

Informal name	Code identifier	Summary
"uninitialized"	STAT_UNUSED	/* Empty formatted memory card. */
"downloaded"	STAT_DOWNLOADED	/* Downloaded memory card - pre-election mode. */
"election mode"	STAT_ELECTION	/* Election counting mode. */
(none)	STAT_ELECTION_DONE	/* Ender card fed, printing totals report. */
(none)	STAT_DONE	/* Post-election mode - ready for upload. */
(none)	STAT_UPLOADED	/* Upload done, ready for audit. */
(none)	STAT_AUDIT_DONE	/* Final audit report printed. */

Figure 1: The modes that the AV-OS memory card can be in. For each mode, we list the informal name we use in this report, the symbolic name found in the source code, and a brief description taken from comments in the source code.

requirements documents, architecture documents, design documents, threat model documentation, or security analysis documents—all of which would be present were high assurance development techniques used.

We also expect that if one were going to use high-assurance programming practices anywhere in a voting system, the interpreter would be one of the most likely places to use it. If high-assurance practices had been used during the design and implementation of the AV-OS and AV-TSx, the vulnerabilities we found would likely have been avoided.

**Finding 10** *The AV-OS is at risk from Harri Hursti's attacks no matter what state the memory cards are in when they are transported to the polls. Even if the memory cards are not put into election mode until the polls are opened, Hursti's attack is still possible.*

The AV-OS can be in one of several modes (e.g., pre-election, election mode, post-election). This is determined by a value stored on the memory card. It has been suggested that, if election workers were to wait to put the card into election mode until polls are opened, this might provide some level of defense against Hursti's attack. We find that this scheme does not, in fact, provide any useful protection.

Because the mode is stored on the memory card, whether or not the memory card is in election mode while in transit makes essentially no security difference. An attacker who can modify the object code and vote counts on the memory card (as Mr. Hursti did) could just as easily modify the election mode indicator too. In addition, all of the vulnerabilities described earlier (due to bugs in the code) are still exploitable, no matter what mode the memory card is in.

A detailed technical analysis of the election mode issue can be found in Section 4.1.

#### 4.1 Technical details: Election mode and the AV-OS

In the AV-OS, memory cards can be in one of 7 modes, indicated by a field stored on the memory card (namely, `mCardHeader.electionStatus` in the source code). The states are documented in Figure 1. The mode of the memory card at the time when the machine is booted determines what functions the AV-OS will execute. The AV-OS also updates the mode of the card in response to operator input.

The memory card also contains many counters, including candidate counters (which contain, for each candidate, the number of votes cast for that candidate), race counters (which contain, for each race, the number of votes cast in that race), and card counters (which contain the total

number of "cards cast" or, in other words, the number of ballots scanned). In each case, there are three values stored: the number of absentee votes, the number of election-day votes, and the total number of votes (which should be the sum of the previous two values). This reflects the fact that the machine can be set into a mode to count absentee votes or to count at the polling place. Note that there is some redundancy among these counter values: for instance, under normal operation, if Smith and Jones are the only two candidates in one race, then the race counter should equal the sum of Smith's candidate counter and Jones' candidate counter.

In Harri Hursti's demonstration, apparently the memory card was already placed into "election mode" before Hursti was given the card. It has been suggested that if the card had been in one of the two pre-election modes ("initialized" or "downloaded") when it was given to Hursti, then the Hursti attack would not work, because the process of placing the card into "election mode" would cause the vote counters to be zeroed.

Recall that Hursti's attack, in its most dangerous form, involved two components: (a) modifying the vote counters on the memory card to pre-load it with some non-zero number of votes for each of the candidates (e.g., +7 votes for Smith and -7 votes for Jones); (b) replacing 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. The ability to print a false zero report enabled Hursti to conceal the fact that he had stuffed the digital ballot box. This attack was demonstrated in a scenario where the card was set into "election mode" in the warehouse, before there was an opportunity to tamper with its contents. Might it perhaps be possible to defeat this attack if memory cards were left in pre-election mode at the warehouse, transported in this mode, and then poll workers were asked to set the card to "election mode" at the opening of polls? The idea is that, in the process of setting the card into "election mode", the AV-OS will zero out the vote counters on the card, thereby undoing any pre-loading of the memory card with fraudulent votes that might have occurred before that point. We were asked to characterize the behavior of election mode and investigate whether defenses of this form would provide any value in defending against Hursti's ballot stuffing attack.

**Boot behavior.** When starting the AV-OS machine, the operator has the option of holding the YES button or the YES and NO buttons (simultaneously) to execute special diagnostic, supervisory, and setup functions. When the machine boots, it will enter one of several modes, depending on how it is started up:

- If the operator holds the YES and NO buttons down while machine is booting, the machine enters diagnostics mode. In diagnostics mode, the operator can set the clock, dump the memory card image via a serial port, and test various physical components of the voting machine.
- If the operator holds only the YES button and the card is initialized (i.e., in any state other than "initialized", or in other words, `mCardHeader.electionStatus`  $\neq$  `STAT_UNUSED`), then it gives the operator the option to enter supervisor mode. To enter supervisor mode, the operator must enter the four digit PIN. In supervisor mode, the operator can modify the setup parameters, duplicate or clear the memory card, re-enter election mode after an "ender" card has been read, and reset the card to pre-election mode. In setup mode, the operator can change the phone number and configure the autofeeder and other physical devices.
- If the card is "uninitialized" (`mCardHeader.electionStatus` = `STAT_UNUSED`), the machine enters the aforementioned setup mode. Curiously, in this case the operator can enter setup

---

mode without entering a PIN. This means that it would be possible in this case to change the phone number it dials to transmit election results, without entering a PIN. (We are not aware of any California jurisdiction that uses the AV-OS's modem capabilities, so this is of little practical relevance in California.)

After these functions complete or if the operator chose not enter them, the machine displays

```
SYSTEM TEST
*** PASSED ***
```

and enters the main control loop. The main control loop works as follows:

- If the card state is “initialized” (STAT\_UNUSED) or “downloaded” (STAT\_DOWNLOADED), the machine executes pre-election functionality. Then, the machine goes back to the beginning of the loop.
- If the card state is in “election mode” (STAT\_ELECTION), the machine executes the election functionality and begins accepting and counting ballots. Then, the machine goes back to the beginning of the loop.
- If the card state is in any of the four post-election states (STAT\_ELECTION\_DONE, STAT\_DONE, STAT\_UPLOADED, or STAT\_AUDIT\_DONE), it executes the post-election functionality. Then, the machine goes back to the beginning of the loop.

**The behavior of the AV-OS.** We focus on three modes, “uninitialized”, “downloaded”, and “election mode”, and describe how the AV-OS behaves when loaded with a card in one of those three states.

If the card is “uninitialized”, the AV-OS enters a mode of operation for downloading data to the memory card. If the download is successful, the operator can print an optional zero report using the AccuBasic interpreter and then the card is set to “downloaded” mode. At this point, or if a card in “downloaded” state is inserted into the AV-OS at any time, the AV-OS provides the operator with the option of performing pre-election testing. Pre-election testing includes reading blank and full marked ballots, counting test ballots, moving the ballot deflector, testing upload of results, and printing test total and audit reports.

After testing, the machine prompts the operator if he or she wants to enter election mode. If the operator answers yes, then the card is set to “election mode” (i.e., the field `mCardHeader.electionStatus` on the card is set to the value `STAT_ELECTION`) and the AV-OS proceeds to clear the election counters. The step of entering election mode zeroes out the card counters, race counters, and candidate counters. In other words, it clears the number of votes registered for each candidate, the number of votes registered in each race, and the total number of “cards cast” (i.e., the number of ballots scanned).

After the counters are zeroed, the AV-OS machine begins executing election functionality. This code first checks the card for errors. Then, it checks if any ballots have yet been counted by checking a counter stored on the memory card containing the total number of ballots that have been counted (`mCardHeader.numBalCounted[CTR_TOTAL]`). If no ballots have been counted, the AV-OS invokes the AccuBasic interpreter to print a zero report (without first prompting the operator) and then begins to accept and count ballots. If this counter is non-zero, then it skips the zero report step and immediately begins to accept and count ballots.

---

**The proposed defense.** The Hursti attack works by maliciously preloading some of the vote counters with fraudulent non-zero values. It was suggested to us that having poll workers putting the card into election mode at the polling place would defeat this attack, but it wasn't clear whether this would involve delivering memory cards in the "uninitialized" or "downloaded" state.

