Building Secure Software

A Difficult But Critical Step in Protecting Your Business

Gary McGraw, Ph.D.

Cigital, Inc.
21351 Ridgetop Circle
Suite 400
Dulles, Virginia
20166
info@cigital.com

www.cigital.com

Building Secure Software

The Internet continues to change the role that software plays in the business world, fundamentally and radically. Software no longer simply supports back offices and home entertainment; instead, software has become the lifeblood of our businesses and has become deeply entwined into our lives. The invisible hand of Internet software enables e-business, automates supply chains, and provides instant, worldwide access to information. At the same time, Internet software is moving into our cars, our televisions, our home security systems, and even our toasters. All of this software introduces security risk.

Software is the biggest problem in computer security today.[1] A majority of organizations invest in security by buying and maintaining a firewall, but go on to let anybody access multiple Internet-enabled applications through the firewall. These applications are often remotely exploitable, rendering the firewall impotent (not to mention the fact that the firewall is often a piece of fallible software itself).

Contrary to popular belief, cryptography alone cannot solve the software security problem. *Trust in Cyberspace* reports, for example, that 85% of CERT security advisories (through 1998) could not have been prevented with cryptography.[2] Internet-enabled applications present the largest category of security risk today. Real attackers compromise software.

The ultimate answer to the computer security problem lies in making software behave. Current approaches, based on fixing things only after they have been exploited in fielded systems, address only symptoms, ignoring the cause of the problem. This paper discusses an approach to security that goes beyond firewalls and cryptography to treat software and software development as essential aspects of a mature risk management approach to security. I present a unified view that emphasizes the understanding and management of both architecture and implementation risks.

Software Security Woes

A Growing Problem

Simply put, security holes in software are common. The December 2000 issue of *Industry Standard* featured an article titled “Asleep at the Wheel,” that emphasizes the magnitude of today’s security problem.[3] Using numbers derived from Bugtraq (a mailing list dedicated to reporting security vulnerabilities), author David Lake showed the number of new vulnerabilities reported monthly from the start of 1998 until the end of September 2000 (*see Figure 1*).

According to this data, the number of software holes being reported is growing. These figures suggest that approximately 20 new vulnerabilities in software are made public each week. Note that “tried-and-true” software may not be as safe as one might think; many vulnerabilities that have been discovered in software existed for months, years and even decades before discovery.

1 McGraw, Gary. “Software Assurance for Security.” IEEE Computer, Vol. 32, No. 4 (April 1999), pp. 103-105.
2 Schneider, Fred (ed.), *Trust In Cyberspace*. National Academy Press, 1998.
3 Lake, David. “Asleep at the Wheel.” *Industry Standard* (11 December 2000).

Figure 1: Bugtraq Vulnerabilities by Month from January 1998 to September 2000

The consequences of security flaws vary. Consider that the goal of most malicious hackers is to take control of a networked computer. Attacks tend to be either remote or local. In a remote attack, a malicious attacker can break onto a machine that is connected to the same network, usually through some flaw in software. If the software is available through a firewall, then the firewall will be powerless to protect the computer. In a local attack, a malicious user can gain additional privileges on a machine—usually administrative privileges. Once an attacker has a foothold on a target machine, it is difficult to keep him or her from getting administrative access. Operating systems and the privileged applications that are found on such machines constitute such a large and complex body of code that the presence of some as-yet-undiscovered security hole is always likely.

Trends Affecting Software Security

Most modern computing systems are susceptible to software security problems, so why is software security a bigger problem now than in the past? I argue that a small number of trends have a large influence on the growth and evolution of the problem.[4]

Networks Are Everywhere

The growing connectivity of computers through the Internet has increased both the number of attack vectors and the ease with which an attack can be made. This puts software at greater risk. More and more computers, ranging from home PCs to systems that control critical infrastructures (e.g., the power grid), are being connected to enterprise networks and to the Internet. Furthermore, people, businesses, and governments are increasingly dependent upon network-enabled communication such as e-mail or Web pages provided by information systems. Unfortunately, as these systems are connected to the Internet, they become vulnerable to software-based attacks from distant sources. An attacker no longer needs physical access to a system to exploit vulnerable software.

