Analysis of an Electronic Voting System

Johns Hopkins Information Security Institute Technical Report TR-2003-19, July 23, 2003

Authors

Tadayoshi Kohno
Adam Stubblefield
Aviel D. Rubin
Dan S. Wallach

Paper: PDF

Response to Diebold's Technical Analysis

Diebold posted an analysis of our report at http://www2.diebold.com/checksandbalances.pdf. Throughout their document, they refer to details of our paper as "allegations," and they attempt to argue away these allegations with logic that is often contrived.

We have no personal ill-will toward Diebold as a company; our aim was to provide a technical analysis of the code that we had at our disposal. While our conclusions have upset those who stand to lose financially from these conclusions and those who are embarrassed by decisions they have made without the knowledge of the insecurity in the code, we firmly stand behind our findings.

Diebold claims that we are not very familiar with the election processes. However, we have extensive academic and industrial experience in software engineering, computer security, and cryptography, which forms the bases of our analysis. In this response, we show that Diebold's arguments often miss the point, do not address many of our most serious findings, and demonstrate a considerable lack of knowledge of the technical matter, including a misunderstanding of technical terms such as "safe language," as we describe below.

Also, Diebold criticized our network-based attacks as being unrealistic since the voting machines will not be networked in practice. The Diebold code we examined contains many different configuration options, including the use of wired or wireless networks and the use of modems. Any communication, whether wired or wireless, whether over the Internet or over private phone lines, is fair game for an analysis of what can be intercepted by an intruder. If there is no such communication, *only then* would the Diebold system be safe against such attacks. Our paper carefully stated these assumptions, while Diebold's response blurs this distinction.

Rather than respond with a 35 page point by point rebuttal and risk a continuously growing exchange, we focus on a few examples of Diebold's misinformation, and we show some examples of security problems that appeared in our report and for which there were no counter arguments in the Diebold "analysis". The primary example is Diebold's claim, which was widely cited in the press, that we ran the code on a different platform from the one that was intended. While this is true, it in no way reflects on our analysis. In fact, the main reason that we ran the code at all was to test whether or not it worked, and thus to help draw on opinion about whether or not it was production code. Our entire paper and our entire analysis were based on manual inspection of the source code. Thus, the platform on which we ran the code did not play into any of our findings, and Diebold's attempt to focus attention on that is a clear effort to misdirect readers. Unfortunately, many reporters and election officials have latched onto this issue as though it was meaningful. We could have written the same paper without running the code, and perhaps we should have never even mentioned that we ran it, to avoid this confusion.

At the end of our response, we provide a list of questions people should ask, not only of Diebold, but of any direct recording electronic (DRE) voting system. If these systems are going to be used for our elections, they deserve the scrutiny that we, and others, can bring to them. Voting systems are one of the pillars of democracy. If they fail, democracy itself will fail with them.

Cryptography

Cryptography is an important part of good security designs. It is, however, notoriously difficult to get right. We have claimed that, in the Diebold code we examined, "cryptography, when used at all, is used incorrectly." We stand by this claim.

Throughout Diebold's response they seek to marginalize their dependence on cryptography (e.g., Diebold's responses to "allegation 1" and "allegation 8"), however the fact remains that although cryptography was used in their design, it was used incorrectly. In their response to allegation 8, Diebold states that our claims are "based on the presumption that there is a single correct means of using cryptography." This play on words disguises the real problem: While there is no single correct use of cryptography, the existence of many correct uses does not imply that the cryptography in the Diebold code is used correctly. As we have described in the full paper, and will summarize below, every use of cryptography in the Diebold code is flawed.

The first problem we address is key management. In modern security systems, key management is one of the chief design challenges present when using cryptography to protect data against eavesdroppers and intruders. Such systems need to be careful to change the keys on a regular basis and to limit the damage that can occur should any one key be compromised. Systems like SSL/TLS, used to communicate securely with e-commerce web sites, have remarkably sophisticated key management systems that have been carefully studied by researchers worldwide.

In the Diebold code we analyzed, both the keys for the smartcard and the keys used to encrypt the votes were static entries in the source code. This means that the same keys are used on every voting device. Thus, an attacker who was able to compromise a single voting device would have access to the keys for *all* other voting devices running the same software.

In Diebold's response to "allegation 28" they acknowledge the problems with a hard-coded smartcard key by claiming that "[t]his issue has since been resolved in subsequent versions of the software." They do not, however, explain *how* this issue has been resolved. Moreover, in their response to "allegation 43," they seemingly admit that the keys used to protect the votes are still static and fixed. Instead they claim that "[a]n attacker would need access to *both* the source code *and* the physical storage." This is not correct. The attacker only needs access to the physical storage as the key is also stored in the executable code.