We believe that transporting memory cards to the polling place in the "uninitialized" state doesn't make much sense. This would mean that the cards have not been programmed and initialized yet. It seems unlikely poll workers would be expected to program and initialize the memory cards.

Therefore, we assume that this procedural defense would involve initializing memory cards at the county headquarters, so that when they arrive at the polling place they are in the "downloaded" state. This means that the memory cards will have been programmed and initialized and are ready to be put into election mode when the AV-OS machine is turned on. After the machine starts and completes the optional diagnostics mode (see above), it will prompt the operator (in order) to:

1. To count test ballots (optional);
2. To move the ballot deflector (optional);
3. To test the upload option (optional);
4. To print a totals report (optional);
5. To print an audit report (optional);
6. To prepare for the election (optional);
7. To enter supervisor mode (optional).

To enter election mode, the operator should answer yes to the 6th prompt. At that time, the AV-OS machine will clear the counters (see above) and start counting ballots.

**Analysis.** Unfortunately, the proposed defense against Hursti's attack is not effective. An adversary with access to the memory card could maliciously set the card into election mode, by setting the `mCardHeader.electionStatus` field on the card to the value `STAT_ELECTION` using a hex editor or by other means. When this card is inserted into the AV-OS, the AV-OS will not clear the counters, because the card is already in election mode. (The counters are only cleared when a card in the "downloaded" state is inserted into the AV-OS and explicitly put into election mode by the operator.)

On first consideration, one might expect that this attack could be detected. After all, an observant operator might notice that he or she did not have to navigate the prompts to explicitly put the machine into election mode, and thereby may be able to deduce that the card must have already been in election mode. Unfortunately, we cannot count on this defense, because things are more complex than they may initially appear.

Recall that if the memory card is in election mode and if the counter for the total number of ballots scanned (`mCardHeader.numBallCounted[CTR_TOTAL]`) is zero, then the AV-OS will execute an AccuBasic script to print a zero report before accepting ballots. The operator is not prompted before the AccuBasic script begins running. Of course, if we assume that an adversary has unsupervised access to the memory card while it is in transport, the adversary could have replaced the

AccuBasic script on the memory card with a malicious script, and this malicious script will start running as soon as the machine is turned on. Moreover, recall that AccuBasic scripts have the power to issue prompts to the LCD display on the AV-OS. This means that an adversary could write a malicious script which simulates the prompts the operator is expecting to see, to provide the illusion that the card is not already in election mode. When the operator answers yes to the 6th prompt, the AccuBasic script can print a zero report and exit, and the machine will start counting ballots.

In this scenario, as far as the operator can see, the machine will behave exactly as it would if the card had started in "downloaded" mode and if the operator had put it into election mode, clearing the counters. Nonetheless, in reality nothing could be farther from the truth. In this scenario, the card has been tampered with to pre-load it with votes, to set it into election mode so that these vote counters won't be cleared, and the AccuBasic script on the card has been tampered with so that the operator won't notice anything unusual and the zero report will not show these pre-loaded votes.

This shows that it is possible for an adversary to tamper with the memory card in a way that cannot be detected by the operator and that bypasses the clearing of the vote counters. In other words, even if memory cards are not put into election mode until the opening of polls, the election will still be vulnerable to a variation on Harri Hursti's attack. Therefore, it is our conclusion that procedures based on putting the AV-OS into election mode at the start of the day cannot be counted upon to protect the AV-OS machine against the vulnerabilities Harri Hursti found.

## 4.2 Checksums

We were asked to investigate what checksums exist in the AV-OS and AV-TSx, what types they are, and what they cover. We discuss these issues next.

**Background.** A checksum detects *accidental* changes to data. It reduces a large amount of data down to a fixed size value. This provides a level of redundancy: if the data is changed, then the checksum almost always changes as well. Hence, the checksum may provide a way to detect the change to the data.

Note that checksums are used to detect accidental changes to data values, but they are not at all useful in detecting malicious change. An example of an accidental change is a faulty memory cell on the memory card. If it cannot properly store the value it is supposed to, the computed checksum of the data will not equal the stored checksum, and a problem will be detected. On the other hand, if an adversary changes the data as well as all copies of the checksum value, it will be impossible to notice that the data was modified.

The AV-OS uses 16-bit checksums: a checksum can take on one of 65536 different values. The AV-OS computes numerous checksums over the data structures stored on the memory card. These checksum values are stored on the card and are also available to AccuBasic scripts to be printed in reports. A properly implemented checksum would likely detect any accidental corruption of the election setup parameters. Alternatively, a checksum printed over a memory card's vote totals at the close of polls could be compared with the same value at the county election offices to detect changes to the vote totals.

**What is covered by the AV-OS checksums.** The AV-OS memory card contains quite a few checksums. We list them, and what they cover, below:

1. *Election checksum*: covers the password, and flags controlling machine.
2. *Precinct checksum*: covers a few fields describing the precinct: its number, check digit, number of voters, sequence number, and precinct ID string.
3. *Precinct-card checksum*: covers fields that tie the precinct to the card structures.
4. *Race checksum*: all fields governing the race.
5. *Race counters checksum*: covers the total number of votes for each race, write ins, over-votes, under-votes, and blank votes.
6. *Candidate checksum*: covers the candidate number and party number.
7. *Candidate counters checksum*: covers all fields in the candidate structure.
8. *Card checksum*: covers all fields in the card.
9. *Card counters checksum*: covers the precinct number, card number, number of over-votes, under-votes, and blank votes for each card-counter.
10. *Voting positions checksum*: covers all fields governing where the candidate structure is.
11. *Text checksum*: covers all text fields (election title, vote center, vote date, straight party options, address, district name, race titles, and candidate names).
12. *Audit log checksum*: not used.

In summary, only some of the election setup parameters are covered by the AV-OS checksum. For example, the voting type field in the precinct (which governs whether it is early, absentee, or precinct voting) is not covered by any checksum. Additionally, the audit log is not covered by any checksum. It is difficult to determine how modifications to the fields not covered by the checksums could cause adverse effects, though it is a source of minor concern. Ideally, these checksums would cover all of the election parameters.

**The AV-OS checksum algorithms.** There are many ways to generate a checksum. The AV-OS code uses two separate techniques to compute a checksum. In the first, the checksum value is simply the arithmetic sum of the data being computed. As an example, if the vote counts were as follows:

Smith:	100
Jones:	32
Roberts:	7

then the checksum would be 139. If the value for any counter changes without the corresponding checksum value changing, it would be easy to notice the discrepancy and investigate what happened. However, using addition as a checksum, while simple to compute, fails to catch many classes of errors. For example, if the vote totals for Smith and Jones were switched, the checksum would still be 139. There are other classes of changes for which addition is not ideal and will not detect changes.

The AV-OS computes checksums over textual data in a slightly different, but related, manner. The checksum depends on the value of each of the names as well as their position (first, second, or so on).

---

**The AV-OS checksum does not detect malicious attacks.** An adversary with the ability to read and write to the memory card can always engineer the checksum to match what the malicious data they place. However, relying on the checksum to guarantee that data didn't change due to a malicious individual is not possible.

Using the addition operator (+) as a checksum may catch certain classes of non-malicious changes. However, an attacker can easily produce two different memory cards which have the same checksums. This means the checksum should not be used to determine malicious tampering. The textual checksum is also vulnerable to similar attacks.

If there was a desire to use checksums to detect malicious tampering with the contents of memory cards, a different checksum algorithm would be needed. One possibility would be to compute and print a cryptographic hash of the contents of the entire memory card at the beginning and end of the day, so that election officials can verify that the contents of the memory card had not been changed during transport. A cryptographic hash function is related to a checksum but instead of 65536 outputs, has over  $2^{160}$  possible values; furthermore, it is specially designed to protect against reordering and malicious tampering. Examples of cryptographic hash functions include SHA-1 or SHA-256. If this route were taken, the cryptographic hash function should be applied to the entire contents of the memory card, including all election parameters and the audit log. Another possibility would be to use cryptographic digital signatures, either a public-key signature as discussed later, or a symmetric-key MAC like the one used by the TSx (see below).

**The TSx “checksum”.** The AccuVote TSx operates differently. It reads the election parameters from a file on the memory card. There is a symmetric-key message authentication code (MAC) that protects the data from tampering. This computation depends on a secret key, and the MAC is designed so that anyone who does not know the key will not be able to tamper with the data without being detected. Thus, as long as the key is secret and unpredictable, it will detect malicious third party tampering, as well as problems with the storage media. A cryptographic MAC has all the advantages of a conventional checksum, in that it can detect accidental changes or corruption of the data, plus it can also detect malicious tampering as well. Thus, a cryptographic MAC is much better than a checksum in every way, and we expect the TSx to be extremely effective at detecting accidental data corruption.

