71058

Software Review and Security Analysis of the Diebold Voting Machine Software

Ryan Gardner Alec Yasinsac Matt Bishop Tadayoshi Kohno
Zachary Hartley John Kerski David Gainey
Ryan Walega Evan Hollander Michael Gerke

Security and Assurance in Information Technology Laboratory
Florida State University
Tallahassee, Florida

July 27, 2007

Final Report
For the Florida Department of State

Software Review and Security Analysis of the Diebold Voting Machine Software

Table of Contents

Software Review and Security Analysis of the Diebold Voting Machine Software
Final Report

1 Executive Summary

On May 14th 2007, the Florida Department of State (FLDoS) commissioned an independent expert review of Diebold Voting System Software, as documented in their Statement of Work [1]. The team, led by Florida State University's (FSU) Security and Assurance in Information Technology (SAIT) Laboratory, was commissioned to conduct a software code review as part of the state's voting system certification process. This report is the culmination of that review.

1.1 The Analysis' Scope

The scope of the investigation, as defined in the Statement of Work (SoW), is:

> *This review is for the purpose of yielding technological data to DOS [FLDoS] to ensure voting system effectiveness and security in Florida elections by investigating for potential flaws in target software as documented in reported literature and other published studies [1].*

The team constructed a flaw list from surveyed literature and this list drove the analysis. FLDoS provided the team fully functional hardware and accessories, which we utilized to test and confirm the code's operation. We did not conduct a comprehensive software review nor penetration testing, as each of these was outside the project scope.

We emphasize that our technical analysis reflects neither endorsement of, nor opposition to, certification. We present this technical data for consideration by FLDoS in their decision processes.

1.2 Systems Analyzed

The two primary systems analyzed consist of the Diebold Optical Scan, firmware version 1.96.8, and Touch Screen, firmware version 4.6.5. We also examined the Diebold Touch Screen bootloader version 1.3.6 as well as GEMS server software version 1.18.25. We considered flaws in previous versions of the software for all parts of each system, including those found in the AccuBasic interpreters.

1.3 Findings Summary

Our primary findings are:

The version of the Optical Scan and Touch Screen software that we examined:

(a) fixed many of the flaws in earlier versions, but

(b) retain significant flaws that are documented in this report.

As an example of the issues that remain, flaws in the Optical Scan software enable a type of vote manipulation if an adversary can introduce an unofficial memory card into an active terminal before the voting (or early voting) period (e.g., during "sleepover"). Such a card can be preprogrammed to alter the correspondence between physical bubbles on the scanned paper ballots and the candidates with which they are associated. Specifically, it can be used to essentially swap the electronically tabulated votes for two candidates, reroute all of a candidate's to a different candidate, or tabulate votes for several candidates of choice toward another chosen candidate. We implemented this attack in the laboratory. The attack succeeds despite new protection mechanisms apparently designed to protect against similarly-documented attacks in previous studies.

Many reported flaws were removed from the Touch Screen software. Nonetheless, we identified many that still exist. As one example, we found an attack that allows an adversary to prepare official, activated voter smart cards that would enable voters to cast multiple ballots in a ballot-stuffing attack. Creation of the cards requires an adversary able to insert a custom smart card into a legitimate voting terminal and to read the data off of a valid voter card (these steps could be done by separate adversaries.) Once the adversary obtained the necessary information in this way, she could then create smart cards that could be used at any precinct throughout a county. Even if detected, this attack is not correctable: the malicious ballots, either in electronic or paper form, are essentially unidentifiable and thus cannot be removed.

We provide our detailed findings and supporting analysis in the sections below.

2 Project Introduction and Background

2.1 Report Organization

This document is the project report. It contains all of our pertinent findings and conclusions and the technical analysis that supports these conclusions. The document is written in two parts. The public part (Sections 1-7 and Appendix A) constitutes the public report in its entirety; it contains our findings and the analysis to support these findings. It is intended for public dissemination. In accordance with the terms of the Statement of Work, we have avoided revealing proprietary information in the public part of the report, and we are careful to avoid revealing information that would describe how to attack an election. The public report stands on its own and reflects the totality of our findings regarding known flaws.

The private part consists of Appendix B and Appendix C, which are confidential as required by the Statement of Work because (a) they contain vendor-proprietary information; (b) they contain information about flaws that are not relevant to this investigation; or (c) they describe how to mount security attacks in detail and thus are not appropriate for the public report.

This document first gives background information about the known flaws that guided our analysis. We then describe our findings and conclusions.

2.2 Terminology

2.2.1 SAIT Team. This phrase refers to the investigators organized for this project by SAIT Laboratory. The team members are identified in Section 2.3 below.

2.2.2 Flaw/Fault/Vulnerability/Threat. In this report's context, these four terms are closely related, distinguished only by subtleties and nuance. A software *flaw* is a defect that may or may not have an impact on normal program operation. We use the terms *flaw* and *fault* interchangeably. Software *vulnerability* is a code state or status that may lead to improper program operation. Thus, a flaw may introduce a vulnerability. The term *threat* refers to a potential occurrence that exploits a vulnerability, and is sometimes loosely used to refer to the threat manifestation.

2.2.3 Adversary/Attacker/Intruder. We use these terms interchangeably to refer to any malicious party that may try to manipulate a voting system. Our analysis focuses on two attacker categories, based on their role in the voting process. The first prospective attacker class is *Voter*. Three important voter properties are: (1) there are many voters; (2) they are untrusted; and (3) they have very limited access to the system.

The second major prospective attacker category is *Poll Worker*. There are fewer poll workers than voters, but they have greater system access in terms of time and capability.

Elections officials and voting system vendors represent two other recognized potential attacker categories. However, we consider attacks by these parties to be largely outside the scope of this review.

2.2.4 Adversarial Skill. In this report we shall often refer to resources required by an adversary, one of which might be technical sophistication. For example, in Section 3.6 we state that an adversary should have the requisite knowledge and skills. When we make such statements, we generally speak of the *first* adversary to develop the attack, *not* necessarily the individuals carrying out the attack. In many cases, the individuals carrying out the attacks do not need to have the same knowledge and skills as the individuals developing the attacks.

2.2.5 Fixed. A flaw is *fixed* if the corresponding vulnerability is no longer present in the source code. We emphasize that it is impossible to identify all flaws, so any statement in this report that an item is fixed simply reflects our best professional opinions.

2.2.6 Benign. A flaw is *benign* if its technical effect does not interfere with the intended operation. Note that this is our technical assessment based on the extent of this study and the assumptions stated herein. Readers should consider all reported and potential flaws with a broader eye for system impact, in particular taking into account the specific policies and procedures under which the systems will be used.

2.2.7 ".abo" File. The Diebold AccuBasic Object scripts reside in files whose extension is .abo. We occasionally refer to AccuBasic Object script programs as .abo files.

2.2.8 TSX™/OS™/AccuVote. This report uses numerous voting machine model identifiers. We generally refer to the Diebold Touch Screen system as the TSX and the Optical Scan device as the OS. Both are members of the Diebold AccuVote series.

2.3 The Software Review Team

2.3.1 Senior Investigators

Matt Bishop is a Professor of Computer Science at the University of California at Davis. He is an expert in secure software and electronic voting systems, having participated in several widely recognized electronic voting software systems code reviews. His computer security textbook, *Computer Security: Art and Science*, is the acknowledged benchmark against which all others related to this topic are measured.

Tadayoshi Kohno is an Assistant Professor of Computer Science and Engineering at the University of Washington. He is an expert in cryptography and secure software and is an author on the seminal "Hopkins Paper" [8] that triggered the current movement to more aggressive voting system code review.

Alec Yasinsac is an Associate Professor of Computer Science at Florida State University, a co-Director of SAIT Laboratory, and is the lead Principal Investigator on this project.

2.3.2 Investigators

David Gainey is a computer science graduate student, a member of SAIT Laboratory, and is a member of the technical staffat the Florida State University Office of Technology Integration.

Ryan Gardner is a doctoral student at Johns Hopkins University and a member of the ACCURATE center for voting systems research.

Michael Gerke is a computer science graduate student and a member of SAIT Laboratory at Florida State University. He is presently employed by the Florida Department of State, Division of Elections.

Zachary Hartley graduated from FSU with a Computer Science Masters degree in April 2007. He is presently employed by the Florida Department of State, Division of Elections.

Evan Hollander is a computer science graduate student, a systems administrator in the Computer Science Department, and a member of SAIT Laboratory at Florida State University.

John Kerski is a computer science graduate student and a member of SAIT Laboratory at Florida State University.

Ryan Walega is a computer science graduate student and a member of SAIT Laboratory at Florida State University.

2.3.3 Team Organization

2.3.3.1 Internal Team Structure and Operation. All investigators conducted hands-on source code analysis. The list of previously noted flaws drove the analysis. Members received analysis assignments and were also free to investigate independently. Several investigators exercised automated analysis tools. The Lead PI coordinated analysis activities.

Team members conducted structured note taking and referred to these notes during final report preparation.

2.3.4 External Communication and Coordination

2.3.4.1 Florida Department of State (FLDoS). As noted in the SoW, FLDoS was entitled to observe the code review process at their discretion; they chose to limit their interaction. FLDoS only interacted with the team at our invitation. They proved to be a valuable information resource, providing election configuration files, general election knowledge, and hardware demonstrations to support our analysis. Their support was consistently prompt and complete. The FLDoS placed no restrictions on our activities within the SoW.

2.3.4.2 Diebold Election Systems. The SAIT Team established a communication channel with the vendor within the first project week and interacted with them on several occasions. The purpose of these communications was for the SAIT Team to gather information and to clarify issues relative to the target systems and source code.

The relationship with Diebold Election Systems was professional and cordial. Diebold Election Systems assisted the SAIT Team by providing specific feedback relative to the gathered flaw list and also provided the team a copy of the private appendix to the 2006 California VSTAAB review [2]. We also conducted a conference call between the SAIT Team and Diebold engineers that facilitated our review.

2.3.4.3 Florida State University. FSU and SAIT Laboratory hosted the code review and provided invaluable analysis resources and administrative support beginning the first active SoW day. The spaces and resources were ideal for this type of review.

