itself as there are portions of the `AVTSCE` code, including entire classes, that are either simply not used or removed through the use of `#ifdef` statements. Many of these functions are either incomplete or, worse, do not perform the function that they imply as is the case with `CompareFiles` in `Utilities/FileUtil.cpp`:

```
BOOL CompareFiles(const CString& file1, const CString& file2)
{
  /* XXX use a CRC or something similar */
  BOOL exists1, exists2;
  HANDLE hFind;
  WIN32_FIND_DATA fd1, fd2;

  exists1 = ((hFind = ::FindFirstFile(file1, &fd1)) != INVALID_HANDLE_VALUE);
  ::FindClose(hFind);

  exists2 = ((hFind = ::FindFirstFile(file2, &fd2)) != INVALID_HANDLE_VALUE);
  ::FindClose(hFind);

  return (exists1 && exists2 && fd1.nFileSizeLow == fd2.nFileSizeLow);
}
```

Currently the code will declare any two files to be the same that have the same size. The author's comment to use a CRC doesn't make much sense, as a byte-by-byte comparison would be more efficient. If this code were ever used, its inaccuracies could lead to wide variety of subsequent errors.

While most of the preprocessor directives that remove code correctly use `#if 0` as their condition, some use `#ifdef XXX`. There is no reason that a later programmer should realize that defining `XXX` will cause blocks of code to be reincluded in the system (causing unpredictable results, at best). We also noticed

[13]http://dallas.bizjournals.com/dallas/stories/2001/09/10/daily2.html

#ifdef LOUISIANA in the code. Prudent software engineering would recommend a single implementation of the voting software, where individual states or municipalities could have their desired custom features expressed in configuration files.

6.2 Coding style

While the system is implemented in an unsafe language (C++), the code reflects an awareness of avoiding such common hazards as buffer overflows. Most string operations already use their safe equivalents, and there are comments reminding the developers to change others (e.g., `should really use _snprintf`). While we are not prepared to claim that there are no buffer overflows in the current code, there are at the very least no glaringly obvious ones. Of course, a better solution would have been to write the entire system in a safe language, such as Java or C#.

The core concepts of object oriented programming such as encapsulation are well represented, though in some places C++'s non-typesafe nature is exploited with casts that could conceivably fail. This could cause problems in the future as these locations are not well documented.

Overall, the code is rather unevenly commented. While most files have a description of their overall function, the meanings of individual functions, their arguments, and the algorithms within are more often than not undocumented. An extreme example of a complex but completely undocumented function is the `CBallotRelSet::Open` function from `TSElection/TSElectionSet.cpp`:

```
void CBallotRelSet::Open(const CDistrict* district, const CBaseunit* baseunit,
                                      const CVGroup* vgroup1, const CVGroup* vgroup2)
{
   ASSERT(m_pDB != NULL);
   ASSERT(m_pDB->IsOpen());
   ASSERT(GetSize() == 0);
   ASSERT(district != NULL);
   ASSERT(baseunit != NULL);

   if (district->KeyId() == -1) {
      Open(baseunit, vgroup1);
   } else {
      const CDistrictItem* pDistrictItem = m_pDB->Find(*district);
      if (pDistrictItem != NULL) {
         const CBaseunitKeyTable& baseunitTable = pDistrictItem->m_BaseunitKeyTable;
         int count = baseunitTable.GetSize();
         for (int i = 0; i < count; i++) {
            const CBaseunit& curBaseunit = baseunitTable.GetAt(i);
            if (baseunit->KeyId() == -1 || *baseunit == curBaseunit) {
               const CBallotRelationshipItem* pBalRelItem = NULL;
               while ((pBalRelItem = m_pDB->FindNextBalRel(curBaseunit, pBalRelItem))){
                  if (!vgroup1 || vgroup1->KeyId() == -1 ||
                    (*vgroup1 == pBalRelItem->m_VGroup1 && !vgroup2) ||
                    (vgroup2 && *vgroup2 == pBalRelItem->m_VGroup2 &&
                       *vgroup1 == pBalRelItem->m_VGroup1))
                     Add(pBalRelItem);
               }
            }
         }
         m_CurIndex = 0;
         m_Open = TRUE;
      }
   }
}
```

Nothing about this code makes its purpose readily apparent. Certainly, it has two nested loops and is doing all kinds of comparisons. Beyond that, most readers of the code would need to invest significant time to learn the meaning of the various names shown here.