See Finding 3 for a discussion of what data is protected by the cryptographic MAC on the TSx.

Since the TSx systems can read the AV-OS memory cards, they also include compatibility support for the data on those cards. Of course, those cards are only protected by the AV-OS checksums discussed earlier and are thus subject to the same caveats regarding tampering.

## 5 Mitigating the Risks

We next discuss several possible steps that could be taken to mitigate or ameliorate the risks discussed in this report. We start by discussing the full set of mitigations that might be possible in the long run; then, we discuss some short-term mitigation options.

### 5.1 Long-term Mitigation Strategies

**Mitigation 1** *Adopt procedures that eliminate the possibility of a single person tampering with the memory card at any time during the lifetime of a memory card.*

---

One approach to mitigating the risk of tampering with the memory cards is to adopt various standard handling procedures that prevent someone from tampering without the risk of detection. These procedural controls would need be maintained throughout the lifetime of the memory card. They would affect procedures for writing memory cards at county offices, for opening and closing the polls, and for transport and storage of memory cards. Training of precinct judges and precinct clerks would need to be augmented to stress the critical nature of these procedural controls. Among the possibilities are these:

- Adopt the principle that no one should ever alone with memory cards, i.e. there should always be two or more persons present (or none). This parallels the common requirement that no one should be alone with ballots (blank or voted).
- Use numbered, tamper-evident seals to protect memory cards when they are stored or when they are inserted in a voting machine. Keep records, and train poll workers to monitor those seals and their numbers and report anomalies. No one person should be entrusted with that task; all poll workers should sign off that the seals were intact.
- Permanently affix serial numbers to the memory cards and adopt written chain-of-custody procedures for transfer of custody from one pair of people to another, including poll workers.
- Train all personnel, including poll workers, that memory cards are ballot boxes and must be treated with the same degree of care and security.
- Whenever the procedures outlined are breached for some reason, take the memory card(s) in question out of service and zero them (in the presence of at least two people) before using them again.

It would help if memory cards were sealed inside the AV-OS at county headquarters, and AV-OS machines delivered to the polling place with the card already inserted and protected by tamper-evident seals. At the close of polls, it would help if poll workers did not break the seal, but rather returned the entire unit (with memory card still sealed inside) to county headquarters. This would reduce the opportunity for poll workers to tamper with memory cards.

When the AV-OS is used as a central-count machine (e.g., to count absentee votes), similar processes could be used to ensure that officials never insert a memory card into the AV-OS unless they are sure no one has had unsupervised access to the memory card. Because central-count machines reside in a controlled environment with physical security protections, and only a limited number of individuals have access to them, it should be much easier to apply very strong procedural controls to these machines.

**Mitigation 2** *Revise the source code of the AccuBasic interpreter to fix these vulnerabilities, introduce the use of defensive programming practices, and use security practices that will eliminate the possibility of any other vulnerabilities of the sort we discovered here.*

We can break this mitigation down into several (closely related) steps:

- Fix the AV-OS AccuBasic interpreter to eliminate the bugs we found. Every one of the bugs we found should be fixed. Any other bugs of the same sort should also be fixed.

---

It is not enough merely to introduce narrow changes to patch the specific bugs we found. Those bugs were symptoms of more fundamental flaws in the programming practices used to build the interpreter. The only way to be sure that all the bugs have been eliminated is to fix the root cause. We explain next what would be involved in doing so.

- Revise the interpreter source code, line by line, to eliminate all trust in the contents of the memory card. One of the reasons that these vulnerabilities existed was because the programmer implicitly assumed that the memory card would not be tampered with, and that the AccuBasic object code (.abo file) on the memory card was produced by a legitimate AccuBasic compiler. The source code should be changed to eliminate all instances of this kind of trust. For instance, when reading an integer from the memory card, the interpreter should first check that it is within the expected range. When reading a string from the memory card, the interpreter should not blindly assume that the string is '\0'-terminated, but should check that this is true before relying on it. Thus, this would involve identifying every point in the code that reads data from the memory card (or any other untrusted source) and inserting appropriate input validation checks at that point.

Likewise, every place where the code manipulates a vote counter, the code should check that the vote counter is (a) non-negative, and (b) arithmetic on it (e.g., incrementing a vote counter) does not wrap or overflow. If the code always checked that every vote counter were non-negative, and eliminated all possibility of arithmetic overflow or wrap-around modulo 65536, Hursti would not have been able to pre-load a negative number of votes for one candidate on the memory card. If the code had checked that all vote counters were zero at the start of the day, Hursti would not have been able to pre-load a positive number of votes for any candidate, either.

In addition, it would be prudent to revise the source code of the interpreter to prevent infinite loops and infinite recursion. One way to do this would be to introduce a timeout of some sort, and check for timeout every time the AccuBasic script executes any kind of backward jump, call, or control transfer.

- Revise the interpreter, line by line, to incorporate defensive programming throughout the code. If the code had been written to follow defensive programming practices in a more disciplined way, these vulnerabilities could not have existed.

Programming and driving a car are similar in that the programmer, like the driver, cannot control his or her environment; he or she can merely control how the software, or the car, reacts to that environment. Driving courses emphasize "defensive driving". Driving students learn to prepare for other drivers taking unexpected, and dangerous, actions. They understand that they cannot control other drivers, and that they must avoid accidents even if those accidents are not their fault.

Similarly, programmers should develop software with the understanding that the environment is not trusted. Users may enter incorrect input; system hardware may fail; touch screens may be miscalibrated and so return nonsensical values to the program. Good programming style is to build software that either functions correctly in the face of such errors, or else reports the error and terminates gracefully. This style of defensive programming is called "robust programming".

As an example, a buffer overflow occurs when an input is larger than the memory allocated to hold that input. The excess input can change internal values, causing the software to malfunction and return incorrect results. In some cases, this allows a malicious user to breach security. Robust programming requires that *every* input be checked; were this style followed, buffer overflows would not occur because the program would check the length of the input, determine it was too long, and reject it.

More generally, defensive programming generally means that every module should apply these checks to data it receives from other modules, and should refrain from trusting other modules. Just as drivers are taught that they cannot control what other drivers may do, defensive programming teaches that programmers cannot control what other modules may do, and so should treat them as untrusted and ensure that other modules cannot compromise their own integrity.

Thus, defensive programming often involves disciplined use of various idioms that ensure the safety of the code. Before copying a string into the buffer, one inserts code to check that there is sufficient room for the string. Before dereferencing a pointer, one writes code to check that the pointer is not NULL. Before adding two integers, one checks that the addition will not overflow. Code is added to perform these checks, even when they seem unnecessary, because sometimes one's assumption that the check is not necessary turns out to be inaccurate.

Our review of the interpreter source code showed that the programmers could have applied this principle of robust programming more extensively to the code. Specifically, the code had shortcomings (detailed above) that would not occur when software is designed and written to be robust. Hence, when the bugs in the AccuBasic interpreter are fixed, it seems prudent to also revise the code to be robust in the face of erroneous, unexpected, and malicious input, and other failures such as hardware failure.

- After the source code is revised, it would make sense to commission an independent source code review to confirm whether all of the vulnerabilities have been eliminated and to assess whether the code has used structured programming practices that are adequate to have confidence that no other security vulnerabilities of this sort are likely to be present.

If the source code is not revised, anyone with unsupervised access to a memory card, or with access to the GEMS server, may be able to exploit the vulnerabilities we found to take control of voting machines and compromise the electronic tallies. Such an attack might be able to cause lasting effects that persist across elections, and it is not clear whether there would be any way to repair the resulting damage. If the source code is revised to fix the vulnerabilities we found, these attacks would not be possible.

Even if the interpreter source code is fixed, it would still be possible for an individual who can introduce a malicious AccuBasic script to cause fraudulent zero tapes and fraudulent summary reports to be printed. Depending on whether the arithmetic overflows are fixed, such an individual might also be able to pre-load a memory card with a positive or negative number of votes for some candidates.

