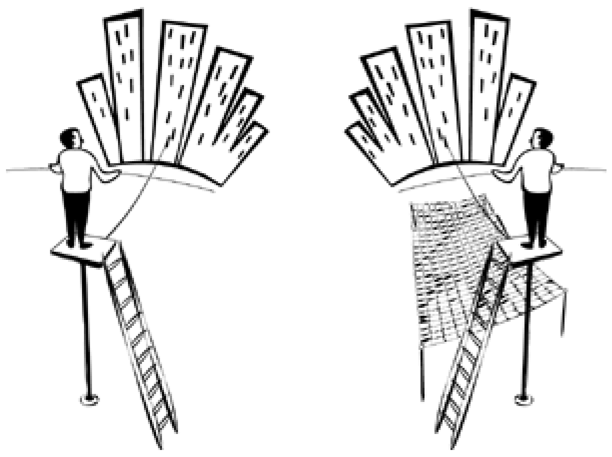
# Chapter 1. Why Is Security a Software Issue?[*]

## 1.1. Introduction

Software is everywhere. It runs your car. It controls your cell phone. It's how you access your bank's financial services; how you receive electricity, water, and natural gas; and how you fly from coast to coast [McGraw 2006]. Whether we recognize it or not, we all rely on complex, interconnected, software-intensive information systems that use the Internet as their means for communicating and transporting information.

Building, deploying, operating, and using software that has not been developed with security in mind can be high risk—like walking a high wire without a net (Figure 1–1). The degree of risk can be compared to the distance you can fall and the potential impact (no pun intended).

**Figure 1-1.** *Developing software without security in mind is like walking a high wire without a net*



This chapter discusses why security is increasingly a software problem. It defines the dimensions of software assurance and software security. It identifies threats that target most software and the shortcomings of the software development process that can render software vulnerable to those threats. It closes by introducing some pragmatic solutions that are expanded in the chapters to follow. This entire chapter is relevant for executives (E), project managers (M), and technical leaders (L).

## 1.2. The Problem

Organizations increasingly store, process, and transmit their most sensitive information using software-intensive systems that are directly connected to the Internet. Private citizens' financial transactions are exposed via the Internet by software used to shop, bank, pay taxes, buy insurance, invest, register children for school, and join various organizations and social networks. The increased exposure that comes with global connectivity has made sensitive information and the software systems that handle it more vulnerable to unintentional and unauthorized use. In short, software-intensive systems and other software-enabled capabilities have provided more open, widespread access to sensitive information—including personal identities—than ever before.

Concurrently, the era of information warfare **[Denning 1998]**, cyberterrorism, and computer crime is well under way. Terrorists, organized crime, and other criminals are targeting the entire gamut of software-intensive systems and, through human ingenuity gone awry, are being successful at gaining entry to these systems. Most such systems are not attack resistant or attack resilient enough to withstand them.

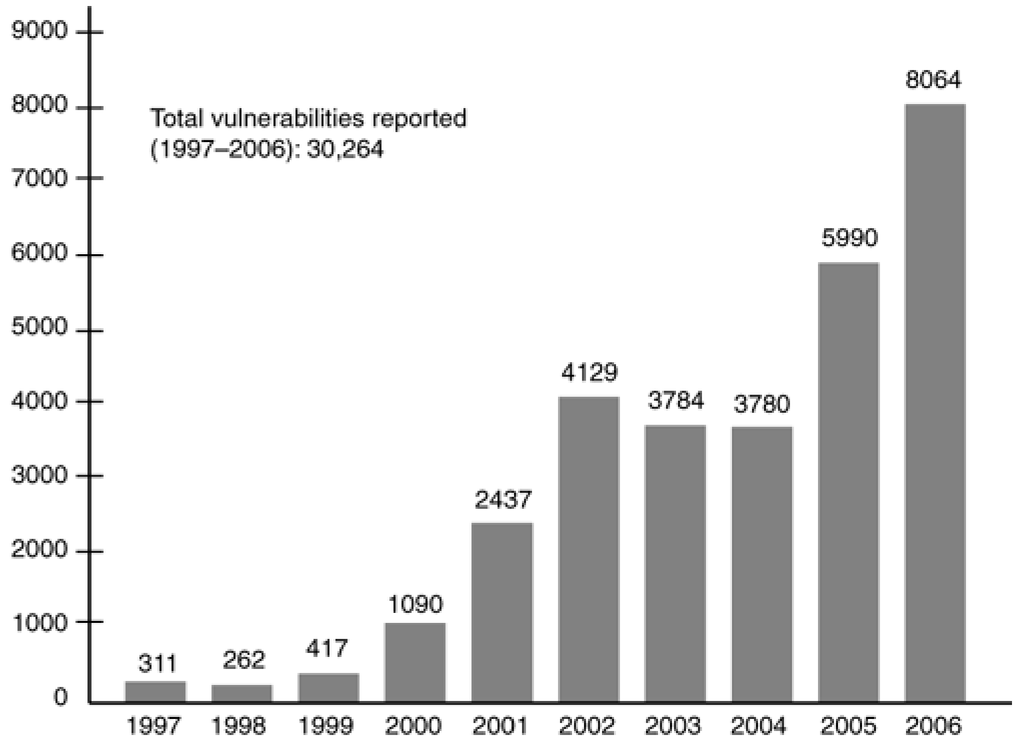
In a report to the U.S. president titled *Cyber Security: A Crisis of Prioritization* **[PITAC 2005]**, the President's Information Technology Advisory Committee summed up the problem of nonsecure software as follows:

> Software development is not yet a science or a rigorous discipline, and the development process by and large is not controlled to minimize the vulnerabilities that attackers exploit. Today, as with cancer, vulnerable software can be invaded and modified to cause damage to previously healthy software, and infected software can replicate itself and be carried across networks to cause damage in other systems. Like cancer, these damaging processes may be invisible to the lay person even though experts recognize that their threat is growing.