6.3 Coding process

An important point to consider is how code is added to the system. From the CVS logs, we can see that most code updates are in response to specific bugs that needed to be fixed. There are numerous authors who have committed changes to the CVS tree, and the only evidence that we have found that the code undergoes any sort of review process comes from a single log comment: "Modify code to avoid multiple exit points to meet Wyle requirements." This could refer to Wyle Laboratories whose website claims that they provide all manner of testing services.[14]

There are also pieces of the voting system that come from third parties. Most obviously is the operating system, either Windows 2000 or Windows CE. Both of these OSes have had numerous security vulnerabilities and their source code is not available for examination to help rule out the possibility of future attacks. Besides the operating system, an audio library called "fmod" is used.[15] While the source to fmod is available with commercial licenses, unless the code is fully audited there is no proof that fmod itself does not contain a backdoor.

Due to the lack of comments, the legacy nature of the code, and the use of third-party code and operating systems, we believe that any sort of comprehensive, top-to-bottom code review would be nearly impossible. Not only does this increase the chances that bugs exist in the code, but it also implies that any of the coders could insert a malicious backdoor into the system. The current design deficiencies provide enough other attack vectors that such an explicit backdoor is not required to successfully attack the system. Regardless, even if the design problems are eventually rectified, the problems with the coding process may well remain intact.

6.4 Code completeness and correctness

While the code we studied implements a full system, the implementors have included extensive comments on the changes that would be necessary before the system should be considered complete. It is unclear whether the programmers actually intended to go back and remedy all of these issues as many of the comments existed, unchanged, for months, while other modifications took place around them. It is also unclear whether later version of AVTSCE were subsequently created. (Modification dates and locations are easily visible from the CVS logs.) These comments come in a number of varieties. For illustrative purposes, we have chosen to show a few such comments from the subsystem that plays audio prompts to visually-impaired voters.

- Notes on code reorganization:
  ```
  /* Okay, I don't like this one bit.  Its really tough to tell
  where m_AudioPlayer should live.  [...]  A reorganization might be
  in order here.  */
  ```

- Notes on parts of code that need cleaning up:
  ```
  /* This is a bit of a hack for now.  [...]  Calling from the timer
  message appears to work.  Solution is to always do a 1ms wait
  between audio clips.  */
  ```

[14] `http://www.wylelabs.com/te.html`
[15] `http://www.fmod.org/`

- Notes on bugs that need fixing:
  ```
  /* need to work on exception *caused by audio*.  I think they will
  currently result in double-fault.  */
  ```

There are, however, no comments that would suggest that the design will radically change from a security perspective. None of the security issues that have been discussed in this paper are pointed out or marked for correction. In fact, the only evidence at all that a redesign might at one point have been considered comes from outside the code: the Crypto++ library[16] is included in another CVS archive in `cvs.tar`. However, the library was added in September 2000 and was never used or updated. We infer that one of the developers may have thought that improving the cryptography would be useful, but then got distracted with other business.

7 Conclusions

Using publicly available source code, we performed an analysis of a voting machine. This code was apparently developed by a company that sells to states and other municipalities that use them in real elections. We found significant security flaws: voters can trivially cast multiple ballots with no built-in traceability, administrative functions can be performed by regular voters, and the threats posed by insiders such as poll workers, software developers, and even janitors, is even greater. Based on our analysis of the development environment, including change logs and comments, we believe that an appropriate level of programming discipline for a project such as this was not maintained. In fact, there appears to have been little quality control in the process.

For quite some time, voting equipment vendors have maintained that their systems are secure, and that the closed-source nature makes them even more secure. Our glimpse into the code of such a system reveals that there is little difference in the way code is developed for voting machines relative to other commercial endeavors. In fact, we believe that an open process would result in more careful development, as more scientists, software engineers, political activists, and others who value their democracy would be paying attention to the quality of the software that is used for their elections. (Of course, open source would not solve all of the problems with electronic elections. It is still important to verify somehow that the binaries running in the machine correspond to the source code and that the compilers used on the source code are non-malicious. However, open source is a good start.) Such open design processes have proven successful in projects ranging from very focused efforts, such as specifying the Advanced Encryption Standard (AES) [NBB+00], through very large and complex systems such as maintaining the Linux operating system.