**Mitigation 3** *Protect AccuBasic object code from tampering and modification, either by (a) storing AccuBasic object code on non-removable storage and treating it like firmware, or by (b) protecting AccuBasic object code from modification through the use of strong cryptography (particularly public-key signatures).*

All of the vulnerabilities we uncovered were due to the fact that part of the code of the voting system (namely, the AccuBasic object code) was not adequately protected from modification. Thus, one effective mitigation would be to protect the code from modification, using one of two strategies:

- (a) Protect AccuBasic object code in the same way that the rest of the firmware object code is protected, by placing the AccuBasic object code on physically secured non-removable storage. Normally, firmware is protected from modification by storing it on a non-removable storage device (e.g., EEPROM) that is not easily externally accessible and that is protected from casual tampering through some kind of physical security protection. AccuBasic object code could be stored in the same way. If this were done, it would eliminate an entire attack vector, because attackers would no longer have the opportunity to replace the AccuBasic object code with a malicious AccuBasic script.

Of course, in this approach AccuBasic code would need to be protected with the same protections that are afforded to *firmware code*. If there is any way to update AccuBasic object code (or any other code), the update process must be strongly authenticated, and updates to the AccuBasic object code must be authenticated as securely as updates to the firmware. (By *authenticated*, we mean that there are procedural and technological controls which ensure that only authorized individuals can update the code, and only under appropriate circumstances.)

We recognize that different jurisdictions may require different AccuBasic scripts. One way to handle this would be for each jurisdiction to update the firmware with the appropriate AccuBasic script. Another possibility would be for the vendor to store all the different versions of AccuBasic object files that might ever be needed on the firmware, and for the memory card to contain an index (e.g., numbered from 1 to  $n$ , where  $n$  is the number of different AccuBasic scripts stored in the firmware) identifying which of these .abo files is to be used. Depending on the circumstances, this index might need to be protected from modification.

- (b) Alternatively: Use strong cryptography to protect the AccuBasic object code while it is stored on removable media. The appropriate protection would involve signing the AccuBasic object code with a cryptographically strong public-key signature scheme (e.g., RSA, DSA, or some other appropriate public-key algorithm) and arranging for the firmware to check the validity of this signature before executing the AccuBasic code. The private key would need to be guarded zealously (e.g., using a hardware security module (HSM)). In addition, considerable thought needs to be given to key management as well as to which part of the data is signed by which principals (e.g., by the vendor, by the GEMS server, or by other authorities).

While the AV-TSx cryptography is a good first step in this direction, it falls short in several respects:

- The use of symmetric-key cryptography in the AV-TSx increases the risk of key exposure. It would be safer to use public-key (*asymmetric*) digital signatures for this purpose.
- The use of hard-coded symmetric keys that are the same for all AV-TSx units is highly inappropriate for this purpose, and should be avoided at all costs.
- The existence of any kind of default key is a usability pitfall, because it makes it possible for election officials to forget to change the keys, thereby leaving them unaware of their vulnerability. This is an additional problem with hard-coded symmetric keys. We recommend that default keys be avoided.

- Insufficient thought has been given to the topic of key management and which entities are in possession of the appropriate cryptographic keys.

Fixing these shortcomings would prevent unauthorized individuals from introducing malicious AccuBasic scripts.

Of course, in both approaches the AccuBasic scripts need to be considered part of the codebase of the system, and should be reviewed as part of the qualification and certification process.

In the long run, the consequences of not protecting AccuBasic code from modification are that anyone who gains unsupervised access to memory cards can tamper with their contents, attack the voting systems (e.g., using Hursti-style attacks), and potentially manipulate the electronic vote tallies.

**Mitigation 4** *Change the architecture of the AV-OS and the AV-TSx so they do not store code on removable memory cards.*

In the long run there are good reasons for changing the AV-OS and AV-TSx software architectures so that they do not rely on interpreted code stored on a removable memory card, or that they do not use interpreted code at all and eliminate AccuBasic. All of the potential vulnerabilities discussed here are rooted in the fact the code is stored on the removable memory cards, and these cards are handled by, and in the custody of, many people in a major election. There does not seem to be any *fundamental* reason why the AccuBasic code cannot be part of the firmware codebase, rather than stored on the removable memory card. That change would not only eliminate these attacks, but some GEMS-based attacks on the code as well. Of course there would need to be enough firmware storage space in the machines to hold the AccuBasic code, but we suspect that is not an insoluble problem. This change would reduce the vendor's flexibility in providing different reporting options to different jurisdictions (i.e. different AccuBasic scripts). But if it is accepted that the AccuBasic scripts are part of the voting system "code", as they are, and that therefore they must be subject to testing and code review by federal and state examiners, then that flexibility would be lost anyway, since it cannot be expected that the examiners would be able to study hundreds of variations on the AccuBasic script packages produced for different jurisdictions.

**Mitigation 5** *Change the architecture of the AV-OS and the AV-TSx so they do not contain any interpreter or use any kind of interpreted code.*

There are also good arguments for eliminating AccuBasic interpreted code entirely from voting system software. The FEC 2002 Voluntary Voting System Standards expressly forbid interpreted code in section 4.2.2. Perhaps the standard writers had in mind forbidding only powerful, interpreted *programming* languages, such as Visual Basic, and not relatively benign and limited *rendering* languages such as HTML. AccuBasic falls somewhere in the middle on the more benign side (assuming the interpreter bugs are fixed). But the text of the standard is pretty clear, and the same language from the 2002 standards has been preserved in the EAC's new successor standard, the Voluntary Voting Systems Guidelines, as section 5.2.2. To be in compliance it would seem that AccuBasic would have to be eliminated, or the standard would have to be changed.

In any case, the inclusion of interpreted languages in a voting system causes great burdens on examiners and code reviewers, who have to be highly skilled and do considerable analysis of the compiler and interpreter in order to verify that it does not present security vulnerabilities or permit

---

malicious code to go unnoticed. It seems untenable to us that every time there is a change to the AccuBasic language or interpreter another round of detailed code review such as we have done would be required; however, an interpreter is such a delicate and powerful feature (from a security point of view) that we cannot recommend shortcuts in its examination either.

## 5.2 Short-term Mitigation Strategies for Local Elections

One disadvantage of several of these mitigation strategies (e.g., revising or eliminating the AccuBasic interpreter, improving the cryptography, etc.) is that changes to the source code will incur significant delays. Source code changes would need to be approved by the federal qualification process as well as the state certification process. Therefore, in the short term it seems appropriate to consider mitigation strategies that do not involve changing the source code.

For local elections (i.e., elections that do not span the entire state), we believe there are mitigation strategies that could be viable for the short term. For instance, one possibility might be the following two-prong approach:

- For the AV-TSx, update the cryptographic keys on every AV-TSx machine and rely on the cryptography to prevent tampering with memory cards. Election officials would need to first choose a secret and unguessable cryptographic key. The new cryptographic key *should* be chosen at random by county staff, should not be divulged to anyone, not even the vendor (because anyone who learns the secret key gains the ability to tamper undetectably with memory cards), should not be shared across counties, and should be tightly controlled. Then, the process of updating the keys requires inserting a smartcard into every AV-TSx machine. Officials could adopt checklists or some other process to ensure that every AV-TSx machine has had its keys updated before it is sent into the field. Election officials should be warned that if they forget to change the cryptographic keys, the machine will outwardly appear to function correctly, but will be vulnerable to attack.
- For the AV-OS, deploy strict procedural safeguards to prevent anyone from gaining unsupervised access to a memory card. We would suggest dual-person controls over the entire lifecycle of the memory card, chain of custody provisions, and use of numbered tamper-evident seals. It would also help to load and seal the memory card into the AV-OS unit at the warehouse in advance of the election, ship it in this state, and when the election is over, have poll workers return the entire machine (with the memory card still sealed inside) to the county collection point, where election officials would check that the seal remains undisturbed and record the number on the seal before removing the memory card. This would ensure that the memory card is protected by a tamper-evident seal for the entire time that it is outside the control of county staff and would reduce the opportunities for someone to tamper with the memory card while it is in transit. We recognize that these heightened procedural protections are likely to be somewhat burdensome, but as a short-term protection (until the source code can be fixed), they may be appropriate. See Mitigation 1 for further discussion of procedural mitigations.

While these strategies do not completely eliminate all risk, we expect they would be capable of reducing the risk to a level that is manageable for local elections in the short term.

In the longer term, or for statewide elections, the risks of not fixing the vulnerabilities in the AccuBasic interpreter become more pronounced. Larger elections, such as a statewide election, provide a greater incentive to hack the election and heighten the stakes. Also, the longer these

---

vulnerabilities are left unfixed, the more opportunity it gives potential attackers to learn how to exploit these vulnerabilities. For statewide elections, or looking farther into the future, it would be far preferable to fix the vulnerabilities discussed in this report.