Software defects with security ramifications—including coding bugs such as buffer overflows and design flaws such as inconsistent error handling—are ubiquitous. Malicious intruders, and the malicious code and botnets[1] they use to obtain unauthorized access and launch attacks, can compromise systems by taking advantage of software defects. Internet-enabled software applications are a commonly exploited target, with software's increasing complexity and extensibility making software security even more challenging **[Hoglund 2004]**.

The security of computer systems and networks has become increasingly limited by the quality and security of their software. Security defects and vulnerabilities in software are commonplace and can pose serious risks when exploited by malicious attacks. Over the past six years, this problem has grown significantly. **Figure 1–2** shows the number of vulnerabilities reported to CERT from 1997 through 2006. Given this trend, "[T]here is a clear and pressing need to change the way we (project managers and software engineers) approach computer security and to develop a disciplined approach to software security" **[McGraw 2006]**.

**Figure 1-2.** *Vulnerabilities reported to CERT*



In Deloitte's *2007 Global Security Survey*, 87 percent of survey respondents cited poor software development quality as a top threat in the next

12 months. “Application security means ensuring that there is secure code, integrated at the development stage, to prevent potential vulnerabilities and that steps such as vulnerability testing, application scanning, and penetration testing are part of an organization’s software development life cycle [SDLC]” [Deloitte 2007].

The growing Internet connectivity of computers and networks and the corresponding user dependence on network-enabled services (such as email and Web-based transactions) have increased the number and sophistication of attack methods, as well as the ease with which an attack can be launched. This trend puts software at greater risk. Another risk area affecting software security is the degree to which systems accept updates and extensions for evolving capabilities. Extensible systems are attractive because they provide for the addition of new features and services, but each new extension adds new capabilities, new interfaces, and thus new risks. A final software security risk area is the unbridled growth in the size and complexity of software systems (such as the Microsoft Windows operating system). The unfortunate reality is that in general more lines of code produce more bugs and vulnerabilities [McGraw 2006].

### 1.2.1. System Complexity: The Context within Which Software Lives

Building a trustworthy software system can no longer be predicated on constructing and assembling discrete, isolated pieces that address static requirements within planned cost and schedule. Each new or updated software component joins an existing operational environment and must merge with that legacy to form an operational whole. Bolting new systems onto old systems and Web-enabling old systems creates systems of systems that are fraught with vulnerabilities. With the expanding scope and scale of systems, project managers need to reconsider a number of development assumptions that are generally applied to software security:

- Instead of centralized control, which was the norm for large stand-alone systems, project managers have to consider multiple and often independent control points for systems and systems of systems.

• Increased integration among systems has reduced the capability to make wide-scale changes quickly. In addition, for independently managed systems, upgrades are not necessarily synchronized. Project managers need to maintain operational capabilities with appropriate security as services are upgraded and new services are added.

• With the integration among independently developed and operated systems, project managers have to contend with a heterogeneous collection of components, multiple implementations of common interfaces, and inconsistencies among security policies.

• With the mismatches and errors introduced by independently developed and managed systems, failure in some form is more likely to be the norm than the exception and so further complicates meeting security requirements.

There are no known solutions for ensuring a specified level or degree of software security for complex systems and systems of systems, assuming these could even be defined. This said, **Chapter 6**, Security and Complexity: System Assembly Challenges, elaborates on these points and provides useful guidelines for project managers to consider in addressing the implications.

## 1.3. Software Assurance and Software Security

The increasing dependence on software to get critical jobs done means that software's value no longer lies solely in its ability to enhance or sustain productivity and efficiency. Instead, its value also derives from its ability to continue operating dependably even in the face of events that threaten it. The ability to trust that software will remain dependable under all circumstances, with a justified level of confidence, is the objective of software assurance.

Software assurance has become critical because dramatic increases in business and mission risks are now known to be attributable to exploitable software **[DHS 2003]**. The growing extent of the resulting risk exposure is rarely understood, as evidenced by these facts:

- Software is the weakest link in the successful execution of interdependent systems and software applications.

- Software size and complexity obscure intent and preclude exhaustive testing.

- Outsourcing and the use of unvetted software supply-chain components increase risk exposure.

- The sophistication and increasingly more stealthy nature of attacks facilitates exploitation.

- Reuse of legacy software with other applications introduces unintended consequences, increasing the number of vulnerable targets.

- Business leaders are unwilling to make risk-appropriate investments in software security.

According to the U.S. Committee on National Security Systems' "National Information Assurance (IA) Glossary" **[CNSS 2006]**, software assurance is

> the level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at any time during its life cycle, and that the software functions in the intended manner.

Software assurance includes the disciplines of software reliability[2] (also known as software fault tolerance), software safety,[3] and software security. The focus of *Software Security Engineering: A Guide for Project Managers* is on the third of these, software security, which is the ability of software to resist, tolerate, and recover from events that intentionally threaten its dependability. The main objective of software security is to build more-robust, higher-quality, defect-free software that continues to function correctly under malicious attack **[McGraw 2006]**.