A second set of problems has to do with the way that the Diebold code encrypts the votes and audit logs. The files that hold the votes are encrypted using the Data Encryption Standard (DES) algorithm in CBC mode. There are problems with the use of both DES and the CBC mode, as we describe below.

In their response to "allegation 44," Diebold states that "[t]here are stronger forms of compression than DES, but the authors' implication that the keys can be recovered `in a short time' is deliberately misleading." We assume that Diebold meant to claim that there are stronger *encryption* algorithms available, as DES is not a compression algorithm. To support our "in a short time" claim we cited Cracking DES. This work describes the design and construction of a machine specifically engineered to recover DES keys. Using 1998 technology, the machine cost under $250,000 and was able recover a DES key in under 3 days. With today's computer technology such a machine could be made both significantly faster and less expensive. That Diebold considers the possibility of such a machine being used to find keys for an election machine ``incredibly unrealistic" demonstrates a misunderstanding of the threat model. It is not inconceivable that a well funded adversary such as the intelligence service of a foreign government would be interested in tampering with the results of a U.S. election. We note, however, that the DES Cracker was financed not by a government or university, but by a private individual. Of course, since the Diebold code included a static key, no cracking is required to compromise the security of the system if any one voting terminal can be stolen ahead of the election and disassembled to learn its key.

Not included among Diebold's list of allegations is our statement that the Diebold code uses DES incorrectly. On page 15, we note that "DES is being used in CBC mode which requires an initialization vector to ensure its security." We go on to show that the Diebold code does not provide the necessary initialization vectors. A detailed explanation of this problem is highly technical; we refer the interested reader to A Concrete Security Treatment of Symmetric Encryption: Analysis of the DES Modes of Operation. That Diebold does not attempt to refute this claim is troubling, especially given that in "allegation 8" they state that "the cryptography in the software is used as the developers intended." Though Diebold qualifies this by claiming that "additional security measures" and "future development" might be needed, such a clear error demonstrates a lack of cryptographic knowledge on the part of the

developers.

Also omitted from Diebold's list of allegations is a third problem: the use of a CRC as a method of providing data integrity. It is a common misconception outside of the cryptographic community that encrypted data can not be meaningfully modified by an attacker. Unfortunately, this misconception is false. There are, however, cryptographic primitives known as Message Authentication Codes (MACs) which can detect such tampering. MACs are used in many widely publicized protocols, including the security protocols for the Internet. However, instead of using such a MAC, the Diebold code uses a non-cryptographic checksum called a CRC to detect whether a file has been tampered with. This is completely insecure as is discussed on page 15 of our paper. The use of CRCs instead of MACs has long been documented in the security literature as a very serious mistake.

This list of problems is not meant to be complete; there are other issues discussed in the paper such as the protected counter and the random number generator which we do not feel that Diebold has successfully refuted. However, we do believe that this list is sufficient to show the lack of cryptographic expertise utilized in the system's design. If, as Diebold claims, "the cryptography in the software is used as the developers intended" we have no faith that any cryptography present in future versions of the code will be used correctly.

Smartcards

Diebold's response raises some issues with respect to our smartcard-based attacks against the voting terminals. We first point out that, as we stated in our original report and as Diebold agrees in their response to "allegation 21," an adversary using a special "attack smartcard" could cast multiple votes. (An attacker could also make homebrew administrator and ender cards. For brevity, we focus our discussion on voting multiple times.)

The first interesting technical question that this raises is: How easy would it be for an adversary to make such a special "attack smartcard?" Diebold's response suggests that making such an "attack smartcard" would be very difficult. As we stated in our original report, we believe otherwise and explain our reasoning in the following sections. The second interesting technical question is: What happens if an adversary inserts multiple votes? We treat these technical questions separately.

How to make "attack smartcards"

Let's look at this question from two perspectives. First we'll assume that the adversary knows the Diebold source code. We believe this is a reasonable assumption since, as recently exhibited, source code cannot always be kept secret. However, we will also examine this for the case where the adversary does not know the Diebold source code.

If the adversary knows the Diebold source code, then the adversary will know the protocol between the smartcard and the voting terminal. The adversary's goal in this case is to make his own smartcard that tricks the voting terminal into believing that it is a legitimate smartcard when it is really an "attack smartcard" or "homebrew smartcard." We observe that a computer-savvy adversary with a few hundred dollars could produce his own attack smartcard. The adversary doesn't have to work for Diebold or their third-party smartcard vendor. How would such an adversary go about producing such an attack smartcard? By purchasing a user-programmable smartcard (perhaps a Java Card) and a smartcard reader/writer and programming it appropriately. The adversary would know how to program the smartcard since he can deduce the protocol between the smartcard and the terminal from the source code. (See our paper for further discussions.)