Because access through a network does not require human intervention, launching automated attacks is relatively easy. The ubiquity of networking means that there are more software systems to attack, more attacks, and greater risks from poor software security practice than in the past.

Systems are Easily Extensible

A second trend negatively affecting software security is the degree to which systems have become extensible. An extensible host accepts updates or extensions, sometimes referred to as mobile code, so that the functionality of the system can be evolved in an incremental fashion.[5] For example, the plug-in architecture of Web browsers makes it easy to install viewer extensions for new document types as needed. Today's operating systems support extensibility through dynamically loadable device drivers and modules. Today's applications, such as word processors, e-mail clients, spreadsheets, and Web-browsers, support extensibility through scripting, controls, components, and applets.

From an economic standpoint, extensible systems are attractive because they provide flexible interfaces that can be adapted through new components. In today's marketplace, it is crucial that software be deployed as rapidly as possible in order to gain market share. Yet the marketplace also demands that applications provide new features with each release. An extensible architecture makes it easy to satisfy both demands by allowing the base application code to be shipped early and feature extensions to be shipped later as needed.

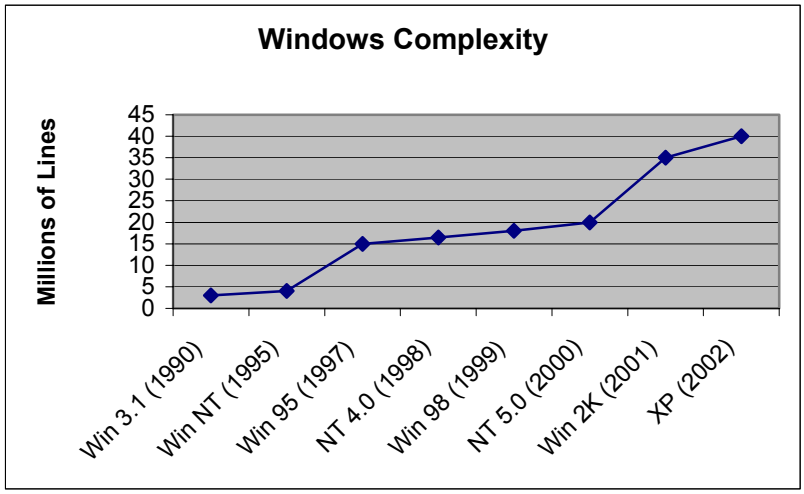
Unfortunately, the very nature of extensible systems makes it hard to prevent software vulnerabilities from slipping in as an unwanted extension. Advanced languages and platforms, including Sun Microsystem's Java and Microsoft's .NET Framework, are making extensibility commonplace.

System Complexity is Rising

A third trend impacting software security is the unbridled growth in the size and complexity of modern information systems, especially software systems.

A desktop system running Windows NT and associated applications depends upon the proper functioning of the kernel as well as the applications to ensure that vulnerabilities cannot compromise the system. However, NT itself consists of at least 20 million lines of code, and end user applications are becoming equally, if not more, complex. When systems become this large, bugs cannot be avoided. Figure 2 shows how the complexity of Windows (measured in lines of code) has grown over the years.

Figure 2: Growth of the Microsoft Operating System Code Base From 1990 to 2002



[4] Interestingly, these three general trends are also responsible for the alarming rise of malicious code [McGraw and Morrisett, 2000].
[5] McGraw, Gary, and Edward Felten. *Securing Java: Getting Down to Business with Mobile Code*. John Wiley & Sons, 1999.

The complexity problem is exacerbated by the use of unsafe programming languages (e.g., C or C++) that do not protect against simple kinds of attacks, such as buffer overflows. In theory, we could analyze and prove that a small program was free of problems, but this task is impossible for even the simplest desktop systems today, much less the enterprise-wide systems used by businesses or governments.