Software security matters because so many critical functions are completely dependent on software. This makes software a very high-value target for attackers, whose motives may be malicious, criminal, adversarial, competitive, or terrorist in nature. What makes it so easy for attackers to target software is the virtually guaranteed presence of known vulnerabilities with known attack methods, which can be exploited to violate one or more of the software's security properties or to force the software into an insecure state. Secure software remains dependable (i.e., correct and predictable) despite intentional efforts to compromise that dependability.

The objective of software security is to field software-based systems that satisfy the following criteria:

• The system is as vulnerability and defect free as possible.

• The system limits the damage resulting from any failures caused by attack-triggered faults, ensuring that the effects of any attack are not propagated, and it recovers as quickly as possible from those failures.

• The system continues operating correctly in the presence of most attacks by either **resisting** the exploitation of weaknesses in the software by the attacker or **tolerating** the failures that result from such exploits.

Software that has been developed with security in mind generally reflects the following properties throughout its development life cycle:

• **Predictable execution**. There is justifiable confidence that the software, when executed, functions as intended. The ability of malicious input to alter the execution or outcome in a way favorable to the attacker is significantly reduced or eliminated.

• **Trustworthiness**. The number of exploitable vulnerabilities is intentionally minimized to the greatest extent possible. The goal is no exploitable vulnerabilities.

• **Conformance**. Planned, systematic, and multidisciplinary activities ensure that software components, products, and systems conform to requirements and applicable standards and procedures for specified uses.

These objectives and properties must be interpreted and constrained based on the practical realities that you face, such as what constitutes an adequate level of security, what is most critical to address, and which actions fit within the project's cost and schedule. These are risk management decisions.

In addition to predictable execution, trustworthiness, and conformance, secure software and systems should be as attack resistant, attack tolerant, and attack resilient as possible. To ensure that these criteria are satisfied, software engineers should design software components and systems to recognize both legitimate inputs and known attack patterns in the data or signals they receive from external entities (humans or processes) and reflect this recognition in the developed software to the extent possible and practical.

To achieve attack resilience, a software system should be able to recover from failures that result from successful attacks by resuming operation at or above some predefined minimum acceptable level of service in a timely manner. The system must eventually recover full service at the specified level of performance. These qualities and properties, as well as attack patterns, are described in more detail in **Chapter 2**, What Makes Software Secure?

### 1.3.1. The Role of Processes and Practices in Software Security

A number of factors influence how likely software is to be secure. For instance, software vulnerabilities can originate in the processes and practices used in its creation. These sources include the decisions made by software engineers, the flaws they introduce in specification and design, and the faults and other defects they include in developed code, inadvertently or intentionally. Other factors may include the choice of programming languages and development tools used to develop the software, and the configuration and behavior of software components in their development and operational environments. It is increasingly observed, however, that *the most critical difference between secure software and insecure software lies in the nature of the processes and practices used to specify, design, and develop the software* **[Goertzel 2006]**.

The return on investment when security analysis and secure engineering practices are introduced early in the development cycle ranges from 12 percent to 21 percent, with the highest rate of return occurring when the analysis is performed during application design [Berinato 2002; Soo Hoo 2001]. This return on investment occurs because there are fewer security defects in the released product and hence reduced labor costs for fixing defects that are discovered later.

A project that adopts a security-enhanced software development process is adopting a set of practices (such as those described in this book's chapters) that initially should reduce the number of exploitable faults and weaknesses. Over time, as these practices become more codified, they should decrease the likelihood that such vulnerabilities are introduced into the software in the first place. More and more, research results and real-world experiences indicate that *correcting potential vulnerabilities as early as possible in the software development life cycle, mainly through the adoption of security-enhanced processes and practices, is far more cost-effective* than the currently pervasive approach of developing and releasing frequent patches to operational software [Goertzel 2006].

## 1.4. Threats to Software Security

In information security, the threat—the source of danger—is often a person intending to do harm, using one or more malicious software agents. Software is subject to two general categories of threats:

• *Threats during development* (mainly insider threats). A software engineer can sabotage the software at any point in its development life cycle through intentional exclusions from, inclusions in, or modifications of the requirements specification, the threat models, the design documents, the source code, the assembly and integration framework, the test cases and test results, or the installation and configuration instructions and tools. The secure development practices described in this book are, in part, designed to help reduce the exposure of software to insider threats during its development process. For more information on this aspect, see "Insider Threats in the SDLC" [Cappelli 2006].

• *Threats during operation* (both insider and external threats). Any software system that runs on a network-connected platform is likely to have its vulnerabilities exposed to attackers during its operation. Attacks may take advantage of publicly known but unpatched vulnerabilities, leading to memory corruption, execution of arbitrary exploit scripts, remote code execution, and buffer overflows. Software flaws can be exploited to install spyware, adware, and other malware on users' systems that can lie dormant until it is triggered to execute.[4]

Weaknesses that are most likely to be targeted are those found in the software components' external interfaces, because those interfaces provide the attacker with a direct communication path to the software's vulnerabilities. A number of well-known attacks target software that incorporates interfaces, protocols, design features, or development faults that are well understood and widely publicized as harboring inherent weaknesses. That software includes Web applications (including browser and server components), Web services, database management systems, and operating systems. Misuse (or abuse) cases can help project managers and software engineers see their software from the perspective of an attacker by anticipating and defining unexpected or abnormal behavior through which a software feature could be unintentionally misused or intentionally abused **[Hope 2004]**. (See **Section 3.2**.)

