

Position Paper: Applying Engineering Principles to System Security Design & Implementation

David A. Wheeler

September 30, 2002

INSTITUTE FOR DEFENSE ANALYSES

This position paper outlines some of my positions on applying engineering principles to system security design and implementation (rather than arguing for a single position). I argue that there is a need for security education & training that the same mistakes are made again and again, that security information has been inaccessible and still needs to be more formally available, that defense in depth and failing soft are needed, that it must be easier to use development tools to create secure software, that penetrate and patch don't work (but a variant is necessary), and that open source can help but it's not a panacea. Note that this is personal position paper, and not the position of IDA or the U.S. Department of Defense.

Introduction

It has been reported that when Gandhi was asked, "what do you think about Western civilization?," he answered, "It would be a good idea." In the same vein, what do I think about applying engineering principles to system security design & implementation? Answer: It would be a good idea.

This position paper outlines some of my positions on applying engineering principles to system security design and implementation. In particular:

1. *No education or training* Software developers are almost never given any instruction or training in how to write secure programs. Until this is changed, we will continue to have woefully insecure software.
2. *Same mistakes*. The same mistakes in requirements, design, and implementation, are being made over and over again.
3. *Information has been inaccessible, and is still not formalized*. While information on how to write secure programs has existed for years, it generally has been in inaccessible formats for most programmers.
4. *Defense in depth and fail soft are needed*. Since nothing is invincible, we need to develop systems that have multiple defenses and fail soft when they do fail.
5. *Reduce the number of sharp edges*. Current development tools tend to be "insecure by default;" making it difficult to write secure programs.
6. *Penetrate and patch doesn't work, but a variant is necessary*. The term "penetrate and patch" has different meanings. As I define it, it doesn't work, but a variant is necessary.
7. *Open Source Can Help, but It's No Panacea*. Open source software can improve security, but it's not a miraculous panacea.

No Education or Training

We require plumbers to know about plumbing. In contrast, software developers are generally never provided with the information on how to develop secure applications, and employers neither require it of their employees nor (generally) do they provide any remedial training on how to develop secure applications.

Software developers are almost never given any instruction or training in how to write secure programs. Those who go to universities and take Computer Science are often taught many exotic algorithms, techniques to implement compilers and operating systems, how to determine $O()$, and other subjects. Those who study Software Engineering often learn how to manage large groups of developers (e.g., project management, design strategies, and so on). In either case, the subject of how to develop applications that are resistant to attack is generally not covered in most university curricula. Where are buffer overflows, format strings, input validation, and so on taught? The answer is clear: in general, they aren't.

Even those curricula which specialize in Computer Security often do not cover how to develop attacker-resistant software. I have observed Computer Security curricula that covers various protocols in depth (TCP/IP, IPsec, SSH, SSL, Kerberos), examines common encryption algorithms (such as 3DES), and the purpose of common products such as firewalls. What is sad is that, for most participants, this is a complete waste of time. Few attendees should ever re-implement SSL, for example. Instead, they need to know how to *use* these services to build secure applications. Knowing how they work can help, of course, but simply knowing how these services work will not lead to secure applications. Sadly, the Computer Security curricula I have observed almost never discuss how to write attacker-resistant programs.

Of course, a vast number of software developers do *not* get a degree related to computing; many do not go to a university at all. And in addition, even if all new developers were required to gain this knowledge, that means that nearly all of today's developers would not have this knowledge.

The ACSAC Call for Participation claims that "We repeat the same failures again and again. Indeed, the ubiquitous buffer overflow vulnerability accounts for more than fifty percent of all attacks reported by the Carnegie Mellon CERT." But I argue that the "we" in the sentence is the wrong focus. It is true that "we," the collective of software developers, repeat the same failures again and again. But it is different people that insert the mistakes. Once the issue is explained, in my experience that particular person tries to avoid making that specific mistake again. The failure is that there *no* effective process for ensuring that *all* developers have the information necessary to write secure software.

Since software developers are generally never given any information on how to write secure programs, we should be shocked that any programs resist any attacks at all. Until we can presume that all software developers already know about common vulnerabilities, and how to avoid them, trivial attacks will continue to work.

If the information only goes to the "security engineers," we have failed. Guess what? Software development is done by *software developers*. If only a small subset of software developers have the necessary information, than only a small subset of available software (if that) will be resistant to even trivial attacks.

Same Mistakes

It could be claimed that there are no "software engineering principles" for security, but I reject that argument. Certainly, there are many things that are not known, and simply following a set of principles will not guarantee a secure program. But there are many issues that *are* known, and it has been a monumental failure that this information has not gotten out to typical software developers.