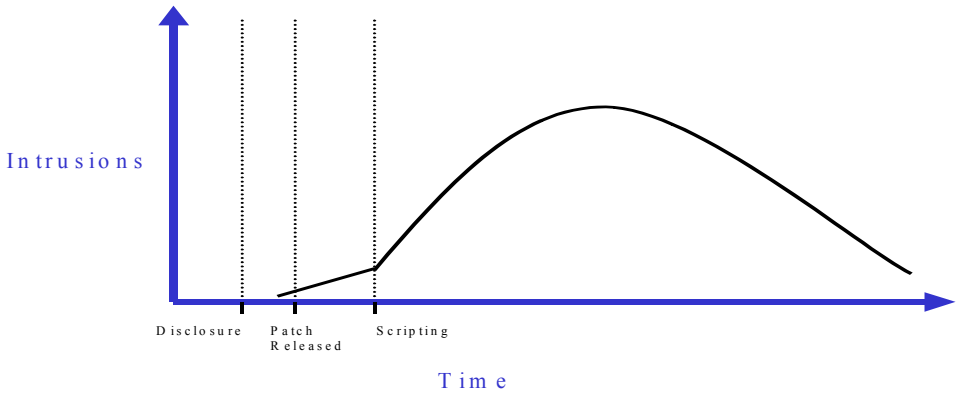
Penetrate and Patch is Bad

Many software vendors fail to understand that security is not an add-on feature. They continue to design and create products while paying little attention to security. They start to worry about security only after their product has been publicly (and often spectacularly) broken by someone. Then they rush out a patch instead of coming to the realization that designing security in from the start might be a better idea.

My goal is to minimize the unfortunately pervasive "penetrate-and-patch" approach to security and thus prevent the problem of desperately having to come up with a fix to a problem that is being actively exploited by attackers.[6] In simple economic terms, it is orders of magnitude cheaper and more effective to find and remove bugs in a software system before its release than trying to fix systems after release.[7] Designing a system for security, carefully implementing the system, and testing the system extensively before release present a much better alternative.

The fact that the existing penetrate-and-patch framework is so poorly implemented is yet another reason why the approach needs to be changed. Researchers Bill Arbaugh, Bill Fithen, and John McHugh discuss a life-cycle model for system vulnerabilities that emphasizes how big the problem is.[8] Data from their study shows that intrusions increase once a vulnerability is discovered, the rate continues to increase even though the vendor releases a patch, and exploits continue to occur even after the patch is issued – sometimes years after (*see Figure 3*). It takes a long time before most users and administrators upgrade to patched software versions.

Figure 3: Average Curve of Number of Intrusions for a Security Bug Over Time, as Reported by Arbaugh, Fithen and McHugh, 2000.



[6] McGraw, Gary. "Testing for Security During Development: Why We Should Scrap Penetrate-and-Patch." *IEEE Aerospace and Electronic Systems*, Vol. 13, No. 4 (April 1998), pp. 13-15.

[7] [7] Brooks, Frederick, Jr. *The Mythical Man-Month: Essays on Software Engineering* (2nd Edition). Addison-Wesley, 1995.

[8] Arbaugh, Bill, Bill Fithen, and John McHugh. "Windows of Vulnerability: A Case Study Analysis." *IEEE Computer*, Vol. 33, No. 12 (December 2000), pp. 52-59.

Software Quality Management For Security

There is no substitute for working software security as deeply into the software development process as possible, taking advantage of the engineering lessons software practitioners have learned over the years. Which particular software process is followed is not as important as the act of thinking about security as software is designed and built.

Both software engineering discipline and security standards such as the Common Criteria both provide many useful tools that good software security can leverage. The key to building secure software is treating software security as risk management and applying the tools in a manner that is consistent with the purpose of the software itself.

Bricks, Walls, And Software Security Risk Analysis

The aphorism "keep your friends close and your enemies closer" applies quite aptly to software security. Software security is really risk management and thus includes risk identification and assessment as a critical step. The key to an effective risk assessment is expert knowledge of security. Being able to recognize situations where common attacks can be applied is half the battle.

Software security risks come in two main varieties: architectural problems and implementation errors. Most software security material focuses on implementation errors leading to problems such as buffer overflows, race conditions, randomness problems, and a handful of other common mistakes. These issues are important, but focusing solely on the implementation level will not solve the software security problem.

Building secure software is like building a house. I liken correct use of security-critical system and library calls (such as strncpy() versus strcpy()) to using solid bricks as opposed to using bricks made of sawdust. The kinds of bricks you use are important to the integrity of your house, but even more important (if you want to keep bad things out) is having a solid foundation, four walls and a roof in the design. The same thing goes for software: what system primitives you use and how you use them is important, but overall design properties often count for more. The following two sections are devoted to bricks and walls, respectively.