Alternatively, security models such as the voter-verified audit trail allow for electronic voting systems that produce a paper trail that can be seen and verified by a voter. In such a system, the correctness burden on the voting terminal's code is less extreme because voters can see and verify a physical object that embodies their vote. Even if, for whatever reason, the machines cannot name the winner of an election, then the paper ballots can be recounted, either mechanically or manually, to gain progressively more accurate election results.

The model where individual vendors write proprietary code to run our elections appears to be unreliable, and if we do not change the process of designing our voting systems, we will have no confidence that our election results will reflect the will of the electorate. We owe it to ourselves and to our future to have robust, well-designed election systems to preserve the bedrock of our democracy.

[16] `http://www.eskimo.com/~weidai/cryptlib.html`

Acknowledgments

We thank Cindy Cohn, David Dill, Badri Natarajan, Jason Schultz, David Wagner, and Richard Wiebe for their suggestions and advice.

References

[BCK96] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology: Proceedings of CRYPTO '96*, pages 1–15. Springer-Verlag, 1996.

[BDJR97] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 394–403. IEEE Computer Society Press, 1997.

[Cal00] California Internet Voting Task Force. *A report on the feasiblility of Internet voting*, January 2000. `http://www.ss.ca.gov/executive/ivote/`.

[Cal01] *Voting: What Is; What Could be*, July 2001. `http://www.vote.caltech.edu/Reports/`.

[CBR03] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin. *Firewalls and Internet Security; Repelling the Wily Hacker*. Addison-Wesley, Reading, MA, 2003.

[Die03] Diebold Election Systems. `AVTSCE/` source tree, 2003. Available in `http://users.actrix.co.nz/dolly/Vol2/cvs.tar`.

[DJ02] J. Daemen and V. Jijmen. *The Design of Rijndael: AES–The Advanced Encryption Standard*. Springer, 2002.

[DMNW03] D. L. Dill, R. Mercuri, P. G. Neumann, and D. S. Wallach. *Frequently Asked Questions about DRE Voting Systems*, February 2003. `http://www.verifiedvoting.org/drefaq.asp`.

[Gil98] J. Gilmore, editor. *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design*. O'Reilly, July 1998.

[Har03] B. Harris. *Black Box Voting: Vote Tampering in the 21st Century*. Elon House/Plan Nine, July 2003.

[Jon01] D. W. Jones. *Problems with Voting Systems and the Applicable Standards*, May 2001. Testimony before the U.S. House of Representatives' Committee on Science, `http://www.cs.uiowa.edu/~jones/voting/congress.html`.

[Jon03] D. W. Jones. *The Case of the Diebold FTP Site*, July 2003. `http://www.cs.uiowa.edu/~jones/voting/dieboldftp.html`.

[Ker83] A. Kerckhoffs. *La Cryptographie Militaire*. Libraire Militaire de L. Baudoin & Cie, Paris, 1883.

[Mer00] R. Mercuri. *Electronic Vote Tabulation Checks and Balances*. PhD thesis, University of Pennsylvania, Philadelphia, PA, October 2000.

[Nat01] National Science Foundation. *Report on the National Workshop on Internet Voting: Issues and Research AGenda*, March 2001. `http://news.findlaw.com/cnn/docs/voting/nsfe-voterprt.pdf`.

[NBB$^+$00] J. Nechvatal, E. Barker, L. Bassham, W. Burr, M. Dworkin, J. Foti, and E. Roback. *Report on the Development of the Advanced Encryption Standard (AES)*, October 2000.

[NBS77] NBS. Data encryption standard, January 1977. Federal Information Processing Standards Publication 46.

[Rub02] A. D. Rubin. Security considerations for remote electronic voting. *Communications of the ACM*, 45(12):39–44, December 2002. `http://avirubin.com/e-voting.security.html`.

[Sch96] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, New York, second edition, 1996.

[Sch00] B. Schneier. *Secrets and Lies*. John Wiley & Sons, New York, 2000.