2.4 The Investigative Process

The vast majority of this work took place in SAIT Laboratory. In accordance with our project plan, the investigation began with a short collaborative planning phase. The team met in SAIT Laboratory and spent several days examining code, documentation, and subject reports to understand the problem and to formulate an investigative approach. We then followed the resulting plan that was submitted to the Florida Department of State in accordance with the Statement of Work [1].

As the first objective, the team constructed a flaw list from surveyed literature; the flaw list is given in Appendix A. This list drove the analysis. The team did *not* conduct a comprehensive software review. Rather, we sought to determine if previously documented flaws exist in the target software. It is possible that there are other flaws that were not identified in previous reports; this study would not have identified them because the team was only looking at the question of whether previously identified flaws were fixed. This is different from a comprehensive security review. Despite not actively looking for new flaws, we did identify some and include a discussion of them in the private Appendix C.

The Florida Department of State provided the team fully functional hardware and accessories matching the target software. We utilized these systems to confirm our understanding of the software. We did *not* conduct penetration or red team testing for these systems, as that was outside our charter.

The two target systems are the Diebold Optical Scan and Touch Screen systems that are presently submitted to the Florida Department of State Bureau of Voting Systems Certification. Specifically, we were provided the AccuVote TSX™ version 4.6.5, TSX bootloader version 1.3.6, and AccuVote OS™ version 1.96.8 software. We considered previously-identified flaws in earlier versions of all software for both systems including the AccuBasic interpreters. Additionally, we were provided GEMS 1.18.25 to analyze interaction between this election management system and the two target voting devices.

2.4.1 Limits of this Study

Our analysis examined only those flaws previously reported in the cited literature. We examined the source code and the systems to determine whether or not the reported flaws still exist. Where appropriate, we attempted an attack that would exploit a reported flaw to demonstrate our findings.

Any study involving systems and source code of the complexity of the Diebold Optical Scan and Touch Screen raises questions of completeness: could the investigators have missed problems? We have documented our efforts to allow others to evaluate the thoroughness of our study. When we have found successful attacks, we describe them in sufficient detail in the private appendices to allow others to duplicate them. Where we believe the flaw has been mitigated or fixed, we describe our basis for that assertion. Throughout, we document our assumptions so others may evaluate our results in the context of their environments.

It is important to understand that our conclusions were guided by the source code examined. This means that if the code on a given system does not correspond to the source code we examined, our results may not apply. Further, if the programs that compile, link, load, or install the software, or any libraries or code linked in, disrupt the correspondence between this source and the given system, our results may not apply.

Finally, we do not claim that our results extend beyond the scope of our investigation. We reiterate that the purpose of this report is to evaluate the technical properties of the current systems with respect to our list of previously-identified flaws. The purpose of this report is neither to condemn nor endorse these systems and the findings herein should be considered in the context of the overall election process. We specifically do not contend that these systems are correct or secure beyond the specific results given here. This report is concerned solely with the question posed in the Statement of Work [1] and we do not claim that these results extend to a broader context.

2.4.2 Diebold Known Flaw Details

We analyzed reports discussing previously discovered flaws in the systems [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,15]. The resulting flaw list given in Appendix A identifies one hundred and twenty six (126) items and we confined our study to them.

It is also important to note that the items in this list have been rigorously debated by the public and for many of the listed flaws, their proper threat characterization remains unclear and sometimes highly contentious. In our assessment, some of the items are benign. Additionally, several items that made the list do not apply to this study, for example, they apply to a fundamentally different architecture than the software we reviewed. To retain the integrity of the list, we chose to retain them, but do not address them further.

2.5 Diebold Software Architecture

To protect intellectual property, we avoid providing details where these details are not relevant to our findings. We provide the following observations to give context to our findings.

2.5.1 Code Structure

As we expected, the code style and readability varied significantly across the code base. One documented item (#29) identified complicated code as a problem, and there is substantial complicated code throughout the system. Other flaw list items (#29, 30, 32) pointed to poor internal program documentation, such as missing and weak comments. Another item (#31) highlighted design documentation weaknesses. Other than the latter of these, we found the code pretty much as described in the literature. While the documentation we received was much more comprehensive than we expected, detailed design documentation was not available.

2.5.2 Off-The-Shelf Software

As with most modern applications, there are several proprietary components to the reviewed voting systems. While we considered application-operating system interactions and documented flaws relative to bootloaders, we did not independently investigate the proprietary operating system, database system, or driver software. The systems also incorporated non-proprietary components developed openly by the public software development community.

3 Findings

We group our findings according to their related functional areas. We address software components grouped by optical scan firmware, bootloader, interpreter, touch screen firmware, etc. Because many flaws overlap, we cross-reference extensively throughout the document to ensure consistency and to provide context to related flaws.

3.1 Overview

The Team's primary finding is that while we find many improvements in mitigating the vulnerabilities in both the Optical Scan and Touch Screen software that we reviewed, both still retain significant software flaws. In many cases it appears that the vendor attempted to fix these flaws but that the attempted fixes introduced regression faults.

As example of the issues that remain, flaws in the Optical Scan software enable an adversary to introduce an unofficial memory card into an active terminal before the voting (or early voting) period begins. This memory card can be preprogrammed to redistribute votes cast for selected candidates on that terminal including swapping the votes for two candidates. The attack can be carried out with a reasonably low probability of detection assuming that audits with paper ballots are infrequent and that the preprogrammed cards are not detected before use. We implemented this attack in the laboratory and it succeeds despite new protection mechanisms apparently designed to protect against similarly-documented attacks in previous studies.

Many reported flaws were eliminated on the Touch Screen system as well. Nonetheless, we identified and constructed an attack that would allow an adversary to convert official, activated voter cards into smart cards that would enable voters to cast multiple ballots in a ballot-stuffing attack. These cards could be used at any precinct throughout a county. While polling place procedures may mitigate this attack, the attack might evade even rigorous policy enforcement. More damaging is that this attack is not easily correctable. The malicious ballots, whether electronic or paper, would be essentially impossible to identify, so they could not be removed.

That said, we caution the reader that our findings are slanted towards negative results. We focus on flaws that are not completely fixed. Even where we describe flaws that are greatly improved, our focus is on any remaining weakness, as is our charter. Conversely, we do not discuss the many flaws that may have been removed, nor do we describe the structural or general system improvements that we see, though in our conclusion we do identify some cases where flaws appear fixed.

We reiterate that the purpose of this report is to evaluate the *technical properties* of the current systems with respect to previously-identified flaws. Its purpose is neither to condemn nor endorse these systems itself, but rather the findings herein should be carefully considered in the context of the overall election process.

3.2 Prerequisites and Mitigation

As noted above, not all software flaws create vulnerability. In order to expand the context for each flaw, we include a description of circumstances or environments sufficient to exploit the associated potential vulnerability resulting from each remaining flaw.

Additionally, we point out potential procedural or technical processes that could mitigate, reduce, or eliminate the associated vulnerability. Our list of potential mitigation strategies is not meant to be comprehensive, but rather to indicate one direction for addressing the vulnerabilities. As is the nature of computer security, we also recommend that any implementation of a mitigation strategy be subject to a rigorous security evaluation.

Finally, a standard tenant in security is "defense in depth," which suggests that a system should provide adequate security even if individual components fail. We recommend that this principle be considered when evaluating candidate protection mechanisms, whether procedural or technical.

3.3 Prospective Flaws Identified in the Literature

As we noted above, our investigation began with an intense literature review. This Findings Section reports technical information that our analysis revealed about the items documented in the flaw list.

By our analysis, one hundred eight (108) of the one hundred twenty-six (126) flaws that we identified are pertinent to our review. Thirty-seven (37) pertinent flaws were not repaired, thirty-one (31) flaws were improved without complete removal, and forty (40) flaws were removed, corrected, or mitigated.

3.4 Hardware and Physical Security Issues

The flaw list contains fifteen items related to hardware and physical security issues. Our findings suggest that all hardware flaws are either corrected or not applicable to this review, except the one item below.

3.4.1 Diagnostic Mode Not Protected (#55)

An individual with unsupervised terminal access could place an Optical Scan terminal into *diagnostics* mode by simply pressing the two exposed "on" and "off" buttons on the machine and resetting it. From there, she could, for example, connect a device to the serial port and obtain all or most of the data on the memory card as described in Section 3.8.1.1, or reset the machine clock.

Attack Prerequisites: In order to exploit this vulnerability, the attacker must:

1. Have the necessary hardware resources. (Depending on what she wants to do in diagnostics mode, she may need none.)
2. Have a brief period of unsupervised access to an Optical Scan terminal.

Potential Mitigation:

1. Election officials: Rigorously enforce rigid physical access control to optical scan terminals.
2. Vendor: Protect access to the optical scan diagnostics mode or employ sufficient cryptographic protection mechanisms.

3.5 The Signature Flaw

The fulcrum for many of the intended repairs is the vendor's RSA signature. Thus, the weakness we uncovered in this scheme is one the widest ranging issues with the optical scan software and is also important to several remaining issues with the TSX.

The hand-coded RSA signature verification is insecure and signatures generated with the implemented method can be forged. This is true of the signatures on the AccuBasic scripts that are run on the optical scan machines and of the signatures on the operating system images that are to be installed on the touch screen terminal. The ability to run arbitrary AccuBasic scripts on the optical scan machines partially enabled the Hursti attack [5], and the ability to replace unverified boot loaders and operating system images enabled the Princeton attack [4].

The code applies what might initially appear to be a fairly standard RSA signature, using a SHA1 hash, a 2048 bit (or sometimes a 1024 bit) RSA modulus, and a public exponent of three (3). Since the SHA1 digest is 160 bits, the RSA input is *padded with zeros*. To verify the signature, a decode operation computes the modular exponentiation of the signature with the corresponding public key to transform it back into the SHA1 hash of the signed file. However, when the result of the computation is checked against the computed SHA1 hash of the signed file, only the 160 bits that correspond to the SHA1 digest are compared to the hash. The other 1888 bits are not examined. Not examining the other 1888 bits renders this RSA signature variant vulnerable to forgery attacks.

