



## Foreword

---

In early July 2003 I received a call from David Dill, a computer science professor at Stanford University. Dill informed me that the source code to an electronic voting machine produced by Diebold Election Systems, one of the top vendors, had leaked onto the Internet, and that perhaps it would be worth examining it for security vulnerabilities. This was a rare opportunity, because voting system manufacturers have been very tight with their proprietary code. What we found was startling: Security and coding flaws were so prevalent that an attack might be delayed because the attacker might get stuck trying to choose from all the different vulnerabilities to exploit without knowing where to turn first. (Such delay tactics are *not* recommended as a security strategy.) There were large, complex chunks of code with no comments. There was a single static key hard wired into the code for encrypting vote tallies. Insecure pseudorandom number generators and noncryptographic checksums were used. And inspection of the CVS logs revealed an arbitrary, seemingly ad hoc source code management process. And then there were the serious flaws.

Was the Diebold voting machine example an isolated incident of poor quality control? I don't think so. Many companies such as Diebold are hard pressed to get their products to market before their competitors. The company with the best, functionally correct system wins. This incentive model rewards the company with the product that is available first and has the most features, not the one with the most secure software. Getting security right is very difficult, and the result is not always tangible. Diebold was unlucky: Their code was examined in a public forum and was shown to be completely broken. Most companies are relatively safe in the assumption that independent analysts will only get to see their code under strict non-disclosure agreements. Only when they are held to the fire do companies pay the kind of attention to security that is warranted. Diebold's voting machine

code was not the first highly complex system that I had ever looked at that was full of security flaws. Why is it so difficult to produce secure software?

The answer is simple. *Complexity*. Anyone who has ever programmed knows that there are unlimited numbers of choices when writing code. An important choice is which programming language to use. Do you want something that allows the flexibility of pointer arithmetic with the opportunities it allows for manual performance optimization, or do you want a type-safe language that avoids buffer overflows but removes some of your power? For every task, there are seemingly infinite choices of algorithms, parameters, and data structures to use. For every block of code, there are choices on how to name variables, how to comment, and even how to lay out the code in relation to the white space around it. Every programmer is different, and every programmer is likely to make different choices. Large software projects are written in teams, and different programmers have to be able to understand and modify the code written by others. It is hard enough to manage one's own code, let alone software produced by someone else. Avoiding serious security vulnerabilities in the resulting code is challenging for programs with hundreds of lines of code. For programs with millions of lines of code, such as modern operating systems, it is impossible.

However, large systems must be built, so we cannot just give up and say that writing such systems securely is impossible. McGraw and Hoglund have done a marvelous job of explaining why software is exploitable, of demonstrating how exploits work, and of educating the reader on how to avoid writing exploitable code. You might wonder whether it is a good idea to demonstrate how exploits work, as this book does. In fact, there is a tradeoff that security professionals must consider, between publicizing exploits and keeping them quiet. This book takes the correct position that the only way to program in such a way that minimizes the vulnerabilities in software is to understand why vulnerabilities exist and how attackers exploit them. To this end, this book is a must-read for anybody building any networked application or operating system.

*Exploiting Software* is the best treatment of any kind that I have seen on the topic of software vulnerabilities. Gary McGraw and Greg Hoglund have a long history of treating this subject. McGraw's first book, *Java Security*, was a groundbreaking look at the security problems in the Java runtime environment and the security issues surrounding the novel concept of untrusted mobile code running inside a trusted browser. McGraw's later book, *Building Secure Software*, was a classic, demonstrating concepts that could be used to avoid many of the vulnerabilities described in the current book.

Hoglund has vast experience developing rootkits and implementing exploit defenses in practice.

After reading this book, you may find it surprising not that so many deployed systems can be hacked, but that so many systems have not yet been hacked. The analysis we did of an electronic voting machine demonstrated that software vulnerabilities are all around us. The fact that many systems have not yet been exploited only means that attackers are satisfied with lower hanging fruit right now. This will be of little comfort to me the next time I go to the polls and am faced with a Windows-based electronic voting machine. Maybe I'll just mail in an absentee ballot, at least that voting technology's insecurities are not based on software flaws.

*Aviel D. Rubin*

Associate Professor, Computer Science

Technical Director, Information Security Institute

Johns Hopkins University