Today, most project and IT managers responsible for system operation respond to the increasing number of Internet-based attacks by relying on operational controls at the operating system, network, and database or Web server levels while failing to directly address the insecurity of the application-level software that is being compromised. This approach has two critical shortcomings:

1. The security of the application depends completely on the robustness of operational protections that surround it.

2. Many of the software-based protection mechanisms (controls) can easily be misconfigured or misapplied. Also, they are as likely to contain exploitable vulnerabilities as the application software they are (supposedly) protecting.

The wide publicity about the literally thousands of successful attacks on software accessible from the Internet has merely made the attacker's job easier. Attackers can study numerous reports of security vulnerabilities in a wide range of commercial and open-source software programs and access publicly available exploit scripts. More experienced attackers often develop (and share) sophisticated, targeted attacks that exploit specific vulnerabilities. In addition, the nature of the risks is changing more rapidly than the software can be adapted to counteract those risks, regardless of the software development process and practices used. To be 100 percent effective, defenders must anticipate *all* possible vulnerabilities, while attackers need find only *one* to carry out their attack.

## 1.5. Sources of Software Insecurity

Most commercial and open-source applications, middleware systems, and operating systems are extremely large and complex. In normal execution, these systems can transition through a vast number of different states. These characteristics make it particularly difficult to develop and operate software that is consistently correct, let alone consistently secure. The unavoidable presence of security threats and risks means that project managers and software engineers need to pay attention to software security even if explicit requirements for it have not been captured in the software's specification.

A large percentage of security weaknesses in software could be avoided if project managers and software engineers were routinely trained in how to address those weaknesses systematically and consistently. Unfortunately, these personnel are seldom taught how to design and develop secure applications and conduct quality assurance to test for insecure coding errors and the use of poor development techniques. They do not generally understand which practices are effective in recognizing and removing faults and defects or in handling vulnerabilities when software is exploited by attackers. They are often unfamiliar with the security implications of certain software requirements (or their absence). Likewise, they rarely learn about the security implications of how software is architected, designed, developed, deployed, and operated. The absence of this knowledge means that security requirements are likely to be inadequate and that the resulting software is likely to deviate from specified (and unspecified) security requirements. In addition, this lack of knowledge prevents the manager and engineer from recognizing and understanding how mistakes can manifest as exploitable weaknesses and vulnerabilities in the software when it becomes operational.

Software—especially networked, application-level software—is most often compromised by exploiting weaknesses that result from the following sources:

• Complexities, inadequacies, and/or changes in the software's processing model (e.g., a Web- or service-oriented architecture model).

• Incorrect assumptions by the engineer, including assumptions about the capabilities, outputs, and behavioral states of the software's execution environment or about expected inputs from external entities (users, software processes).

• Flawed specification or design, or defective implementation of

– The software's interfaces with external entities. Development mistakes of this type include inadequate (or nonexistent) input validation, error handling, and exception handling.

– The components of the software's execution environment (from middleware-level and operating-system-level to firmware- and hardware-level components).

• Unintended interactions between software components, including those provided by a third party.

Mistakes are unavoidable. Even if they are avoided during requirements engineering and design (e.g., through the use of formal methods) and development (e.g., through comprehensive code reviews and extensive testing), vulnerabilities may still be introduced into software during its assembly, integration, deployment, and operation. No matter how faithfully a security-enhanced life cycle is followed, as long as software continues to grow in size and complexity, some number of exploitable faults and other weaknesses are sure to exist.

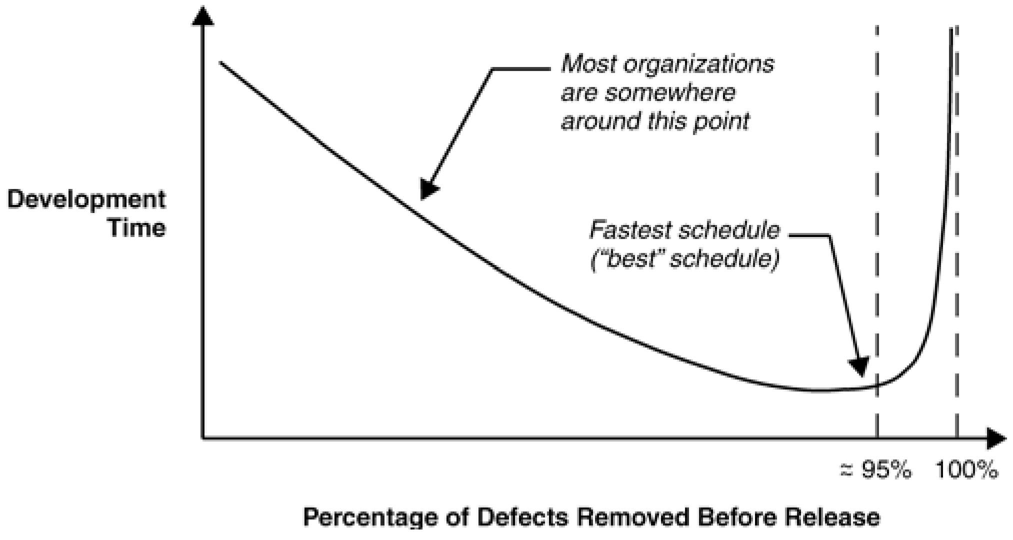
In addition to the issues identified here, Chapter 2, What Makes Software Secure?, discusses a range of principles and practices, the absence of which contribute to software insecurity.