Now what if the adversary didn't know the Diebold source code? As we note in our paper, by inserting a "wire-tap device" between the voting terminal and the smartcard, an adversary could learn enough about the protocol between the terminal and the smartcard to create his own smartcards for use by some conspirator later in the day. Diebold does point out that such an attack might be risky since the adversary might get caught. Of course, all it takes is one malicious poll worker/volunteer to ensure that some adversary somewhere doesn't get caught. Subsequently, that adversary could share his knowledge with others. (As an aside, Diebold makes the claim that voters must "sign in" when they vote, and thus some of the attacks we describe are "high risk." To this we remark that in many states it is illegal to ask the voter to present identification, and thus an attacker can pretend to be some other registered voter.)

In our paper, we stated that: "As we noted in Section 3.1, some smartcards allow a user to get a listing of all the files on a card. If the system uses such a card and also uses the manufacturer's default password ..., then an attacker, even without any knowledge of the source code and without the ability to intercept the connection between a legitimate card and a voting terminal, but with access to a legitimate voter card, will still be able to learn enough about the smartcards to be able to create counterfeit voter cards." Diebold responded that all their voter cards do not use the manufacturer password. This makes the construction of homebrew cards harder for an adversary without knowledge of the source code and who doesn't want to or can't use a wire-tap device. That is good. But, as we pointed out above and in our paper, it is not unreasonable to assume that some other adversary might have access to the source code or a wire-tap device.

Diebold uses an insecure protocol that makes them vulnerable to counterfeit smartcards. Modern smartcards can perform cryptographic operations, allowing for more sophisticated protocols. If Diebold used such protocols, their system would be robust against our attacks.

What happens if an adversary (or adversaries) vote multiple times?

First, if an adversary votes multiple times, the tally of votes presented by the voting terminal will be incorrect. As we note in our paper: *If* there are no additional mechanisms to detect the presence of over-votes, then an adversary might successfully modify the outcome of the election. However, we are pleased to learn that (from Diebold's response): "before results are made official, the signatures are

reconciled with the number of ballots cast on the voting machines. If the totals do not reconcile, an investigation is launched."

On page 10 of our paper we did note that "If we assume the number of collected votes becomes greater than the number of people who showed up to vote, and if the polling locations keep accurate counts of the number of people who show up to vote, then the back-end system, if designed properly, should be able to detect the existence of counterfeit votes." But, continuing in that same paragraph from page 10 of our report, "... there will be no way for the tabulating system to count the true number of voters or distinguish the real votes from the counterfeit votes. This would cast serious doubt on the validity of the election results." Diebold claims that if the existence of counterfeit votes were to be detected, an investigation would be launched. But what does that mean? It would seem impossible to call all the legitimate voters back to re-vote, especially if counterfeit votes were detected in a large number of precincts, and it is not clear that this is even legal. As we say in our paper, this could cast doubt on the validity of the election results. All it takes is for one person to figure out an attack for it to become widespread. On the Internet, most attacks are from so-called "Script Kiddies" who run malicious programs designed by others. This is the common method for attacks to spread, and so we must concern ourselves with what an intelligent, dedicated, and well-funded adversary could accomplish, as opposed to asking how sophisticated someone must be to launch this attack. Furthermore, it goes without saying that many people and organizations have a great amount at stake in an election. Voting systems must be robust against even the most sophisticated and well-funded adversary.

Software Engineering

Our original paper makes strong claims about Diebold's software engineering processes. We claim their coding standards are unsuitable for the security requirements that a voting terminal must face. Below we outline how we came to these conclusions.

In their response to "allegation 86," Diebold claims that their development process "followed common professional software engineering practice." This may very well be the case. However, building software that's intended to be secure, from day one, is different from traditional software engineering. Good designs start from an honest evaluation of the threats the system will face, and then a high-level design is gradually refined down into the specifics of the implementation. At each step of the process, the design needs to be carefully reviewed, and best practices should be employed to guarantee that high-level goals aren't lost in the translation to low-level code.

Diebold claims in their response to "allegation 78" that "the correctness of the software has been proven though [sic] an extensive testing process, both within the company and by independent testing authorities, and ultimately though [sic] logic and accuracy tests by the election officials themselves." Unlike traditional software engineering, where testing can be used to show that a feature functions correctly under normal circumstances, security engineering is concerned with *abnormal circumstances*.