Given the way that signatures are verified, we can forge a signature on any file that has an odd SHA1 hash, i.e. a SHA1 hash with a one in the least significant bit. Since we can almost always add spaces, no-ops, or other data that will not affect the way files are interpreted, we can forge a signature on a message with essentially any semantic meaning we choose. Our code automatically modifies an AccuBasic script to give it an odd SHA1 hash without altering the script's functionality and then forges signature on the script. In its entirety, the code consists of approximately 250 lines of Java, using the standard Java 5 Application Programming Interface (API), and executes in a negligible amount of time. See Inset 1 below for more details of the signature verification process and an overview of the attack.

Attack Prerequisites: This functional flaw is not exploited independently, but is utilized to manifest vulnerability from other flaws. In order to exploit this vulnerability, the attacker must:

1. Discover a corresponding application vulnerability protected by the signature.
2. Gain appropriate access and resources to forge the necessary signature (the public RSA key and a commodity PC).
3. Inject malicious data with the forged signature into the application.

Potential Mitigation:

1. Election officials: Rigorously enforce rigid physical access to media and devices.
2. Vendor: Employ a standardized and widely-accepted mode for public key signature generation and verification. Consider, for example, the PKCS #1 RSA Cryptography Standard.

3.6 Optical Scan Memory Card is Not Integrity Protected (#90)

The data on optical scan memory cards is neither encrypted nor authenticated save the insecure signature on the AccuBasic script, which resides on the card. This vulnerability leads to many potential attacks. In Section 3.8.1.4 below, we describe how we constructed several exploits that manipulate the vote counts on a memory card during the voting day.

With an understanding of the checksums on the memory card data, manipulating any of the unauthenticated data is not difficult. Thus, we further constructed exploits that could be carried out by an adversary with access to a memory card prior to election day and also note other possible threats.

The signature is computed as a plain RSA signature on the SHA1 digest of the message. (We inferred this generation procedure from the signature verification code.) Specifically, let n be the public RSA modulus, d be the private RSA key (exponent), 3 be the RSA public key (exponent), and M be the message to be signed. Furthermore, let $x : a - b$ denote the a through b^{th} bits of x where bit 0 is the least significant bit of x. Then the signature σ is computed as follows:

$$H = \mathrm{SHA1}(M)$$

$$\sigma = H^d \bmod n$$

To verify that σ' is a valid signature on message M', the code computes and checks the following:

$$H' = \mathrm{SHA1}(M')$$

$$H'' = \sigma'^3 \bmod n$$

If the least significant 160 bits of H'' match H', then report VERIFICATION SUCCEEDED. Otherwise, report VERIFICATION FAILED.

If a σ' can be found for many messages M' without knowledge of d such that the verification succeeds, then signatures can be forged. Two steps are sufficient to produce such forgeries: 1) Modify the message M' to an M'' such that $\mathrm{SHA1}(M'')$ is odd. 2) Construct σ' such that $\mathrm{SHA1}(M'') = (\sigma'^3 \bmod n) : 0 - 159$.

The first step requires simple insertion of characters that will not change the meaning of M' such as trailing spaces, no-ops, and other possible alterations that will not significantly affect the way the message is interpreted.

For the second step, we developed a simple algorithm for computing the forged signature itself, which we have omitted from this report. We confirmed the algorithm's correctness and efficiency by forging signatures on many custom AccuBasic scripts that properly verified and executed on the optical scan machine.

Inset 1. Overview of the RSA Signature Verification Process and Attack.

By manipulating the physical ballot data stored on the memory card, we were able to:

1. Swap two candidate's vote counters.
2. Cause all votes in a race to be tallied for a single candidate of our choice.
3. Cause all votes for one or more lesser-known candidates to tally for a selected candidate.
4. (Many alternative distributions.)

Aside from the exploits we conducted in our lab, the lack of authentication of card data presents several other potential threats. The memory card contains data that is used for a variety of purposes throughout the AV-OS firmware. Because identifying and preventing all possible buffer overflows is an exceptionally difficult task, there is little assurance that a buffer overflow exploit could not be conducted through the manipulation of memory card data. Such an exploit could, for example, allow an attacker to alter optical scan machine memory or enable her to execute code of her choice and essentially gain complete control of the machine.

Finally, the memory card for the optical scan machines contains many memory pointers. The pointers are used to address candidate vote counters, ballot data, scripts, audit logs, and other structures. There is no guarantee that an adversary would not be able to cleverly manipulate these pointers in a way such that she could change vote counts or otherwise maliciously alter the functionality of the machine.

As with other attacks on the optical scan terminal, these attacks might be detected by aggressive auditing or during a recount. Once detected, they could be corrected by properly counting the paper ballots.

Attack Prerequisites: In order to exploit this vulnerability, the attacker must:

1. Have the requisite knowledge and skills.
2. Acquire a current memory card or complete knowledge of memory card structure and the ballot and the equipment necessary to write to the card.
3. Gain sufficient unsupervised access to the terminal to replace the card before, during, or after the election but before the counts are sent to the central server.

Potential Mitigation:

1. Election officials: Rigorously enforce rigid access to media and equipment to limit exploit opportunity.
2. Vendor: Authenticate and verify all data on the AV-OS memory card whenever it is accessed. Since the AV-OS is very limited in computational power, one efficient way of doing this may be the following, although all solutions we are aware of have some security weaknesses if secure hardware is not used: a.) Create a random, symmetric authentication key during some election initialization step, and store the key in inaccessible, persistent storage. b.) After initially verifying a public key signature on the memory card, use the generated key to update a Message Authentication Code (MAC) on the entire contents of the memory card and a counter value. c.) Increment the counter and update the MAC every time the machine writes to the memory card. (The counter should not be stored on the card.) d.) Verify the MAC with the current counter every time the card is accessed. e.) At the end of the election, sign the contents of the memory card with a unique private key. (If secure hardware is not used to store the key, this scheme will have fundamental security limitations, but the vendor can attempt to store it the most inaccessible location possible and encourage election officials to save the key there for the minimal necessary amount of time.)

3.7 AV-TSX Issues

3.7.1 AV-TSX Firmware Faults

3.7.1.1 Cryptographic Key Management (#21)

Two 128-bit AES keys are used for all authentication and encryption.

The "data encryption key" is used for almost all authentication and encryption purposes, including encrypting and generating the message authentication codes for the stored electronic ballots (described in Section 3.7.1.10) and generating the message authentication codes for the election database file (described in Section 3.7.1.8). It is also used to key the hash that stores the supervisor PIN (described in Section 3.7.1.4).

The "system key" is pivotal to key management. It is used to encrypt the file where the data encryption key is stored along with the smart card passwords ("keys" and "magic numbers") and encrypting audit logs. See Section 3.7.1.3 for the specific ways the smart cards are authenticated and see Sections 3.7.1.8 and 3.7.1.10 for the specific usage of and issues with the keys with the election database file and the stored voter ballots respectively.

The system key is generated by computing an MD5 hash of the machine serial number. Its value is never changed after generation. Since the machine serial number is public, the system key is also essentially public. Anyone who knows this procedure can generate the system key and can access anything it protects, including the data encryption key and anything that it is used to encrypt.

The data encryption key is loaded in one of two ways. First, it can be read from a locally stored file. As noted earlier, the key file is encrypted under the publicly computable system key, so it is an open file to anyone who knows the encryption procedure (which is standard and well known).

Alternatively, the data encryption key may be loaded from a "security key card". The security key cards are insecurely protected as described in Section 3.7.1.3 (the same as all other smart cards), which allows anyone to read all data from them. They also initially use the default password and always use a fixed magic number (see Section 3.7.1.3 for a description of the magic number). Hence, security key cards should also not be used for storing secret information like keys for non-negligible amounts of time.

Above we noted that the same system uses the same cryptographic key for multiple purposes, including both encryption and message integrity protection. As a general principle, such key-reuse is strongly discouraged by the cryptographic community since the reuse of keys can introduce vulnerability.

Attack Prerequisites: This flaw is not independently exploited; rather it is exercised as a component of an attack on another voting application. In order to exploit this vulnerability, the attacker must:

1. Gain access to the machine serial number and compute an MD5 hash.
2. Gain access to a memory card.
3. Or alternatively, gain access to a current security key (smart) card.

Potential Mitigation:

1. Elections officials: Generate the keys on the security key cards immediately before loading the keys onto the machines and then securely delete them as soon as practicable.
2. Vendor: Use accepted key management techniques and public key cryptography.
3. Vendor: Use secure hardware private key storage and operations.

3.7.1.2 Memory Card Update File is Unprotected (#42, 49, 51)

The RABA Study [11] identified attacks that were allowed by modifying unauthenticated and unencrypted files stored on the memory card. Most files on the memory card are encrypted under the present firmware version. However, the file assure.ini remains unencrypted and unauthenticated and is subject to malicious manipulation.

Given the lack of integrity protection on the assure.ini file, an attacker who could remove the card from the TSX terminal could modify the file to set the machine into pre-election mode. While the terminal is in this mode, the attacker could create valid voter cards. If the authentication key necessary to validate voter cards is the same across precincts, as we understand to be common practice in Florida, these cards could also easily be modified to be used at any other precinct within a county. In some cases modifications may not be required. Therefore, the scalability of this exploit could go far beyond one precinct and affect a whole county. We implemented and demonstrated this attack in our lab.

Attack Prerequisites: In order to exploit this vulnerability, the attacker must:

1. Have relevant partial knowledge of the assure.ini file layout.
2. Provide a supply of appropriately formatted smart cards.
3. Gain sufficient access to a touch screen machine and its memory card. The number of voter cards the attacker could produce is dependent on the amount of time she has with the machine. Note that machines can also be taken off their stands and operate for a significant amount of time on battery power.

Potential Mitigation:

1. Election officials: Rigorously enforce rigid physical access procedures for all electronic voting terminals.
2. Vendor: Authenticate the assure.ini file, preferably with a public key signature. Note that encryption on its own may still not provide integrity protection for this file.
3. Precinct logs might, in some cases, detect this attack after the fact. However, even if the ballot stuffing attack is detected, there are no mechanisms built into the software for identifying the malicious ballots and recovering from the attack.

3.7.1.3 Smart Card Authentication Uses Only a Hard Coded Password (#13, 40, 41, 69)

The data and smart card passwords can now be set by election workers. Nevertheless, the implemented authentication protocol is not secure, allowing an attacker to create counterfeit, validating smart cards, including voter cards.