## 1.6. The Benefits of Detecting Software Security Defects Early[5]

Limited data is available that discusses the return on investment (ROI) of reducing security flaws in source code (refer to Section 1.6.1 for more on this subject). Nevertheless, a number of studies have shown that significant cost benefits are realized through improvements to reduce software defects (including security flaws) throughout the SDLC [Goldenson 2003]. The general software quality case is made in this section, including reasonable arguments for extending this case to include software security defects.

Proactively tackling software security is often under-budgeted and dismissed as a luxury. In an attempt to shorten development schedules or decrease costs, software project managers often reduce the time spent on secure software practices during requirements analysis and design. In addition, they often try to compress the testing schedule or reduce the level of effort. Skimping on software quality[6] is one of the worst decisions an organization that wants to maximize development speed can make; higher quality (in the form of lower defect rates) and reduced development time go hand in hand. Figure 1–3 illustrates the relationship between defect rate and development time.

**Figure 1-3.** *Relationship between defect rate and development time*



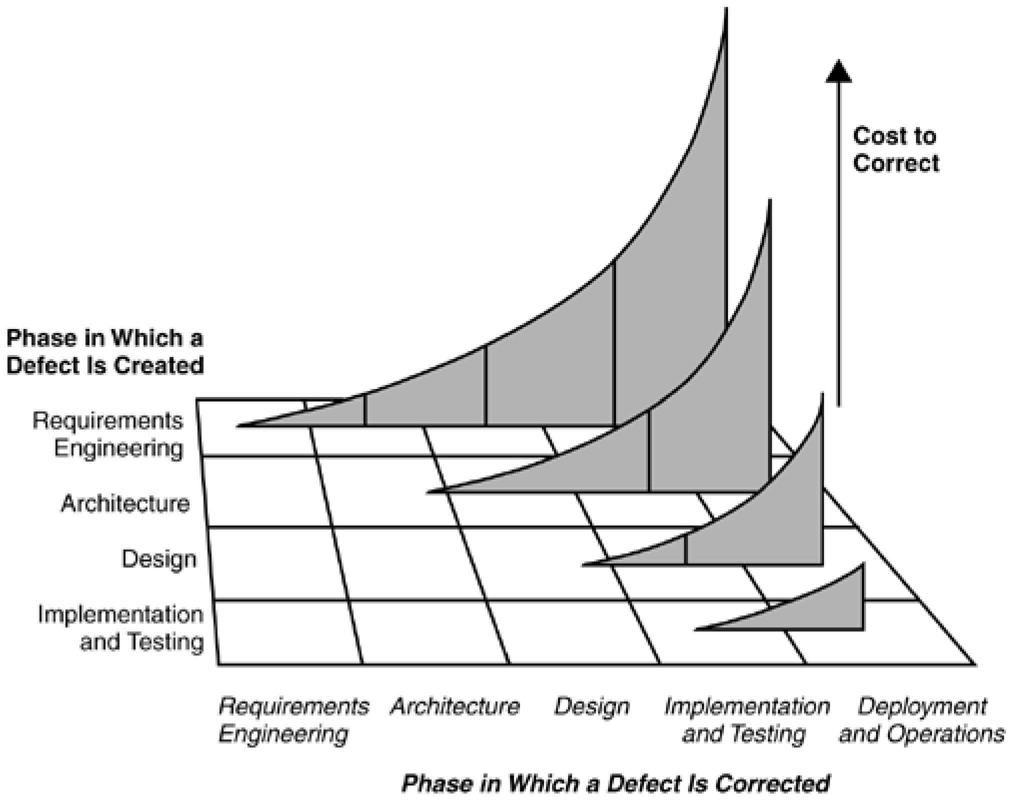
Projects that achieve lower defect rates typically have shorter schedules. But many organizations currently develop software with defect levels that result in longer schedules than necessary. In the 1970s, studies performed by IBM demonstrated that software products with lower defect counts also had shorter development schedules [Jones 1991]. After surveying more than 4000 software projects, Capers Jones [1994] reported that poor quality was one of the most common reasons for schedule overruns. He also reported that poor quality was a significant factor in approximately 50 percent of all canceled projects. A Software Engineering Institute survey found that more than 60 percent of organizations assessed suffered from inadequate quality assurance [Kitson 1993]. On the curve in Figure 1–3, the organizations that experienced higher numbers of defects are to the left of the "95 percent defect removal" line.

The "95 percent defect removal" line is significant because that level of prerelease defect removal appears to be the point at which projects achieve the shortest schedules for the least effort and with the highest levels of user satisfaction [Jones 1991]. If more than 5 percent of defects are found after a product has been released, then the product is vulnerable to the problems associated with low quality, and the organization takes longer to develop its software than necessary. Projects that are completed with undue haste are particularly vulnerable to short-changing quality assurance at the individual developer level. Any developer who has been pushed to satisfy a specific deadline or ship a product quickly knows how much pressure there can be to cut corners because "we're only three weeks from the deadline." As many as four times the average number of defects are reported for released software products that were developed under excessive schedule pressure. Developers participating in projects that are in schedule trouble often become obsessed with working harder rather than working smarter, which gets them into even deeper schedule trouble.