Indeed, there is a standard set of names for these mistakes: buffer overflows (including the subset “stack smashing”), format strings, and so on.

Clearly, a “software engineering principle” should be “avoid common security mistakes,” and developers of secure software must *know* what those mistakes are (see the discussion below on reducing the number of sharp edges in the tools).

Information has been inaccessible, and is still not formalized

Until recently, books have not been available on how to write secure software. Instead, short works that only told small fractions of the issues have been available. The best available was one chapter in a Garfinkel and Spafford’s book *Practical Unix & Internet Security* and some short papers by Matt Bishop. There have been “top ten” lists and short FAQs available on the Internet. All of these were good as far as they went, but since they were all relatively short, they lacked necessary breadth and detail, and all omitted a vast number of critical issues. More detailed information was available if you knew where to look, but few developers had the stamina to find and then read thousands of documents, in an attempt to distill the key information from them. Many of the documents were written from an attacker’s point of view, for example; most developers don’t want to learn how to attack, or even how to defend against a specific attack. Instead, most developers wanted to know a smaller set of general approaches that when applied will defend against a wide range of attacks.

I was so frustrated with this situation that on my own time I wrote a book on how to write secure programs on Unix-like systems, titled *Secure Programming for Linux and Unix HOWTO*. In the hopes that others would actually use and apply its information, I give the book away freely on the Internet (<http://www.dwheeler.com/secure-programs>) – feel free to download it, print it, and redistribute it. More recently, there have been other books published on the subject, *Building Secure Software* by John Viega and Gary McGraw and *Writing Secure Code* by Michael Howard and David LeBlanc (this one only covers Windows systems). With these three works, at least some of the issues are finally available in a form accessible to ordinary developers.

But these are simply books for the ordinary developer. Where is the curricula module that will assure that all future developers will be aware of the issues? Are universities willing to use these resources as textbooks, and if not, how can this information be made available to developers going to universities if the information cannot be embedded into their tools?

Defense in Depth and Fail Soft are Necessary

When asked if approaches like StackGuard encouraged bad programming, Crispin Cowan once said “Plan A: write perfect code. Wait! I need plan B!” I completely agree with this sentiment.

Since nothing is invincible, we need to develop systems that have multiple defenses and fail soft when they do fail. This means that software infrastructure components need to anticipate common failure modes of programs, and reduce their effects. Intrusion prevention techniques (such as Crispin Cowan’s work on StackGuard to reduce stack smashing attacks to at most a denial-of-service attack in most cases and TempGuard) are an excellent means of reducing the effect of attacks. Systems should “fail soft,” instead of assuming that a failure cannot happen. Some of these techniques imply a small loss in performance to increase security. For most applications, this loss is well worth the trade, and yet system vendors are hesitant to actually provide this as the default.

Reduce the Number of Sharp Edges

Current development tools tend to be “insecure by default,” that is, they make it difficult to write secure programs. For example, the two most popular languages used in Red Hat Linux (counting by

lines of code) are C and C++, which seem to be specifically designed to encourage buffer overflows. Until recently, the web programming language PHP was designed so that every variable was controlled by an attacker unless specifically overruled by the developer. Often, cryptographic libraries are hard to use, and require many steps to use practically.

Languages, including their libraries, need to be designed so that “naïve” programs are secure by default, and only allow insecure constructs when they are specifically requested. While recent languages such as C# and Java make a step in this direction, this must include a large set of easy-to-use libraries that naïve developers can call to make “trivially correct” programs (and, if there is a failure, they can be fixed by repairing the library so that one fix repairs *all* programs).

Penetrate and Patch doesn't work, but a variant is necessary.

The term “penetrate and patch” has different meanings. I define “penetrate and patch” as “a development process in which security is not seriously considered during development; instead, the primary means for detecting problems is penetration of the system after development by experts, and fixes are applied to deal with problems they identify.”

Under this definition, “penetrate and patch” just doesn't work well. “Experts” who aren't really experts won't find anything at all. True experts will find a large number of problems in the system. However, because they come in so late in the development cycle, significant changes to the design will not be considered, so “simple fixes” that don't fully address the problems will be all that can be accomplished. And when there are many vulnerabilities, it becomes very likely that there were even more serious undiscovered vulnerabilities not noted by the experts.