Implementation Risks (Brick Problems)

Software security is often approached as an exercise in addressing a clear set of known problems found in software implementations. Though this is not a complete solution, implementation risks are an important class of problems that deserve attention. The problems highlighted in this section are the most common ways of exploiting software at the lowest levels.

Seven Common Problems[9]

1. Buffer overflows

Buffer overflows have been causing serious security problems for decades.[10] Buffer overflows accounted for more than 50% of all major security bugs resulting in CERT/CC advisories in 1999, and the data show that the problem is growing instead of shrinking. The root cause behind buffer overflow problems is that C is inherently unsafe, as is C++. There are no bounds checks on array and pointer references, and there are many unsafe string operations in the standard C library. For these reasons, it is imperative that C and C++ programmers writing security-critical code learn about the buffer overflow problem.

The most dangerous buffer overflows result in stack-smashing attacks, which target a specific programming fault: careless use of data buffers allocated on the program's runtime stack (which contains local variables and function arguments). The results of a successful stack-smashing attack can be very serious. A creative attacker taking

[9] Note that there are many other implementation-level software security problems. The seven outlined are the most commonly encountered.

10 Wagner, David, Jeffrey Foster, Eric Brewer, and Alexander Aiken. "A First Step Towards Automated Detection of Buffer-Overflow Attacks." In Proceedings of the Year 2000 Network and Distributed System Security Symposium (NDSS). Internet Society, 2000.

advantage of a buffer overflow vulnerability through stack smashing can usually run arbitrary code. The idea is pretty straightforward: place some attack code somewhere (for example, code that invokes a shell) and overwrite the stack in such a way that control gets passed to the attack code.

strcpy is one of the most notorious of all dangerous library functions. The following code is a dangerous use of strcpy that can be leveraged into a stack-smashing exploit:

```
void main(int argc, char **argv) {
  char program_name[256];

    strcpy(program_name, argv[0]);
}
```

ITS4 and Stackguard are both extremely useful in combating buffer overflows.[11, 12]

2. Race conditions

Race conditions are possible only in environments where there are multiple threads or processes occurring at once that may potentially interact (or some other form of asynchronous processing). The term “race condition” implies a race going on between the attacker and the assumptions coded into the program by a developer. In fact, the attacker must “race” to invalidate assumptions about the system that the programmer may have made in the interval between operations, say between the time a parameter is checked and when it is subsequently used. A successful attack involves a quick and dirty change to the situation in a way that has not been anticipated. Problems with file access control and assumptions about the condition of the filesystem have been studied extensively.[13]

Programmers who have experience with multithreaded programming have almost certainly encountered race conditions whether or not they know the term. Race conditions are an insidious problem, because a program that seems to work fine may still harbor them. They are very hard to detect, especially if you’re not looking for them. They are often difficult to fix, even when you are aware of their existence. In a world where multithreading, multiprocessing, and distributed computing are becoming more and more prevalent, race conditions will continue to become a bigger and bigger problem. ITS4 is an effective tool against classical race conditions.

3. Access control problems

Once users have successfully authenticated themselves to a system, the system needs to determine what resources each user should be able to access. There are many different access control models for answering that question. Some of the most complicated are used in distributed computing architectures and mobile code systems, such as CORBA and Java’s EJB models. Often, access control systems are based on complex mathematical models that might be hard to use in practice. Misuse of complex access control systems is a common source of security problems in software.

---

[11] Viega, John, J.T. Bloch, Tadayoshi Kohno, Gary McGraw. “ITS4: A Static Vulnerability Scanner for C and C++ Code.” In *Proceedings of Annual Computer Security Applications Conference*. IEEE Computer Society, 2000.

[12] Cowan, Crispin, et. al. “Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks.” In *Proceedings of the Seventh USENIX Security Symposium*, pp. 63-77, USENIX (Advanced Computing Systems Association), 1998.

[13] Bishop, Matt, and Mike Dilger. “Checking for Race Conditions in File Access.” *Computing Systems*, Vol. 9, No. 2 (Spring 1996) pp. 131-152.