When a smart card is inserted, the terminal sends a "smart card key" to the card. Thus, an attacker can obtain it by inserting a custom smart card into the machine and recording the received key. Once the attacker has the key, she can pass it to her own machine (such as a PDA), which she can then authenticate as a legitimate voting terminal to a good smart card. At this point, she can read the "magic number" (essentially a global, legitimate smart card password) from the smart card and can use it to forge smart cards of her choice. Ultimately, the attacker only needs the magic number and not the key since she can make cards that always respond to key verifications with success messages, regardless of value.

Attack Prerequisites: In order to exploit this vulnerability, the attacker must:

1. Have knowledge and skill to accomplish the attack.
2. Have appropriate equipment.
3. Have voter access to the terminal.

Potential Mitigation:

1. Election officials: Carefully monitor suspicious smart card handling, including times when cards are in pockets.
2. Vendor: Use an established secure smart card authentication protocol that uses a smart card's ability to compute cryptographic operations. (Several such secure protocols have been studied and established.)

3.7.1.4 Supervisor PIN is Not Cryptographically Protected (#14, 69)

The supervisor PIN is now stored on supervisor smart cards as a keyed hash of the actual PIN. Specifically, the PIN is concatenated with part of the "data encryption key" (the first 64 bits), and an MD5 sum is computed over the resulting string. The first 4 bytes of the MD5 are stored on the smart card.

The most significant weakness of this approach again concerns the key management of the "data encryption key" as described in Section 3.7.1.1. The key can be compromised by an adversary with sufficient access to a voting terminal, and an adversary with it can find the PIN using a simple brute force computation.

Again, the act of using the same key for more than one purpose is generally considered poor practice within the cryptographic community. Moreover, the input to the hash function is 64 bits of the 128-bit data encryption key. Using only 64 bits of the 128-bit AES key in this manner may allow an adversary to recover the data encryption key significantly faster than exhaustive search.

Attack Prerequisites: In order to exploit this vulnerability, the attacker must:

1. Gain unsupervised access to a voting machine or security key card.
2. Obtain the "data encryption key". (See Section 3.7.1.1).
3. Gain access to a supervisor card.
4. Read the PIN from the supervisor card. (See Section 3.7.1.3.)

Potential Mitigation:

1. Election officials: Enforce strict access to voting machines, security key cards, and supervisor cards.
2. Vendor: Store the PIN in protected memory on the smart card and use the smart card's ability to compute cryptographic operations to securely verify the PIN.

3.7.1.5 Insecure Storage Mount (#15)

The storage device is mounted by name only, and "\Storage Card" is the directory that Windows CE normally mounts the storage card to. However, this name is also a legitimate standard directory name that may not be associated with the intended purpose. Hence, this method of assessing whether or not the memory card is present is not secure. As one simple example, an attacker could exploit this vulnerability to force the terminal to save votes to an unexpected location, such as one off of the memory card.

Attack Prerequisites: In order to exploit this vulnerability, the attacker must:

1. Gain access to a touch screen voting terminal.
2. Create a "\Storage Card" directory in the Windows CE installation on the terminal.

Potential Mitigation:

1. Election officials and vendor: Ensure that voting terminals are not in debug mode
2. Election officials: Rigorously enforce rigid physical access procedures for all electronic voting terminals.
3. Vendor: Verify that the FILE_ATTRIBUTE_TEMPORARY attribute is set for the directory [8].

3.7.1.6 System Configuration Information is Unprotected (#16)

A large portion of the system configuration is still stored in the system registry and can be altered by directly modifying the registry. The file settings.h defines several large macros for accessing the registry, and appsettings.h defines many specific application settings and functions for accessing those settings using the macros of settings.h. These configurations include informative data such as machine serial number and precinct number and security critical settings such as whether or not the machine should use SSL for network connections.

Attack Prerequisites: In order to exploit this vulnerability, the attacker must:

1. Have access to a touch screen terminal, memory card, and the memory card slot.
2. Have skill and knowledge to load malicious code that would change the system registry or to load a new operating system image with an altered registry, both of which require knowledge of the bootloader process.

Potential Mitigation:

1. Election officials: Rigorously enforce rigid physical access procedures for all electronic voting terminals.
2. Vendor: Authenticate this configuration data.

3.7.1.7 Protective Counter Stored in a Mutable File (#17)

The "protected" counter is read from a mutable file located in the "system directory", which is specified in a registry key. No cryptographic or other checks are made. The counter is written back to the same mutable file. Again, no cryptography or additional precautions are used.

Attack Prerequisites: In order to exploit this vulnerability, the attacker must:

1. Have access to a touch screen terminal.
2. Have skill and knowledge to manipulate or destroy the counter.

Potential Mitigation:

1. Election officials: Rigorously monitor physical access to voting terminals.
2. Vendor: Use secure hardware with a monotonic counter or non-erasable storage to provide a reasonable guarantee that an attacker could not undetectably roll back the counter.
3. Vendor: Use hash chains and other techniques from cryptographically secure audit logs to help prevent an adversary from rolling back the counter prior to the point of compromise.

3.7.1.8 Ballot Definition File is Unprotected (#18)

The ballot definition file is now authenticated with a home-grown Message Authentication Code (MAC). The software uses an encrypted hash value for message integrity protection, which mitigates this vulnerability but does not eliminate the problem. The employed message integrity computation is a non-standard construction that is well-known to be insecure under the standard definition of integrity for message authentication codes; details are given below. The code in question authenticates the file by computing an MD5 hash over the file's contents and then encrypting the hash using AES in ECB mode.

There are two main concerns that result from this authentication method. The most direct concern regards key management. The AES key used is stored in an encrypted file whose encryption key is deterministically constructed from the machine serial number (see Section 3.7.1.1). This gives an attacker the ability to modify the ballot definition file and generate a new, valid authenticator.

The second concern arises from the fact that MD5 is a deprecated hash function and is no longer considered secure [13]. Researchers have developed efficient methods for finding collisions in the function. Thus, even assuming sound key management, there is no guarantee that an attacker could not create two ballot definition files that yield the same hash and hence both validate using the same MAC, possibly leaving the ballot definition file susceptible to attack by someone with control of the ballot definition. Furthermore, the existence of collision-finding attacks against MD5 means that the message authentication scheme used in this software does not meet the standard definition of integrity for such cryptographic objects. CMAC-AES and HMAC-SHA256 are two common message authentication schemes of choice.

Finally, note that since stored votes are only associated with a candidate number and not a name (see Section 3.7.1.12), the ability to create custom ballot definition files allows one to alter or switch candidate names without any record in the vote counts or electronically stored ballots.

Attack Prerequisites: In order to exploit the most direct vulnerability above, the attacker must:

1. Know the machine serial number and the key construction.
2. Understand how the ballot definition file authenticator is computed.
3. Know the memory card file structure.
4. Gain unsupervised access to the voting terminal.

Potential Mitigation:

1. Election officials: Rigorously enforce rigid media protection control procedures.
2. Vendor: Sign and verify the file contents with a secure signature scheme.

3.7.1.9 Impersonate a TSX Terminal to GEMS (#20)

This issue is discussed in detail in GEMS Section 3.10.3 below.

3.7.1.10 No Integrity Protection of Stored Electronic Ballots (#22, 23)

The authenticity protection mechanisms used here are similar to the protections used for the election database file as explained in Section 3.7.1.8, and they share the same issues.

Using the vender's terminology, a "signature" is generated on each ballot/ResultRecord. The code MD5 hashes the ResultRecord and then uses AES in ECB mode to encrypt the hash using the "data encryption key" to produce the authenticator, or Message Authentication Code (MAC).

The most significant problem here, as with the election database file (Section 3.7.1.8), is that there are no clear means of securely managing the key used for the message authentication code ("signature") and encryption. Additionally, given historical attacks against other systems that reused cryptographic keys for multiple purposes, we recommend that the same key not be used for both generating the message authentication code and for encrypting the message.

Furthermore, as also discussed in Section 3.7.1.8, MD5 is deprecated and researchers have developed efficient methods for finding collisions in the function. The existence of collision-finding attacks against MD5 implies that the message authentication scheme used in this software does not meet the standard definition of integrity for such cryptographic objects. CMAC-AES and HMAC-SHA256 are two common message authentication schemes of choice.

Finally, given the use of CBC mode for encryption, the initialization vector used for each encryption should be unique and unpredictable.

Attack Prerequisites: In order to exploit the most direct vulnerability above, the attacker must:

1. Know the machine serial number and the key construction.
2. Understand how the electronic ballot authenticators are computed.
3. Know the memory card file structure.
4. Gain unsupervised access to the voting terminal or memory card before the electronic ballots are tabulated or transferred.

Potential Mitigation:

1. Election officials: Rigorously enforce rigid media protection control procedures.
2. Vendor: Use public key encryption and signatures to encrypt and authenticate each electronic ballot.

3.7.1.11 Ballots are Stored Sequentially (#26, 27)

When a ballot is cast, it is encrypted and appended to the ballot files in the primary and secondary storage directories. Thus, the ballots are stored in the order that they are cast and anyone with the encryption key could correlate votes with voters.

Furthermore, a timestamp, with second granularity, is also stored (encrypted) with each ballot. So, not even subsequently shuffling the order of ballots will prevent someone with the encryption key from linking votes with the voters who cast them.

Attack Prerequisites: In order to exploit this vulnerability, the attacker must:

1. Have unsupervised access to the voting machine
2. Have access to the proper encryption key (see Section 3.7.1.1).

3. Know when voters vote or the order in which they vote.

Potential Mitigation:

1. Election officials: Rigorously enforce rigid key management practices.
2. Vendor: Store the ballots in a non-serial order and without a timestamp.

3.7.1.12 Candidate Information is Not Stored in the Results File (#19)

The cast and electronically stored ballots ("ResultRecords") do not contain any information about candidates. For all non-write-in candidates, each ballot only stores, for example, that candidate number 1 received a vote, candidate number 5 received a vote, etc. Thus, if the names on the screen do not correspond to what is expected, votes will be counted for the wrong candidates. Furthermore, an attack that could alter the presentation of the names without trace, for example by temporarily modifying the ballot definition file, would be particularly effective since there would be no way to detect the attack.