One aspect of quality assurance that is particularly relevant during rapid development is the presence of error-prone modules—that is, modules that are responsible for a disproportionate number of defects. Barry Boehm reported that 20 percent of the modules in a program are typically responsible for 80 percent of the errors [Boehm 1987]. On its IMS project, IBM found that 57 percent of the errors occurred in 7 percent of the modules [Jones 1991]. Modules with such high defect rates are more expensive and time-consuming to deliver than less error-prone modules. Normal modules cost about $500 to $1000 per function point to develop, whereas error-prone modules cost about $2000 to $4000 per function point to develop [Jones 1994]. Error-prone modules tend to be more complex, less structured, and significantly larger than other modules. They often are developed under excessive schedule pressure and are not fully tested. If development speed is important, then identification and redesign of error-prone modules should be a high priority.

If an organization can prevent defects or detect and remove them early, it can realize significant cost and schedule benefits. Studies have found that reworking defective requirements, design, and code typically accounts for 40 to 50 percent of the total cost of software development [Jones 1986b]. As a rule of thumb, every hour an organization spends on defect prevention reduces repair time for a system in production by three to ten hours. In the worst case, reworking a software requirements problem once the software is in operation typically costs 50 to 200 times what it would take to rework the same problem during the requirements phase [Boehm 1988]. It is easy to understand why this phenomenon occurs. For example, a one-sentence requirement could expand into 5 pages of design diagrams, then into 500 lines of code, then into 15 pages of user documentation and a few dozen test cases. It is cheaper to correct an error in that one-sentence requirement at the time requirements are specified (assuming the error can be identified and corrected) than it is after design, code, user documentation, and test cases have been written. Figure 1–4 illustrates that the longer defects persist, the more expensive they are to correct.

**Figure 1-4.** *Cost of correcting defects by life-cycle phase*



The savings potential from early defect detection is significant: Approximately 60 percent of all defects usually exist by design time [Gilb 1988]. A decision early in a project to exclude defect detection amounts to a decision to postpone defect detection and correction until later in the project, when defects become much more expensive and time-consuming to address. That is not a rational decision when time and development dollars are at a premium. According to software quality assurance empirical research, $1 required to resolve an issue during the design phase grows into $60 to $100 required to resolve the same issue after the application has shipped [Soo Hoo 2001].

When a software product has too many defects, including security flaws, vulnerabilities, and bugs, software engineers can end up spending more time correcting these problems than they spent on developing the software in the first place. Project managers can achieve the shortest possible schedules with a higher-quality product by addressing security throughout the SDLC, especially during the early phases, to increase the likelihood that software is more secure the first time.

### 1.6.1. Making the Business Case for Software Security: Current State[7]

As software project managers and developers, we know that when we want to introduce new approaches in our development processes, we have to make a cost–benefit argument to executive management to convince them that this move offers a business or strategic return on investment. Executives are not interested in investing in new technical approaches simply because they are innovative or exciting. For profit-making organizations, we need to make a case that demonstrates we will improve market share, profit, or other business elements. For other types of organizations, we need to show that we will improve our software in a way that is important—in a way that adds to the organization's prestige, that ensures the safety of troops in the battlefield, and so on.

In the area of software security, we have started to see some evidence of successful ROI or economic arguments for security administrative operations, such as maintaining current levels of patches, establishing organizational entities such as computer security incident response teams (CSIRTs) to support security investment, and so on **[Blum 2006, Gordon 2006, Huang 2006, Nagaratnam 2005]**. In their article "Tangible ROI through Secure Software Engineering," Kevin Soo Hoo and his colleagues at @stake state the following:

> Findings indicate that significant cost savings and other advantages are achieved when security analysis and secure engineering practices are introduced early in the development cycle. The return on investment ranges from 12 percent to 21 percent, with the highest rate of return occurring when analysis is performed during application design.
> Since nearly three-quarters of security-related defects are design issues that could be resolved inexpensively during the early stages, a significant opportunity for cost savings exists when secure software engineering principles are applied during design.

However, except for a few studies [Berinato 2002; Soo Hoo 2001], we have seen little evidence presented to support the idea that investment during software development in software security will result in commensurate benefits across the entire life cycle.

Results of the Hoover project [Jaquith 2002] provide some case study data that supports the ROI argument for investment in software security early in software development. In his article "The Security of Applications: Not All Are Created Equal," Jaquith says that "the best-designed e-business applications have one-quarter as many security defects as the worst. By making the right investments in application security, companies can out-perform their peers—and reduce risk by 80 percent."

In their article "Impact of Software Vulnerability Announcements on the Market Value of Software Vendors: An Empirical Investigation," the authors state that "On average, a vendor loses around 0.6 percent value in stock price when a vulnerability is reported. This is equivalent to a loss in market capitalization values of $0.86 billion per vulnerability announcement." The purpose of the study described in this article is "to measure vendors' incentive to develop secure software" [Telang 2004].

We believe that in the future Microsoft may well publish data reflecting the results of using its Security Development Lifecycle [Howard 2006, 2007]. We would also refer readers to the business context discussion in chapter 2 and the business climate discussion in chapter 10 of McGraw's recent book [McGraw 2006] for ideas.

## 1.7. Managing Secure Software Development

The previous section put forth useful arguments and identified emerging evidence for the value of detecting software security defects as early in the SDLC as possible. We now turn our attention to some of the key project management and software engineering practices to aid in accomplishing this goal. These are introduced here and covered in greater detail in subsequent chapters of this book.

### 1.7.1. Which Security Strategy Questions Should I Ask?

Achieving an adequate level of software security means more than complying with regulations or implementing commonly accepted best practices. You and your organization must determine your own definition of "adequate." The range of actions you must take to reduce software security risk to an acceptable level depends on what the product, service, or system you are building needs to protect and what it needs to prevent and manage.