4. Randomness problems

Random numbers are important in security for generating cryptographic keys and many other things. Many developers assume that C's random() and similar functions produce unpredictable results. Unfortunately, that's a flawed assumption. A call to random() is really a call to a traditional "pseudo-random" number generator (PRNG) that happens to be quite predictable. Even some developers who are aware of the potential problems convince themselves that an attack is too difficult to be practical. Attacks on random number generators may seem hard, but they are usually reasonably easy to carry out.

5. Misuse of cryptography

Cryptography is something most developers know a little about – usually just enough to be dangerous. There is one sweeping recommendation that applies to every use of cryptography: *Never "roll your own" cryptography!* The next most commonly encountered crypto mistakes include failing to apply cryptography when it's really called for and incorrect application of cryptography even when the need has been properly identified. The usual result of misuse or nonuse of cryptography is that software applications end up vulnerable to a smorgasbord of network-based attacks.

6. Input validation mistakes

Humans have a tendency to make poor assumptions about whom and what they can trust. Even developers have this tendency. Trust isn't something that should be extended lightly. Sound security practice dictates the assumption that everything be untrusted by default, and trust should only be extended out of necessity.

Much of the time, developers do not even realize when they are making decisions about trust. This leads to situations in which a developer writes what seems to be correct, but what turns out to be wrong. There are plenty of tricky problems that can slip by programmers. One common problem that has been around for years but was only first exploited in the year 2000 is a "format string attack" in C. Do not allow untrusted input to have any input at all to a format string.

7. Password problems

Almost everyone who has used a computer in the last 20 years knows what a password is. For better or worse, username/password pairs are the most popular method for authenticating users. The underlying premise of passwords is that the user and an authenticating agent (e.g., the user's machine) share a secret—the password. When it comes time to authenticate, the user types the password, and if it matches the one stored by the authenticating agent, the connection is approved.

Like many security technologies, the idea is simple and elegant, but getting everything exactly right is much harder than it first appears. Two areas of particular vulnerability include password storage and user authentication with passwords.

Auditing Code with ITS4

Trying to understand and analyze an entire program implementation is more work than most people are willing to undertake. While a thorough review is possible, most people are willing to settle for a "good enough" source code audit focused on finding common problems. With that in mind, a basic strategy for auditing source code is to:

- **Identify all points in the source code where the program might take input from a user** (remote or local). Similarly, look for any places where the program might take input from another program or any other potentially untrusted source.
- **Treat any internal API that gathers input as if it were a standard set of input calls.**
- **Look for symptoms of problems.** Experience in spotting problems (both statically and dynamically) is important. Much of what to look for consists of function calls to standard libraries that are frequently misused. Use a tool to find such problems.

- **Hand-analyze code to determine whether there is a vulnerability.** Doing this can be a challenge. (Sometimes it turns out to be better to rewrite any code that shows symptoms of being vulnerable, whether it is or not. This is because it is rare to be able to determine with absolute certainty that a vulnerability exists just from looking at the source code.)

One of the worst things about *architectural* analysis is that there are no tools available to automate the process or to encode some of the necessary expertise. Fortunately, when auditing C and C++ code, there is some help in the area of *implementation* analysis—the tool ITS4. With languages other than C and C++, code review remains a manual process. However, code auditing tends to be easier in other languages (such as Java), because such languages tend not to allow the programmer to add security flaws in the implementation to the same extent that C and C++ do.

ITS4 is a tool distributed by Cigital for statically scanning C and C++ source code for function calls that are known to be "bad."[14] It searches for functions that are commonly involved in implementation flaws. For example, it will look for uses of the gets() function, which is often misused in such a way as to enable buffer overflows.

The goal of ITS4 is to focus the person performing an implementation analysis. Instead of having an analyst search through an entire program, ITS4 provides an analyst with a list of potential trouble spots. Something similar can be done with grep. However, with grep, you need to remember what to look for every single time. ITS4 encodes knowledge about what to look for. In its current form, ITS4 searches for almost 150 vulnerabilities. ITS4 also performs some basic analysis to try to rule out conditions that are obviously not problems. For example, though sprintf() is a frequently misused function, if the format string is constant, and contains no "%s," then it probably isn't worth looking at. ITS4 knows this and thus discounts such calls.