Attack Prerequisites: In order to exploit this vulnerability, the attacker could:

1. Design and authenticate a custom ballot definition file with altered candidate names (see Section 3.7.1.8) or find another exploit.
2. Gain access to a terminal memory card or an election office where ballot definition files are stored.
3. Overwrite the legitimate ballot definition file with the custom one.

Potential Mitigation:

1. Election officials: Rigorously enforce access control to voting machines and ballot definition files.
2. Vendor: Store the displayed names or other explicit information corresponding to every vote with each electronically cast ballot.

3.7.1.13 Audit Logs are Not Cryptographically Protected (#28, 49)

The logs are encrypted and authenticated the same way as the electronic ballots, using the "system key", which is insecure. (See Section.3.7.1.10) Hence, they are also susceptible to viewing and modification by an adversary. Among other things, such an ability may allow an adversary to erase record of an attack without detection.

3.7.1.14 Data is Neither Authenticated Nor Encrypted Over the Communication Link (#25, 35, 45, 46)

The TSX firmware now supports SSL protection of its GEMS connections, though its use is optional. This issue is discussed in detail in GEMS Section 3.10.3 below.

Additionally, the SSL pseudorandom number generator is seeded with poor entropy, based on tightly bounded clock measurements. Cryptographic protocols often provide degraded security when the random numbers are not securely generated. To mitigate potential concerns, the vendor could seed the cryptographically secure pseudorandom number generator with sufficient entropy.

3.7.2 AV-TSX Bootloader Faults

3.7.2.1 Bootloader Automatically Replaces Itself (#3, 4, 6, 11)

There are now two processes to update software in the TSX. One of these automatically updates the unsigned bootloader with a file named "eboot.nb0" if it is found on the memory card. This method is conditionally compiled. However, in the software version that we are analyzing, the code is included.

In this approach to updating, the bootloader does not perform any kind of cryptographic authentication of the replacement bootloader found on the memory card. This vulnerability is mitigated by the requirement that the machine be in "debug mode" for the update to occur. An attack loading a custom, malicious bootloader is immediately possible if the terminal is delivered to the polling place already in debug mode. Otherwise, an attacker must open the case and move a hardware switch to enable this attack.

While enabling debug mode can only be done with physical access to the hardware, this protection is not as strong as would be provided by requiring a signed bootloader or another strong access control method.

The other software update method that appears to be implemented more recently conducts software updates through the supervisor account when the normal voting machine software is running. Nevertheless, we detected a shortcoming in the supervisor PIN verification process (see Section 3.7.1.4 above).

Attack Prerequisites: In order to exploit this vulnerability, the attacker must:

1. Have a modified memory card with a tested bootloader and
    a. Gain access to a terminal in debug mode or
    b. Obtain a compromised supervisor PIN.

Potential Mitigation:

1. Elections officials must rigorously ensure that TSX terminals are not routinely in debug mode.
2. Vendor: Remove the automatic filename update code from the source.
3. Vendor: Fix the supervisor PIN problem (see Section 3.7.1.4 for the PIN problem description).

3.7.2.2 Bootloader Automatically Replaces Local Operating System (#7)

The flaw has been changed, but still exists in a different form. If the bootloader finds a file named "nk.bin" (or some other nk.* files) on the terminal's memory card, rather than installing it as the new permanent operating system, it boots it as the current one. Once again, the file is not authenticated. This only occurs when the machine is in debug mode.

Attack Prerequisites: In order to exploit this vulnerability, the attacker must:

1. Generate a Windows CE operating system image (Microsoft provides tools to do this) with malicious voting software.
2. Gain access to a terminal in debug mode.

Potential Mitigation:

1. Election officials: Rigorously enforce rigid media protection control procedures.
2. Vendor: Sign and verify the file contents with a secure signature scheme.

3.7.2.3 Bootloader Automatically Runs .ins Files on the Memory Card (#8)

The bootloader also uses .ins files to install new operating system images. Specifically, if the machine is in debug mode and the bootloader finds the file "avtsx.ins" on the memory card, it will attempt to install a new operating system image contained within the file. The .ins files are now signed with an RSA signature variant, but the signature is not implemented properly and can be forged. The details of the signature and its weakness are described in Section 3.5. An attacker could therefore create a Windows CE image of her choice that properly authenticates. Our code for forging signatures on .abo scripts can be ported to do the same for .ins files because they are authenticated by the same faulty verification code.

Attack Prerequisites: In order to exploit this vulnerability, the attacker must:

1. Generate a custom Windows CE operating system image (Microsoft provides tools to do this) with malicious voting software and insert it into an "avtsx.ins" file.
2. Gain access to a voting terminal in debug mode or gain several minutes of access to any terminal.
3. Insert a new memory card with the custom "avtsx.ins" file.

Potential Mitigation:

1. Election officials must ensure that debug mode is disabled on all voting terminals.
2. Election officials must strictly monitor access to voting terminals.
3. Vendor: Properly cryptographically sign and verify all .ins file before running them.

3.8 AV-OS Software Issues

3.8.1 AV-OS Firmware Faults

3.8.1.1 Leaks Memory Card Contents (#56)

In this vulnerability, an attacker uses a designed machine function in precisely the way that it was intended. That is, the attacker can copy the memory card contents to her own laptop by connecting the laptop to the optical scanner serial port or phone line, turning the machine on, entering diagnostic mode by simultaneously pressing the "yes" and "no" buttons, and selecting a menu option to dump the memory card's contents. The attacker's only technical contribution is establishing a routine terminal emulation to synchronize the laptop with the Optical Scanner, which is a standard function. Alternatively, the modem can also be used to dump the data.

We connected the optical scanner to a Windows XP computer and we were able to dump the contents of the card onto the computer using HyperTerminal. Though our experiments only dumped the first 4kB of the 128kB memory card, the data included all of the ballot definition and scripts.

Attack Prerequisites: In order to exploit this vulnerability, the attacker must:

1. Have the knowledge of the exploit.
2. Have unsupervised access to a terminal.

Potential Mitigation:

1. Election officials: Rigorously monitor access to the optical scan machine.
2. Vendor: Allow memory card dumps only from supervisor mode.

3.8.1.2 Supervisor PIN Not Cryptographically Protected (#57, 92)

The supervisor PIN was removed from the source code and an encoded version is now placed on the memory card. When a user wants to enter supervisor mode, the following authorization protocol takes place:

- The shuffled PIN and a decryption key are read by the terminal from the election header fields on the memory card.
- The user enters her PIN on the machine.
- The shuffled PIN is decoded using the decryption key and compared against the PIN entered by the user.

The remaining weakness is that in this protocol, the key that reveals the PIN is also on the memory card, so anyone with access to a card and reader and knowledge of the shuffling algorithm can extract the shuffled PIN and the key and thus can decipher the supervisor PIN.

Attack Prerequisites: In order to exploit this vulnerability, the attacker must:

1. Have unsupervised access to a machine to dump a current memory card or access to a memory card itself.
2. Have the necessary equipment to read from the memory card.
3. Have knowledge of the key shuffling algorithm.

Potential Mitigation:

1. Election officials: Exercise rigorous control procedures over voting terminals and removable media.

2. Vendor: Securely encrypt and authenticate the PIN and deliver the key to the machine through a medium that is as inaccessible as possible to most people. Ideally, the decryption key should be stored in secure hardware, and unless secure hardware is used, there is a bootstrapping problem that creates a fundamental limitation in the privacy of the PIN.

3.8.1.3 No Authentication Between GEMS and the Terminal (#58)

The primary concern with this vulnerability is if the devices were to connect across the Internet. The vendor strongly states that no such connection occurs. If the connectivity is through a serial connection with trustworthy technicians utilizing the connection, the potential impact is largely negated.

However, if the connection is via a modem, a man in the middle attack may be possible. The vendor wrote code to mutually authenticate the GEMS server and optical scan terminal, employing a home-grown encryption algorithm. Much like the protection of the Supervisor PIN, described in Section 3.8.1.2 above, critical components are provided in encrypted form within messages that contain the decryption key. This practice does not provide the intended security properties. More specifically, the protocol relies on the exchange of a password that is encrypted using weak encryption functions and the keys are exchanged in the clear at the beginning of each handshake, leaving the protocol vulnerable to an eavesdropping attack.

Attack Prerequisites: In order to exploit this vulnerability, the attacker must:

1. Have access to the memory card or a mechanism to eavesdrop on the communications.
2. Have knowledge of the communication protocol.

Potential Mitigation:

1. Elections officials: Never connect to the AV-OS terminals to the Internet or other untrusted networks.
2. Vendor: Remove code supporting networking connections or require use of a secure, mutually authenticated protocol, such as SSL.

3.8.1.4 Attacker Can Hide Preloaded Votes (#59, 85, 89, 90, 94, 95, 96)

The original attack has been largely mitigated. However, it is still possible to load selected vote counts on a memory card that the terminal will recognize if inserted and it is still possible to forge AccuBasic Interpreter scripts. We constructed four exploits of this vulnerability in the lab.

1. We prepared a memory card with preloaded votes and inserted it into the terminal after the zero report printed. The machine accurately displays the number of ballots that have been cast (or loaded), which could be detected by a conscientious precinct clerk or poll worker. The attack requires less than a minute of unfettered access to the terminal at the beginning of the voting day.
2. In our second exercise, we extracted a memory card from the terminal after several votes had been registered, modified the counters to redistribute votes, but also to maintain the same total vote count, and reinserted the memory card in the terminal. This process took several minutes of access to the terminal, and if it were accomplished in a real election, it would be detectable by a recount of the paper ballots.
3. In the third exploit, we prepared a memory card with a predetermined vote count and waited until the number of votes on the card were normally registered on the terminal. We then replaced the official memory card with our forged card. The machine display reflected the correct, expected number of votes cast and the precinct count printout showed the specific candidate votes that we injected. This attack took only a few moments access to the terminal, though it demands that the memory card be inserted when the appropriate number of ballots have been cast. If this attack were conducted by a poll worker who could switch cards at the precise point when the machine count

matched the malicious card's vote count, it is unlikely that the attack would be detected unless a recount occurred. In a recount, the count would be corrected based on the paper records.

4. We constructed an interpreter script that disregards the counters on the memory card and prints the counts that we preload. This attack would normally be detected and corrected when the electronic vote count is transferred to the SoE. If the discrepancy between the two electronic counts is noted, the correct count could then be confirmed by a recount of the paper ballots.