Consider the following questions from an enterprise perspective. Answers to these questions aid in understanding security risks to achieving project goals and objectives.

• What is the value we must protect?

• To sustain this value, which assets must be protected? Why must they be protected? What happens if they're not protected?

• What potential adverse conditions and consequences must be prevented and managed? At what cost? How much disruption can we stand before we take action?

• How do we determine and effectively manage residual risk (the risk remaining after mitigation actions are taken)?

• How do we integrate our answers to these questions into an effective, implementable, enforceable security strategy and plan?

Clearly, an organization cannot protect and prevent everything. Interaction with key stakeholders is essential to determine the project's risk tolerance and its resilience if the risk is realized. In effect, security in the context of risk management involves determining what could go wrong, how likely such events are to occur, what impact they will have if they do occur, and which actions might mitigate or minimize both the likelihood and the impact of each event to an acceptable level.

The answers to these questions can help you determine how much to invest, where to invest, and how fast to invest in an effort to mitigate software security risk. In the absence of answers to these questions (and a process for periodically reviewing and updating them), you (and your business leaders) will find it difficult to define and deploy an effective security strategy and, therefore, may be unable to effectively govern and manage enterprise, information, and software security.[8]

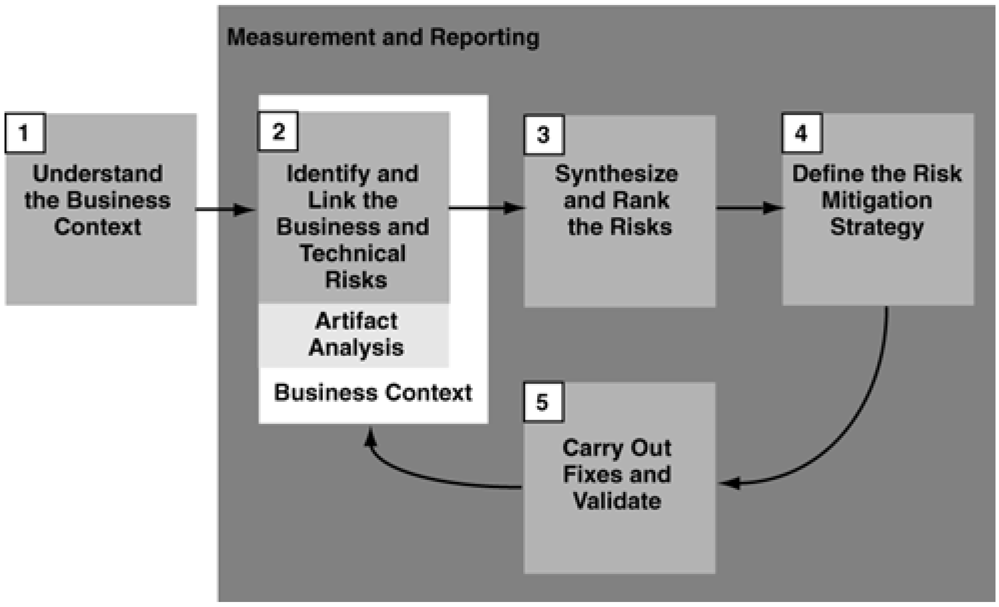
The next section presents a practical way to incorporate a reasoned security strategy into your development process. The framework described is a condensed version of the Cigital Risk Management Framework, a mature process that has been applied in the field for almost ten years. It is designed to manage software-induced business risks. Through the application of five simple activities (further detailed in **Section 7.4.2**), analysts can use their own technical expertise, relevant tools, and technologies to carry out a reasonable risk management approach.

### 1.7.2. A Risk Management Framework for Software Security[9]

A necessary part of any approach to ensuring adequate software security is the definition and use of a continuous risk management process. Software security risk includes risks found in the outputs and results produced by each life-cycle phase during assurance activities, risks introduced by insufficient processes, and personnel-related risks. The risk management framework (RMF) introduced here and expanded in **Chapter 7** can be used to implement a high-level, consistent, iterative risk analysis that is deeply integrated throughout the SDLC.

**Figure 1–5** shows the RMF as a closed-loop process with five activity stages. Throughout the application of the RMF, measurement and reporting activities occur. These activities focus on tracking, displaying, and understanding progress regarding software risk.

**Figure 1-5.** *A software security risk management framework*


Measurement and Reporting
1
Understand the Business Context
2
Identify and Link the Business and Technical Risks
Artifact Analysis
Business Context
3
Synthesize and Rank the Risks
4
Define the Risk Mitigation Strategy
5
Carry Out Fixes and Validate

### 1.7.3. Software Security Practices in the Development Life Cycle

Managers and software engineers should treat all software faults and weaknesses as potentially exploitable. Reducing exploitable weaknesses begins with the specification of software security requirements, along with considering requirements that may have been overlooked (see **Chapter 3**, Requirements Engineering for Secure Software). Software that includes security requirements (such as security constraints on process behaviors and the handling of inputs, and resistance to and tolerance of intentional failures) is more likely to be engineered to remain dependable and secure in the face of an attack. In addition, exercising misuse/abuse cases that anticipate abnormal and unexpected behavior can aid in gaining a better understanding of how to create secure and reliable software (see **Section 3.2**).