## 6 Conclusions

We have detailed a number of security vulnerabilities in the AV-OS and AV-TSx implementations of the AccuBasic interpreter. In the long term, these vulnerabilities can be easily fixed and the risks eliminated or mitigated. We have made recommendations about several ways in which that might be accomplished. In the short term, we believe the risks can be mitigated through appropriate use procedures.

## 7 Glossary

**.abo file** a file containing AccuBasic object code (byte code)

**AccuBasic** a Diebold-proprietary programming language used (in slightly different versions) in both the AV-OS and AV-TSx machines; AccuBasic programs allow very limited control over the behavior of the voting system

**buffer** a fixed-size area of memory

**buffer overrun** a type of program bug in which the program attempts to write more data into a buffer than the buffers size permits. The extra data is thus written beyond the end of the buffer into other memory, where it often overwrites something else of significance, i.e. either other data, or control information, or even instructions. When that happens, the program is corrupted, and any of a vast number of unpredictable things might ensue. One common hacker attack is to deliberately take advantage of a buffer overrun bug, corrupting the program in a specific way that allows the hacker to do things he otherwise would not be able to do. (Usually the goal is to take complete control of the machine.)

**byte code** object code of a relatively simple kind (e.g., that happens to be encoded as characters (bytes) instead of binary data)

**C** a very widely used programming language

**C++** another widely used programming language, more modern than C, and (roughly) including C as a subset

**compiler** a program that translates another program from its source language (the human readable form) into an object language (a form not so easily human readable, but much more convenient for machine execution). The AccuBasic compiler translates AccuBasic programs (source code) into AccuBasic object code (also known as byte code in this case).

**file system** hierarchical collection of files and directories (folders), along with their names, types, and the software to read and write them

---

**firmware** software resident inside the voting machine (i.e. not on a removable memory card) and that is (or should be) unmodifiable once the machine is in operation

**hex editor** an editor that can modify data directly at the binary level. (Hex refers to hexadecimal (base-16) arithmetic, which is extremely closely related to binary, but more compact.) A hex editor is a universal editor, in that it can edit absolutely any kind of digital data, although it requires some knowledge and skill to use it in any particular case.

**interpreter** a program whose function is to execute another program, usually one that is in the form of object code. The AccuBasic interpreter is part of the firmware of the AV-OS or AV-TSx, and executes AccuBasic object code, i.e. .abo files.


**memory mapped** memory mapped data is data that resides on some attached memory device, and yet is made to appear as if it is in main memory. (In the technical jargon, the data on the attached device is mapped onto a portion of the machines memory address space.)

**object code** a program represented in the form of discrete instructions that are easy for a computer (or an interpreter) to execute efficiently. It is more difficult for humans to read and write object code than source code, but it can be done with only modest skill. Usually object code is produced with the aid of a compiler, but it does not have to be.

**scripting language** a programming language designed primarily so that the programs written in it can easily manipulate character data and files (as opposed to, e.g. binary data), and can easily invoke and control other programs; AccuBasic can be described as a limited-purpose scripting language.

**scripts** programs written in a scripting language like AccuBasic

**source code** any software in the original form as written by a human programmer; this is the form in which code is easily read and written by programmers, but cannot be directly executed by a computer or an interpreter



**Diebold Election Systems, Inc.  
Source Code Review  
and Functional Testing**

February 23, 2006

**Prepared For:**

Diebold Election Systems, Inc.  
1253 Allen Station Parkway  
Allen, Texas 75002  
(469) 675-8990 (office)  
(214) 383-1596 (fax)

**ciber**  
— ALWAYS ABLE —

**Prepared By:**  
CIBER Huntsville and  
CIBER's Global Security Practice

CIBER, Inc.  
7501 South Memorial Pkwy  
Suite 107  
Huntsville, AL 35802  
256.882.6900

## Table of Contents

REVISION HISTORY.....	3
INTRODUCTION.....	4
BACKGROUND.....	4
<i>Diebold</i> .....	4
<i>CIBER's Global Security Practice</i> .....	4
OVERVIEW AND APPROACH.....	4
REFERENCE MATERIALS.....	5
EXECUTIVE SUMMARY.....	5
INTERPRETER ASSESSMENT.....	6
OVERVIEW.....	6
GENERAL INTERPRETER REQUIREMENTS.....	7
<i>Report Writing Functionality</i> .....	7
<i>Special Purpose Functionality</i> .....	7
<i>Program Behavior</i> .....	7
SPECIFIC INTERPRETER REQUIREMENTS.....	8
<i>Report Printing</i> .....	8
<i>Bounded Stack and Memory</i> .....	8
<i>Operating System Services</i> .....	9
<i>Method Invocation and Data Structures</i> .....	9
<i>Exception Handling</i> .....	10
<i>Script Operations</i> .....	10
<i>Halting</i> .....	10
<i>Infinite Loops</i> .....	11
INTERPRETER – OTHER FINDINGS.....	11
COMPILER ASSESSMENT.....	11
OVERVIEW.....	11
SPECIFIC COMPILER REQUIREMENTS.....	12
<i>Commands, Operators, Functions, and Files</i> .....	12
<i>Additional Code</i> .....	12
<i>Compilation Environment</i> .....	<b>Error! Bookmark not defined.</b>
<i>Witness of Build</i> .....	<b>Error! Bookmark not defined.</b>
COMPILER – OTHER FINDINGS.....	12
ACCUBASIC SCRIPTS ASSESSMENT.....	13
OVERVIEW.....	13
SPECIFIC SCRIPT FILE REQUIREMENTS.....	13
<i>Infinite Loops</i> .....	13
<i>Binary Code</i> .....	13
SCRIPT FILES – OTHER FINDINGS.....	13

### USE AND DISCLOSURE OF INFORMATION IN THIS DOCUMENT

This report contains information that is proprietary to Diebold Election Systems, Inc. This document and all the information contained within it and its attachment(s) may be used by the intended recipient only. All information remains the property of CIBER and Diebold Election Systems, Inc. and any other use or disclosure of this information requires prior written approval.

**REVISION HISTORY**

Date	Version	Description	Author
02/23/2006	V1.0	Document Creation	CIBER Huntsville and CIBER's Global Security Practice

**USE AND DISCLOSURE OF INFORMATION IN THIS DOCUMENT**

This report contains information that is proprietary to Diebold Election Systems, Inc. This document and all the information contained within it and its attachment(s) may be used by the intended recipient only. All information remains the property of CIBER and Diebold Election Systems, Inc. and any other use or disclosure of this information requires prior written approval.

© 2006, CIBER, Inc.

## INTRODUCTION

### BACKGROUND

#### Diebold

Diebold Election Systems, Inc. retained the services of CIBER Huntsville and CIBER's Global Security Practice professionals in order to determine the degree to which the Diebold Election Systems compiler, interpreter, and script source code and functionality comply with security industry and coding best practices and are not vulnerable to compromise per the following order:

"The Diebold AccuVote-OS (AV-OS) optical scan and AccuVote-TSX (TSX) touchscreen voting systems contain a report writing facility in which scripts, written in a proprietary high-level AccuBasic programming language, are first compiled into a low-level tokenized language, and then the tokenized code is interpreted using an interpreter module in the firmware of the AV-OS or TSX unit. The safety of the interpreted code is a vital security issue, so it is important to verify that it is not possible to compromise an election in any way through the (mis)use of AccuBasic, *including an unintentional error or malicious AccuBasic script.*" [Reference: Request for ITA Review of Diebold AccuBasic Interpreter, Compiler, and Scripts.pdf]

#### CIBER's Global Security Practice

CIBER® (NYSE: CBR) is a leading international system integration consultancy with superior value-priced services for both private and government sector clients. CIBER's services are offered on a project or strategic staffing basis, in both custom and enterprise resource planning (ERP) package environments, and across all technology platforms, operating systems and infrastructures.

CIBER's Global Security Practice, focuses exclusively on information security. Its professional staff designs, implements, and manages security solutions for critical information systems in a wide range of commercial and Federal environments. Since 1992, organizations desiring superior security engineering and consulting services have turned to CIBER Security to fulfill their information security needs.

### OVERVIEW AND APPROACH

The CIBER Huntsville and CIBER Global Security teams were tasked with performing a combination of testing and analysis of the Diebold Election System's Source Code to identify security and functionality vulnerabilities. The testing was structured to identify and evaluate as much potential vulnerability as possible within a reasonable/controlled level of effort.