There are two software issues related to this vulnerability. The first is the RSA signature described in Section 3.5 above. As we demonstrated in the lab, without knowing any key, we can forge a signature on essentially any functional interpreter script that we desire. This issue applies to bullet 4 above.

The second issue concerns the fact that vote counts on the memory card are not properly protected. This allowed us to edit fields to change votes with our only challenge being to appropriately set error correcting checksum values. This issue applies to bullets 1, 2, 3, and 4 above.

Attack Prerequisites: In order to exploit this vulnerability, the attacker must:

1. Have unsupervised access to the optical scanner.
2. Have access to uncommon memory cards and equipment to read and write them.
3. Have significant computer skill and access to the proper ballot definition.

Potential Mitigation:

1. Election officials: Restrict access to removable media, particularly memory cards.
2. Vendor: Authenticate the contents of the memory cards including vote counts. (More details on this are suggested in Section 3.6.)
3. Vendor: Employ secure audit logging techniques as developed within the security and applied cryptography literature.

3.8.1.5 Vote Counters Are Not Directly Checked for Overflow (#85, 89, 95, 96)

Candidate ballot counters are protected from overflow by a total ballot counter. This is an imprecise, and apparently an overly strict approach that detects when a machine has reached the limit for overall votes. Since individual candidate vote counters can hold as many votes as the total ballot counter, this approach appears to prevent overrunning all vote counters.

This check depends on an invariance between the protected and protecting variables. It assumes that votes are always recorded through the voter interface and processed by the firmware that ensures the necessary consistency. This works if the counts start at zero and votes are always recorded through the intended paper ballot interface. However, this may not be the case. Specifically, Hursti programmed a memory card to effectively store a negative vote count by storing a very large vote count, which then overflowed [5], though this did not cause the total ballot count to overflow.

Other consistency checks, such as whether the total number of votes in a race equals the sum of the votes for each candidate in the race are also now used, and may prevent a possible exploit of this flaw.

While we do not see any immediate attacks caused by this issue, it invites consistency attacks, for example, that may attempt to subtract votes from a candidate with zero votes by manipulating the memory card. There are mechanisms that appear to prevent attacks that we examined in the code. However, defensive programming practice suggests implementing overflow protection at the point where the variable is computed.

Attack Prerequisites: In order to exploit this vulnerability, the attacker must:

1. Find a malicious means of overflowing a vote counter.
2. Acquire a memory card and knowledge of its structure.
3. Create a memory card such that the appropriate vote counter would overflow.

4. Insert the memory card into an optical scan machine.

Potential Mitigation:

1. Election officials: Strictly monitor access to optical scan machines and memory cards.
2. Vendor: Directly verify that vote counters do not overflow every time they are incremented.

3.9 AccuBasic Interpreter Faults

3.9.1 Error Checking is Inadequate (#67, 68, 81, 82)

The interpreter does not provide descriptive prompts as to errors that were encountered. Most of the prompts are along the lines of "worked/didn't work". Once the main interpretation has started, there seems to be no printing of error messages to indicate problems.

While we did not develop an example attack, this design weakness might make an attack indistinguishable from a random error if such an attack were employed.

3.9.2 Error Codes Returned by the AV-OS System are Ignored (#83)

A specific function provides the entry point to the AccuBasic interpreter and returns a status code indicating the success or failure of script interpretation. In all instances within the AV-OS code where this function is called the return value is ignored.

As above, we did not identify any direct exploits resulting from this flaw, but the lack of error checking may make it easier to execute other attacks without detection..

3.9.3 AccuBasic Scripts Can Be Misused (#62, 86, 87, 97)

The interpreter allows AccuBasic code to perform conditional operations based on comparisons of data including vote counts, time, date, candidate names and any other data to which it has access. This may enable an attacker to hide exploits by presenting the user with conditional information.

Additionally, AccuBasic scripts can interact with the user through the terminal's LCD screen. This may allow a malicious script to use social engineering techniques to enable attacks. For example, an attacker may preload a script that prints a message such as "Bad memory card. Please insert another" in order to accomplish the memory card swapping attack described in Section 3.8.1.4 above.

To illustrate this flaw at a basic level, we wrote AccuBasic scripts that print text of our choice and that also passed signature verification; for additional details see Sections 3.5 and 3.8.1.4.

3.9.4 Public Key Hard-Coded into the Source (#91)

Embedding the public key in the source code does not pose a direct security problem. However, having the public key hard-coded into the source code prevents the vendor from routinely changing the public/private key pair. On the other hand, hard-coding the public key also prevents an adversary from easily changing it. The particular trade-offs between these two approaches should be evaluated in the broader context of election systems.

3.9.5 Unchecked String Operation: Allows Overwrite of Stack Memory (#113, 114)

The vulnerable string operations for these two flaws still exist in the code as described in the VSTAAB report [2]. However, the repairs made to correct other flaws mitigate the vulnerability introduced by these flaws. So the flaws are still there, but the corresponding vulnerability (based on another flaw) seems to be fixed, as we report in Table 1 below. Due to the level of detail involved, we defer further discussion to the private Appendix B.

3.10 GEMS Server Faults

3.10.1 AccuBasic Scripts are Not Authenticated on the GEMS Server (#118)

GEMS itself does no checks to the AccuBasic byte code (.abo) because they rely on the (problematic) RSA signatures (see Section 3.5) on the OS machine to verify the AccuBasic code. GEMS is apparently backwards-compatible with versions of the AV-OS/AccuBasic that did not contain signatures. Any .abo code could be placed on the GEMS server and selected to send to the AV-OS if there are no procedural/physical safeguards. This would be a vector for loading a malicious .abo script, although it is more complicated than simply writing one to the memory card (due to formatting differences in the .abo files), and requires access to GEMS rather than the card itself.

Attack Prerequisites:

1. Have access to the GEMS folder that contains the .abo files and to the option within GEMS itself to select which .abo file will be sent.
2. Create a malicious AccuBasic script, which would require knowledge of the RSA signature flaw. (See Section 3.5.)

Potential Mitigation:

1. Election Officials: Safeguard access to the GEMS server.
2. Vendor: Use a secure signature scheme to sign and verify the AccuBasic scripts.

3.10.2 Password Does not Protect Access to GEMS or Audit Logs (#43, 123)

The GEMS password authentication process can be defeated with a simple attack, given a few moments access to the computer. This attack is publicly known and revolves around the way the password is stored in a Microsoft Access database. We constructed a proof of concept exploit for this vulnerability to confirm its applicability.

The same vulnerability occurs with the audit logs of the TSX.

Attack Prerequisites: In order to exploit this vulnerability, the attacker must:

1. Have knowledge of the exploit. (This attack is described in detail on a public Internet web site.)
2. Have remote or local access to a GEMS server

Potential Mitigation:

1. Election officials: Rigorously protect physical access to GEMS servers.
2. Election officials: Never connect GEMS servers to the Internet or an untrusted network.

3.10.3 Incomplete Implementation of the SSL Protocol (#124)

The use of SSL with the GEMS server is optional. The user is presented a dialog box that allows her to choose both whether or not she wants to use SSL and whether or not she wants to require authentication of the client. Both of these should be made mandatory. The corresponding setting on the TSX client is stored in the machine registry.

When SSL is used, if the option to require authentication is set, GEMS does require authentication from the client. Additionally, the software includes the option to connect using an early version of SSL (V2) that is subject to downgrade attacks.

We also note that the GEMS server does not use SSL when communicating with the AV-OS. Rather, they employ a vendor generated communication protocol for the AV-OS that does not include any SSL calls. (See Section 3.8.1.3 above.)

The TSX will verify that it is establishing an SSL connection with a GEMS server, but not which GEMS server. Similarly, the GEMS server will verify that it is establishing a connection with a GEMS client, but not which GEMS client. While this is an improvement over the SSL implementation evaluated by the RABA Study [11], this implementation is still vulnerable to man-in-the-middle attacks assuming that multiple states or election districts use the same public keys for the Diebold certificate authority (which we assume to be the case since they are hard-coded into the software). For example, an insider from District A with sufficient infrastructure access could use a copy of her GEMS private key and a TSX client's private key to mount a man-in-the-middle attack against the TSX machines and GEMS server in District B.

Attack Prerequisites: In order to exploit this vulnerability, the attacker must:

1. Have knowledge of the weakness and exploit techniques.
2. Have sufficient knowledge of the operating environment.
3. Possess the proper equipment and infrastructure access.

Potential Mitigation:

1. Election officials: Never attach GEMS, AV-OS, or TSX terminals to the Internet or an untrusted network.
2. Vendor: Require client authentication whenever connecting to a remote host.
3. Vender: Verify that the client and server are communicating with the intended parties, not simply a certain class of devices.

4 Non-Pertinent Faults

The team ran automated analysis tools over the code base. The non-pertinent flaws that we identified are contained in the private Appendix C.

Additionally, flaw #44 noted that the code contains third party components. We also found such references in the source that we reviewed. Since we did not find discussions of specific flaws related to third party software in the previous reports that we reviewed, we do not include a discussion of these third-party products in the public portion of this report. We list the third party products that we identified, along with their versions, in private Appendix C.

5 Conclusions

Electronic voting systems offer tremendous opportunity to expand accessibility and reduce costs even in the face of increasing safety and accuracy demands. Conversely, they also offer virtually unbounded opportunity to manipulate elections if they are not properly secured. Code review is one step in this process.

This report presents the background, organization, process, findings, and opinions of our software code review. We conclude with the following summarizing issues.

5.1 Flaw Retention Overview

The team was tasked to determine if flaws identified in the literature remain in the software version submitted for certification. Table 1 reflects our precise answer to this question under the definitions and terminology outlined in Section 2.2.