By default, ITS4 orders its findings in order of risk: No Risk, Low Risk, Some Risk, Risky, Very Risky, Urgent. Also by default, the tool doesn't report things of low risk, because such items aren't real problems often enough to be worth looking into. Plus, ITS4 tends to give a lot of output; more than most analysts are willing to wade through!

ITS4 points the analyst to a potentially vulnerable function. It's the responsibility of the auditor to determine whether or not that function is used properly. It would be nice if ITS4 could automate the semantic analysis of source code that a programmer must do. Such tools do exist in the research lab.[15]

Here are some initial conclusions based on experiences using ITS4:

- ITS4 still requires a significant level of expert knowledge. While ITS4 encodes knowledge on vulnerabilities (such that they no longer must be kept in the analyst's head), an expert still does a much better job than a novice at manually performing the static analysis necessary to determine whether a real exploit is possible.
- ITS4 only eliminates one-quarter to one-third of the time it takes to perform a source code analysis, because the manual analysis part is so time consuming. Code inspection takes a long time, even for an expert.
- ITS4 helps significantly with fighting the "get done, go home" effect, because it prioritizes one instance of a problematic function call over another.
- ITS4 has been used to find security problems in real applications.

Architecture Risks (Wall Problems)

Architectural analysis is more important to building secure software than implementation analysis. It is also much harder and much less understood.

---

[14] Viega, John, J.T. Bloch, Tadayoshi Kohno, Gary McGraw. "ITS4: A Static Vulnerability Scanner for C and C++ Code." In *Proceedings of Annual Computer Security Applications Conference*. IEEE Computer Society, 2000.

[15] Weber, Michael, Viren Shah, and Chris Ren. "A Case Study in Detecting Security Vulnerabilities Using Constraint Optimization." In *Proceedings of the 1st IEEE Workshop on Source Code Analysis and Manipulation (SCAM 2001)*. IEEE Computer Society, 2001.

This brief section introduces issues that primarily apply at design time, including use of international standards (such as the Common Criteria) and following general principles for developing secure software systems. Proper software security at an architectural level requires careful integration into a mature software engineering process.

Common Criteria

Some standardized methodologies do exist for building secure systems. Over the last handful of years, several governments around the world including the US government in concert with major user's groups have been working on a standardized system for design and evaluation of security-critical systems. They have produced a system known as the Common Criteria. Theoretically, the Common Criteria can be applied just as easily to software systems as to any other computing system. The compelling idea behind the Common Criteria is to create a security assurance system that can be systematically applied to diverse security-critical systems.

An example can help clarify how the Common Criteria work. Suppose that Wall Street gets together and decides to produce a Protection Profile for the firewalls used to protect the machines holding customers' financial information. Wall Street produces the Protection Profile, and Protection Profile gets evaluated according to the Common Evaluation Methodology to make sure that it is consistent and complete. Once this effort is complete, the Protection Profile is published widely. A vendor produces its version of a firewall that is designed to meet Wall Street's Protection Profile. Once the product is ready to go, it becomes a Target of Evaluation and is sent to an accredited lab for evaluation.

The Protection Profile defines the Security Target for which the Target of Evaluation was designed. The lab applies the Common Evaluation Methodology using the Common Criteria to determine if the Target of Evaluation meets the Security Target. If the Target of Evaluation meets the Security Target, every bit of the lab testing documentation is sent to National Voluntary Laboratory Accreditation Program (NVLAP) for validation. Depending on the relationship between NVLAP and the lab, this could be a rubber stamp or it could involve a careful audit of the evaluation performed by the lab.

If the Target of Evaluation is validated by NVLAP, then the product gets a certificate and is listed alongside other evaluated products. It is then up to the individual investment houses on Wall Street to decide which brands of the firewall they want to purchase from the list of certified Targets of Evaluation that meet the Security Target defined by their chosen Protection Profile.

Though the Common Criteria is certainly a good idea whose time has come, security evaluation is unfortunately not as simple as applying a standard Protection Profile to a given Target of Evaluation. The problem with the Common Criteria is evident right in the name. That is, "common" is often not good enough when it comes to security.