#### USE AND DISCLOSURE OF INFORMATION IN THIS DOCUMENT

This report contains information that is proprietary to Diebold Election Systems, Inc. This document and all the information contained within it and its attachment(s) may be used by the intended recipient only. All information remains the property of CIBER and Diebold Election Systems, Inc. and any other use or disclosure of this information requires prior written approval.

© 2006, CIBER, Inc.

The engagement was comprised of four phases that were collectively designed to provide a comprehensive inspection of the source code using primary and secondary development requirements as the evaluation metric. The phases were:

**Phase 1:** Portions of the code specific to the interpreter were assessed. Manual reviews broke up interpreter code into that which is specific to the AV-OS, and that which is specific to the TSX. For each interpreter code set, nine (9) primary requirements, three (3) of which contained a total of 12 sub-requirements, were reviewed.

**Phase 2:** Portions of the code specific to the compiler were reviewed. Four (4) primary requirements were reviewed for this purpose.

**Phase 3:** CIBER's team conducted manual testing of the AccuBasic scripts, for which two (2) primary requirements were relevant.

**Phase 4:** Code was reviewed for other forms of potential risks, hazards, or vulnerabilities *not* relative to any of the specified primary or secondary requirements.

## REFERENCE MATERIALS

The following reference materials were used for the source code review:

1. ABasic 2-1-9 to 2-1-9-1 Change Release Summary
2. ACCU-BASIC User Guide Release 1.92
3. TSX AccuBasic Interpreter Source Code
4. AV-OS AccuBasic Interpreter Source Code
5. AccuBasic Compiler
6. Provided Scripts written in AccuBasic

## EXECUTIVE SUMMARY

The TSX interpreter inspected appears to be ready for an election. The AV-OS interpreter inspected appears to be sufficiently secure to run an election if the recommended corrective measures are applied to the interpreter and rechecked. If trusted chain-of-custody were established to prevent tampering with memory cards between the GEMS system and the AV-OS voting machines, then the existing units would be safe for an election.

The fact that the programs appear to provide adequate security shall not be interpreted to mean that the programs are without security vulnerabilities or are impenetrable. It does mean that the programs appear to provide reasonable assurance that it can protect the confidentiality, integrity, and availability of the information it processes, stores, and communicates.

It is standard practice at CIBER to provide recommendations in addition to review findings. In addition to the recommendations that will be placed throughout this report, one high-level recommendation is provided:

### USE AND DISCLOSURE OF INFORMATION IN THIS DOCUMENT

This report contains information that is proprietary to Diebold Election Systems, Inc. This document and all the information contained within it and its attachment(s) may be used by the intended recipient only. All information remains the property of CIBER and Diebold Election Systems, Inc. and any other use or disclosure of this information requires prior written approval.

- Certain vulnerabilities in this report *may* require a portion of the code to be modified in order to correct the vulnerabilities identified. To ensure that the efforts to correct vulnerabilities do not introduce new vulnerabilities, CIBER strongly recommends retesting of the remediated code prior to its migration to a production environment.

The interpreter had three security vulnerabilities and a small number of requirement violations that were not capable of being exploited by malicious code or operators. Of the three serious problems, they can be fixed with minor code changes.

No issues were discovered with the compiler that impacts the security of the system. There were no findings in the inspection of the AccuBasic Scripts that would materially impact the security of the system.

## INTERPRETER ASSESSMENT

### OVERVIEW

On December 28<sup>th</sup>, Diebold Elections Systems, Inc. requested that CIBER perform a source code review of Diebold's AccuBasic Interpreter. From the statement of work, the purpose of this task was to identify the validity of the following statement:

*"The safety of the interpreted code is a vital security issue, so it is important to verify that it is not possible to compromise an election in any way through the (mis)use of ABasic, including an unintentional error or malicious ABasic script."*

Nine primary and twelve secondary requirements were identified as essential. Along with the presented requirements, CIBER was encouraged to provide additional, experience-related concerns or comments as appropriate.

To minimize assumptions made in the code review, the review was performed solely on the provided materials and their ability to handle both normal and malicious use, even if the malicious behavior came from any point outside of the code base.

The AccuVote-OS Interpreter was provided in the form of two files. The TSX Interpreter was provided in the form of three files. Microsoft Visual Studio 2003 was used to examine the software but was used only for text viewing and for reference searching. CIBER inspected the source code on a function-by-function basis, as well as used Internet references as needed for validation of the functionality of commands from external libraries.

The Interpreter source code that was provided was not a stand-alone application and the programs that call the interpreter were not present. The calling programs were deemed "not trusted" and capable of providing invalid input in the course of this review. A series of requirements were given that were used to evaluate the security and functioning of the code.

#### USE AND DISCLOSURE OF INFORMATION IN THIS DOCUMENT

This report contains information that is proprietary to Diebold Election Systems, Inc. This document and all the information contained within it and its attachment(s) may be used by the intended recipient only. All information remains the property of CIBER and Diebold Election Systems, Inc. and any other use or disclosure of this information requires prior written approval.

## GENERAL INTERPRETER REQUIREMENTS

General requirements for review of the interpreter necessitate that the interpreter address those properties defined in the following sub-sections.

### Report Writing Functionality

**Requirement:** The language and interpreter are used only for the report-writing functions of the AV-OS and TSX.

**AV-OS Finding:** No violations were found. A total of six locations were found in the AV-OS source code where the interpreter is called, all for the purpose of printing a report.

**TSX Finding:** No violations were found. One location was found in the TSX source code for calling the interpreter that was written for the sole purpose of printing reports.

### Special Purpose Functionality

**Requirement:** The language and interpreter do not constitute a general purpose embedded programming system, but are instead extremely limited and special-purpose in their capability and linkage to the rest of the firmware environment, and possess essentially no more capability than is required for their report-writing function.

**AV-OS and TSX Finding:** No violations were found. The compiler is separate from the interpreters and not included on the Diebold voting units. The interpreters have extremely limited functionality for the creation of a report and the interpreter cannot change its own parameters of execution. The Interpreter cannot create or write to any files and only sends information to the printer and the display.

### Program Behavior

**Requirement:** No program in the compiled, tokenized form (whether or not generated by an AccuBasic compiler) can ever, directly or indirectly,

- a. Modify any votes or vote counters,
- b. Modify the electronic audit trail, or
- c. Cause any other behavior that might compromise the integrity of an election, even if the code is maliciously engineered, and even if election procedures have not been properly followed.

**AV-OS and TSX Finding:** Three violations exist that allow manipulation and reading of data in global space. Three different types of modified tokens used to index data outside of their intended memory range cause the vulnerabilities, each with slightly different effects. These can only be exploited by a modified AccuBasic object file.

---

#### USE AND DISCLOSURE OF INFORMATION IN THIS DOCUMENT

This report contains information that is proprietary to Diebold Election Systems, Inc. This document and all the information contained within it and its attachment(s) may be used by the intended recipient only. All information remains the property of CIBER and Diebold Election Systems, Inc. and any other use or disclosure of this information requires prior written approval.

It is quite possible that these exploits can be used in conjunction with each other in a way to produce an escalation of privileges, depending on the operating environment and the compiler settings. The evaluation team confirmed the flaws are present and considered dangerous, but proof-positive exploit for an escalation was not possible without access to a working development environment and appropriate development software.

The TSX environment contains a check to validate the AccuBasic object files, so if a file is tampered, the tampering will be detected. Therefore, this problem is more severe for AV-OS than it is for TSX. TSX can still be considered election ready because such tampering will be detected.

## **SPECIFIC INTERPRETER REQUIREMENTS**

Specific requirements for review of each of the two interpreters necessitate that each interpreter address those properties defined in the following sub-sections.

### **Report Printing**

**Requirement:** Verify the interpreter is not called or executed when printing a report or the sequence of question and response selecting the options and sequence of the report(s).

**AV-OS and TSX Finding:** No violations discovered. The interpreter itself does not have an ability to detect multiple instances of itself being run, however the rest of the source code indicates that the voting software does not support multi-threading per code instance, and only one instance of a running program is allowed at a time.

### **Bounded Stack and Memory**

**Requirement:** Verify that it is not possible for the interpreter to write to main memory outside the bounds of its own stack and memory. As part of this, verify that

- a. The interpreter's stack (and heap if present) are bounded, so that they cannot overflow their fixed area without triggering an exception;
- b. The AccuBasic language itself does not provide for dynamic storage allocation, nor any use of pointer-like variables or variable offsets and hence such language constructs cannot be misused to write main memory arbitrarily;
- c. The only indexed data structures supported in the AccuBasic language are strings (not arrays), and they can only be *read* through the index, but not *written*, so that in particular it is not possible to overwrite main storage arbitrarily using an out of bounds index in a string assignment;
- d. String and substring copy operations are protected in the interpreter so no buffer overflows can occur, as a result of copying a big string into a small buffer.