Applicable Component	No Change	Improved	Fixed
Physical Security	11, 55		48, 50, 51
OS Firmware	56, 62, 64, 92, 93	57, 58, 59	84, 85, 88, 89
OS Interpreter	81, 82, 83, 8, 86, 87, 91, 97, 98	90, 94, 96, 105, 113, 114	80, 95, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112
Bootloader	3, 6	4, 5, 7, 8	1
TSX Firmware	15, 16, 17, 19, 22, 24, 26, 28, 30, 32, 40, 41, 42, 43, 44, 51	13, 14, 18, 20, 21, 23, 25, 27, 29, 31, 34, 35, 45, 46, 49, 69	24, 33, 36, 38
TSX Interpreter	67, 68		65, 66, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79
GEMS	123	118, 124	

Table 1. Flaw Categorization by Flaw Number (See Appendix A)

5.2 The Optical Scan Software

The vendor implemented integrity verification and encryption to address some of the reported flaws. In many cases, unproven designs were applied that left vulnerability where stronger tools could provide measurable protection. We demonstrated several attacks in our laboratory. For one attack, we exploited a flaw in the AV-OS RSA implementation to create a custom AccuBasic script with a forged RSA signature. Our custom AccuBasic script will cause the AV-OS to output incorrect election totals. Many of the remaining vulnerabilities are detectable and correctible through normal elections procedures and would certainly be detected during audits and recounts.

5.3 The Touch Screen Software

The vendor has made improvements to address many of the specific flaws expressed to date. Despite these improvements, many flaws remain in the TSX system. In our opinion, the vendor could make significant additional improvements by employing strong signature protocols and improved key management techniques.

5.4 Removable Media

All removable media associated with electronic voting systems is highly sensitive, and must be protected year-round with the same status and priority as voted ballots. Unsupervised access to an item as simple as a TSX supervisor card and its accompanying PIN allows an individual to create valid voter cards. In the wrong hands, these cards can cause significant damage to a county's vote count validity.

5.5 "Security Through Obscurity" is Not Sufficient to Protect Voting Systems

Two important transitions heightened this realization: (1) the rapid expansion of computer use in elections; and (2) the explosive growth of computing in general accelerated by the explosion of the Internet. The processes and methods behind these systems are often publicly exposed, as was witnessed when the Diebold Touch Screen source code surfaced on the Internet in 2003 [14]. The exploit described in Section 3.5 clearly illustrates the convergence of theory and practice in electronic voting security. The signature scheme applies well-known cryptographic primitives and is employed in an efficient and intuitively appealing, though home-grown, approach. However, security theory and mathematical analysis revealed a simple, efficient algorithm that allows an attacker to forge essentially any arbitrary message, thus easily subverting this approach.

5.6 Engineering In Security

As with any system, it is most effective and least costly to incorporate necessary components up front rather than adding them later. Just as a house that is designed and constructed with a garage may cost more up front, its overall cost is generally much less than the cost of the house without the garage, and the additional cost of adding the garage later. Worse yet, if the original design did not include provisions for a prospective garage, adding one may not be possible at all.

The issue is similar with voting systems security. Systems designed with security in mind have a much better chance of addressing evolving threats than after-the-fact security response, particularly when prospective security features were not considered or provided for in the original design.

This report amplifies this challenge. A primary issue in our analysis is the impact of regression faults, or faults that occur as a result of modifying a system. In this case regression faults leaked into the system as the result of modifications to correct other faults. Implementing security is much easier if done from the ground up.

5.7 Persistence of Vulnerability

The flaws examined in this study have been published in public reports, several of them over three years ago. Many of these flaws appear in multiple studies, reiterating that they existed. While the vendor has fixed many of these flaws, many important vulnerabilities remain unaddressed, or the attempted fixes leave vulnerabilities in place. Solutions to most of the issues in these systems have been studied in depth and added to the public knowledge base. Although such information is available in a variety of resources, as are knowledgeable professionals, studies have repeatedly shown that home-grown security mechanisms rarely provide the intended security properties. Home grown mechanisms cannot replace systematically developed and time tested security solutions.

5.8 Key Management is a Difficult [Voting System] Problem

It is a well-known that key management is one of the most challenging issues in using cryptography to protect information systems. Whether a system embeds public keys in the source code or attempts to bootstrap a private key separate from application initialization, the challenges of managing keys that enable even mutual authentication are great.

5.9 Issues to Improve Voting System Security Evaluation

With the growth of electronic voting systems, software review is becoming more and more commonplace. We offer the following observations that may help future analysis efforts.

5.9.1 Require Vendors to Deliver a Complete Development Environment with Software

Software review and proof of concept testing are resource intensive processes that demand domain appropriate skills. Conversely, software development environments can vary greatly. In order to reduce both spin up and proof of concept construction time in the review process, states should require vendors to file complete development environments with their systems. The environment delivered should enable the state, and any review teams, to rebuild the certified binaries. During a review, these should be delivered with the source code to the review team.

5.9.2 Require Vendors to File Design Documentation with the Department of State

Similarly to having development environments, software review can be simplified if accurate, detailed design documentation is available. States should require such documentation and make it available to any software analysis effort.

5.9.3 Independent Security Evaluation

We recommend that voting systems be subjected to ongoing reviews so that flaws can be discovered before an election and before a system is adopted, and proposed fixes can be evaluated for completeness, rather than after an election outcome comes into doubt.

As another efficiency consideration, it is unlikely that all states are fiscally capable of conducting rigorous voting system analysis. Thus, we emphasize that a uniform testing process, such as the one presently under construction by the United States Elections Assistance Commission (EAC) Technical Guidelines Development Committee (TDGC), be adopted and that standard practices for sharing review results among states be implemented.

6 Acknowledgments

As part of our work we used automated source code analysis tools Fortify Source Code Analysis (SCA), made by Fortify Software and Coverity Prevent by Coverity in order to assist with the code review process. Fortify Software donated the tool to us free of charge for use on this project and we thank them for their contribution. We note that one member of the team (Bishop) is on Fortify Software's Technical Advisory Board and Coverity is a SAIT Laboratory partner.

We also acknowledge the major contributions of the ACCURATE Center in this review. We note that the final report's first author is an ACCURATE member and we received significant advice and resources from several ACCURATE members throughout this process, for which we are grateful.

We thank Doug Jones, Avi Rubin, and David Jefferson, who provided several documents that we used in research for this work.

A special thanks to Avi Rubin and Breno de Medeiros for their comments on a draft of this report.

7 References

1 "Software Review and Security Analysis for Diebold Voting Machine Software.", Joint Florida Department of State and Florida State University Statement of Work, May 14, 2007.

2 David Wagner, David Jefferson, and Matt Bishop, "Security analysis of the Diebold AccuBasic Interpreter", Voting Systems Technology Assessment Advisory Board (VSTAAB), University of California, Berkeley, February 14, 2006.

3 Diebold Election Systems, Inc., "Source Code Review and Functional Testing," CIBER, Inc., 7501 South Memorial Pkwy, Suite 107, Huntsville, AL 35802, February 23, 2006.

4 Ariel J. Feldman, Alex Halderman, and Edward W. Felten, "Security Analysis of the Diebold AccuVote-TS Voting Machine", Center for Information Technology Policy and Dept. of Computer Science, Princeton University, September 13, 2006.

5 The Black Box Report, SECURITY ALERT: Critical Security Issues with Diebold Optical Scan Design, July 4, 2005.

6 Harri Hursti, "Diebold TSX evaluation: Critical security issues with Diebold TSX.," Available at http: www.bbvdocs.org/reports/BBVreportIIunredacted.pdf, May 2006.

NF

7 A. Kiayisas, L. Michel, A. Russell, A. A. Shvartsman, "Security Assessment of the Diebold Optical Scan Voting Terminal", UConn VoTeR Center and Department of Computer Science and Engineering, University of Connecticut, October 30, 2006.

8 Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach, "Analysis of an Electronic Voting System", IEEE Symposium on Security and Privacy, May 9-12, 2004, pp. 27-40.

9 Maryland State Board of Elections, "Response to: Department of legislative services trusted agent report on Diebold AccuVote-TS voting system", Available at http://mlis.state.md.us/Other/voting system/sbe response.pdf, January 2004.

10 Ohio Secretary of State, "Direct Recording Electronic (DRE), Technical Security Assessment Report", Compuware Corporation, 1103 Schrock Road, Suite 205, Columbus, Ohio 43229, November 21, 2003.

11 RABA Technologies. Trusted agent report: Diebold AccuVote-TS voting system. Available at http: www.raba.com/press/TA Report AccuVote.pdf, January 2004.

12 Science Applications International Corporation, "State of Maryland Risk Assessment Report, Diebold AccuVote-TS Voting System and, Processes, SAIC-6099-2003-261, September 2, 2003.

13 Xiaoyun Wang and Hongbo Yu, "How to Break MD5 and Other Hash Functions", EUROCRYPT, May, 2005, pp. 19-35.

14 Bev Harris, "Black Box Voting: Vote Tampering in the 21st Century". Elon House/Plan Nine. July, 2003.

15 Collaborative Audit Committee, "Collaborative Public Audit of the November 2006 General Election". Available at http://urban.csuohio.edu/cei/public_monitor/cuyahoga_2006_audit_rpt.pdf, April 2007.

Appendix A Flaw List

ID	Description	Source	Page #
1	Machine boots into Windows explorer rather than BallotStation if explorer.glb is found on memory card	[4]	5
2	Machine door lock is consistently pickable in <10 seconds	[4]	5
3	fboot.nb0 can be used to load malicious software	[4]	5
4	fboot.nb0 can be overwritten without authentication or integrity checks	[4]	16
5	Denial of Service with PowerOffSystem() API	[4]	16
6	Bootloader replaces itself with eboot.nb0 or fboot.nb0 if found on memory card on boot, no authentication	[6]	5
7	Bootloader replaces OS with nk.bin (or some other nk.xxx's) if found on memory card, no authentication	[6]	7
8	Bootloader "runs" any .ins files in memory card on boot after prompting user, no authentication	[6]	8
9	Back of machine casing comes off and leaves seals in tact	[6]	9
10	Hidden SD memory card slot inside the machine case	[6]	10
11	Jumpers on machine motherboard enable debug features	[6]	10
12	Hidden, voter accessible button on back of case	[6]	11
13	Lack of smartcard authentication (use a hardcoded password)	[8]	9
14	PIN sent from smartcard to terminal in cleartext	[8]	11
15	Insecure file mount, does not ensure writing to removable media	[8]	12
16	Unprotected system configuration file	[8]	12
17	Protective counter stored in mutable file	[8]	12
18	Ballot definition unprotected and unauthenticated	[8]	13
19	Candidate information is not stored in the results file (only vote counts)	[8]	14
20	Impersonate a legitmate voting terminal, no authentication and data can be gathered from ballot definition files	[8]	14
21	Cryptographic key management beyond hard-coded DESKEY	[8]	14
22	DES CBC encryption uses constant 0 for IV	[8]	15
23	No integrity protection of stored vote counts and data	[8]	15
24	No sequence numbers stored with votes (to help detect record deletion, may be problematic with respect to privacy)	[8]	15