One critical concern is that software quality management decisions, including security decisions, are sensitive to their context. There is no such thing as "secure" against all levels of threat, yet the Common Criteria set out to create a Target of Evaluation based on a standardized Protection Profile. The very real danger is that the Protection Profile will amount to a least common denominator approach to security.

All standards appear to suffer from a similar least common denominator problem. Standards focus on the "what" while underspecifying the "how." The problem is the "how" tends to require deep expertise—a rare commodity. Standards tend to provide a rather low bar (one that some vendors have trouble hurdling nonetheless). An important question to ponder when thinking about standards is how much real security will be demanded of the system.

All in all, the move toward the Common Criteria is probably a good thing. Any approach that can be used to separate the wheat from the chaff in today's over-promising and under-delivering security world is likely to be useful. Just don't count on the Common Criteria to manage all software security risks. Often, much more expertise and insight are required than has been captured in commonly adopted Protection Profiles. Risk management is about more than hurdling the lowest bar.

Checklists and Guidelines

The biggest open research issue in software security is that there is currently no good standard language of discourse for software design. Lacking the ability to specify an application formally, tools and technologies for automated analysis of software security at the architectural level lag significantly behind implementation tools such as ITS4. Until the research community makes more progress on this issue, architectural risk analysis will remain a high-expertise practice.

One way of capturing software security design advice is to create general guidelines. John Viega and I present a list of ten such guidelines in *Building Secure Software*. I include the guidelines here without commentary.[16] Note that the goal of these principles is to identify and highlight the most important objectives software security practitioners should keep in mind when designing and building a secure system. Following these principles should help developers and architects who are not necessarily versed in security to avoid a number of common security problems.

- Secure the weakest link.
- Practice defense in depth.
- Fail securely.
- Follow the principle of least privilege.
- Compartmentalize.
- Keep it simple.
- Promote privacy.
- Remember that hiding secrets is hard.
- Be reluctant to trust.
- Use your community resources.

Some caveats are in order. No list of principles like the ones above is ever perfect. There is no guarantee that following these principles will result in secure software. Not only do the principles present an incomplete picture, but also they sometimes even conflict with each other. As with any complex set of principles, there are often subtle tradeoffs involved, and application of these 10 principles must be sensitive to context. A mature software risk management approach provides the sort of data required to apply the principles intelligently.

A Call To Arms

Computer security is a vast topic that is becoming more important because the world is becoming highly interconnected, with networks being used to carry out critical transactions. The environment in which machines must survive has changed radically since the popularization of the Internet. The root of most security problems is software that fails in unexpected ways. Though software security as a field has much maturing to do, it has much to offer to those practitioners interested in striking at the heart of security problems. This paper merely scratches the surface of the problem.

Good software security practices can help ensure that software behaves properly. Safety-critical and high-assurance system designers have always taken great pains to analyze and track software behavior. Security-critical system designers must follow suit. We can avoid the band-aid–like penetrate-and-patch approach to security only by

[16] For more information, see: Viega, John and Gary McGraw. *Building Secure Software.* Addison-Wesley, 2001.

considering security as a crucial system property. This requires integrating software security into the software engineering process.

Software practitioners are only now becoming aware of software security as an issue. Plenty of work remains to be done. The most pressing current need involves understanding architectural-level risks and flaws. Today's apprenticeship-based approach does not scale well, especially considering the growth of the software security problem.

Gary McGraw, Ph.D., is the Chief Technology Officer at Cigital, Inc. He can be reached at gem@cigital.com.

This paper was drawn from Building Secure Software (Addison-Wesley, 2001), which was co-authored by McGraw and John Viega.

This paper appeared as an article titled "On Bricks and Walls: Why Building Secure Software is Hard," in the May 2002 issue (Vol. 15, No. 5) of the Cutter IT Journal (www.cutter.com). It is reprinted here with their permission. Dr. McGraw wishes to thank Greg Morrisett, Fred Schneider and other members of the Infosec Research Council's study group on malicious code for their insight.

Cigital, Inc.
21351 Ridgetop Circle
Suite 400
Dulles, Virginia
20166

T 800 824 0022
703 404 9293
F 703 404 9295
E info@cigital.com

www.cigital.com