#### **USE AND DISCLOSURE OF INFORMATION IN THIS DOCUMENT**

This report contains information that is proprietary to Diebold Election Systems, Inc. This document and all the information contained within it and its attachment(s) may be used by the intended recipient only. All information remains the property of CIBER and Diebold Election Systems, Inc. and any other use or disclosure of this information requires prior written approval.

**TSX Finding:** The bounds checking on the stack and heap segments were not detected, although it may have been outside the scope of the assessment area for this team. Bounds checking is performed inside the code and can be used to prevent overflows. Also, a “stack guard” (or non-executable stack) software package may help lower the risk. The lack of non-executable stack software does not imply that a malicious overflow does exist, nor does the presence of one guarantee that an overflow attack cannot still work.

**Both:** A previously mentioned violation for “Program Behavior” demonstrates ability for a maliciously constructed file to access global memory.

A minor violation of the requirements exists in that arrays do exist in AccuBasic, but can only be seen and used through an external data record, and have bounds checking. This data cannot be written to. Therefore, the arrays are not exploitable for out-of-bounds memory attacks. Bounds-checking does exist at the I/O perimeters to prevent buffer overflows as per the requirements. All memory used by the interpreter is fixed in length, cannot be dynamically allocated, and is bounds-checked to prevent access to main memory.

### Operating System Services

**Requirement:** Verify that the interpreter is extremely limited in the operating system services it requests (I.e., to the small number needed) to

- a. Request a yes/no input from a user,
- b. Write a report to the printer,
- c. Print message to the LCD screen (or touchscreen)
- d. Append audit trail entries, and
- e. Retrieve date/time.

**AV-OS and TSX Finding:** The I/O is limited to the requirements as stated and has no violation of this requirement.

### Method Invocation and Data Structures

**Requirement:** Verify that the interpreter does not invoke other methods or reference other data structures in the firmware codebase (outside the interpreter) except as required to perform its report-writing function, and that all such references are read-only. In particular, verify that the interpreter cannot

- a. Write any variables outside its own memory,
- b. Modify files on any file system (in the TSX),
- c. Launch any application programs.

**AV-OS and TSX Finding:** No violations were found of this requirement. Both versions of the interpreter do not support the ability to write to external variables

---

#### USE AND DISCLOSURE OF INFORMATION IN THIS DOCUMENT

This report contains information that is proprietary to Diebold Election Systems, Inc. This document and all the information contained within it and its attachment(s) may be used by the intended recipient only. All information remains the property of CIBER and Diebold Election Systems, Inc. and any other use or disclosure of this information requires prior written approval.

© 2006, CIBER, Inc.

or files and have no provision or function that allows for the launching of any application programs.

### Exception Handling

**Requirement 1:** Verify that the interpreter properly fields and properly handles all possible exceptions that can be generated in the AccuBasic language, especially string bounds errors, zerodivides, integer overflows, stack overflows and underflows, etc.

**AV-OS and TSX Finding:** No violations discovered. Appropriate error checking was made to catch and handle all forms of standard interpreter errors that come from the provided scripts.

**Requirement 2:** Verify that any exceptions generated by the interpreter's own execution are all properly caught and handled as well, including such exceptions as heap allocation failures, end-of-file, file access errors, I/O errors, volume overflow (for the volume containing the audit trail), etc.

**Both:** No violations were discovered. The critical points of failure, such as the reading of the object files and opening I/O streams to the printer, are checked for errors. Writing typically isn't checked for failure but, because they are open-ended streams and buffered, an error won't trigger even if the code checks for them.

### Script Operations

**Requirement:** Verify that the script operations terminate. That is, that all explicit and implicit looping or recursive operations have a definitive exit and the exit condition which will be achieved in a limited number of passes and that any exception handling or conditions result in an exit and return to the system operations and are not hung up in a wait state unless the wait is conditioned by an alert and visible response request to the control screen.

**AV-OS and TSX Finding:** No violations were discovered. A definitive exit condition and exit exists for all functions that exist in both versions of the interpreter as well as for all functions called by the interpreters.

### Halting

**Requirement:** Verify that the interpreter does not halt except at the appropriate time.

**AV-OS and TSX Finding:** No violations of this requirement were discovered. All detected failures will result in error codes being propagated to the calling point of the interpreter and passed to the calling program. There were no clear bugs discovered in the code where the system would suddenly halt based on normal usage.

---

#### USE AND DISCLOSURE OF INFORMATION IN THIS DOCUMENT

This report contains information that is proprietary to Diebold Election Systems, Inc. This document and all the information contained within it and its attachment(s) may be used by the intended recipient only. All information remains the property of CIBER and Diebold Election Systems, Inc. and any other use or disclosure of this information requires prior written approval.

© 2006, CIBER, Inc.

## Infinite Loops

**Requirement:** Verify that the interpreter does not infinitely loop (unless interpreting an AccuBasic script that contains an infinite loop at that level).

**AV-OS and TSX Finding:** No violations of this requirement were found, all loops in both interpreters have a clearly defined beginning and end point with an appropriate index that increments so that an end-point is always reached.

## INTERPRETER – OTHER FINDINGS

**Requirement:** While the above requirements are intended to give explicit guidance for the conduct of the review, the ITA (I.e., CIBER) is also expected to apply due diligence in reporting other potential risks, hazards, or vulnerabilities in the review.

**Both:** Error handling appears to be adequate for a system that executes in a perfect running environment. However, the interpreters do not have the proper degree of error checking to identify or recover from key failures in a damaged, altered or dysfunctional environment.

Often it is difficult to tell the difference between a ‘hack’ and a system failure for an operator. Programs that have high-assurance capabilities will provide additional information about sources of errors that can be useful to avoid negative speculation and recover faster in the advent of a catastrophic failure.

Our reasoning for increasing the security on the code is because the object code traverses potentially untrustworthy hands in the process of its distribution of scripts from the GEMS to the interpreter. Since the object code is on the memory cards being distributed, it is a prime target for potential tampering.

**AV-OS Finding:** The error codes returned by the interpreter to the AV-OS system are ignored. Although this isn’t a security violation, it would assist as being a validation of the procedures for if a problem does occur.

## COMPILER ASSESSMENT

### OVERVIEW

Four source code files and a Microsoft Visual C++ 6.0 Project file were submitted for code review of the AccuBasic Compiler. Microsoft Visual Studio 2005 was used to perform this source code review. The AccuBasic Compiler was compiled and built in a Visual C++ environment.

*Note: The User Guide is Release 1.92, and was revised on August 31, 1994. However, the source codes currently in review are Version 1.95.2, and are dated as 2005/12/18.*

#### USE AND DISCLOSURE OF INFORMATION IN THIS DOCUMENT

This report contains information that is proprietary to Diebold Election Systems, Inc. This document and all the information contained within it and its attachment(s) may be used by the intended recipient only. All information remains the property of CIBER and Diebold Election Systems, Inc. and any other use or disclosure of this information requires prior written approval.

## SPECIFIC COMPILER REQUIREMENTS

Requirements for review of the compiler necessitate that the compiler address those properties defined in the following sub-sections.

### Commands, Operators, Functions, and Files

**Requirement:** Verify that the compiler does not support additional AccuBasic commands and access methods not documented in the AccuBasic programming language manual.

**Finding:** No security violations were discovered. There were some inconsistencies between the manual and the provided functionality, although the differences are not malignant. Our recommendation is to update the manual to reflect the improvements.

### Additional Code

**Requirement:** Verify that the compiler does not interject additional code beyond what is specified in the AccuBasic report file source code scripts.

**Finding:** No security violations were discovered. The AccuBasic Compiler parses source code scripts. During the compilation, the compiled information was stored in the internal memory buffer. At the end of compilation, the Compiler writes the internal buffer into the output file to generate the object file.

There are 79 instances in which the Compiler writes the internal memory buffer. Besides the code (commands, functions, files, and etc.) specified in source code scripts, the Compiler also writes the series number into an object file. The Compiler also inserts internal flags into an Object file.

This extra information is necessary to maintain the integrity of the object files. Since they are very short strings and single-character flags, it is very unlikely that this kind of information will be interpreted as meaningful binary code when loaded by the firmware.