25	No integrity checks, authentication, or encryption on votes transferred to the backend server	[8]	16
26	Votes are written serially	[8]	16
27	Bad RNG for randomizing vote order	[8]	16
28	Audit log problems: no consistency for what is logged, does not verify that printer is attached before writing to it	[8]	17
29	Complicated, undocumented code segment	[8]	18
30	Poor change-control process info in code comments	[8]	19
31	Missing design documents	[8]	19
32	Code comments suggesting fixes, not documented further	[8]	20
33	1.06 Parameters and return values of functions are generally not commented	[10]	32
34	1.26 DES encryption key is hard-coded	[10]	36
35	1.27 Data is not encrypted when transmitted over data link	[10]	36
36	1.31 The supervisor's password is hard-coded	[10]	36
37	2.06 System can be locked up by pressing F4 and choosing to open BallotStation.exe	[10]	38
38	2.09 PIN for all smart cards is the same and the factory default (1111)	[10]	39
39	2.14 Several TCP/UDP ports are open	[10]	40
40	Can make counterfeit voter and supervisor cards	[10]	52
41	Can vote multiple times with a card writer	[10]	52
42	PCMCIA cards are not encrypted	[10]	52
43	Microsoft Access database has no password protection (used for ballot definition, audit logs, and tally results)	[10]	52
44	1.16 Software contains third party components	[10]	56
45	Digital certificates only exist at the servers and these are neither signed nor authenticated by the AccuVote terminals	[11]	9
46	No GEMs authentication by the AccuVote-TS terminals	[11]	9
47	AccuVote-TS terminals have two locking bays - all terminal locks are identical	[11]	18
48	With a keyboard, attacker can access options (unnecessary test code) "Save As", "Finish Recording", and "Open" to overwrite results and audit file	[11]	18
49	Only files with cryptographic protection on PCMCIA card are the results file and the audit file (extent of protection, even on these, may not be great)	[11]	18
50	Can attach a keyboard to the terminal and access functionality in the software that allows the attacker to view the entire	[11]	18

	directory tree on the machine's internal memory and on the PCMCIA card		
51	Attacker can load a PCMCIA card with an update file	[11]	18
52	Training does not include an information security component	[12]	5
53	No documentation that identifies the process for maintaining access controls	[12]	7
54	Executes code (Accubasic object bytecode) off the memory card	[5]	1
55	Machine enters diagnostic mode if 2 buttons are depressed when it is powed on, no password needed (allows reinitializing of the machine and its clock, for example)	[7]	6
56	Leaks memory card contents	[7]	6
57	Supervisor PIN not cryptographically protected	[7]	7
58	No authentication between GEMS and the terminal	[7]	9
59	Executes code (Accubasic object bytecode) off the memory card	[7]	4
60	Machine allows multiple feeding of ballots due to feed sensor position	[7]	12
61	AccuBasic interpreter firmware chip is designed to be replaceable	[7]	13
62	Memory card access works as an extension of main memory rather than as a file system	[2]	9
63	Machines do not allow for counting of ballots but only ballot pages (report after 2006 election)	[15]	36
64	Machines do not report data at a machine level of precision, only a precinct level (report after 2006 election)	[15]	36
65	Three types of tokens used to potentially read and modify data in global memory (no specifics given)	[3]	7
66	Bounds on the heap and stack segments do not appear to be checked	[3]	9
67	Error checking is inadequate for identifying and recovering from failures in a damaged or disfunctional environment	[3]	11
68	Error handling makes it difficult to differentiate between malicious activity and failure	[3]	11
69	The contents of the smartcards are neither encrypted nor digitally signed	[11]	17
70	W1 Array bounds violation: Overwrite any memory address with a 4-byte value that the adversary has partial control over. Allows attacker to inject malicious code and take complete control of the machine	[2]	15
71	W3 Input validation error: Choose any memory location and begin executing it as .abo code; could be used to conceal malicious .abo code in unexpected locations, or to crash the machine	[2]	15
72	W6 Array bounds violation: Overwrite any memory location with any desired value. Allows attacker to inject malicious code and take complete control of the machine	[2]	15
73	W7 Buffer overrun: Corrupt memory, crash the machine	[2]	15
74	W8 Buffer overrun, integer conversion bug: Corrupt memory until the machine crashes	[2]	15

75	W10 Buffer overrun: Overwrite return address on the stack. Allows attacker to inject malicious code and take complete control of the machine	[2]	15
76	W11 Array bounds violation: Information disclosure: read from potentially any memory address. Crash the machine	[2]	15
77	W12 Array bounds violation: Writes any 4-byte value to any address. Allows attacker to inject malicious code and take complete control of the machine	[2]	15
78	W13 Array bounds violation: Information disclosure: read a 4-byte value from any address	[2]	15
79	W14 Pointer arithmetic error: Crash machine. Could begin interpreting random memory locations as though they were .abo code	[2]	15
80	Three types of tokens used to potentially read and modify data in global memory. (no specifics given)	[3]	7
81	Error checking is inadequate for identifying and recovering from a failures in a damaged on disfunctional environment	[3]	11
82	Error handling makes it difficult to differentiate between malicious activity and failure	[3]	11
83	Error codes returned by the AV-OS system are ignored.	[3]	11
84	Attacker can hide preloaded votes (exact mechanism not given)	[5]	8
85	Vote counter allows integer overflows	[5]	8
86	Allows programming conditional behavior based on time, number of votes counted, and others	[5]	8
87	Enables AccuBasic program interaction over the LCD screen (can pose as normal firmware, for example)	[5]	20
88	AV-OS fails to check that the vote counters are zero at the start of election day	[2]	18
89	The code does contain a check to ensure that it will not accept more than 65535 ballots. It manipulates vote counters values without first checking them for overflow as 16-bit if more than 65535 votes are cast, the vote counters will wrap	[2]	18
90	Cards' integrity is protected by symmetric MAC rather than a much more ideal pk signature	[2]	20
91	Default cryptographic keys that are hard-coded into the source code	[2]	20
92	PIN is stored in an obfuscated format, but this obfuscation offers limited protection due to reliance on hard-coded magic constants	[2]	21
93	No requirements documents, architecture documents, design documents, threat model documentation, or security analysis documents	[2]	24
94	Modify the vote counters on the memory card to pre-load it with some non-zero number of votes for each candidate	[2]	25
95	Replace the AccuBasic script with a malicious script that falsely printed a zero report showing zeros, even though the vote counters were in fact not zero	[2]	25
96	Attacker can maliciously preload some of the vote counters with fraudulent non-zero values	[2]	27
97	Enables AccuBasic program interaction over the LCD screen (can pose as normal firmware, for example)	[2]	28
98	Several fields not covered by checksums	[2]	29

99	V1 Array bounds violation: Overwrite any memory address within �215 bytes of the global context structure with a 2-byte value that the adversary has partial control over. Might allow attacker to inject malicious code and take complete control of the machine	[2]	14
100	V2 Format string vulnerability: Crash the machine; read the contents of memory within a narrow range	[2]	14
101	V3 Input validation error: Choose any location on the memory card and begin executing it as .abo code; could be used to conceal malicious .abo code in unexpected locations, or to crash the machine	[2]	14
102	V4 Array bounds violation: Memory corruption; crash the machine	[2]	14
103	V5 Double-free() vulnerability: Overwrite any desired 4-byte memory address with any desired 4-byte value. Allows attacker to inject malicious code and take complete control of the machine	[2]	14
104	V6 Array bounds violation: Memory corruption: overwrite any memory address up to 216 bytes after the global context structure with a 2-byte value that the adversary has no control over. Might allow overwriting vote counters	[2]	14
105	V7 Buffer overrun: Memory corruption; crash the machine	[2]	14
106	V8 Buffer overrun: integer conversion bug: Memory corruption: overwrite up to 215 consecutive bytes of memory starting at global context structure. Might allow attacker to inject malicious code and take complete control of the machine. Might allow overwriting vote counters. Information disclosure: read any memory location 215 bytes away from global context structure. Crash the machine	[2]	14
107	V9 Buffer underrun: Memory corruption: overwrite up to 215 consecutive bytes of memory extending backwards from the global context structure. Might allow attacker to inject malicious code and take control of the machine. Might allow overwriting vote counters. Information disclosure: read any memory location within this window. Crash the machine	[2]	14
108	V10 Buffer overrun: Overwrite return address on the stack. Allows attacker to inject malicious code and take complete control of the machine	[2]	14
109	V11 Array bounds violation: Information disclosure: read from potentially any memory address. Crash the machine	[2]	14
110	V12 Array bounds violation: Write any 2-byte value to any address up to 216 bytes after the global context structure. Might allow attacker to inject malicious code and take complete control of the machine. Might allow overwriting vote counters	[2]	14
111	V13 Array bounds violation: Information disclosure: Read any 2-byte value from any address up to 216 bytes after the global context structure	[2]	14
112	V14 Pointer arithmetic error: Crash machine. Could begin interpreting random memory locations as though they were .abo code	[2]	14
113	V15 Unchecked string operation: Machine might crash or become unresponsive	[2]	14
114	V16 Unchecked string operation: Overwrite stack memory. Might allow attacker to inject malicious code and take complete control of the machine	[2]	14

115	Some default values not found in the user's guide	[3]	13
116	Default values defined in the source code	[3]	13
117	Contains "malign" undocumented functions	[3]	13
118	AccuBasic executable files (.abo) on the server are not authenticated	[5]	9
119	Failure to regularly install MS Windows updates	[11]	20
120	Server enables "autorun" feature, thus software can be installed via CD by anyone with physical access	[11]	20
121	Open USB port on the back of the server	[11]	21
122	CD is Bootable	[11]	21
123	Database password and audit logs stored within the database itself	[11]	21
124	Incomplete implementation of the SSL 3.0	[11]	21
125	Lack of routine server security practices, such as firewall protection, etc.	[11]	21
126	BIOS not password protected	[11]	22