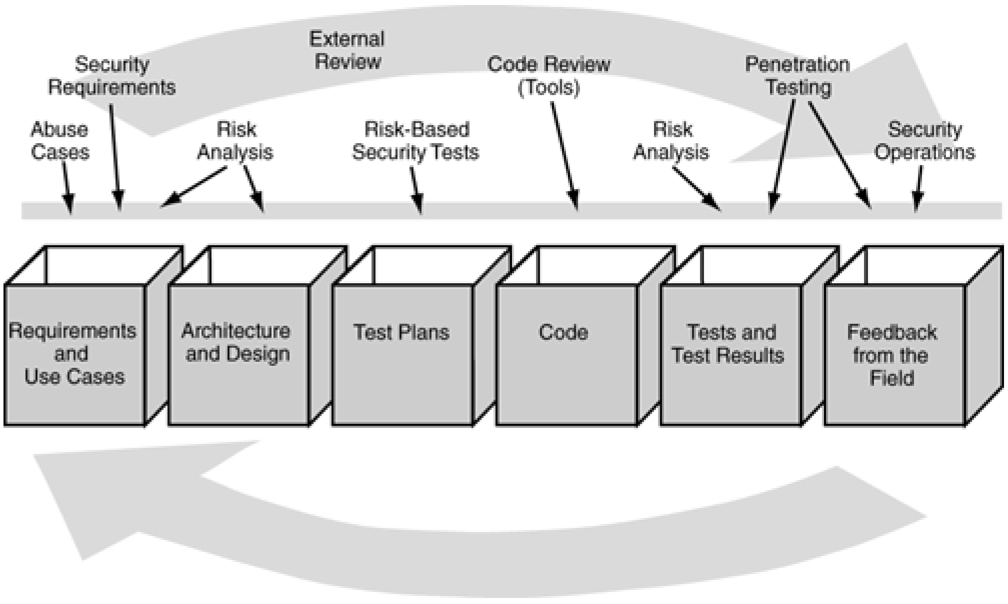
Developing software from the beginning with security in mind is more effective by orders of magnitude than trying to validate, through testing and verification, that the software is secure. For example, attempting to demonstrate that an implemented system will *never* accept an unsafe input (that is, proving a negative) is impossible. You can prove, however, using approaches such as formal methods and function abstraction, that the software you are designing will never accept an unsafe input. In addition, it is easier to design and implement the system so that input validation routines check *every* input that the software receives against a set of predefined constraints. Testing the input validation function to demonstrate that it is consistently invoked and correctly performed every time input enters the system is then included in the system's functional testing.

Analysis and modeling can serve to better protect your software against the more subtle, complex attack patterns involving externally forced sequences of interactions among components or processes that were never intended to interact during normal software execution. Analysis and modeling can help you determine how to strengthen the security of the software's interfaces with external entities and increase its tolerance of all faults. Methods in support of analysis and modeling during each life-cycle phase such as attack patterns, misuse and abuse cases, and architectural risk analysis are described in subsequent chapters of this book.

If your development organization's time and resource constraints prevent secure development practices from being applied to the entire software system, you can use the results of a business-driven risk assessment (as introduced earlier in this chapter and further detailed in **Section 7.4.2**) to determine which software components should be given highest priority.

A security-enhanced life-cycle process should (at least to some extent) compensate for security inadequacies in the software's requirements by adding risk-driven practices and checks for the adequacy of those practices during all software life-cycle phases. **Figure 1–6** depicts one example of how to incorporate security into the SDLC using the concept of touchpoints **[McGraw 2006**; **Taylor 2005]**. Software security best practices (touchpoints shown as arrows) are applied to a set of software artifacts (the boxes) that are created during the software development process. The intent of this particular approach is that it is process neutral and, therefore, can be used with a wide range of software development processes (e.g., waterfall, agile, spiral, Capability Maturity Model Integration [CMMI]).

**Figure 1-6.** *Software development life cycle with defined security touchpoints* **[McGraw 2006]**



Security controls in the software's life cycle should not be limited to the requirements, design, code, and test phases. It is important to continue performing code reviews, security tests, strict configuration control, and quality assurance during deployment and operations to ensure that updates and patches do not add security weaknesses or malicious logic to production software.[10] Additional considerations for project managers, including the effect of software security requirements on project scope, project plans, estimating resources, and product and process measures, are detailed in Chapter 7.

## 1.8. Summary

It is a fact of life that software faults, defects, and other weaknesses affect the ability of software to function securely. These vulnerabilities can be exploited to violate software's security properties and force the software into an insecure, exploitable state. Dealing with this possibility is a particularly daunting challenge given the ubiquitous connectivity and explosive growth and complexity of software-based systems.

Adopting a security-enhanced software development process that includes secure development practices will reduce the number of exploitable faults and weaknesses in the deployed software. Correcting potential vulnerabilities as early as possible in the SDLC, mainly through the adoption of security-enhanced processes and practices, is far more cost-effective than attempting to diagnose and correct such problems after the system goes into production. It just makes good sense.

Thus, the goals of using secure software practices are as follows:

• Exploitable faults and other weaknesses are eliminated to the greatest extent possible by well-intentioned engineers.

• The likelihood is greatly reduced or eliminated that malicious engineers can intentionally implant exploitable faults and weaknesses, malicious logic, or backdoors into the software.

• The software is attack resistant, attack tolerant, and attack resilient to the extent possible and practical in support of fulfilling the organization's mission.

To ensure that software and systems meet their security requirements throughout the development life cycle, review, select, and tailor guidance from this book, the BSI Web site, and the sources cited throughout this book as part of normal project management activities.