## COMPILER - OTHER FINDINGS

**Requirement:** While the above requirements are intended to give explicit guidance for the conduct of the review, the ITA (I.e., CIBER) is also expected to apply due diligence in reporting other potential risks, hazards, or vulnerabilities in the review.

**Findings:** No security violations were discovered. The compiler is largely immune to security issues because it is a stand-alone system and never leaves the development environment. Undocumented features discovered were all non-malicious and did not extend the capabilities of AccuBasic in any way.

---

#### USE AND DISCLOSURE OF INFORMATION IN THIS DOCUMENT

This report contains information that is proprietary to Diebold Election Systems, Inc. This document and all the information contained within it and its attachment(s) may be used by the intended recipient only. All information remains the property of CIBER and Diebold Election Systems, Inc. and any other use or disclosure of this information requires prior written approval.

## ACCUBASIC SCRIPTS ASSESSMENT

### OVERVIEW

There were 19 AccuBasic script files subjected to review. Microsoft Visual Studio 2005 was used to perform this source code review, but only served as a text file reader due to the nature of the files.

### SPECIFIC SCRIPT FILE REQUIREMENTS

Requirements for review of the scripts necessitate that the scripts address those properties defined in the following sub-sections.

#### Infinite Loops

**Requirement:** Verify, to the extent possible, that none of the scripts will infinitely loop under any conditions.

**Finding:** No infinite loops are contained within any of the scripts when the environment is running as designed.

#### Binary Code

**Requirement:** Verify that none of the scripts contain character string constants or other data that the compiler would include in the AccuBasic object files, but that might be interpretable as binary code when loaded by the firmware.

**Finding:** While certain values may be defined as binary, they are very short strings and single-character flags that are very unlikely to be interpreted by the firmware as meaningful binary code.

### SCRIPT FILES - OTHER FINDINGS

**Requirement:** While the above requirements are intended to give explicit guidance for the conduct of the review, the ITA (I.e., CIBER) is also expected to apply due diligence in reporting other potential risks, hazards, or vulnerabilities in the review.

**Finding:** No additional security problems were discovered. However, inconsistencies were found in coding style and documentation. These include default values that were not found in the user's guide, default values defined in the source code and malign undocumented functions. We recommend updating the documentation to keep configuration control.

#### USE AND DISCLOSURE OF INFORMATION IN THIS DOCUMENT

This report contains information that is proprietary to Diebold Election Systems, Inc. This document and all the information contained within it and its attachment(s) may be used by the intended recipient only. All information remains the property of CIBER and Diebold Election Systems, Inc. and any other use or disclosure of this information requires prior written approval.

© 2006, CIBER, Inc.

The New York Times

Archive

NYTimes Go to a Section

Welcome, [susanpynchon](#) - [Member Center](#) - [Log](#)

SEARCH

NYT Since 1981

Search

TimesSelect FREE 14-DAY TRIAL!

---

Tip for TimesSelect subscribers: Want to easily save this page? Use Times File by simply clicking on the Save Article icon in the Article Tools box below.

---

## NATIONAL DESK

## New Fears of Security Risks In Electronic Voting Systems

By **MONICA DAVEY**; **GRETCHEN RUETHLING** CONTRIBUTED REPORTING FROM CHICAGO FOR THIS ARTICLE, AND **JOHN SCHWARTZ** FROM NEW YORK. (NYT)

788 words

Published: May 12, 2006

CHICAGO, May 11 - With primary election dates fast approaching in many states, officials in Pennsylvania and California issued urgent directives in recent days about a potential security risk in their Diebold Election Systems touch-screen voting machines, while other states with similar equipment hurried to assess the seriousness of the problem.

"It's the most severe security flaw ever discovered in a voting system," said Michael I. Shamos, a professor of computer science at Carnegie Mellon University who is an examiner of electronic voting systems for Pennsylvania, where the primary is to take place on Tuesday.

Officials from Diebold and from elections' offices in numerous states minimized the significance of the risk and emphasized that there were no signs that any touch-screen machines had been tampered with. But computer scientists said the problem might allow someone to tamper with a machine's software, some saying they preferred not to discuss the flaw at all for fear of offering a roadmap to a hacker.

"This is the barn door being wide open, while people were arguing over the lock on the front door," said Douglas W. Jones, a professor of computer science at the University of Iowa, a state where the primary is June 6.

The latest concern about the touch-screen machines was only the newest chapter in an

emerging political and legal fight around the country over voting machines. While some voting officials defend the ease of touch-screens (similar to A.T.M.'s), some advocacy groups argue that optical scan machines, using paper ballots, are far more secure.

The wave of high-tech voting machines was prompted by the 2000 election in Florida, which spotlighted the problems of old-fashioned punch card ballots. But the machines that soon followed have spurred division. Here in Chicago, where voters used both touch-screen and optical-scan systems in a March primary, it took officials a week to tally all the votes because of technical problems and human errors, touching off a flurry of criticism over the Sequoia Voting Systems equipment.

In Maryland this spring, the State House of Delegates passed a bill that would have scrapped touch-screen machines, but the Senate last month took no action on the bill, effectively killing the idea.

This week, Voter Action, a nonprofit group, assisted voters in Arizona in filing for a legal injunction to try to block the state from buying touch-screen electronic voting systems. The suit is among several the group says it has pursued, in states including California, New York and New Mexico.

The new concerns about Diebold's equipment were discovered by Harri Hursti, a Finnish computer expert who was working at the request of Black Box Voting Inc., a nonprofit group that has been critical of electronic voting in the past. The group issued a report on the findings on Thursday.

Computer scientists who have studied the vulnerability say the flaw might allow someone with brief access to a voting machine and with knowledge of computer code to tamper with the machine's software, and even, potentially, to spread malicious code to other parts of the voting system.

As word of Mr. Hursti's findings spread, Diebold issued a warning to recipients of thousands of its machines, saying that it had found a "theoretical security vulnerability" that "could potentially allow unauthorized software to be loaded onto the system."

The company's letter went on: "The probability for exploiting this vulnerability to install unauthorized software that could affect an election is considered low."

David Bear, a spokesman for Diebold Election Systems, said the potential risk existed

because the company's technicians had intentionally built the machines in such a way that election officials would be able to update their systems in years ahead.

"For there to be a problem here, you're basically assuming a premise where you have some evil and nefarious election officials who would sneak in and introduce a piece of software," he said. "I don't believe these evil elections people exist."

Still, he said, the company will in the coming months solve the vulnerability, but not before most primary elections occur.

In places where the machines are used, most election officials said they were not worried.

"We're prepared for those types of problems," said Deborah Hench, the registrar of voters in San Joaquin County, Calif. "There are always activists that are anti-electronic voting, and they're constantly trying to put pressure on us to change our system."

Aviel Rubin, a professor of computer science at Johns Hopkins University, did the first in-depth analysis of the security flaws in the source code for Diebold touch-screen machines in 2003. After studying the latest problem, he said: "I almost had a heart attack. The implications of this are pretty astounding."

Photo: Retrieving votes from an electronic machine in Cleveland on May 3. (Photo by Jamie-Andrea Yanak/Associated Press)



Those familiar with the actual election process—by and large run by honest people but historically subject to partisan politicking, dirty tricks and sloppy practices—are less sanguine. "It gives me a bit of alarm that the voting systems are subject to tampering and errors," says Democratic Rep. William Lacy Clay, who worries that machines in his own St. Louis district might be affected by this vulnerability. (In Maryland and Georgia, all the machines are Diebold's.)

The Diebold security gap is only the most vivid example of the reality that no electronic voting system can be 100 percent safe or reliable. That's the reason behind an initiative to augment these systems, adding a paper receipt that voters can check to make sure it conforms with their choices. The receipt is retained at the polling place so a physical count can be conducted. "When you're using a paperless voting system, there is no security," says David Dill, a Stanford professor who founded the election-reform organization Verified Voting.

To their credit, 26 states have taken action to implement paper trails. But the U.S. Congress has yet to pass legislation introduced last year by Rep. Rush Holt, Democrat of New Jersey, that would extend this protection nationwide. Holt says his bill is slowly gaining support. "The voters are saying that every vote should count, and the only way to do this is by verified audit trails," he says. But even an optimistic scenario for passage would challenge his goal of mandatory paper receipts for November's elections. In other words, it's unlikely that every voter using an electronic voting device in 2006 will know for sure that his or her vote will be reflected in the actual totals. Six years after the 2000 electoral debacle, how can this be?

© 2006 MSNBC.com

URL: <http://msnbc.msn.com/id/12888600/site/newsweek/>