Thus, testing can only be used to show that a system is not vulnerable to a given set of attacks, not that a system is secure. Diebold also claims that in their "comprehensive top-to-bottom" code reviews they concentrate on "those parts of the code that tabulate vote results." Again, they are following software design methodology and not security design methodology. Even if the code that is designed to tabulate the votes works correctly, that fact provides no information as to whether the vote tabulations can then be modified from some unassociated part of the code.

The correct way to design a secure system is to first identify the threats that such a system must defend against, then create detailed design documents that shows how each threat is mitigated. Only then can the design be implemented. This is the very same process mandated by the Department of Defense for certifying highly secure computer systems used to handle classified message traffic in the so-called Orange Book.

Maybe Diebold has some great high-level design documents. We never saw them, but we also never saw any reference to them in the source code. If the code were written in a truly rigorous fashion, you'd expect to see commentary in the code quoting chapter and verse from the design documents. It's just not there.

(For more information on how security engineering is fundamentally different from software engineering we refer the interested reader to Building Secure Software or Security Engineering.)

Furthermore, when building software that's meant to be robust against attacks, one of the most painful lessons of the past decade of computer science has been the damage that can be caused when a program is vulnerable to memory-corruption and type-confusion attacks (including the well known "buffer overflow"). These vulnerabilities are a special case of a general problem with the C and C++ programming languages (among others). These languages are not *memory safe* or *type safe*. While formally defining these terms is beyond the scope of this document, a programming language is *safe* if and only if no primitive operation in any program ever misinterprets data. For example, an integer is never misinterpreted as a pointer to a string. (This terminology is standard in the computer security and programming language communities; see this list for some examples). Such enforcement does not prevent all software bugs, but it does guarantee that a program's operation is predictable, and many important classes of bugs, including buffer overflows, can be *guaranteed* to never occur. Modern programming languages, including Java, C#, Ada, Modula-3, and many others have this important safety property. When Diebold's claims that "programming in any language can be safe or unsafe" (their responses to allegations 11, 71, and 72), they are really saying that they are unaware of one of the most fundamental improvements in software engineering since the invention of high-level programming. Even though Diebold criticizes our report for not "offering any evidence of such an exploit or failure" (their response to allegation 73), they cannot prove the *lack* of any such exploits or failures. Had they implemented their software with a modern programming language, such proof could be easily demonstrated.

Questions you can ask of vendors

Several people have asked us what questions they should be asking voting system vendors with regards to security. Here are some useful ones:

- Has your system been reviewed by a large number of outside security experts?
  - If so, who?
  - What are their credentials?
  - Do their areas of expertise cover a wide area of specialties within the discipline of cryptography and computer security?
  - Can we see an executive summary of their reports?

- Do you allow the public to review the security and reliability of your voting system's source code?
  - Is the security of your system dependent on your source code being secret?
  - If so, how do you address the fact that the source code could leak to the public (or to well-funded adversaries)?
  - And how do you address the fact that an attacker might be an insider who knows the source code?

- Would you be willing to have a panel of outside security experts review the source code for your system?
  - Would you allow them to publish an executive summary of their findings?
  - If not, why not?

- Who designed and developed the source code used in your systems?
  - What are their credentials with respect to cryptography and computer security?
  - Where were they trained?
  - Have these developers worked on cryptography and computer security in other systems outside of voting software?

- How confident are you in the security and reliability of your product? Will you "certify" the security and reliability of your product?
  - Will you offer a full refund, plus "damages," if somebody purchases your equipment and later find that it is vulnerable to certain types of attacks? (Which types of attacks?)
  - Will you offer a full refund, plus "damages," if after an election it is determined that more votes were collected than people who voted (on a given terminal), but that it cannot be determined which were the legitimate votes?
  - Will you offer a full refund, plus "damages," if after an election it is determined that your machines reported an inaccurate total (either because of an attack or a system glitch)?
  - Will you offer a full refund, plus "damages," if after an election it is determined that

voters' anonymity was compromised, allowing votes to be bought and sold?
    - Under what other situations would you offer a full refund, plus "damages?"

- In your system, what can voters do if they feel that their votes were not recorded properly?
    - Are there any mechanisms for voters to verify their votes are correct?
    - What happens in the case of a dispute?
    - Is a manual recount (i.e., not requiring any computer software) possible?

- Does your system conform to the requirements of the Holt bill? Details can be found at http://holt.house.gov/issues2.cfm?id=5996.

Last updated August 1, 2003