However, a variant of “penetrate and patch” is simply trying to find problems *after* much up-front work has been done—I will call this variant “security testing.” This variant *is* necessary, because developers eventually gain blind spots from hidden assumptions after working with their system. A fresh team of people can analyze a system without a developer's hidden assumptions. This includes approaches such as “Red Teams,” fuzz-like random input generators, and operational test & evaluation processes that take the source code (decompiled if necessary) and use pattern-matching programs (like my flawfinder and Viega's RATS) to search for common vulnerabilities. But security testing only works well if vulnerabilities have been actively countered throughout development.

Open Source Can Help, but It's No Panacea

Revealing source code helps both attackers *and* defenders, and the question is who does it help more. There's been a long debate on whether or not open source software¹/free software² (OSS/FS) is more secure than proprietary software. I believe the evidence is generally in OSS/FS's favor, but it isn't as simple as either some proprietary or some OSS/FS proponents wish to make it.

Attackers generally don't need source code to find a vulnerability in programs delivered to customers. Attackers often use “dynamic” approaches, where an attacker runs the program, sends it data (often problematic data), and sees if the programs' response indicates a common vulnerability. Open and closed programs have no difference here, obviously. Attackers may also use “static” approaches by looking at the code. But closed source software doesn't prevent this; attackers can search the machine code for the same vulnerability patterns (using tools such as disassemblers to aid them); they can even

¹ See the Open Source Initiative's definition of “open source software” at <http://www.opensource.org/osd.html>. The opposite is “closed source” or “proprietary” software.

² See the Free Software Foundation's definition of “Free Software” at <http://www.gnu.org/philosophy/free-sw.html>. As used here, the term refers to “freedom” and not “no cost.”

use tools called "decompilers" that turn the machine code back into source code. The output of such tools is not usually very good for people trying to modify the software, but they are quite good when searching for vulnerabilities using pattern matching. Thus, even if an attacker wanted to use source code to find a vulnerability, a closed source program won't really prevent it; the attacker can use a disassembler or decompiler to re-create the source code of the product. It's simply not as hard to attack closed source programs as closed source vendors want users to believe. And this assumes that the source is truly secret; source code can be easily sniffed, uploaded, emailed, placed on media, or copied onto a PDA and transported out by an insider or attacker.

Case in point - Microsoft doesn't give its source code away to just anyone, yet look at all the vulnerabilities that have been found in their code. Clearly, having closed source code isn't a good defense against attackers, and the available quantitative data suggests that at least some OSS/FS programs are far more secure than their proprietary competition³. For example, it's been clearly demonstrated that the Apache web server has had far fewer vulnerabilities and web site defacements than Microsoft's IIS. Web servers are a particularly useful data point, because web server exploitations are far more public than many exploitations; many financial institutions and military organizations would rather hide an exploitation than reveal that one occurred, but they can't easily hide web server exploits. Please note that neither open source software nor proprietary software is immune to security problems. No real programs are perfectly secure; the issue is simply what's better.

Fundamentally, since OSS/FS exposes the source code to examination by everyone, both attackers and defenders, the real question is who does it help more in the end. I believe that when a program began as closed source and is then first made open source, it often starts less secure for its users (through exposure of vulnerabilities), and over time (say a few years) it has the potential to be much more secure than a closed program. If the program began as open source software, the public scrutiny is more likely (than closed source) to improve its security before it's ready for use by significant numbers of users, but there are several caveats to this statement. Just making a program open source doesn't suddenly make a program secure, and just because a program is open source does not guarantee security:

1. The good guys have to actually review the code. The most common reason to review code is that someone wants to add a feature in that particular area of the code. In open source software with many developers, the number of reviewers increases greatly. In addition, in most open source software systems there is usually a trusted set of maintainers, who review the contributions by others (e.g., Linus Torvalds and a lieutenant review proposed changes to the Linux kernel). It is this review process that forces code review to take place (and as a side-effect, trains programmers who make security flaws). Thus, having many developers increases the review, and having many users increases the number of developers (because a subset of users eventually become co-developers). Having many developers, many users, and following common development conventions increases the likelihood of this review.
2. At least some of the people developing and reviewing the code must know how to write secure programs. Hopefully, my freely-available book helps.
3. Once found, these problems need to be fixed quickly and their fixes distributed.

This work was conducted under contract DASW01 98 C 0067, Task T-X0-0000. The publication of this IDA memorandum does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that

³ http://www.dwheeler.com/oss_fs_why.html#security

Agency. The material may be reproduced by or for the U.S. Government pursuant to the copyright license under the clause at DFARS 252.227-7013 (NOV 95).